The Addison-Wesley Signature Series

A KENT BECK SIGNATURE BOOK

TEST-DRIVEN DEVELOPMENT

BY EXAMPLE

KENT BECK

Test-Driven Development

The Addison-Wesley Signature Series

The Addison-Wesley Signature Series provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: great books come from great authors. Books in the series are personally chosen by expert advisors, world-class authors in their own right. These experts are proud to put their signatures on the covers, and their signatures ensure that these thought leaders have worked closely with authors to define topic coverage, book scope, critical content, and overall uniqueness. The expert signatures also symbolize a promise to our readers: you are reading a future classic.

The Addison-Wesley Signature Series

Signers: Kent Beck & Martin Fowler

Martin Fowler has been a pioneer of object technology in enterprise applications. His central concern is how to design software well. He focuses on getting to the heart of how to build enterprise software that will last well into the future. He is interested in looking behind the specifics of technologies to the patterns, practices, and principles that last for many years; these books should be usable a decade from now. Martin's criterion is that these are books he wished he could write.

Kent Beck has pioneered people-oriented technologies like JUnit, Extreme Programming, and patterns for software development. Kent is interested in helping teams do well by doing good – finding a style of software development that simultaneously satisfies economic, aesthetic, emotional, and practical constraints. His books focus on touching the lives of the creators and users of software.

Titles in the Series

Patterns of Enterprise Application Architecture
Martin Fowler, ISBN: 0321127420

Beyond Software Architecture: Creating and Sustaining Winning Solutions
Luke Hohmann, ISBN: 0201775948

Test-Driven Development: By Example
Kent Beck, ISBN: 0321146530

Test-Driven Development

By Example

Kent Beck

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Beck, Kent.
Test-driven development : by example / Kent Beck.
p. cm.
Includes index.
ISBN 0-321-14653-0 (alk. paper)
1. Computer software—Testing. 2. Computer software—Development. 3. Computer programming. I. Title.

QA76.76.T48 B43 2003
005.1'4—dc21

2002028037

ISBN 0-321-14653-0
Text printed on recycled paper
2 3 4 5 6 7 8 9 10 11—CRS—0706050403
Second printing, April 2003

To Cindee: wings of your own

Contents

Preface ix
Acknowledgments xv
Introduction xvii

PART I: The Money Example 1
Chapter 1: Multi-Currency Money 3
Chapter 2: Degenerate Objects 11
Chapter 3: Equality for All 15
Chapter 4: Privacy 19
Chapter 5: Franc-ly Speaking 23
Chapter 6: Equality for All, Redux 27
Chapter 7: Apples and Oranges 33
Chapter 8: Makin' Objects 35
Chapter 9: Times We're Livin' In 39
Chapter 10: Interesting Times 45
Chapter 11: The Root of All Evil 51
Chapter 12: Addition, Finally 55
Chapter 13: Make It 61
Chapter 14: Change 67

Chapter 15: Mixed Currencies 73
Chapter 16: Abstraction, Finally 77
Chapter 17: Money Retrospective 81

PART II: The xUnit Example 89
Chapter 18: First Steps to xUnit 91
Chapter 19: Set the Table 97
Chapter 20: Cleaning Up After 101
Chapter 21: Counting 105
Chapter 22: Dealing with Failure 109
Chapter 23: How Suite It Is 113
Chapter 24: xUnit Retrospective 119

PART III: Patterns for Test-Driven Development 121
Chapter 25: Test-Driven Development Patterns 123
Chapter 26: Red Bar Patterns 133
Chapter 27: Testing Patterns 143
Chapter 28: Green Bar Patterns 151
Chapter 29: xUnit Patterns 157
Chapter 30: Design Patterns 165
Chapter 31: Refactoring 181
Chapter 32: Mastering TDD 193

Appendix I: Influence Diagrams 207
Appendix II: Fibonacci 211

Afterword 215
Index 217

Preface

Clean code that works, in Ron Jeffries' pithy phrase, is the goal of Test-Driven Development (TDD). Clean code that works is a worthwhile goal for a whole bunch of reasons.

- It is a predictable way to develop. You know when you are finished, without having to worry about a long bug trail.
- It gives you a chance to learn all of the lessons that the code has to teach you. If you only slap together the first thing you think of, then you never have time to think of a second, better thing.
- It improves the lives of the users of your software.
- It lets your teammates count on you, and you on them.
- It feels good to write it.

