КЕНТ БЕК

ЭКСТРЕМАЛЬНОЕ ПРОГРАММИРОВАНИЕ

«ЧИСТЫЙ КОД, КОТОРЫЙ РАБОТАЕТ, — ВОТ ЦЕЛЬ, К КОТОРОЙ СТОИТ СТРЕМИТЬСЯ»

РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ

TDD

Addison-Wesley

Annotation

Возвращение знаменитого бестселлера. Изящный, гибкий и понятный код, который легко модифицировать, который корректно работает и который не подкидывает своим создателям неприятных сюрпризов. Неужели подобное возможно? Чтобы достичь цели, попробуйте тестировать программу еще до того, как она написана. Именно такая парадоксальная идея положена в основу методики TDD (Test-Driven-Development – разработка, основанная на тестировании). Бессмыслица? Не спешите делать скороспелые выводы. Рассматривая применение TDD на примере разработки реального программного кода, автор демонстрирует простоту и мощь этой методики. В книге приведены два программных проекта, целиком и полностью реализованных с использованием TDD. За рассмотрением примеров следует обширный каталог приемов работы в стиле TDD, а также паттернов и рефакторингов, имеющих отношение к TDD. Книга будет полезна для любого программиста, желающего повысить производительность своей работы и получить удовольствие от программирования.

- Кент Бек
  -
  - Предисловие
  - Благодарности
    -
  - Введение
  - Часть I
    -
    - 1. Мультивалютные деньги
    - 2. Вырождающиеся объекты
    - 3. Равенство для всех
    - 4. Данные должны быть закрытыми
    - 5. Поговорим о франках
    - 6. Равенство для всех, вторая серия
    - 7. Яблоки и апельсины
    - 8. Создание объектов
    - 9. Потребность в валюте
    - 10. Избавление от двух разных версий times()

        - 11. Корень всего зла
        - 12. Сложение, наконец-то
        - 13. Делаем реализацию реальной
        - 14. Обмен валюты
        - 15. Смешение валют
        - 16. Абстракция, наконец-то!
        - 17. Ретроспектива денежного примера
    - Часть II
        -
        - 18. Первые шаги на пути к xUnit
        - 19. Сервируем стол (метод setUp)
        - 20. Убираем со стола (метод tearDown)
        - 21. Учет и контроль
        - 22. Обработка неудачного теста
        - 23. Оформляем тесты в набор
        - 24. Ретроспектива xUnit
    - Часть III. Шаблоны разработки через тестирование
        -
        - 25. Шаблоны разработки через тестирование
        - 26. Шаблоны красной полосы
        - 27. Шаблоны тестирования
        - 28. Шаблоны зеленой полосы
        - 29. Шаблоны xUnit
        - 30. Шаблоны проектирования
        - 31. Рефакторинг
        - 32. Развитие навыков TDD
    - Приложение I
    - Приложение II
    - Послесловие
- notes
    - 1
    - 2
    - 3
    - 4
    - 5
    - 6
    - 7
    - 8
    - 9

- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33

Кент Бек
Экстремальное программирование: разработка через тестирование

Посвящается Синди: крыльям моей души

Права на издание получены по соглашению с Addison-Wesley Longman. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0321146533 англ.
ISBN 978-5-496-02570-6

© 2003 by Pearson Education, Inc.
© Перевод на русский язык ООО Издательство «Питер», 2017
© Издание на русском языке, оформление ООО Издательство «Питер», 2017
© Серия «Библиотека программиста», 2017

Предисловие

Чистый код, который работает (clean code that works), – в этой короткой, но содержательной фразе, придуманной Роном Джеффризом (Ron Jeffries), кроется весь смысл методики разработки через тестирование (Test-Driven Development, TDD). Чистый код, который работает, – это цель, к которой стоит стремиться потому, что

• это предсказуемый способ разработки программ. Вы знаете, когда работу можно считать законченной и не беспокоиться о длинной череде ошибок;

• дает шанс усвоить уроки, которые преподносит код. Если вы воспользуетесь первой же идеей, которая пришла в голову, у вас не будет шанса реализовать вторую, лучшую идею;

• улучшает жизнь пользователей ваших программ;

• позволяет вашим коллегам рассчитывать на вас, а вам – рассчитывать на них;

• писать такой код приятнее.

Но как получить чистый код, который работает? Многие силы мешают нам получить чистый код, а иногда не удается даже получить код, который просто работает. Чтобы избавиться от множества проблем, мы будем разрабатывать код, опираясь на автоматизированное тестирование. Такой стиль программирования называется разработкой через тестирование. Согласно этой методике

• новый код пишется только после того, как будет написан автоматический тест, завершающийся неудачей;

• любое дублирование устраняется.

Два простых правила, не правда ли? Однако они генерируют сложное индивидуальное и групповое поведение со множеством технических последствий:

• в процессе проектирования мы постоянно запускаем код и получаем представление о его работе, это помогает принимать правильные решения;

• мы сами пишем тесты, так как не можем ждать, что кто-то другой напишет тесты для нас;

• наша среда разработки должна быстро реагировать на небольшие модификации кода;

• дизайн программы должен базироваться на использовании множества автономных, слабо связанных компонентов, чтобы упростить тестирование

кода.

Два упомянутых правила TDD определяют порядок этапов программирования.

1. Красный – напишите небольшой тест, который не работает, а возможно, даже не компилируется.

2. Зеленый – заставьте тест работать как можно быстрее, при этом не думайте о правильности дизайна и чистоте кода. Напишите ровно столько кода, чтобы тест сработал.

3. Рефакторинг – устраните из написанного кода любое дублирование.

Красный – зеленый – рефакторинг – это мантра TDD.

Если допустить, что такой стиль программирования возможен, можно предположить, что благодаря его использованию код будет содержать существенно меньше дефектов, кроме того, цель работы будет ясна всем, кто принимает в ней участие. Если так, тогда разработка только кода, необходимого для прохождения тестов, приводит также к социальным последствиям:

- при достаточно низкой плотности дефектов команда контроля качества (Quality Assurance, QA) сможет перейти от реагирования на ошибки к их предупреждению;
- с уменьшением количества неприятных сюрпризов менеджеры проекта смогут точнее оценить трудозатраты и вовлечь заказчиков в процесс разработки;
- если темы технических дискуссий будут четко определены, программисты смогут взаимодействовать друг с другом постоянно, а не раз в день или раз в неделю;
- и снова при достаточно низкой плотности дефектов мы сможем каждый день получать интегрированный рабочий продукт с добавленной в него новой функциональностью, благодаря чему мы сможем вступить с нашими заказчиками в деловые отношения совершенно нового типа.

Итак, идея проста, но в чем наш интерес? Почему программист должен взять на себя дополнительную обязанность писать автоматизированные тесты? Зачем программисту двигаться вперед малюсенькими шажками, когда его мозг в состоянии продумать гораздо более сложную структуру дизайна? Храбрость.

Храбрость

TDD – это способ управления страхом в процессе программирования.

Я не имею в виду страх падения со стула или страх перед начальником. Я имею в виду страх перед задачей, «настолько сложной, что я пока понятия не имею, как ее решить». Боль – это когда природа говорит нам: «Стоп!», а страх – это когда природа говорит нам: «Будь осторожен!» Осторожность – это совсем не плохо, однако помимо пользы страх оказывает на нас некоторое негативное влияние:

• страх заставляет нас заблаговременно и тщательно обдумывать, к чему может привести то или иное действие;

• страх заставляет нас меньше общаться;

• страх заставляет нас пугаться отзывов о нашей работе;

• страх делает нас раздражительными.

Ничего из этого нельзя назвать полезным для процесса программирования, особенно если вы работаете над сложной задачей. Итак, перед нами встает вопрос, как выйти из сложной ситуации и

• не пытаться предсказать будущее, а немедленно приступить к практическому изучению проблемы;

• не отгораживаться от остального мира, а повысить уровень коммуникации;

• не избегать откликов, а, напротив, установить надежную обратную связь и с ее помощью тщательно контролировать результаты своих действий;

• (с раздражением вы должны справиться самостоятельно).

Сравним программирование с подъемом ведра из колодца. Ведро наполнено водой, вы вращаете рычаг, наматывая цепь на ворот и поднимая ведро наверх. Если ведро небольшое, вполне подойдет обычный, свободно вращающийся ворот. Но если ведро большое и тяжелое, вы устанете прежде, чем поднимете его. Чтобы получить возможность отдыхать между поворотами рычага, необходим храповой механизм, позволяющий фиксировать рычаг. Чем тяжелее ведро, тем чаще должны следовать зубья на шестеренке храповика.

Тесты в TDD – это зубья на шестеренке храповика. Заставив тест работать, мы знаем, что теперь тест работает, отныне и навеки. Мы стали на шаг ближе к завершению работы, чем были до того, как тест заработал. После этого мы заставляем работать второй тест, затем третий, четвертый и т. д. Чем сложнее проблема, стоящая перед программистом, тем меньше функциональных возможностей должен охватывать каждый тест.

Читатели книги *Extreme Programming Explaine*[1], должно быть, обратили внимание на разницу в тоне между экстремальным программированием (Extreme Programming, XP) и разработкой через

тестирование (Test-Driven Development, TDD). В отличие от XP методика TDD не является абсолютной. XP говорит: «чтобы двигаться дальше, вы обязаны освоить это и это». TDD – менее конкретная методика. TDD предполагает наличие интервала между принятием решения и получением результатов, и предлагает инструменты управления продолжительностью этого интервала. «Что, если в течение недели я буду проектировать алгоритм на бумаге, а затем напишу код, использовав подход “сначала тесты”? Будет ли это соответствовать TDD?» Конечно, будет. Вы знаете величину интервала между принятием решения и оценкой результатов и осознанно контролируете этот интервал.

Большинство людей, освоивших TDD, утверждают, что их практика программирования изменилась к лучшему. *Инфицированные тестами* (test infected) – такое определение придумал Эрих Гамма (Erich Gamma), чтобы описать данное изменение. Освоив TDD, вы обнаруживаете, что пишете значительно больше тестов, чем раньше, и двигаетесь вперед малюсенькими шагами, которые раньше показались бы вам бессмысленными. С другой стороны, некоторые программисты, познакомившись с TDD, решают вернуться к использованию прежних практик, зарезервировав TDD для особых случаев, когда обычное программирование не приводит к желаемому прогрессу.

Определенно, существуют задачи, которые невозможно (по крайней мере, на текущий момент) решить только при помощи тестов. В частности, TDD не позволяет механически продемонстрировать адекватность разработанного кода с точки зрения безопасности данных и надежности выполнения параллельных операций. Безусловно, безопасность основана на коде, в котором не должно быть дефектов, однако она основана также на участии человека в процедурах защиты данных. Тонкие проблемы параллельного выполнения операций невозможно с уверенностью воспроизвести, просто запустив некоторый код.

Прочитав эту книгу, вы сможете:

• начать применять TDD;

• писать автоматические тесты;

• выполнять рефакторинг, воплощая решения по одному за раз.

Книга разделена на три части.

Часть I. На примере денег. Пример разработки типичного прикладного кода с использованием TDD. Этот пример позаимствован мною у Уорда Каннингэма (Ward Cunningham) много лет назад, и с тех пор я неоднократно использовал его для демонстрации TDD. В нем

рассматривается мультивалютная арифметика: выполнение математических операций над денежными величинами, выраженными в различных валютах. Этот пример научит вас писать тесты до тестируемого ими кода и органически развивать проект.

Часть II. На примере xUnit. Пример тестирования более сложной логики, использующей механизм рефлексии и исключения. В примере рассматривается разработка инфраструктуры автоматического тестирования. Этот пример познакомит вас также с архитектурой xUnit, которая лежит в основе множества инструментов тестирования. Во втором примере вы научитесь двигаться вперед еще меньшими шажками, а также разрабатывать систему с использованием механизмов самой этой системы.

Часть III. Шаблоны разработки через тестирование. Здесь рассматриваются шаблоны, которые помогут найти ответы на множество вопросов, в частности: какие тесты писать и как их писать с использованием xUnit. Кроме того, здесь вы найдете описание некоторых избранных шаблонов проектирования и рефакторинга, использовавшихся при создании примеров для данной книги.

Я написал примеры так, будто мы с вами принимаем участие в сеансе парного программирования. Если перед прогулкой вы предпочитаете прежде посмотреть на карту, можете сначала ознакомиться с шаблонами в третьей части книги, а затем рассматривать примеры как их иллюстрацию. Если вы предпочитаете сначала погулять, а потом посмотреть на карте, где побывали, тогда сначала прочитайте первые две части с примерами и обращайтесь к третьей части за справками по мере необходимости. Некоторые из рецензентов данной книги, отмечали, что примеры усваиваются лучше, если во время чтения запустить среду разработки, набирать код и запускать тесты.

Касательно примеров хочу отметить следующее. Оба примера, мультивалютные вычисления и инфраструктура тестирования, могут показаться чрезвычайно простыми. Существуют более сложные, дефектные и уродливые решения этих же самых задач (мне лично неоднократно приходилось сталкиваться с подобными решениями). Чтобы сделать книгу более похожей на реальность, я мог бы продемонстрировать одно из таких решений. Однако моя и, я надеюсь, ваша цель – написать чистый код, который работает. Прежде чем пенять на излишнюю простоту примеров, на несколько секунд представьте себе мир программирования, в котором весь код выглядит также чисто и понятно, в котором нет слишком сложных решений, только проблемы, которые кажутся слишком сложными лишь с

первого взгляда. Сложные проблемы нуждаются в тщательном обдумывании. TDD поможет добиться этого.

Благодарности

Спасибо всем, кто с необычайным усердием и самоотверженностью просматривал рукопись данной книги. Я беру на себя всю ответственность за представленный в книге материал, однако без посторонней помощи данная книга была бы куда менее читабельной и менее полезной. Перечислю всех, кто помогал мне, в произвольном порядке: Стив Фриман (Steve Freeman), Франк Вестфал (Frank Westphall), Рон Джеффриз (Ron Jeffries), Дирк Кёниг (Dirk Koning), Эдвард Хейят (Edward Heiatt), Таммо Фриис (Tammo Freese), Джим Ньюкирк (Jim Newkirk), Йоханнес Линк (Johannes Link), Манфред Ланж (Manfred Lange), Стив Хайес (Steve Hayes), Алан Френсис (Alan Francis), Джонатан Расмуссон (Jonathan Rasmusson), Шейн Клаусон (Shane Clauson), Саймон Крэйз (Simon Crase), Кай Пентекост (Kay Pantecost), Мюррей Бишоп (Murrey Bishop), Райан Кинг (Ryan King), Билл Уэйк (Bill Wake), Эдмунд Швепп (Edmund Schweppe), Кевин Лауренс (Kevin Lawrence), Джон Картер (John Carter), Флип (Phlip), Петер Хансен (Peter Hansen), Бен Шрёдер (Ben Schroeder), Алекс Чаффи (Alex Chaffee), Петер ван Руйен (Peter van Rooijen), Рик Кавала (Rick Kawala), Марк ван Хамерсвельд (Mark van Hamersveld), Дуг Шварц (Doug Swartz), Лорен Боссави (Laurent Bossavit), Илья Преуз (Ilia Preuz), Дэниэл Ле Берре (Daniel Le Berre), Франк Карвер (Frank Carver), Майк Кларк (Mike Clark), Кристиан Пекелер (Christian Pekeler), Карл Скотланд (Karl Scotland), Карл Манастер (Carl Manaster), Дж. Б. Рэйнсбергер (J.B. Rainsberger), Петер Линдберг (Peter Lindberg), Дарач Эннис (Darach Ennis), Кайл Кордес (Kyle Cordes), Джастин Сампсон (Justin Sampson), Патрик Логан (Patrik Logan), Даррен Хоббс (Darren Hobbs), Аарон Сансоне (Aaron Sansone), Сайвер Энстад (Syver Enstad), Шинобу Каваи (Shinobu Kawai), Эрик Мид (Erik Meade), Патрик Логан (Patrik Logan), Дан Росторн (Dan Rawsthorne), Билл Рутисер (Bill Rutiser), Эрик Хэрман (Eric Herman), Пол Чишолм (Paul Chisholm), Аэзим Джалис (Asim Jalis), Айвэн Мур (Ivan Moor), Леви Первис (Levi Purvis), Рик Магридж (Rick Mugridge), Энтони Адаши (Antony Adachi), Найджел Торн (Nigel Thorne), Джон Блей (John Bley), Кари Хойджарви (Kari Hoijarvi), Мануэль Амаго (Manuel Amago), Каору Хосокава (Kaouru Hosokawa), Пэт Эйлер (Pat Eyler), Росс Шоу (Ross Shaw), Сэм Джэнтл (Sam Gentle), Джин Райотт (Jean Rajotte), Филип Антрас (Phillipe Antras) и Джейме Нино (Jaime Nino).

Я хотел бы выразить свою признательность всем программистам, с которыми разрабатывал код в стиле «сначала тесты». Спасибо вам за терпение и внимание к идее, которая звучала полным сумасшествием, в особенности в самом начале развития TDD. Благодаря вам я научился значительно большему, чем если бы действовал самостоятельно. Мое обучение было наиболее успешным, когда я сотрудничал с Массимо Арнольди (Massimo Arnoldi), Ральфом Битти (Ralph Beatti), Роном Джеффрисом (Ron Jeffries), Мартином Фаулером (Martin Fowler) и (безусловно, не в последнюю очередь) Эрихом Гаммой (Erich Gamma), однако я хотел бы отметить, что помимо этих людей были и другие, благодаря которым я тоже научился очень многому.

Я хотел бы поблагодарить Мартина Фаулера (Martin Fowler) за помощь с FrameMaker. Этот человек должен быть самым высокооплачиваемым на планете специалистом в области подготовки текста к печати (к счастью, он не против, чтобы гонорар за эту книгу целиком достался мне).

Моя карьера настоящего программиста нчалась благодаря наставничеству Уорда Каннингэма и постоянному сотрудничеству с ним. Иногда я рассматриваю разработку через тестирование как попытку дать каждому программисту, работающему в произвольной среде, ощущение комфорта и душевности, которое было у нас с Уордом, когда мы вместе разрабатывали программы на Smalltalk. Не существует способа определить первоначальный источник идей, если два человека обладают одним общим мозгом. Если вы предположите, что все хорошие идеи на самом деле придумал Уорд, вы будете не далеки от истины.

В последнее время сформировалось клише: выражать глубочайшую признательность за те жертвы и лишения, которые вынуждена терпеть семья, один из членов которой заболел идеей написать книгу. Дело в том, что семейные жертвы и лишения так же необходимы для написания книги, как и бумага. Я выражаю свою самую глубочайшую благодарность моим детям, которые не могли приступить к завтраку, пока я не закончу очередную главу, а также моей жене, которая в течение двух месяцев была вынуждена повторять каждую свою фразу три раза.

Спасибо Майку Хэндерсону (Mike Henderson) за воодушевление, а также Марси Барнс (Marcy Barns) за то, что она пришла на помощь в трудную минуту.

Наконец, спасибо неизвестному автору книги, которую я прочитал в 12-летнем возрасте. В той книге было предложено сравнивать две ленты: с реальными результатами и ожидаемыми, и дорабатывать программу, пока реальные результаты не совпадут с ожидаемыми. Спасибо, спасибо,

спасибо.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты *comp@piter.com* (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства *http://www.piter.com* вы найдете подробную информацию о наших книгах.

Введение

Однажды рано утром в пятницу к Уорду Каннингэму зашел босс и представил его Питеру, перспективному заказчику системы WyCash. Эта система предназначалась для управления портфелем облигаций, ее разработкой и продажей занималась компания Уорда. «Возможности вашей системы впечатляют, – сказал Питер. – Но вот в чем проблема: я собираюсь открыть новый фонд облигаций. Как я понял, ваша система поддерживает облигации, номинированные только в долларах США. Мне же понадобится система, поддерживающая разные валюты». Босс повернулся к Уорду и спросил: «Мы сможем это сделать?»

Вот он, кошмарный сценарий для любого разработчика. Все шло хорошо, пока события развивались по намеченному плану, и вдруг все меняется. Надо сказать, это было кошмаром не только для Уорда – босс, съевший собаку на управлении программными проектами, тоже не знал, что ответить.

Система WyCash была разработана небольшой командой программистов за пару лет. Она позволяла работать с большинством ценных бумаг с фиксированным доходом, имеющих хождение на американском рынке. Более того, она поддерживала некоторые редкие инструменты рынка ценных бумаг, например гарантированные инвестиционные контракты (Guaranteed Investment Contracts), и этим выгодно отличалась от конкурентов.

В основу разработки WyCash легли объектно-ориентированные технологии, также была использована объектная база данных. Базовой абстракцией системы был класс Dollar, класс, который отвечал за вычисления и форматирование результатов. В самом начале работы над системой его разработку поручили отдельной группе хороших программистов.

В течение последних шести месяцев Уорд и остальные члены команды постепенно уменьшали количество обязанностей класса Dollar. Оказалось, что числовые классы языка Smalltalk вполне подошли для реализации вычислений, а для округления до трех десятичных знаков был написан специальный код. Результаты вычислений становились все точнее и точнее, и в конце концов сложные алгоритмы тестирования, выполнявшие сравнение величин с учетом погрешности, были заменены простым сравнением реального и ожидаемого результатов.

За форматирование результатов в действительности отвечали классы пользовательского интерфейса, а не класс Dollar. Так как соответствующие тесты были написаны на уровне этих классов, в частности для подсистемы отчетов[2], поэтому предполагаемые изменения не должны были их коснуться. В результате, спустя шесть месяцев, у объекта Dollar осталось не так уж много обязанностей…

Один из наиболее сложных алгоритмов, вычисление средневзвешенных величин, также постепенно менялся. Вначале существовало много различных реализаций этого алгоритма, разбросанных по всему коду. Однако позже, с появлением подсистемы отчетов, стало очевидно, что существует только одно место, где этот алгоритм должен быть реализован, – класс AveragedColumn. Именно этим классом и занялся Уорд.

Если бы удалось внедрить в этот алгоритм поддержку работы с несколькими валютами, система в целом смогла бы стать «мультивалютной». Центральная часть алгоритма отвечала бы за хранение количества денег «в столбце». При этом алгоритм должен быть достаточно абстрактным для вычисления средневзвешенных величин любых объектов, которые поддерживали арифметические операции. К примеру, с его помощью можно было бы вычислять средневзвешенное календарных дат.

Выходные прошли как обычно – за отдыхом, а в понедельник утром босс поинтересовался: «Ну как, мы сможем это сделать?» – «Дайте мне еще день, и я скажу точно», – ответил Уорд.

В вычислении средневзвешенной величины объект Dollar как бы являлся переменной. В случае наличия нескольких валют потребовалось бы по одной переменной на каждый тип валюты, нечто вроде многочлена. Только вместо 3x2 и 4y3 – 15 USD и 20 °CHF[3].

Быстрый эксперимент показал, что при вычислениях можно работать не с объектом Dollar (доллар), а с более общим объектом – Currency (валюта). При этом, если выполнялась операция над двумя различными валютами, значение следовало возвращать в виде объекта PolyCurrency (мультивалютный). Сложность заключалась в том, чтобы добавить новую функциональность, не сломав при этом то, что уже работает. А что, если просто прогнать тесты?

После добавления к классу Currency нескольких (пока нереализованных) операций большинство тестов все еще успешно выполнялось; к концу дня проходили все тесты. Уорд интегрировал новый код в текущую версию и пошел к боссу. «Мы сможем это сделать», –

уверенно сказал он.

Давайте задумаемся над этой историей. Через пару дней потенциальный рынок для системы WyCash увеличился в несколько раз, соответственно подскочила ее ценность. Важно, что возможность создать значительную бизнес-ценность за такое короткое время не была случайной. Свою роль сыграли следующие факторы:

• Метод – Уорду и команде разработки WyCash потребовался опыт в пошаговом наращивании проектных возможностей системы, с хорошо отработанным механизмом внесения изменений.

• Мотив – Уорду и его команде было необходимо четкое представление о значимости поддержки мультивалютности в WyCash, а также потребовалась смелость взяться за такую на первый взгляд безнадежную задачу.

• Возможность – сочетание всеохватывающей, продуманной системы тестов и хорошо структурированной программы; язык программирования, обеспечивающий локализацию проектных решений и тем самым упрощающий идентификацию ошибок.

Мотив – это то, чем вы не можете управлять; сложно сказать, когда он у вас появится и заставит заняться техническим творчеством для решения бизнес-задач. Метод и возможность, с другой стороны, находятся под вашим полным контролем. Уорд и его команда создали метод и возможность благодаря таланту, опыту и дисциплине. Значит ли это, что, если вы не входите в десятку лучших разработчиков планеты и у вас нет приличного счета в банке (настолько приличного, чтобы попросить босса погулять, пока вы занимаетесь делом), такие подвиги не для вас?

Нет, вовсе нет. Всегда можно развернуть проект так, чтобы работа над ним стала творческой и интересной, даже если вы обычный разработчик и прогибаетесь под обстоятельства, когда приходится туго. Разработка через тестирование (Test-Driven Development, TDD) – это набор способов, ведущих к простым программным решениям, которые может применять любой разработчик, а также тестов, придающих уверенность в работе. Если вы гений, эти способы вам не нужны. Если вы тугодум – они вам не помогут. Для всех остальных, кто находится между этими крайностями, следование двум простым правилам поможет работать намного эффективнее:

• перед тем как писать любой фрагмент кода, создайте автоматизированный тест, который поначалу будет терпеть неудачу;

• устраните дублирование.

Как конкретно следовать этим правилам, какие существуют в данной

области нюансы и какова область применимости этих способов – все это составляет тему книги, которую вы сейчас читаете. Вначале мы рассмотрим объект, созданный Уордом в момент вдохновения, – мультивалютные деньги (multi-currency money).

Часть I
На примере денег

Мы займемся реализацией примера, разрабатывая код полностью на основе тестирования (кроме случаев, когда в учебных целях будут допускаться преднамеренные ошибки). Моя цель – дать вам почувствовать ритм разработки через тестирование (TDD). Кратко можно сказать, что TDD заключается в следующем:

• Быстро создать новый тест.

• Запустить все тесты и убедиться, что новый тест терпит неудачу.

• Внести небольшие изменения.

• Снова запустить все тесты и убедиться, что на этот раз все тесты выполнились успешно.

• Провести рефакторинг для устранения дублирования.

Кроме того, придется найти ответы на следующие вопросы:

• Как добиться того, чтобы каждый тест охватывал небольшое приращение функциональности?

• Как и за счет каких небольших и, наверное, неуклюжих изменений обеспечить успешное прохождение новых тестов?

• Как часто следует запускать тесты?

• Из какого количества микроскопических шагов должен состоять рефакторинг?

1. Мультивалютные деньги

Вначале мы рассмотрим объект, созданный Уордом для системы WyCash, – мультивалютные деньги (см. «Введение»). Допустим, у нас есть отчет вроде этого.

Компания	Количество акций	Цена	Всего
IBM	1000	25	25 000
GE	400	100	40 000
		Итого:	65 000

Добавив различные валюты, получим мультивалютный отчет.

Компания	Количество акций	Цена	Всего
IBM	1000	25 USD	25 000 USD
Novartis	400	150 CHF	60 000 CHF
		Итого	65 000 USD

Также необходимо указать курсы обмена.

Из	В	Курс
CHF	USD	1,5

$5 + 1 °CHF = $10, если курс обмена 2:1
$5 * 2 = $10

Что нам понадобится, чтобы сгенерировать такой отчет? Или, другими словами, какой набор успешно выполняющихся тестов сможет гарантировать, что созданный код правильно генерирует отчет? Нам понадобится:

• выполнять сложение величин в двух различных валютах и конвертировать результат с учетом указанного курса обмена;

• выполнять умножение величин в валюте (стоимость одной акции) на количество акций, результатом этой операции должна быть величина в

валюте.

Составим список задач, который будет напоминать нам о планах, не даст запутаться и покажет, когда все будет готово. В начале работы над задачей выделим ее жирным шрифтом, **вот так**. Закончив работу над ней – вычеркнем, вот так. Когда придет мысль написать новый тест, добавим новую задачу в наш список.

Как видно из нашего списка задач, сначала мы займемся умножением. Итак, какой объект понадобится нам в первую очередь? Вопрос с подвохом. Мы начнем не с объектов, а с тестов. (Мне приходится постоянно напоминать себе об этом, поэтому я просто притворюсь, что вы так же забывчивы, как и я.)

Попробуем снова. Итак, какой тест нужен нам в первую очередь? Если исходить из списка задач, первый тест представляется довольно сложным. Попробуем начать с малого – умножение, – сложно ли его реализовать? Займемся им для начала.

Когда мы пишем тест, мы воображаем, что у нашей операции идеальный интерфейс. Попробуем представить, как будет выглядеть операция снаружи. Конечно, наши представления не всегда будут находить воплощение, но в любом случае стоит начать с наилучшего возможного программного интерфейса (API) и при необходимости вернуться назад, чем сразу делать вещи сложными, уродливыми и «реалистичными».

Простой пример умножения[4]:

```
public void testMultiplication() {
Dollar five = new Dollar(5);
five.times(2);
assertEquals(10, five.amount);
}
```

(Знаю, знаю: публичные поля, побочные эффекты, целые числа для денежных величин и все такое. Маленькие шаги – помните? Мы отметим, что где-то есть душок[5], и продолжим дальше. У нас есть тест, который не выполняется, и мы хотим как можно скорее увидеть зеленую полоску[6].)

$5 + 1 °CHF = $10, если курс обмена 2:1
$5 * 2 = $10
Сделать переменную amount закрытым членом класса
Побочные эффекты в классе Dollar?

Округление денежных величин?

Тест, который мы только что создали, даже не компилируется, но это легко исправить. (О том, когда и как создаются тесты, я расскажу позже – когда мы будем подробнее говорить о среде тестирования, JUnit.) Как проще всего заставить тест компилироваться (пусть он пока и будет терпеть неудачу)? У нас четыре ошибки компиляции:
• нет класса Dollar;
• нет конструктора;
• нет метода times(int);
• нет поля (переменной) amount.
Устраним их одну за другой. (Я всегда ищу некоторую численную меру прогресса.) От одной ошибки мы избавимся, определив класс Dollar:

Dollar
```
class Dollar
```

Одной ошибкой меньше, осталось еще три. Теперь нам понадобится конструктор, причем совершенно необязательно, чтобы он что-то делал – лишь бы компилировался.

Dollar
```
Dollar(int amount) {
}
```

Осталось две ошибки. Необходимо создать заготовку метода times(). Снова мы выполним минимум работы, только чтобы заставить тест компилироваться:

Dollar
```
void times(int multiplier) {
}
```

Теперь осталась только одна ошибка. Чтобы от нее избавиться, нужно создать поле (переменную) amount:

Dollar
```
int amount;
```

Отлично! Теперь можно запустить тест и убедиться, что он не выполняется: ситуация продемонстрирована на рис. 1.1.

Загорается зловещий красный индикатор. Фреймворк тестирования (JUnit в нашем случае) выполнил небольшой фрагмент кода, с которого мы начали, и выяснил, что вместо ожидаемого результата «10» получился «0». Ужасно…

Рис. 1.1. Прогресс! Тест терпит неудачу

Вовсе нет! Неудача – это тоже прогресс. Теперь у нас есть конкретная мера неудачи. Это лучше, чем просто догадываться, что у нас что-то не так. Наша задача «реализовать мультивалютность» превратилась в «заставить работать этот тест, а потом заставить работать все остальные тесты». Так намного проще и намного меньше поводов для страха. Мы заставим этот тест работать.

Возможно, вам это не понравится, но сейчас наша цель не получить идеальное решение, а заставить тест выполняться. Мы принесем свою жертву на алтарь истины и совершенства чуть позже.

Наименьшее изменение, которое заставит тест успешно выполняться, представляется мне таким:

Dollar
int amount = 10;

Рисунок 1.2 показывает результат повторного запуска теста. Теперь мы видим ту самую зеленую полоску, воспетую в поэмах и прославленную в веках.

Вот оно, счастье! Но радоваться рано, ведь цикл еще не завершен. Уж слишком мал набор входных данных, которые заставят такую странно попахивающую и наивную реализацию работать правильно. Перед тем как двигаться дальше, немного поразмышляем.

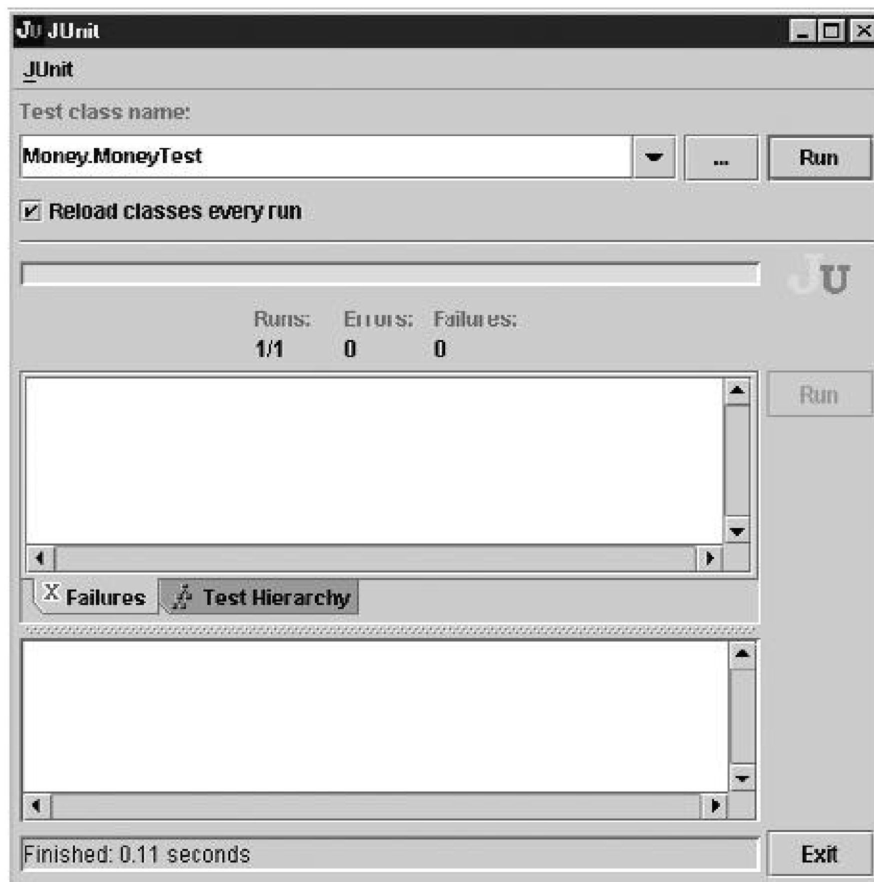

Рис. 1.2. Тест успешно выполняется

Вспомним, полный цикл TDD состоит из следующих этапов:

1. Добавить небольшой тест.
2. Запустить все тесты и убедиться, что новый тест терпит неудачу.

3. Внести небольшое изменение.

4. Снова запустить тесты и убедиться, что все они успешно выполняются.

5. Устранить дублирование с помощью рефакторинга.

ЗАВИСИМОСТЬ И ДУБЛИРОВАНИЕ

Стив Фримен (Steve Freeman) указал, что проблема с тестами и кодом заключается не в дублировании (на которое я еще не указал вам, но сделаю это, как только закончится отступление). Проблема заключается в зависимости между кодом и тестами – вы не можете изменить одно, не изменив другого. Наша цель – иметь возможность писать новые осмысленные тесты, не меняя при этом код, что невозможно при нашей текущей реализации.

Зависимость является ключевой проблемой разработки программного обеспечения. Если фрагменты SQL, зависящие от производителя используемой базы данных, разбросаны по всему коду и вы хотите поменять производителя, то непременно окажется, что код зависит от этого производителя. Вы не сможете поменять производителя базы данных и при этом не изменить код.

Зависимость является проблемой, а дублирование – ее симптомом. Чаще всего дублирование проявляется в виде дублирования логики – одно и то же выражение появляется в различных частях кода. Объекты – отличный способ абстрагирования, позволяющий избежать данного вида дублирования.

В отличие от большинства проблем в реальной жизни, где устранение симптомов приводит только к тому, что проблема проявляется в худшей форме где-то еще, устранение дублирования в программах устраняет и зависимость. Именно поэтому существует второе правило TDD. Устраняя дублирование перед тем, как заняться следующим тестом, мы максимизируем наши шансы сделать его успешным, внеся всего одно изменение.

Мы выполнили первые четыре пункта цикла, и все готово к устранению дублирования. Но где же оно? Обычно мы замечаем дублирование в нескольких разных фрагментах кода, однако в нашем случае – друг друга дублируют тест и тестируемый код. Еще не видите? Как насчет того, чтобы написать так:

Dollar

```
int amount = 5 * 2;
```

Теперь ясно, откуда мы взяли число 10. Видимо, мы в уме произвели умножение, причем так быстро, что даже не заметили. Произведение «5 умножить на 2» присутствует как в тесте, так и в тестируемом коде. Только изначально в коде оно было представлено в виде константы 10. Сейчас же 5 и 2 отделены друг от друга, и мы должны безжалостно устранить дублирование, перед тем как двинуться дальше. Такие вот правила.

Действия, с помощью которого мы устранили бы 5 и 2 за один шаг, не существует. Но что, если переместить установку поля (переменной) amount в метод times()?

Dollar

```
int amount;

void times(int multiplier) {
amount= 5 * 2;
}
```

Тест все еще успешно выполняется, и индикатор остался зеленым. Успех нам пока сопутствует.

Такие шаги кажутся вам слишком мелкими? Помните, TDD не обязывает двигаться только микроскопическими шагами, речь идет о способности совершать эти микроскопические шаги. Буду ли я программировать день за днем такими маленькими шагами? Нет. Но когда дела совсем плохи, я рад возможности выполнять хоть такие шаги. Примените микроскопические шаги к любому собственному примеру. Если вы сможете продвигаться маленькими шагами, вы сумеете делать шаги более крупного и подходящего размера. Если же вы способны делать только огромные шаги, вы никогда не распознаете ситуацию, в которой более уместны меньшие шаги.

Оставим рассуждения. На чем мы остановились? Ну да, мы избавлялись от дублирования между кодом теста и рабочим кодом. Где мы можем взять 5? Это значение передавалось конструктору, поэтому его можно сохранить в переменной amount:

Dollar

```
Dollar(int amount) {
this.amount = amount;
}
```

и использовать в методе times():

Dollar

```
void times(int multiplier) {
amount = amount * 2;
}
```

Число 2 передается в параметре multiplier, поэтому подставим параметр вместо константы:

Dollar

```
void times(int multiplier) {
amount= amount * multiplier;
}
```

Чтобы продемонстрировать, как хорошо мы знаем синтаксис языка Java, используем оператор *= (который, кстати, уменьшает дублирование):

Dollar

```
void times(int multiplier) {
amount *= multiplier;
}
```

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
Сделать переменную amount закрытым членом класса
Побочные эффекты в классе Dollar?
Округление денежных величин?

Теперь можно пометить первый тест как завершенный. Далее мы позаботимся о тех странных побочных эффектах; но сначала давайте подведем итоги. Мы сделали следующее:

• создали список тестов, которые – мы знаем – нам понадобятся;
• с помощью фрагмента кода описали, какой мы хотим видеть нашу операцию;

- временно проигнорировали особенности среды тестирования JUnit;
- заставили тесты компилироваться, написав соответствующие заготовки;
- заставили тесты работать, использовав сомнительные приемы;
- слегка улучшили работающий код, заменив константы переменными;
- добавили пункты в список задач, вместо того чтобы заняться всеми этими задачами сразу.

2. Вырождающиеся объекты

Обычный цикл разработки на основе тестирования состоит из следующих этапов:

1. Напишите тест. Представьте, как будет реализована в коде воображаемая вами операция. Продумав ее интерфейс, опишите все элементы, которые, как вам кажется, понадобятся.

2. Заставьте тест работать. Первоочередная задача – получить зеленую полоску. Если напрашивается простое и элегантное решение, используйте его. Если же на реализацию такого решения потребуется время, отложите его. Просто отметьте, что к нему нужно вернуться, когда будет решена основная задача – быстро получить зеленый индикатор. Такой подход довольно неприятен для опытных разработчиков (в эстетическом плане), ведь они следуют только правилам хорошей разработки. Но зеленая полоска прощает все грехи, правда, всего лишь на мгновение.

3. Улучшите решение. Теперь, когда система работает, избавьтесь от прошлых огрехов и вернитесь на путь истинной разработки. Устраните дублирование, которое вы внесли, и быстро сделайте так, чтобы полоска снова стала зеленой.

Наша цель – *чистый код, который работает* (отдельное спасибо Рону Джеффризу за этот слоган). Иногда такой код не по силам даже самым лучшим программистам, и почти всегда он не достижим для большинства программистов (вроде меня). Разделяй и властвуй, приятель, – в этом весь смысл! Сначала мы напишем код, «который работает», после чего создадим «чистый код». Такой подход противоречит модели разработки на основе архитектуры, в которой вы сначала пишете «чистый код», а потом мучаетесь, пытаясь интегрировать в проект код, «который работает».

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
Сделать переменную amount закрытым членом класса
Побочные эффекты в классе Dollar?
Округление денежных величин?

Мы получили один рабочий тест, но в процессе заметили нечто странное: при выполнении операции с объектом Dollar изменяется сам объект. Хотелось бы написать так:

```
public void testMultiplication() {
Dollar five = new Dollar(5);
five.times(2);
assertEquals(10, five.amount);
five.times(3);
assertEquals(15, five.amount);
}
```

Я не могу представить простого способа, который заставит этот тест выполняться. После первого вызова метода times() пять уже больше не пять – на самом деле это уже десять. Если же метод times() будет возвращать новый объект, тогда мы сможем умножать наши исходные пять баксов хоть целый день, и они не изменятся. Для реализации этой идеи нам потребуется изменить интерфейс объекта Dollar и, соответственно, изменить тест. Это нормально, ведь вполне возможно, что наши догадки о правильном интерфейсе не более правдоподобны, чем догадки о правильной реализации.

```
public void testMultiplication() {
Dollar five = new Dollar(5);
Dollar product = five.times(2);
assertEquals(10, product.amount);
product = five.times(3);
assertEquals(15, product.amount);
}
```

Новый тест не будет компилироваться, пока мы не изменим объявление метода Dollar.times():

Dollar

```
Dollar times(int multiplier) {
amount *= multiplier;
return null;
}
```

Теперь тест компилируется, но не работает. И это тоже прогресс! Чтобы заставить его работать, придется возвращать новый объект Dollar с правильным значением:

Dollar

```
Dollar times(int multiplier) {
return new Dollar(amount * multiplier);
}
```

$5 + 1 °CHF = $10, если курс обмена 2:1

~~$5 * 2 = $10~~

Сделать переменную amount закрытым членом класса

~~Побочные эффекты в классе Dollar?~~

Округление денежных величин?

В главе 1, когда мы заставляли тест работать, мы начинали с заготовки и постепенно улучшали код, пока он не стал полноценным. Теперь мы написали сразу правильную реализацию и молились, пока выполнялись тесты (довольно короткие молитвы, честно говоря – выполнение тестов занимает миллисекунды). Нам повезло, тесты выполнились успешно, и мы вычеркнули еще один пункт.

Мне известны три способа быстрого получения зеленого индикатора. Вот первые два:

• подделать реализацию, иначе говоря, создать заглушку, возвращающую константу, и постепенно заменять константы переменными до тех пор, пока не получится настоящий код;

• использовать очевидную реализацию – просто написать сразу настоящую реализацию.

Используя TDD на практике, я периодически переключаюсь между двумя этими способами. Когда все идет гладко и я знаю, что делать, – я просто создаю одну за другой очевидные реализации (каждый раз запуская тесты, чтобы убедиться, что решение, очевидное для меня, также очевидно для компьютера). Как только я натыкаюсь на красный индикатор, я возвращаюсь к методике «поддельная реализация», после чего провожу рефакторинг. Когда уверенность возвращается, я снова использую методику «очевидная реализация».

Есть еще одна, третья методика, «Триангуляция» (Triangulation), которую мы рассмотрим в главе 3. Подведем итоги. Мы выполнили следующее:

• сформулировали дефект проектирования (побочный эффект) в виде теста, который потерпел неудачу (из-за дефекта);

• создали заглушку, обеспечившую быструю компиляцию кода;

• заставили тест успешно выполняться, написав вроде бы правильный

код.

