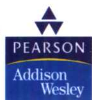
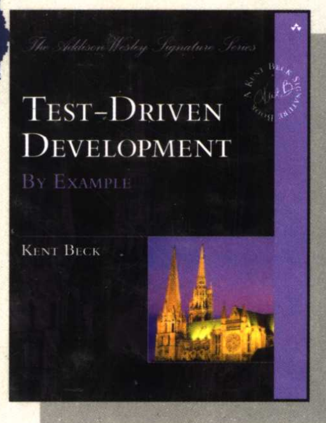


Test-Driven Development By Example

[美] Kent Beck 著
孙 方 注释

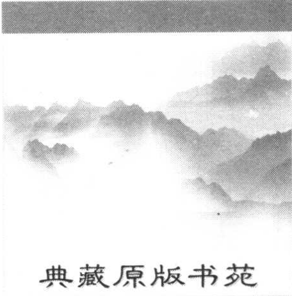
测试驱动开发

注释版



Software Development Productivity大奖作品

人民邮电出版社
POSTS & TELECOM PRESS



典藏原版书苑

测试驱动开发

（注释版）

[美] Kent Beck 著

孙方 注释

人民邮电出版社

北 京

图书在版编目（CIP）数据

测试驱动开发（注释版）/（美）贝克（Beck, K.）著；孙方注释．—北京：人民邮电出版社，2007.11
（典藏原版书苑）
ISBN 978-7-115-15620-4

Ⅰ．测… Ⅱ．①贝…②孙… Ⅲ．软件开发—英文
Ⅳ．TP311.52

中国版本图书馆 CIP 数据核字（2007）第 087157 号

版 权 声 明

Original edition, entitled Test-driven Development: By Example, 0321146530 by Kent Beck, published by Pearson Education, Inc, publishing as Addison-Wesley, Copyright © 2003 by Pearson Education, Inc. and Kent Beck.
All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.
China edition published by PEARSON EDUCATION ASIA LTD., and POSTS & TELECOMMUNICATIONS PRESS Copyright © 2007.
This edition is manufactured in the People's Republic of China, and is authorized for sale only in People's Republic of China excluding Hong Kong, Macau and Taiwan.
仅限于中华人民共和国境内（不包括中国香港、澳门特别行政区和中国台湾地区）销售。
本书封面贴有 Pearson Education**（培生教育出版集团）激光防伪标签。无标签者不得销售。**

典藏原版书苑

测试驱动开发（注释版）

◆ 著　［美］Kent Beck
注　释　孙　方
责任编辑　刘映欣
◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京铭成印刷有限公司印刷
新华书店总店北京发行所经销
◆ 开本：800×1000　1/16
印张：16
字数：354 千字　2007 年 11 月第 1 版
印数：1 – 3 500 册　2007 年 11 月北京第 1 次印刷
著作权合同登记号　图字：01-2006-3667 号

ISBN 978-7-115-15620-4/TP

定价：49.00 元

读者服务热线：(010)67132705　印装质量热线：(010)67129223

内容提要

近几年，极限编程（XP）在中国的软件企业界越来越受到重视，越来越多的开发团队开始应用 XP 的方法并从中受益。测试驱动开发（TDD）是极限编程的重要特点，它以不断的测试推动代码的开发，既简化了代码，又保证了软件质量。本书从头至尾追随两个 TDD 项目，详细阐述以简单有效的方式提高程序员工作质量的技术。书中的每个示例之后是对重要 TDD 模式和重构方法的引用。

本书的注释内容并不是对原文的简单摘译，而是加入了 TDD 实践人员在工作中的大量实践经验和心得体会，以期引导读者更深入准确地领会到原著的内涵。附注中包括注释者对书中注释关键点的汇总以及将 xUnit 实例用 Java 语言改写的版本。

出版说明

正如软件工程名家 Roger S. Pressman 所言，“软件工程从少数拥护者所实践的朦胧的思想，演化成一个正式的工程学科。今天，它已经被承认为一个值得认真地研究、细心地学习和热烈地争论的主题。”

在计算机专业书籍中，软件工程领域的许多经典可能既是最易阅读的，又是最需要花费时间去领悟的。它既不富含烦琐的形式化推导，亦无特定的运行环境，更不充斥确定的源代码，读者可以一目十行地顺畅浏览软件工程书籍中大段的文字描述，不会因为时而出现的代码片段或公式阻碍阅读速度，然而读毕掩卷，有时一片茫然，犹如入宝山而空手归，有时满怀感想，却一时理不出头绪。此刻，如果能在书边看到行家的注解或点评，无论是旁敲侧击，还是当头棒喝，常能有拨云见日、豁然开朗之效。中国古典文化有注疏传统，将这类述而不作的经典诠释方式结合到新兴的技术学科中，未尝不是一种有益的探索。