But how do we get to clean code that works? Many forces drive us away from clean code, and even from code that works. Without taking too much counsel of our fears, here's what we do: we drive development with automated tests, a style of development called Test-Driven Development (TDD). In Test-Driven Development, we

- Write new code only if an automated test has failed
- Eliminate duplication

These are two simple rules, but they generate complex individual and group behavior with technical implications such as the following.

- We must design organically, with running code providing feedback between decisions.
- We must write our own tests, because we can't wait 20 times per day for someone else to write a test.

- Our development environment must provide rapid response to small changes.
- Our designs must consist of many highly cohesive, loosely coupled components, just to make testing easy.

The two rules imply an order to the tasks of programming.

1. Red—Write a little test that doesn't work, and perhaps doesn't even compile at first.
2. Green—Make the test work quickly, committing whatever sins necessary in the process.
3. Refactor—Eliminate all of the duplication created in merely getting the test to work.

Red/green/refactor—the TDD mantra.

Assuming for the moment that such a programming style is possible, it further might be possible to dramatically reduce the defect density of code and make the subject of work crystal clear to all involved. If so, then writing only that code which is demanded by failing tests also has social implications.

- If the defect density can be reduced enough, then quality assurance (QA) can shift from reactive work to proactive work.
- If the number of nasty surprises can be reduced enough, then project managers can estimate accurately enough to involve real customers in daily development.
- If the topics of technical conversations can be made clear enough, then software engineers can work in minute-by-minute collaboration instead of daily or weekly collaboration.
- Again, if the defect density can be reduced enough, then we can have shippable software with new functionality every day, leading to new business relationships with customers.

So the concept is simple, but what's my motivation? Why would a software engineer take on the additional work of writing automated tests? Why would a software engineer work in tiny little steps when his or her mind is capable of great soaring swoops of design? Courage.

Courage

Test-driven development is a way of managing fear during programming. I don't mean fear in a bad way—*pow widdle prwogwammew needs a pacifiew*—but fear in the legitimate, this-is-a-hard-problem-and-I-can't-see-the-end-from-the-beginning sense. If pain is nature's way of saying "Stop!" then fear is nature's way of saying "Be careful." Being careful is good, but fear has a host of other effects.

- Fear makes you tentative.
- Fear makes you want to communicate less.
- Fear makes you shy away from feedback.
- Fear makes you grumpy.

None of these effects are helpful when programming, especially when programming something hard. So the question becomes how we face a difficult situation and,

- Instead of being tentative, begin learning concretely as quickly as possible.
- Instead of clamming up, communicate more clearly.
- Instead of avoiding feedback, search out helpful, concrete feedback.
- (You'll have to work on grumpiness on your own.)

Imagine programming as turning a crank to pull a bucket of water from a well. When the bucket is small, a free-spinning crank is fine. When the bucket is big and full of water, you're going to get tired before the bucket is all the way up. You need a ratchet mechanism to enable you to rest between bouts of cranking. The heavier the bucket, the closer the teeth need to be on the ratchet.

The tests in test-driven development are the teeth of the ratchet. Once we get one test working, we know it is working, now and forever. We are one step closer to having everything working than we were when the test was broken. Now we get the next one working, and the next, and the next. By analogy, the tougher the programming problem, the less ground that each test should cover.

Readers of my book *Extreme Programming Explained* will notice a difference in tone between Extreme Programming (XP) and TDD. TDD isn't an absolute the

way that XP is. XP says, "Here are things you must be able to do to be prepared to evolve further." TDD is a little fuzzier. TDD is an awareness of the gap between decision and feedback during programming, and techniques to control that gap. "What if I do a paper design for a week, then test-drive the code? Is that TDD?" Sure, it's TDD. You were aware of the gap between decision and feedback, and you controlled the gap deliberately.

That said, most people who learn TDD find that their programming practice changed for good. *Test Infected* is the phrase Erich Gamma coined to describe this shift. You might find yourself writing more tests earlier, and working in smaller steps than you ever dreamed would be sensible. On the other hand, some software engineers learn TDD and then revert to their earlier practices, reserving TDD for special occasions when ordinary programming isn't making progress.

There certainly are programming tasks that can't be driven solely by tests (or at least, not yet). Security software and concurrency, for example, are two topics where TDD is insufficient to mechanically demonstrate that the goals of the software have been met. Although it's true that security relies on essentially defect-free code, it also relies on human judgment about the methods used to secure the software. Subtle concurrency problems can't be reliably duplicated by running the code.

Once you are finished reading this book, you should be ready to

- Start simply
- Write automated tests
- Refactor to add design decisions one at a time

This book is organized in three parts.