Преобразование чувства (например, отвращения, вызываемого побочными эффектами) в тест (например, двукратное перемножение одного и того же объекта Dollar) – обычная практика в TDD. Чем дольше я этим занимаюсь, тем легче эстетические суждения переводятся в тесты. В результате мои рассуждения о проектировании становятся более интересными. Сначала мы обсуждаем, должна ли система работать *так* или *по-другому*. После определения правильного поведения системы можно поговорить о наилучшем способе его реализации. Можно сколь угодно долго рассуждать об истине и совершенстве за пивом, но раз мы занимаемся программированием, у нас есть возможность оставить пустые разговоры и перейти к конкретике.

3. Равенство для всех

Если у меня есть целое число и я прибавляю к нему 1, то не предполагаю, что изменится исходное число, – в результате я ожидаю получить новое число. Объекты же обычно ведут себя иначе. К примеру, если у меня есть контракт и я добавлю 1 к его сумме, это будет означать, что сумма контракта должна измениться (да, несомненно, это пример для обсуждения многих интересных законов бизнеса, которые мы здесь рассматривать *не* будем).

Мы можем использовать объекты в качестве значений, так же как используем наш объект Dollar. Соответствующий шаблон называется «Объект-значение» (Value Object). Одно из ограничений этого шаблона заключается в том, что значения атрибутов объекта устанавливаются в конструкторе и никогда в дальнейшем не изменяются.

Значительное преимущество использования шаблона «Объект-значение» состоит в том, что не нужно беспокоиться о проблеме наложения имен (aliasing). Скажем, у меня есть объект Check, представляющий собой чек, и я устанавливаю его сумму – $5, а затем присваиваю эти же $5 сумме другого объекта Check. Одна из самых неприятных проблем на моей памяти заключалась в том, что изменение суммы в первом объекте может приводить к непреднамеренному изменению суммы во втором. Это и есть проблема наложения имен.

Используя объекты-значения, не нужно беспокоиться о наложении имен. Если у меня есть пять долларов ($5), они всегда гарантированно будут оставаться именно пятью долларами ($5). Если вдруг кому-то понадобятся $7, придется создать новый объект.

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
Сделать ~~переменную~~ amount закрытым членом класса
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
equals()

Одно из следствий использования шаблона «Объект-значение» заключается в том, что все операции должны возвращать результаты в виде новых объектов, как было покзано в главе 2. Другое следствие заключается

в том, что объекты-значения должны реализовывать метод equals(), операцию проверки равенства, потому что одни $5 ничем не отличаются от других.

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
Сделать ~~переменную~~ «amount» закрытым членом
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
equals()
hashCode()

Кроме того, если использовать Dollar в качестве ключа хеш-таблицы, вместе с equals() придется реализовать и hashCode(). Добавим этот пункт в список задач и вернемся к нему, когда это будет необходимо.

Вы ведь не собираетесь немедленно приступать к реализации метода equals()? Отлично, я тоже об этом не думаю. Ударив себя линейкой по руке, я стал размышлять над тем, как протестировать равенство. Для начала $5 должны быть равны $5:

```
public void testEquality() {
assertTrue(new Dollar(5). equals(new Dollar(5)));
}
```

Полоска окрасилась красным. Поддельная реализация могла бы просто вернуть значение true:

Dollar
```
public boolean equals(Object object) {
return true;
}
```

Конечно, мы с вами знаем, что на самом деле true – это «5 == 5», что, в свою очередь, означает «amount == 5», что соответствует «amount == dollar.amount». Но если бы я сразу проследил все эти шаги, я не смог бы продемонстрировать третью и наиболее консервативную методику реализации – триангуляцию.

Если две станции слежения, находящиеся на известном расстоянии друг от друга, смогут измерить азимут некоторого источника радиосигнала

(взять пеленг), этого вполне достаточно, чтобы вычислить местоположение источника радиосигнала (как вы помните из курса тригонометрии, в отличие от меня). Это вычисление и называется триангуляцией.

По аналогии, используя метод триангуляции, мы обобщаем код только в том случае, когда у нас два примера или больше. При этом мы ненадолго игнорируем дублирование между тестом и самим кодом (приложения). Когда второй пример потребует более общего решения, тогда и только тогда мы выполним обобщение.

Итак, для триангуляции нам понадобится второй пример. Как насчет того, чтобы проверить $5!= $6?

```
public void testEquality() {
assertTrue(new Dollar(5). equals(new Dollar(5)));
assertFalse(new Dollar(5). equals(new Dollar(6)));
}
```

Теперь необходимо обобщить равенство (equality):

Dollar

```
public boolean equals(Object object) {
Dollar dollar = (Dollar)object;
return amount == dollar.amount;
}
```

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
Сделать ~~переменную~~ amount закрытым членом класса
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()

Мы могли бы использовать триангуляцию и для управления обобщением метода times(). Если бы у нас были примеры $5 * 2 = $10 и $5 * 3 = $15, нам не смогли бы просто возвращать константу.

Думаю, триангуляция – довольно интересная вещь. Я использую ее в случае, если не знаю, как выполнять рефакторинг. Если же я представляю, как устранить дублирование между кодом и тестами и создать более общее решение, я просто создаю его. С какой стати я должен создавать еще один тест, если сразу могу выполнить обобщение?

Однако когда когда в голову не приходит ничего умного, триангуляция дает шанс посмотреть на проблему с другой стороны. Сколько степеней свободы вы хотите поддерживать в вашем приложении (какую степень универсальности, другими словами)? Просто попробуйте ввести некоторые из них, и, возможно, ответ станет очевиднее.

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
Сделать ~~переменную~~ amount закрытым членом класса
~~Побочные эффекты в классе Dollar~~?
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов

Итак, сейчас операция проверки равенства реализована полностью. Но как учесть сравнение со значением null и сравнение с другими объектами? Это часто используемые операции, пока они нам еще не нужны, поэтому мы просто добавим их в список задач.

Теперь, когда у нас есть операция проверки равенства, можно напрямую сравнивать объекты Dollar. Это позволит нам сделать переменную amount закрытой, какой и должна быть добропорядочная переменная экземпляра. Резюмируем все вышесказанное:

• поняли, что для использования шаблона проектирования «Объект-значение» необходимо реализовать операцию проверки равенства;

• создали тест для этой операции;

• реализовали ее простейшим способом;

• продолжили тестирование (вместо того, чтобы сразу приступить к рефакторингу);

• выполнили рефакторинг так, чтобы охватить оба теста сразу.

4. Данные должны быть закрытыми

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
Сделать переменную amount закрытым членом класса
~~Побочные эффекты в классе Dollar~~?
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов

Теперь, когда определена операция проверки равенства, с ее помощью можно повысить наглядность тестов. По идее, метод Dollar.times() должен возвращать новый объект Dollar, величина которого равна величине исходного объекта (метод которого мы вызываем), умноженной на коэффициент. Однако наш тест не показывает этого явно:

```
public void testMultiplication() {
Dollar five = new Dollar(5);
Dollar product = five.times(2);
assertEquals(10, product.amount);
product = five.times(3);
assertEquals(15, product.amount);
}
```

Мы можем переписать первую проверку и сравнить в ней объекты Dollar:

```
public void testMultiplication() {
Dollar five = new Dollar(5);
Dollar product = five.times(2);
assertEquals(new Dollar(10), product);
product = five.times(3);
assertEquals(15, product.amount);
}
```

Выглядит неплохо, поэтому перепишем и вторую проверку:

```
public void testMultiplication() {
Dollar five = new Dollar(5);
Dollar product = five.times(2);
assertEquals(new Dollar(10), product);
product = five.times(3);
assertEquals(new Dollar(15), product);
}
```

Теперь нам не нужна вспомогательная переменная product, поэтому устраним ее:

```
public void testMultiplication() {
Dollar five= new Dollar(5);
assertEquals(new Dollar(10), five.times(2));
assertEquals(new Dollar(15), five.times(3));
}
```

Согласитесь, этот вариант теста значительно нагляднее.

Учтем внесенные изменения. Теперь только класс Dollar использует переменную экземпляра amount, поэтому мы можем сделать ее закрытой:

Dollar

```
private int amount;
```

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым членом класса~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов

Вычеркиваем еще один пункт из списка задач. Заметьте, мы подвергли себя риску: если тест, проверяющий равенство, не смог бы точно определить корректность операции сравнения, тогда и тест умножения не смог бы проверить, правильно ли оно работает. В TDD принято активное

управление риском. Мы не гонимся за совершенством. Выражая все двумя способами – тестами и кодом, – мы надеемся уменьшить дефекты настолько, чтобы уверенно идти дальше. Время от времени наши рассуждения будут нас подводить, позволяя появляться ошибкам. Когда это случится, мы вспомним урок о том, что надо написать тест и двигаться дальше. Все остальное время мы отважно продвигаемся вперед под победно развевающейся зеленой полоской нашего индикатора (вообще-то мой индикатор не развевается, но я люблю помечтать).

Подведем итоги:

- использовали только что разработанную функциональность для улучшения теста;
- заметили, что, если одновременно два теста терпят неудачу, наши дела плохи;
- продолжили несмотря на риск;
- использовали новую функциональность тестируемого объекта для уменьшения зависимости между тестами и кодом.

5. Поговорим о франках

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?Округление денежных величин?~~
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов
5 CHF * 2 = 1 °CHF

Можем ли мы приступить к реализации первого, самого интересного теста в данном списке? Мне все еще кажется, что это будет слишком большой шаг. Я не представляю себе, как можно написать этот тест за один маленький шажок. Мне кажется, что вначале необходимо создать объект наподобие Dollar, который соответствовал бы не долларам, а франкам. Пусть это будет объект с названием Franc. Для начала объект Franc может функционировать в точности как объект Dollar – если у нас будет такой объект, нам будет проще размышлять о реализации теста, связанного со смешанным сложением двух разных валют.

А если объект Franc работает так же, как объект Dollar, значит, мы можем просто скопировать и слегка отредактировать тест для объекта Dollar:

```
public void testFrancMultiplication() {
Franc five = new Franc(5);
assertEquals(new Franc(10), five.times(2));
assertEquals(new Franc(15), five.times(3));
}
```

(Хорошо, что в главе 4 мы упростили тест для Dollar. Благодаря этому работа по редактированию теста существенно упростилась. Похоже, в данной книге дела идут довольно гладко, однако я не могут гарантировать, что в будущем все будет так же хорошо.)

Теперь нам надо получить зеленую полоску. Какой способ будет самым простым? Проще всего скопировать код класса Dollar и заменить

Dollar на Franc.

Стоп. Подождите-ка. Я уже вижу, как некоторые наиболее ярые сторонники правильных подходов начинают морщиться и плеваться. Повторное использование кода путем его дублирования через буфер обмена? Пренебрежение абстракцией? А как же все эти разговоры об основополагающих принципах ООП и чистом дизайне?

Если вам не по себе, глубоко вдохните через нос, досчитайте до трех и медленно выдохните через рот. Вам лучше? Теперь вспомните, что наш цикл состоит из пяти этапов. Иногда последовательное выполнение всех этапов занимает всего несколько секунд, однако в любом случае мы обязательно выполняем каждый из них:

1. Написать тест.
2. Добиться его безошибочной компиляции.
3. Запустить тест и убедиться, что он потерпел неудачу.
4. Добиться успешного выполнения теста.
5. Устранить дублирование.

На разных этапах решаются разные задачи, преследуются разные цели. То, что совершенно недопустимо для одного из этапов, может быть вполне приемлемым для другого этапа. Однако в целом методика TDD работает только в случае, если ни один из этапов не упущен. Если вы пропустите хотя бы одно звено, развалится вся цепочка.

Первые три фазы цикла разработки TDD должны выполняться как можно быстрее. Определяющая характеристика этих этапов – скорость. На этих этапах в жертву скорости можно принести очень многое, в том числе чистоту дизайна. Честно говоря, сейчас я несколько волнуюсь. Я только что разрешил вам забыть о принципах хорошего дизайна. Представляю, как вы приходите к своим коллегам, подчиненным и во всеуслышание объявляете: «Кент сказал, что все эти разговоры про хороший дизайн – полная ерунда!» Остановитесь. Цикл еще не закончен. Четырехногий уродец из благородного семейства пятиногих стульев вечно падает. Первые четыре шага нашего цикла не работают без пятого. Хороший дизайн в подходящее время! Сначала сделаем, чтобы код заработал, потом сделаем, чтобы код был правильным (make it run, make it right).

Теперь мне стало легче. Теперь я уверен, что до тех пор, пока вы не избавитесь от дублирования, вы не покажете свой код никому за исключением своего партнера по паре. На чем мы остановились? Ах, да. Забываем о принципах хорошего дизайна в угоду скорости (мы будем заниматься искуплением этого греха на протяжении нескольких следующих глав).

Franc

```
class Franc {
private int amount;
Franc(int amount) {
this.amount = amount;
}

Franc times(int multiplier) {
return new Franc(amount * multiplier);
}

public boolean equals(Object object) {
Franc franc = (Franc) object;
return amount == franc.amount;
}
}
```

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов
~~5 CHF * 2 = 1 °CHF~~
Дублирование Dollar/Franc
Общие операции equals()
Общие операции times()

Чтобы запустить код, нам не потребовалось прикладывать каких-либо усилий, поэтому мы смогли «перепрыгнуть» через этап «добиться безошибочной компиляции кода» (Make it compile).

Зато теперь в нашем коде полно повторяющихся фрагментов. Прежде чем приступить к разработке следующего теста, мы должны избавиться от дублирования. Думаю, что следует начать с обобщения метода equals(). Однако об этом в следующей главе. На текущий момент мы можем вычеркнуть из нашего списка еще один пункт, однако вместе с этим нам

придется добавить в него два дополнительных пункта.

В данной главе мы

- решили отказаться от создания слишком большого теста и вместо этого создали маленький, чтобы обеспечить быстрый прогресс;
- создали код теста путем бесстыдного копирования и редактирования;
- хуже того, добились успешного выполнения теста путем копирования и редактирования разработанного ранее кода;
- дали себе обещание ни в коем случае не уходить домой до тех пор, пока не устраним дублирование.

6. Равенство для всех, вторая серия

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов
~~5 CHF * 2 = 1 °CHF~~
Дублирование Dollar/Franc

Общие операции equals()
Общие операции times()

В книге *Crossing to Safety* автор Вэлленц Стегнер (Wallance Stegner) описывает рабочее место одного из персонажей. Каждый инструмент находится на предназначенном для него месте, пол чисто вымыт и подметен, повсюду превосходный порядок и чистота. Однако чтобы добиться подобного положения вещей, персонаж не делал никаких специальных подготовительных процедур. «Подготовка была делом всей его жизни. Он подготавливался, затем убирался на рабочем месте». (Конец книги заставил меня громко рассмеяться в бизнес-классе трансатлантического «Боинга-747». Так что если решите ее прочитать, читайте с осторожностью.)

В главе 5 мы добились успешного выполнения теста. Это значит, что требуемая функциональность реализована. Однако чтобы сделать это быстро, нам пришлось продублировать огромный объем кода. Теперь пришло время убрать за собой.

Мы можем сделать так, чтобы один из разработанных нами классов стал производным от другого. Я попробовал сделать это, однако понял, что в этом случае ничего не выигрываю. Вместо этого удобнее создать суперкласс, который станет базовым для обоих разработанных нами классов. Ситуация проиллюстрирована на рис. 6.1. (Я уже пробовал так поступить и пришел к выводу, что это именно то, что нужно, однако

придется приложить усилия.)

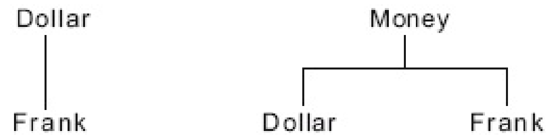

Рис. 6.1. Общий суперкласс для двух разработанных нами классов

Для начала попробуем реализовать в базовом классе Money общий для обоих производных классов метод equals(). Начнем с малого:

Money

```
class Money
```

Запустим тесты – они по-прежнему выполняются. Конечно же, мы пока не сделали ничего такого, что нарушило бы выполнение наших тестов, однако в любом случае лишний раз запустить тесты не помешает. Теперь попробуем сделать класс Dollar производным от класса Money:

Dollar

```
class Dollar extends Money {
private int amount;
}
```

Работают ли тесты? Работают. Можем двигаться дальше. Перемещаем переменную amount в класс Money:

Money

```
class Money {
protected int amount;
}
```

Dollar

```
class Dollar extends Money {
}
```

Режим видимости переменной amount потребовалось изменить: теперь вместо private используем модификатор доступа protected. В противном

случае подкласс не сможет обратиться к этой переменной. (Если бы мы хотели двигаться еще медленнее, мы могли бы на первом шаге объявить переменную в классе Money, а на втором шаге удалить ее объявление из класса Dollar, однако я решил действовать смело и решительно.)

Теперь можно переместить код метода equals() вверх по иерархии классов, то есть в класс Money. Прежде всего мы изменим объявление временной переменной:

Dollar
```
public boolean equals(Object object) {
Money dollar = (Dollar) object;
return amount == dollar.amount;
}
```

Все тесты по-прежнему работают. Теперь попробуем изменить приведение типа.

Dollar
```
public boolean equals(Object object) {
Money dollar = (Money) object;
return amount == dollar.amount;
}
```

Чтобы исходный код получился более осмысленным, изменим имя временной переменной:

Dollar
```
public boolean equals(Object object) {
Money money = (Money) object;
return amount == money.amount;
}
```

Теперь переместим метод из класса Dollar в класс Money:

Money
```
public boolean equals(Object object) {
Money money= (Money) object;
return amount == money.amount;
}
```

Теперь настало время удалить метод Franc.equals(). Прежде всего мы обнаруживаем, что у нас до сих пор нет теста, проверяющего равенство двух объектов класса Franc, – когда мы, особо не раздумывая, дублировали код класса Dollar, мы нагрешили еще больше, чем думали. Поэтому, прежде чем модифицировать код, мы должны написать все необходимые тесты.

В ближайшем будущем, скорее всего, вам придется использовать подход TDD в отношении кода, который не сопровождается достаточным количеством тестов. В отсутствие адекватного набора тестов любой рефакторинг может привести к нарушению работоспособности кода. Иными словами, в ходе рефакторинга можно допустить ошибку, при этом все имеющиеся тесты будут выполняться как ни в чем не бывало. Ошибка может вскрыться слишком поздно, а ее устранение может стоить слишком дорого. Что же делать?

Прежде чем что-либо менять в коде, вы должны написать все тесты, которые кажутся вам необходимыми. Если этого не сделать, рано или поздно, выполняя рефакторинг, вы чего-нибудь поломаете. Код перестанет работать так, как должен. Вы потратите кучу времени на поиск ошибки и сформируете предубеждение против рефакторинга. Если подобный инцидент повторится, вы можете вообще перестать делать рефакторинг. Дизайн начнет деградировать. Вас уволят с работы. От вас уйдет ваша любимая собака. Вы перестанете мыться и чистить зубы. У вас начнется кариес. Чтобы сохранить зубы здоровыми, всегда сначала пишите тесты и только после этого выполняйте рефакторинг.

К счастью, в нашем случае написать тесты совсем несложно. Для этого достаточно скопировать и немножко отредактировать тесты для класса Dollar:

```
public void testEquality() {

assertTrue(new Dollar(5). equals(new Dollar(5)));
assertFalse(new Dollar(5). equals(new Dollar(6)));
assertTrue(new Franc(5). equals(new Franc(5)));
assertFalse(new Franc(5). equals(new Franc(6)));
}
```

Снова дублирование. Целых две строчки! Этот грех нам тоже придется искупить. Но чуть позже.

Теперь, когда тесты на месте, мы можем сделать класс Franc

производным от класса Money:

Franc
```
class Franc extends Money {
private int amount;
}
```

Далее мы можем уничтожить поле amount в классе Franc, так как это значение будет храниться в одноименном поле класса Money:

Franc
```
class Franc extends Money {
}
```

Метод Franc.equals() выглядит фактически так же, как и метод Money.equals(). Сделав их абсолютно одинаковыми, мы сможем удалить реализацию этого метода из класса Franc. При этом смысл нашей программы не изменится. Для начала изменим объявление временной переменной:

Franc
```
public boolean equals(Object object) {
Money franc = (Franc) object;
return amount == franc.amount;
}
```

После этого изменим операцию преобразования типа:

Franc
```
public boolean equals(Object object) {
Money franc = (Money) object;
return amount == franc.amount;
}
```

Теперь, даже не меняя имя временной переменной, можно видеть, что метод получился фактически таким же, как одноименный метод в классе Money. Однако для пущей уверенности переименуем временную переменную:

Franc

```
public boolean equals(Object object) {
Money money = (Money) object;
return amount == money.amount;
}
```

$5 + 1 °CHF = $10, если курс обмена 2:1

~~$5 * 2 = $10~~

~~Сделать переменную amount закрытым (private) членом~~

~~Побочные эффекты в классе Dollar?~~

Округление денежных величин?

~~equals()~~

hashCode()

Равенство значению null

Равенство объектов

~~5 CHF * 2 = 1 °CHF~~

Дублирование Dollar/Franc

~~Общие операции equals()~~

Общие операции times()

Сравнение франков (Franc) и долларов (Dollar)

Теперь нет никакой разницы между методами Franc.equals() и Money.equals(), и мы можем удалить избыточную реализацию этого метода из класса Franc. Запускаем тесты. Они выполняются успешно.

Что должно происходить при сравнении франков и долларов? Мы рассмотрим этот вопрос в главе 7.

В данной главе мы

- поэтапно переместили общий код из одного класса (Dollar) в суперкласс (Money);
- сделали второй класс (Franc) подклассом общего суперкласса (Money);
- унифицировали две реализации метода equals() и удалили избыточную реализацию в классе Franc.

7. Яблоки и апельсины

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
equals()
hashCode()
Равенство значению null
Равенство объектов
~~5 CHF * 2 = 1 °CHF~~
Дублирование Dollar/Franc
~~Общие операции equals()~~
Общие операции times()
Сравнение франков (Franc) и долларов (Dollar)

В конце главы 6 перед нами встал интересный вопрос: что будет, если мы сравним франки и доллары? Мы немедленно добавили соответствующий пункт в список предстоящих задач. Нам никак не избавиться от этой мысли. И в самом деле, что произойдет?

```
public void testEquality() {
assertTrue(new Dollar(5). equals(new Dollar(5)));
assertFalse(new Dollar(5). equals(new Dollar(6)));
assertTrue(new Franc(5). equals(new Franc(5)));
assertFalse(new Franc(5). equals(new Franc(6)));
assertFalse(new Franc(5). equals(new Dollar(5)));
}
```

Тест завершается неудачей. С точки зрения написанного кода доллары – это франки. Прежде чем у наших швейцарских клиентов глаза вылезут на лоб, давайте попробуем исправить код. Код сравнения двух денежных значений должен убедиться в том, что он не сравнивает доллары с франками. Для этого мы должны проверить классы сравниваемых объектов – два объекта класса Money считаются равными только в том случае, если у них равны значения amount и классы.

```
public boolean equals(Object object) {
Money money = (Money) object;
return amount == money.amount
&& getClass(). equals(money.getClass());
}
```

Подобное использование классов, по правде сказать, отдает неприятным запашком. Предпочтительнее было бы использовать критерий из области финансов, а не из области объектов языка Java. Однако на текущий момент в нашей программе еще нет ничего, что соответствовало бы финансовому понятию «валюта», и пока я не вижу достаточно весомой причины, чтобы вводить в программу подобное понятие. Поэтому пока оставим код таким, какой он есть.

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов
~~5 CHF * 2 = 1 °CHF~~
Дублирование Dollar/Franc
~~Общие операции equals()~~
Общие операции times()
~~Сравнение франков (Franc) и долларов (Dollar)~~
Валюта?

Теперь пришла пора избавиться от дублирующегося кода в методах times(), после этого мы сможем перейти к реализации смешанной валютной арифметики. Однако прежде, чем двинуться дальше, подведем итоги данной главы:

• мы превратили мучающее нас сомнение в тест;

• добились успешного выполнения теста приемлемым, но не идеальным способом – getClass();

• решили не добавлять в программу дополнительной логики, пока у

нас не появится более весомая мотивация.

8. Создание объектов

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов
~~5 CHF * 2 = 1 °CHF~~
Дублирование Dollar/Franc
~~Общие операции equals()~~
Общие операции times()
~~Сравнение франков (Franc) и долларов (Dollar)~~
Валюта?

Две разные реализации метода times() выглядят на удивление похоже:

Franc
```
Franc times(int multiplier) {
return new Franc(amount * multiplier)
}
```
Dollar
```
Dollar times(int multiplier) {
return new Dollar(amount * multiplier)
}
```

Мы можем сделать их еще более похожими, изменив тип возвращаемого значения на Money:

Franc
Money times(int multiplier) {
return new Franc(amount * multiplier)
}
Dollar

```
Money times(int multiplier) {
return new Dollar(amount * multiplier)
}
```

Следующий шаг менее очевиден. Два подкласса, производных от класса Money, мало чем отличаются друг от друга. Возникает желание избавиться от них. Однако мы не можем сделать это за один большой шаг, так как это нельзя будет назвать наглядной демонстрацией методики TDD.

Но что же делать? Полагаю, мы сможем приблизиться к решению задачи об уничтожении подклассов, если избавимся от прямых ссылок на подклассы. Для этого мы можем добавить в класс Money фабричный метод, который возвращал бы объект класса Dollar. Этот метод можно было бы использовать следующим образом:

```
public void testMultiplication() {
Dollar five = Money.dollar(5);
assertEquals(new Dollar(10), five.times(2));
assertEquals(new Dollar(15), five.times(3));
}
```

Реализация этого метода создает объект класса Dollar и возвращает его:

```
Money
static Dollar dollar(int amount) {
return new Dollar(amount);
}
```

Однако мы хотим избавиться от ссылок на Dollar, поэтому изменим объявление переменной в коде теста:

```
public void testMultiplication() {
Money five = Money.dollar(5);
assertEquals(new Dollar(10), five.times(2));
assertEquals(new Dollar(15), five.times(3));
}
```

Компилятор вежливо сообщает нам, что метод times() в классе Money не определен. На текущий момент мы не можем реализовать его, поэтому

объявим класс Money абстрактным (может быть, с этого стоило начать?) и объявим также абстрактным метод Money.times():

```
Money
abstract class Money
abstract Money times(int multiplier);
```

Теперь мы можем изменить объявление фабричного метода:

```
Money
static Money dollar(int amount) {
return new Dollar(amount);
}
```

Все тесты выполняются успешно, стало быть, по крайней мере, мы ничего не сломали. Теперь мы можем использовать фабричный метод повсюду в тестах:

```
public void testMultiplication() {
Money five = Money.dollar(5);
assertEquals(Money.dollar(10), five.times(2));
assertEquals(Money.dollar(15), five.times(3));
}
public void testEquality() {
assertTrue(Money.dollar(5). equals(Money.dollar(5)));
assertFalse(Money.dollar(5). equals(Money.dollar(6)));
assertTrue(new Franc(5). equals(new Franc(5)));
assertFalse(new Franc(5). equals(new Franc(6)));
assertFalse(new Franc(5). equals(Money.dollar(5)));
}
```

Теперь мы находимся в несколько более выгодной позиции, чем раньше. Клиентский код ничего не знает о существовании подкласса Dollar. Освободив код тестов от ссылок на подклассы, мы получили возможность изменять структуру наследования, не внося при этом каких-либо изменений в клиентский код.

Прежде чем механически исправлять код теста testFrancMultiplication(), обратите внимание, что теперь он не тестирует никакой логики, кроме той, что уже протестирована функцией

testMultiplication(). Напрашивается вопрос: нужна ли нам функция testFrancMultiplication()? Если мы удалим этот тест, потеряем ли мы уверенность в нашем коде? Похоже, что нет, однако мы все же сохраним пока этот тест просто так – на всякий случай.

```
public void testEquality() {
assertTrue(Money.dollar(5). equals(Money.dollar(5)));
assertFalse(Money.dollar(5). equals(Money.dollar(6)));
assertTrue(Money.franc(5). equals(Money.franc(5)));
assertFalse(Money.franc(5). equals(Money.franc(6)));
assertFalse(Money.franc(5). equals(Money.dollar(5)));
}
public void testFrancMultiplication() {
Money five = Money.franc(5);
assertEquals(Money.franc(10), five.times(2));
assertEquals(Money.franc(15), five.times(3));
}
```

Реализация метода Money.franc() почти такая же, как и реализация метода Money.dollar():

Money

```
static Money franc(int amount) {
return new Franc(amount);
}
```

$5 + 1 °CHF = $10, если курс обмена 2:1

~~$5 * 2 = $10~~

~~Сделать переменную amount закрытым (private) членом~~

~~Побочные эффекты в классе Dollar?~~

Округление денежных величин?

~~equals()~~

hashCode()

Равенство значению null

Равенство объектов

~~5 CHF * 2 = 1 °CHF~~

Дублирование Dollar/Franc

~~Общие операции equals()~~

Общие операции times()

~~Сравнение франков (Franc) и долларов (Dollar)~~

Валюта?

Нужен ли тест testFrancMultiplication()?

Далее мы планируем перейти к устранению дублирования в методах times(). А сейчас вспомним, что в данной главе мы

• сделали шаг на пути к устранению дублирования – сформировали общую сигнатуру для двух вариантов одного метода – times();

• добавили объявление метода в общий суперкласс;

• освободили тестовый код от ссылок на производные классы, для этого были созданы фабричные методы;

• заметили, что, когда подклассы исчезли, некоторые тесты стали избыточными, однако никаких действий предпринято не было.

9. Потребность в валюте

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов
~~5 CHF * 2 = 1 °CHF~~
Дублирование Dollar/Franc
~~Общие операции equals()~~
Общие операции times()
~~Сравнение франков (Franc) и долларов (Dollar)~~
Валюта?
Нужен ли тест testFrancMultiplication()?

Есть ли в нашем списке задач какой-либо пункт, который помог бы нам избавиться от этих надоедливых подклассов? Что произойдет, если мы попробуем ввести в нашу программу понятие валюты?

Каким образом мы можем реализовать понятие валюты в данный момент? Черт! Опять я говорю ерунду! Вместо того чтобы снова бить себя линейкой по рукам, попробую перефразировать: каким образом мы можем протестировать понятие валюты в данный момент? Слава богу, мои руки спасены.

Возможно, в будущем нам захочется создать специальный класс валюты, применив шаблон «Приспособленец» (Flyweight Factory), чтобы избежать создания лишних объектов. Однако на текущий момент понятие валюты вполне можно реализовать в виде обычных строк:

```
public void testCurrency() {
assertEquals("USD", Money.dollar(1). currency());
assertEquals("CHF", Money.franc(1). currency());
}
```

Прежде всего объявим метод currency() в классе Money:

Money
```
abstract String currency();
```

Теперь реализуем этот метод в обоих подклассах:

Franc
```
String currency() {
return "CHF";
}
```

Dollar
```
String currency() {
return "USD";
}
```

Однако хотелось бы иметь одну и ту же реализацию в обоих подклассах, поэтому сохраним идентификатор валюты в специальном поле класса и просто вернем его значение в методе currency(). (Я начинаю делать рефакторинг быстрее, чтобы сэкономить время. Если вам покажется, что я рассказываю слишком быстро, не стесняйтесь попросить меня замедлиться. Постойте-ка, это же книга, – наверное, мне не стоит слишком уж ускоряться.)

Franc
```
private String currency;
Franc(int amount) {
this.amount = amount;
currency = "CHF";
}
String currency() {
return currency;
}
```

То же самое сделаем в классе Dollar:

Dollar
```
private String currency;
```

```
Dollar(int amount) {
this.amount = amount;
currency = "USD";
}
String currency() {
return currency;
}
```

Теперь мы можем переместить объявление переменной и реализацию метода currency() в базовый класс Money, так как в обоих подклассах они одинаковые:

Money

```
protected String currency;
String currency() {
return currency;
}
```

Если переместить строковые констаны «USD» и «CHF» в статические фабричные методы, конструкторы двух производных классов станут идентичными, в результате мы сможем создать общую реализацию.

Вначале добавим в конструктор дополнительный параметр:

Franc

Franc(int amount, **String currency**) {
this.amount = amount;
this.currency = "CHF";
}

При этом возникают ошибки в двух местах при обращении к конструктору:

Money

```
static Money franc(int amount) {
return new Franc(amount, null);
}
```

Franc

```
Money times(int multiplier) {
```

```
return new Franc(amount * multiplier, null);
}
```

Постойте-ка! Почему это метод Franc.times() вызывает конструктор вместо фабричного метода? Будем ли мы заниматься этим сейчас или отложим любые связанные с этим модификации на более позднее время? В рамках догмы мы должны оставить посторонние дела на потом – не следует прерывать то, чем мы сейчас занимаемся. Однако на практике я иногда позволяю себе отвлечься, но лишь ненадолго, и ни в коем случае я не разрешаю себе прерывать прерывание (этому правилу научил меня Джим Коплаен – Jim Coplien). В данном случае будет лучше, если мы подчистим метод times(), прежде чем продолжить:

Franc

```
Money times(int multiplier) {
return Money.franc(amount * multiplier);
}
```

Теперь фабричному методу можно передать значение «CHF»:

Money

```
static Money franc(int amount) {
return new Franc(amount,«CHF»);
}
```

Наконец, мы можем присвоить значение параметра полю класса:

Franc

```
Franc(int amount, String currency) {
this.amount = amount;
this.currency = currency;
}
```

Может показаться, что я снова перемещаюсь вперед слишком маленькими шажками. Действительно ли я рекомендую вам работать в таком же темпе? Нет. Я рекомендую вначале научиться работать в таком темпе, а затем самостоятельно определять скорость работы, которая покажется вам наиболее эффективной. Я всего лишь попробовал двигаться вперед большими шагами и на половине дороги допустил глупую ошибку. Запутавшись, я вернулся назад на несколько минут, перешел на

пониженную передачу и сделал работу заново, более мелкими шажками. Сейчас я чувствую себя уверенней, поэтому мы можем попробовать внести такие же изменения в класс Dollar за один большой шаг:

```
Money
static Money dollar(int amount) {
return new Dollar(amount,«USD»);
}

Dollar
Dollar(int amount, String currency) {
this.amount = amount;
this.currency = currency;
}
Money times(int multiplier) {
return Money.dollar(amount * multiplier);
}
```

И это сработало с первого раза. Классно!

Подобная настройка скорости весьма характерна для TDD. Вам кажется, что слишком маленькие шажки ограничивают вас? Попробуйте двигаться быстрее. Почувствовали неуверенность? Переходите на короткий шаг. TDD – это процесс плавного управления – немного в одну сторону, немного в другую сторону. Не существует одного-единственного наиболее правильного размера шага, ни сейчас, ни в будущем.

Теперь два конструктора выглядят абсолютно одинаково, и мы можем переместить реализацию в базовый класс:

```
Money
Money(int amount, String currency) {
this.amount = amount;
this.currency = currency;
}

Franc
Franc(int amount, String currency) {
super(amount, currency);
}
```

Dollar

Dollar(int amount, String currency) {
super(amount, currency);
}

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов
~~5 CHF * 2 = 1 °CHF~~
Дублирование Dollar/Franc
~~Общие операции equals()~~
Общие операции times()
~~Сравнение франков (Franc) и долларов (Dollar)~~
~~Валюта?~~
Нужен ли тест testFrancMultiplication()?

Мы уже почти готовы переместить реализацию times() в базовый класс, но прежде вспомним, что в данной главе мы

- на некоторое время заблудились в крупномасштабных идеях дизайна и, чтобы разобраться в проблеме, решили начать с решения небольшой задачи, на которую мы уже обратили внимание ранее;
- сделали одинаковыми два конструктора, переместив отличающийся код в вызывающий (фабричный) метод;
- на короткое время отвлеклись от рефакторинга, чтобы добавить в метод times() вызов фабричного метода;
- выполнили аналогичный рефакторинг в отношении класса Dollar за один большой шаг;
- получили два абсолютно идентичных конструктора и переместили код в базовый класс.

10. Избавление от двух разных версий times()

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов
~~5 CHF * 2 = 1 °CHF~~
Дублирование Dollar/Franc
~~Общие операции equals()~~
Общие операции times()
~~Сравнение франков (Franc) и долларов (Dollar)~~
~~Валюта?~~
Нужен ли тест testFrancMultiplication()?

В конце данной главы мы должны получить единый класс Money, соответствующий понятию «деньги». Две реализации метода times() близки друг к другу, однако они не идентичны:

Franc
```
Money times(int multiplier) {
return Money.franc(amount * multiplier);
}
```

Dollar
```
Money times(int multiplier) {
return Money.dollar(amount * multiplier);
}
```

Увы, я не вижу простого способа добиться идентичности этих методов, однако в некоторых ситуациях, для того чтобы продвинуться дальше, требуется вернуться немного назад, – это напоминает кубик Рубика. Что будет, если мы заменим вызовы фабричных методов

операторами new? (Я отлично понимаю, что совсем недавно мы выполнили обратную процедуру – заменили new вызовами фабричных методов. Но что я могу поделать – сейчас мы решаем несколько иную задачу. Понимаю, что это может показаться обескураживающим, однако потерпите немного.)

Franc
```
Money times(int multiplier) {
return new Franc(amount * multiplier, "CHF");
}
```

Dollar
```
Money times(int multiplier) {
return new Dollar(amount * multiplier, "USD");
}
```

Мы абсолютно уверены, что в экземплярах класса Franc значение поля currency всегда будет равно «CHF», поэтому можем написать:

Franc
```
Money times(int multiplier) {
return new Franc(amount * multiplier, currency);
}
```

Сработало! Теперь тот же трюк можно проделать и в отношении класса Dollar:

Dollar
```
Money times(int multiplier) {
return new Dollar(amount * multiplier,currency);
}
```

Мы почти закончили. Имеет ли значение, что мы используем в данном случае – Franc или Money? Об этом можно рассуждать в течение некоторого времени исходя из имеющихся знаний о внутреннем устройстве нашей системы, однако у нас есть чистый код и тесты, которые дают нам уверенность в том, что код работает так, как надо. Вместо того чтобы тратить несколько минут на рассуждения, мы можем спросить об этом компьютер. Для этого достаточно внести интересующие нас изменения в код и запустить тесты. Обучая методике TDD, я наблюдаю подобную

ситуацию постоянно – опытные умные программисты тратят от 5 до 10 минут на обсуждение вопроса, на который компьютер может дать ответ в течение 15 секунд. Если у вас нет тестов, вам остается только размышлять и предполагать. Если же у вас есть тесты, вместо того, чтобы напрасно тратить время, вы можете провести быстрый эксперимент. Как правило, если у вас есть тесты, быстрее спросить компьютер.

Чтобы провести интересующий нас эксперимент, модифицируем код так, чтобы метод Franc.times() возвращал значение типа Money:

Franc
```
Money times(int multiplier) {
return new Money (amount * multiplier, currency);
}
```

В ответ компилятор сообщил, что Money должен быть конкретным (не абстрактным) классом:

Money
```
class Money
Money times(int amount) {
return null;
}
```

Получаем красную полоску и сообщение об ошибке: «expected: <Money.Franc@31aebf> but was:<Money.Money@478a43>». Не очень-то информативно. Не так информативно, как нам хотелось бы. Чтобы получить более осмысленное сообщение об ошибке, добавим метод toString():

Money
```
public String toString() {
return amount + " " + currency;
}
```

О, ужас! Код без тестов?! Допустимо ли такое? Конечно же, прежде чем писать код метода toString, мы должны были написать соответствующий тест, однако

• мы увидим результаты работы этого метода на экране;

• метод toString() используется только для отладки, поэтому риск,

связанный с потенциальными ошибками, невелик;

• перед нами красная полоса, а мы предпочитаем не писать новых тестов, пока не избавимся от красной полосы.

Обстоятельства приняты к сведению.

Теперь сообщение об ошибке изменилось: "expected:<1 °CHF> but was: <1 °CHF>". Выглядит осмысленней, однако сбивает с толку. В двух объектах хранятся одни и те же данные, однако при этом объекты не считаются равными. Проблема кроется в реализации метода equals():

Money

```
public boolean equals(Object object) {
Money money = (Money) object;
return amount == money.amount
&& getClass(). equals(money.getClass());
}
```

В данном случае происходит сравнение имен классов, в то время как логичнее сравнивать идентификаторы валют.

Лучше не писать никаких новых тестов, если перед вами красная полоса. Однако нам нужно внести изменения в разрабатываемый код, и мы не можем изменить код, не обладая соответствующим тестом. Консервативный подход заключается в том, чтобы отменить изменение, которое привело к появлению красной полосы. В этом случае мы вновь получим зеленую полосу. После этого мы сможем модифицировать тест для метода equals(), исправить его реализацию и вновь применить изначальное изменение.

В данном случае мы будем действовать консервативно. (Иногда я плюю на все и пишу тест, не обращая внимания на красную полосу, однако я поступаю так, только когда дети уже спят.)

Franc

```
Money times(int multiplier) {
return new Franc (amount * multiplier, currency);
}
```

Перед нами снова зеленая полоса. Мы попали в ситуацию, когда объект Franc(10,"CHF") не равен объекту Money(10,"CHF"), хотя нам хотелось бы, чтобы эти объекты были равны. Превращаем наше желание в тест:

```
public void testDifferentClassEquality() {
assertTrue(new Money(10, "CHF"). equals(new Franc(10, "CHF")));
}
```

Как и ожидалось, тест потерпел неудачу. Код метода equal() должен сравнивать идентификаторы валют, а не имена классов:

Money

```
public boolean equals(Object object) {
Money money = (Money) object;
return amount == money.amount
&& currency(). equals(money.currency());
}
```

Теперь метод Franc.times() может возвращать значение Money, и все тесты будут по-прежнему успешно выполняться:

Franc

```
Money times(int multiplier) {
return new Money(amount * multiplier, currency);
}
```

Сработает ли этот трюк для метода Dollar.times()?

Dollar

```
Money times(int multiplier) {
return new Money (amount * multiplier, currency);
}
```

Да! Теперь две реализации абсолютно идентичны, и мы можем переместить их в базовый класс.

Money

```
Money times(int multiplier) {
return new Money(amount * multiplier, currency);
}
```

$5 + 1 °CHF = $10, если курс обмена 2:1

~~$5 * 2 = $10~~

~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов
~~5 CHF * 2 = 1 °CHF~~
Дублирование Dollar/Franc
~~Общие операции equals()~~
~~Общие операции times()~~
~~Сравнение франков (Franc) и долларов (Dollar)~~
~~Валюта?~~
Нужен ли тест testFrancMultiplication()?

Метод умножения там, где ему следует быть, теперь мы готовы удалить ненужные нам производные классы.

В данной главе мы

• сделали идентичными две реализации метода times(), для этого мы избавились от вызовов фабричных методов в них, и заменили константы переменными;

• добавили в класс отладочный метод toString() без теста;

• попробовали модифицировать код (заменили тип Franc возвращаемого значения на Money) и обратились к тестам, чтобы узнать, сработает ли это;

• отменили изменения и написали еще один тест, добились успешного выполнения теста и вновь применили изменения.

11. Корень всего зла

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов
~~5 CHF * 2 = 1 °CHF~~
Дублирование Dollar/Franc
~~Общие операции equals()~~
~~Общие операции times()~~
~~Сравнение франков (Franc) и долларов (Dollar)~~
~~Валюта?~~
Нужен ли тест testFrancMultiplication()?

Два производных класса, Dollar и Franc, обладают только конструкторами, однако конструктор – это недостаточная причина для создания подкласса. Мы должны избавиться от бесполезных подклассов.