由于软件工程著作之精妙之处常在概念和方法，故而多半都是自然语言的文字论述，而并不以程序语言为载体，这使得对软件工程著作的翻译相比其他领域更富挑战性。有如文学作品的翻译，往往难以顾全“信、达、雅”，软件工程经典的译者，也很少能有自信声明自己的译文在保持技术准确的同时流畅地传递了原文的语言个性——很多思维的微妙之处，都在原著文字的起承转合和字里行间。为原文提供中文注释，在技术上的启发意义之外也能扫清一些语言和文化差异造成的障碍，让更多力所能及的读者都能方便地直接欣赏原文，窥经典之全豹。

另一方面，正如 Roger S. Pressman 说：“软件工程将发生变化——对此我们可以肯定。”Frederick P. Brooks 在《人月神话》中所倡导的“唯一不变的是变化本身”，人们只有在变化中才能体味永恒。在当前的实践和技术氛围中咀嚼软件工程的理论，这样才是这一学科真正的活力源泉。注释者结合软件工程近年发展趋势，重访当年的经典，令读者能在软件技术发展的脉络中领会经典的精粹。借用 UMLChina 团队的口号“软件以用为本”，经典终究是为实践服务的，而我们出注释版图书最终的目的也是希望通过国内相关领域专业人士的注释，令英文原版符合国内读者的阅读需求，为原文增添技术上的辅助、语言上的答疑，以及抛砖引玉，激发读者的思辨。

试读结束：需要全本请在线购买： www.ertongbook.com

To Cindee: wings of your own

注释者序

在谈及 JUnit 时，Martin Fowler 有一句名言："世界上从没有任何软件能像 JUnit 一样用如此简洁的代码给人类带来如此伟大的财富。"如果作为测试驱动开发工具的 JUnit 能获得如此的评价，那么这本软件大师 Kent Beck 的名著，应该获得怎样的评价呢？我们是否可以说"世界上从来没有任何软件书籍能像《测试驱动开发》这样用如此简洁的文字给人类带来如此伟大的财富"？

测试驱动开发不是一种方法，而是一种理念。就像用 Java 时也可以把所有的代码都编写在 main 方法里，这样的代码毫无面向对象技术可言。编写测试用例也不代表测试驱动开发，重点是在"驱动"二字上，如果不能满足"没有测试用例决不编写任何代码"这一先决条件，就算不上测试驱动。

非常荣幸能够获得为本书做注释的机会。当我得到这个机会的时候，刚好距离我在网上发表关于 TDD 的博客一周年。回想自己这一年来的经历，TDD 几乎像狂风暴雨一样席卷了我的所有开发任务，我也经历了个人软件生涯中的第二次革命，而第一次，是从 C 语言转到面向对象。

所以当我得知能有这样的机会，为这本荣获美国第 14 届 Technical 效能大奖的书撰写注释的时候，激动的几夜都没有睡好。我觉得是时候向国内的读者介绍这种思想了，既然 TDD 可以成为 Eclipse 这样大型项目的灵魂，就一定能为国内诸多软件项目带来福音。

在本书的注释过程中，我没有拘泥于作者的原意，而是加入了大量 TDD 实践人员在工作中所有的实践经验和心得体会。正因为如此，注释并不是对原文的简单摘译，毕竟 Kent Beck 所领悟的真理并不是国内每一个程序员都能领悟到的。而注释版的最大优势在于让读者看到大师的本意，避免每一个人在"一千个林黛玉"中迷失了方向。

本书中有多种不同风格的注释内容，其中一些以小标题加以提示。本书的注释约定包括以下几类：

中文目录及每一章的中心思想

在原书目录之后给出了全书的中文目录，并提炼了每一章的中心思想，给出该章的整体线索，以便在读者阅读时有清晰的阅读思路。

核心理念

提炼出技术核心点和阅读关键点，对其进行详细说明和相关扩展，从而为读者在实际工作中遇到的问题提供更多的解决策略和更大的发挥空间。

术语解读

注释时从专业人员的角度对书中专业术语的出处、意义以及用法进行阐述和解释。

触类旁通

原书中的内容对读者的启迪，以及对当今技术发展的现实意义。

最佳实践

注释时基于我自己的行业实践经验总结的在测试驱动开发过程中需要注意的事项和规则。

关键步骤

对整个测试过程中比较关键的步骤进行的归纳总结。

用友软件公司的陈立新是国内 TDD 的专家，本书获得了他的大量帮助。陈立新认为虽然经过了数十年技术和方法论的改良，软件的设计环节和编码环节之间仍然存在巨大的鸿沟。而 TDD 致力于填补这个鸿沟，将设计和编码无缝地连接起来，由 TDD 建立的整个开发链条，上承自然语言描述的 Use Case，下启重构而来的“最合适”软件架构，正是软件开发人员——掌握着技术的人——所最需要的。

