大 师 签 名 系 列

Test-Driven Development By Example

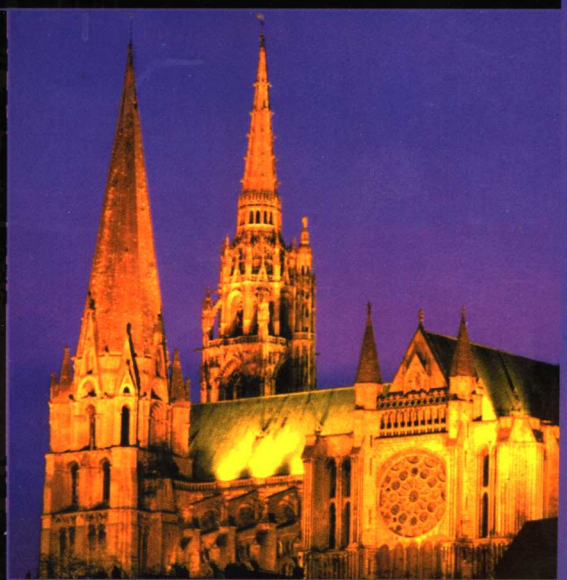
MASTER SIGNATURE SERIES 大师签名系列

测试驱动开发（中文版）

Software Development Productivity 大奖作品

[美] Kent Beck 著
孙平平 张小龙 赵辉 等译
崔凯 校

- 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



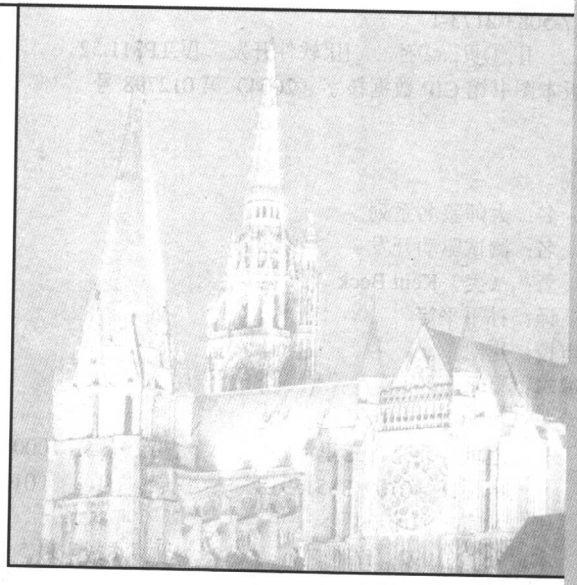
PEARSON Addison Wesley
Pearson Education
中国电力出版社
www.infopower.com.cn

大 师 签 名 系 列

Test-Driven Development
By Example

测试驱动开发
（中文版）

[美] Kent Beck 著
孙平平 张小龙 赵辉 等译
崔凯 校



中国电力出版社
www.infopower.com.cn

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

图书在版编目（CIP）数据
测试驱动开发 /（美）贝克著；孙平平等译. —北京：中国电力出版社，2004
（大师签名系列）
ISBN 7-5083-2173-1
Ⅰ.测... Ⅱ.①贝...②孙... Ⅲ.软件开发 Ⅳ.TP311.52
中国版本图书馆 CIP 数据核字（2004）第 012798 号

丛 书 名：大师签名系列
书　　名：测试驱动开发
编　　著：（美）Kent Beck
翻　　译：孙平平等
技术审校：崔凯
责任编辑：夏平
出版发行：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：北京丰源印刷厂
开　　本：787×1092　1/16　　印　　张：11.5　　字　　数：252千字
书　　号：ISBN 7-5083-2173-1
版　　次：2004 年 3 月北京第 1 版　　2004 年 5 月第 2 次印刷
定　　价：28.00 元

译者序

测试驱动开发（TDD）以测试作为开发过程的中心，它要求在编写任何产品代码之前，首先编写用于定义产品代码行为的测试，而编写的产品代码又要以使测试通过为目标。测试驱动开发要求测试可以完全自动化地运行，在对代码进行重构前后必须运行测试。这是一种革命性的开发方法，能够造就简单、清晰、高质量的代码。

