КЕНТ БЕК

ЭКСТРЕМАЛЬНОЕ ПРОГРАММИРОВАНИЕ

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

РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ

Addison-Wesley

ПИТЕР

Библиотека программиста (Питер)

Кент Бек

Экстремальное программирование. Разработка через тестирование

«Питер»
2003

УДК 681.3.06
ББК 32.973.2-018

Бек К.

Экстремальное программирование. Разработка через тестирование / К. Бек — «Питер», 2003 — (Библиотека программиста (Питер))

ISBN 978-5-496-02570-6

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. В формате a4.pdf сохранен издательский макет.

УДК 681.3.06
ББК 32.973.2-018

ISBN 978-5-496-02570-6

Содержание

Предисловие 7
Благодарности 11
Введение 13
Часть I 16
1. Мультивалютные деньги 17
2. Вырождающиеся объекты 24
Конец ознакомительного фрагмента. 26

Кент Бек
Экстремальное программирование: разработка через тестирование

Посвящается Синди: крыльям моей души

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

[1] Бек К. *Экстремальное программирование*. СПб.: Питер, 2002. ISBN 5-94723-032-1.

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, отме-

чали, 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 началась 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.

[2] Подробнее о подсистеме отчетов рассказано на c2.com/doc/oopsla91.html.

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 $3x^2$ и $4y^3$ – 15 USD и 200 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.

[3] USD —доллары США, CHF – швейцарские франки. – *Примеч. пер.*

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 + 10 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 + 10 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.