感谢陈立新和丁旭对注释工作的帮助，感谢杜银翠，是她提醒我应该将文中的 Python 代码改成 Java 语言描述，为国内的读者提供更多的方便。感谢国内致力于研究和推广 TDD 的同仁们，我们在网上收到了很多来自他们的意见和启发。感谢 Kent Beck、Martin Fowler、Eric Gamma 和其他 TDD 的先行者，为我们创造了如此的财富。感谢 Eclipse，如果没有这样的成功的案例，TDD 也不会有今天！

孙　方

SVN 咨询师

用友软件股份有限公司 U8 平台项目经理

国内最早一批推广测试驱动开发理念的专家

前　言

“代码整洁有效”（Clean code that works），Ron Jeffrey 的这句精辟的短语成为了测试驱动开发（TDD）的目标。基于以下的原因，Clean code that works 是值得我们去实现的目标。

- 它是一种可预测的开发方式。你可以知道什么时候能做完，而不用担心会有一长串的 bug 尾随着你。
- 它给你一个机会，让你可以学习所有代码能够教给你的方法。如果只是采用首先想到的方法，那么你将没有时间再考虑另一个更好的方法。
- 它将给予你的软件使用者更美好的使用体验。
- 它可以让团队内部之间互相信赖。
- 把它写出来的感觉简直太棒了。

但是我们怎样才能编写出整洁有效的代码呢？有许多外界因素使我们远离整洁的代码，甚至远离有效的代码。不过不用过多考虑这些恐惧，我们应该做的是：使用一组自动测试——测试驱动开发（TDD）的风格——来推动我们的开发工作。在测试驱动开发中，应该：

- 仅当测试失败的时候编写新代码；
- 消除重复代码。

这是两条简单的规则，但是它们会产生复杂个体和群体的行为规范，并具有以下技术含义：

- 必须有机地进行设计，让运行着的代码在不同决策之间提供反馈；
- 必须自己动手编写测试程序，因为我们不能每天浪费很多时间去等待别人编写测试程序；
- 我们的开发环境必须对微小的变化提供快速的反应；
- 为了简化测试，我们的设计必须由许多高内聚、低耦合的部分组成。

这两条原则还暗示了编程任务的一种顺序。

（1）红色——编写一个短小的、不能运行的甚至一开始可能无法编译通过的测试程序。

（2）绿色——尽快使测试得以通过，为此不惜做一些合理但是必要的事情。

（3）重构——消除上一步中产生的所有重复代码。

红色/绿色/重构——这是 TDD 的口头禅。

假设现在这种编程风格已经可行，那么它可能更进一步极大地减少代码的错误密度，并且使工作的主题对所有相关的人来说更加透明。如果是这样，那么仅编写那些失败的测试所需要的代码会有更深远的意义。

- 如果错误密度能够被充分减少，那么质量保证（QA）工作就能够从被动变为主动。
- 如果令人不愉快的意外可以充分减少，那么项目经理就能足够精确地估算，以便让实际客户参与日常开发。
- 如果技术讨论的主题可以足够清晰，那么软件工程师们大家合作就可以随时进行协同工作，而不是每次都要等一天或者一个星期。
- 再者，如果错误密度可以被充分减少，那么我们就可以每天都获得包含新功能的软件成品，从而打开通向与客户的商业关系的大门。

这样看来，这种概念很简单，但动机是什么？是什么驱使一个软件工程师花额外的精力来编写自动测试呢？是什么驱使一个软件工程师要把工作分成很多微小的步骤，即使他或她的头脑已经足够处理设计过程中的大量的宏观与微观的信息？答案是勇气。

勇气

测试驱动开发是在编程中控制恐惧的一种方法。当然，我不是说那种毫无道理的恐惧（客怜滴效衬许员徐要俺慰[1]），而是一种合理的但让人有“这是一个困难的问题，我看到了开头，却看不到结局”的恐惧感。如果痛苦是对“停下！”的本能反应，那么恐惧就是对“小心！”的本能反应。小心总有好处，但恐惧却有许多其他不良影响：