测试驱动开发是一种我们编程时使用的技术。无论我们在开始编程之前进行了怎样的设计和建模，TDD都有助于我们提高代码质量。测试驱动开发可以赋予你对代码质量的自信以及对代码进行重构的勇气。试想如果没有办法保证我们对可运行代码的修改不会破坏任何先前的行为，那么怎么能够对代码进行修改？如果对代码的重构或修改无意中引入了bug但却没有一套可以立刻把这种情况告诉你的测试集，那么怎么能够进行集成？

测试驱动开发是一种在极限编程（XP）中处于核心地位的技术。要想采用极限编程过程，熟练掌握测试驱动开发将会有莫大的帮助。即便你选用的软件开发过程不是XP，测试驱动开发也能让你从中获益。采用测试驱动开发，我们将会得到简单、清晰的设计，我们的代码也将是清晰和bug-free的。同时采用测试驱动开发的结果就是可以让我们拥有一套伴随产品代码的详尽的自动化测试集。将来无论出于什么原因（新的需求，变化了的需求，性能调整等等）需要对产品代码进行维护时，在这套测试集的辅助（驱动）下工作，我们的代码将会一直是健壮的。

本书的作者是极限编程过程的缔造者，一线软件开发人员，其有关XP的书籍深受广大软件开发人员、项目经理的喜爱。本书语言朴实、诙谐，就像是结对编程（Pair Programming）中的同伴，耐心地传授自己的心得，实在是一本不可多得的讲述测试驱动开发的经典图书。译者在翻译过程中也是获益匪浅。由于作者在叙述过程中大量使用了俚语和俗语，所以译者也不敢有丝毫懈怠，力求在翻译过程中忠实反映作者的原意。但由于时间紧张，其中肯定还有因疏忽造成译文不妥的地方，恳请读者批评指正。

本书前11章由孙平平、张国强翻译，第12章～第19章由张小龙、张佳宁翻译，第20章～第26章由赵辉、唐晋涛翻译，最后6章由李恒、杨先炬翻译，全书由张伟统稿，由崔凯审校。如果本书能够对你有所帮助，那将是我们最大的心愿。

译者

前　言

代码整洁可用（clean code that works），Ron Jeffries 这句言简意赅的话，正是测试驱动开发（Test-Driven Development，TDD）所追求的目标。代码整洁可用之所以是一个值得追求的目标，是基于以下的一系列原因：

- 它是一个可预测的开发方法。你知道什么时候可以完工，而不用去担心是否会长期被 bug 困扰。
- 它给你一个全面正确地认识和利用代码的机会。如果你总是草率地利用你最先想到的方法，那么你可能再也没有时间去思考另一种更好的方法。
- 它改善了你的软件用户的生活。
- 它让软件开发小组成员之间相互信赖。
- 这样的代码写起来感觉很好。

但是我们要怎样做才能使代码整洁可用呢？很多因素妨碍我们得到整洁的代码，甚至是可用的代码。无需为此征求很多的意见，我们只需用自动运行的测试来推动开发，这是一种被称为测试驱动开发（TDD）的开发方式。在测试驱动开发中，我们要这样做：

- 只有自动测试失败时，我们才重写代码
- 消除重复设计，优化设计结构

这是两条很简单的规则，但是由此产生了复杂的个人和小组行为规范，技术上的含意是：

- 我们必须通过运行代码所提供的反馈来做决定，并以此达到有机设计的目的。
- 我们必须自己写测试程序，这是因为测试很多，很频繁，我们不能每天把大量的时间浪费在等待他人写测试程序上。
- 我们的开发环境必须能迅速响应哪怕是很小的变化。
- 为使测试简单，我们的整个规划必须是由许多高内聚、低耦合的部分组成。