- Part I, The Money Example—An example of typical model code written using TDD. The example is one I got from Ward Cunningham years ago and have used many times since: multi-currency arithmetic. This example will enable you to learn to write tests before code and grow a design organically.
- Part II, The xUnit Example—An example of testing more complicated logic, including reflection and exceptions, by developing a framework for automated testing. This example also will introduce you to the xUnit architecture that is at the heart of many programmer-oriented testing tools. In the second example, you will learn to work in even smaller steps

than in the first example, including the kind of self-referential hoo-ha beloved of computer scientists.

- Part III, Patterns for Test-Driven Development—Included are patterns for deciding what tests to write, how to write tests using xUnit, and a greatest-hits selection of the design patterns and refactorings used in the examples.

I wrote the examples imagining a pair programming session. If you like looking at the map before wandering around, then you may want to go straight to the patterns in Part III and use the examples as illustrations. If you prefer just wandering around and then looking at the map to see where you've been, then try reading through the examples, referring to the patterns when you want more detail about a technique, and using the patterns as a reference. Several reviewers of this book commented they got the most out of the examples when they started up a programming environment, entered the code, and ran the tests as they read.

A note about the examples. Both of the examples, multi-currency calculation and a testing framework, appear simple. There are (and I have seen) complicated, ugly, messy ways of solving the same problems. I could have chosen one of those complicated, ugly, messy solutions, to give the book an air of "reality." However, my goal, and I hope your goal, is to write clean code that works. Before teeing off on the examples as being too simple, spend 15 seconds imagining a programming world in which all code was this clear and direct, where there were no complicated solutions, only apparently complicated problems begging for careful thought. TDD can help you to lead yourself to exactly that careful thought.

Acknowledgments

Thanks to all of my many brutal and opinionated reviewers. I take full responsibility for the contents, but this book would have been much less readable and much less useful without their help. In the order in which I typed them, they were: Steve Freeman, Frank Westphal, Ron Jeffries, Dierk König, Edward Hieatt, Tammo Freese, Jim Newkirk, Johannes Link, Manfred Lange, Steve Hayes, Alan Francis, Jonathan Rasmusson, Shane Clauson, Simon Crase, Kay Pentecost, Murray Bishop, Ryan King, Bill Wake, Edmund Schweppe, Kevin Lawrence, John Carter, Phlip, Peter Hansen, Ben Schroeder, Alex Chaffee, Peter van Rooijen, Rick Kawala, Mark van Hamersveld, Doug Swartz, Laurent Bossavit, Ilja Preuß, Daniel Le Berre, Frank Carver, Justin Sampson, Mike Clark, Christian Pekeler, Karl Scotland, Carl Manaster, J. B. Rainsberger, Peter Lindberg, Darach Ennis, Kyle Cordes, Justin Sampson, Patrick Logan, Darren Hobbs, Aaron Sansone, Syver Enstad, Shinobu Kawai, Erik Meade, Patrick Logan, Dan Rawsthorne, Bill Rutiser, Eric Herman, Paul Chisholm, Asim Jalis, Ivan Moore, Levi Purvis, Rick Mugridge, Anthony Adachi, Nigel Thorne, John Bley, Kari Hoijarvi, Manuel Amago, Kaoru Hosokawa, Pat Eyler, Ross Shaw, Sam Gentle, Jean Rajotte, Phillipe Antras, Jaime Nino, and Guy Tremblay.

To all of the programmers I've test-driven code with, I certainly appreciate your patience going along with what was a pretty crazy sounding idea, especially in the early years. I've learned far more from you all than I could ever think of myself. Not wishing to offend everyone else, but Massimo Arnoldi, Ralph Beattie, Ron Jeffries, Martin Fowler, and last but certainly not least Erich Gamma stand out in my memory as test drivers from whom I've learned much.

I would like to thank Martin Fowler for timely FrameMaker help. He must be the highest-paid typesetting consultant on the planet, but fortunately he has let me (so far) run a tab.

My life as a real programmer started with patient mentoring from and continuing collaboration with Ward Cunningham. Sometimes I see Test-Driven

Development (TDD) as an attempt to give any software engineer, working in any environment, the sense of comfort and intimacy we had with our Smalltalk environment and our Smalltalk programs. There is no way to sort out the source of ideas once two people have shared a brain. If you assume that all of the good ideas here are Ward's, then you won't be far wrong.