- 恐惧让你犹豫不决；
- 恐惧让你自闭保守；
- 恐惧让你畏首畏尾；
- 恐惧让你脾气暴躁。

这些不良影响对编程没有帮助，尤其是编写困难的程序的时候。因此问题变成了怎样面对一个艰难的环境，并且做到：

- 尽快开始具体的学习，而不是犹豫不决；
- 更清晰地交流，而不是自闭保守；
- 寻找具体的有帮助的反馈，而不是畏首畏尾；

[1] 这句话是“可怜的小程序员需要安慰”的谐音。

- （至于脾气问题，还是自己解决吧）。

设想编程就像转动摇柄把从井里提水一样。当水桶很小的时候，一个自由旋转的摇柄就够了。而当水桶很大并且装满水的时候，你可能会在水桶提上来之前就筋疲力尽。这时你需要一个防倒转的单向齿轮装置来让你有机会休息一下。水桶越重，单向齿轮装置上的两片齿之间就需要越靠近。

测试驱动开发中的测试就相当于单向齿轮装置上的每片齿。一旦让一个测试通过，我们就知道它将永远通过。每通过一个测试，就向成功迈近了一步。现在再让下一个测试通过，然后再下一个，再再下一个……以此类推，编程问题越艰巨，每个测试覆盖的范围就越小。

阅读过我的另一本书 *Extreme Programming Explained* 的读者可能会注意到极限编程（XP）和 TDD 在论调上的不同。TDD 不像 XP 那样绝对。XP 说的是“这里是一些你必须做的事情，为进一步演化做准备”，而 TDD 则有些含糊。TDD 是对编程过程中的决定和反馈间的差距的认知，以及控制这种差距的技术。“如果我花一星期在纸上做一个设计，然后通过测试驱动编码怎么样？那是 TDD 么？”当然，这是 TDD。你意识到了决定和反馈之间的差距，而且你有意地去控制了这种差距。

大部分学习 TDD 的人发现他们的编程实践朝好的方向改变了。“测试感染”（Test Infected）是 Erich Gamma 发明的用来描述这种变化的短语。你可能发现自己会更早地编写更多的测试，而且对你一直梦想的更细致的工作方式有了感觉。另一种情况是，一些软件工程师学习 TDD 之后再回到以前的编程实践，只是把 TDD 作为普通编程手段行不通时的特别的备用方法。

当然，（至少）不是所有的编程任务都能单独由测试来驱动的。例如，安全性软件和并发行为就是两个 TDD 不能够机械地演示软件目标的主题。尽管安全性确实依赖于无缺陷的代码，但是它同样依赖于人们对于为了保护软件而采用何种方法的判断。微小的并发行为也不能在代码的运行过程中可靠地再现。

阅读完这本书，你就可以：

- 随时开始；
- 编写自动测试；
- 用重构每次增加一个设计决策。

这本书分为 3 部分。

- 第 1 部分——货币例子，一个使用 TDD 编写代码的典型例子。许多年前我从 Ward Cunningham 那里得到这个例子，并且自从引入“多币种算法”以来已经多次用到过。这个例子将教会你如何在编写代码前先写测试，并有机地添加设计。

- 第 2 部分——xUnit 例子，一个逻辑上更复杂的程序的例子（包括映射和异常），为自动测试而开发一个框架。这个例子还会向你介绍 xUnit 结构，这是许多面向程序员的测试工具的核心组件。在第二个例子中，你会学到如何进行比第一个例子中更细致的工作，其中包括令许多计

试读结束：需要全本请在线购买： www.ertongbook.com

算机科学家激动不已的呼喊式自我提示（self-referential hoo-ha）。

- 第 3 部分——测试驱动开发的模式，这部分介绍决定编写什么样的测试怎样用 xUnit 编写测试的模式，以及如何成功地选择在上面的例子中用到的设计模式和重构方法。

我编写这些例子的时候考虑了两类不同的编程群体。如果你喜欢在四处乱逛之前先看地图，那么你可能会想直接跳到第 3 部分，然后照着例子的样子开始工作。如果你更喜欢先四处走走然后看地图以便知道你到了哪些地方，那么请试着认真看完每个例子，想了解一个技术细节的时候看一下模式，并把模式作为一种参考来使用。本书的一些审阅者评论说：当他们打开编程环境，输入代码，然后运行他们读到的测试，却发现他们最大的收获反而在这些例子之外。

对于例子的一点说明：多币种计算和测试框架这两个例子看上去简单一点。（我曾经见过）解决这些问题有另外一些复杂的令人不快的杂乱的方法。我本可以从这些复杂、令人不快的、杂乱的方法中选出一个来让这本书呈现一种“真实”的样子，但是，我的目标是——并且我希望你的目标也是——使代码整洁有效。在开始讨论这些“过于”简单的例子之前，请花 15 秒钟，想象一个编程的世界，其中的代码如此整洁、如此直接，不存在任何复杂的解决方案，只有显然很复杂的问题有待谨慎的思考。TDD 可以帮助你引导自己准确地进行这种谨慎的思考。

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.

试读结束：需要全本请在线购买： www.ertongbook.com