这两条规则实际上蕴含了开发过程中所经历的阶段：

（1）不可运行——写一个不能工作的测试程序，一开始这个测试程序甚至不能编译

（2）可运行——尽快让这个测试程序工作，为此可以在程序中使用一些不合情理的方法

（3）重构——消除在让测试程序工作的过程中产生的重复设计，优化设计结构

不可运行/可运行/重构——这就是测试驱动开发的口号。

现在假设这样的开发方式是可能的，那么，再进一步，显著地减少代码的错误密度（defect density），让所有参与某一工作的开发人员对工作主题足够明了的假定也将成为可能。如果是这样的话，那么只有测试失败时才需要重写代码，其社会意义是：

- 如果代码的错误密度能够充分地减少，那么软件的质量保证（QA）工作可以由被动保证软件质量转变为主动保证软件质量。

- 如果开发过程中令人不快的意外能够充分地减少，那么项目经理能对软件开发进度有一个精确的把握，以便让实际用户参与日常开发。
- 如果每次技术讨论的主题都足够明确，那么软件工程师之间的合作是以分钟计算的，而不是按每天或每周计算。
- 再者，如果代码错误密度能够充分地减少，那么我们每天都可以得到有新功能的软件成品，并以此招揽新的用户群。

如此说来，观念是很简单的，但我的动机是什么呢？为什么一个软件工程师要做额外的写自动测试程序的工作？为什么一个设计观念可以瞬息万变的软件工程师却只能一小步一小步地进行工作？我们需要的是勇气。

勇气

测试驱动开发是一种可以在开发过程中控制忧虑感的开发方法。我并非指那些毫无意义的没有必要的担忧——（pow widdle prwogwammew needs a pacifiew[①]）——而是指合理的担忧，担忧是否合理是个很困难的问题，不能从一开始就看出来。如果说疼痛自然就会叫“停！”，那么担忧自然就会说“小心！”。小心谨慎是好的，但它也会产生以下一系列负面影响：

- 让你一直处于试验性的阶段。
- 让你不愿意与他人交流。
- 让你羞于面对反馈。
- 让你变得脾气暴躁。

这些负面影响对编程都是有害无益的，尤其是当需要编程解决的问题比较困难的时候。所以问题变为当我们面临一个比较困难的局面的时候，如何才能做到：

- 尽快开始具体的学习，而不是一直处于试验性的阶段。
- 更多地参与交流和沟通，而不是一直拒不开口。
- 寻找那些有益的、建设性的反馈，而不是尽量避免反馈。
- （依靠自己改掉坏脾气。）

设想把编程看成是转动曲柄从井里提一桶水上来的过程。如果水桶比较小，那么仅需一个能自由转动的曲柄就可以了。如果水桶比较大而且装满了水，那么还没等水桶被提上来你就会很累了。你需要一个防倒转的装置，以保证每转一次可以休息一会儿。水桶越重，防倒转的棘齿相距就应该越近。

测试驱动开发中的测试程序就是防倒转装置上的棘齿。一旦我们的某个测试程序能工作了，我们就知道，它从现在开始并且以后永远都可以工作了。相对于测试程序没有通过，我们距离让所有的测试程序都工作又近了一步。现在我们的工作是让下一个测试程序工作，然后再

① 这句话模仿了卡通人物 Elmer Fudd 的发音，意思是“poor little programmer needs a pacifier（可怜的小程序员需要安慰）”。——译者注

下一个，就这样一直进行下去。分析表明，编程解决的问题越难，每次测试所覆盖的范围就应该越小。

看过我写的《Extreme Programming Explained》[2]一书的读者可能会注意到我讲极限编程（XP）与讲测试驱动开发的语气是有区别的：讲测试驱动开发不像讲极限编程那么绝对。讲极限编程时我会说“这些是想进一步学习所必须具备的基础”，而讲测试驱动开发时要模糊一些。测试驱动开发教你认识编程过程中的反馈与欲实现的构思之间的差距，并且提供了控制这个差距大小的技术。“如果我在纸上作了一周的规划，然后通过测试驱动编码，这是否就是测试驱动开发？”当然，这就是测试驱动开发。你知道欲实现的构思与反馈之间的差距，并且有意识地控制了这个差距。