It is a bit cliché to recognize the sacrifices a family makes once one of its members catches the peculiar mental affliction that results in a book. That's because family sacrifices are as necessary to book writing as paper is. To my children, who waited breakfast until I could finish a chapter, and most of all to my wife, who spent two months saying everything three times, my most-profound and least-adequate thanks.

Thanks to Mike Henderson for gentle encouragement and to Marcy Barnes for riding to the rescue.

Finally, to the unknown author of the book which I read as a weird 12-year-old that suggested you type in the expected output tape from a real input tape, then code until the actual results matched the expected result, thank you, thank you, thank you.

Introduction

Early one Friday, the boss came to Ward Cunningham to introduce him to Peter, a prospective customer for WyCash, the bond portfolio management system the company was selling. Peter said, "I'm very impressed with the functionality I see. However, I notice you only handle U.S. dollar denominated bonds. I'm starting a new bond fund, and my strategy requires that I handle bonds in different currencies." The boss turned to Ward, "Well, can we do it?"

Here is the nightmarish scenario for any software designer. You were cruising along happily and successfully with a set of assumptions. Suddenly, everything changed. And the nightmare wasn't just for Ward. The boss, an experienced hand at directing software development, wasn't sure what the answer was going to be.

A small team had developed WyCash over the course of a couple of years. The system was able to handle most of the varieties of fixed income securities commonly found on the U.S. market, and a few exotic new instruments, like Guaranteed Investment Contracts, that the competition couldn't handle.

WyCash had been developed all along using objects and an object database. The fundamental abstraction of computation, `Dollar`, had been outsourced at the beginning to a clever group of software engineers. The resulting object combined formatting and calculation responsibilities.

For the past six months, Ward and the rest of the team had been slowly divesting `Dollar` of its responsibilities. The Smalltalk numerical classes turned out to be just fine at calculation. All of the tricky code for rounding to three decimal digits got in the way of producing precise answers. As the answers became more precise, the complicated mechanisms in the testing framework for comparison within a certain tolerance were replaced by precise matching of expected and actual results.

Responsibility for formatting actually belonged in the user interface classes. As the tests were written at the level of the user interface classes, in particular

the report framework,[1] these tests didn't have to change to accommodate this refinement. After six months of careful paring, the resulting `Dollar` didn't have much responsibility left.

One of the most complicated algorithms in the system, weighted average, likewise had been undergoing a slow transformation. At one time, there had been many different variations of weighted average code scattered throughout the system. As the report framework coalesced from the primordial object soup, it was obvious that there could be one home for the algorithm, in `AveragedColumn`.

It was to `AveragedColumn` that Ward now turned. If weighted averages could be made multi-currency, then the rest of the system should be possible. At the heart of the algorithm was keeping a count of the money in the column. In fact, the algorithm had been abstracted enough to calculate the weighted average of any object that could act arithmetically. One could have weighted averages of dates, for example.

The weekend passed with the usual weekend activities. Monday morning the boss was back. "Can we do it?"

"Give me another day, and I'll tell you for sure."

`Dollar` acted like a counter in weighted average; therefore, in order to calculate in multiple currencies, they needed an object with a counter per currency, kind of like a polynomial. Instead of $3x^2$ and $4y^3$, however, the terms would be 15 USD and 200 CHF.

A quick experiment showed that it was possible to compute with a generic `Currency` object instead of a `Dollar`, and return a `PolyCurrency` when two unlike currencies were added together. The trick now was to make space for the new functionality without breaking anything that already worked. What would happen if Ward just ran the tests?

After the addition of a few unimplemented operations to `Currency`, the bulk of the tests passed. By the end of the day, all of the tests were passing. Ward checked the code into the build and went to the boss. "We can do it," he said confidently.

Let's think a bit about this story. In two days, the potential market was multiplied several fold, multiplying the value of WyCash several fold. The ability to create so much business value so quickly was no accident, however. Several factors came into play.

- Method—Ward and the WyCash team needed to have constant experience growing the design of the system, little by little, so the mechanics of the transformation were well practiced.

1. For more about the report framework, refer to c2.com/doc/oopsla91.html.

- Motive—Ward and his team needed to understand clearly the business importance of making WyCash multi-currency, and to have the courage to start such a seemingly impossible task.
- Opportunity—The combination of comprehensive, confidence-generating tests; a well-factored program; and a programming language that made it possible to isolate design decisions meant that there were few sources of error, and those errors were easy to identify.

You can't control whether you ever get the motive to multiply the value of your project by spinning technical magic. Method and opportunity, on the other hand, are entirely under your control. Ward and his team created method and opportunity through a combination of superior talent, experience, and discipline. Does this mean that if you are not one of the ten best software engineers on the planet and don't have a wad of cash in the bank so you can tell your boss to take a hike, then you're going to take the time to do this right, that such moments are forever beyond your reach?