Ссылки на подклассы можно заменить ссылками на суперкласс, не изменив при этом смысл кода. Начнем с класса Franc:

Franc
static Money franc(int amount) {
return new **Money** (amount, «CHF»);
}

Затем перейдем к классу Dollar:

Dollar
static Money dollar(int amount) {
return new **Money** (amount, «USD»);
}

Ссылок на класс Dollar больше нет, поэтому мы можем удалить этот класс. Однако в только что написанном нами тесте есть одна ссылка на класс Franc:

```
public void testDifferentClassEquality() {
assertTrue(new Money(10, "CHF"). equals(new Franc(10, "CHF")));
}
```

Если равенство объектов достаточно хорошо протестировано другими тестами, значит, мы можем безбоязненно удалить этот тест. Давайте взглянем на другие тесты:

```
public void testEquality() {
assertTrue(Money.dollar(5). equals(Money.dollar(5)));
assertFalse(Money.dollar(5). equals(Money.dollar(6)));
assertTrue(Money.franc(5). equals(Money.franc(5)));
assertFalse(Money.franc(5). equals(Money.franc(6)));
assertFalse(Money.franc(5). equals(Money.dollar(5)));
}
```

Похоже, что все возможные случаи определения равенства достаточно полно охвачены другими тестами. Я даже сказал бы, что тестов слишком много. Мы можем удалить третье и четвертое выражение assert, так как они дублируют первое и второе:

```
public void testEquality() {
assertTrue(Money.dollar(5). equals(Money.dollar(5)));
assertFalse(Money.dollar(5). equals(Money.dollar(6)));
assertFalse(Money.franc(5). equals(Money.dollar(5)));
}
```

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
~~Сделать переменную amount закрытым (private) членом~~
~~Побочные эффекты в классе Dollar?~~
Округление денежных величин?
~~equals()~~
hashCode()
Равенство значению null
Равенство объектов

~~5 CHF * 2 = 1 °CHF~~
~~Дублирование Dollar/Franc~~
~~Общие операции equals()~~
~~Общие операции times()~~
~~Сравнение франков (Franc) и долларов (Dollar)~~
~~Валюта?~~
~~Нужен ли тест testFrancMultiplication()?~~

Тест testDifferentClassEquality() служит доказательством того, что, сравнивая объекты, мы сравниваем различные валюты, но не различные классы. Этот тест имеет смысл только в случае, если в программе существует несколько различных классов. Однако мы уже избавились от класса Dollar и намерены точно так же избавиться от класса Franc. Иными словами, в нашем распоряжении останется только один денежный класс: Money. С учетом наших намерений, тест testDifferentClassEquality() оказывается для нас излишней обузой. Мы удалим его, а затем избавимся от класса Franc.

Обратите также внимание, что в программе присутствуют отдельные тесты для проверки умножения франков на доллары. Если заглянуть в код, можно увидеть, что на текущий момент логика метода, реализующего умножение, не зависит от типа валюты (зависимость была бы только в случае, если бы мы использовали два различных класса). То есть мы можем удалить функцию testFrancMultiplication(), не опасаясь, что потеряем уверенность в правильности работы системы.

Итак, в нашем распоряжении единый денежный класс, и мы готовы приступить к реализации сложения.

Но сначала подведем итоги. В этой главе мы

• закончили потрошить производные классы и избавились от них;

• удалили тесты, которые имели смысл только при использовании старой структуры кода, но оказались избыточными в коде с новой структурой.

12. Сложение, наконец-то

$5 + 1 °CHF = $10, если курс обмена 2:1

Наступил новый день, и я заметил, что список задач переполнен вычеркнутыми пунктами. Лучше всего переписать оставшиеся не зачеркнутыми пункты в новый свежий список. (Я люблю физически копировать пункты из старого списка в новый список. Если в старом списке много мелких недоделанных задач, вместо того, чтобы копировать их в новый список, я просто добавляю в программу соответствующий код. В результате из-за моей лени куча мелочей, которая могла бы расти со временем, просто исчезает. Используйте свои слабости.)

$5 + 1 °CHF = $10, если курс обмена 2:1
$5 + $5 = $10

Пока что я не представляю себе, как можно реализовать смешанное сложение долларов и франков, поэтому предлагаю начать с более простой задачи: $5 + $5 = $10.

```
public void testSimpleAddition() {
Money sum = Money.dollar(5). plus(Money.dollar(5));
assertEquals(Money.dollar(10), sum);
}
```

Мы могли бы подделать реализацию, просто вернув значение Money.dollar(10), однако в данном случае реализация кажется очевидной. Давайте попробуем:

Money

```
Money plus(Money addend) {
return new Money(amount + addend.amount, currency);
}
```

(Далее я буду ускорять процесс разработки, чтобы сэкономить бумагу и сохранить ваш интерес. Там, где дизайн не очевиден, я буду подделывать реализацию и выполнять рефакторинг. Я надеюсь, что благодаря этому вы

увидите, каким образом в TDD выполняется контроль над величиной шагов.)

Сказав, что планирую увеличить скорость, я немедленно замедляю процесс разработки. Однако я не планирую замедлять процесс написания кода, который обеспечивает успешное тестирование. Я планирую замедлить процесс написания самих тестов. Некоторые ситуации и некоторые тесты требуют тщательного обдумывания. Каким образом мы планируем представить арифметику со смешанными валютами? Это как раз тот случай, когда требуется тщательное обдумывание.

Наиболее важное и сложное ограничение, с которым нам приходится иметь дело, заключается в том, что мы не хотим, чтобы код нашей системы знал о существовании каких-либо валют. Нам хотелось бы, чтобы система имела дело с деньгами и не зависела от того, в какой валюте они представлены. Возможная стратегия состоит в том, чтобы немедленно преобразовывать любые денежные значения в некоторую единую валюту (попробуйте угадать, какая валюта является самой любимой у американских программистов). Однако подобное решение не позволит нам с легкостью варьировать соотношения (курсы обмена) между различными валютами.

Вместо этого мы хотели бы найти решение, которое позволило бы нам в удобной форме реализовать механизм обменных курсов и при этом обеспечить запись арифметических выражений в форме, близкой к стандартной арифметической записи.

Решение основано на объектах. Если имеющийся объект ведет себя не так, как нам хотелось бы, мы создаем еще один объект, обладающий точно таким же внешним протоколом, но отличающейся внутренней реализацией. Этот шаблон называется «Самозванец» (Imposter).

Возможно, многим это покажется хиромантией. Каким образом в данной ситуации можно использовать шаблон «Самозванец»? Однако я не собираюсь шутить над вами – не существует формулы, позволяющей генерировать гениальные дизайнерские решения. Решение проблемы было придумано Уордом Каннигемом десятилетие назад. Я еще не встречал человека, который независимо от Уорда придумал бы нечто подобное. К сожалению, методика TDD не гарантирует генерацию гениальных идей. Вместе с тем благодаря TDD вы имеете тесты, формирующие вашу уверенность в коде, а также тщательно вылизанный код, – все это является хорошей почвой для возникновения идеи и ее воплощения в реальность.

Итак, что же является решением в нашем случае? Предлагается создать объект, который ведет себя как объект Money, однако соответствует

сумме двух объектов Money. Чтобы объяснить эту идею, я пробовал использовать несколько разных метафор. Например, можно рассматривать сумму различных денежных величин как *бумажник*. В один бумажник можно положить несколько банкнот разных валют и разных достоинств.

Еще одна метафора: *выражение*. Имеется в виду математическое выражение, например: (2 + 3) * 5. В нашем случае мы имеем дело с денежными величинами, поэтому выражение может быть таким: ($2 + 3 CHF) * 5. Класс Money – это атомарная форма выражения. В результате выполнения любых операций над денежными величинами получается объект класса Expression. Одним из таких объектов может быть объект Sum[7]. После того как операция (например, сложение нескольких значений в разных валютах) выполнена, полученный объект Expression можно привести к некоторой заданной валюте. Преобразование к некоторой валюте осуществляется на основании набора курсов обмена.

Как выразить эту метафору в виде набора тестов? Прежде всего, мы знаем, к чему мы должны прийти:

```
public void testSimpleAddition() {
…
assertEquals(Money.dollar(10), reduced);
}
```

Переменная reduced – это объект класса Expression, который создан путем применения обменных курсов в отношении объекта Expression, полученного в результате выполнения математической операции. Кто в реальном мире отвечает за применение обменных курсов? *Банк*. Стало быть, было бы неплохо, если бы мы могли написать

```
public void testSimpleAddition() {
…
Money reduced = bank.reduce(sum, "USD");
assertEquals(Money.dollar(10), reduced);
}
```

(Плохо, когда в одной программе смешиваются две разные метафоры: *банк* и *математическое выражение*. Однако сейчас предлагаю не заострять на этом внимания. Сначала воплотим в жизнь то, что запланировали, а затем посмотрим, можно ли улучшить нашу систему с литературно-художественной точки зрения.)

Обратите внимание на важное дизайнерское решение: метод reduce() принадлежит объекту bank. С такой же легкостью мы могли бы написать

…educed = sum.reduce(«USD», bank).

Почему ответственным за выполнение операции reduce() сделан именно объект bank? На самом деле ответ следующий: «Это первое, что пришло мне в голову», однако такой ответ нельзя считать удовлетворительным. Почему мне в голову пришло сделать ответственным за выполнение операции reduce() именно объект класса Bank, а не объект класса Expression? Вот что мне известно на текущий момент:

• Объекты класса Expression, по всей видимости, лежат в самом сердце того, что мы делаем. Я стараюсь делать объекты, являющиеся сердцем системы, как можно менее зависимыми от всего остального мира. Благодаря этому они остаются гибкими в течение длительного времени («гибкие» в данном случае означает «простые для понимания, тестирования и повторного использования»).

• Я могу предположить, что класс Expression будет нести ответственность за множество операций. Значит, мы должны по возможности освободить этот класс от лишней ответственности и переложить часть ответственности на другие классы там, где это допустимо. В противном случае класс Expression разрастется до неконтролируемых размеров.

Конечно, это всего лишь догадки – этого не достаточно, чтобы принимать какие-либо окончательные решения, однако этого вполне достаточно, чтобы я начал двигаться в избранном направлении. Безусловно, если выяснится, что наша система вполне может обойтись без класса Bank, я переложу ответственность за выполнение метода reduce() на класс Expression. Если мы используем объект bank, значит, его необходимо создать:

```
public void testSimpleAddition() {
…
Bank bank = new Bank();
Money reduced = bank.reduce(sum, "USD");
assertEquals(Money.dollar(10), reduced);
}
```

Сумма двух объектов Money – это объект класса Expression:

```
public void testSimpleAddition() {
…
Expression sum = five.plus(five);
Bank bank = new Bank();
Money reduced = bank.reduce(sum, "USD");
assertEquals(Money.dollar(10), reduced);
}
```

Наконец, операция, в которой мы абсолютно уверены, – создание пяти долларов:

```
public void testSimpleAddition() {
Money five = Money.dollar(5);
Expression sum = five.plus(five);
Bank bank = new Bank();
Money reduced = bank.reduce(sum, "USD");
assertEquals(Money.dollar(10), reduced);
}
```

Что надо сделать, чтобы данный код откомпилировался? Для начала создадим интерфейс Expression (мы могли бы создать класс, однако интерфейс обладает существенно меньшим весом):

```
Expression
interface Expression
```

Метод Money.plus() должен возвращать значение типа Expression:

Money

```
Expression plus(Money addend) {
return new Money(amount + addend.amount, currency):
}
```

Это означает, что класс Money должен реализовать интерфейс Expression (это очень просто, так как в этом интерфейсе пока что нет ни одной операции):

Money
```
class Money implements Expression
```

Кроме того, нам потребуется пустой класс Bank:

Bank
```
class Bank
```

Добавим в этот класс заглушку для метода reduce():

Bank
```
Money reduce(Expression source, String to) {
return null;
}
```

Теперь код компилируется и выдает нам красную полоску. Ура! У нас прогресс! Теперь можем легко подделать реализацию:

Bank
```
Money reduce(Expression source, String to) {
return Money.dollar(10);
}
```

Зеленая полоса! Теперь мы готовы выполнить рефакторинг. Но сначала подведем итоги главы. В этой главе мы

• вместо большого теста реализовали меньший тест, чтобы добиться быстрого прогресса (вместо операции $5 + 1 °CHF ограничились более простой операцией $5 + $5);

• основательно обдумали возможные метафоры для нашего предполагаемого дизайна;

• переписали первоначальный тест в свете новой метафоры;

• как можно быстрее добились компиляции теста;

• добились успешного выполнения теста;

• с трепетом посмотрели вперед, оценив объем рефакторинга, который необходим, чтобы сделать реализацию реальной.

7 В переводе на русский язык *sum* – это сумма. – *Примеч. пер.*

13. Делаем реализацию реальной

$5 + 1 °CHF = $10, если курс обмена 2:1
$5 + $5 = $10

Мы не можем вычеркнуть пункт $5 + $5, пока не удалим из кода все повторяющиеся фрагменты. Внимательно рассмотрим код. В нем нет повторяющегося кода, но есть повторяющиеся данные – $10 в «поддельной» реализации:

```
Bank
Money reduce(Expression source, String to) {
return Money.dollar(10);
}
```

Это выражение по своей сути дублирует выражение $5 + $5 в коде теста:

```
public void testSimpleAddition() {
Money five = Money.dollar(5);
Expression sum = five.plus(five);
Bank bank = new Bank();
Money reduced = bank.reduce(sum, "USD");
assertEquals(Money.dollar(10), reduced);
}
```

Раньше, если у нас имелась «поддельная» реализация, для нас было очевидным, как можно вернуться назад и сформировать реальную реализацию. Для этого достаточно было заменить константы переменными. Однако в данном случае пока не понимаю, как вернуться назад. Поэтому, несмотря на некоторый риск, я решаю двигаться вперед:

$5 + 1 °CHF = $10, если курс обмена 2:1
$5 + $5 = $10
Операция $5 + $5 возвращает объект Money

Прежде всего, метод Money.plus() должен возвращать не просто объект

Money, а реальное выражение (Expression), то есть сумму (Sum). (Возможно, в будущем мы оптимизируем специальный случай сложения двух одинаковых валют, однако это произойдет позже.)

Итак, в результате сложения двух объектов Money должен получиться объект класса Sum:

```
public void testPlusReturnsSum() {
Money five = Money.dollar(5);
Expression result = five.plus(five);
Sum sum = (Sum) result;
assertEquals(five, sum.augend);
assertEquals(five, sum.addend);
}
```

(Вы когда-нибудь слышали, что в английском языке первое слагаемое обозначается термином *augend*, а второе слагаемое – термином *addend*? Об этом не слышал даже автор до тех пор, пока не приступил к написанию данной книги.)

Только что написанный тест, скорее всего, проживет недолго. Дело в том, что он сильно связан с конкретной реализацией разрабатываемой нами операции и мало связан с видимым внешним поведением этой операции. Однако, заставив его работать, мы окажемся на шаг ближе к поставленной цели. Чтобы скомпилировать тест, нам потребуется класс Sum с двумя полями: augend и addend:

Sum

```
class Sum {
Money augend;
Money addend;
}
```

В результате получаем исключение преобразования классов (ClassCastException) – метод Money.plus() возвращает объект Money, но не объект Sum:

Money

```
Expression plus(Money addend) {
return new Sum(this, addend);
```

```
}
```

Класс Sum должен иметь конструктор:

Sum
```
Sum(Money augend, Money addend) {
}
```

Кроме того, класс Sum должен поддерживать интерфейс Expression:

Sum
```
class Sum implements Expression
```

Наша система компилируется, однако тесты терпят неудачу – это из-за того, что конструктор класса Sum не присваивает значений полям (мы могли бы создать «поддельную» реализацию, инициализировав поля константами, однако я обещал двигаться быстрее):

Sum
```
Sum(Money augend, Money addend) {
this.augend = augend;
this.addend = addend;
}
```

Теперь в метод Bank.reduce() передается объект класса Sum. Если суммируются две одинаковые валюты и целевая валюта совпадает с валютой обоих слагаемых, значит, результатом будет объект класса Money, чье значение будет равно сумме значений двух слагаемых:

```
public void testReduceSum() {
Expression sum = new Sum(Money.dollar(3), Money.dollar(4));
Bank bank = new Bank();
Money result = bank.reduce(sum, "USD");
assertEquals(Money.dollar(7), result);
}
```

Я тщательно выбираю значения параметров так, чтобы нарушить работу существующего теста. Когда мы приводим (метод reduce()) объект класса Sum к некоторой валюте, в результате (с учетом упомянутых

упрощенных условий) должен получиться объект класса Money, чье значение (amount) совпадает с суммой значений двух объектов Money, переданных конструктору объекта Sum, а валюта (currency) совпадает с валютой обоих этих объектов:

Bank
```
Money reduce(Expression source, String to) {
Sum sum = (Sum) source;
int amount = sum.augend.amount + sum.addend.amount;
return new Money(amount, to);
}
```

Код выглядит уродливо по двум причинам:

• мы выполняем приведение к типу Sum, в то время как код должен работать с любым объектом типа Expression;

• мы используем общедоступные поля и два уровня ссылок на поля объектов.

Это достаточно легко исправить. Вначале переместим тело метода в класс Sum и благодаря этому избавимся от лишнего уровня ссылок:

Bank
```
Money reduce(Expression source, String to) {
Sum sum = (Sum) source;
return sum.reduce(to);
}
```

Sum
```
public Money reduce(String to) {
int amount = augend.amount + addend.amount;
return new Money(amount, to);
}
```

На секундочку заглянем в будущее. Приведение (reduce) суммы к некоторой валюте не может быть выполнено, если объект Sum не знает об обменном курсе. Однако обменный курс хранится в классе Bank, значит, скорее всего, в будущем нам потребуется передавать в метод Sum.reduce() еще один параметр типа Bank. Однако сейчас наш код не требует этого. Поэтому мы не добавляем никаких лишних параметров, чтобы лишний раз в них не путаться. (Что касается меня, то искушение было столь велико, что

я все-таки добавил этот параметр, когда в первый раз писал данный код, – мне очень, очень стыдно.)

$5 + 1 °CHF = $10, если курс обмена 2:1
$5 + $5 = $10
Операция $5 + $5 возвращает объект Money
Bank.reduce(Money)

Так, а что же происходит в случае, если аргументом метода Bank.reduce() является объект Money?

Давайте напишем тест, слава богу, перед нами зеленая полоса и мы не видим каких-либо других очевидных способов модификации кода:

```
public void testReduceMoney() {
Bank bank = new Bank();
Money result = bank.reduce(Money.dollar(1), "USD");
assertEquals(Money.dollar(1), result);
}
```

Bank

```
Money reduce(Expression source, String to) {
if (source instanceof Money) return (Money) source;
Sum sum= (Sum) source;
return sum.reduce(to);
}
```

Какой кошмар! Отвратительно! Тем не менее мы получили зеленую полоску и можем приступать к рефакторингу. Прежде всего, вместо прямой проверки класса всегда следует использовать полиморфизм. Класс Sum реализует метод reduce(String), и, если этот метод добавить в класс Money, мы сможем включить reduce(String) в состав интерфейса Expression.

Bank

```
Money reduce(Expression source, String to) {
if (source instanceof Money)
return (Money) source.reduce(to);
Sum sum = (Sum) source;
return sum.reduce(to);
}
```

Money
```
public Money reduce(String to) {
return this;
}
```

Включаем метод reduce(String) в состав интерфейса Expression:

Expression
```
Money reduce(String to);
```

Теперь можно избавиться от этих уродливых операций приведения типа и проверок классов:

Bank
```
Money reduce(Expression source, String to) {
return source.reduce(to);
}
```

Я не вполне доволен ситуацией, когда в интерфейсе Expression и классе Bank присутствуют методы с одинаковыми именами, но с разным набором параметров. Я так и не смог найти приемлемого решения этой проблемы в Java. В языках, где поддерживаются ключевые параметры, разница между методами Bank.reduce(Expression, String) и Expression.reduce(String) делается очевидной благодаря синтаксису языка. Однако в языках, в которых различие параметров определяется различием их позиций в списке параметров, разница между двумя подобными методами становится менее очевидной.

$5 + 1 °CHF = $10, если курс обмена 2:1
$5 + $5 = $10
Операция $5 + $5 возвращает объект Money
~~Bank.reduce(Money)~~
Приведение объекта Money с одновременной конверсией валют
Reduce(Bank,String)

Теперь можно приступить к задаче реального обмена одной валюты на другую.

В данной главе мы

• не отметили тест как завершенный, так как не избавились от дублирования;

• чтобы прояснить реализацию, решили двигаться вперед вместо того, чтобы двигаться назад;

• написали тест, чтобы форсировать создание объекта, который, как нам кажется, потребуется в будущем (объект класса Sum);

• ускорили процесс реализации (конструктор класса Sum);

• реализовали код с приведением типов в одном месте, добились успешного выполнения тестов, а затем переместили код туда, где он должен находиться;

• использовали полиморфизм, чтобы избавиться от явной проверки типа (класса).

14. Обмен валюты

$5 + 1 °CHF = $10, если курс обмена 2:1
$5 + $5 = $10
Операция $5 + $5 возвращает объект Money
~~Bank.reduce(Money)~~
Приведение объекта Money с одновременной конверсией валют
Reduce(Bank,String)

Изменения, перемены, обмены – их объятия заслуживают внимания (особенно если у вас есть книга с фразой в заголовке «в объятиях изменений» (embrace change))[8]. Впрочем, нас заботит простейшая форма обмена – у нас есть два франка и мы хотим получить один доллар. Это звучит как готовый тест:

```
public void testReduceMoneyDifferentCurrency() {
Bank bank = new Bank();
bank.addRate("CHF", "USD", 2);
Money result = bank.reduce(Money.franc(2), "USD");
assertEquals(Money.dollar(1), result);
}
```

Когда я конвертирую франки в доллары, я просто делю значение на два (мы по-прежнему игнорируем все эти неприятные проблемы, связанные с дробными числами). Чтобы сделать полоску зеленой, мы добавляем в код еще одну уродливую конструкцию:

Money

```
public Money reduce(String to) {
int rate = (currency.equals("CHF") && to.equals("USD"))
? 2
: 1;
return new Money(amount / rate, to);
}
```

Получается, что класс Money знает о курсе обмена. Это неправильно. Единственным местом, в котором выполняются любые операции,

связанные с курсом обмена, должен быть класс Bank. Мы должны передать параметр типа Bank в метод Expression.reduce(). (Вот видите? Мы так и думали, что нам это потребуется. И мы оказались правы.) Вначале меняем вызывающий код:

Bank
```
Money reduce(Expression source, String to) {
return source.reduce(this, to);
}
```

Затем меняем код реализаций:

Expression
```
Money reduce(Bank bank, String to);
```

Sum
```
public Money reduce(Bank bank, String to) {
int amount = augend.amount + addend.amount;
return new Money(amount, to);
}
```

Money
```
public Money reduce(Bank bank, String to) {
int rate = (currency.equals("CHF") && to.equals("USD"))
? 2
: 1;
return new Money(amount / rate, to);
}
```

Методы должны быть общедоступными (public), так как все методы интерфейсов должны быть общедоступными (я надеюсь, можно не объяснять, почему).

Теперь мы можем вычислить курс обмена внутри класса Bank:

Bank
```
int rate(String from, String to) {
return (from.equals("CHF") && to.equals("USD"))
? 2
: 1;
```

```
}
```

И обратиться к объекту bank с просьбой предоставить значение курса обмена:

Money

```
public Money reduce(Bank bank, String to) {
int rate = bank.rate(currency, to);
return new Money(amount / rate, to);
}
```

Эта надоедливая цифра 2 снова отсвечивает как в разрабатываемом коде, так и в теле теста. Чтобы избавиться от нее, мы должны создать таблицу обменных курсов в классе Bank и при необходимости обращаться к этой таблице для получения значения обменного курса. Для этой цели мы могли бы воспользоваться хеш-таблицей, которая ставит в соответствие паре валют соответствующий обменный курс. Можем ли мы в качестве ключа использовать двухэлементный массив, содержащий в себе две валюты? Проверяет ли метод Array.equals() эквивалентность элементов массива?

```
public void testArrayEquals() {
assertEquals(new Object[] {"abc"}, new Object[] {"abc"});
}
```

Нет. Тест провалился. Придется создавать специальный объект, который будет использоваться в качестве ключа хеш-таблицы:

Pair

```
private class Pair {
private String from;
private String to;

Pair(String from, String to) {
this.from = from;
this.to = to;
}
}
```

Мы планируем использовать объекты Pair в качестве ключей, поэтому нам необходимо реализовать методы equals() и hashCode(). Я не собираюсь писать для этого тесты, так как мы разрабатываем код в контексте рефакторинга. Дело в том, что от работоспособности этого кода жестко зависит успешное выполнение существующих тестов. Если код работает неправильно, существующие тесты потерпят неудачу. Однако если бы я программировал в паре с кем-то, кто плохо представлял бы себе направление дальнейшего движения, или если бы логика кода была более сложной, я несомненно приступил бы к разработке специальных тестов.

Pair

```
public boolean equals(Object object) {
Pair pair = (Pair) object;
return from.equals(pair.from) && to.equals(pair.to);
}
public int hashCode() {
return 0;
}
```

0 – ужасное хеш-значение, однако такой метод хеширования легко реализовать, стало быть, мы быстрее получим работающий код. Поиск валюты будет осуществляться простым линейным перебором. Позже, когда у нас будет множество валют, мы сможем тщательнее проработать этот вопрос, использовав реальные данные.

Теперь нам нужно место, в котором мы могли бы хранить значения обменных курсов:

Bank

```
private Hashtable rates= new Hashtable();
```

Нам также потребуется метод добавления нового курса обмена:

Bank

```
void addRate(String from, String to, int rate) {
rates.put(new Pair(from, to), new Integer(rate));
}
```

И метод, возвращающий обменный курс:

```
Bank
int rate(String from, String to) {
Integer rate = (Integer) rates.get(new Pair(from, to));
return rate.intValue();
}
```

Подождите-ка минутку! Перед нами красная полоса. Что случилось? Взглянув на код, мы видим, что проблема в неправильном значении курса при обмене доллара на доллары. Мы ожидаем, что при обмене USD на USD курс обмена будет равен 1, однако на текущий момент это не так. Поскольку эта ситуация стала для нас сюрпризом, оформим ее в виде дополнительного теста:

```
public void testIdentityRate() {
assertEquals(1, new Bank(). rate("USD", "USD"));
}
```

Теперь у нас три ошибки, однако все они могут быть исправлены при помощи одного небольшого изменения:

```
Bank
int rate(String from, String to) {
if (from.equals(to)) return 1;
Integer rate = (Integer) rates.get(new Pair(from, to));
return rate.intValue();
}
```

Зеленая полоска!

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 + $5 = $10~~
Операция $5 + $5 возвращает объект Money
~~Bank.reduce(Money)~~
~~Приведение объекта Money с одновременной конверсией валют~~
~~Reduce(Bank,String)~~

Далее мы переходим к нашему последнему, самому большому тесту, $5 + 1 °CHF. В данной главе мы применили несколько важных технологий:

• добавили параметр, который может нам понадобиться;

• удалили дублирование между кодом и тестами;

• написали тест (testArrayEquals), чтобы проверить порядок функционирования встроенной операции Java;

• создали вспомогательный закрытый (private) класс, не обладающий собственными тестами;

• допустили ошибку при рефакторинге и написали еще один тест, чтобы изолировать проблему.

8 Используя игру слов (английское change означает как «изменение», так и «обмен»), автор намекает на свою знаменитую книгу-бестселлер *Extreme Programming Explained: Embrace Change*. Русский перевод: Бек К. Экстремальное программирование. СПб.: Питер, 2002. 224 с. – *Примеч. ред*.

15. Смешение валют

$5 + 1 °CHF = $10, если курс обмена 2:1
~~$5 + $5 = $10~~
Операция $5 + $5 возвращает объект Money
~~Bank.reduce(Money)~~
~~Приведение объекта Money с одновременной конверсией валют~~
~~Reduce(Bank,String)~~

Теперь мы готовы написать тест, с которого все началось, – $5 + 1 °CHF:

```
public void testMixedAddition() {
Expression fiveBucks = Money.dollar(5);
Expression tenFrancs = Money.franc(10);
Bank bank = new Bank();
bank.addRate("CHF", "USD", 2);
Money result = bank.reduce(fiveBucks.plus(tenFrancs), "USD");
assertEquals(Money.dollar(10), result);
}
```

Именно такой код нам хотелось бы написать. К сожалению, мы сразу же получаем кучу ошибок компиляции. Обобщая код в процессе перехода от Money к Expression, мы оставили много висящих хвостов, на которые я, конечно же, обратил внимание, но решил вас не беспокоить. Теперь настало время заняться устранением дефектов.

Мы не сможем достаточно быстро обеспечить компиляцию предыдущего теста. Как только мы внесем в код первое изменение, нам потребуется внести в код еще изменения, и так далее. Теперь мы можем двигаться дальше одним из двух путей. Мы можем заставить тест работать быстро, для этого надо написать более специфичный тест и затем выполнить обобщение. Второй путь: довериться компилятору и с его помощью найти все ошибки. Давайте попробуем действовать медленно (на практике я внес бы в код все необходимые изменения за один раз).

```
public void testMixedAddition() {
Money fiveBucks = Money.dollar(5);
```

```
Money tenFrancs = Money.franc(10);
Bank bank = new Bank();
bank.addRate("CHF", "USD", 2);
Money result = bank.reduce(fiveBucks.plus(tenFrancs), "USD");
assertEquals(Money.dollar(10), result);
}
```

Тест терпит неудачу. Мы получаем 15 USD вместо 10 USD. Дело в том, что метод Sum.reduce() не выполняет приведение аргументов:

Sum

```
public Money reduce(Bank bank, String to) {
int amount = augend.amount + addend.amount;
return new Money(amount, to);
}
```

Если выполнить приведение обоих аргументов, тест должен сработать:

Sum

```
public Money reduce(Bank bank, String to) {
int amount= augend.reduce(bank, to). amount
+ addend.reduce(bank, to). amount;
return new Money(amount, to);
}
```

И действительно, тест срабатывает. Теперь мы можем заменить тип Money на тип Expression. Чтобы избежать взаимовлияний, мы начнем издалека и будем двигаться в направлении тестирующего кода. Итак, поля augend и addend теперь могут иметь тип Expression:

Sum

```
Expression augend;
Expression addend;
```

Аргументы конструктора тоже могут иметь тип Expression:

Sum

```
Sum(Expression augend, Expression addend) {
this.augend = augend;
```

```
this.addend = addend;
}
```

(Класс Sum начинает напоминать мне шаблон «Компоновщик» (Composite), однако еще не настолько, чтобы я захотел обобщить его.) С классом Sum, пожалуй, закончили, а что насчет Money?

Аргумент метода plus() может иметь тип Expression:

Money
```
Expression plus(Expression addend) {
return new Sum(this, addend);
}
```

Метод times() может возвращать значение типа Expression:

Money
```
Expression times(int multiplier) {
return new Money(amount * multiplier, currency);
}
```

Это означает, что операции plus() и times() должны входить в состав интерфейса Expression. С классом Money закончили. Теперь можно изменить аргументы метода plus() в реализации теста:

```
public void testMixedAddition() {
Money fiveBucks = Money.dollar(5);
Expression tenFrancs = Money.franc(10);
Bank bank = new Bank();
bank.addRate("CHF", "USD", 2);
Money result = bank.reduce(fiveBucks.plus(tenFrancs), "USD");
assertEquals(Money.dollar(10), result);
}
```

Объект tenFrancs теперь принадлежит типу Expression, а это значит, что мы должны внести в код некоторые изменения. К счастью, компилятор подсказывает нам, что именно мы должны сделать. Прежде всего вносим изменение:

```
public void testMixedAddition() {
```

```
Expression fiveBucks = Money.dollar(5);
Expression tenFrancs = Money.franc(10);
Bank bank = new Bank();
bank.addRate("CHF", "USD", 2);
Money result = bank.reduce(fiveBucks.plus(tenFrancs), "USD");
assertEquals(Money.dollar(10), result);
}
```

Компилятор вежливо сообщает, что plus() не является методом интерфейса Expression. Добавим этот метод в интерфейс:

```
Expression
Expression plus(Expression addend);
```

Теперь мы должны добавить этот метод в классы Money и Sum. Money? Да, этот метод должен быть открытым (public) в классе Money:

```
Money
public Expression plus(Expression addend) {
return new Sum(this, addend);
}
```

Что касается класса Sum, просто добавим заглушку и отметим необходимость реализации этого метода в списке задач:

```
Sum
public Expression plus(Expression addend) {
return null;
}
```

~~$5 + 1 °CHF = $10, если курс обмена 2:1~~
~~$5 + $5 = $10~~
Операция $5 + $5 возвращает объект Money
~~Bank.reduce(Money)~~
~~Приведение объекта Money с одновременной конверсией валют~~
~~Reduce(Bank,String)~~
Sum.plus
Expression.times

Теперь программа компилируется и все тесты выполняются успешно.

Мы готовы завершить обобщение класса Money до Expression, но прежде, как всегда, подведем краткий итог. В этой главе мы

- за один шаг написали необходимый тест и затем модифицировали его, чтобы добиться успешного его выполнения;
- выполнили обобщение (использовали более абстрактное объявление);
- воспользовались подсказками компилятора, чтобы внести изменения (Expression fiveBucks), которые привели к необходимости дополнительных изменений (добавление метода plus() в интерфейс Expression и т. п.).

16. Абстракция, наконец-то!

~~$5 + 1 °CHF = $10, если курс обмена 2:1~~
~~$5 + $5 = $10~~
Операция $5 + $5 возвращает объект Money
~~Bank.reduce(Money)~~
~~Приведение объекта Money с одновременной конверсией валют~~
~~Reduce(Bank,String)~~
Sum.plus
Expression.times

Чтобы завершить добавление метода Expression.plus, мы должны реализовать метод Sum.plus(). Затем нам останется добавить метод Expression.times(), и мы сможем считать пример завершенным. Вот тест для метода Sum.plus():

```
public void testSumPlusMoney() {
Expression fiveBucks = Money.dollar(5);
Expression tenFrancs = Money.franc(10);
Bank bank = new Bank();
bank.addRate("CHF", "USD", 2);
Expression sum = new Sum(fiveBucks, tenFrancs). plus(fiveBucks);
Money result = bank.reduce(sum, "USD");
assertEquals(Money.dollar(15), result);
}
```

Мы могли бы создать объект Sum путем сложения fiveBucks и tenFrancs, однако приведенный код, который явно создает объект Sum, выглядит более понятным. Ведь мы пишем эти тесты не только ради удовольствия от программирования, но также для того, чтобы будущие поколения программистов могли оценить нашу гениальность. Однако они не смогут сделать этого, если код будет непонятным. Поэтому, разрабатывая любой код, думайте о тех, кто будет его читать.

В данном случае код теста длиннее, чем сам тестируемый код. Код точно такой же, как код в классе Money (кажется, я уже предвижу необходимость создания абстрактного класса):

Sum

```
public Expression plus(Expression addend) {
return new Sum(this, addend);
}
```

~~$5 + 1 °CHF = $10, если курс обмена 2:1~~
~~$5 + $5 = $10~~
Операция $5 + $5 возвращает объект Money
~~Bank.reduce(Money)~~
~~Приведение объекта Money с одновременной конверсией валют~~
~~Reduce(Bank,String)~~
~~Sum.plus~~
Expression.times

При использовании TDD вы часто будете сталкиваться с тем, что количество строк в тестовом коде будет приблизительно таким же, как и количество строк в тестируемом коде. Чтобы методика TDD обладала экономическим смыслом, вы должны либо записывать в два раза большее количество строк кода, чем обычно, либо реализовывать ту же самую функциональность при помощи количества строк, в два раза меньшего, чем обычно. Эти показатели рекомендуется оценить самостоятельно на примере собственной практики. Однако, выполняя оценку, вы должны принять во внимание время, которое тратится на отладку, интеграцию и объяснение внутреннего устройства другим людям.

~~$5 + 1 °CHF = $10, если курс обмена 2:1~~
~~$5 + $5 = $10~~
Операция $5 + $5 возвращает объект Money
~~Bank.reduce(Money)~~
~~Приведение объекта Money с одновременной конверсией валют~~
~~Reduce(Bank,String)~~
~~Sum.plus~~
~~Expression.times~~

Если мы получили работающий метод Sum.times(), значит, объявление Expression.times() не составит для нас труда. Вот соответствующий тест:

```
public void testSumTimes() {
Expression fiveBucks = Money.dollar(5);
Expression tenFrancs = Money.franc(10);
```

```
Bank bank = new Bank();
bank.addRate("CHF", "USD", 2);
Expression sum = new Sum(fiveBucks, tenFrancs). times(2);
Money result = bank.reduce(sum, "USD");
assertEquals(Money.dollar(20), result);
}
```

И снова тест получился длиннее тестируемого кода. (Те, кто достаточно много работал с JUnit, должно быть уже догадались, как решить эту проблему. Остальным я рекомендую прочитать раздел «*Fixture (Фикстура)*» в главе 29, посвященной шаблонам xUnit.)

Sum

```
Expression times(int multiplier) {
return new Sum(augend.times(multiplier), addend.times(multiplier));
}
```

В предыдущей главе мы изменили тип переменных augend и addend на Expression, поэтому теперь, чтобы скомпилировать код, нам необходимо добавить в интерфейс Expression метод times():

Expression

```
Expression times(int multiplier);
```

При этом нам следует изменить режим видимости методов Money.times() и Sum.times() (они должны стать общедоступными):

Sum

```
public Expression times(int multiplier) {
return new Sum(augend.times(multiplier), addend.times(multiplier));
}
```

Money

```
public Expression times(int multiplier) {
return new Money(amount * multiplier, currency);
}
```

~~$5 + 1 °CHF = $10, если курс обмена 2:1~~
~~$5 + $5 = $10~~
Операция $5 + $5 возвращает объект Money

~~Bank.reduce(Money)~~
~~Приведение объекта Money с одновременной конверсией валют~~
~~Reduce(Bank,String)~~
~~Sum.plus~~
~~Expression.times~~

Все заработало.

Осталось провести эксперимент для случая, когда в результате выполнения операции $5 + $5 получается объект Money. Вот соответствующий тест:

```
public void testPlusSameCurrencyReturnsMoney() {
Expression sum = Money.dollar(1). plus(Money.dollar(1));
assertTrue(sum instanceof Money);
}
```

Тест выглядит несколько неопрятно, так как тестирует внутреннюю реализацию, а не внешнее поведение объектов. Однако он принуждает нас внести в программу изменения, которые нам необходимы, и, в конце концов, это всего лишь эксперимент. Вот код, который мы должны модифицировать, чтобы заставить тест работать:

Money

```
public Expression plus(Expression addend) {
return new Sum(this, addend);
}
```

~~$5 + 1 °CHF = $10, если курс обмена 2:1~~
~~$5 + $5 = $10~~
Операция $5 + $5 возвращает объект Money
~~Bank.reduce(Money)~~
~~Приведение объекта Money с одновременной конверсией валют~~
~~Reduce(Bank,String)~~
~~Sum.plus~~
~~Expression.times~~

Не существует очевидного и ясного способа проверить валюту аргумента, если этот аргумент является объектом класса Money (по крайней мере, я не могу найти такого способа, однако вы можете над этим

подумать). Эксперимент окончился неудачей, мы удаляем тест (который нам все равно не нравился).

Подводим итог. Мы

• написали тест так, чтобы его смысл легко был понят другими программистами, которые в будущем будут читать разработанный нами код;

• наметили эксперимент, призванный сравнить эффективность TDD по отношению к обычному стилю программирования, используемому вами на текущий момент;

• снова столкнулись с необходимостью изменения множества объявлений в разрабатываемом коде и снова воспользовались услугами компилятора, чтобы исправить все неточности;

• попробовали провести быстрый эксперимент, однако отказались от идеи, так как она не сработала, и уничтожили соответствующий тест.

17. Ретроспектива денежного примера

Давайте еще раз окинем взглядом пример реализации мультивалютных вычислений и попробуем оценить использованный нами подход и полученные результаты. Вот несколько тезисов, которых хотелось бы коснуться:

Что дальше? Как определить дальнейшее направление разработки?

Метафора. Впечатляющий эффект, который метафора оказывает на структуру дизайна.

Использование JUnit. Как часто мы запускали тесты и как мы использовали JUnit?

Метрики кода. Численные характеристики получившегося кода.

Процесс. Мы говорим: «красный – зеленый – рефакторинг». Как много усилий прикладывается на каждом из этих этапов?

Качество тестов. Каким образом характеристики тестов TDD соотносятся с характеристиками обычных методик тестирования?

Что дальше?

Можно ли считать код завершенным? Нет. Методы Sum.plus() и Money.plus() во многом дублируют друг друга. Если мы преобразуем интерфейс Expression в класс (не совсем обычное преобразование – чаще классы становятся интерфейсами), мы получим возможность переместить общий код в единый общий метод.

На самом деле мне сложно представить себе код, который полностью завершен. Методику TDD можно использовать как способ приближения к идеалу, однако это будет не самое эффективное ее использование. Если вы имеете дело с крупной системой, тогда части системы, с которыми вы работаете каждый день, должны быть тщательно «вылизаны». Иными словами, дизайн должен быть чистым и понятным, а код должен быть хорошо протестированным. В этом случае вы изо дня в день сможете вносить в систему необходимые изменения, не теряя при этом уверенности в работоспособности кода. Однако на периферии системы, где располагаются части, к которым вы обращаетесь относительно редко, количество тестов может быть меньше, а дизайн – уродливее.

Когда я завершаю решение всех очевидных задач, я люблю запускать

инструмент проверки оформления кода (например, SmallLint для Smalltalk). Многие полученные в результате этого предложения мне и без того известны. Со многими предложениями я не согласен. Однако автоматизированный инструмент проверки кода ни о чем не забывает, поэтому иногда он обнаруживает то, что было упущено мною из виду.

Еще один полезный вопрос: «Какие дополнительные тесты необходимо написать для системы?» Иногда кажется, что некоторый тест должен потерпеть неудачу, однако, добавив его в тестовый набор, вы обнаруживаете, что он работает. В этом случае необходимо определить, почему так происходит. Иногда тест, который не должен работать, действительно не работает, и вы добавляете его в набор, как признак известного вам ограничения разрабатываемой системы или как напоминание о работе, которую необходимо выполнить позднее.

Наконец, когда список задач пуст, неплохо еще раз проверить дизайн. Удовлетворяет ли данная реализация всем предъявляемым требованиям? Существует ли дублирование, которое сложно устранить при использовании данного дизайна? (Сохранившееся дублирование – признак нереализованного дизайна.)

Метафора

Лично для меня самым большим сюрпризом в данном примере явилось впечатляющее отличие окончательного дизайна от тех разработок, с которыми мне приходилось иметь дело до написания этой книги. Я выполнял разработку аналогичного мультивалютного кода для различных программных систем, реально используемых в производстве, по меньшей мере три раза (насколько я могу припомнить). Кроме того, я использовал эту же задачу для разного рода публикаций еще раз шесть или семь. Помимо публикаций я пятнадцать раз программировал этот пример перед аудиторией на различных конференциях (программировал со сцены – звучит здорово, но выглядит менее впечатляюще). Наконец, прежде чем написать окончательный вариант первой части данной книги, я перебрал три или четыре различных направления разработки кода (я менял направление своих мыслей в соответствии с поступавшими ранними рецензиями и отзывами о написанном материале). И вот, пока я работал над текстом первой части, мне в голову пришла мысль использовать в качестве метафоры *математические выражения* (expressions). В результате дизайн стал развиваться по совершенно иному, не известному мне ранее пути.

Я никогда не думал, что метафора – это настолько мощный инструмент. Многие думают, что метафора – это всего лишь источник имен. Разве не так? Похоже, что нет.

Для представления «комбинации нескольких денежных величин, которые могут быть выражены в разных валютах», Уорд Каннингэм использовал метафору *вектора*. Имеется в виду математический вектор – набор коэффициентов, каждому из которых соответствует некоторая валюта. Лично я некоторое время использовал метафору *суммы денег* (MoneySum), затем придумал *денежный мешок* (MoneyBag) – звучит понятно и близко к реальности, – наконец, остановился на метафоре *бумажника* (Wallet). Что такое бумажник и как он функционирует, известно абсолютно всем. Все эти метафоры подразумевают, что набор денежных значений (объектов Money) является плоским. Иначе говоря, выражение 2 USD + 5 CHF + 3 USD эквивалентно выражению 5 USD + 5 CHF. Два значения в одной и той же валюте автоматически сливаются в одно.