绝大多数学习测试驱动开发的人发现他们的编程习惯被永久地改变了。“测试感染”（Test Infected）是 Erich Gamma 所杜撰的用以描述这种转变的词语。你可能发现写测试程序变得容易了，并且相对较小的工作节奏比以前所梦想的节奏更明智。另一方面，一些学习测试驱动开发的软件工程师重新回到了以前的程序开发方法，并且保留测试驱动开发方法作为当其他开发方法不能奏效的特殊情况下的秘密武器。

当然也存在一些编程任务不能仅仅（或者根本就不能）由测试程序来驱动开发的情况。举个例子来说，软件的安全性和并行性，测试驱动开发方法就不能充分地从机械证明的角度说明软件是否达到了这两个目标。软件安全性从本质上来说依赖于无缺陷的代码。确实如此，但它同时也依赖于人们对软件安全机制的判断。精妙的并行问题不是仅靠再次运行代码就能可靠地再现的。

一旦读完本书，你要准备：

- 从简单的例子开始。
- 写自动测试程序。
- 重构，每次增加一个新的设计构思。

这本书是由三个部分组成的：

- 第一部分，资金实例（The Money Example）——一个典型的完全由测试驱动的代码模型的例子。这个例子是几年前我从 Ward Cunningham 那儿得到的，并且自从引入多币种算法以来已经多次用到过。你将从中学会如何在写代码之前写好测试程序，并最终发展成为一个有机的规划方案
- 第二部分，xUnit 实例（The xUnit Example）——一个逻辑上更复杂的程序的例子，包含反射（reflection）和异常（exception），通过建立自动测试框架来测试。这个例子同时也将向你介绍作为许多面向程序员的测试工具灵魂的 xUnit 结构体系。在第二个例子中你将学会以甚至比第一个例子更小的开发步骤工作，同时也包括深受许多计算机专家喜爱的呼喊式的自我提醒（self-referential hoo-ha）。
- 第三部分，测试驱动开发模式（Patterns for Test-Driven Development）——包括决定

② 本书影印版《解析极限编程》已由中国电力出版社引进出版。详情请访问：http//www.infopower.com.cn。——译者注

写哪些测试的模式，如何用 xUnit 写测试的模式和大量的设计模式精选以及例子中所用到的重构。

我写了关于结对编程（pair programming）的例子。如果你习惯于在四处转一转之前先看地图的话，你可以直接到第三部分去看那些模式，并将那些例子作为说明。如果你习惯于先到四处转一转，然后再看地图以确定自己处于什么位置的话，试着通读例子，当你需要了解更多关于某一技术问题的细节时，可以查阅后面所讲的模式，并将这些模式作为参考。本书的一些技术评审人指出，当他们启动编程环境，输入代码，运行所读到的测试程序时，最大的收获却在这些例子之外。

关于这些例子要注意一点。这两个例子，多币种计算和测试框架，看上去很简单。而解决同一个问题却也存在（我曾经见到过）一些复杂、风格很差、近乎弱智的解决方案。我本可以从这些复杂、风格很差、近乎弱智的解决方案中采用一个以使本书有一种“真实”感。然而，我的目标是写出整洁可用的代码，希望你的目标也是这样。在以那些被认为很简单的例子开始之前，花 15 秒的时间设想一下，如果所有的代码都能如此清晰和直接，没有复杂的解决方案，只有显然需要认真思考的很复杂的问题，那么这个世界会是什么样子。测试驱动开发可以引导你这样去认真思考。

致 谢

感谢那些始终对我提出严厉批评的审阅者。尽管本书所有的内容都是我一人所写，但如果没有他们的帮助，本书的可读性以及可用性均要大打折扣。按照我键入名字的顺序，他们分别是：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，Patrick Logan，Darren Hobbs，Aaron Sansone，Syver Enstad，Shinobu Kawai，Erik Meade，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 和 Jaime Nino。