No. You absolutely can place your projects in a position for you to work magic, even if you are a software engineer with ordinary skills and you sometimes buckle under and take shortcuts when the pressure builds. Test-driven development is a set of techniques that any software engineer can follow, which encourages simple designs and test suites that inspire confidence. If you are a genius, you don't need these rules. If you are a dolt, the rules won't help. For the vast majority of us in between, following these two simple rules can lead us to work much more closely to our potential.

- Write a failing automated test before you write any code.
- Remove duplication.

How exactly to do this, the subtle gradations in applying these rules, and the lengths to which you can push these two simple rules are the topic of this book. We'll start with the object that Ward created in his moment of inspiration—multi-currency money.

Chapter 3

Equality for All

If I have an integer and I add 1 to it, I don't expect the original integer to change, I expect to use the new value. Objects usually don't behave that way. If I have a contract and I add one to its coverage, then the contract's coverage should change (yes, yes, subject to all sorts of interesting business rules which do *not* concern us here).

We can use objects as values, as we are using our `Dollar` now. The pattern for this is Value Object. One of the constraints on Value Objects is that the values of the instance variables of the object never change once they have been set in the constructor.

There is one huge advantage to using Value Objects: you don't have to worry about aliasing problems. Say I have one check and I set its amount to $5, and then I set another check's amount to the same $5. Some of the nastiest bugs in my career have come when changing the first check's value inadvertently changed the second check's value. This is aliasing.

When you have Value Objects, you needn't worry about aliasing. If I have $5, then I am guaranteed that it will always and forever be $5. If someone wants $7, then they will have to make an entirely new object.

```
$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side effects?~~
Money rounding?
equals()
```

One implication of Value Objects is that all operations must return a new object, as we saw in Chapter 2. Another implication is that Value Objects should implement `equals()`, because one $5 is pretty much as good as another.

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side effects?~~
Money rounding?
equals()
hashCode()

If you use `Dollar`s as the key to a hash table, then you have to implement `hashCode()` if you implement `equals()`. We'll put that on the to-do list, too, and get to it when it's a problem.

You aren't thinking about the implementation of `equals()`, are you? Good. Me neither. After snapping the back of my hand with a ruler, I'm thinking about how to test equality. First, $5 should equal $5:

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

The bar turns obligingly red. The fake implementation is just to return `true`:

Dollar

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

You and I both know that `true` is really "5 == 5", which is really "amount == 5", which is really "amount == dollar.amount". If I went through these steps, though, I wouldn't be able to demonstrate the third and most conservative implementation strategy: Triangulation.

If two receiving stations at a known distance from each other can both measure the direction of a radio signal, then there is enough information to calculate the range and bearing of the signal (if you remember more trigonometry than I do, anyway). This calculation is called Triangulation.

By analogy, when we triangulate, we only generalize code when we have two examples or more. We briefly ignore the duplication between test and model code. When the second example demands a more general solution, then and only then do we generalize.

So, to triangulate we need a second example. How about $5 != $6?

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

Now we need to generalize equality:

Dollar

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

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()

We could have used Triangulation to drive the generalization of `times()` also. If we had $5 x 2 = $10 and $5 x 3 = $15, then we would no longer have been able to return a constant.

Triangulation feels funny to me. I use it only when I am completely unsure of how to refactor. If I can see how to eliminate duplication between code and tests and create the general solution, then I just do it. Why would I need to write another test to give me permission to write what I probably could have written in the first place?

However, when the design thoughts just aren't coming, Triangulation provides a chance to think about the problem from a slightly different direction. What axes of variability are you trying to support in your design? Make some of them vary, and the answer may become clearer.

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object

So, equality is done for the moment. But what about comparing with null and comparing with other objects? These are commonly used operations but not necessary at the moment, so we'll add them to the to-do list.