Метафора *математического выражения* избавила меня от множества неприятных проблем, связанных со слиянием дублирующихся валют. Результирующий код получился чище, чем я когда-либо видел. Конечно же, я несколько обеспокоен производительностью кода, основанного на подобной метафоре, однако, прежде чем приступать к оптимизации, я намерен проанализировать статистику обращений к различным участкам кода.

Почему я был вынужден переписать заново то, что я уже писал до этого не меньше 20 раз? Буду ли я и дальше сталкиваться с подобными сюрпризами? Существует ли способ, который позволит мне найти правильное решение, по крайней мере в течение первых трех попыток? А может быть, этот способ позволит мне найти правильное решение с первой попытки?

Использование JUnit

Я поручил инфраструктуре JUnit вести журнал в процессе разработки мультивалютного примера. Выяснилось, что за все время я нажал клавишу Enter ровно 125 раз. Оценку интервала между запусками тестов нельзя считать достоверной, так как в ходе работы я не только программировал, но и писал текст книги. Однако когда я занимался только программированием, я запускал тесты приблизительно раз в минуту.

На рис. 17.1 представлена гистограмма интервалов между запусками

тестов. Большое количество длительных интервалов, скорее всего, обусловлено тем, что я тратил значительное время на написание текста книги.

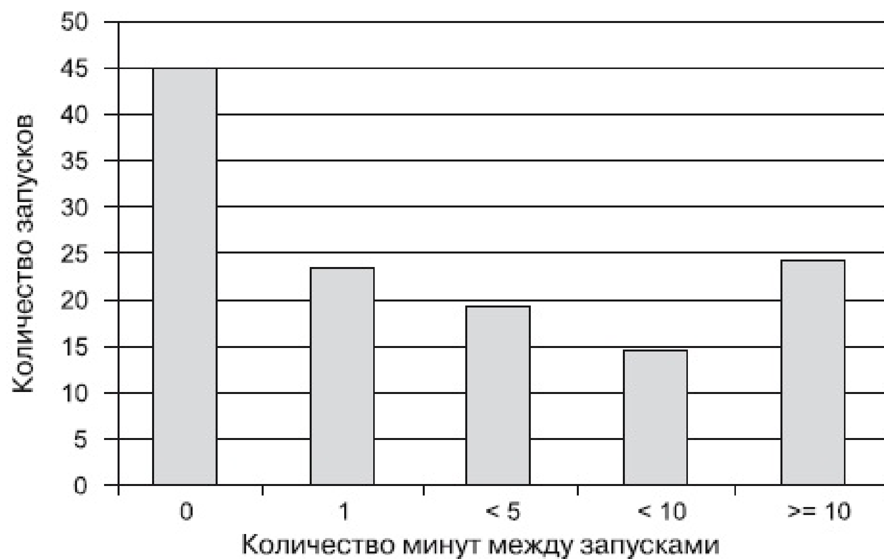

Рис. 17.1. Гистограмма интервалов времени между запусками тестов

Метрики кода

В табл. 17.1 приводятся некоторые статистические данные, характеризующие код.

Таблица 17.1. Метрики кода

	Функциональный код	Тесты
Количество классов	5	1
Количество функций (1)	22	15
Количество строк кода (2)	91	89
Цикломатическая сложность кода (3)	1,04	1
Количество строк/Количество функций	4,1 (4)	5,9 (5)

Вот некоторые примечания к данной таблице:

1. Мы не реализовали весь программный интерфейс (API) целиком, поэтому не можем достоверно оценить полное количество функций, или

количество функций на один класс, или количество строк кода на один класс. Однако соотношения этих параметров можно считать поучительными. Количество функций и количество строк в тестах приблизительно такое же, как и в функциональном коде.

2. Количество строк кода в тестах можно сократить, если извлечь из кода операции подготовки тестовых данных. Однако общее соотношение между строками функционального кода и строками тестирующего кода при этом сохранится.

3. Цикломатическая сложность (cyclomatic complexity) – это величина, характеризующая сложность обычного потока управления в программе. Цикломатическая сложность тестов равна 1, так как в тестирующем коде нет ни ветвлений, ни циклов. Цикломатическая сложность функционального кода близка к единице, так как вместо явных ветвлений для передачи управления чаще используется полиморфизм.

4. Оценка количества строк в функции дана с учетом заголовка функции и закрывающей скобки.

5. Количество строк на функцию для тестирующего кода в нашем случае больше чем могло бы быть, так как мы не выделили общий код в отдельные функции. Об этом рассказывается в главе 29, которая посвящена методам работы с xUnit.

Процесс

Цикл TDD выглядит следующим образом:

• написать тест;

• запустить все тесты и убедиться, что добавленный тест терпит неудачу;

• внести в код изменения;

• запустить тесты и убедиться, что все они выполнились успешно;

• выполнить рефакторинг, чтобы устранить дублирование.

Если исходить из того, что разработка теста – это один шаг, какое количество изменений требуется сделать, чтобы выполнить компиляцию, запуск и рефакторинг? (Под изменением я подразумеваю изменение определения метода или класса.) На рис. 17.2 показана гистограмма количества изменений для каждого из тестов «денежного» примера, над которым мы работали в первой части книги.

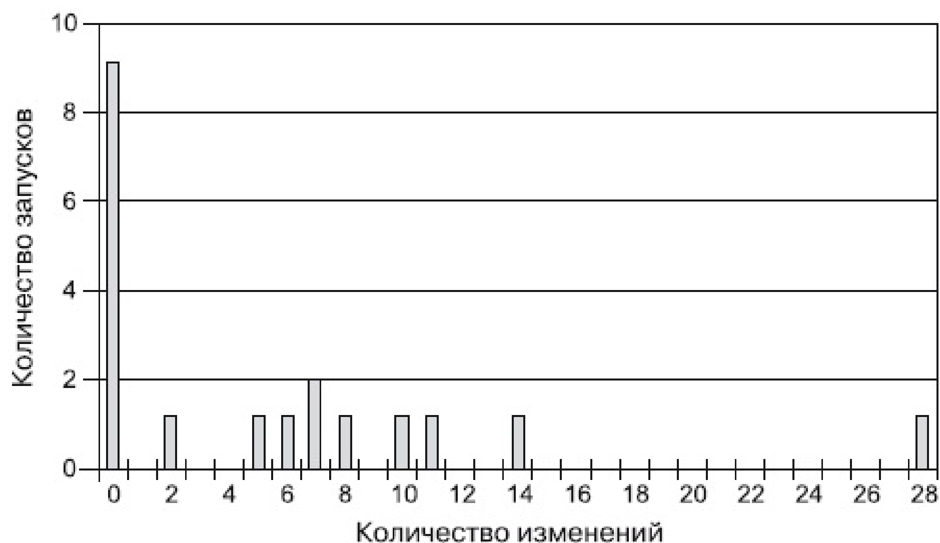

Рис. 17.2. Гистограмма количества изменений, приходящихся на каждый период рефакторинга

Я полагаю, что если бы мы собирали статистику для достаточно крупного проекта, мы обнаружили бы, что количество изменений, необходимых для компиляции и запуска кода, очень невелико (это количество можно уменьшить, если среда разработки будет понимать, что пытаются ей сказать тесты, и, например, автоматически добавлять в функциональный код необходимые заглушки). Однако количество изменений, вносимых в код во время рефакторинга, должно соответствовать (вот главный тезис) кривой распределения с эксцессом больше нормального, то есть с большим числом изменений, чем предсказывается стандартной кривой нормального распределения. Подобный профиль характерен для многих других естественных процессов, например для изменения стоимости акций на рынке ценных бумаг[9].

Качество тестов

Тесты являются неотъемлемой частью методики TDD. Они могут запускаться в любое время работы над программой, а также после того, как программа будет завершена. Однако не стоит путать их с другими важными типами тестирования:

- тестированием производительности;
- нагрузочным тестированием;

• тестированием удобства использования.

Тем не менее, если плотность вероятности дефектов в коде, разработанном с использованием TDD, невелика, роль профессионального тестирования меняется. Если обычно профессиональное тестирование используется для постоянного надзора за работой программистов, то при использовании TDD профессиональное тестирование больше напоминает вспомогательный инструмент, облегчающий коммуникацию между теми, кто знает, как должна работать система, и теми, кто создает систему.

Как можно оценить качество разработанных нами тестов? Вот два широко распространенных метода:

Охват кода (statement coverage). Для оценки качества тестов этой характеристики недостаточно, однако ее можно использовать как отправную точку. Если программист ревностно следует всем требованиям TDD, тесты должны охватывать 100 % кода. Для оценки этой характеристики можно использовать специальные программные средства. Например, программа JProbe (www.sitaka.com/software/jprobe) сообщает нам, что в нашем примере не охваченной тестами осталась всего одна строка в одном методе – Money.toString(). Напомню, что эта строка была добавлена в отладочных целях, фактически она не является функциональным кодом.

Намеренное добавление дефекта (defect insertion). Это еще один способ проверки качества тестов. Идея проста: изменить значение строки кода и убедиться, что тест перестал работать. Делать это можно вручную или при помощи специального инструмента, такого как Jester (jester.sourceforge.net). Этот инструмент сообщает нам, что в нашей программе существует всего одна строка, которую можно изменить, не нарушив работы тестов. Вот эта строка: Pair.hashCode(). Здесь мы просто подделали реализацию – вместо хеш-кода метод возвращает постоянное значение: 0. Если одно постоянное значение заменить другим, смысл программы не изменится (одна подделка ничем не лучше другой), поэтому подобную модификацию кода нельзя считать дефектом.

Флип, один из рецензентов моей книги, сообщил мне некоторые дополнительные соображения относительно охвата тестами. Абсолютный показатель охвата вычисляется следующим образом: количество тестов, предназначенных для тестирования различных аспектов программы, необходимо разделить на количество аспектов, которые нуждаются в тестировании (сложность логики программы). Существует два способа улучшить показатель охвата тестами. Во-первых, можно написать больше тестов. Отсюда разница в количестве тестов, которые пишутся

разработчиком, использующим TDD, и профессиональным тестером. (В главе 32 приводится пример задачи, для решения которой я написал 6 тестов, а человек, профессионально занимающийся тестированием, – 65 тестов.) Однако существует и другой способ улучшить охват – ограничиться фиксированным набором тестов и упростить логику программы. Подобный эффект зачастую достигается в процессе рефакторинга – условные операторы заменяются сообщениями классов или вовсе удаляются из программы. Флип выражает эту мысль так: «Вместо того чтобы увеличить количество тестов и тем самым охватить всевозможные комбинации входных данных (говоря точнее, эффективное подмножество всех комбинаций), мы оставляем количество тестов неизменным и меняем количество внутренних структурных комбинаций кода».

Последний взгляд назад

Существует три важных навыка, которые необходимо освоить тем, кто впервые изучает TDD:

- три основных подхода, которые используются, чтобы заставить тест работать: подделка реализации, триангуляция и очевидная реализация;
- устранение дублирования между функциональным кодом и тестами – важный способ формирования дизайна;
- способность контролировать расстояние между тестами: когда дорога становится скользкой, необходимо двигаться маленькими шажками; когда дальнейший путь ясен, можно увеличить скорость.

Часть II
На примере xUnit

Какой подход использовать при создании инструмента для разработки через тестирование? Естественно, разработку через тестирование.

Архитектура xUnit хорошо реализуется на языке Python, поэтому во второй части книги я перейду на использование Python. Не беспокойтесь, для тех, кто никогда раньше не имел дела с Python, я добавлю в текст необходимые пояснения. Когда вы прочитаете вторую часть, вы, во-первых, освоите базовые навыки программирования на Python, во-вторых, узнаете, как самому разработать свою собственную инфраструктуру для автоматического тестирования, и, в-третьих, ознакомитесь с более сложным примером использования методики TDD – три по цене одного!

18. Первые шаги на пути к xUnit

Разработка инструмента тестирования с использованием самого этого инструмента для тестирования многим может показаться чем-то, напоминающим хирургическую операцию на своем собственном мозге. («Только не вздумай трогать центры моторики! О! Слишком поздно! Игра окончена».) Сначала эта идея может показаться жутковатой. Однако инфраструктура тестирования обладает более сложной внутренней логикой, если сравнивать с относительно несложным денежным примером, рассмотренным в первой части книги. Часть II можно рассматривать как шаг в сторону разработки «настоящего» программного обеспечения. Кроме того, вы можете рассматривать этот материал как упражнение в самодокументируемом программировании.

Прежде всего, у нас должна быть возможность создать тест и запустить тестовый метод. Например: TestCase("testMethod"). run(). Возникает проблема: мы собираемся написать тест для программного кода, который мы будем использовать для написания тестов. Так как у нас пока еще нет даже намека на инфраструктуру тестирования, мы вынуждены проверить правильность нашего самого первого шага вручную. К счастью, мы достаточно хорошо отдохнули, а значит, вероятность того, что мы допустим ошибку, относительно невелика. Однако чтобы сделать ее еще меньше, мы планируем двигаться маленькими-маленькими шажками, тщательно проверяя все, что мы делаем. Вот список задач, который приходит на ум, когда начинаешь размышлять о разработке собственной инфраструктуры тестирования:

Вызов тестового метода
Вызов метода setUp перед обращением к методу
Вызов метода tearDown после обращения к методу
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
Отчет о результатах

Конечно же, мы по-прежнему работаем в стиле «сначала тесты». Для нашего прототеста нам потребуется небольшая программа, которая должна отображать на экране значение «истина», если произошло обращение к тестовому методу, и значение «ложь» в противном случае. Теперь

представим, что у нас есть тест, который устанавливает флаг внутри тестового метода, в этом случае мы могли бы после выполнения теста отобразить состояние флага на экране и самостоятельно убедиться в том, что флаг установлен правильно. Выполнив проверку вручную, мы сможем попробовать автоматизировать процесс.

Итак, у нас наметилась следующая стратегия. Мы создаем объект, который соответствует нашему тесту. В объекте содержится флаг. Перед выполнением тестового метода флаг должен быть установлен в состояние «ложь». Тестовый метод устанавливает флаг в состояние «истина». После выполнения тестового метода мы должны проверить состояние флага. Назовем наш тестовый класс именем WasRun[10], так как объект этого класса будет сигнализировать нам о том, был ли выполнен тестовый метод. Флаг внутри этого класса также будет называться wasRun (это несколько сбивает с толку, однако wasRun – такое подходящее имя). Собственно объект (экземпляр класса WasRun) будет называться просто test. То есть мы сможем написать инструкцию assert test.wasRun (assert – встроенная инструкция языка Python).

Язык программирования Python является интерпретируемым – команды исполняются по мере чтения их из файла с исходным кодом. Поэтому, чтобы выполнить тестирование, мы можем написать следующий короткий файл и попробовать запустить его:

```
test = WasRun(«testMethod»)
print(test.wasRun)
test.testMethod()
print(test.wasRun)
```

Мы ожидаем, что эта миниатюрная программа напечатает None до выполнения тестового метода и 1 – после. (В языке Python значение None является аналогом null или nil и наряду с числом 0 соответствует значению «ложь».) Однако программа не делает того, что мы от нее ждем. И немудрено – мы еще не определили класс WasRun (сначала тесты!).

WasRun
```
class WasRun:
pass
```

(Ключевое слово pass используется в случае, если реализация класса или метода отсутствует.) Теперь интерпретатор сообщает нам, что в классе

WasRun нет атрибута с именем wasRun. Создание атрибута происходит в момент создания объекта (экземпляра класса), то есть в процессе выполнения конструктора (для удобства конструктор любого класса называется __init__). Внутри конструктора мы присваиваем флагу wasRun значение None (ложь):

WasRun

```
class WasRun:
def __init__(self, name):
self.wasRun = None
```

Теперь программа действительно отображает на экране значение None, однако после этого интерпретатор сообщает нам, что мы должны определить в классе WasRun метод testMethod. (Было бы неплохо, если бы среда разработки автоматически реагировала на это: самостоятельно создавала бы функцию-заглушку и открывала редактор с курсором, установленным в теле этой функции. Не правда ли, это было бы просто здорово? Кстати, некоторые производители IDE уже додумались до этого.)

WasRun

```
def testMethod(self):
pass
```

Запускаем файл и видим на экране два значения: None и None[11]. Нам хотелось бы видеть None и 1. Чтобы получить желаемый результат, в теле метода testMethod присвоим флагу wasRun желаемое значение:

WasRun

```
def testMethod(self):
self.wasRun = 1
```

Запускаем программу – то, что нужно! Мы получили желаемый результат. Зеленая полоса – ур-р-ра! Нам предстоит сложный рефакторинг, однако если мы видим перед собой зеленую полосу, значит, мы добились прогресса.

Теперь, вместо того чтобы напрямую обращаться к нашему тестовому методу, мы должны использовать наш реальный интерфейс – метод run(). Изменим тест следующим образом:

```
test= WasRun(«testMethod»)
print(test.wasRun)
test.run()
print(test.wasRun)
```

Чтобы заставить тест работать, достаточно воспользоваться следующей несложной реализацией:

WasRun

```
def run(self):
self.testMethod()
```

Наша тестовая программа снова печатает на экране то, что нам нужно. Зачастую во время рефакторинга возникает ощущение, что необходимо разделить код, с которым вы работаете, на две части, чтобы работать с ними по отдельности. Если в конце работы они снова сольются воедино, – замечательно. Если нет, значит, вы можете оставить их отдельно друг от друга. В данном случае со временем мы планируем создать класс TestCase, однако вначале мы должны обособить части нашего примера.

Следующий этап – динамический вызов метода testMethod. Одной из приятных отличительных характеристик языка Python является возможность использования имен классов и методов в качестве функций (см. создание экземпляра класса WasRun). Получив атрибут, соответствующий имени теста, мы можем обратиться к нему, как к функции. В результате будет выполнено обращение к методу с соответствующим именем[12].

WasRun

```
class WasRun:
def __init__(self, name):
self.wasRun = None
self.name = name
def run(self):
method = getattr(self, self.name)
method()
```

Это еще один шаблон рефакторинга: разработать код, который работает с некоторым конкретным экземпляром, и обобщить его, чтобы он мог работать со всеми остальными экземплярами, для этого константы

заменяются переменными. В данном случае роль константы играет не некоторое значение, а фиксированный код (имя конкретного метода). Однако принцип остается тем же. В рамках TDD эта проблема решается очень легко: методика TDD снабжает вас конкретными работающими примерами, исходя из которых можно выполнить обобщение. Это значительно проще, чем выполнять обобщение исходя только из собственных умозаключений.

Теперь наш маленький класс WasRun занят решением двух разных задач: во-первых, он следит за тем, был ли выполнен метод; во-вторых, он динамически вызывает метод. Пришло время разделить полномочия (разделить нашу работу на две разные части). Прежде всего, создадим пустой суперкласс TestCase и сделаем класс WasRun производным классом:

TestCase
```
class TestCase:
pass
```

WasRun
```
class WasRun(TestCase):.
```

Теперь переместим атрибут name из подкласса в суперкласс:

TestCase
```
def __init__(self, name):
self.name = name
```

WasRun
```
def __init__(self, name):
self.wasRun = None
TestCase.__init__(self, name)
```

Наконец, замечаем, что метод run() использует только атрибуты суперкласса, значит, скорее всего, он должен располагаться в суперклассе. (Я всегда стараюсь размещать операции рядом с данными.)

TestCase
```
def __init__(self, name):
self.name= name
def run(self):
```

```
method = getattr(self, self.name)
method()
```

Естественно, между выполнениями этих модификаций я каждый раз запускаю тесты, чтобы убедиться, что все работает как надо.

Нам надоело смотреть на то, как наша программа каждый раз печатает одно и то же: None и 1. Использовав разработанный механизм, мы можем теперь написать:

TestCaseTest

```
class TestCaseTest(TestCase):
def testRunning(self):
test = WasRun("testMethod")
assert(not test.wasRun)
test.run()
assert(test.wasRun)
TestCaseTest("testRunning"). run()
```

~~Вызов тестового метода~~
Вызов метода setUp перед обращением к методу
Вызов метода tearDown после обращения к методу
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
Отчет о результатах

Сердцем этого теста являются операторы print, превращенные в выражения assert, таким образом, вы можете видеть, что выполненная нами процедура – это усложненный шаблон рефакторинга «Выделение метода» (Extract Method).

Я открою вам маленький секрет. Шажки, которыми мы двигались от теста к тесту в данной главе, выглядят смехотворно маленькими. Однако до того, как получить представленный результат, я пытался выполнить разработку более крупными шагами и потратил на это около шести часов (конечно, мне пришлось тратить дополнительное время на изучение тонкостей языка Python). Я два раза начинал с нуля и каждый раз думал, что мой код сработает, однако этого не происходило. Я понял, что попытка пропустить самый начальный, самый примитивный этап разработки, – это грубейшее нарушение принципов TDD.

Конечно же, вы не обязаны постоянно перемещаться такими

лилипутскими шажками. После того как вы освоите TDD, вы сможете двигаться вперед более уверенно, реализуя между тестами значительно больший объем функциональности. Однако чтобы в совершенстве освоить TDD, вы должны научиться перемещаться маленькими шажками тогда, когда это необходимо.

Далее мы планируем перейти к решению задачи обращения к методу setUp(). Однако вначале подведем итог.

В данной главе мы

- поняли, как начать работу со смехотворно малюсенького шага;
- реализовали функциональность путем создания фиксированного кода, а затем обобщения этого кода путем замены констант на переменные;
- использовали шаблон «Встраиваемый переключатель» (Pluggable Selector) и дали себе обещание не использовать его вновь в течение как минимум четырех месяцев, так как он существенно усложняет анализ кода;
- начали работу над инфраструктурой тестирования маленькими шажками.

19. Сервируем стол (метод setUp)

Начав писать тесты, вы обнаружите, что действуете в рамках некоторой общей последовательности (Билл Уэйк (Bill Wake) придумал сокращение 3A – *Arrange, Act, Assert*):

- вначале вы создаете некоторые тестовые объекты – *Arrange*;
- затем заставляете эти объекты действовать – *Act*;
- потом проверяете результаты их работы – *Assert*.

~~Вызов тестового метода~~
Вызов метода setUp перед обращением к методу
Вызов метода tearDown после обращения к методу
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
Отчет о результатах

Первый этап – *Arrange* – зачастую совпадает для нескольких разных тестов, в то время как второй и третий этапы для разных тестов различаются. У меня есть два числа: 7 и 9. Если я сложу их, я должен получить 16; если я вычту второе из первого, я ожидаю получить –2; наконец, если я перемножу их, я полагаю, должно получиться 63. Операции и ожидаемые результаты различаются, однако исходные данные одни и те же – два числа: 7 и 9.

Если подобное повторяется раз за разом в разных масштабах (а ведь так оно и есть), мы должны подумать о том, как можно оптимизировать создание тестовых объектов. Возникает конфликт между следующими двумя ограничениями:

Производительность. Мы хотим, чтобы тесты срабатывали как можно быстрее. Отсюда следует, что если одни и те же объекты используются в нескольких тестах, желательно, чтобы создание этих объектов выполнялось всего один раз.

Изоляция. Успех или неудача одного теста никак не должны влиять на работу других тестов. Если несколько тестов используют одни и те же объекты и если один из тестов меняет внутреннее состояние какого-либо объекта, результаты работы остальных тестов, скорее всего, изменятся.

Взаимозависимость между тестами приводит к одному весьма неприятному эффекту: если один тест перестает работать, остальные

десять тестов тоже перестают работать, несмотря на то, что тестируемый ими код выполняется правильно. Характерен также и другой, еще более неприятный эффект, когда порядок выполнения тестов имеет значение: если я запускаю тест А перед тестом Б, оба теста выполняются успешно, если я запускаю тест Б перед тестом А, тест А не выполняется. Или, еще хуже, код, проверяемый тестом Б, действует неправильно, однако из-за того, что тест А запускается перед тестом Б, тест Б выполняется успешно.

Итак, мы хотим избежать взаимозависимости между тестами. Предположим, что мы можем сделать процедуру создания объектов достаточно быстрой. В этом случае мы могли бы создавать объекты для теста каждый раз перед выполнением очередного теста. Этот подход в замаскированном виде уже использовался нами в классе WasRun, в котором требовалось, чтобы перед запуском теста флаг wasRun сбрасывался в состояние «ложь». Напишем тест:

TestCaseTest
```
def testSetUp(self):
test= WasRun("testMethod")
test.run()
assert(test.wasSetUp)
```

Чтобы запустить этот код, необходимо добавить в конец нашего файла строку TestCaseTest(«testSetUp»). run(). Интерпретатор вежливо сообщает нам, что атрибут с именем wasSetUp отсутствует. И немудрено, ведь мы пока еще не определили значение этого атрибута. Вот необходимый для этого код:

WasRun
```
def setUp(self):
self.wasSetUp= 1
```

Однако метод setUp() должен быть откуда-то вызван. Обращение к методу setUp() – это работа класса TestCase. Добавим соответствующий код:

TestCase
```
def setUp(self):
pass
def run(self):
```

```
self.setUp()
method = getattr(self, self.name)
method()
```

Чтобы заставить тест работать, мы сделали целых два шага – это слишком много, особенно если учесть, что мы движемся почти вслепую. Проверим, работает ли тест? Да, работает. Однако если вы хотите чему-то научиться, попробуйте придумать, как мы можем заставить тест работать, изменяя лишь по одному методу за один шаг.

Немедленно воспользуемся новым механизмом, чтобы сократить длину наших тестов. Прежде всего упростим класс WasRun, для этого перенесем процедуру установки флага wasRun в метод setUp():

WasRun

```
def setUp(self):
self.wasRun = None
self.wasSetUp = 1
```

Теперь можно упростить метод testRunning() – освободить его от

обязанности проверять состояние флага перед вызовом тестового метода. Можем ли мы быть настолько уверенными в правильной работе нашего кода? Только при условии, что в наборе тестов присутствует тестовый метод testSetUp(). Это часто встречающийся шаблон – один тест может быть простым, только если в системе имеется другой тест, выполняющийся успешно:

TestCaseTest

```
def testRunning(self):
test = WasRun("testMethod")
test.run()
assert(test.wasRun)
```

Мы также можем упростить сами тесты. В обоих случаях мы создаем экземпляр класса WasRun, а ведь задача создания тестовых объектов возлагается на подготовительный этап – именно об этом мы с вами говорили. Стало быть, мы можем создать объект WasRun в методе setUp(), а затем использовать его в тестовых методах. Каждый тестовый метод выполняется в отдельном экземпляре класса TestCaseTest, поэтому два разных теста не могут быть взаимозависимы. (Мы исходим из того, что объект не будет взаимодействовать с внешним миром некоторым

непредусмотренным уродливым способом, например путем изменения значений глобальных переменных.)

TestCaseTest

```
def setUp(self):
self.test = WasRun("testMethod")
def testRunning(self):
self.test.run()
assert(self.test.wasRun)
def testSetUp(self):
self.test.run()
assert(self.test.wasSetUp)
```

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
Вызов метода tearDown после обращения к методу
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
Отчет о результатах

Теперь сделаем так, чтобы после выполнения тестового метода обязательно выполнялся метод tearDown().

В данной главе мы

- решили, что на текущий момент тестов важнее простота, чем их производительность;
- написали тест для метода setUp() и реализовали этот метод;
- использовали метод setUp(), чтобы упростить тестируемый объект-контейнер теста;
- использовали метод setUp(), чтобы упростить тесты, проверяющие созданный нами тестовый объект (я же говорил, что временами это напоминает нейрохирургическую операцию на собственном мозге).

20. Убираем со стола (метод tearDown)

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
Вызов метода tearDown после обращения к методу
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
Отчет о результатах

Иногда для выполнения теста требуется выделить некоторые внешние ресурсы. Очевидно, что связанные с этим операции должны выполняться в теле метода setUp(). Если мы хотим, чтобы тесты были независимыми друг от друга, мы должны позаботиться об освобождении этих ресурсов. Для выполнения связанных с этим операций предлагаю использовать специальный метод tearDown(), который будет автоматически выполняться после завершения теста.

Как можно протестировать выполнение метода tearDown()? Проще всего – использовать еще один флаг. Однако все эти флаги начинают сбивать меня с толку. Если мы будем использовать флаги, мы упустим один очень важный аспект: метод setUp() должен быть выполнен непосредственно перед обращением к тестовому методу, а метод tearDown() – непосредственно после обращения к тестовому методу. Чтобы убедиться в этом, я намерен изменить стратегию тестирования. Предлагаю создать миниатюрный журнал, в котором будет отмечаться последовательность выполнения методов. Каждый метод будет добавлять в конец журнала соответствующую запись. Таким образом, просмотрев журнал, мы сможем установить порядок выполнения методов.

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
Вызов метода tearDown после обращения к методу
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
Отчет о результатах
Строка журнала в классе WasRun

WasRun

```
def setUp(self):
self.wasRun = None
self.wasSetUp = 1
self.log = "setUp "
```

Теперь можно изменить метод testSetUp(), чтобы вместо флага он проверял содержимое журнала:

TestCaseTest

```
def testSetUp(self):
self.test.run()
assert("setUp " == self.test.log)
```

После этого мы можем удалить флаг wasSetUp. Мы также можем добавить в журнал запись о выполнении метода:

WasRun

```
def testMethod(self):
self.wasRun = 1
self.log = self.log + "testMethod "
```

В результате нарушается работа теста testSetUp(), так как в момент выполнения этого метода журнал содержит строку «setUp testMethod». Изменяем ожидаемое значение:

TestCaseTest

```
def testSetUp(self):
self.test.run()
assert("setUp testMethod " == self.test.log)
```

Теперь этот тест выполняет работу обоих тестов, поэтому можно удалить testRunning и переименовать testSetUp:

TestCaseTest

```
def setUp(self):
self.test = WasRun("testMethod")
def testTemplateMethod(self):
self.test.run()
assert("setUp testMethod " == self.test.log)
```

Мы используем экземпляр класса WasRun всего в одном месте, поэтому необходимо отменить добавленный ранее хитрый трюк, связанный с setUp():

TestCaseTest

```
def testTemplateMethod(self):
test = WasRun("testMethod")
test.run()
assert("setUp testMethod " == test.log)
```

Нет ничего страшного в том, что мы сделали рефакторинг, исходя из нескольких ранних соображений, а чуть позже отменили его, – подобная ситуация складывается достаточно часто. Некоторые предпочитают подождать, пока у них накопится достаточное количество оснований для рефакторинга, иными словами, они оттягивают выполнение рефакторинга, чтобы быть полностью уверенными в его необходимости. Они поступают так потому, что не любят аннулировать результаты проделанной работы. Однако я предпочитаю не отвлекаться на рассуждения о том, не придется ли в будущем отменять то или иное исправление, необходимое мне в настоящем. Вместо этого я предпочитаю сосредоточиться на дизайне. По этой причине я рефлекторно делаю рефакторинг тогда, когда считаю нужным, ни капли не опасаясь, что сразу после этого мне, возможно, придется отменить его.

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
Вызов метода tearDown после обращения к методу
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
Отчет о результатах
~~Строка журнала в классе WasRun~~

Теперь мы готовы к реализации метода tearDown(). Ага! Опять я вас поймал! Теперь мы готовы к тестированию метода tearDown():

TestCaseTest

```
def testTemplateMethod(self):
test = WasRun("testMethod")
```

```
test.run()
assert("setUp testMethod tearDown " == test.log)
```

Он потерпел неудачу. Чтобы заставить его работать, выполняем несложные добавления:

TestCase

```
def run(self, result):
result.testStarted()
self.setUp()
exec "self." + self.name + "()"
self.tearDown()
```

WasRun

```
def setUp(self):
self.log = "setUp "
def testMethod(self):
self.log = self.log + "testMethod "
def tearDown(self):
self.log = self.log + "tearDown "
```

Неожиданно мы получаем ошибку не в классе WasRun, а в классе TestCaseTest. У нас нет «пустой» реализации метода teardown() в классе TestCase:

TestCase

```
def tearDown(self):
pass
```

Мы начинаем получать пользу от разрабатываемой инфраструктуры. Замечательно! Никакого рефакторинга не требуется. Очевидная реализация, созданная нами после обнаружения ошибки, сработала, и код получился чистым.

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
~~Вызов метода tearDown после обращения к методу~~
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов

Отчет о результатах
~~Строка журнала в классе WasRun~~

Далее мы перейдем к формированию отчета о результатах выполнения тестов. Вместо использования встроенного в Python механизма обработки ошибок мы планируем реализовать и использовать собственный механизм наблюдения за работой тестов.

В данной главе мы

- перешли от использования флагов к использованию журнала;
- создали тесты для метода tearDown() и реализовали этот метод с использованием нового механизма журналирования;
- обнаружили проблему и вместо того чтобы возвращаться назад, смело исправили ошибку.

21. Учет и контроль

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
~~Вызов метода tearDown после обращения к методу~~
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
Отчет о результатах
~~Строка журнала в классе WasRun~~

Метод tearDown() должен выполняться, даже когда в процессе выполнения теста возникло исключение. Однако чтобы добиться этого, мы должны перехватывать исключения. (Честно говоря, я уже пытался реализовать это, но у меня не получилось, и я вернул все на свои места.) Если при реализации перехвата исключений мы допустим ошибку, мы можем не заметить этого, так как стандартный механизм доклада об исключениях не будет функционировать.

Работая в стиле TDD, важно понимать, что особое значение имеет порядок, в котором вы реализуете тесты. Выбирая тест, над которым я буду работать дальше, я стараюсь выбрать тот, который, во-первых, послужит для меня источником новых знаний, а во-вторых, достаточно прост, чтобы я был уверен в том, что могу заставить его работать. Если я добиваюсь успешного выполнения этого теста, но захожу в тупик при реализации следующего, я вполне могу выполнить откат назад на два шага. Было бы неплохо, если бы среда разработки оказывала мне в этом помощь. Например, было бы неплохо, если бы в момент срабатывания всех тестов автоматически создавалась резервная копия всего исходного кода, с которым я работаю.

После выполнения всех тестов желательно получить информацию о том, как они выполнились, например: «запущено 5, неудачных 2: TestCaseTest.testFooBar – ZeroDivideException, MoneyTest.testNegation – AssertionError». Если тесты перестают выполняться или результаты перестают отображаться на экране мы, по крайней мере, сможем обнаружить ошибку. Однако наша инфраструктура не обязана знать обо всех разработанных тестах.

Пусть метод TestCase.run() возвращает объект класса TestResult с результатами выполнения теста (вначале тест будет только один, однако

позже мы усовершенствуем этот объект).

```
TestCaseTest
def testResult(self):
test = WasRun("testMethod")
result = test.run()
assert("1 run, 0 failed" == result.summary())
```

Начнем с поддельной реализации:

```
TestResult
class TestResult:
def summary(self):
return "1 run, 0 failed"
```

Теперь сделаем так, чтобы в результате выполнения метода TestCase.run() возвращался объект класса TestResult:

```
TestCase
def run(self):
self.setUp()
method = getattr(self, self.name)
method()
self.tearDown()
return TestResult()
```

Теперь, когда все тесты выполнились успешно, можно сделать реализацию метода summary() реальной. Как и раньше, будем двигаться маленькими шажками. Для начала заменим количество выполненных тестов константой:

```
TestResult
def __init__(self):
self.runCount = 1
def summary(self):
return "%d run, 0 failed" % self.runCount
```

(Оператор % в языке Python является аналогом функции sprintf в языке C.) Однако runCount не может быть константой, это должна быть

переменная, значение которой вычисляется исходя из количества выполненных тестов. Мы можем инициализировать эту переменную значением 0, а затем увеличивать ее на единицу при выполнении очередного теста.

TestResult

```
def __init__(self):
self.runCount = 0
def testStarted(self):
self.runCount = self.runCount + 1
def summary(self):
return "%d run, 0 failed" % self.runCount
```

Теперь мы должны позаботиться о вызове этого нового метода:

TestCase

```
def run(self):
result = TestResult()
result.testStarted()
self.setUp()
method = getattr(self, self.name)
method()
self.tearDown()
return result
```

Мы точно так же могли бы преобразовать константу «0», обозначающую количество тестов, потерпевших неудачу, в переменную, как сделали это с переменной runCount, однако существующие тесты этого не требуют. Поэтому напишем новый тест:

TestCaseTest

```
def testFailedResult(self):
test = WasRun("testBrokenMethod")
result = test.run()
assert("1 run, 1 failed", result.summary)
```

Здесь:

WasRun

```
def testBrokenMethod(self):
```

raise Exception
~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
~~Вызов метода tearDown после обращения к методу~~
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
~~Отчет о результатах~~
~~Строка журнала в классе WasRun~~
Отчет о неудачных тестах

Мы немедленно замечаем, что исключение, генерируемое в методе WasRun.testBrokenMethod(), не перехватывается. Нам хотелось бы перехватить это исключение и в отчете о результатах тестирования отметить, что тест потерпел неудачу. Добавим соответствующий пункт в список задач.

Подведем итог. Мы

• разработали поддельную реализацию и начали поэтапно делать ее реальной путем замены констант переменными;

• написали еще один тест;

• когда тест потерпел неудачу, написали еще один тест меньшего масштаба, чтобы обеспечить выполнение неудачного теста.

22. Обработка неудачного теста

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
~~Вызов метода tearDown после обращения к методу~~
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
~~Отчет о результатах~~
~~Строка журнала в классе WasRun~~
Отчет о неудачных тестах

Напишем еще один тест меньшего масштаба, демонстрирующий, что при обнаружении неудачного теста наша инфраструктура распечатывает на экране корректный результат:

TestCaseTest

```
def testFailedResultFormatting(self):
result = TestResult()
result.testStarted()
result.testFailed()
assert("1 run, 1 failed" == result.summary())
```

В данном тесте фигурируют два новых метода: testStarted() и testFailed(). Первый из них должен вызываться в начале работы теста, а второй – в случае, если тест не срабатывает. Тестовый метод testFailedResultFormatting() предполагает, что, если эти два метода вызываются в указанном порядке, отчет о результате тестирования должен выглядеть корректно. Если мы заставим этот тестовый метод работать, наша проблема сведется к тому, чтобы обеспечить вызов метода testStarted() в начале выполнения теста и вызов testFailed() в случае, если тест потерпел неудачу.

Чтобы реализовать функциональность методов testStarted() и testFailed(), воспользуемся двумя счетчиками: счетчиком запущенных тестов и счетчиком неудачных тестов:

TestResult

```
def __init__(self):
```

```
self.runCount = 0
self.errorCount = 0
def testFailed(self):
self.errorCount = self.errorCount + 1
```

Если счетчик будет работать корректно (мы должны были бы это протестировать, однако кофе, похоже, ударил мне в голову), отчет будет напечатан корректно:

TestResult

```
def summary(self):
return "%d run, %d failed" % (self.runCount, self.failureCount)
```

Теперь можно предположить, что, если обращение к методу testFailed() будет выполнено корректно, мы получим на экране желаемый результат. Но где следует вызвать этот метод? В обработчике исключения, перехваченного в процессе выполнения тестового метода:

TestCase

```
def run(self):
result = TestResult()
result.testStarted()
self.setUp()
try:
method = getattr(self, self.name)
method()
except:
result.testFailed()
self.tearDown()
return result
```

В этом коде присутствует одно тонкое место: если исключение возникает во время выполнения метода setUp(), оно не будет перехвачено. Это означает, что в определенной степени тесты зависят друг от друга. Однако прежде, чем менять код, мы должны написать еще один тест. (Я научил мою старшую дочку Бетани программировать в стиле TDD, когда ей было 12 лет. Это самый первый стиль программирования, с которым она познакомилась. Она не умеет программировать иначе и абсолютно уверена, что добавлять в программу новый функциональный код можно только в

случае, если существует тест, который терпит неудачу. Однако более опытные программисты, как правило, вынуждены напоминать себе о том, что сначала необходимо писать тесты, а потом – код, заставляющий их выполняться.) Я оставляю этот следующий тест и его реализацию вам в качестве самостоятельного упражнения (мои пальцы опять устали):

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
~~Вызов метода tearDown после обращения к методу~~
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
~~Отчет о результатах~~
~~Строка журнала в классе WasRun~~
~~Отчет о неудачных тестах~~
Перехват и отчет об ошибках setUp

Далее мы сделаем так, чтобы наша инфраструктура смогла запустить несколько тестов подряд.

В данной главе мы
• обеспечили успешное выполнение нашего теста меньшего масштаба;
• заново приступили к реализации более крупного теста;
• обеспечили успешное выполнение крупного теста, воспользовавшись механизмом, реализованным для маленького теста;
• обратили внимание на потенциальную проблему, но вместо того, чтобы немедленно браться за ее решение, добавили соответствующую пометку в список задач.

23. Оформляем тесты в набор

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
~~Вызов метода tearDown после обращения к методу~~
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
~~Отчет о результатах~~
~~Строка журнала в классе WasRun~~
~~Отчет о неудачных тестах~~
Перехват и отчет об ошибках setUp

Мы не можем оставить работу над xUnit, не реализовав класс TestSuite, представляющий собой набор тестов. Конец нашего файла, где мы запускаем все наши тесты, выглядит весьма неопрятно:

```
print(TestCaseTest(«testTemplateMethod»). run(). summary())
print(TestCaseTest("testResult"). run(). summary())
print(TestCaseTest("testFailedResultFormatting"). run(). summary())
print(TestCaseTest("testFailedResult"). run(). summary())
```

Дублирование – это всегда плохо, за исключением случаев, когда вы используете его в качестве мотивации для поиска недостающего элемента дизайна. В данном случае нам хотелось бы сгруппировать тесты и запустить их при помощи одной команды. (Мы приложили массу усилий для изоляции тестов, однако все эти усилия не окупят себя, если мы будем запускать тесты по одному.) Еще одна хорошая причина, по которой было бы неплохо реализовать TestSuite, заключается в том, что этот класс хорошо демонстрирует использование шаблона «Компоновщик» (Composite), – мы хотим, чтобы набор тестов вел себя в точности так же, как единичный тест.

Мы хотим обладать возможностью создать набор тестов (объект TestSuite), добавить в него несколько тестов и получить общий результат выполнения всех этих тестов:

TestCaseTest
```
def testSuite(self):
suite = TestSuite()
```

```
suite.add(WasRun("testMethod"))
suite.add(WasRun("testBrokenMethod"))
result = suite.run()
assert("2 run, 1 failed" == result.summary())
```

Для успешного выполнения теста создадим в объекте TestSuite список тестов и реализуем метод add(), который просто добавляет тест, переданный в качестве аргумента, в список:

TestSuite

```
class TestSuite:
def __init__(self):
self.tests = []
def add(self, test):
self.tests.append(test)
```

(В языке Python оператор [] создает пустую коллекцию.)

Однако с реализацией метода run() возникают проблемы. Мы хотим, чтобы результаты срабатывания всех тестов накапливались в едином объекте класса TestResult. Таким образом, мы можем написать следующий код:

TestSuite

```
def run(self):
result = TestResult()
for test in tests:
test.run(result)
return result
```

Здесь оператор цикла «for test in tests» выполняет итерации по всем элементам последовательности tests, присваивает их по одному переменной цикла test и запускает соответствующий тест. Однако шаблон «Компоновщик» (Composite) подразумевает, что набор объектов должен обладать точно таким же интерфейсом, каким обладает отдельный объект. Если мы передаем параметр методу TestCase.run(), значит, мы должны передавать точно такой же параметр методу TestSuite.run(). Можно использовать одну из трех альтернатив.

• Воспользоваться встроенным в язык Python механизмом параметров

со значениями по умолчанию. К сожалению, значение параметра по умолчанию вычисляется во время компиляции, но не во время выполнения, а мы не хотим повторно использовать один и тот же объект TestResult.

• Разделить метод на две части – одна создает объект TestResult, а вторая выполняет тест, используя переданный ей объект TestResult. Я не могу придумать хороших имен для двух частей метода, а это означает, что данная стратегия не является самой лучшей.

• Создавать объекты TestResult в вызывающем коде.

Мы будем создавать объекты TestResult в вызывающем коде. Этот шаблон называется «Накапливающий параметр» (Collecting Parameter).