所有曾与我并肩一起进行测试驱动开发的人，我非常感激你们能够执著地研究这种早些年听起来近乎疯狂的软件开发方法。我从你们那里所学到的要比我自己独自领悟到的多得多。在此，我不想冒犯所有其他的人，不过 Massimo Arnoldi，Ralph Beattie，Ron Jeffries，Martin Fowler 以及最后一位（重要人物）Erich Gamma 留在了我的记忆中，他们在驾御测试方面是最突出的，我从他们那里学到了很多东西。

我要感谢 Martin Fowler，感谢他给予我在使用 FrameMaker 上及时的帮助。他一定是这个世界上报酬最高的排版顾问，但幸运的是，他没有向我收取费用（到目前为止只有我一人受过这样的待遇）。

我的程序员生涯始于同我极富耐心的良师 Ward Cunningham 的合作，并且我将一如既往地同他进行合作。有时候我把测试驱动开发看成是一种可以给在任何环境下工作的软件工程师带去安慰感和亲切感的尝试。这种安慰感和亲切感我曾经在 Smalltalk 环境下开发 Smalltalk 程序时有过。如果两个人心有灵犀，那么就没有办法指出某个想法到底首先出自哪个人。如果你认为这本书中所有好的想法全是出自 Ward，那么也不见得就大错特错。

如果在此评说当一个家庭成员呕心沥血将自己怪异的思想编撰成书时这个家庭付出的牺牲的话，那么多少有点儿老调常弹了。这是因为写书过程中家庭所付出的牺牲就如同写书不能没有的纸张一样平常。感谢只有等我写完一章才能吃到早饭的孩子们，特别是要感谢我的妻子，在近两个月的时间里，凡事都再三提醒我，谨向他们致以最衷心的感谢。

感谢 Mike Henderson 温和的鼓励，感谢 Marcy Barnes 骑马救急。

最后，感谢在我令人琢磨不透的 12 岁时读过的一本书的无名作者。书中建议根据实际的输入计算相应的输出，然后开始编码直至实际结果与预期结果相一致。谢谢你们，谢谢，谢谢。

导 言

一个星期五的早晨，老板来找 Ward Cunningham，并把 Ward 介绍给 Peter 认识。Peter 有望成为公司开发的有价证券管理系统（WyCash）的用户。Peter 说："贵公司这套系统的功能给我留下了很深的印象，但是，我注意到这套系统仅能处理美元证券，我开设了一家新的证券基金，我的发展战略要求能够处理不同币种的基金"。老板转问 Ward，"那么，你看我们能做到这一点吗？"

这是任何软件设计者都可能遇到的噩梦般的一幕。你先前一直在一组假设条件下顺利而愉快地进行开发。而突然间，一切都变了。这个噩梦并非只针对 Ward 一个人，公司的老板，一个在指导软件开发方面经验丰富的老板，同样也不知道答案会是什么。

WyCash 系统是公司一个规模不大的开发小组两年来辛勤工作的成果。这个系统可以处理绝大部分美国市场上常见的各种各样的固定收益有价证券，还可以处理其他国家的一些新的投资证券，例如保利投资证券，而这是其他同类产品所不能处理的。

WyCash 系统一直是采用对象和对象数据库来进行开发的。作为构成基础计算要素抽象的 Dollar（美元），一开始是外包给一组聪明的程序员来完成的，他们开发的对象合并了信息格式化与计算两种功能。

在过去的六个月中，Ward 和小组的其他人员开始将 Dollar 对象的操作逐渐剥离出来。事实证明，Smalltalk 的数值类在计算方面工作得还是挺好的。用于四舍五入至三位十进制数字的复杂代码实际上有碍于产生精确的结果。随着结果精确度的提高，测试框架中用于在一定误差范围内进行比较的复杂机制由与预期或实际结果进行精确匹配的方法所取代。

用于完成信息格式化的操作实际上应由用户界面类来负责。由于测试代码，尤其是报告生成架构部分[①]的代码是在用户界面类一级编写的，所以这些测试程序无需修改就能适应这些改动。在经过六个月的认真剥离之后，Dollar 类所负责的操作已经所剩无几了。