Now that we have equality, we can directly compare `Dollar`s to `Dollar`s. That will let us make "amount" private, as all good instance variables should be. To review the above, we

- Noticed that our design pattern (Value Object) implied an operation
- Tested for that operation
- Implemented it simply
- Didn't refactor immediately, but instead tested further
- Refactored to capture the two cases at once

Index

A
Addition process, writing/testing code
  adding parameters, 67–70
  declarations, causing other changes to ripple, 73–80
  eliminating data duplication, 67–70
  implementing/moving code, 61–65
  metaphors, 56–59
AllTests suite testing, 163–164
Application test-driven development (ATDD), *versus* TDD, 199
Arrange/Act/Assert (3A) pattern, 97
Assert-First testing, 128–129
Assertions design pattern, 157–158
ATDD (application test-driven development), *versus* TDD, 199

B
Bootstrapping problems, test method, 91–95
Broken Tests, 148–149

C
Child Tests, 143
Classes
  code metrics, 84
  concrete, 46
  defining, 5
  eliminating explicit class checking with polymorphism, 61–65
  equality testing, 15–18, 27–31
  subclasses, 27–31, 36–37
  subclasses, eliminating, 39–43
  subclasses, replacing references with references to superclasses, 51–53
  superclasses, 27–31
Clean code check-in, 149–150
Code duplication
  and code dependency, 7, 8
  removing, 24, 25
Code metrics, 84
Code refactoring, 40–43
Collecting Parameter design pattern, 114, 166, 178–179
Command design pattern, 165, 166, 167
Composite design pattern, 113–117, 166, 176–178
  and Impostors, 176
Concrete classes, 46
Crash Test Dummy, 147–148
  extracting interfaces, 187
Cyclomatic complexity, code metrics, 84

D
Dependency on code, and duplication of code, 7, 8
Design patterns
  Broken Tests, 148–149
  Child Tests, 143
  clean code check-in, 149–150
  Collecting Parameters, 114, 166, 178–179
  Command, 165, 166, 167
  Composite, 113–117, 166, 176–178
  Crash Test Dummy, 147–148, 187
  Factory Methods, 166, 174–175
  Impostors, 166, 175–176
  Log Strings, 146–147
  Mock Objects, 187
  Null Objects, 166, 169–170
  overview, 165–166
  Pluggable Objects, 166, 171–173
  Pluggable Selectors, 166, 173–174
  Self Shunt Tests, 145–146
  Singletons, 179
  Template Method, 166, 170–171
  Value Objects, 165, 166, 167–169
Design patterns, green bar patterns
  Fake It, 151–153
  Obvious Implementation, 154–155
  One to Many, 155–156, 183–184
  Triangulation, 153–154
Design patterns, red bar patterns
  Explanation Tests, 135–136
  Learning Tests, 136–137
  One Step Tests, 133–134
  Regression Tests, 137–138
  Starter Tests, 134–135
Design patterns, xUnit framework
  AllTests, 163–164
  Assertions, 157–158
  Exception Tests, 163

Design patterns, xUnit framework, *continued*
Fixtures, 158–160
Fixtures, External, 160–161
Test Method, 161–163
Duplication of code
and dependency on code, 7, 8
removing, 24, 25

E
Equality testing, 15–18, 27–31, 33–34
Error messages, 47
Evident Data, 130–131
Exceptions
exception handling, 105–107
Exception Tests, 163
Explanation Tests, 135–136
Explicit class checking, eliminating with polymorphism, 61–65
Extract Interface and Self Shunt Tests, 146
Extract Method, 95
versus Method Objects, 189
refactoring, 183, 184–185
Extract Objects, 183
Extreme Programming (XP), *versus* TDD, 204–205

F
Factory Methods, 36–37
design patterns, 166, 174–175
Failures in testing
Mean Time Between Failures, 196–197
multiplication process, 4–6
xUnit framework, 109–111
Fake It implementations of TDD, 13, 106, 138
Crash Test Dummy, 147
green bar design patterns, 151–153
Feedback
amount needed in TDD, 196–197
influence diagrams, 208–209
Fibonacci, 211–213
Flags of called methods, 91–100
versus logs, 101
Functions, code metrics, 84

G–H
Green bar design patterns
Fake It, 151–153
Obvious Implementation, 154–155
One to Many, 155–156
One to Many data migration, 183–184
Triangulation, 153–154

I
Implementations of TDD
Fake It, 13, 103, 138
Fake It, and Crash Test Dummy, 147
Fake It , green bar design patterns, 151–153
Obvious, 13, 103, 138
Obvious Implementation, green bar patterns, 154–155
Triangulation, 13, 16–17, 138
Triangulation , green bar design patterns, 153–154
Impostor design pattern, 166, 175–176
Influence diagrams, 123–124, 127
elements, 207–208
feedback, 208–209
systems of development practices, 209–210
Inline Method, 185–186
Instance variables, private, 20–21
Isolated tests, design patterns, 125–126
Isolating changes
One to Many design patterns, 155–156
refactoring, 182–183