TestCaseTest

```
def testSuite(self):
suite = TestSuite()
suite.add(WasRun("testMethod"))
suite.add(WasRun("testBrokenMethod"))
result = TestResult()
suite.run(result)
assert("2 run, 1 failed" == result.summary())
```

При использовании данного подхода метод run() не возвращает никакого явного значения:

TestSuite

```
def run(self, result):
for test in tests:
test.run(result)
```

TestCase

```
def run(self, result):
result.testStarted()
self.setUp()
try:
method = getattr(self, self.name)
method()
except:
result.testFailed()
self.tearDown()
```

Теперь мы можем облагородить обращение к тестовым методам в конце файла:

```
suite = TestSuite()
suite.add(TestCaseTest("testTemplateMethod"))
suite.add(TestCaseTest("testResult"))
suite.add(TestCaseTest("testFailedResultFormatting"))
suite.add(TestCaseTest("testFailedResult"))
suite.add(TestCaseTest("testSuite"))
result = TestResult()
suite.run(result)
print(result.summary())
```

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
~~Вызов метода tearDown после обращения к методу~~
Метод tearDown должен вызываться даже в случае неудачи теста
Выполнение нескольких тестов
~~Отчет о результатах~~
~~Строка журнала в классе WasRun~~
~~Отчет о неудачных тестах~~
Перехват и отчет об ошибках setUp
Создать объект TestSuite автоматически на основе класса TestCase

Здесь слишком много повторяющегося кода, от которого можно избавиться, если обеспечить способ конструирования набора тестов автоматически, исходя из предоставленного класса TestCase.

Однако вначале восстановим корректное выполнение четырех не неудачных тестов (эти тесты используют старый интерфейс функции run() без аргументов):

TestCaseTest

```
def testTemplateMethod(self):
test = WasRun("testMethod")
result = TestResult()
test.run(result)
assert("setUp testMethod tearDown " == test.log)
def testResult(self):
test = WasRun("testMethod")
```

```
result = TestResult()
test.run(result)
assert("1 run, 0 failed" == result.summary())
def testFailedResult(self):
test = WasRun("testBrokenMethod")
result = TestResult()
test.run(result)
assert("1 run, 1 failed" == result.summary())
def testFailedResultFormatting(self):
result = TestResult()
result.testStarted()
result.testFailed()
assert("1 run, 1 failed" == result.summary())
```

Обратите внимание, что каждый из тестов создает экземпляр класса TestResult – эту операцию можно выполнить однократно внутри метода setUp(). Благодаря реализации этой идеи мы упростим тесты, однако сделаем их несколько более сложными в прочтении:

TestCaseTest

```
def setUp(self):
self.result = TestResult()
def testTemplateMethod(self):
test = WasRun("testMethod")
test.run(self.result)
assert("setUp testMethod tearDown " == test.log)
def testResult(self):
test = WasRun("testMethod")
test.run(self.result)
assert("1 run, 0 failed" == self.result.summary())
def testFailedResult(self):
test = WasRun("testBrokenMethod")
test.run(self.result)
assert("1 run, 1 failed" == self.result.summary())
def testFailedResultFormatting(self):
self.result.testStarted()
self.result.testFailed()
assert("1 run, 1 failed" == self.result.summary())
def testSuite(self):
```

```
suite = TestSuite()
suite.add(WasRun("testMethod"))
suite.add(WasRun("testBrokenMethod"))
suite.run(self.result)
assert("2 run, 1 failed" == self.result.summary())
```

~~Вызов тестового метода~~
~~Вызов метода setUp перед обращением к методу~~
~~Вызов метода tearDown после обращения к методу~~
Метод tearDown должен вызываться даже в случае неудачи теста
~~Выполнение нескольких тестов~~
~~Отчет о результатах~~
~~Строка журнала в классе WasRun~~
~~Отчет о неудачных тестах~~
Перехват и отчет об ошибках setUp
Создать объект TestSuite автоматически на основе класса TestCase

Все эти бесчисленные ссылки self выглядят ужасно, однако без этого в языке Python никак не обойтись. Если бы этот язык изначально был объектно-ориентированным, наверное, в этих ссылках не было бы надобности, а ссылки на глобальные переменные требовали бы квалификации. Однако язык Python изначально является интерпретируемым языком с добавленной в него поддержкой объектов (надо отметить, что поддержка объектов в этом языке реализована великолепно). В результате по умолчанию переменные считаются глобальными, а явные ссылки на self – необходимыми.

Я оставляю реализацию оставшихся пунктов вам в качестве упражнения. Надеюсь, обретенные навыки работы в стиле TDD помогут вам.

Чтобы подвести итог, напомню, что в данной главе мы

• написали тест для класса TestSuite;

• написали часть реализации, однако не добились успешного выполнения тестов – это нарушение правил (я уверен, что существует простая поддельная реализация, которая заставила бы тесты работать, благодаря чему мы могли бы выполнять рефакторинг, имея перед глазами зеленую полоску, однако сейчас я не хочу думать на эту тему);

• изменили интерфейс метода run(), благодаря чему набор тестов можно использовать точно так же, как и отдельный тест, – в результате тесты наконец выполнились успешно;

- выполнили рефакторинг имеющихся тестов – переместили общий код создания объекта результатов в метод setUp().

24. Ретроспектива xUnit

Если перед вами встала задача разработки своей собственной инфраструктуры тестирования, методика, описанная в части II данной книги, послужит вам руководством. Не следует слишком много внимания уделять деталям реализации – значительно больший интерес представляют тесты. Если вы напишете код, обеспечивающий успешное выполнение представленных здесь тестов, в вашем распоряжении окажется минимальная инфраструктура тестирования, пригодная для запуска тестов в условиях изоляции и обеспечивающая композицию тестов. Вы сможете приступить к разработке программного кода в стиле TDD.

На момент написания данной книги инфраструктура тестирования xUnit адаптирована для более чем 30 языков программирования. Язык, на котором вы программируете, скорее всего, уже обладает своей собственной реализацией xUnit. Однако, даже если кто-то уже сделал это до вас, возможно, будет лучше, если вы попробуете разработать свою собственную новую версию xUnit самостоятельно. На то есть две важные причины:

Контроль над реализацией. Основополагающая характеристика xUnit – это простота. Мартин Фаулер (Martin Fowler) сказал: «Никогда в истории программной индустрии еще не было случая, чтобы столь многие разработчики были обязаны столь немногому количеству строк программного кода». На мой взгляд, некоторые реализации xUnit к настоящему времени стали слишком большими и сложными. Если вы разработаете собственную версию xUnit, то получите инструмент, который вы будете контролировать в полном объеме.

Обучение. Когда я сталкиваюсь с необходимостью изучить новый язык программирования, я приступаю к реализации xUnit. Когда я добиваюсь срабатывания первых восьми-десяти тестов, я овладеваю навыками работы с основными конструкциями и возможностями нового для меня языка.

Когда вы начнете работать с xUnit, вы обнаружите, что существует значительная разница между выражениями assert, потерпевшими неудачу, и ошибками других типов, возникающими в процессе выполнения тестов. В отличие от остальных ошибок выражения assert требуют больше времени для отладки. Из-за этого большинство реализаций xUnit отличает сбои операторов assert от всех остальных ошибок: в рамках GUI зачастую информация об ошибках отображается в начале списка.

Инфраструктура JUnit объявляет простой интерфейс Test, который

реализуется классами TestCase и TestSuite. Если вы хотите создать тестовый класс, который мог бы взаимодействовать со стандартными средствами тестирования, встроенными в JUnit, вы можете реализовать функции интерфейса Test самостоятельно:

```
public interface Test {
public abstract int countTestCases();
public abstract void run(TestResult result);
}
```

В языках с оптимистическим (динамическим) приведением типов можно даже не объявлять о поддержке этого интерфейса – достаточно реализовать входящие в его состав операции. При использовании языка сценариев Сценарий может ограничивать реализацию countTestCases() возвратом единицы и выполнять проверку TestResult на отказ, а вы можете выполнять ваши сценарии с обычными объектами TestCase.

Часть III. Шаблоны разработки через тестирование

Далее следуют «величайшие хиты» – шаблоны разработки через тестирование. Некоторые из них являются эффективными приемами работы в стиле TDD, другие – шаблонами проектирования и, наконец, третьи – шаблонами рефакторинга. Третья часть книги является коллекцией справочного материала, необходимого как для освоения представленных в книге примеров, так и для самостоятельного совершенствования навыков работы в стиле TDD. Здесь представлены сведения, которые помогут лучше понять смысл примеров, рассмотренных в первых двух частях книги, а также подогреют ваш интерес и стимулируют обратиться к дополнительной информации, которую следует искать в других источниках.

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.

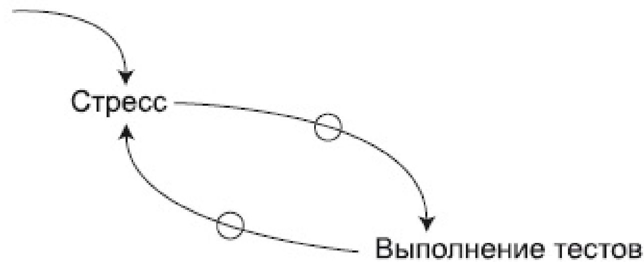

Рис. 25.1. Зловещая спираль «нет времени для тестирования»

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[13]? Попробуйте писать их в первую очередь. Неужели вам не нравится самоподобие?

• С чего следует начать построение системы? С формулировки пожеланий[14] 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.

• 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.

26. Шаблоны красной полосы

В данной главе речь пойдет о шаблонах, которые подскажут вам, когда писать тесты, где писать тесты и когда прекратить писать тесты.

Тест одного шага (One Step Test)

Какой следующий тест лучше всего выбрать из списка задач для реализации? Выбирайте тест, который, во-первых, научит вас чему-либо, а во-вторых, который вы сможете реализовать.

Каждый тест должен соответствовать одному шагу в направлении к вашей основной цели. Взгляните на этот список тестов и попробуйте определить, какой тест лучше всего выбрать в качестве следующего для реализации:

плюс;
минус;
умножение;
деление;
сложение с такой же валютой;
равенство;
равенство нулю;
нулевой обмен;
обмен одной и той же валюты;
обмен двух валют;
курс кросс-обмена.

Не существует единственно правильного ответа. То, что для меня, ни разу не занимавшегося реализацией этих объектов, будет выглядеть как один шаг, для вас, обладающих достаточным опытом, может оказаться одной десятой шага. Если вы не можете найти в списке тест, соответствующий одному шагу, добавьте в список дополнительные тесты, реализация которых поможет вам приблизиться к реализации тестов, уже присутствующих в списке.

Когда я смотрю на список тестов, я рассуждаю: «Это очевидно, это очевидно, об этом я не имею ни малейшего представления, это очевидно,

здесь – никаких идей, о чем я думал, когда писал это? А! Вспомнил! Я думаю, что мог бы это сделать». Этот последний тест я реализую следующим. С одной стороны, он не кажется мне очевидным, с другой стороны, я уверен в том, что смогу заставить его работать.

Программа, выросшая из подобных тестов, может быть написана в рамках нисходящего подхода (сверху вниз), так как вы можете начать с теста, который ориентирован на вариант полного вычисления. Программа, выросшая из тестов, может быть написана и в рамках восходящего подхода (снизу вверх), так как вы начинаете с небольших кусочков и собираете их в конструкцию постепенно увеличивающегося размера.

И нисходящий, и восходящий подходы не представляют реального описания процесса. Во-первых, вертикальная метафора – это упрощенная визуализация процесса изменения программы в течение разработки. Для описания процесса разработки, основанной на тестировании, лучше подходит метафора *Развития* или *Эволюции*: внешняя среда влияет на программу, а программа влияет на внешнюю среду. Во-вторых, если мы хотим, чтобы в нашей метафоре присутствовало направление, лучшим описанием будет «от известного к неизвестному». Подразумевается, что мы обладаем некоторыми знаниями и опытом и ожидаем, что в процессе разработки мы будем узнавать нечто новое. Объединим эти две метафоры и получим, что программа эволюционирует от известного к неизвестному.

Начальный тест (Starter Test)

С какого теста следует начать разработку? Начните с тестирования варианта операции, который не подразумевает выполнения каких-либо осмысленных действий, то есть ничего не делает.

Приступая к реализации операции, вы прежде всего должны ответить на вопрос: «Где она должна располагаться?» Пока вы не ответите на этот вопрос, вы не будете знать, какой код необходимо написать, чтобы протестировать эту операцию. Как уже неоднократно рекомендовалось, не следует решать несколько проблем одновременно. Значит, вы должны выбрать такой тест, который позволит вам искать ответ только на один этот вопрос и на время забыть обо всех остальных вопросах.

Если вы с самого начала приступите к реализации реалистичного теста, вам придется искать ответы на несколько вопросов одновременно:

- Где должна располагаться операция?
- Какие входные данные считать корректными?

• Каким должен быть корректный результат выполнения операции при использовании выбранных входных данных?

Если вы начнете с реалистичного теста, вы слишком долгое время будете вынуждены действовать без обратной связи. Красный – зеленый – рефакторинг, красный – зеленый – рефакторинг. На выполнение этого цикла должно уходить всего несколько минут.

Но как сократить время цикла? Для этого вы можете воспользоваться тривиальными входными и выходными данными. Вот простой пример: если функция должна складывать многозначные вещественные числа с точностью до тысячного знака после запятой, вовсе не обязательно начинать ее реализацию с теста, проверяющего результат сложения таких огромных чисел. Вполне можно начать с тривиального теста 3 + 4 = 7. Вот еще один пример. В группе электронных новостей, посвященной экстремальному программированию, один из участников поинтересовался, как написать программу минимизации количества полигонов (многоугольников), составляющих некоторую поверхность. На вход подается набор полигонов, комбинация которых представляет собой некоторый трехмерный объект. На выходе должна получиться комбинация полигонов, которая описывает точно такой же объект (поверхность), но включает в себя минимальное возможное количество полигонов. «Как я могу разработать подобную программу, если для того, чтобы заставить тест сработать, я должен быть как минимум доктором наук?»

Используя шаблон «Начальный тест» (Starter Test), мы получаем ответ:

• Вывод должен быть точно таким же, как ввод. Некоторые комбинации полигонов изначально являются минимальными.

• Ввод должен быть как можно меньшего размера. Например, единственный полигон или даже пустой список полигонов.

Мой начальный тест выглядел следующим образом:

```
Reducer r = new Reducer(new Polygon());
assertEquals(0, reducer.result(). npoints);
```

Отлично! Первый тест заработал. Теперь можно перейти к остальным тестам в списке…

К начальному тесту следует применить рассмотренное ранее правило «Тест одного шага» (One Step Test): самый первый тест должен научить вас чему-то новому, кроме того, вы должны обладать возможностью достаточно быстро заставить его работать. Если вы реализуете подобный код уже не в первый раз, вы можете выбрать начальный тест для одной или

даже двух операций. Вы должны быть уверены, что сможете быстро заставить тест работать. Если вы приступаете к реализации чего-либо достаточно сложного и делаете это впервые, начните с самого простого теста, который вы только можете представить.

Я часто замечаю, что мой начальный тест работает на достаточно высоком уровне и скорее напоминает тест всего приложения. Например, простой сетевой сервер. Самый первый тест выглядит следующим образом:

```
StartServer
Socket= new Socket
Message = "hello"
Socket.write(message)
AssertEquals(message, socket.read)
```

Остальные тесты пишутся только на стороне сервера: «Предположим, что мы получаем строки наподобие этой…»

Объясняющий тест (Explanation Test)

Как распространить в своей команде использование автоматического тестирования? Для любых объяснений используйте тесты и спрашивайте тесты у тех, кто пытается вам что-либо объяснить.

Если вы единственный член команды, работающий в стиле TDD, вы можете почувствовать себя неуютно и одиноко. Однако вскоре после того, как вы начнете работать в стиле TDD, вы обратите внимание на уменьшение количества проблем, связанных с интеграцией, и снижение количества дефектов, обнаруженных в проверенном коде. Дизайн вашего кода будет проще, и его легче будет объяснять. Может случиться так, что ваши коллеги проявят интерес к тестированию и предварительному тестированию разрабатываемого кода.

Опасайтесь оголтелого энтузиазма со стороны новичков. Подобный энтузиазм может оттолкнуть тех, кто еще не до конца понял преимущества и необходимость предварительного тестирования. Если внедрение TDD производить насильственными методами, это может привести к негативным результатам. Если вы руководитель или лидер, вы не должны насильно заставлять людей менять стиль, в рамках которого они работают.

Но что можно сделать? Лучше всего предлагать вашим коллегам объяснять работу кода в форме тестов: «Подожди-ка, если я правильно

понял, объект Foo будет таким, а объект Bar будет таким, значит, в результате получится 76?» Кроме того, вы можете объяснять работу кода в виде тестов: «Вот как это работает. Если объект Foo будет таким, а объект Bar будет таким, в результате получится 76. Однако если объект Foo будет таким, а объект Bar будет таким, в результате получится 67».

Вы можете делать это на более высоком уровне абстракции. Если кто-то пытается объяснить вам работу кода при помощи диаграммы последовательности обмена сообщениями, вы можете предложить ему преобразовать эту диаграмму в более понятную форму. После этого вы пишете тест, содержащий в себе все видимые на диаграмме объекты и сообщения.

Тест для изучения (Learning Test)[15]

Когда необходимо писать тесты для программного обеспечения, разработанного сторонними разработчиками? Перед тем как вы впервые воспользуетесь новыми возможностями этого программного обеспечения.

Предположим, что вы приступаете к разработке программы, основанной на использовании библиотеки Mobile Information Device Profile (MIDP) для языка Java. Вы собираетесь сохранить некоторые данные в объекте RecordStore и затем извлечь их оттуда. Должны ли вы просто написать код в надежде на то, что он заработает? Это один из возможных методов разработки.

Есть альтернативный метод. Обратите внимание на то, что вы собираетесь использовать новый метод нового класса. Вместо того чтобы просто воспользоваться им внутри разрабатываемого вами кода, вы пишете небольшой тест, который позволяет вам убедиться в том, что API работает так, как вы того ожидаете. Таким образом, вы можете написать:

```
RecordStore store;

public void setUp() {
store = RecordStore.openRecordStore("testing", true);
}

public void tearDown() {
RecordStore.deleteRecordStore("testing");
}
```

```
public void testStore() {
int id = store.addRecord(new byte[] {5, 6}, 0, 2);
assertEquals(2, store.getRecordSize(id));
byte[] buffer = new byte[2];
assertEquals(2, store.getRecord(id, buffer, 0));
assertEquals(5, buffer[0]);
assertEquals(6, buffer[1]);
}
```

Если ваше понимание API совпадает с действительностью, значит, тест сработает с первого раза.

Джим Ньюкирк рассказал мне о проекте, в котором разработка тестов для обучения выполнялась на регулярной основе. Как только от сторонних разработчиков поступала новая версия пакета, немедленно запускались имеющиеся тесты. В случае необходимости в тесты вносились исправления. Если тесты завершались неудачей, не было никакого смысла запускать приложение, так как оно определенно не заработает. Если же все тесты выполнялись успешно, значит, и приложение заработает.

Еще один тест (Another Test)

Как предотвратить уход дискуссии от основной темы? Когда возникает посторонняя, но интересная мысль, добавьте в список еще один тест и вернитесь к основной теме.

Я люблю пространные дискуссии (вы уже прочитали большую часть книги, поэтому, скорее всего, пришли к такому же выводу самостоятельно). Если постоянно жестко следить за соблюдением основной темы обсуждения, можно потерять множество бриллиантовых идей. Вы перескакиваете с одной темы на другую, затем на третью и, наконец, не успеваете заметить, что ушли далеко от того, с чего начали. А кого это волнует, ведь обсуждать вещи в таком стиле – это круто!

Иногда программирование – это прорыв, генерация гениальной идеи и вдохновение. Однако в большинстве случаев программирование – весьма рутинная работа. У меня есть десять вещей, которые я должен реализовать. Я постоянно откладываю на потом задачу номер четыре. Один из способов избежать работы (и, возможно, сопутствующего страха) – вступить в длинные пространные рассуждения на самые разные темы.

Потеряв огромное количество времени впустую, я пришел к выводу, что иногда лучше сосредоточиться на конкретной проблеме и не отвлекаться на побочные мысли. Когда я работаю в подобном стиле, я приветствую любые новые идеи, однако не позволяю им слишком сильно отвлекать мое внимание. Я записываю новые идеи в список и затем возвращаюсь к тому, над чем я работаю.

Регрессионный тест (Regression Test)

Что необходимо сделать в первую очередь в случае, если обнаружен дефект? Написать самый маленький из всех возможных тестов, который не работает, и восстановить его работоспособность.

Регрессионные тесты – это тесты, которые вы наверняка написали бы в процессе обычной разработки, если бы своевременно обнаружили проблему. Каждый раз, столкнувшись с необходимостью написать регрессионный тест, подумайте о том, как вы могли бы узнать о необходимости написать этот тест ранее, то есть тогда, когда выполняли разработку.

Полезным может оказаться тестирование на уровне всего приложения. Регрессионные тесты для всего приложения дают пользователям возможность сообщить вам, что неправильно и чего они на самом деле ожидают. Регрессионные тесты меньшего масштаба являются для вас способом улучшить качество тестирования. Если вы получили доклад о дефекте, в котором описывается появление большого отрицательного целого числа в отчете, значит, в будущем вам необходимо уделить дополнительное внимание тестированию граничных значений для целых чисел.

Возможно, для того чтобы изолировать дефект, вам потребуется выполнить рефакторинг системы. В этом случае, демонстрируя дефект, система как бы говорит вам: «Ты еще не вполне закончил проектировать меня».

Перерыв (Break)

Что делать, если вы почувствовали усталость или зашли в тупик? Прервите работу и отдохните.

Выпейте кофе, пройдитесь или даже вздремните. Стряхните с себя

эмоциональное напряжение, связанное с решениями, которые вы принимаете, и символами, которые вы набираете на клавиатуре.

Иногда самого короткого перерыва достаточно, чтобы недостающая идея возникла в вашей голове. Возможно, вставая из-за компьютера, вы неожиданно нащупаете нужную нить: «Да я же не попробовал этот метод с пересмотренными параметрами!» В любом случае прервитесь. Дайте себе пару минут. Идея никуда не убежит.

Если, несмотря на отдых, идея не приходит вам в голову, пересмотрите цели, которые вы поставили перед собой для текущего сеанса программирования. Можно ли считать эти цели по-прежнему реалистичными, или вы должны выбрать новые цели? Является ли то, чего вы пытаетесь достичь, невозможным? Если так, то каким образом это повлияет на всю вашу команду?

Дэйв Унгар (Dave Ungar) называет это Методологией душа (Shower Methodology). Если вы знаете, что писать, – пишите. Если вы не знаете, что писать, примите душ и стойте под ним до тех пор, пока не поймете, что нужно писать. Очень многие команды были бы более счастливыми, более продуктивными и пахли бы существенно лучше, если бы воспользовались этим советом.

TDD – это усовершенствование предложенной Унгаром методологии душа. Если вы знаете, что писать, пишите очевидную реализацию. Если вы не знаете, что писать, создайте поддельную реализацию. Если правильный дизайн по-прежнему не ясен, триангулируйте. Если вы по-прежнему не знаете, что писать, можете, наконец, принять душ.

На рис. 26.1 показана динамика процессов, связанных с перерывом. В процессе работы вы устаете. В результате внимание рассеивается, и вам становится сложнее заметить, что вы устали. Поэтому вы продолжаете работать и устаете еще больше.

Чтобы разорвать этот замкнутый круг, необходимо добавить дополнительный внешний элемент.

• В масштабе нескольких часов держите бутылку с водой рядом с вашей клавиатурой и время от времени прихлебывайте из нее. Благодаря этому естественная физиология будет подсказывать вам, когда и зачем необходимо сделать короткий перерыв в работе.

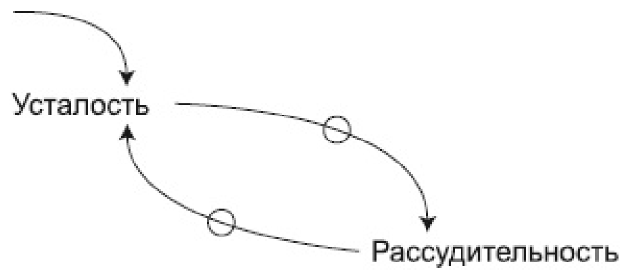

Рис. 26.1. Усталость негативно влияет на рассудительность, которая негативно влияет на усталость

• В масштабе дня вы должны хорошо отдохнуть после завершения рабочего времени.

• В масштабе недели вы отдыхаете в выходные дни. Отдых наполнит вас силами и идеями, благодаря чему вы сможете приступать к новой рабочей неделе. (Моя жена утверждает, что самые лучшие идеи возникают у меня вечером в пятницу.)

• В масштабе года вы получаете отпуск, что позволяет вам полностью освежиться. Французы подходят к этому вопросу очень правильно – двух последовательных недель отпуска недостаточно. В течение первой недели вы сбрасываете с себя рабочее напряжение, а в течение второй недели подсознательно готовите себя к работе. Поэтому, чтобы хорошо отдохнуть и эффективно работать в течение всего следующего года, требуется три, а лучше четыре недели отдыха.

Существует обратная сторона данного шаблона. Иногда, если перед вами стоит сложная проблема, требуется, наоборот, поднажать, поднапрячься и потратить дополнительное время и усилия, чтобы решить ее. Однако большинство программистов инфицировано духом саморазрушения: «Я угроблю свое здоровье, отрекусь от своей семьи и даже выпрыгну из окна, лишь бы этот код заработал». Поэтому я не буду давать здесь каких-либо советов. Если вы чувствуете, что у вас развивается болезненное пристрастие к кофе, наверное, вам не стоит делать слишком частых перерывов. В крайнем случае, просто пройдитесь.

Начать сначала (Do over)

Что делать, если вы зашли в тупик? Выкиньте код и начните работу сначала.

Вы заблудились. Вы решили передохнуть. Вымыли руки. Еще раз попытались вспомнить дальнейший путь. И все равно вы заблудились. Код, который выглядел так неплохо всего час назад, теперь выглядит запутанно и непонятно, одним словом, отвратительно. Вы не можете представить себе, как заставить работать следующий тест, а впереди у вас еще 20 тестов, которые необходимо реализовать.

Подобное случалось со мной несколько раз, пока я писал эту книгу. Код получался слишком кривым. «Но я должен закончить книгу. Мои дети хотят есть, а сборщики налогов стучаться в мою дверь.» У меня возникает желание выпрямить код настолько, чтобы можно было продолжать двигаться вперед. Однако на самом деле в большинстве случаев продуктивнее отдохнуть немного и начать все заново. Однажды я был вынужден выкинуть 25 страниц рукописи потому, что она была основана на очевидно глупом программистском решении.

Хорошую историю на эту тему рассказал мне Тим Макиннон (Tim Mackinnon). Однажды он проводил собеседование с потенциальным новым сотрудником. Чтобы оценить уровень его мастерства, он предложил ему программировать в паре в течение часа. К концу этого часа они реализовали несколько новых тестов и провели несколько сеансов рефакторинга. Однако это был конец рабочего дня, они оба чувствовали себя усталыми, поэтому решили полностью убрать из системы результаты своей работы.

Если вы программируете в паре, смена партнера – это хороший повод отказаться от плохого кода и начать решение задачи с начала. Вы пытаетесь объяснить смысл запутанного кода, над которым работали до этого, и вдруг ваш партнер, совершенно не связанный с ошибками, которые вы допустили, берет у вас клавиатуру и говорит: «Я ужасно извиняюсь за мою тупость, но что, если мы попробуем начать по-другому?»

Дешевый стол, хорошие кресла (Cheap Desk, Nice Chair)

В какой физической обстановке следует использовать TDD? Используйте удобное, комфортное кресло. На всей остальной мебели можно сэкономить.

Вы не сможете хорошо программировать, если ваша спина будет болеть. К сожалению, организации, которые выкладывают по $100 000 в месяц за работу команды программистов, как правило, отказываются тратить $10 000 на покупку хороших кресел.

Я предлагаю использовать дешевые, самые примитивные и ужасные на вид столы для установки компьютеров, но купить самые лучшие кресла, которые я только смогу найти. Дешевых столов можно купить столько, сколько нужно, значит, я получаю в свое распоряжение достаточное количество рабочего места и могу легко его увеличить. При этом я чувствую себя комфортно за компьютером, моя спина не устает.

Если вы программируете в паре, позаботьтесь о том, чтобы вам было удобно. Расчистите пространство на столе, чтобы вам было удобно передавать клавиатуру из рук в руки. Когда я работаю наставником, я люблю выполнять один простой прием: незаметно подходить со спины к программирующей паре и ненавязчиво поправлять клавиатуру так, чтобы она располагалась удобно по отношению к человеку, который с ней работает.

Манфред Лэндж (Manfred Lange) считает, что аккуратное распределение ресурсов необходимо выполнить также в отношении компьютерного аппаратного обеспечения. Рекомендуется использовать дешевые/медленные/старые компьютеры для индивидуальной электронной почты и работы с Интернетом, но зато приобрести самые современные и самые быстрые компьютеры для разработки.

27. Шаблоны тестирования

В данной главе более подробно описываются методики разработки тестов.

Дочерний тест (Child Test)

Как заставить работать тест, который оказался слишком большим? Напишите тест меньшего размера, который представляет собой неработающую часть большого теста. Добейтесь успешного выполнения маленького теста. Заново напишите большой тест.

Ритм красный – зеленый – рефакторинг чрезвычайно важен для достижения успеха. Не бойтесь потратить дополнительные усилия, чтобы поддерживать этот ритм, – дополнительные усилия с лихвой окупят себя. Я достаточно часто попадаю в подобную ситуацию: сначала записываю тест, а потом оказывается, что для его реализации требуется выполнить не одно, а несколько изменений. Я неожиданно оказываюсь на большом расстоянии от зеленой полосы. Даже десять минут с красной полосой заставляют меня нервничать.

Когда тест оказывается слишком большим, я, прежде всего, пытаюсь усвоить урок. Почему тест оказался слишком большим? Что надо было сделать иначе, чтобы тест получился меньше по размеру?

Покончив с размышлениями, я удаляю изначальный тест и начинаю заново. «Похоже, заставить все эти три вещи работать одновременно – это слишком сложная задача. Однако если я вначале добьюсь успешной работы A, B и C, мне не составит труда заставить работать всю эту штуку целиком.» Иногда я действительно удаляю тест, однако в некоторых случаях я просто изменяю его имя так, чтобы оно начиналось на x, – в этом случае тестовый метод не будет выполнен. (Скажу вам по секрету, что иногда я вообще не трогаю изначальный тест. Да, да! Только т-с-с-с! Никому об этом не рассказывайте! Слава богу, в большинстве подобных случаев мне удается быстро заставить работать дочерний тест. Однако получается, что я в течение пары минут живу вместе с двумя сломанными тестами. Возможно, когда я так поступаю, я совершаю ошибку. Этот пережиток сохранился у меня с тех времен, когда я выполнял тестирование после завершения разработки или вообще не тестировал свой код.)

Попробуйте оба варианта. Прислушайтесь к своим ощущениям. Если у вас есть два сломанных теста, вы, как правило, начинаете программировать иначе. Делайте выводы.

Поддельный объект (Mock Object)

Как выполнять тестирование объекта, который базируется на сложном и тяжеловесном ресурсе? Создайте поддельную версию ресурса, которая будет возвращать константы.

Использование поддельных объектов – это тема для отдельной книги. Существует огромное количество материала, посвященного поддельным объектам[16]. Здесь я попытаюсь очень коротко познакомить читателей с этой концепцией.

Классическим примером является база данных. Чтобы запустить базу данных, требуется значительное время, поддержка чистоты базы данных требует дополнительных затрат, кроме того, если база данных располагается на удаленном сервере, ваши тесты будут связаны с конкретным физическим местоположением в сети. Наконец, база данных является емким источником ошибок разработки.

Чтобы уменьшить количество проблем, рекомендуется в процессе тестирования отказаться от работы непосредственно с базой данных. Большинство тестов пишется в отношении объекта, который функционирует подобно базе данных, однако располагается в оперативной памяти.

```
public void testOrderLookup() {
Database db = new MockDatabase();
db.expectQuery("select order_no from Order where cust_no is 123");
db.returnResult(new String[] {"Order 2","Order 3"});
.
}
```

Если объект MockDatabase не принимает ожидаемого запроса, он генерирует исключение. Если запрос корректен, объект возвращает нечто, напоминающее результирующий набор данных, состоящий из нескольких постоянных строк.

Помимо высокой производительности и надежности поддельные объекты обладают еще одним преимуществом: читабельностью. Если вы

работаете с реальной базой данных, заполненной реальными данными, в результате обработки запроса вы можете получить ответ, состоящий из 14 строк. Возможно, вам будет нелегко понять, откуда взялось число 14 и в чем, собственно, состоит смысл теста.

Если вы хотите воспользоваться поддельными объектами, не следует хранить тяжеловесные ресурсы в глобальных переменных (даже если они замаскированы с использованием шаблона «Одиночка» (Singleton)). Если вы так поступите, вам придется вначале настроить глобальный поддельный объект, затем выполнить тест, а затем позаботиться о том, чтобы вернуть поддельный объект в исходное состояние.

В свое время я очень строго следил за выполнением этого правила. Мы вместе с Массимо Арнольди (Massimo Arnoldi) разрабатывали код, который взаимодействовал с набором курсов обмена валют, хранящимся в глобальной переменной. Для разных тестов требовалось использовать разные наборы данных, и в некоторых случаях курсы обмена валют должны были быть разными для разных тестов. Вначале мы пытались использовать для тестирования глобальную переменную, однако в конце концов нас это утомило, и однажды утром (смелые решения, как правило, приходят ко мне по утрам) мы решили передавать объект Exchange (в котором хранились курсы обмена) в качестве параметра везде, где это было необходимо. Мы думали, что нам придется модифицировать сотни методов. Однако дело кончилось тем, что мы добавили дополнительный параметр в десять или пятнадцать методов и по ходу дела подчистили другие аспекты дизайна.

Шаблон поддельных объектов заставляет тщательно следить за видимостью объектов, снижая взаимозависимости между ними. Поддельные объекты добавляют в проект некоторый риск, – что, если поддельный объект ведет себя не так, как реальный объект? Чтобы снизить этот риск, вы можете разработать специальный набор тестов для поддельных объектов, которые должны быть выполнены в отношении реального объекта, чтобы убедиться в том, что имитация достаточно близка к оригиналу.

Самошунтирование (Self Shunt)

Как можно убедиться в том, что один объект корректно взаимодействует с другим? Заставьте тестируемый объект взаимодействовать не с целевым объектом, а с вашим тестом.

Предположим, что вы хотите динамически обновлять зеленую полосу, отображаемую в рамках тестируемого пользовательского интерфейса. Если мы сможем подключить наш объект к объекту TestResult, значит, мы сможем получать оповещения о запуске теста, о том, что тест не сработал, а также о том, что весь набор тестов начал работу или, наоборот, завершил работу. Каждый раз, получив оповещение о запуске теста, мы можем выполнить обновление интерфейса. Вот соответствующий тест:

ResultListenerTest
```
def testNotification(self):
result = TestResult()
listener = ResultListener()
result.addListener(listener)
WasRun("testMethod"). run(result)
assert 1 == listener.count
```

Тест нуждается в объекте, который подсчитывал бы количество оповещений:

ResultListener
```
class ResultListener:
def __init__(self):
self.count = 0
def startTest(self):
self.count = self.count + 1
```

Подождите-ка! Зачем нам нужен отдельный объект Listener? Все необходимые функции мы можем возложить на объект TestCase. В этом случае объект TestCase становится подобием поддельного объекта.

ResultListenerTest
```
def testNotification(self):
self.count = 0
result = TestResult()
result.addListener(self)
WasRun("testMethod"). run(result)
assert 1 == self.count
def startTest(self):
self.count = self.count + 1
```

Тесты, написанные с использованием шаблона «Самошунтирование» (Self Shunt), как правило, читаются лучше, чем тесты, написанные без него. Предыдущий тест является неплохим примером. Счетчик был равен 0, а затем стал равен 1. Вы можете проследить за последовательностью действий прямо в коде теста. Почему счетчик стал равен 1? Очевидно, кто-то обратился к методу startTest(). Где произошло обращение к методу startTest()? Это произошло в начале выполнения теста. Вторая версия теста использует два разных значения переменной count в одном месте, в то время как первая версия присваивает переменной count значение 0 в одном классе и проверяет эту переменную на равенство значению 1 в другом.

Возможно, при использовании шаблона «Самошунтирование» (Self Shunt) вам потребуется применить шаблон рефакторинга «Выделение интерфейса» (Extract Interface), чтобы получить интерфейс, который должен быть реализован вашим тестом. Вы должны сами определить, что проще: выделение интерфейса или тестирование существующего объекта в рамках концепции «черный ящик». Однако я часто замечал, что интерфейсы, выделенные при выполнении самошунтирования, в дальнейшем, как правило, оказываются полезными для решения других задач.

В результате использования шаблона «Самошунтирование» (Self Shunt) вы можете наблюдать, как тесты в языке Java обрастают разнообразными причудливыми интерфейсами. В языках с оптимистической типизацией класс теста обязан реализовать только те операции интерфейса, которые действительно используются в процессе выполнения теста. Однако в Java вы обязаны реализовать абсолютно все операции интерфейса несмотря на то, что некоторые из них будут пустыми. По этой причине интерфейсы следует делать как можно менее емкими. Реализация каждой операции должна либо возвращать значение, либо генерировать исключение – это зависит от того, каким образом вы хотите быть оповещены о том, что произошло нечто неожиданное.

Строка-журнал (Log String)

Как можно убедиться в том, что обращение к методам осуществляется в правильном порядке? Создайте строку, используйте ее в качестве журнала. При каждом обращении к методу добавляйте в строку-журнал некоторое символьное сообщение.

Данный прием продемонстрирован ранее, в главе 20, где мы тестировали порядок обращения к методам класса TestCase. В нашем распоряжении имеется шаблонный метод, который, как мы предполагаем, обращается к методу setUp(), затем к тестовому методу, а затем к методу tearDown(). Нам хотелось бы убедиться в том, что обращение к методам осуществляется в указанном порядке. Реализуем методы так, чтобы каждый из них добавлял свое имя в строку-журнал. Исходя из этого можно написать следующий тест:

```
def testTemplateMethod(self):
test = WasRun("testMethod")
result = TestResult()
test.run(result)
assert("setUp testMethod tearDown " == test.log)
```

Реализация тоже очень проста:

WasRun

```
def setUp(self):
self.log = "setUp "
def testMethod(self):
self.log = self.log + "testMethod "
def tearDown(self):
self.log = self.log + "tearDown "
```

Шаблон «Строка-журнал» (Log String) особенно полезен в случае, когда вы реализуете шаблон «Наблюдатель» (Observer) и желаете протестировать порядок поступления оповещений. Если вас прежде всего интересует, какие именно оповещения генерируются, однако порядок их поступления для вас не важен, вы можете создать множество строк, добавлять строки в множество при обращении к методам и в выражении assert использовать операцию сравнения множеств.

Шаблон «Строка-журнал» (Log String) хорошо сочетается с шаблоном «Самошунтирование» (Self Shunt). Объект-тест реализует методы шунтируемого интерфейса так, что каждый из них добавляет запись в строку-журнал, затем проверяется корректность этих записей.

Тестирование обработки ошибок (Crush Test Dummy)

Как можно протестировать работу кода, обращение к которому происходит в случае ошибки, возникновение которой маловероятно? Создайте специальный объект, который вместо реальной работы генерирует исключение.

Непротестированный код считается неработающим. Но что же тогда делать с кодом обработки ошибок? Надо ли его тестировать? Только в случае, если вы хотите, чтобы он работал.

Предположим, что мы хотим проверить, что происходит с нашим приложением в случае, если на диске не остается свободного места. Неужели для этого необходимо вручную создавать огромное количество файлов? Есть альтернатива. Мы можем *сымитировать* заполнение диска, фактически использовав шаблон «Подделка» (Fake It).

Вот наш тест для класса File:

```
private class FullFile extends File {
public FullFile(String path) {
super(path);
}
public boolean createNewFile() throws IOException {
throw new IOException();
}
}
```

Теперь мы можем написать тест *ожидаемого* исключения:

```
public void testFileSystemError() {
File f = new FullFile("foo");
try {
saveAs(f);
fail();
} catch (IOException e) {
}
}
```

Тест кода обработки ошибки напоминает шаблон «Поддельный объект» (Mock Object), однако в данном случае нам не надо подделывать весь объект. Для реализации этой методики удобно использовать анонимные внутренние классы языка Java. При этом вы можете переопределить только один необходимый вам метод. Сделать это можно

прямо внутри теста, благодаря чему код теста станет более понятным:

```
public void testFileSystemError() {
File f = new File("foo") {
public boolean createNewFile() throws IOException {
throw new IOException();
}
};
try {
saveAs(f);
fail();
} catch (IOException e) {
}
}
```

Сломанный тест (Broken Test)

Как следует завершить сеанс программирования, если вы программируете в одиночку? Оставьте последний тест неработающим.

Этому приему научил меня Ричард Гэбриел (Richard Gabriel). Вы заканчиваете сеанс на середине предложения. Когда вы возвращаетесь к работе, вы смотрите на начало предложения и вспоминаете, о чем вы думали, когда писали его. Вспомнив ход своих мыслей, вы завершаете предложение и продолжаете работу. Если по возвращении к работе вам не надо завершать никакого предложения, вы вынуждены потратить несколько минут, чтобы сначала просмотреть уже написанный код, изучить список задач, вспомнить, над чем вы собирались работать, затем попытаться восстановить прежнее состояние ваших мыслей и наконец приступить к работе.

Я использовал подобный трюк при работе над моими одиночными проектами и мне очень понравился эффект. В конце рабочего сеанса следует написать тест и запустить его, чтобы убедиться, что он не работает. Вернувшись к коду, вы сразу увидите место, откуда можно продолжить. Вы получаете хорошо заметную закладку, которая помогает вам вспомнить, о чем вы думали. Оставленный неработающим тест должен быть прост в реализации, благодаря чему вы сможете быстро выйти на прежний маршрут и продолжить движение к победе.

Сначала я думал, что оставленный недоделанным тест будет

действовать мне на нервы. Однако нет. Ведь вся программа в целом еще далека от завершения, один сломанный тест не сделает ее менее завершенной, и я отлично знаю об этом. Возможность быстро восстановить общий ход разработки спустя несколько недель простоя стоит того, чтобы немножко поступиться принципами и оставить работу, имея перед глазами красную полосу.

Чистый выпускаемый код (Clean Check-in)

Как следует завершить сеанс программирования, если вы программируете в составе команды? Все ваши тесты должны работать.

Я противоречу сам себе? Да.

Бубба Уитман (Bubba Whitman), брат Уолта – портового грузчика

Когда вы работаете над кодом не один, а вместе с вашими коллегами, ситуация полностью меняется. Когда вы возвращаетесь к работе над кодом, над которым помимо вас работают еще несколько человек, вы не можете сказать точно, что именно произошло с кодом с того момента, как вы видели его последний раз. По этой причине, прежде чем выпускать разрабатываемый вами код, убедитесь, что все тесты выполняются успешно. (Это напоминает правило изоляции тестов, в соответствии с которым каждый тест гарантированно оставляет систему в хорошем известном состоянии. Это правило можно применить и в отношении программистов, которые поочередно интегрируют свой код в систему, каждый раз проверяя, остается система в хорошем состоянии или нет.)

Набор тестов, которые вы запускаете в ходе интеграции, может быть существенно объемнее набора тестов, который вы запускаете каждую минуту в процессе разработки функциональности. (До того времени, пока это не станет слишком утомительным, я рекомендую вам постоянно запускать полный набор тестов.) Что делать, если при попытке интеграции вы обнаружили, что один тест из полного набора завершается неудачей?

Самое простое правило: отбросьте проделанную работу и начните все заново. Сломанный сигнализирует, что вы не обладаете достаточным запасом знаний, чтобы запрограммировать то, над чем вы работаете. Если команда будет действовать в соответствии с этим правилом, у ее членов

появится тенденция выполнять интеграцию более часто, так как тот, кто успеет приступить к интеграции раньше других, не рискует потерять проделанную работу. По всей видимости, частая интеграция и проверка – это неплохая практика.

