原版风暴 · 软件工程系列

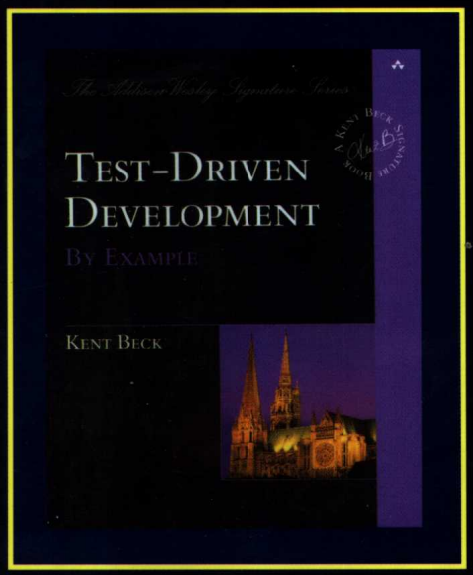
PEARSON
Addison Wesley

Test-Driven Development
By Example

测试驱动开发
（影印版）

本年度美国
Software Development Productivity 奖

[美] Kent Beck 著



- 采用项目实例讲述测试驱动开发原理和方法
- 提供测试驱动开发中最有特色的模式和重构实例
- 极限编程创始人 Kent Beck 又一力作

Pearson Education
中国电力出版
www.infopower.com

Test-Driven Development

测试驱动开发

（影印版）

原版风暴·软件工程系列

Test-Driven Development
By Example

测试驱动开发
（影印版）

[美] Kent Beck 著

中国电力出版社

Test-Driven Development: By Example (ISBN 0-321-14653-0)
Kent Beck

北京市版权局著作合同登记号：图字：01-2003-3825

图书在版编目（CIP）数据

测试驱动开发 /（美）贝克著. —影印本. —北京：中国电力出版社，2003
（原版风暴·软件工程系列）
ISBN 7-5083-1401-8
Ⅰ.测... Ⅱ.贝... Ⅲ.软件开发－英文 Ⅳ.TP311.52
中国版本图书馆 CIP 数据核字（2003）第 048302 号

责任编辑： 姚贵胜
丛 书 名： 原版风暴·软件工程系列
书　　名： 测试驱动开发（影印版）
编　　著：（美）Kent Beck
出 版 者： 中国电力出版社
地址：北京市三里河路6号　邮政编码：100044
电话：（010）88515918　传真：（010）88518169
印　　刷： 北京地矿印刷厂
开　　本： 787×1092 1/16　**印　　张：** 15.5
书　　号： ISBN 7-5083-1401-8
版　　次： 2003年8月北京第一版
印　　次： 2003年8月第一次印刷
定　　价： 32.00 元

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, and Jaime Nino.

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.

Contents

PART I

The Money Example

In Part I, we will develop typical model code driven completely by tests (except when we slip, purely for educational purposes). My goal is for you to see the rhythm of Test-Driven Development (TDD), which can be summed up as follows.

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a little change.
4. Run all tests and see them all succeed.
5. Refactor to remove duplication.

The surprises are likely to include

- How each test can cover a small increment of functionality
- How small and ugly the changes can be to make the new tests run
- How often the tests are run
- How many teensy-weensy steps make up the refactorings