系统中最复杂的算法之一，加权平均（weighted average），同样也经历了一个逐渐转变的过程。曾经有段时间，加权平均算法代码的各种变种遍布整个系统。就像报告生成架构是由最初众多的对象整合而来的一样，加权平均算法同样也会有一个容纳它的地方，这就是 AveragedColumn。

AveragedColumn 现在就是 Ward 要着手工作的地方。如果加权平均能够支持多种货币，那么系统剩下的部分就好办了。该算法的核心是将货币的数额保存在相应栏内。实际上，这个运算规则已经被抽象得足以计算任何对象的加权平均。举个例子来说，它可以计算日期的加权平均。

这个周末像往常一样地过去了。星期一早晨老板又过来了，他问道："怎么样，能做吗？"

---

① 有关报告生成架构的更多信息，请参见 c2.com/doc/oopsla91.html。

“再多给我一天时间，我将给你一个确切的答案。”Ward 说道。

Dollar 就如同加权平均里的一个计数器；因此，为使其支持多种货币，系统必须要有一种对象，这种对象针对每种货币都有一个计数器，就如同多项式一样。只不过其中的项可能是 15 USD 和 200 CHF，而不是 $3x^2$ 和 $4y^3$。

一个快速试验显示，用一个通用的 Currency（货币）对象代替 Dollar 对象进行计算是可能的，并且当两个不同币种的对象相加时返回一个 PolyCurrency 对象（多币种）。现在的问题是如何在不影响现有功能正常工作的情况下，为新功能腾出空间。如果 Ward 运行这些测试程序会是怎样呢？

在为 Currency 增加一些未实现的操作后，大多数的测试都通过了。在这一天结束的时候，所有的测试都通过了。Ward 把代码上传至源码控制系统并加入到产品建造过程中，然后跑到老板那里，“我们能做！”他自信地说道。

让我们总结一下这个故事。在两天的时间里，潜在的市场扩大了数倍，WyCash 系统的价值也扩大了数倍。有能力在这么短的时间里创造出这么大的商业价值绝非偶然。以下几个因素促成了这件事情的成功：

- 方法——Ward 和 WyCash 系统开发小组需要有一定的经验，知道如何持续渐进地进行系统设计。正是因为如此，为改进系统而采用的机制才能很好地工作。
- 动机——Ward 和他的小组必须对使 WyCash 系统支持多币种的商业价值有一个很明确的认识，并要有勇气接受这样一个看起来似乎是不可能完成的任务。
- 机遇——周全而激发自信的测试、构造良好的代码以及一种能够隔离设计构思的编程语言，这三者的结合意味着出问题的机会将会更少，并且出现的问题很容易辨认。

将项目的价值翻几番这种事情不是通过技术的魔力就能驾御的。而另一方面，方法和机遇却完全是在你的控制之中的。Ward 及其小组通过自己非凡的智慧、经验和行动准则创造方法和机遇。这是否意味着，如果你不是这个世界上最优秀的十名软件工程师之一，如果你在银行里没有一打一打的钞票，你就要对你的老板说你要求加薪，那么现在正好要行使这个权利，而这成功的时刻永远不会属于你呢？

当然不。即使你只是一个技能一般的软件工程师，即使面对压力曾使你乱成一团，偷工减料，你也绝对可以为你的项目找一个位置，在这个位置上你可以以魔术般不可思议的效果工作。测试驱动开发是任何软件工程师都可以学习的一个技术的集合，它鼓励采用简单的设计和增强自信的测试套件。如果你是个天才，你根本不需要学习这些浅显的规则。如果你是个傻子，学了也没有用。对于像我们这样介于二者之间的大多数人来说，遵从这两条简单的规则可以使我们以更加接近我们全部潜能的效率去工作。