Существует другой, менее строгий подход: вы можете исправить дефект и попробовать снова выполнить интеграцию. Однако не забывайте, что вы не должны слишком долго занимать интеграционные ресурсы: если вы не можете решить проблему в течение нескольких минут, откажитесь от идеи интеграции и начните работу заново. Об этом можно не говорить, но я все-таки скажу, что комментирование одного теста с целью заставить работать весь набор строго запрещается и должно приводить к самым серьезным карательным санкциям.

28. Шаблоны зеленой полосы

Когда у вас есть сломанный тест, вы должны заставить его работать. Если вы рассматриваете красную полосу как состояние, из которого следует выйти как можно быстрее, вы должны овладеть приемами быстрого получения зеленой полосы. Используйте следующие шаблоны, чтобы заставить ваш тест выполниться (даже если полученный в результате этого код не просуществует и часа).

Подделка (Fake It)

Если у вас есть тест, завершающийся неудачей, какой должна быть самая первая реализация? Сделайте так, чтобы тестируемый метод возвращал константу. После того как тест начал работать, постепенно трансформируйте константу в выражение с использованием переменных.

Пример использования этого подхода продемонстрирован в ходе разработки нашей реализации xUnit. Вначале мы использовали строковую константу:

```
return «1 run, 0 failed»
```

Затем эта строка была преобразована в выражение:

```
return «%d run, 0 failed» % self.runCount
```

Однако этим дело не кончилось. В конце мы получили выражение:

```
return «%d run, %d failed» % (self.runCount, self failureCount)
```

Шаблон «Подделка» (Fake It) напоминает страховочную веревку, которая соединяет вас с верхней точкой маршрута, когда вы карабкаетесь по скале. Пока что вы еще не забрались на самый верх (тест на месте и работает, но тестируемый код некорректен). Однако в любой точке маршрута вы держитесь за веревку и знаете, что когда достигнете самого верха, то будете в безопасности (тест работает в ходе рефакторинга, а также после получения окончательного кода).

Шаблон «Подделка» (Fake It) многим может показаться совершенно бесполезным. Зачем писать код, который абсолютно точно придется заменить другим? Дело в том, что иметь хоть какой-то работающий код – это лучше, чем вообще не иметь работающего кода, в особенности если у вас есть тесты, которые могут доказать работоспособность кода. Петер Хансен (Peter Hansen) рассказал мне следующую историю:

Буквально вчера два новичка в области TDD – мой партнер и я – решили в точности следовать букве закона. То есть мы написали тест, а затем написали самый простой, но совершенно бесполезный код, который обеспечивал срабатывание теста. Пока мы писали этот код, мы обнаружили, что тест написан неправильно.

Каким образом поддельная реализация подсказала им, что написанный ими тест некорректен? Я понятия не имею, однако я счастлив, что они вовремя обнаружили это. Быть может, если они не воспользовались бы поддельной реализацией, они пошли бы по ложному пути. Возможно, исправление связанных с этим ошибок обошлось бы им дороже.

При использовании шаблона «Подделка» (Fake It) возникает как минимум два положительных эффекта:

Психологический. Если перед вами зеленая полоса, вы чувствуете себя совершенно иначе, чем когда перед вами красная полоса. Когда полоса зеленая, вы знаете, на чем стоите. Вы можете смело и уверенно приступать к рефакторингу.

Контроль над объемом работы. Программисты привыкли пытаться предвидеть появление в будущем самых разнообразных проблем. Если вы начинаете с конкретного примера и затем осуществляете обобщение кода, это помогает вам избавиться от лишних опасений. Вы можете сконцентрироваться на решении конкретной проблемы и поэтому выполнить работу лучше. При переходе к следующему тесту вы опять же концентрируетесь на нем, так как знаете, что предыдущий тест гарантированно работает.

Нарушает ли шаблон «Подделка» (Fake It) правило о том, что не следует писать код, который вам не потребуется? Я так не думаю, ведь на этапе рефакторинга вы удаляете дублирование данных между тестом и тестируемым кодом. Допустим, я написал[17]:

```
assertEquals(new MyDate(«28.2.02»), new MyDate(«1.3.02»).yesterday());
```

MyDate

```
public MyDate yesterday() {
return new MyDate("28.2.02");
}
```

Между тестом и кодом существует дублирование. Попробуем исправить ситуацию:

MyDate

```
public MyDate yesterday() {
return new MyDate(new MyDate("1.3.02"). days()-1);
}
```

Однако дублирование по-прежнему присутствует. Чтобы избавиться от него, заменяем MyDate(«1.3.02») на this (в моем тесте эти значения равны). Получается:

MyDate

```
public MyDate yesterday() {
return new MyDate(this.days()-1);
}
```

Однако увидеть возможность подобных подстановок с первого взгляда удается далеко не всегда и далеко не всем, поэтому для пущей ясности вы можете использовать триангуляцию, по крайней мере до тех пор, пока вам не надоест. Когда вам надоест, вы чаще будете пользоваться шаблоном «Подделка» (Fake It) или «Очевидная реализация» (Obvious Implementation).

Триангуляция (Triangulate)

Какой самый консервативный способ позволяет формировать абстракцию при помощи тестов? Делайте код абстрактным только в случае, если у вас есть два или более примера.

Рассмотрим пример. Предположим, мы хотим написать функцию, которая возвращает сумму двух целых чисел. Мы пишем:

```
public void testSum() {
assertEquals(4, plus(3, 1));
```

```
}

private int plus(int augend, int addend) {
return 4;
}
```

Чтобы получить представление о правильном дизайне, мы добавляем еще один пример:

```
public void testSum() {
assertEquals(4, plus(3, 1));
assertEquals(7, plus(3, 4));
}
```

Теперь, когда у нас есть еще один пример, мы можем сделать реализацию метода plus() абстрактной:

```
private int plus(int augend, int addend) {
return augend + addend;
}
```

Триангуляция выглядит привлекательно, так как правила ее выполнения вполне понятны. Правила для шаблона «Подделка» (Fake It) основаны на ощущении дублирования кода между тестом и поддельным кодом. Это ощущение может быть субъективным, поэтому правила выглядят несколько туманными. Несмотря на то, что они кажутся простыми, правила триангуляции создают замкнутый цикл. После того как мы написали два выражения assert и сформировали абстрактную корректную реализацию метода plus(), мы можем уничтожить одно из выражений assert, так как теперь оно является избыточным. А сделав это, мы сможем упростить реализацию plus(), чтобы этот метод возвращал константу. После этого нам надо будет снова добавить выражение assert.

Я использую триангуляцию только в случае, если я действительно не уверен, какая абстракция является корректной. В других ситуациях я предпочитаю использовать шаблон «Подделка» (Fake It) или «Очевидная реализация» (Obvious Implementation).

Очевидная реализация (Obvious Implementation)

Как реализоать простую операцию? Просто реализуйте ее.

Шаблоны «Подделка» (Fake It) и «Триангуляция» (Triangulate) позволяют вам двигаться маленькими шажками. Но иногда вы абсолютно уверены в том, как можно корректно реализовать операцию. Вперед! Пишите то, что вы думаете. Например, должен ли я использовать шаблон «Подделка» (Fake It) для реализации чего-либо столь же простого, как метод plus()? Как правило, нет. Обычно для таких простых методов я просто пишу очевидную реализацию. Если при этом передо мной неожиданно появляется красная полоса за красной полосой, я перехожу на более короткий шаг.

В шаблонах «Подделка» (Fake It) и «Триангуляция» (Triangulate) не существует никакой особенной добродетели. Если вы знаете, что писать, и если это получится достаточно быстро, то смело пишите готовый код. Однако помните, что, используя только очевидную реализацию, вы требуете от себя совершенства[18]. С психологической точки зрения это может быть разрушительный ход. Что, если написанное вами на самом деле не является самым простым изменением, которое заставляет тест работать? Что, если ваш партнер покажет вам еще более простой вариант кода? Вы проиграли! Ваш мир рухнул! Вы в ступоре.

Известный слоган гласит: «чистый код, который работает». Если вы будете решать проблему «чистый код» одновременно с проблемой «который работает», для вас это может оказаться слишком много. Как только вы поймете это, вернитесь обратно к решению проблемы «который работает» и только после этого принимайтесь за решение проблемы «чистый код».

При использовании шаблона «Очевидная реализация» (Obvious Implementation) следите за тем, насколько часто вы сталкиваетесь с красной полосой. Часто приходится попадать в ловушку: я записываю очевидную реализацию, но она не работает. Но теперь я точно знаю, что именно я должен написать. Поэтому я вношу в код изменения. Однако тест по-прежнему не работает. Но теперь-то я уж точно знаю… Это часто случается при возникновении ошибок типа «индекс отличается на единицу» и «положительные/отрицательные числа».

Прежде всего вы должны следить за соблюдением ритма красный – зеленый – рефакторинг. Очевидная реализация – это вторая передача. Будьте готовы снизить скорость, если ваш мозг начинает выписывать чеки, которые не могут быть оплачены вашими пальцами.

От одного ко многим (One to Many)

Как реализовать операцию с коллекцией объектов? Сначала реализуйте эту операцию, манипулирующую единственным объектом, затем модернизируйте ее для работы с коллекцией таких объектов.

Например, предположим, что мы разрабатываем функцию, которая суммирует массив чисел. Мы можем начать с одного числа:

```
public void testSum() {
assertEquals(5, sum(5));
}
private int sum(int value) {
return value;
}
```

(Я добавил метод sum() в класс TestCase, чтобы не создавать новый класс ради одного метода.)

Теперь мы хотим протестировать sum(new int[] {5, 7}). Для начала добавим в метод sum() параметр, соответствующий массиву целых чисел:

```
public void testSum() {
assertEquals(5, sum(5, new int[] {5}));
}
private int sum(int value, int[] values) {
return value;
}
```

Этот этап можно рассматривать как пример применения шаблона «Изоляция изменения» (Isolate Change). После того как мы добавили параметр, мы можем менять реализацию, не затрагивая код теста.

Теперь мы можем использовать коллекцию вместо единственного значения:

```
private int sum(int value, int[] values) {
int sum = 0;
for (int i = 0; i < values.length; i++)
sum += values[i];
return sum;
```

```
}
```

Теперь можно удалить неиспользуемый параметр:

```
public void testSum() {
assertEquals(5, sum(new int[] {5}));
}
private int sum(int[] values) {
int sum = 0;
for (int i = 0; i < values.length; i++)
sum += values[i];
return sum;
}
```

Предыдущий шаг – это тоже демонстрация шаблона «Изоляция изменения» (Isolate Change). Мы изменили код и в результате можем менять тест, не затрагивая код. Теперь мы можем расширить тест, как планировали:

```
public void testSum() {
assertEquals(12, sum(new int[] {5, 7}));
}
```

29. Шаблоны xUnit

В этой главе рассматриваются шаблоны, предназначенные для использования при работе с xUnit.

Проверка утверждений

Как убедиться в том, что тест работает правильно? Напишите логическое выражение, которое автоматически подтвердит ваше мнение о том, что код работает.

Если мы хотим сделать тесты полностью автоматическими, значит, абсолютно все предположения о работе тестируемого кода необходимо превратить в тесты, при этом результат выполнения этих тестов должен говорить нам, работает код корректно или нет. Проще говоря, мы должны обладать возможностью щелкнуть на кнопке и через короткое время узнать, работает код корректно или нет. Отсюда следует, что

• в результате выполнения теста должно получиться логическое значение: «истина» (True) указывает, что все в порядке, а «ложь» – что произошло нечто непредвиденное;

• проверка результата каждого теста выполняется компьютером автоматически при помощи какой-либо разновидности оператора assert().

Мне приходилось видеть выражения наподобие assertTrue(rectangle.area()!= 0). Чтобы тест выполнился успешно, метод area() должен вернуть любое ненулевое значение – это не очень полезный тест. Делайте тесты более конкретными. Если площадь прямоугольника должна быть равна 50, так и пишите: assertTrue(rectangle.area() == 50). Во многих реализациях xUnit присутствует специальное выражение assert() для тестирования равенства (эквивалентности). Отличительная его черта состоит в том, что вместо одного логического параметра выражение assertEquals() принимает два произвольных объекта и пытается определить, являются ли они эквивалентными. Преимущество состоит в том, что в случае неудачи выражение assertEquals() сгенерирует более информативное сообщение с указанием двух несовпадающих значений. Ожидаемое значение, как правило, указывается первым. Например, предыдущее выражение в среде JUnit можно переписать следующим образом: assertEquals(50, rectangle.area()).

Думать об объектах, как о черных ящиках, достаточно тяжело. Представим, что у нас есть объект Contract, состояние которого хранится в поле status и может быть экземпляром класса Offered или Running. В этом случае можно написать тест исходя из предполагаемой реализации:

```
Contract contract = new Contract(); // по умолчанию состояние Offered
contract.begin(); // состояние меняется на Running
assertEquals(Running.class, contract.status.class);
```

Этот тест слишком сильно зависит от текущей реализации объекта status. Однако тест должен завершаться успешно, даже если поле status станет логическим значением. Может быть, когда status меняется на Running, можно узнать дату начала работы над контрактом:

```
assertEquals(…, contract.startDate()); // генерирует исключение, если
// status является экземпляром Offered
```

Я признаю, что пытаюсь плыть против течения, когда настаиваю на том, что все тесты должны быть написаны только с использованием общедоступного (public) протокола. Существует специальный пакет JXUnit, который является расширением JUnit и позволяет тестировать значения переменных, даже тех, которые объявлены как закрытые.

Желание протестировать объект в рамках концепции белого ящика – это не проблема тестирования, это проблема проектирования. Каждый раз, когда у меня возникает желание протестировать значение переменной-члена, чтобы убедиться в работоспособности кода, я получаю возможность улучшить дизайн системы. Если я забываю о своих опасениях и просто проверяю значение переменной, я теряю такую возможность. Иначе говоря, если идея об улучшении дизайна не приходит мне в голову, ничего не поделаешь. Я проверяю значение переменной, смахиваю непрошеную слезу, вношу соответствующую отметку в список задач и продолжаю двигаться вперед, надеясь, что наступит день, когда смогу найти подходящее решение.

Самая первая версия xUnit для Smalltalk (под названием SUnit) обладала очень простыми выражениями assert. Если одно из выражений терпело неудачу, автоматически открывалось окно отладчика, вы исправляли код и продолжали работу. Среда разработки Java не настолько совершенна, к тому же построение приложений на Java часто выполняется в пакетном режиме, поэтому имеет смысл добавлять в выражение assert()

дополнительную информацию о проверяемом условии. Чтобы в случае неудачи выражения assert() можно было вывести на экран дополнительную информацию.

В JUnit это реализуется при помощи необязательного первого параметра[19]. Например, если вы напишете assertTrue(«Должно быть True», false) и тест не сработает, то вы увидите на экране приблизительно следующее сообщение: Assertion failed: Должно быть True. Обычно подобного сообщения достаточно, чтобы направить вас напрямую к источнику ошибки в коде. В некоторых группах разработчиков действует жесткое правило, что все выражения assert() должны снабжаться подобными информационными сообщениями. Попробуйте оба варианта и самостоятельно определите, окупаются ли для вас затраты, связанные с информационными сообщениями.

Фикстура[20] (Fixture)

Как создаются общие объекты, которые используются в нескольких тестах? Конвертируйте локальные переменные из тестов в переменные-члены класса TestCase. Переопределите метод setUp() и инициализируйте в нем эти переменные (то есть выполните создание всех необходимых объектов).

Если мы привыкли удалять дублирование из функционального (тестируемого) кода, должны ли мы удалять его из тестирующего кода? Может быть.

Существует проблема: зачастую вам приходится писать больше кода для того, чтобы установить объекты, используемые тестируемым методом, в интересующее вас состояние. Код, инициализирующий объекты, часто оказывается одинаковым для нескольких тестов. Такие объекты называются *фикстурой* теста (используется также английский термин *scaffolding* – *строительные леса, подмостки*). Дублирование подобного кода – это плохо. Вот две основные причины:

• написание подобного кода требует дополнительного времени, даже если мы просто копируем блоки текста через буфер обмена. Но наша задача – добиться того, чтобы написание тестов занимало как можно меньше времени;

• если приходится вручную менять интерфейс, перед нами встает необходимость изменять его в нескольких разных тестах (именно этого

всегда следует ожидать от дублирования).

Однако дублирование кода инициализации объектов обладает также некоторыми преимуществами. Если код инициализации располагается непосредственно рядом с тестирующими выражениями assert(), весь код теста можно прочитать от начала и до конца. Если мы выделили код инициализации в отдельный метод, нам приходится помнить о том, что этот метод вызывается, нам приходится вспоминать, как именно выглядят объекты, и только вспомнив все это, мы можем написать остальную часть теста.

Среда xUnit поддерживает оба стиля написания тестов. Если вы думаете, что читателям будет сложно вспомнить объекты фикстуры, вы можете разместить код создания фикстуры непосредственно в теле теста. Однако вы также можете переместить этот код в метод с названием setUp(). В этом методе вы можете создать все объекты, которые будут использоваться в тестовых методах.

Далее приводится пример, который слишком прост, чтобы мотивировать выделение общего кода фикстуры, но зато достаточно короток, чтобы поместиться в данной книге. Мы можем написать:

EmptyRectangleTest

```
public void testEmpty() {
Rectangle empty = new Rectangle(0,0,0,0);
assertTrue(empty.isEmpty());
}

public void testWidth() {
Rectangle empty = new Rectangle(0,0,0,0);
assertEquals(0.0, empty.getWidth(), 0.0);
}
```

(Помимо прочего здесь также демонстрируется версия assertEquals() для чисел с плавающей точкой, которая принимает третий параметр – точность сравнения.) Мы можем избавиться от дублирования, написав:

EmptyRectangleTest

```
private Rectangle empty;

public void setUp() {
empty = new Rectangle(0,0,0,0);
```

```
}

public void testEmpty() {
assertTrue(empty.isEmpty());
}
public void testWidth() {
assertEquals(0.0, empty.getWidth(), 0.0);
}
```

Общий код выделен в виде отдельного метода. Среда xUnit гарантирует, что метод setUp() объекта TestCase будет обязательно вызван перед обращением к любому тестовому методу этого объекта. Теперь тестовые методы выглядят проще, однако, прежде чем понять их смысл, мы должны вспомнить о существовании метода setUp() и уточнить, что происходит внутри этого метода.

Какой из этих двух стилей предпочтительней? Попробуйте использовать каждый из них. Я фактически всегда выделяю общий код фикстуры и перемещаю его в метод setUp(), однако у меня хорошая память. Те, кто читает мои тесты, часто жалуются, что им приходится вспоминать слишком о многом. Значит, возможно, мне следует выделять меньшей объем кода, чтобы сделать тесты более понятными.

Взаимоотношения между подклассами класса TestCase и экземплярами этих подклассов являются наиболее запутанной стороной инфраструктуры xUnit. Каждый новый тип фикстуры требует создания нового подкласса класса TestCase. Каждая новая фикстура создается внутри экземпляра подкласса, используется один раз, а затем уничтожается.

В предыдущем примере, если мы хотим написать тесты для непустого прямоугольника (Rectangle), нам придется создать новый класс, который можно назвать, например, NormalRectangleTest. У этого класса будет свой собственный метод setUp(), в котором будет создан новый экземпляр Rectangle, необходимый ему для тестирования. Этот экземпляр Rectangle будет соответствовать непустому прямоугольнику. В общем случае, если я хочу использовать несколько отличающуюся фикстуру, я создаю новый подкласс класса TestCase.

Это означает, что не существует прямого простого соответствия между классами тестов и функциональными (тестируемыми) классами. Иногда одна фикстура используется для тестирования нескольких классов (подобное случается нечасто). Иногда для тестирования единственного функционального класса требуется создать две или три фикстуры. На

практике в большинстве случаев получается, что количество классов тестов приблизительно совпадает с количеством функциональных классов. Однако это происходит вовсе не потому, что для каждого функционального класса вы создаете один-единственный класс теста.

Внешняя фикстура (External Fixture)

Как осуществляется освобождение внешних ресурсов в фикстуре? Переопределите метод tearDown() и освободите в нем ресурсы, выделенные в ходе создания фикстуры.

Помните, что каждый тест должен оставить рабочую среду в том же состоянии, в котором она была до того, как тест начал работу. Например, если внутри теста вы открываете файл, вы должны позаботиться о том, чтобы закрыть его перед тем, как тест завершит работу. Вы можете написать:

```
testMethod(self):
file = File("foobar"). open()
try:
…âûïîëíèòü òåñò…
finally:
file.close()
```

Если файл используется в нескольких тестах, вы можете сделать его частью общей фикстуры:

```
setUp(self):
self.file = File("foobar"). open()
testMethod(self):
try:
…выполнить тест…
finally:
self.file.close()
```

Во-первых, возникает неприятное дублирование выражений finally – это означает, что мы упустили что-то в дизайне. Во-вторых, при написании подобного метода можно легко допустить ошибку, например забыть добавить ключевое слово finally или вообще забыть о необходимости

закрытия файла. Наконец, в этом тесте существует три сбивающих с толку строки – try, finally и сама команда close, – эти выражения не относятся непосредственно к процедуре тестирования.

Инфраструктура xUnit гарантирует вызов метода под названием tearDown() после выполнения тестового метода. Метод tearDown() будет вызван вне зависимости от того, что случится внутри тестового метода (однако следует иметь в виду, что, если сбой произойдет в ходе выполнения метода setUp(), метод tearDown() вызываться не будет). Мы можем преобразовать предыдущий тест следующим образом:

```
setUp(self):
self.file = File("foobar"). open()
testMethod(self):
…выполнить тест…
tearDown(self):
self.file.close()
```

Тестовый метод (Test Method)

Что такое единичный тест? Это метод, имя которого начинается с префикса test.

В процессе разработки вам придется иметь дело с сотнями, а может быть, и тысячами тестов, как можно уследить за всеми этими тестами?

Языки объектно-ориентированного программирования обеспечивают трехуровневую организацию исходного кода:

- модуль (в языке Java – *пакет*, по-английски, *package*);
- класс;
- метод.

Если мы пишем тесты как обычный исходный код, мы должны найти способ организации тестов с использованием элементов этой структуры. Если мы используем классы для представления фикстур, значит, методы этих классов являются естественным местом размещения тестирующего кода. Все тесты, использующие некоторую фикстуру, должны быть методами одного класса. Тесты, работающие с другой фикстурой, должны располагаться в другом классе.

В xUnit используется соглашение, в соответствии с которым имя тестового метода должно начинаться с префикса test. Специальные инструменты могут автоматически производить поиск таких методов и

создавать из них наборы тестов (TestSuite). Остальная часть имени теста должна информировать будущего, ни о чем не ведающего читателя, зачем написан данный тест. Например, в наборе тестов, созданных при разработке инфраструктуры JUnit, можно обнаружить тест с именем testAssertPosInfinityNotEqualsNeglnfinity. Я не помню, чтобы я писал этот тест, однако, исходя из имени, могу предположить, что в какой то момент разработки было обнаружено, что код метода assert() инфраструктуры JUnit для чисел с плавающей точкой не делал различия между положительной и отрицательной бесконечностью. Использовав тест, я могу быстро найти код JUnit, осуществляющий сравнение чисел с плавающей точкой, и посмотреть, как осуществляется обработка положительной и отрицательной бесконечности. (На самом деле код выглядит не идеально – для поддержки бесконечности используется условный оператор).

Код тестового метода должен легко читаться и быть максимально прямолинейным. Если вы разрабатываете тест и видите, что его код становится слишком длинным, попробуйте поиграть в «детские шажки». Цель игры – написать самый маленький тестовый метод, который представляет собой реальный прогресс в направлении вашей конечной цели. Размер в три строки, судя по всему, является минимальным размером (если, конечно, вы не хотите делать тест намеренно бессмысленным). И постоянно помните о том, что вы пишете тесты для людей, а не только для компьютера и себя самого.

Патрик Логан (Patrick Logan) рассказал об идее, с которой я намерен поэкспериментировать. Эта идея также описана Макконнеллом (McConnell) [21], а также Кэйном (Caine) и Гордоном (Gordon)[22]:

В *последнее время я фактически постоянно применяю методику «основных тезисов» в любой моей работе. Тестирование не является исключением. Когда я пишу тесты, я прежде всего записываю план из нескольких пунктов – тезисов, – которые я хотел бы реализовать в этом тесте. Например*:

/* Добавить в пространство кортежей[23] */
/* Извлечь из пространства кортежей */
/* Читать из пространства кортежей */

Это самые основные тезисы, однако я добавляю в каждую из этих категорий конкретные тесты. Когда я добавляю тесты, я добавляю в мой список тезисов еще один уровень комментариев:

```
/* Добавить в пространство кортежей */
/* Извлечь из пространства кортежей */
/** Извлечение несуществующего элемента **/
/** Извлечение существующего элемента **/
/** Извлечение нескольких элементов **/
/* Читать из пространства кортежей */
```

Как правило, мне хватает двух-трех уровней комментариев. Я не могу представить ситуацию, в которой мне могло бы потребоваться больше уровней. Список тезисов становится документацией контракта для тестируемого класса. Приведенные здесь примеры, конечно же, сокращены, однако в языках программирования, поддерживающих контракты, тезисы могли бы быть более конкретными. (Я не использую какие-либо добавления к Java, обеспечивающие автоматизацию в стиле Eiffel.)

Сразу же после самого низкого уровня комментариев располагается исходный код теста.

Тест исключения (Exception Test)

Как можно протестировать ожидаемое исключение? Перехватите исключение и игнорируйте его, тест должен терпеть неудачу только в случае, если исключение не сгенерировано.

Предположим, что мы пишем код, осуществляющий поиск значения. Если значение не обнаружено, мы хотим сгенерировать исключение. Тестирование механизма поиска выполняется относительно просто:

```
public void testRate() {
exchange.addRate("USD", "GBP", 2);
int rate = exchange.findRate("USD", "GBP");
assertEquals(2, rate);
}
```

Тестирование исключения может оказаться неочевидным. Вот как мы это делаем:

```
public void testMissingRate() {
```

```
try {
exchange.findRate("USD", "GBP");
fail();
} catch (IllegalArgumentException expected) {
}
}
```

Если метод findRate() не генерирует исключения, произойдет обращение к методу fail() – это метод xUnit, который докладывает о том, что тест потерпел неудачу. Обратите внимание, что мы перехватываем только то исключение, которое должно быть сгенерировано методом findRate(). Благодаря этому, если будет сгенерировано какое-либо другое (неожиданное для нас) исключение (включая сбой метода assert), мы узнаем об этом.

Все тесты (All Tests)

Как можно запустить все тесты вместе? Создайте тестовый набор, включающий в себя все имеющиеся тестовые наборы, – один для каждого пакета (package) и один, объединяющий в себе все тесты пакетов для всего приложения.

Предположим, вы добавили подкласс класса TestCase и в этот подкласс вы добавили тестовый метод. В следующий раз, когда будут выполняться все тесты, добавленный вами тестовый метод также должен быть выполнен. (Во мне опять проснулась привычка действовать в стиле TDD – должно быть, вы заметили, что предыдущее предложение – это эскиз теста, который я, наверное, написал бы, если бы не был занят работой над данной книгой.) К сожалению, в большинстве реализаций xUnit, равно как и в большинстве IDE, не поддерживается стандартный механизм запуска абсолютно всех тестов, поэтому в каждом пакете необходимо определить класс AllTests, который реализует статический метод suite(), возвращающий объект класса TestSuite. Вот класс AllTests для «денежного» примера:

```
public class AllTests {
public static void main(String[] args) {
junit.swingui.TestRunner.run(AllTests.class);
}
public static Test suite() {
```

```
TestSuite result = new TestSuite("TFD tests");
result.addTestSuite(MoneyTest.class);
result.addTestSuite(ExchangeTest.class);
result.addTestSuite(IdentityRateTest.class);
return result;
}
}
```

Вы также должны включить в класс AllTests() метод main(), благодаря чему класс можно будет запустить напрямую из IDE или из командной строки.

30. Шаблоны проектирования

В чем заключается основная идея шаблонов? Нам кажется, что мы постоянно сталкиваемся с разнообразными, неповторяющимися проблемами, однако на деле оказывается, что большая часть проблем, которые нам приходится решать, обусловлена используемыми нами инструментами, но не основной задачей, которая перед нами стоит[24]. Если исходить из этого предположения, то можно найти (и мы действительно находим) общие проблемы со стандартными решениями, несмотря на все разнообразие контекстов, в рамках которых нам приходится работать.

Использование объектов для организации вычислений – это один из лучших примеров стандартного решения, направленного на устранение множества общих проблем, с которыми программистам приходится сталкиваться при разработке самого разнообразного программного обеспечения. Колоссальный успех шаблонов проектирования (design patterns) является доказательством общности проблем, с которыми сталкиваются программисты, использующие объектно-ориентированные языки программирования[25]. Книга *Design Patterns* («Паттерны проектирования») имела большой успех, однако ее популярность стала причиной сужения взгляда на шаблоны проектирования. Что я имею в виду? Книга рассматривает дизайн как фазу разработки программы, однако авторы совершенно не учитывают, что рефакторинг – это мощный инструмент формирования дизайна. Дизайн в рамках TDD требует несколько иного взгляда на шаблоны проектирования.

В данной главе я расскажу о нескольких полезных шаблонах проектирования. Безусловно, мое изложение не претендует на полноту. Представленная здесь информация может оказаться полезной при изучении рассматриваемых в книге примеров. Вот краткое перечисление рассмотренных здесь шаблонов:

- «Команда» (Command) – обращение к некоторому коду представляется в виде объекта, а не в виде простого сообщения;
- «Объект-значение» (Value Object) – после создания объекта его значение никогда не меняется, благодаря этому удается избежать проблем, связанных с наложением имен (aliasing);
- «Нуль-объект» (Null Object) – соответствует базовому случаю вычислений объекта;
- «Шаблонный метод» (Template Method) – представляет собой

инвариантную последовательность операций, определяемую при помощи абстрактных методов, которые можно переопределить с помощью наследования;

• «Встраиваемый объект» (Pluggable Object) – представляет собой вариацию в виде объекта с двумя реализациями или большим их количеством;

• «Встраиваемый переключатель» (Pluggable Selector) – позволяет избежать создания многочисленных подклассов путем динамического обращения к различным методам для различных экземпляров класса;

• «Фабричный метод» (Factory Method) – вместо конструктора для создания объекта используется специальный метод;

• «Самозванец» (Imposter) – представляет собой вариацию путем создания новой реализации существующего протокола;

• «Компоновщик» (Composite) – композиция объектов ведет себя так же, как один объект;

• «Накапливающий параметр» (Collecting Parameter) – результаты вычислений, выполняемых в разных объектах, накапливаются в специальном объекте, который передается объектам, выполняющим вычисления, в качестве параметра.

В табл. 30.1 описывается, на каких этапах TDD используется тот или иной шаблон проектирования.

Таблица 30.1. Использование шаблонов проектирования при разработке через тестирование (TDD)

Шаблон	Написание тестов	Рефакторинг
«Команда» (Command)	✓	
«Объект-значение» (Value Object)	✓	
«Нуль-объект» (Null Object)		✓
«Шаблонный метод» (Template Method)		✓

Шаблон	Написание тестов	Рефакторинг
«Встраиваемый объект» (Pluggable Object)		✓
«Встраиваемый переключатель» (Pluggable Selector)		✓
«Фабричный метод» (Factory Method)	✓	✓
«Самозванец» (Imposter)	✓	✓
«Компоновщик» (Composite)	✓	✓
«Накапливающий параметр» (Collecting Parameter)	✓	✓

Команда (Command)

Что делать, если выполнение некоторой операции представляет собой нечто более сложное, чем простое обращение к методу? Создайте объект, соответствующий этой операции, и обратитесь к этому объекту.

Передача сообщений – это отличный механизм. Языки программирования делают передачу сообщений синтаксически простым действием, а среды разработки позволяют с легкостью манипулировать сообщениями (например, автоматическое выполнение рефакторинга по переименованию метода). Однако в некоторых случаях простой передачи сообщения недостаточно.

Например, представьте, что вы хотели бы занести в журнал запись о том, что сообщение было передано. Для этой цели можно воспользоваться средствами языка (например, методы-обертки), однако простые операции журналирования – это далеко не все, в чем вы можете нуждаться. Представьте, что мы хотим вызвать некоторую процедуру, но несколько позднее. Для этой цели можно создать новый программный поток, сразу же приостановить его работу, а затем, когда это потребуется, запустить его. Однако в подобной ситуации нам придется иметь дело с параллельными потоками, а это может оказаться слишком тяжеловесным подходом.

Для выполнения операций с подобными дополнительными условиями зачастую требуются сложные затратные механизмы. Однако в большинстве случаев мы можем избежать излишней сложности и лишних затрат. Проблему вызова можно решить с помощью более конкретной и гибкой формы, чем сообщение. Для этого достаточно создать специальный объект.

Создайте объект, представляющий собой вызов операции. Занесите в этот объект все необходимые параметры операции. Когда операция готова к выполнению, используйте для этого универсальный протокол, например метод run().

Отличным примером использования данного подхода является интерфейс Runnable языка Java:

Runnable
interface Runnable
public abstract void run();

В рамках реализации метода run() вы можете делать все, что вам нравится. К сожалению, Java не поддерживает синтаксически легковесного способа создания объектов Runnable и обращения к этим объектам, поэтому они не используются так часто, как их эквиваленты в других языках (блоки или лямбда-выражения в Smalltalk/Ruby или LISP).

Объект-значение (Value Object)

Как следует спроектировать объект, который будет широко использоваться, но для которого идентификация не имеет особого значения? Настройте состояние объекта в момент его создания и никогда не меняйте его. В результате выполнения любых операций с данным объектом должен получаться новый объект.

Объектно-ориентированный подход – это великолепная вещь. Надеюсь, я имею право написать эту фразу в данной книге. Объекты являются отличным способом организации логики для последующего понимания и роста. Однако существует одна маленькая проблема (хорошо, хорошо, вообще-то проблем больше, однако сейчас мы коснемся только одной из них).

Представьте, что я – объект и у меня есть прямоугольник (Rectangle). Я вычисляю некоторое значение, зависящее от этого прямоугольника, например его площадь. Чуть позже некто (например, другой объект) вежливо просит меня предоставить ему мой прямоугольник для выполнения некоторой операции. Чтобы не показаться невежливым, я предоставляю ему мой прямоугольник. А через пару мгновений, вы только посмотрите, прямоугольник был модифицирован у меня за спиной! Значение площади, которое я вычислил ранее, теперь не соответствует

действительности, и не существует способа известить меня об этом.

Это классический пример проблемы наложения имен (aliasing). Если два объекта ссылаются на один и тот же третий объект и если один из первых двух тем или иным образом изменяет третий, общий для них, объект, второму объекту лучше не полагаться на текущее состояние общего объекта.

Существует несколько способов решения проблемы наложения имен. Во-первых, вы можете никому не отдавать объект, от состояния которого вы зависите. Вместо этого в случае необходимости вы можете создавать копии этого объекта. Такой подход может потребовать слишком много времени и слишком много пространства, кроме того, игнорируется ситуация, когда вы хотите сделать изменения некоторого объекта общими для нескольких других объектов, зависящих от его состояния. Еще одно решение – шаблон «Наблюдатель» (Observer). В этом случае, если вы зависите от состояния некоторого объекта, вы должны предварительно сообщить ему об этом, иначе говоря, зарегистрироваться. Объект, за состоянием которого следят, оповещает все зарегистрированные им объекты-наблюдатели о своем изменении. Шаблон «Наблюдатель» (Observer) может затруднить понимание последовательности выполнения операций, кроме того, логика формирования и удаления зависимостей между объектами выглядит далеко не идеальной.

Еще одно решение предлагает несколько ограничить возможности, которыми обладает типичный объект в рамках ООП. Образно говоря, объект становится «менее чем объектом». Что это значит? Обычные объекты обладают состоянием, которое изменяется с течением времени. Если мы захотим, мы можем запретить им меняться. Если у меня есть объект и я знаю, что он не может измениться, я могу передавать ссылки на этот объект любому другому объекту, не беспокоясь при этом о проблеме наложения имен. Если объект не поддерживает возможности своего изменения, никаких модификаций у меня за спиной не может произойти.

Я помню, как похожая ситуация возникла с целыми числами, когда я впервые изучал язык Smalltalk. Если я изменяю бит 2 на 1, почему все двойки не становятся шестерками?

```
a:= 2.
b:= a.
a:= a bitAt: 2 put: 1.
a => 6
b => 2
```

Целые числа – это значения, которые маскируются под объекты. В языке Small-talk это утверждение является истиной для небольших целых чисел и имитируется в случае, если целое число не умещается в машинное слово. Когда я устанавливаю бит, то получаю в свое распоряжение новый объект с установленным битом. Старый объект остается неизменным.

В рамках шаблона «Объект-значение» (Value Object) каждая операция должна возвращать новый объект, а первоначальный объект должен оставаться неизменным. Пользователи должны знать, что они используют объект-значение. В этом случае полученный объект следует сохранить (как в предыдущем примере). Конечно же, из-за необходимости создания новых объектов полученный в результате код может оказаться медленным. Однако в данном случае любые проблемы с производительностью должны решаться в точности так же, как и любые другие проблемы с производительностью: вы должны оценить производительность при помощи тестов с реальными данными, определить, насколько часто производится обращение к медленному коду, выполнить профилирование и определить, какой именно код должен быть оптимизирован и как лучше всего этого достичь.

Я предпочитаю использовать «Объект-значение» (Value Object) в ситуациях, когда операции, выполняемые над объектами, напоминают алгебру. Например, пересечение и объединение геометрических фигур, операции над значениями, с каждым из которых хранится единица измерения, а также операции символьной арифметики. Каждый раз, когда использование «Объект-значение» (Value Object) имеет хоть какой-то смысл, я пытаюсь его использовать, так как результирующий код проще читать и отлаживать.

Все объекты-значения должны реализовать операцию сравнения (а во многих языках подразумевается, что они должны реализовать также операцию хеширования). Если я имею один контракт и другой контракт и они не являются одним и тем же объектом, значит, они не равны. Однако если у меня есть одни пять франков и другие пять франков, для меня не имеет значения тот факт, что это два разных объекта – пять франков и в Африке пять франков – они равны.

Нуль-объект (Null Object)

Как реализовать специальные случаи использования объектов?

Создать специальный объект, представляющий собой специальный случай. Специальный объект должен обладать точно таким же протоколом, что и обычный объект, но он должен вести себя специальным образом.

В качестве примера рассмотрим код, который я позаимствовал из java.io.File:

java.io.File

```
public boolean setReadOnly() {
SecurityManager guard = System.getSecurityManager();
if (guard!= null) {
guard.canWrite(path);
}
return fileSystem.setReadOnly(this);
}
```

В классе java.io.File можно обнаружить 18 проверок guard!= null. Я преклоняюсь перед усердием, с которым разработчики библиотек Java стараются сделать файлы безопасными для всего остального мира, однако я также начинаю немножко нервничать. Будут ли программисты Oracle и в будущем столь же аккуратны, чтобы не забыть проверить результат выполнения метода getSecurityManager() на равенство значению null?

В рамках альтернативного решения можно создать новый класс LaxSecurity, который вообще не генерирует исключений:

LaxSecurity

```
public void canWrite(String path) {
}
```

Если кто-то пытается получить SecurityManager, однако предоставить такой объект нет возможности, вместо него мы возвращаем LaxSecurity:

SecurityManager

```
public static SecurityManager getSecurityManager() {
return security == null? new LaxSecurity(): security;
}
```

Теперь мы можем не беспокоиться о том, что кто-то забудет проверить результат выполнения метода на равенство значению null. Изначальный код становится существенно более чистым:

File

```
public boolean setReadOnly() {
SecurityManager security = System.getSecurityManager();
security.canWrite(path);
return fileSystem.setReadOnly(this);
}
```

Однажды во время выступления на конференции OOPSLA нас с Эр

ихом Гаммой (Erich Gamma) спросили, можно ли использовать «Нуль-объект» (Null Object) в рамках одного из классов JHotDraw. Я принялся рассуждать о преимуществах такой модернизации, в то время как Эрих посчитал, что для этого нам придется увеличить код на десять строк, при этом мы избавимся от одного условного оператора – преимущество сомнительно. (К тому же аудитория была весьма недовольна нашей несогласованностью.)

Шаблонный метод (Template Method)

Как можно запрограммировать инвариантную последовательность операций, обеспечив при этом возможность модификации или замены отдельных действий в будущем? Напишите реализацию метода исключительно в терминах других методов.

В программировании существует огромное количество классических последовательностей:

- ввод – обработка – вывод;
- отправить сообщение – принять ответ;
- прочитать команду – вернуть результат.

Нам хотелось бы четко и понятно обозначить универсальность этих последовательностей и при этом обеспечить возможность варьирования реализаций каждого из отдельных этапов.

Поддерживаемый любым объектно-ориентированным языком механизм наследования обеспечивает простой способ определения универсальных последовательностей. В суперклассе создается метод, целиком и полностью написанный в терминах других методов. Каждый из подклассов может реализовать эти методы так, как ему удобнее. Например, базовая последовательность выполнения теста определяется в инфраструктуре JUnit следующим образом:

TestCase

```
public void runBare() throws Throwable {
setUp();
try {
runTest();
}
finally {
tearDown();
}
}
```

Классы, производные от TestCase, могут реализовать setUp(), runTest() и tearDown() так, как им этого хочется.

При использовании шаблона «Шаблонный метод» (Template Method) возникает вопрос: надо ли создавать для подметодов реализации по умолчанию? В TestCase.runBare() все три подметода обладают реализациями по умолчанию:

• методы setUp() и tearDown() не выполняют никаких операций;

• метод runTest() динамически обнаруживает и запускает все тестовые методы, исходя из имени класса-теста.

Если общая последовательность не имеет смысла, когда не определен один из ее этапов, вы должны отметить это, воспользовавшись любой подходящей возможностью используемого вами языка программирования:

• в Java можно объявить подметод абстрактным;

• в Smalltalk создайте реализацию метода, которая генерирует ошибку SubclassResponsibility.

Я не рекомендую изначально проектировать код так, чтобы в нем использовался шаблонный метод. Лучше всего формировать шаблонные методы исходя из накопленного опыта. Каждый раз, когда я говорю себе: «Ага, вот последовательность, а вот – детали реализации», – позднее я всегда обнаруживаю, что мне приходится переделывать созданный мною шаблонный метод, заново перетасовывая код между общим и частным.

Если вы обнаружили два варианта последовательности в двух подклассах, вы должны попытаться постепенно приблизить их друг к другу. После того как вы отделите различающиеся части и выделите общую часть, то, что останется, и есть шаблонный метод. После этого вы можете переместить шаблонный метод в суперкласс и избавиться от дублирования.

Встраиваемый объект (Pluggable Object)

Как можно выразить несколько разных вариантов поведения кода? Проще всего использовать явный условный оператор:

```
if(circle) then {
… код, относящийся к circle.
} else {
… код, не относящийся к circle
}
```

Однако подобный корявый код имеет тенденцию распространяться по всей программе. Если для определения разницы между окружностями и не окружностями вы будете использовать условный оператор хотя бы в одном месте вашего кода, с большой долей уверенности можно сказать, что позднее подобный оператор придется добавить также в другом месте, затем в третьем и т. д.

Вторая по важности задача TDD – устранение дублирования, поэтому вы должны подавить угрозу распространения явных условных операторов в зародыше. Если вы видите, что одно и то же условие проверяется в двух разных местах вашего кода, значит, настало время выполнить базовое объектно-ориентированное преобразование: «Встраиваемый объект» (PluggableObject).