J–K
JProbe, 86
JUnit, using, 83
JXUnit, 158

L
Learning Tests, 136–137
Log Strings, 146–147
Logs of called methods, 101–107
versus flags, 101

M
Mean Time Between Failures (MTBF), 196–197
Metaphors, 56–59
advantages, 82–83
Method Objects, *versus* Extract Method, 189
Methods
Extract Method, refactoring, 183, 184–185
Extract Method, *versus* Method Objects, 189
Factory Methods, 36–37, 166, 174–175
Inline Method, 185–186
Move Method, 187–189
parameters, adding to methods, 190
parameters, moving from methods, 190–191
reconciling, 45–49
setUp(), 97–100
tearDown(), 101–107
Template Method, 146, 166, 170–171
testMethod(), 91–95
Mock Objects, 144–145
and Crash Test Dummy, 148
and extracting objects, 187
Move Method, 187–189
MTBF (Mean Time Between Failures), 196–197
Multi-currency money object, creating, 3–10
Multiplication process, writing/testing code, 7, 11
equality testing, 15–18, 27–31, 33–34
factory methods, 36–37
implementations, Fake It, 13
implementations, Obvious, 13
implementations, redundant, 31
implementations, redundant, eliminating, 51–53
implementations, Triangulation, 13, 16–17
improving tests with functionality, 19–21
refactoring code, 40–43
testing cycle, 1, 7, 11, 84–85
testing cycle, decisions on initial steps, 4–5
testing cycle, failures, 4–6
testing cycle, scope, 5, 23–25
writing tests by copying/editing code, 23–25

N
Negative feedback, influence diagrams, 208–209
Null Objects
  design patterns, 166, 169–170
  and Impostors, 176

O
Objects
  creating, 35–37
  creating, by force of tests, 61–65
  creating, constraints causing conflicts, 97–100
  creating, multi-currency money, 3–10
  Mock Objects, 144–145
  Mock Objects, and Crash Test Dummy, 148
  Mock Objects, and extracting interfaces, 187
  Null Object design patterns, 166, 169–170
  Null Object design patterns, and Impostors, 176
  Open/Closed Principle, 195–196
  Value Objects, 15–18
Obvious Implementation of TDD, 13, 103, 138
  green bar patterns, 154–155
One Step Tests, 133–134
  and Starter Tests, 135
One to Many design patterns, 155–156
  data migration, 183–184
Open/Closed Principle (objects), 195–196

P–Q
Pair programming, physical setup, 140–141
Performance testing, *versus* TDD, 86–87
Pluggable Objects, design patterns, 166, 171–173
Pluggable Selectors, 95
  design patterns, 166, 173–174
Polymorphism, eliminating explicit class checking, 61–65
Positive feedback, influence diagrams, 208–209
Private instance variables, 20–21

R
Realistic Data, 130
Red bar design patterns
  Explanation Tests, 135–136
  Learning Tests, 136–137
  One Step Tests, 133–134
  Regression Tests, 137–138
  Starter Tests, 134–135
Refactoring code, 40–43
  Extract Method, 183, 184–185
  Extract Objects, 183
  extracting interfaces, 187
  Inline Method, 185–186
  isolating changes, 182–183
  Method Objects, 183, 189–190
  migrating data, 183–184
  Move Method, 187–189
  parameters, adding to methods, 190
  parameters, moving from methods, 190–191
  reconciling differences, 181–182
Regression Tests, 137–138

S
Scripts with xUnit testing framework, 120
Self Shunt Tests, 145–146
SetUp() method, 97–100
Shower Methodology, 138
Singleton design patterns, 179
SmallLint for SmallTalk, 82
Starter Tests, 134–135
  and One Step Tests, 135
Stress testing
  influence diagrams, 123–124
  versus TDD, 86–87
Subclasses, 27–31, 36–37
  eliminating, 39–43
  replacing references with references to superclasses, 51–53
Suites for multiple tests
  AllTests, 163–164
  TestSuite, 113–117
Superclasses, 27–31
  replacing references to subclasses, 51–53