- 在你写任何代码之前，先写一个会失败的自动测试程序
- 消除重复设计，优化设计结构

到底要如何这样做？如何巧妙地分阶段应用这些规则？这两条简单的规则可以运用多深？这正是本书要讲的主题。我们将从 Ward 在灵感爆发瞬间所创造的对象——多币种资金（multi-currency money）开始谈起。

写在后面的话

Martin Fowler

有关测试驱动开发最难讲清楚的一种东西就是它把你所带到的那种思维状态。我记得在和Ralph Beattie 开发原先的 C3 项目时，有一回，我们必须要实现一套复杂的支付判定。Ralph把它们分解成一组测试用例，我们逐条地使其运行通过。工作有条不紊地进行，由于工作起来不是慌里慌张的，所以进度似乎显得有些慢，但当我们回过头看看做了多少工作时，发现尽管没有慌里慌张的感觉，但进展着实迅速。

尽管我们拥有各种优秀的工具，但编程还是很难。我记得在我编程时，很多次我都觉得就像一下要保持好几个球在空中，稍有不慎所有的球都会呼啦掉下来。测试驱动开发帮助我们减少这种感觉，因此也能在不慌不忙中快速地进行开发。

我认为存在这种效果的原因是这样的，在你用测试驱动开发方式工作时，你会有一种只保留一个球在空中的感觉，因此你可以全身心地关注那个球，因而可以处理得很好。当我要增加某个新功能时，我不必担心为了这个新功能怎样设计才算好，我只要尽可能简单地写一个测试并通过就行了。同样，在进行重构时我不必操心增加新的功能，而只关心如何进行合理的设计。对于两者我一次只要关注其一，因此我能够更好地集中注意力在一件事情上。

通过测试优先和重构增加功能是编程的两项独立的逻辑。最近在键盘旁工作时，我又发现了另外一个方法：模式拷贝（pattern copying）。那时我在用 Ruby 语言写一个抽取数据库数据的脚本程序。在做这件事情的同时，我开始着手编写一个包装数据库表的类，心想既然我刚刚看完了一本关于数据库模式的书，那我也应该用一用模式。尽管范例代码是 Java，但要把它改成 Ruby 并不难。我编程时并没有真正想过那个问题，我只是考虑如何改编这一模式让其适合这种语言以及我正在操纵的特定数据。

模式拷贝就其本身并不是好的编程方法——这是在我谈模式时总是强调的事实。模式一般都是半成品，用到你的项目中还要再回一次炉。然而处理这一问题的一个好的办法是：一开始先不用管那么多，把模式拷贝过来，接下来采用重构和测试优先相结合的办法对其进行改编。这样一来，当你在做模式拷贝的时候，你可以把注意力集中到模式上——同一时刻只干一件事。

XP（极限编程）社团一直致力于研究在什么地方引入模式。显然 XP 成员喜欢使用模式，毕竟 XP 倡导者与模式倡导者之间有很多共同之处——Ward 和 Kent 同时是这两个领域的带头人。也许模式拷贝是继测试优先和重构之后的第三种单一逻辑模式，和前两者一样，单独使用一种时是危险的，但协调使用时功能强大。

要想系统地组织活动，很大程度上取决于核心任务的辨识，这使得我们每次能够集中注意力在一件事情上。装配线就是这样一个使人头脑麻木的例子——头脑麻木是因为你一直在做同

一件事。也许测试驱动开发所提议的是一种将编程切分成各种模式要素的方法，但是为了避免单调乏味而在这些模式要素间快速切换。单一逻辑模式与各模式间的快速切换相结合有利于你集中注意力并降低对大脑的压力，却没有装配线般的单调感觉。

我得承认这些想法有些不成熟。在我写这段文字的时候，我还无法确信我是否相信自己所说的这些内容，而且我知道自己还得仔细推敲个把月。但是我想也许你会喜欢这些评述的，而且或许能够激励你思考测试优先开发所适合的大环境。究竟是怎样的环境我们目前还说不清楚，但是我想它会慢慢自己展现出来的。

目　录

第一部分　资金实例

第二部分　xUnit 实例

第三部分　测试驱动开发的模式