Иногда обнаружить необходимость применения этого шаблона не так просто. Один из самых любимых мною примеров использования встраиваемого объекта был придуман мною и Эрихом Гаммой. Представьте, что мы занимаемся разработкой графического редактора. Если вы когда-нибудь занимались чем-либо подобным, должно быть, вы знаете, что операция выделения объектов обладает несколько усложненной логикой. Если указатель мыши находится над графической фигурой и пользователь нажимает кнопку мыши, значит, последующие перемещения мыши приводят к перемещению фигуры, а при отпускании кнопки мыши выбранная фигура остается на новом месте. Если указатель мыши не находится над какой-либо фигурой, значит, нажав кнопку, пользователь выделяет несколько фигур, последующие перемещения мыши приводят к изменению размера прямоугольника выделения, а при отпускании кнопки мыши фигуры внутри прямоугольника выделения становятся выделенными. Изначальный код выглядит примерно так:

SelectionTool

```
Figure selected;
```

```
public void mouseDown() {
selected = findFigure();
if (selected!= null)
select(selected);
}
public void mouseMove() {
if (selected!= null)
move(selected);
else
moveSelectionRectangle();
}
public void mouseUp() {
if (selected == null)
selectAll();
}
```

В глаза бросаются три похожих условных оператора (я же говорил, что они плодятся, как мухи). Что делать, чтобы избавиться от них? Создаем встраиваемый объект, SelectionMode, обладающий двумя реализациями: SingleSelection и MultipleSelection.

SelectionTool

```
SelectionMode mode;
public void mouseDown() {
selected = findFigure();
if (selected!= null)
mode = SingleSelection(selected);
else
mode = MultipleSelection();
}
public void mouseMove() {
mode.mouseMove();
}
public void mouseUp() {
mode.mouseUp();
}
```

В языках с явными интерфейсами вы обязаны реализовать интерфейс с двумя (или больше) встраиваемыми объектами.

Встраиваемый переключатель (Pluggable Selector)[26]

Как обеспечить различающееся поведение разных экземпляров одного и того же класса? Сохраните имя метода в переменной и динамически обращайтесь к этому методу.

Что делать, если у вас есть десять подклассов одного базового класса и в каждом из них реализован только один метод? Может оказаться, что создание подклассов – это слишком тяжеловесный механизм для реализации столь небольших различий в поведении объектов.

```
abstract class Report {
abstract void print();
}
class HTMLReport extends Report {
void print() {…
}
}
class XMLReport extends Report {
void print() {…
}
}
```

Альтернативное решение: создать единственный класс с оператором switch. В зависимости от значения поля происходит обращение к разным методам. Однако в этом случае имя метода упоминается в трех местах:

- при создании экземпляра;
- в операторе switch;
- в самом методе.

```
abstract class Report {
String printMessage;

Report(String printMessage) {
this.printMessage = printMessage;
}
void print() {
switch (printMessage) {
case "printHTML":
```

```
printHTML();
break;
case "printXML":
printXML():
break;
}
};

void printHTML() {
}

void printXML() {
}
}
```

Каждый раз, когда вы добавляете новую разновидность печати, вы должны позаботиться о добавлении нового метода печати и редактировании оператора switch.

Шаблон «Встраиваемый переключатель» (Pluggable Selector) предлагает динамически обращаться к методу с использованием механизма рефлексии:

```
void print() {
Method runMethod = getClass(). getMethod(printMessage, null);
runMethod.invoke(this, new Class[0]);
}
```

По-прежнему существует весьма неприятная зависимость между создателями отчетов и именами методов печати, однако, по крайней мере, мы избавились от оператора switch.

Естественно, этим шаблоном не следует злоупотреблять. Самая большая связанная с ним проблема состоит в отслеживании вызываемого кода. Используйте встраиваемый переключатель только в случае, когда вы оказались в стандартной ситуации: каждый из подклассов обладает всего одним методом, и у вас есть желание сделать этот код более чистым.

Фабричный метод (Factory Method)

Как лучше всего создавать объекты в случае, если вы хотите обеспечить гибкость при создании объектов? Вместо того чтобы использовать конструктор, создайте объект внутри специального метода.

Безусловно, конструкторы являются выразительным инструментом. Если вы используете конструктор, всем, кто читает код, однозначно становится ясно, что вы создаете объект. Однако конструкторы, в особенности в Java, не обеспечивают достаточной гибкости.

В рассмотренном ранее «денежном» примере при создании объекта мы хотели бы возвращать объект иного класса. У нас есть следующий тест:

```
public void testMultiplication() {
Dollar five = new Dollar(5);
assertEquals(new Dollar(10), five.times(2));
assertEquals(new Dollar(15), five.times(3));
}
```

Мы хотели бы добавить в программу новый класс Money, однако мы не можем этого сделать, так как для тестирования нам нужен экземпляр класса Dollar. Чтобы решить проблему, достаточно добавить в программу дополнительный уровень перенаправления – метод, который будет возвращать объект иного класса. В этом случае мы сможем оставить выражения assert без изменений:

```
public void testMultiplication() {
Dollar five = Money.dollar(5);
assertEquals(new Dollar(10), five.times(2));
assertEquals(new Dollar(15), five.times(3));
}
```

Money

```
static Dollar dollar(int amount) {
return new Dollar(amount);
}
```

Такой метод называется *фабричным методом* (Factory Method), так как он предназначен для создания объектов.

Недостаток этого шаблона заключается в том, что предназначение фабричного метода не очевидно: вы должны помнить о том, что этот метод создает объекты, вместе с тем это обычный метод, а не конструктор.

Фабричный метод следует использовать только тогда, когда вы действительно нуждаетесь в гибкости, которую он обеспечивает. В противном случае для создания объектов вполне можно использовать обычные конструкторы.

Самозванец (Imposter)

Как можно добавить в программу новую вариацию некоторой функциональности? Создайте новый объект с точно таким же протоколом, как и существующий объект, но с отличающейся реализацией.

При использовании процедурно-ориентированного подхода для решения подобной задачи в программу требуется добавить как минимум один условный оператор. Как было продемонстрировано ранее, при обсуждении шаблона «Встраиваемый переключатель» (Pluggable Selector), такие условные операторы имеют тенденцию плодиться подобно саранче. Чтобы избавиться от дублирования, требуется полиморфизм.

Представьте, что у вас уже есть необходимая инфраструктура. У вас уже есть объект, который реализует необходимую функциональность. Теперь вы хотите, чтобы ваша система делала нечто отличающееся. Если вы обнаружили очевидное место для добавления оператора if и при этом не возникает дублирования какой-либо существующей логики, действуйте смело и решительно. Однако зачастую для добавления вариации требуется внести изменения в код нескольких методов.

Если вы работаете в стиле TDD, решение об использовании самозванца может возникнуть исходя из разных предпосылок. Иногда вы пишете тест и у вас возникает желание реализовать новый сценарий. Однако ни один из существующих объектов не выражает того, что вы хотите выразить. Представьте, что мы тестируем графический редактор и нам уже удалось реализовать корректное рисование прямоугольников:

```
testRectangle() {
Drawing d = new Drawing();
d. addFigure(new RectangleFigure(0, 10, 50, 100));
RecordingMedium brush = new RecordingMedium();
d. display(brush);
assertEquals("rectangle 0 10 50 100\n", brush.log());
}
```

Теперь мы хотим реализовать рисование овалов. В данном случае необходимость применения шаблона «Самозванец» (Imposter) очевидна: заменяем RectangleFigure на OvalFigure.

```
testOval() {
Drawing d = new Drawing();
d. addFigure(new OvalFigure(0, 10, 50, 100));
RecordingMedium brush = new RecordingMedium();
d. display(brush);
assertEquals("oval 0 10 50 100\n", brush.log());
}
```

Как правило, чтобы увидеть необходимость использования этого шаблона еще до начала разработки кода, требуется озарение. Именно озарением можно назвать момент, когда Уорд Каннингэм решил, что вектор объектов Money может вести себя так же, как одиночный объект Money. Сначала можно подумать, что они различаются, однако после вы понимаете, что они одинаковы.

Вот два примера использования «Самозванец» (Imposter) в процессе рефакторинга:

• «Нуль-объект» (Null Object) – вы можете рассматривать отсутствие данных в точности так же, как и присутствие данных;

• «Компоновщик» (Composite) – вы можете рассматривать коллекцию объектов как одиночный объект.

Решение об использовании «Самозванец» (Imposter) в процессе рефакторинга принимается для устранения дублирования, впрочем, целью любого рефакторинга является устранение дублирования.

Компоновщик (Composite)

Как лучше всего реализовать объект, чье поведение является композицией функций некоторого набора других объектов? Примените шаблон «Самозванец» (Imposter) – заставьте этот объект вести себя подобно тому, как ведут себя отдельные объекты, входящие в набор.

Мой любимый пример основан на двух объектах: Account (счет) и Transaction (транзакция). Этот пример помимо прочего демонстрирует некоторую противоречивость шаблона «Компоновщик» (Composite), но об этом позже. В объекте Transaction хранится изменение величины счета

(безусловно, транзакция – это более сложный и интересный объект, однако на данный момент мы ограничимся лишь мизерной долей его возможностей):

```
Transaction
Transaction(Money value) {
this.value = value;
}
```

Объект Accout вычисляет баланс счета путем суммирования значений относящихся к нему объектов Transaction:

```
Account
Transaction transactions[];
Money balance() {
Money sum = Money.zero();
for (int i = 0; i < transactions.length; i++)
sum = sum.plus(transactions[i].value);
return sum;
}
```

Все выглядит достаточно просто:

- в объектах Transaction хранятся значения;
- в объекте Account хранится баланс.

Теперь самое интересное. У клиента есть несколько счетов, и он хочет узнать общий баланс по всем этим счетам. Первая мысль, которая приходит в голову: создать новый класс OverallAccount, который суммирует балансы для некоторого набора объектов Account. Дублирование! Дублирование!

А что, если классы Account и Transaction будут поддерживать один и тот же интерфейс? Давайте назовем его Holding (сбережения), потому что сейчас мне не удается придумать что-либо лучшее:

```
Holding
interface Holding
Money balance();
```

Чтобы реализовать метод balance() в классе Transaction, достаточно вернуть хранящееся в этом классе значение:

Transaction

```
Money balance() {
return value;
}
```

Теперь в классе Account можно хранить не транзации, а объекты Holding:

Account

```
Holding holdings[];
Money balance() {
Money sum = Money.zero();
for (int i = 0; i < holdings.length; i++)
sum = sum.plus(holdings[i].balance());
return sum;
}
```

Проблема, связанная с созданием класса OverallAccount, испарилась в воздухе. Объект OverallAccount – это просто еще один объект Account, в котором хранятся не транзакции, а другие объекты Account.

Теперь о противоречивости. В приведенном примере хорошо чувствуется запах шаблона «Компоновщик» (Composite). В реальном мире транзакция не может содержать в себе баланс. В данном случае программист идет на уловку, которая совершенно не логична с точки зрения всего остального мира. Вместе с тем преимущества подобного дизайна неоспоримы, и ради этих преимуществ можно пожертвовать некоторым концептуальным несоответствием. Если присмотреться, подобные несоответствия встречаются нам на каждом шагу: папки (Folders), в которых содержатся другие папки (Folders), наборы тестов (TestSuites), в которых содержатся другие наборы тестов (TestSuites), рисунки (Drawings), в которых содержатся другие рисунки (Drawings). Любая из этих метафор недостаточно хорошо соответствует взаимосвязи между вещами в реальном мире, однако все они существенно упрощают код.

Я вынужден был длительное время экспериментировать с шаблоном «Компоновщик» (Composite), прежде чем научился понимать, когда его следует использовать, а когда – нет. Наверное, вы уже поняли, что я не могу предоставить вам однозначных рекомендаций относительно решения проблемы, в каких ситуациях коллекция объектов является просто

коллекцией объектов, а в каких это – объект-компоновщик. Хорошая новость состоит в том, что, когда вы достаточно хорошо освоите рефакторинг, вы наверняка сможете обнаружить возникновение дублирования, воспользоваться шаблоном «Компоновщик» (Composite) и обнаружить, что код существенно упростился.

Накапливающий параметр (Collecting Parameter)

Как можно сформировать результат операции, если она распределена между несколькими объектами? Используйте параметр, в котором будут накапливаться результаты операции.

Простым примером является интерфейс java.io.Externalizable. Метод writeExternal этого интерфейса осуществляет запись объекта и всех объектов, на которые ссылается данный объект. Чтобы обеспечить общую запись, все записываемые объекты должны взаимодействовать друг с другом, поэтому методу передается параметр – объект класса ObjectOutput, – в котором осуществляется накопление:

java.io.Externalizable

```
public interface Externalizable extends java.io.Serializable {
void writeExternal(ObjectOutput out) throws IOException;
}
```

Добавление параметра-накопителя зачастую является последствием использования шаблона «Компоновщик» (Composite). В начале разработки JUnit не было необходимости накапливать результаты выполнения нескольких тестов в объекте TestResult до тех пор, пока в инфраструктуру не была добавлена возможность создания и запуска нескольких тестов.

Необходимость использования параметра-накопителя возникает в ситуации, когда возрастает сложность объекта, получаемого в результате комплексной операции. Например, представьте, что нам необходимо реализовать вывод объекта Expression на экран в виде строки символов. Если обычная, не структурированная строка – это все, что нам нужно, значит, конкатенации будет вполне достаточно:

```
testSumPrinting() {
Sum sum = new Sum(Money.dollar(5), Money.franc(7));
assertEquals("5 USD + 7 CHF", sum.toString());
```

```
}
String toString() {
return augend + " + " + addend;
}
```

Однако если мы хотим отобразить объект Expression в виде древовидной структуры, код может выглядеть следующим образом:

```
testSumPrinting() {
Sum sum = new Sum(Money.dollar(5), Money.franc(7));
assertEquals("+\n\t5 USD\n\t7 CHF", sum.toString());
}
```

В этом случае придется воспользоваться параметром-накопителем:

```
String toString() {
IndentingStream writer = new IndentingStream();
toString(writer);
return writer.contents();
}

void toString(IndentingWriter writer) {
writer.println("+");
writer.indent();
augend.toString(writer);
writer.println();
addend.toString(writer);
writer.exdent();
}
```

Одиночка (Singleton)

Как можно реализовать глобальную переменную в языке, в котором не поддерживаются глобальные переменные? Не следует этим заниматься. Ваша программа скажет вам большое спасибо, если вместо этого вы еще раз хорошенько обдумаете дизайн и откажетесь от мысли использовать глобальные переменные.

31. Рефакторинг

Рассматриваемые здесь шаблоны помогут изменить дизайн системы маленькими шажками.

В рамках TDD рефакторинг[27] используется интересным образом. Обычно рефакторинг не может изменить семантику программы ни при каких условиях. В рамках TDD условия семантики формулируются при помощи тестов, которые уже выполняются успешно. Таким образом, в рамках TDD мы можем, например, заменить константы переменными и с чистой совестью назвать эту процедуру рефакторингом, потому что набор успешных тестов при этом не изменился. Однако набор успешных тестов может состоять всего из одного теста. Возможно, семантика программы должна описываться большим количеством тестов. Возможно также, что некоторые из этих потенциальных тестов в результате выполнения рефакторинга перестали бы срабатывать, если бы они существовали. Однако их нет, поэтому мы о них не беспокоимся.

Отсюда следует, что на программиста, работающего в стиле TDD, возлагается важная обязанность: он должен иметь достаточное количество тестов, описывающих семантику программы. Достаточное настолько, насколько он может судить на момент завершения работы над кодом. Необходимо понимать, что рефакторинг выполняется не с учетом всех существующих тестов, а с учетом всех возможных тестов. Фраза: «Я знаю, что там была проблема, но все тесты выполнились успешно, поэтому я посчитал код завершенным и интегрировал его в систему», – не может считаться оправданием. Пишите больше тестов.

Согласование различий (Reconcile Differences)

Как можно унифицировать два схожих фрагмента кода? Постепенно делайте их все более похожими друг на друга. Унифицируйте их только в случае, если они абсолютно идентичны.

Подчас рефакторинг – это весьма нервная работа. Простые изменения в коде очевидны. Если я извлекаю метод и делаю это механически корректно, вероятность того, что поведение системы изменится, чрезвычайно мала. Однако некоторые из изменений заставляют внимательно анализировать последовательность выполнения операций и

порядок модификации данных. Построив длинную цепочку умозаключений, вы приходите к выводу, что запланированное вами изменение кода, скорее всего, не приведет к изменению поведения системы. Однако любой подобный рефакторинг уменьшает количество волос на вашей голове.

Сложные изменения – это именно то, чего мы пытаемся избежать, когда придерживаемся стратегии маленьких шажков и конкретной обратной связи. Полностью избежать сложных изменений невозможно, однако можно уменьшить их влияние на остальной код.

Подобные изменения возникают на разных уровнях:

• Два цикла выглядят похоже. Если вы сделаете их идентичными, вы сможете объединить их в единый цикл.

• Две ветви условного оператора выглядят похоже. Сделав их идентичными, вы сможете избавиться от условного оператора.

• Два метода выглядят похоже. Сделав их идентичными, вы сможете избавиться от одного из них.

• Два класса выглядят похоже. Сделав их идентичными, вы сможете избавиться от одного из них.

Иногда задачу согласования различий удобнее решать в обратном порядке. Иными словами, вы представляете себе самый тривиальный последний этап этой процедуры, а затем двигаетесь в обратном направлении. Например, если вы хотите избавиться от нескольких подклассов, наиболее тривиальный последний шаг можно будет выполнить в случае, если подкласс ничего не содержит. Тогда везде, где используется подкласс, можно будет использовать суперкласс, при этом поведение системы не изменится. Что надо сделать, чтобы очистить подкласс от методов и данных? Для начала метод можно сделать полностью идентичным одному из методов суперкласса. Постепенно переместив все методы и все данные в суперкласс, вы сможете заменить ссылки на подкласс ссылками на суперкласс. После этого подкласс можно уничтожить.

Изоляция изменений (Isolate Change)

Как можно модифицировать одну часть метода или объекта, состоящего из нескольких частей? Сначала изолируйте изменяемую часть.

Мне приходит в голову аналогия с хирургической операцией: фактически все тело оперируемого пациента покрыто специальной

простыней за исключением места, на котором, собственно, осуществляется операция. Благодаря такому покрытию хирург имеет дело с фиксированным набором переменных. Перед выполнением операции врачи сколь угодно долго могут обсуждать, какое влияние на здоровье пациента оказывает тот или иной орган, однако во время операции внимание хирурга должно быть сфокусировано.

Вы можете обнаружить, что после того, как вы изолировали изменение, а затем внесли это изменение в код, результат получился настолько тривиальным, что вы можете отменить изоляцию. Например, если мы обнаружили, что внутри метода findRate() должно присутствовать всего одно действие – возврат значения поля, мы можем вместо обращений к методу findRate() напрямую обратиться к полю. В результате метод findRate() можно будет удалить. Однако подобные изменения не следует выполнять автоматически. Постарайтесь найти баланс между затратами, связанными с использованием дополнительного метода, и пользой, которую приносит дополнительная концепция, добавленная в код.

Для изоляции изменений можно использовать несколько разных способов. Наиболее часто используется шаблон «Выделение метода» (Extract Method), помимо него также используются «Выделение объекта» (Extract Object) и «Метод в объект» (Method Object).

Миграция данных (Migrate Data)

Как можно перейти от одного представления к другому? Временно дублируйте данные.

Как

Вначале рассмотрим версию «от внутреннего к внешнему». В рамках этого подхода вы изменяете вначале внутреннее представление, а затем внешний интерфейс.

1. Создайте переменную экземпляра в новом формате.
2. Инициализируйте переменную нового формата везде, где инициализируется переменная старого формата.
3. Используйте переменную нового формата везде, где используется переменная старого формата.
4. Удалите старый формат.
5. Измените внешний интерфейс так, чтобы использовать новый формат.

Однако в некоторых ситуациях удобнее сначала изменить API. В этом случае рефакторинг выполняется следующим образом.

1. Добавьте параметр в новом формате.
2. Обеспечьте преобразование параметра в новом формате во внутреннее представление, обладающее старым форматом.
3. Удалите параметр в старом формате.
4. Замените использование старого формата на использование нового формата.
5. Удалите старый формат.

Зачем

Проблема миграции данных возникает каждый раз, когда используется шаблон «От одного ко многим» (One to Many). Предположим, что мы хотим реализовать объект TestSuite, используя шаблон «От одного ко многим» (One to Many). Мы можем начать так:

```
def testSuite(self):
suite = TestSuite()
suite.add(WasRun("testMethod"))
suite.run(self.result)
assert("1 run, 0 failed" == self.result.summary())
```

Чтобы реализовать этот тест, начнем с одного элемента test:

```
class TestSuite:
def add(self, test):
self.test = test
def run(self, result):
self.test.run(result)
```

Теперь мы приступаем к дублированию данных. Вначале инициализируем коллекцию тестов:

TestSuite

```
def __init__(self):
self.tests = []
```

В каждом месте, где инициализируется поле test, добавляем новый тест в коллекцию:

TestSuite

```
def add(self, test):
self.test = test
self.tests.append(test)
```

Теперь мы используем коллекцию тестов вместо единичного теста. Исходя из существующего набора тестов данное преобразование можно считать рефакторингом (оно не нарушает семантику), так как в составе коллекции присутствует только один элемент.

TestSuite

```
def run(self, result):
for test in self.tests:
test.run(result)
```

Теперь можно удалить не используемую переменную экземпляра test:

TestSuite

```
def add(self, test):
self.tests.append(test)
```

Поэтапную миграцию данных можно использовать также при переходе между эквивалентными форматами, использующими различные протоколы, например, если речь идет о Java, при переходе от Vector/Enumerator к Collection/Iterator.

Выделение метода (Extract Method)

Как длинный сложный метод можно сделать простым для чтения? Выделите небольшую часть длинного метода в отдельный метод и обратитесь к этому методу из длинного метода.

Как

Выделение метода на самом деле является несколько более сложным атомарным рефакторингом. Здесь я опишу самый типичный случай. К счастью, многие среды разработки поддерживают автоматическое выполнение этого рефакторинга. Итак, чтобы выделить метод:

1. Определите фрагмент кода, который можно выделить в отдельный метод. Хорошими кандидатами являются тела циклов, сами циклы, а также ветви условных операторов.

2. Убедитесь, что внутри этого фрагмента не происходит присваивания значений временным переменным, объявленным вне области видимости, соответствующей этому фрагменту.

3. Скопируйте код из старого метода в новый. Скомпилируйте его.

4. Для каждой временной переменной или параметра первоначального метода, используемого в новом методе, добавьте параметр в новый метод.

5. Сделайте так, чтобы в нужном месте старый метод обращался к новому методу.

Зачем

Я использую «Выделение метода» (Extract Method), когда пытаюсь понять сложный код. «Значит так, этот кусок кода делает вот это. А этот кусок делает это. К чему мы там дальше обращаемся?» Через полчаса код будет выглядеть гораздо лучше, ваш партнер начнет понимать, что вы действительно оказываете ему помощь, а вы – существенно лучше понимать, что же все-таки происходит внутри кода.

Я использую выделение метода, чтобы избавиться от дублирования, когда вижу, что два метода обладают сходными участками кода. В этом случае я выделяю схожие участки в отдельный метод. (Браузер рефакторинга для Smalltalk – Refactoring Browser – выполняет еще более полезную задачу: он просматривает код в поисках метода, аналогичного коду, который вы намерены выделить, и в случае, если такой метод уже есть, предлагает использовать уже существующий метод вместо того, чтобы создавать новый.)

Разделение методов на множество мелких кусочков может зайти слишком далеко. Если я не вижу, куда идти дальше, я часто использую шаблон «Встраивание метода» (Inline Method), чтобы собрать код в одном месте и увидеть новый, более удобный способ разделения.

Встраивание метода (Inline Method)

Как можно упростить код, если становится сложно уследить за последовательностью передачи управления от метода к методу? Замените обращение к методу кодом этого метода.

Как

1. Скопируйте код метода в буфер обмена.
2. Вставьте код метода вместо обращения к методу.
3. Замените все формальные параметры фактическими. Если, например, вы передаете reader.getNext(), то есть выражение, обладающее побочным эффектом, будьте осторожны и присвойте полученное значение временной переменной.

Зачем

Один из моих рецензентов пожаловался на сложность кода в первой части книги, который требует от объекта Bank преобразовать объект Expression в объект Money.

```
public void testSimpleAddition() {
Money five = Money.dollar(5);
Expression sum = five.plus(five);
Bank bank = new Bank();
Money reduced = bank.reduce(sum, "USD");
assertEquals(Money.dollar(10), reduced);
}
```

«Это слишком сложно. Почему бы не реализовать преобразование в самом объекте Money?» Ну что же, поставим эксперимент. Как это сделать? Давайте встроим метод Bank.reduce() и посмотрим, как это будет выглядеть:

```
public void testSimpleAddition() {
Money five = Money.dollar(5);
Expression sum = five.plus(five);
Bank bank = new Bank();
Money reduced = sum.reduce(bank, «USD»);
assertEquals(Money.dollar(10), reduced);
}
```

Возможно, вторая версия понравится вам больше, возможно, нет. Важно понимать, что при помощи шаблона «Встраивание метода» (Inline Method) вы можете экспериментировать с последовательностью выполнения действий. Когда я выполняю рефакторинг, я формирую у себя в голове мысленную картину системы с кусками логики и потоком

выполнения программы, перетекающим от одного объекта к другому объекту. Когда мне кажется, что я вижу нечто многообещающее, я использую рефакторинг, чтобы попробовать это и увидеть результат.

В разгаре битвы я могу вдруг обнаружить, что попался в ловушку собственной гениальности. (Не буду говорить, насколько часто это происходит.) Когда это происходит, я использую «Встраивание метода» (Inline Method), чтобы разобраться в той путанице, которую я создал: «Так, этот объект обращается к этому, этот к этому… не могу понять, что же здесь происходит?» Я встраиваю несколько уровней абстракции и смотрю, что же на самом деле происходит. После этого я могу заново выделить абстракцию, использовав более удобный способ.

Выделение интерфейса (Extract Interface)

Как создать альтернативные реализации операций в языке Java? Создайте интерфейс, в котором будут содержаться общие операции.

Как

1. Напишите объявление интерфейса. Иногда в качестве имени интерфейса используется имя существующего класса. В этом случае вы должны предварительно переименовать класс.
2. Сделайте так, чтобы существующий класс реализовывал объявленный вами интерфейс.
3. Добавьте в интерфейс все обязательные методы. В случае необходимости измените режим видимости методов класса.
4. Там, где это возможно, измените объявления с класса на интерфейс.

Зачем

Иногда необходимость выделения интерфейса возникает в случае, когда вы переходите от одной реализации к другой. Например, у вас есть класс Rectangle (прямоугольник), и вы хотите создать класс Oval (овал) – в этом случае вы создаете интерфейс Shape (фигура). В подобных ситуациях подобрать имя для интерфейса, как правило, несложно. Однако иногда приходится изрядно помучиться, прежде чем обнаружится подходящая метафора.

Иногда, когда нужно выделить интерфейс, вы используете шаблон «Тестирование обработки ошибок» (Crash Test Dummy) или «Поддельный объект» (Mock Object). В этом случае подбор подходящего имени

выполняется сложнее, так как в вашем распоряжении лишь один пример использования интерфейса. В подобных случаях у меня возникает соблазн наплевать на информативность и назвать интерфейс IFile, а реализующий его класс – File. Однако я приучил себя останавливаться на мгновение и размышлять о том, достаточно ли хорошо я понимаю то, над чем работаю? Возможно, интерфейс лучше назвать File, а реализующий его класс – DiskFile, так как соответствующая реализация основана на том, что данные, содержащиеся в файле, хранятся на жестком диске.

Перемещение метода (Move Method)

Как можно переместить метод в новое место, где он должен находиться? Добавьте его в класс, которому он должен принадлежать, затем обратитесь к нему.

Как

1. Скопируйте метод в буфер обмена.
2. Вставьте метод в целевой класс. Присвойте ему подобающее имя. Скомпилируйте его.
3. Если внутри метода происходит обращение к первоначальному объекту, добавьте параметр, при помощи которого методу будет передаваться этот объект. Если внутри метода происходит обращение к переменным-членам первоначального объекта, передавайте их в виде параметров. Если внутри метода переменным-членам первоначального объекта присваиваются значения, вы должны отказаться от идеи переноса метода в новый объект.
4. Замените тело первоначального метода обращением к новому методу.

Зачем

Это один из моих самых любимых шаблонов рефакторинга, выполняемых в процессе консультирования. Дело в том, что он наиболее эффективно демонстрирует неправильные предположения относительно дизайна кода. Вычисление площади – это обязанность объекта Shape (фигура):

Shape

…

```
int width = bounds.right() – bounds.left();
int height = bounds.bottom() – bounds.top();
int area = width * height;
…
```

Каждый раз, когда я вижу, что внутри метода, принадлежащего одному объекту, происходит обращение к нескольким методам другого объекта, я начинаю смотреть на код с подозрением. В данном случае я вижу, что в методе, принадлежащем объекту Shape, происходит обращение к четырем методам объекта bounds (класс Rectangle). Пришло время переместить эту часть метода в класс Rectangle:

Rectangle

```
public int area() {
int width = this.right() – this.left();
int height = this.bottom() – this.top();
return width * height;
}
```

Shape

```
…
int area = bounds.area();
…
```

Шаблон рефакторинга «Перемещение метода» (Move Method) обладает тремя важными преимуществами:

• Очень легко увидеть необходимость применения этого вида рефакторинга, при этом не требуется глубокое понимание смысла кода. Как только вы увидите два или больше сообщения, адресованные другому объекту, значит, можно смело приступать.

• Механика выполнения рефакторинга быстра и безопасна.

• Результаты зачастую приводят к просветлению. «Но класс Rectangle не выполняет никаких вычислений… О! Теперь я вижу. Так действительно лучше.»

Иногда возникает желание переместить только часть метода. Вы можете вначале выделить метод, переместить весь метод, а затем встроить метод в первоначальный класс. Или вы можете придумать способ сделать все это за один шаг.

Метод в объект (Method Object)

Как лучше всего реализовать сложный метод, использующий несколько параметров и локальных переменных? Преобразуйте метод в отдельный объект.

Как

1. Создайте класс с таким же количеством параметров, как и оригинальный метод.
2. Сделайте локальные переменные метода переменными экземпляра нового класса.
3. Определите в новом классе метод с именем run(). Тело этого метода будет таким же, как и тело оригинального метода.
4. В оригинальном методе создайте новый объект и обратитесь к методу run() этого объекта.

Зачем

Объекты-методы полезны в качестве подготовительного этапа перед добавлением в систему абсолютно нового вида внутренней логики. Например, представьте, что для вычисления общего денежного потока используется несколько разных методов, позволяющих учесть в вычислениях несколько разных компонентов общего денежного потока. Вначале можно создать объект-метод, вычисляющий общий денежный поток первым способом. Затем можно описать следующий способ вычислений при помощи тестов меньшего масштаба. После этого добавление в программу нового способа вычислений будет несложным делом.

Объекты-методы также позволяют упростить код, в отношении которого неудобно использовать шаблон «Выделение метода» (Extract Method). В некоторых ситуациях вы вынуждены иметь дело с блоком кода, который работает с обширным набором временных переменных и параметров, и каждый раз, когда вы пытаетесь выделить хотя бы часть этого кода в отдельный метод, вы вынуждены переносить в новый метод пять или шесть временных переменных и параметров. Получившийся выделенный метод выглядит ничем не лучше, чем первоначальный код, так как его сигнатура слишком длинна. В результате создания объекта-метода вы получаете новое пространство имен, в рамках которого можете извлекать методы, без необходимости передачи в них каких-либо

параметров.

Добавление параметра (Add Parameter)

Как можно добавить в метод новый параметр?

Как

1. Если метод входит в состав интерфейса, сначала добавьте параметр в интерфейс.
2. Воспользуйтесь сообщениями компилятора, чтобы узнать, в каких местах происходит обращение к данному методу. В каждом из этих мест внесите необходимые изменения в вызывающий код.

Зачем

Добавление параметра зачастую связано с расширением функциональности. Чтобы обеспечить успешное выполнение первого теста, вы написали код без параметра, однако далее условия изменились, и для корректного выполнения вычислений необходимо принять во внимание дополнительные данные.

Добавление параметра также может быть вызвано необходимостью миграции от одного представления данных к другому. Вначале вы добавляете параметр, затем удаляете из кода все ссылки на старый параметр, затем удаляете сам старый параметр.

Параметр метода в параметр конструктора (Method Parameter to Constructor Parameter)

Как переместить параметр из метода или методов в конструктор?

Как

1. Добавьте параметр в конструктор.
2. Добавьте в класс переменную экземпляра с тем же именем, что и параметр.
3. Установите значение переменной в конструкторе.
4. Одну за другой преобразуйте ссылки parameter в ссылки this.parameter.
5. Когда в коде не останется ни одной ссылки на параметр, удалите

параметр из метода.

6. После этого удалите ненужный теперь префикс this.
7. Присвойте переменной подходящее имя.

Зачем

Если вы передаете один и тот же параметр нескольким разным методам одного и того же объекта, вы можете упростить API, передав параметр только один раз (устранив дублирование). Напротив, если вы обнаружили, что некоторая переменная экземпляра используется только в одном методе объекта, вы можете выполнить обратный рефакторинг.

27 Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley, 1999. Русское издание: Фаулер. М. *Рефакторинг: улучшение существующего кода*. СПб.: Символ-Плюс, 2003

32. Развитие навыков TDD

В данной главе я намерен сформулировать несколько вопросов, над которыми полезно подумать, если вы намерены интегрировать TDD в свой процесс разработки. Некоторые из вопросов просты, другие требуют тщательного обдумывания. Ответы на некоторые из этих вопросов вы найдете здесь же, однако иногда, чтобы ответить на некоторый вопрос, вам придется провести собственные исследования.

Насколько большими должны быть шаги?

В этом вопросе на самом деле скрыто два вопроса:

• Какой объем функциональности должен охватывать каждый тест?

• Как много промежуточных стадий должно быть преодолено в процессе каждого сеанса рефакторинга?

Вы можете писать тесты так, что каждый из них будет требовать добавления в функциональный код единственной строки и выполнения небольшого рефакторинга. Вы также можете писать тесты так, что каждый из них будет требовать добавления в функциональный код сотен строк, при этом у вас будут уходить часы на рефакторинг. Какой из этих путей лучше?

Часть ответа состоит в том, что вы должны уметь работать и так и этак. Общая тенденция TDD очевидна – чем меньше шаги, тем лучше. В данной книге мы занимались разработкой маленьких тестов на уровне отдельных фрагментов программы. Однако некоторые программисты экспериментируют в области разработки программ, исходя из тестов на уровне всего приложения.

Если вы только приступаете к освоению рефакторинга, вы должны двигаться маленькими шажками. Процесс ручного рефакторинга чреват ошибками. Чем больше ошибок вы наделаете, тем меньшим будет ваше желание выполнять рефакторинг в дальнейшем. После того как вы 20 раз проделаете рефакторинг малюсенькими шажками, можете приступать к экспериментам по удлинению этих шажков.

Автоматизация существенно ускоряет процессы, связанные с рефакторингом. То, что раньше требовало выполнения 20 маленьких шагов вручную, теперь становится единственным пунктом в меню. Количество изменений, выполняемых в ходе сеанса рефакторинга увеличивается на

порядок, и это неизменно сказывается на качестве. Когда вы знаете, что в вашем распоряжении великолепный инструмент, вы становитесь более агрессивным при выполнении рефакторинга. Вы пытаетесь ставить больше экспериментов и смотреть, какой способ структурирования кода лучше.

На момент написания данной книги браузер рефакторинга Refactoring Browser for Smalltalk по-прежнему является наилучшим инструментом в этой категории. В настоящее время многие среды разработки для Java поддерживают развитые средства рефакторинга. Кроме того, поддержка рефакторинга появилась и в других языках и средах разработки.

Что не подлежит тестированию?

Флип предложил высказывание, которое может служить ответом на этот вопрос: «Пишите тесты до тех пор, пока страх не превратится в скуку». Высказывание подразумевает, что вы должны найти ответ сами. Однако вы читаете эту книгу для того, чтобы найти в ней ответы на вопросы, поэтому попробуйте воспользоваться следующим списком. Тестировать следует:

- условные операторы;
- циклы;
- операции;
- полиморфизм.

Однако только те из них, которые вы написали сами. Не тестируйте чужой код, если только у вас нет причин не доверять ему. В некоторых ситуациях недостатки (можно сказать жестче: «ошибки») во внешнем коде заставляют добавлять дополнительную логику в разрабатываемый вами код. Надо ли тестировать подобное поведение внешнего кода? Иногда я документирую непредсказуемое поведение (ошибку) внешнего кода при помощи теста, который перестанет выполняться, если в следующей версии внешнего кода ошибка будет исправлена.

Как определить качество тестов?

Тесты – это канарейка, которую берут в угольную шахту, чтобы по ее поведению определить присутствие запаха плохого дизайна. Далее перечисляются некоторые атрибуты тестов, которые указывают на то, что дизайн тестируемого кода начинает плохо пахнуть:

• Длинный код инициализации. Если вы вынуждены написать сотни строк кода, создавая объекты для одного простого оператора assert(), значит, что-то не так, значит, ваши объекты слишком большие и их требуется разделить.

• Дублирование кода инициализации. Если вы не можете быстро найти общее место для общего кода инициализации, значит, у вас слишком много объектов, которые слишком тесно взаимодействуют друг с другом.

• Тесты выполняются слишком медленно. Если тесты TDD работают слишком медленно, значит, они не будут запускаться достаточно часто. Значит, программист будет в течение некоторого времени работать, вообще не запуская тестов. Значит, когда он их все-таки запустит, скорее всего, многие из них не сработают. На самом деле здесь кроется серьезная проблема: если тесты работают медленно, значит, тестирование частей и компонентов разрабатываемого приложения связано с проблемами. Сложности при тестировании частей и фрагментов приложения указывают на существование недостатков дизайна. Иными словами, улучшив дизайн, вы можете увеличить скорость работы тестов. (Продолжительность работы набора тестов не должна превышать десяти минут, по аналогии с ускорением свободного падения в 9,8 м/с2. Если для выполнения набора тестов требуется более 10 минут, этот набор обязательно надо сократить или тестируемое приложение должно быть оптимизировано так, чтобы для выполнения набора тестов требовалось не более 10 минут.)

• Хрупкие тесты. Если ваши тесты неожиданно начинают ломаться в самых непредсказуемых местах, это означает, что одна часть разрабатываемого приложения непредсказуемым образом влияет на работу другой части. В этом случае необходимо улучшить дизайн так, чтобы данный эффект исчез. Для этого можно либо устранить связь между частями приложения, либо объединить две части воедино.

Как TDD способствует созданию инфраструктур?

Инфраструктура (framework) – набор обобщенного кода, который можно использовать в качестве базы при разработке разнообразных прикладных программ. На самом деле TDD является неплохим инструментом разработки инфраструктур. Парадокс: если вы перестаете думать о будущем вашего кода, вы делаете код значительно более адаптируемым для повторного использования в будущем.

Очень многие умные книги говорят об обратном: «кодируйте для

сегодняшнего дня, но проектируйте для завтрашнего» (code for today, design for tomorrow). Похоже, что TDD переворачивает этот совет с ног на голову: «кодируйте для завтрашнего дня, проектируйте для сегодняшнего» (code for tomorrow, design for today). Вот что происходит на практике:

- В программу добавляется первая функциональность. Она реализуется просто и прямолинейно, поэтому реализация выполняется быстро и с наименьшим количеством дефектов.
- В программу добавляется вторая функциональность, которая является вариацией первой. Дублирование между двумя функциональностями объединяется и размещается в одном месте. Различия оказываются в разных местах (как правило, в разных методах или в разных классах).
- В программу добавляется третья функциональность, которая является вариацией первых двух. Уже имеющаяся общая логика, как правило, может использоваться в том виде, в котором она уже присутствует в программе, возможно, потребуется внести незначительные изменения. Отличающаяся логика должна располагаться в отдельном месте – в другом методе или в другом классе.

В процессе разработки мы постепенно приводим код в соответствие с принципом открытости/закрытости (Open/Closed Principle[28]), утверждающим, что объекты должны быть открыты для использования, но закрыты для модификации. Самое интересное, что при использовании TDD этот принцип выполняется именно для тех вариаций, с которыми действительно приходится иметь дело на практике. То есть TDD позволяет формировать инфраструктуры, удобные для представления таких вариаций, с необходимостью реализации которых программист сталкивается на практике. Однако эти инфраструктуры могут оказаться неэффективными в случае, если потребуется реализовать вариацию, которая редко встречается в реальности (или которая не была еще реализована ранее).

Что же произойдет, если необходимость реализации непредвиденной вариации возникнет спустя три года после разработки инфраструктуры? Дизайн быстро эволюционирует так, чтобы сделать вариацию возможной. Принцип открытости/закрытости на короткое время будет нарушен, однако это нарушение обойдется относительно недорого, так как имеющиеся тесты дадут вам уверенность в том, что, изменив код, вы ничего не поломаете.

В пределе, когда вариации возникают достаточно быстро, стиль TDD невозможно отличить от заблаговременного проектирования. Однажды я всего за несколько часов с нуля разработал инфраструктуру составления

отчетов. Те, кто следил за этим, были абсолютно уверены, что это трюк. Они думали, что я сел за разработку, уже имея в голове готовую инфраструктуру. Однако это не так. Просто я долгое время практиковал TDD, благодаря этому я исправляю допущенные мною многочисленные ошибки быстрее, чем вы успеваете заметить, что я их допустил.

Сколько должно быть тестов?

Насколько емкой должна быть обратная связь? Рассмотрим простую задачу: дано три целых числа, обозначающих длины сторон треугольника. Метод должен возвращать:

- 1 – в случае, если треугольник равносторонний;
- 2 – в случае, если треугольник равнобедренный;
- 3 – в случае, если треугольник не равносторонний и не равнобедренный.

Если длины сторон заданы некорректно (невозможно построить треугольник со сторонами заданной длины), метод должен генерировать исключение.

Вперед! Попробуйте решить задачу (мое решение, написанное на языке Smalltalk, приведено в конце данного подраздела).

Это отчасти напоминает игру «Угадай мелодию» («Я могу закодировать задачу за четыре теста!» – «А я – за три!» – «О’кей попробуйте».) Для решения задачи я написал шесть тестов, а Боб Биндер в своей книге *Testing Object-Oriented Systems*[29] («Тестирование объектно-ориентированных систем») для этой же самой задачи написал 65 тестов. Сколько на самом деле нужно тестов? Вы должны решить это сами, исходя из собственного опыта и рассуждений.

Когда я думаю о необходимом количестве тестов, я пытаюсь оценить приемлемое *среднее время между сбоями* (MTBF, Mean Time Between Failures). Например, в языке Smalltalk целые числа ведут себя как целые числа, а не как 32-битные значения. Иными словами, максимально возможное значение целого числа ограничивается не тридцатью двумя битами, а объемом памяти. Это означает, что вы можете обойтись без тестирования MAXINT. Безусловно, определенный предел существует, ведь теоретически можно создать целое число, для хранения которого не хватит имеющейся памяти. Но должен ли я тратить время на написание и реализацию теста, пытающегося заполнить память невероятно огромным целым числом? Как это повлияет на MTBF моей программы? Если я в

обозримом будущем не собираюсь иметь дело с треугольниками, размер сторон которых измеряется такими числами, значит, моя программа не станет существенно менее надежной, если я не реализую такой тест.

Имеет ли смысл писать тот или иной тест? Это зависит от того, насколько аккуратно вы оцените MTBF. Если обстоятельства требуют, чтобы вы увеличили MTBF от 10 лет до 100 лет, значит, имеет смысл уделить время для разработки самых маловероятных и чрезвычайно редко возникающих ситуаций (если, конечно, вы не можете каким-либо иным образом доказать, что подобные ситуации никогда не могут возникнуть).

Взгляд на тестирование в рамках TDD прагматичен. В TDD тесты являются средством достижения цели. Целью является код, в корректности которого мы в достаточной степени уверены. Если знание особенностей реализации без какого-либо теста дает нам уверенность в том, что код работает правильно, мы не будем писать тест. Тестирование черного ящика (когда мы намеренно игнорируем реализацию) обладает рядом преимуществ. Если мы игнорируем код, мы наблюдаем другую систему ценностей: тесты сами по себе представляют для нас ценность. В некоторых ситуациях это вполне оправданный подход, однако он отличается от TDD.