---

[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.*

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…

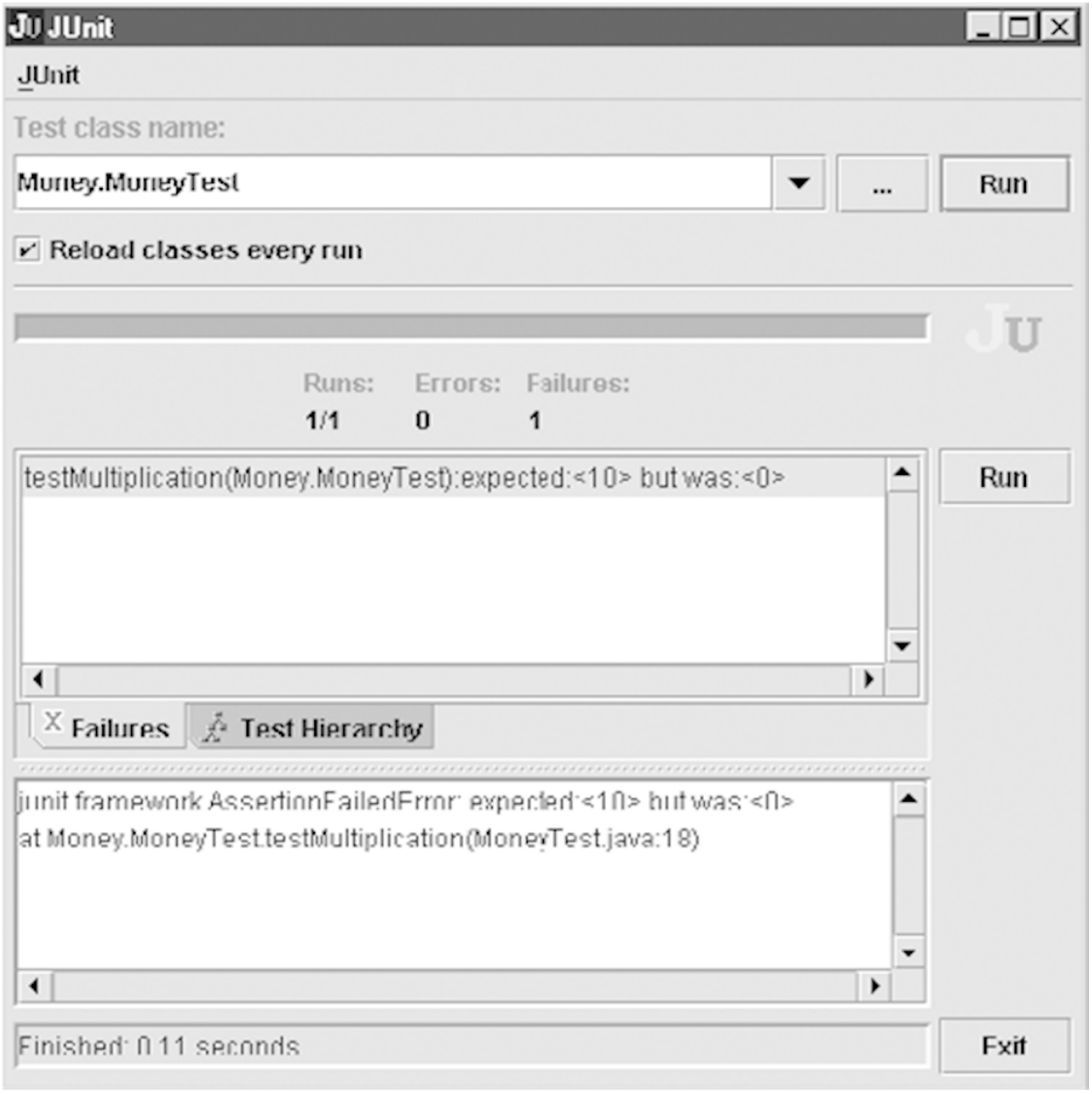

Рис. 1.1. Прогресс! Тест терпит неудачу

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.

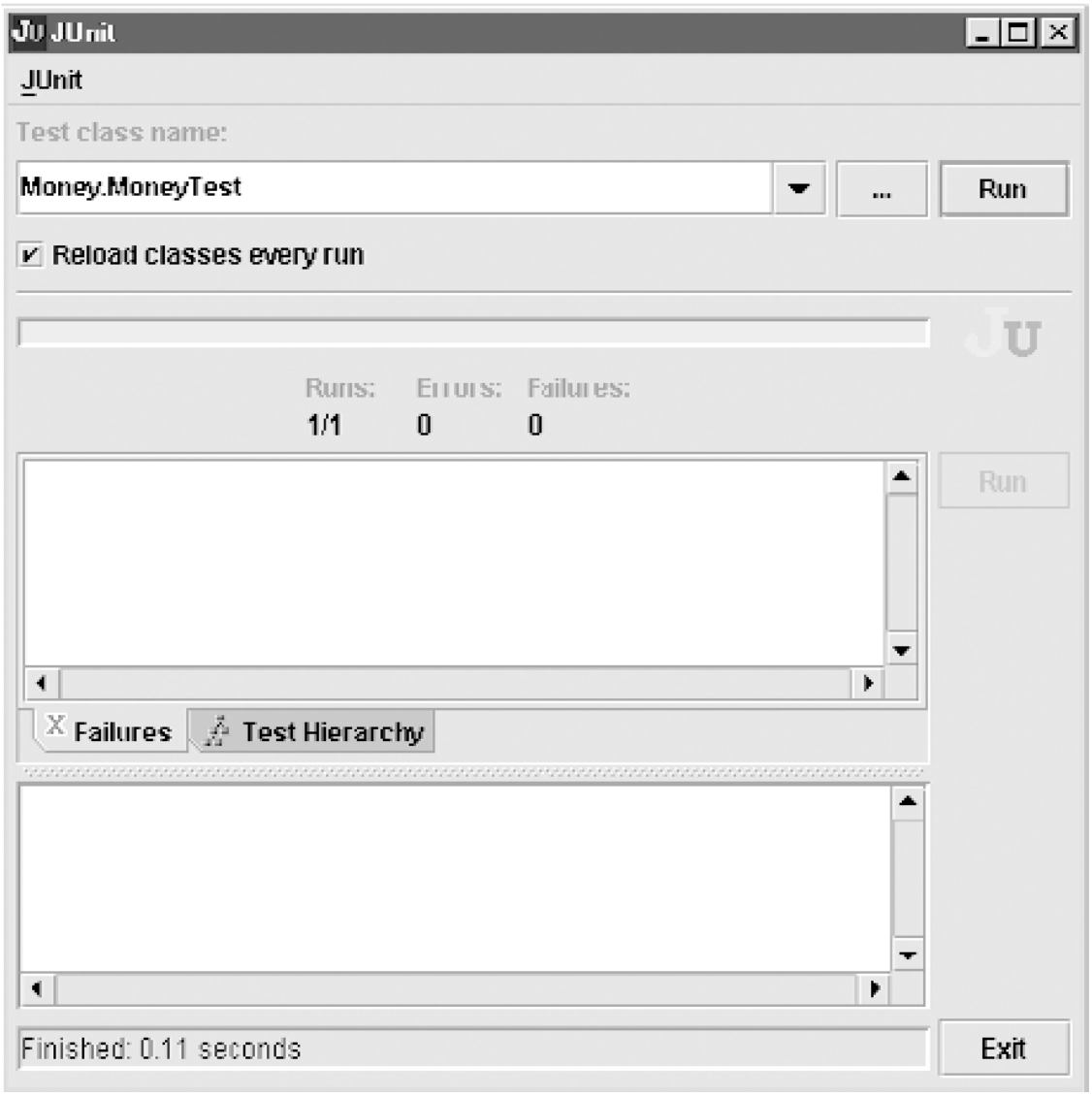

Рис. 1.2. Тест успешно выполняется

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 + 10 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 + 10 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 + 10 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;

Конец ознакомительного фрагмента.