T
TDD
  analysis of term, 203–204
  versus application test-driven development, 199
  attributes of effective tests, 194–195
  beginning TDD practices in middle of projects, 199–200
  decisions on content for testing, 194
  deleting tests, 198
  extending reach of TDD, 205
  versus Extreme Programming, 204–205
  feedback amount needed, 196–197
  Fibonacci, 211–213
  influence of programming languages and environment, 198
  JProbe, 86
  Mean Time Between Failures, 196–197
  Open/Closed Principle (objects), 195–196
  versus other programming styles, 77–80
  versus other types of testing, 86–87
  potential users, 200–201
  reasons for effectiveness, 202–203
  reusable code, 195–196
  scaling to large systems, 198–199
  SmallLint for SmallTalk, 82
  test, definition, 123
  test quality, 86–87
  testing sequence, 201
TDD design patterns, 201–202
  Assert-First, 128–129
  basics, 123–125
  Broken Tests, 148–149
  Child Tests, 143
  clean code check-in, 149–150
  Collecting Parameter, 114
  Collecting Parameters, 166, 178–179
  Command, 165, 166, 167
  Composite, 113–117, 166, 176–178
  Crash Test Dummy, 147–148
  Crash Test Dummy, extracting interfaces, 187
  Evident Data, 130–131
  Factory Methods, 166, 174–175

TDD design patterns, *continued*
Impostors, 166, 175–176
isolated tests, 125–126
Log Strings, 146–147
Mock Objects, 144–145
Null Objects, 166, 169–170
overview, 165–166
Pluggable Objects, 166, 171–173
Pluggable Selectors, 166, 173–174
Realistic Data, 130
Self Shunt Tests, 145–146
Singletons, 179
Template Method, 166, 170–171
Test Data, 129–130
Test-First, 127–128
Test Lists, 126–127
Value Objects, 165, 166, 167–169
TDD design patterns, green bar patterns
Fake It, 151–153
Obvious Implementation, 154–155
One to Many, 155–156
One to Many, data migration, 183–184
Triangulation, 153–154
TDD design patterns, red bar patterns
Explanation Tests, 135–136
Learning Tests, 136–137
One Step Tests, 133–134
Regression Tests, 137–138
Starter Tests, 134–135
TDD design patterns, xUnit framework
AllTests, 163–164
Assertions, 157–158
Exception Tests, 163
Fixtures, 158–160
Fixtures, External, 160–161
Test Method, 161–163
TDD guidelines
physical setup, 140–141
starting over, 139–140
taking breaks, 138–139
TDD refactoring
Extract Method, 183, 184–185
Extract Objects, 183
extracting interfaces, 187
Inline Method, 185–186
isolating changes, 182–183
Method Objects, 183, 189–190
migrating data, 183–184
Move Method, 187–189
parameters, adding to methods, 190
parameters, moving from methods, 190–191
reconciling differences, 181–182
TDD testing cycle, 1, 7, 11, 84–85
failures, 4–6
scope, 5, 23–25
steps, initial steps, 4–5
steps, size of, 193–194
TearDown() method, 101–107
Template Method, 146
design patterns, 166, 170–171
Test coupling, 98
Test Data, 129–130
Test-Driven Development. *See* TDD
Test-First testing, 127–128
Test Lists, 126–127
Testing/writing code, addition process
adding parameters, 67–70
declarations, causing other changes to ripple, 73–80
eliminating data duplication, 67–70
implementing/moving code, 61–65
metaphors, 56–59
Testing/writing code, multiplication process
equality testing, 15–18, 27–31, 33–34
factory methods, 36–37
implementations, Fake It, 13
implementations, Obvious, 13
implementations, redundant, 31
implementations, redundant, eliminating, 51–53
implementations, Triangulation, 13, 16–17
improving tests with functionality, 19–21
refactoring code, 40–43
testing cycle, 1, 7, 11, 84–85
testing cycle, decisions on initial steps, 4
testing cycle, failures, 4–6
testing cycle, scope, 5, 23–25
versus work code, 8, 9
writing tests by copying/editing code, 23–25
TestMethod() method, 91–95
TestSuite, multiple tests, 113–117
3A (Arrange/Act/Assert) pattern, 97
To-do lists, 4
Triangulation implementation of TDD, 13, 16–17, 138
green bar design patterns, 153–154

U–W
Usability testing, *versus* TDD, 86–87

Value Objects, 15–18
design patterns, 165, 166, 167–169
Variables
private instance variables, 20–21
replacing constants, 45–49

Working code, *versus* test code, 8, 9

X–Z
XP (Extreme Programming), *versus* TDD, 204–205
XUnit testing framework
failures, 109–111
methods, setUp(), 97–100
methods, tearDown(), 101–107
methods, testMethod(), 91–95
reasons for implementing, 119
TestSuite, running multiple tests, 113–117
XUnit testing framework, TDD design patterns
AllTests, 163–164
Assertions, 157–158
Exception Tests, 163
Fixtures, 158–160
Fixtures, External, 160–161
Test Method, 161–163