TriangleTest

```
testEquilateral
self assert: (self evaluate: 2 side: 2 side: 2) = 1

testIsosceles
self assert: (self evaluate: 1 side: 2 side: 2) = 2

testScalene
self assert: (self evaluate: 2 side: 3 side: 4) = 3

testIrrational
[self evaluate: 1 side: 2 side: 3]
on: Exception
do: [: ex | ^self].
self fail

testNegative
[self evaluate: -1 side: 2 side: 2]
on: Exception
```

```
do: [: ex | ^self].
self fail

testStrings
[self evaluate: ‘a’ side: ‘b’ side: ‘c’]
on: Exception
do: [: ex | ^self].
self fail

evaluate: aNumber1 side: aNumber2 side: aNumber3
| sides |
sides:= SortedCollection
with: aNumber1
with: aNumber2
with: aNumber3.
sides first <= 0 ifTrue: [self fail].
(sides at: 1) + (sides at: 2) <= (sides at: 3) ifTrue: [self fail].
^sides asSet size
```

Когда следует удалять тесты?

Чем больше тестов, тем лучше, однако если два теста являются избыточными по отношению друг к другу, должны ли вы сохранить оба этих теста в наборе? Это зависит от двух критериев.

• Первый критерий – это уверенность. Никогда не удаляйте тест, если в результате этого снизится ваша уверенность в поведении системы.

• Второй критерий – это коммуникация. Если у вас есть два теста, которые тестируют один и тот же участок кода, однако читателем эти тесты рассматриваются как два различных сценария, сохраните оба теста.

Отсюда следует, что, если у вас есть два теста, которые можно считать избыточными как в отношении уверенности, так и в отношении коммуникации, удалите наименее полезный из этих тестов.

Как язык программирования и среда разработки влияют на TDD?

Попробуйте использовать подход TDD в среде Smalltalk с браузером Refactoring Browser. Теперь попробуйте работать в среде C++ с редактором

vi. Почувствуйте разницу.

В языках программирования и средах разработки, в которых цикл TDD выполняется сложнее (тест – компиляция – запуск – рефакторинг), возникает тенденция двигаться вперед более длинными шагами:

- каждый тест охватывает больший объем кода;
- рефакторинг выполняется с меньшим количеством промежуточных шагов.

Приводит ли это к замедлению разработки, или, наоборот, разработка ускоряется?

В языках программирования и средах, в которых проще выполнить цикл TDD, у вас будет возникать желание больше экспериментировать с кодом. Позволит ли это двигаться быстрее и формировать лучшие решения, или вам кажется, что лучше тратить время на дополнительные размышления о дизайне?

Можно ли использовать TDD для разработки крупномасштабных систем?

Позволяет ли методика TDD разрабатывать крупномасштабные программные проекты? Какие новые типы тестов вам потребуется написать? Какие новые шаблоны рефакторинга могут потребоваться?

Самой крупной программной системой, целиком и полностью разработанной в стиле TDD, в создании которой я принимал участие, является система LifeWare (www.lifeware.ch). Работа над системой велась в течение 4 лет. Объем работ оценивается в 40 человеко-лет. На текущий момент система включает в себя 250 000 строк функционального и 250 000 строк тестирующего кода (на языке Smalltalk). Набор тестов системы включает в себя 4000 тестов, для выполнения которых требуется 20 минут. Полный набор тестов запускается несколько раз каждый день. Реализованный в системе огромный объем функциональности, похоже, никак не снижает эффективности TDD. Избавляясь от дублирования, вы стараетесь создать большое количество маленьких объектов, которые можно тестировать изолированно друг от друга вне зависимости от размера приложения.

Можно ли осуществлять разработку через тестирование на уровне приложения?

Если мы будем выполнять разработку, используя только внутренние программистские тесты (их называют тестами модулей – *unit tests*, – хотя они не вполне соответствуют этому определению), мы рискуем столкнуться с проблемой: полученная в результате этого система может оказаться не совсем тем или, что хуже, совсем не тем, что хочет получить пользователь. Программист будет работать над программой, которая, *по его мнению*, должна быть полезна, однако у пользователя может оказаться совершенно другое мнение. Чтобы решить проблему, можно разработать набор тестов на уровне приложения. Разработкой этих тестов должны заниматься сами пользователи (при поддержке программистов). Написанные пользователями тесты должны точно определять, что именно должна делать разрабатываемая система. Такой стиль можно назвать разработкой через тестирование на уровне приложения (ATDD, Application Test-Driven Development).

Встает техническая проблема: как написать и запустить тест для функциональности, которая еще не существует? Мне кажется, что всегда можно найти способ решения этой проблемы. Например, можно разработать интерпретатор, который будет вежливо сигнализировать о том, что обнаружен тест, выполнить который на данный момент невозможно по причине отсутствия в системе необходимых возможностей.

Существует также социальная проблема. У пользователей (на самом деле я имею в виду команду, в состав которой входят пользователи) появляется новая обязанность: разработка тестов. Процедура разработки тестов уровня приложения требует добавления дополнительного этапа в цикл работы над продуктом, – а именно, разработка пользовательских тестов выполняется перед началом реализации очередного объема функциональности. Организации часто сопротивляются подобному сдвигу ответственности. Новый этап требует координированных усилий множества членов команды, то есть перед тем, как приступить непосредственно к разработке кода, члены команды вынуждены потратить время на разработку пользовательских тестов.

Описанная в данной книге методика TDD целиком и полностью находится под вашим контролем. Иначе говоря, выполнение TDD зависит только от одного человека – от вас. Если у вас возникло желание, вы можете начать использовать ее с сегодняшнего дня. Однако если вы будете смешивать ритм красный – зеленый – рефакторинг с техническими, социальными и организационными проблемами разработки пользовательских тестов, вы вряд ли сможете добиться успеха. В данном случае следует воспользоваться правилом «Тест одного шага» (One Step

Test). Сначала добейтесь равномерности ритма красный – зеленый – рефакторинг в собственной практике, затем расширьте область применения TDD.

Еще один аспект ATDD: определение длины цикла между разработкой теста и получением результатов его работы. Если заказчик написал тест, а потом в течение десяти дней ждет его срабатывания, это значит, что он большую часть времени смотрит на красную полосу. Если я работаю в стиле TDD на уровне программиста, я

- немедленно получаю зеленую полосу;
- упрощаю внутренний дизайн.

Как перейти к использованию TDD в середине работы над проектом?

У вас есть некоторый объем кода, про который можно сказать, что он корректно работает в большей или меньшей степени. Теперь вы хотите разрабатывать весь новый код в рамках концепции TDD. Что делать?

О проблеме перехода на использование TDD в середине работы над проектом можно написать целую книгу (или даже несколько книг). В данном небольшом разделе я очень поверхностно затрону несколько связанных с этим вопросов.

Самая большая проблема заключается в том, что код, изначально написанный без тестов, как правило, сложен в тестировании. Интерфейсы и взаимосвязи между объектами недостаточно хорошо спроектированы, поэтому сложно изолировать некоторый кусок логики, запустить его и проверить результаты.

«Надо это исправить», – скажете вы. Да, однако любой рефакторинг (без применения средств автоматизации), скорее всего, приведет к возникновению ошибок, и эти ошибки сложно будет обнаружить, так как у вас нет тестов. Проблема яйца и курицы. Змея кусает себя за хвост. Замкнутый цикл саморазрушения. Что же делать?

Прежде всего скажу, чего делать не надо: не надо писать тесты для всего кода и выполнять рефакторинг всего кода. Для этого может потребоваться несколько месяцев, в течение которых у вас не будет времени добавить в систему новой функциональности. Если вы тратите имеющиеся у вас деньги и при этом не зарабатываете новых, долго вы не протянете.

Поэтому прежде всего мы должны ограничить область планируемых нами изменений. Если мы видим часть системы, которую можно

существенно улучшить, но которая вполне может быть оставлена без изменений на текущий момент, мы оставляем их без изменений. Возможно, взглянув на код и вспомнив грехи прошлого, вы не сможете удержаться от слез, однако возьмите себя в руки, – если код не требует немедленного вмешательства, лучше не изменять его.

Во-вторых, мы должны разорвать замкнутый круг между тестами и рефакторингом. Мы можем использовать в качестве обратной связи не только тесты, но и другие способы, например чрезвычайно осторожная работа в паре с партнером. Мы можем использовать обратную связь более высокого уровня, например тесты на уровне всей системы. Мы знаем, что такие тесты не являются адекватными, однако они прибавляют нам уверенности. Благодаря такой обратной связи мы можем сделать части кода, нуждающиеся в изменении, более удобными для внесения изменений.

Через некоторое время части системы, которые постоянно меняются, станут выглядеть так, как будто их разработали с использованием TDD. Мы медленно углубимся в дремучий лес наших старых ошибок.

Для кого предназначена методика TDD?

Каждая практика программирования явно или не явно базируется на системе ценностей. TDD не исключение. Если вам нравится лепить вместе куски кода, которые более-менее работают, и вы счастливы думать, что вам не придется возвращаться к полученному в результате этого коду в дальнейшем, значит, TDD – не для вас. Методика TDD базируется на очаровательно-наивном предположении программиста о том, что чем красивее код, тем вероятнее успех. TDD помогает вам обращать внимание на правильные вопросы в подходящие для этого моменты времени. Благодаря этому вы можете делать дизайн чище и модифицировать его по мере того, как перед вами встают новые обстоятельства.

Я сказал, что предположение наивное, однако, скорее всего, я преувеличил. На самом деле наивно предполагать, что чистый код – это все, что необходимо для успеха. Мне кажется, что хорошее проектирование – это лишь 20 % успеха. Безусловно, если проектирование будет выполнено из рук вон плохо, вы можете быть на 100 % уверены, что проект провалится. Однако приемлемый дизайн сможет обеспечить успех проекта только в случае, если остальные 80 % будут там, где им полагается быть.

С этой точки зрения TDD – чрезвычайно мощный инструмент. Если сравнивать со средним уровнем индустрии разработки программного

обеспечения, методика TDD позволяет писать код, содержащий значительно меньше дефектов, и формировать значительно более чистый дизайн. Те, кто стремится к изяществу, могут найти в TDD средство для достижения цели.

Методика TDD также подходит для тех, у кого формируется эмоциональная привязанность к коду. Когда я был молодым программистом, самым большим разочарованием для меня была ситуация, когда проект начинался с огромным воодушевлением, а затем, с течением времени, код становился все более отвратительным. Год спустя у меня, как правило, формировалось устойчивое желание уйти из проекта, чтобы никогда в жизни не иметь дела с этим гнусно пахнущим кодом. TDD позволяет с течением времени поддерживать уверенность в коде. По мере того как тестов становится все больше (а ваше мастерство тестирования улучшается), вы обретаете все большую уверенность в том, что система ведет себя именно так, как вам надо. По мере того как вы улучшаете дизайн, становится возможным все большее количество изменений. Моя цель заключается в том, чтобы через год работы мне было бы интереснее и приятнее работать над проектом, чем в самом начале проекта, и TDD помогает мне достигнуть этой цели.

Зависит ли эффективность TDD от начальных условий?

Складывается впечатление, что разработка идет гладко только в случае, если тесты выполняются в определенном порядке. Тогда мы можем наблюдать классическую последовательность красный – зеленый – рефакторинг – красный – зеленый – рефакторинг. Вы можете попробовать взять те же самые тесты, но реализовать их в другом порядке, и у вас возникнет ощущение, что вы не сможете, как прежде, выполнять разработку маленькими шажками. Действительно ли одна последовательность тестов на порядок быстрее/проще в реализации, чем другая последовательность? Существуют ли какие-либо признаки тестов, которые могут подсказать, в какой последовательности их следует реализовать? Если методика TDD чувствительна к начальным условиям в малом масштабе, можно ли считать ее предсказуемой в более крупном масштабе? (Вот аналогия: отдельные потоки реки Миссисипи непредсказуемы, однако вы можете с уверенностью сказать, что через устье реки протекает приблизительно 2 000 000 кубических футов воды в секунду.)

Как методика TDD связана с шаблонами?

Все мои технические публикации – это поиск фундаментальных правил, которые позволяют обычным людям действовать так, как действуют эксперты. Это связано с тем, как я сам осваиваю то или иное ремесло, – я нахожу эксперта, которому можно подражать, и постепенно выясняю, что, собственно, он делает. Определенно, я не предполагаю, что сформулированные мною правила должны использоваться автоматически, однако именно так и происходит.

Моя старшая дочь (привет, Бетани! Я же говорил тебе, что ты попадешь в мою книгу, – не беспокойся, это не слишком обременительно) в течение семи лет пыталась научиться быстро перемножать числа. Как я, так и моя жена, когда были маленькими, научились этому за значительно более короткий срок. В чем дело? Оказывается каждый раз, когда перед Бетани вставала задача умножить, например, 6 на 9, она складывала число 6 девять раз (или число 9 шесть раз). Таким образом, можно сказать, что Бетани вообще не умела умножать числа так, как это делают другие люди, однако при этом она необычайно быстро складывала числа.

Я обратил внимание на один важный эффект, который, я надеюсь, смогут принять во внимание и другие. Если на основе постоянно повторяющихся действий формулируются правила, дальнейшее применение этих правил становится неосознанным и автоматическим. Естественно, ведь это проще, чем обдумывать все «за» и все «против» того или иного действия с самого начала. Благодаря этому повышается скорость работы, и, если в дальнейшем вы сталкиваетесь с исключением или проблемой, которая не вписывается ни в какие правила, у вас появляется дополнительное время и энергия для того, чтобы в полной мере применить свои творческие способности.

Именно это произошло со мной, когда я писал книгу *Smalltalk Best Practice Patterns*. В какой-то момент я решил просто следовать правилам, описываемым в ней. В начале это несколько замедлило скорость моей работы – мне требовалось дополнительное время, чтобы вспомнить то или иное правило или написать новое. Однако по прошествии недели я заметил, что с моих пальцев почти мгновенно слетает код, над разработкой которого ранее мне приходилось некоторое время размышлять. Благодаря этому у меня появилось дополнительное время для анализа и важных размышлений о дизайне.

Существует еще одна связь между TDD и шаблонами: TDD является

методом реализации дизайна, основанного на шаблонах. Предположим, что в определенном месте разрабатываемой системы мы хотим реализовать шаблон «Стратегия» (Strategy). Мы пишем тест для первого варианта и реализуем его, создав метод. После этого мы намеренно пишем тест для второго варианта, ожидая, что на стадии рефакторинга мы придем к шаблону «Стратегия» (Strategy). Мы с Робертом Мартином занимались исследованием подобного стиля TDD. Проблема состоит в том, что дизайн продолжает вас удивлять. Идеи, которые на первый взгляд кажутся вам вполне уместными, позже оказываются неправильными. Поэтому я не рекомендую целиком и полностью доверять своим предчувствиям относительно шаблонов. Лучше думайте о том, что, по-вашему, должна делать система, позвольте дизайну оформиться так, как это необходимо.

Почему TDD работает?

Приготовьтесь покинуть галактику. Предположите на секунду, что TDD помогает командам разработчиков создавать хорошо спроектированные, удобные в сопровождении системы с чрезвычайно низким уровнем дефектов. (Я не утверждаю, что это происходит на каждом шагу, я просто хочу, чтобы вы немножко помечтали.) Как такое может происходить?

Отчасти этот эффект связан с уменьшением количества дефектов. Чем раньше вы найдете и устраните дефект, тем дешевле это вам обойдется. Иногда разница в затратах огромна (спросите у «Марс-лендера»[30]). Снижение количества дефектов вызывает множество вторичных психологических и социальных эффектов. После того как я начал работать в стиле TDD, программирование стало для меня значительно менее нервным занятием. Когда я работаю в стиле TDD, мне не надо беспокоиться о множестве вещей. Вначале я могу заставить работать только один тест, потом – все остальные. Уровень стресса существенно снизился. Взаимоотношения с партнерами по команде стали более позитивными. Разработанный мною код перестал быть причиной сбоев, люди стали в большей степени рассчитывать на него. У заказчиков тоже повысилось настроение. Теперь выпуск очередной версии системы означает новую функциональность, а не набор новых дефектов, которые добавляются к уже существующим.

Уменьшение количества дефектов. Имею ли я право утверждать, что подобное возможно? Есть ли у меня научное доказательство?

Нет. На текущий момент не проводилось никаких исследований, подтверждающих преимущества TDD по сравнению с альтернативными подходами в смысле качества, эффективности или удовольствия. Однако эпизодические подтверждения преимуществ TDD многочисленны, а вторичные эффекты очевидны. При использовании TDD у программистов действительно снижается стресс, в командах действительно повышается доверие, а заказчики действительно начинают смотреть на каждую новую версию продукта с энтузиазмом. Лично мне не приходилось сталкиваться с обратными эффектами. Однако ваши наблюдения могут оказаться иными, вы должны попробовать для того, чтобы оценить TDD самостоятельно.

Еще одним преимуществом методики TDD, объясняющим ее положительные эффекты, является сокращение времени, которое проходит между принятием проектного решения и проверкой результата его реализации. В рамках TDD это достаточно короткий промежуток времени – несколько секунд или минут. Вы принимаете решение, реализуете его в коде, запускаете тесты и анализируете полученный результат. В начале у вас возникает мысль – возможно, API должен выглядеть так, или, возможно, метафора должна быть такой, – затем вы создаете самый первый пример – тест, который воплощает вашу мысль в реальность. Вместо того чтобы сначала проектировать, а затем в течение нескольких недель или месяцев ожидать, окажется ваше решение правильным или нет, вы получаете результат уже через несколько секунд или минут.

Причудливый ответ на вопрос, «Почему TDD работает?», основан на бредовом видении из области комплексных систем. Неподражаемый Флип пишет:

Следует использовать программистские практики, которые «притягивают» корректный код как предельную функцию, но не как абсолютную величину. Если вы пишете тесты для каждой присутствующей в системе функциональности, если вы добавляете в систему новые функции по одной, и только после того, как выполнятся все тесты, вы создадите то, что математики обозначают термином «точка притяжения (аттрактор)». Точка притяжения – это точка в пространстве состояний, к которой сходятся все потоки. Со временем код с большей вероятностью изменяется в лучшую сторону, а не в худшую; точка притяжения приближается к корректности, как предельная функция.

Это «корректность», которая устраивает всех программистов (за исключением, конечно же, тех, кто работает над медицинским или аэрокосмическим программным обеспечением). Я считаю, что важно

быть знакомым с концепцией точки притяжения – ее не следует отвергать, ею не следует пренебрегать.

Что означает название?

Название методики: Test-Driven Development – *разработка через тестирование*. Буквально можно перевести как «разработка, ведомая тестами» или «разработка исходя из тестов».

Development (разработка) – старый поэтапный подход к разработке программного обеспечения обладает рядом недостатков, так как оценить результат проектного решения очень сложно, если решение и оценка результатов удалены друг от друга по времени. В названии TDD термин «разработка» означает сложную комбинацию анализа, логического проектирования, физического проектирования, реализации, тестирования, пересмотра, интеграции и выпуска.

Driven (исходя из, через) – в свое время я называл TDD термином *test-first programming* (программирование «вначале тесты»). Однако антонимом слова *fist* (вначале) является слово *last* (в конце). Огромное количество людей осуществляют тестирование уже после того, как они запрограммировали функциональный код. Этот подход считается вполне приемлемым. Существует любопытное правило именования, согласно которому противоположность придуманного вами имени должна быть, по крайней мере отчасти, неприятной или неудовлетворительной. (Термин «структурное программирование» звучит привлекательно, так как никто не хочет писать бесструктурный, то есть неорганизованный код.) Если в ходе разработки я исхожу не из тестов, то из чего? Из предположений? Домыслов? Спецификаций? (Обратите внимание, что слово «спецификация» немножко похоже на слово «спекуляция».)

Test (тест) – автоматическая процедура, позволяющая убедиться в работоспособности кода. Нажмите кнопку, и он будет выполнен. Ирония TDD состоит в том, что это вовсе не методика тестирования. Это методика анализа, методика проектирования, фактически методика структурирования всей деятельности, связанной с разработкой программного кода.

Как методика TDD связана с практиками экстремального программирования?

Некоторые из рецензетов данной книги, были обеспокоены тем, что книга целиком и полностью посвящена TDD, в результате читатели могут подумать, что остальными практиками XP (eXtreme Programming – экстремальное программирование) можно пренебречь. Например, если вы работаете в стиле TDD, должны ли вы при этом работать в паре? Далее я привожу перечень соображений относительно того, как остальные практики XP улучшают эффективность TDD и, наоборот, как TDD повышает эффективность использования других практик XP.

Программирование в паре. Тесты, разрабатываемые в рамках TDD, являются превосходным инструментом общения, когда вы программируете в паре. Зачастую, работая в паре, партнеры не могут договориться – какую именно проблему они решают, несмотря на то что работают с одним и тем же кодом. Это звучит бредово, однако подобное происходит постоянно, в особенности когда вы только осваиваете работу в паре. Именно этой проблемы удается избежать благодаря TDD. Существует и обратное влияние: когда вы работаете в паре, у вас есть помощник, который может взять на себя нагрузку в случае, если вы устали. Ритм TDD может исчерпать ваши силы, и тогда вы будете вынуждены программировать, несмотря на усталость. Однако если вы работаете в паре, ваш партнер готов взять у вас клавиатуру и тем самым дать вам возможность немного расслабиться.

Работа на свежую голову. XP рекомендует работать, когда вы полны сил, и останавливать работу, когда вы устали. Если вы не можете заставить следующий тест сработать или заставить работать те два теста одновременно, значит, настало время прерваться. Однажды мы с дядей Бобом Мартином (Bob Martin[31]) работали над алгоритмом разбиения линии, и нам никак не удавалось заставить его работать. Несколько минут мы безуспешно бились над кодом, однако нам стало очевидно, что прогресса нет, поэтому мы просто остановили работу.

Частая интеграция. Тесты – это великолепный ресурс, который позволяет выполнять интеграцию значительно чаще. Вы добились успешного выполнения очередного теста, избавились от дублирования, значит, вы можете интегрировать код. Этот цикл может повторяться каждые 15–30 минут. Возможность частой интеграции позволяет более многочисленным командам разработчиков иметь дело с одной и той же базой исходного кода. Как сказал Билл Уэйк (Bill Wake): «Проблема *n2* не является проблемой, если *n* всегда равно 1».

Простой дизайн. В рамках TDD вы пишете только тот код, который необходим для успешного выполнения тестов, вы удаляете из него любое

дублирование, значит, вы автоматически получаете код, который идеально адаптирован к текущим требованиям и подготовлен к любым будущим пожеланиям. Общая доктрина требует, чтобы дизайна было достаточно для получения идеальной архитектуры для текущей системы. Эта доктрина также облегчает разработку тестов.

Рефакторинг. Устранение дублирования – это основная цель рефакторинга. Тесты дают вам уверенность в том, что поведение системы не изменится даже в случае, если в ходе рефакторинга вы вносите достаточно крупномасштабные изменения. Чем выше ваша уверенность, тем более агрессивно вы выполняете рефакторинг. Рефакторинг продлевает жизнь вашей системе. Благодаря рефакторингу вы упрощаете дальнейшую разработку тестов.

Частые выпуски версий. Если тесты TDD действительно улучшают MTBF вашей системы (в этом вы можете убедиться сами), значит, вы можете чаще внедрять разрабатываемый код в реальные производственные условия и при этом не наносить ущерба вашим заказчикам. Гарет Ривс (Gareth Reeves) приводит аналогию с куплей-продажей ценных бумаг на бирже в течение дня. Если вы занимаетесь краткосрочной спекуляцией ценными бумагами, в конце торгового дня вы должны продать все имеющиеся у вас ценные бумаги, так как вы не хотите принимать риск, связанный с сохранением некоторых ценных бумаг до следующего торгового дня, – этот риск вам не подконтролен. Разрабатывая систему, вы хотите, чтобы вносимые вами изменения как можно быстрее были опробованы в реальных производственных условиях, так как не хотите тратить время на разработку кода, в полезности которого не уверены.

Нерешенные проблемы TDD

Дарач Эннис (Darach Ennis) бросил вызов поклонникам TDD, размышляющим о возможностях расширения области применения TDD. Он сказал:

Множество различных организаций сталкивается с многочисленными проблемами TDD, и эти проблемы никак не затронуты в книге. Возможно, эти проблемы вообще никак не решить в рамках TDD. Вот некоторые из них:

• не существует способа автоматического тестирования GUI (например, Swing, CGI, JSP/Servlets/Struts);

• не существует способа автоматического тестирования

распределенных объектов (например, RPC, Messaging, CORBA/EJB и JMS);

• TDD нельзя использовать для разработки схемы базы данных (например, JDBC);

• нет необходимости тестировать код, разработанный сторонними разработчиками, или код, генерируемый внешними инструментами автоматизации разработки;

• TDD нельзя использовать для разработки компилятора/интерпретатора языка программирования.

Я не уверен, что он прав, но я также не уверен, что он не прав. В любом случае это почва для размышлений о дальнейшем развитии TDD.

Приложение I
Диаграммы взаимовлияния

В данной книге можно встретить несколько примеров диаграмм взаимовлияния. Идея диаграмм взаимовлияния позаимствована из серии книг *Quality Software Management* Джеральда Вейнберга (Gerald Weinberg), точнее говоря, из книги 1: *Systems Thinking*[32]. Цель диаграммы взаимовлияния – продемонстрировать, каким образом элементы системы влияют друг на друга.

Диаграмма взаимовлияния включает элементы трех типов:

• Действие или деятельность[33] – обозначается словом или короткой фразой.

• Положительное соединение – обозначается стрелкой, указывающей от одного действия к другому действию. Положительное соединение сообщает, что усиление интенсивности исходного действия приводит к усилению интенсивности целевого действия, а снижение интенсивности исходного действия приводит к снижению интенсивности целевого действия.

• Отрицательное соединение – обозначается стрелкой между двумя действиями, поверх которой нарисован кружочек. Отрицательное соединение сообщает, что усиление интенсивности исходного действия ведет к снижению интенсивности целевого действия, и, наоборот, снижение интенсивности исходного действия приводит к усилению интенсивности целевого действия.

Слишком много слов для очень простой концепции. На рис. П1.1–П1.3 приводится несколько примеров диаграмм взаимовлияния.

Рис. П1.1. Два действия, которые, по всей видимости, не влияют друг на друга

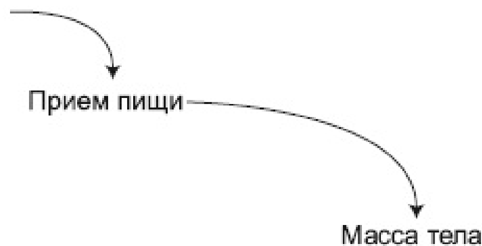

Рис. П1.2. Два действия, связанные положительным соединением

Рис. П1.3. Два действия, связанные отрицательным соединением

Чем больше я ем, тем больше моя масса тела. Чем меньше я ем, тем меньше моя масса тела. Конечно же, масса человеческого тела – это значительно более сложная система. Диаграммы взаимовлияния – это модели, которые помогают понять некоторые аспекты системы, однако они вовсе не предназначены для того, чтобы понимать и контролировать систему в полной мере.

Обратная связь

Влияние распространяется не только в одном направлении. Действие может быть связано обратной связью само с собой. Иначе говоря, в некоторых случаях изменение интенсивности действия влияет на само это действие. Иногда это влияние положительно, а иногда – отрицательно. Пример подобной обратной связи продемонстрирован на рис. П1.4.

Существует два типа обратной связи – положительная и отрицательная. Положительная обратная связь приводит к тому, что интенсивность действия в системе

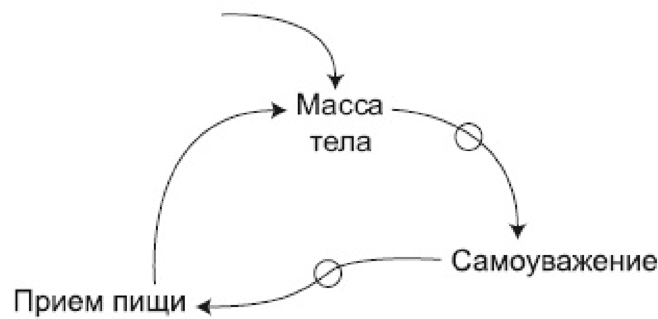

Рис. П1.4. Обратная связь

постоянно увеличивается. Чтобы обнаружить положительную обратную связь, достаточно посчитать количество отрицательных соединений в цикле. Если в цикле четное количество отрицательных соединений, значит, этот цикл является циклом положительной обратной связи. На рис. П1.4 изображен цикл положительной обратной связи: попав в этот цикл, вы продолжаете набирать вес, пока в составе цикла не появится какое-либо дополнительное действие.

Отрицательная обратная связь снижает интенсивность действия. Если в цикле присутствует нечетное количество отрицательных соединений, значит, цикл является циклом отрицательной обратной связи.

Вот три ключа хорошего системного дизайна:

• создание благоприятных циклов, в которых положительная обратная связь приводит к увеличению интенсивности полезных действий;

• устранение вредных циклов, в которых положительная обратная связь приводит к увеличению интенсивности бесполезных, вредных и деструктивных действий;

• создание циклов негативной обратной связи, которые предотвращают чрезмерное использование благоприятных действий.

Контроль над системой

Выбирая систему практик разработки программного обеспечения, добивайтесь, чтобы каждая практика способствовала применению других практик, благодаря этому вы сможете использовать каждую из практик в достаточном объеме даже в состоянии стресса. На рис. П1.5 показан пример системы практик, которая приводит к недостаточному тестированию.

Когда время начинает поджимать, вы снижаете интенсивность тестирования, что приводит к увеличению количества ошибок, что, в свою очередь, приводит к еще большему недостатку времени. Со временем на сцене появляется некоторое внешнее действие (например, недостаток денег), которое заставляет вас завершить работу над проектом, несмотря ни на что.

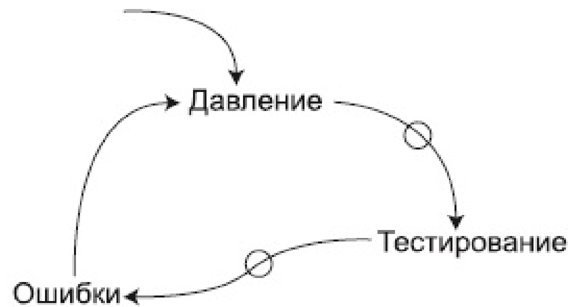

Рис. П1.5. Недостаток времени для тестирования приводит к общему недостатку времени

Если вы имеете дело с системой, которая ведет себя не так, как вам того хотелось бы, у вас есть несколько вариантов исправить ситуацию:

• Сформируйте цикл положительной обратной связи в обратном направлении. Если у вас цикл между тестами и уверенностью и тесты все время терпят неудачу, снижая тем самым уверенность, тогда вы сможете сделать больше успешных тестов, повысив тем самым уверенность в вашей способности увеличить количество работающих тестов.

• Сформируйте цикл отрицательной обратной связи, который позволит вам контролировать действие, интенсивность которого стала слишком большой.

• Создайте или разорвите соединения, чтобы устранить циклы, не являющиеся полезными.

Приложение II
Фибоначчи

В ответ на просьбу одного из моих рецензентов я включил в книгу описание разработки функции вычисления последовательности Фибоначчи в стиле TDD. Некоторые утверждают, что именно этот пример раскрыл им глаза на механику работы TDD. Однако этот пример очень короток, к тому же в нем не используются многие важные приемы, применяемые в рамках TDD. По этой причине его невозможно использовать в качестве замены примеров, рассмотренных ранее в данной книге. Если, ознакомившись с рассмотренными ранее примерами, вы до сих пор не можете понять, как осуществляется разработка в стиле TDD, ознакомьтесь с данным материалом, возможно, он поможет вам прояснить ситуацию.

Первый тест показывает, что fib(0) = 0. Реализация возвращает константу.

```
public void testFibonacci() {
assertEquals(0, fib(0));
}
int fib(int n) {
return 0;
}
```

(Я использую класс TestCase как вместилище кода, так как мы разрабатываем всего одну функцию.)

Второй тест показывает, что fib(1) = 1.

```
public void testFibonacci() {
assertEquals(0, fib(0));
assertEquals(1, fib(1));
}
```

Я просто добавил еще один оператор assert() в тот же самый тестовый метод, так как не вижу особого смысла создавать новый метод с именем testFibonacciOfOneIsOne.

Чтобы заставить тест работать, можно воспользоваться одним из нескольких методов. Я решаю использовать значение 0 как специальный случай:

```
int fib(int n) {
if (n == 0) return 0;
return 1;
}
```

Дублирование в тестирующем методе начинает действовать мне на нервы. По мере добавления новых тестов, дублирование будет только усугубляться. Давайте попробуем выделить общую структуру операторов assert(), для этого добавим в тест таблицу входных и ожидаемых значений функции fib():

```
public void testFibonacci() {
int cases[][] = {{0,0},{1,1}};
for (int i = 0; i < cases.length; i++)
assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Теперь добавление нового теста требует всего шесть нажатий на клавиши и никаких дополнительных строк:

```
public void testFibonacci() {

int cases[][] = {{0,0},{1,1},{2,1}};
for (int i = 0; i < cases.length; i++)
assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Как это ни удивительно, но новый тест работает. Это происходит потому, что константа 1 также подходит и для входного значения 2. Переходим к следующему тесту:

```
public void testFibonacci() {
int cases[][] = {{0,0},{1,1},{2,1},{3,2}};
for (int i = 0; i < cases.length; i++)
assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Ура! Наконец-то тест не сработал. Воспользуемся прежней стратегией (рассматриваем меньшие входные значения как специальные случаи):

```
int fib(int n) {
if (n == 0) return 0;
if (n <= 2) return 1;
return 2;
}
```

Теперь мы готовы обобщить код. Мы написали 2, однако на самом деле мы имели в виду 1 + 1.

```
int fib(int n) {
if (n == 0) return 0;
if (n <= 2) return 1;
return 1 + 1;
}
```

Первая единица в сумме – на самом деле fib(n-1):

```
int fib(int n) {
if (n == 0) return 0;
if (n <= 2) return 1;
return fib(n-1) + 1;
}
```

Вторая единица в сумме – на самом деле fib(n-2):

```
int fib(int n) {
if (n == 0) return 0;
if (n <= 2) return 1;
return fib(n-1) + fib(n-2);
}
```

Теперь окончательно чистим код. Та же самая структура должна работать для fib(2), поэтому мы можем преобразовать второй условный оператор:

```
int fib(int n) {
```

```
if (n == 0) return 0;
if (n == 1) return 1;
return fib(n-1) + fib(n-2);
}
```

Это и есть функция вычисления последовательности Фибоначчи, целиком и полностью разработанная в рамках методики TDD.

Послесловие

Мартин Фаулер (Martin Fowler)

Когда рассказываешь о разработке, основанной на тестировании, сложнее всего передать то психическое состояние, в котором находишься, работая в стиле TDD. Я помню, как в ходе проекта C3 мы с Ральфом Битти (Ralph Beattie) работали над реализацией сложного набора условий выплаты. Ральф сформулировал набор соответствующих тестов, после чего мы приступили к реализации этих тестов одного за другим. Процесс был равномерным и неторопливым, из-за этого казалось, что мы работаем медленно. Однако, взглянув назад на проделанную работу, можно было понять, что, несмотря на кажущуюся неторопливость, мы работали очень даже быстро.

Несмотря на множество появившихся в последнее время мощных инструментов, программирование по-прежнему остается сложной работой. Я часто ощущаю себя в ситуации, когда мне кажется, что я жонглирую шариками и мне приходится следить сразу за несколькими шариками в воздухе: малейшая потеря внимания, и все сыпется на пол. Методика TDD позволяет избавиться от этого ощущения.

Когда вы работаете в стиле TDD, в воздухе постоянно находится лишь один шарик. Вы можете сконцентрироваться на нем, а значит, хорошо справиться со своей работой. Когда я добавляю в программу новую функциональность, я не думаю о том, какой дизайн должен быть реализован в данной функции. Я просто пытаюсь добиться успешного выполнения тестов самым простым из доступных мне способов. Когда я переключаюсь в режим рефакторинга, я не беспокоюсь о добавлении в программу новых функций, я думаю только о правильном дизайне. На каждом из этих этапов я концентрируюсь на единственной задаче, благодаря этому мое внимание не распыляется.

Добавление новой функциональности при помощи тестов и рефакторинг – это две монологические разновидности программирования. Совсем недавно я открыл еще одну разновидность: копирование шаблона. Я занимался разработкой сценария на языке Ruby, извлекающего информацию из базы данных. Я начал с создания класса, являющегося оболочкой таблицы базы данных, а затем сказал себе, что, раз я только что закончил книгу о шаблонах работы с базами данных, я должен

использовать шаблон. Примеры программ в книге были написаны на Java, поэтому нужный мне код легко можно было перенести на Ruby. Когда я программировал, я не думал о решении проблемы, я думал лишь о том, как лучше всего адаптировать шаблон для условий, в рамках которых я работал.

Копирование шаблонов само по себе не является хорошим программированием, – я всегда подчеркиваю этот факт, когда говорю о шаблонах. Любой шаблон – это полуфабрикат, – вы должны адаптировать его для условий своего проекта. Однако чтобы сделать это, лучше всего вначале, особо не задумываясь, скопировать шаблон, а затем, воспользовавшись смесью рефакторинга и TDD, выполнить адаптацию. В этом случае в процессе копирования шаблона вы также концентрируетесь только на одной вещи – на шаблоне.

Сообщество XP активно работает над добавлением шаблонов в общую картину. Со всей очевидностью можно сказать, что сообщество XP любит шаблоны. В конце концов, между множеством приверженцев XP и множеством приверженцев шаблонов существует значительное пересечение: Уорд и Кент являются лидерами обоих направлений. Наверное, копирование шаблона – это третий монологический режим программирования наряду с разработкой в стиле «сначала тесты» и рефакторингом. Как и первые два режима, копирование шаблона – опасная штука, если ее использовать отдельно от двух других режимов. Все три вида программирования проявляют свою мощь только тогда, когда используются совместно друг с другом.

Если вы хотите сделать некоторый процесс эффективным, вы должны идентифицировать основные действия, из которых состоит процесс, а затем добиться, чтобы в каждый момент времени внимание концентрировалось только на одном таком действии. Примером такого подхода является сборочная линия, где каждый рабочий выполняет только одну из множества необходимых процедур. Внимание каждого рабочего сконцентрировано только на одном действии. Методика разработки через тестирование (TDD) подразумевает разделение процесса программирования на элементарные режимы, однако при этом она избавляет от монотонности, позволяя быстро переключаться между этими режимами. Комбинация монологических режимов и переключения между ними обеспечивает должную концентрацию внимания, снижает стресс и избавляет от монотонности сборочной линии.

Я признаю, что все эти мысли несколько сыроваты. Когда я пишу это, я по-прежнему не уверен в том, о чем рассказываю. Я знаю, что буду

обдумывать все эти идеи еще в течение нескольких, а может быть, и многих месяцев. Однако я полагаю, что эти идеи должны вам понравиться. Прежде всего, они стимулируют размышления о более крупной картине, в которую вписывается разработка через тестирование. Мы еще не видим эту картину достаточно четко, однако мне кажется, что она постепенно становится все яснее и яснее.

---

notes

Примечания

1

Бек К. *Экстремальное программирование*. СПб.: Питер, 2002. ISBN 5-94723-032-1.

2

Подробнее о подсистеме отчетов рассказано на c2.com/doc/oopsla91.html.

3

USD – доллары США, CHF – швейцарские франки. – *Примеч. пер*.

4

Название метода times() можно перевести на русский как «умножить на». – *Примеч. пер.*

5

Код с душком (code that smells) – распространенная в XP метафора, означающая плохой код (содержащий дублирование). – *Примеч. пер.*

6

Имеется в виду индикатор успешного выполнения тестов в среде JUnit, имеющий форму полосы. Если все тесты выполнились успешно, полоса становится зеленой. Если хотя бы один тест потерпел неудачу, полоса становится красной. – *Примеч. пер*.

7

В переводе на русский язык *sum* – это сумма. – *Примеч. пер*.

8

Используя игру слов (английское change означает как «изменение», так и «обмен»), автор намекает на свою знаменитую книгу-бестселлер *Extreme Programming Explained: Embrace Change*. Русский перевод: Бек К. Экстремальное программирование. СПб.: Питер, 2002. 224 с. – *Примеч. ред*.

9

Fractals and Scaling in Finance / Benoit Mandelbrot, editor. SpringerVerlag, 1997. ISBN: 0387983635

10

В переводе с английского языка *was run* означает *был выполнен*. – *Примеч. пер*.

11

В языке Python заголовок определения метода (функции) начинается со служебного слова def, а завершается двоеточием. Операторы тела метода записываются ниже, в отдельных строчках. Группировка операторов тела определяется отступами (вместо фигурных скобок). – *Примеч. ред.*

12

Спасибо Дункану Бусу (Duncan Booth) за то, что он исправил допущенную мной ошибку, типичную для малоопытных программистов на Python, и подсказавшему мне решение, в большей степени соответствующее этому языку.

13

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

14

Имеются в виду *пожелания пользователей* (user stories). Другой вариант перевода: *пользовательские истории*. – *Примеч. пер.*

15

Спасибо Джиму Ньюкирку (Jim Newkirk) и Лорану Боссави (Laurent Bossavit) за то, что независимо друг от друга предложили мне этот шаблон.

16

Например, www.mockobjects.com.

17

Спасибо Дирку Кенигу (Dierk König) за пример.

18

Спасибо Лорану Боссави за дискуссию.

19

Обычно необязательные параметры располагаются в конце списка аргументов, однако в данном случае необязательная информационная строка размещается в начале списка, так как благодаря этому тесты удобнее читать.

20

В переводе с английского *fixture* означает арматура, оснастка, зафиксированная деталь некоторого движущегося механизма. – *Примеч. пер.*

21

McConnell, Steve. *Code Complete*, chapter 4. Seattle, Washington: Microsoft Press. 1993.

22

Caine, S. H., Gordon, E. K. 1975. PDL: *A Tool for Software Design, AFIPS Proceedings of the 1975 National Computer Conference.*

23

В среде англоязычных программистов запись в базе данных иногда обозначается термином *tuple – кортеж*. – *Примеч. пер.*

24

Alexander Christopher. *Noteson the Synthesis of Form.* Cambridge, MA: Harvard University Press, 1970.

25

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns: Elements of Reusable Object Oriented Software.* Boston: Addisoh-Wesley, 1995. Русское издание: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. *Приемы объектно-ориентированного проектирования. Паттерны проектирования.* СПб.: Питер, 2001.

26

Подробнее об этом шаблоне рассказывается в книге Beck, K. *The Smalltalk Best Practice Patterns*. Pp. 70–73. Englewood-Cliffs, NJ: Prentice-Hall, 1997. Ссылаться на свои собственные работы – это не самая лучшая идея, однако, как говорил философ Филлис Диллер (Phyllis Diller): «Конечно же, я смеюсь над собственными шутками, просто я не доверяю никому, кроме себя».

27

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley, 1999. Русское издание: Фаулер. М. *Рефакторинг: улучшение существующего кода*. СПб.: Символ-Плюс, 2003

28

Один из основных принципов объектно-ориентированного программирования, утверждающий, что разрабатываемый код должен быть достаточно гибким, чтобы его можно было повторно использовать без дополнительных модификаций (то есть он должен быть открыт для использования, но закрыт для модификаций). – *Примеч. пер.*

29

Binder, Bob. *Testing Object-Oiiented Systems: Models, Patterns, and Tools.* Boston: Addison-Wesley, 1999. Это действительно исчерпывающее руководство по тестированию.

30

Mars Lander – американский космический аппарат, был запущен в сторону Марса 3 января 1999 г. 3 декабря 1999 г. аппарат должен был осуществить посадку на Марс, однако в этот день связь с ним была потеряна, предположительно из-за ошибки в программном обеспечении. Стоимость миссии составила приблизительно 120 млн долларов, не считая стоимости ракеты-носителя и некоторого дополнительного оборудования. – *Примеч. пер.*

31

Боб Мартин – известный деятель движения Agile Development (гибкая разработка), которого часто с уважением называют дядей. – *Примеч. пер.*

32

Weinberg, Gerald. *Systems Thinking. Quality Software Management.* New York: Dorset House, 1992.

33

Если исходить из примеров диаграмм взаимовлияния, приводимых автором книги, элемент этого типа может также называться «характеристика». – *Примеч. пер.*