

大师签名系

本页已使用福昕阅读器进行编辑。
福昕软件(C)2005-2007，版权所有，
仅供试用。

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)

Kent Beck

Copyright ©2003 Pearson Education, Inc.

Original English Language Edition Published by Pearson Education, Inc.

All rights reserved.

Translation edition published by PEARSON EDUCATION ASIA LTD and CHINA ELECTRIC POWER PRESS,
Copyright © 2004.

本书翻译版由 Pearson Education 授权中国电力出版社在中国境内（香港、澳门特别行政区和台湾地区除外）
独家出版、发行。

未经出版者书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有 Pearson Education 防伪标签，无标签者不得销售。

北京市版权局著作权合同登记号 图字：01-2003-1015 号

For sale and distribution in the People's Republic of China exclusively (excluding Taiwan, Hong Kong SAR and
Macao SAR).

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）销售发行。

图书在版编目（CIP）数据

测试驱动开发 / （美）贝克著；孙平等译．—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

版权所有 翻印必究

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 this 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。

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。

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

译者序	
前言	
致谢	
导言	
写在后面的话	

第一部分 资金实例

第 1 章 多币种资金	3
第 2 章 变质的对象	10
第 3 章 一切均等	13
第 4 章 私有性	16
第 5 章 法郎在诉说	18
第 6 章 再谈一切均等	21
第 7 章 苹果和桔子	25
第 8 章 制造对象	27
第 9 章 我们所处的时代	31
第 10 章 有趣的 Times 方法	36
第 11 章 万恶之源	41
第 12 章 加法, 最后的部分	44
第 13 章 完成预期目标	48
第 14 章 变化	53
第 15 章 混合货币	57
第 16 章 抽象, 最后的工作	61
第 17 章 资金实例回顾	65

下一步是什么?	65
比喻	66
JUnit 的用法.....	66
代码统计	67
过程	68
测试质量	68
最后一次回顾	69

第二部分 xUnit 实例

第 18 章 步入 xUnit.....	73
第 19 章 设置表格.....	77
第 20 章 后期整理.....	80
第 21 章 计数.....	83
第 22 章 失败处理.....	86
第 23 章 如何组成一组测试.....	88
第 24 章 xUnit 回顾.....	93

第三部分 测试驱动开发的模式

第 25 章 测试驱动开发模式.....	97
测试 (名词)	97
相互独立的测试 (Isolated Test)	98
测试列表 (Test List)	99
测试优先 (Test First)	100
断言优先 (Assert First)	101
测试数据 (Test Data)	102
显然数据 (Evident Data)	103
第 26 章 不可运行状态模式.....	104
一步测试 (One Step Test)	104
启动测试 (Starter Test)	105
说明测试 (Explanation Test)	106
学习测试 (Learning Test)	106

另外的测试	107
回归测试 (Regression Test)	108
休息	108
重新开始	109
便宜的桌子, 舒适的椅子	110
第 27 章 测试模式	111
子测试 (Child Test)	111
模拟对象 (Mock Object)	111
自分流 (Self Shunt)	112
日志字符串 (Log String)	113
清扫测试死角 (Crash Test Dummy)	114
不完整测试 (Broken Test)	115
提交前保证所有测试运行通过	116
第 28 章 可运行模式	117
伪实现 (直到你成功)	117
三角法 (Triangulation)	119
显明实现 (Obvious Implementation)	119
从一到多 (One to Many)	120
第 29 章 xUnit 模式	122
断言 (Assertion)	122
固定设施 (Fixture)	123
外部固定设施 (External Fixture)	125
测试方法 (Test Method)	125
异常测试 (Exception Test)	127
全部测试 (All Tests)	127
第 30 章 设计模式	129
命令	130
值对象	131
空对象	132
模板方法	133
插入式对象	134
插入式选择器	135
工厂方法	137
冒名顶替	137
递归组合	138

收集参数	140
单例模式 (Singleton)	141
第 31 章 重构	142
调和差异 (Reconcile Differences)	142
隔离变化 (Isolate Change)	143
数据迁移 (Migrate Data)	143
提取方法 (Extract Method)	145
内联方法 (Inline Method)	145
提取接口 (Extract Interface)	146
转移方法 (Move Method)	147
方法对象 (Method Object)	148
添加参数 (Add Parameter)	149
把方法中的参数转变为构造函数中的参数	149
第 32 章 掌握 TDD	150
附录 A 影响图	161
反馈	162
附录 B 斐波纳契数列	164

第一部分

资金实例

在第一部分，我们将会开发一个典型的完全由测试驱动的代码模型（那些跑题的部分除外，它们纯粹是出于教学的需要）。我的目标是让你迅速了解测试驱动开发（Test-Driven Development, TDD）的过程，这个过程大体上可以归纳为以下几个步骤：

- (1) 快速新增一个测试。
- (2) 运行所有的测试，发现最新的测试不能通过。
- (3) 做一些小小的改动。
- (4) 运行所有的测试，并且全部通过。
- (5) 重构（refactor）代码，以消除重复设计（duplication），优化设计结构。

你可能产生的疑问包括：

- 每个测试是怎样覆盖一小部分新增功能的
- 为了使新测试运行通过，我们所做的改动有多少、方法有多么笨
- 多长时间运行一遍测试
- 重构是由多少微小的步骤组成的



为此，我们将建立一个计划清单（to-do list）以提醒我们需要做些什么事情，它将使我们始终保持注意力集中，同时它也可以告诉我们什么时候可以完工。当我们开始某一项工作时，我们用粗体来表示它，**就像这样**。当我们完成某项工作时，我们将其划去，**就像这样**。如果我们想起其他要做的测试，就将其加入清单。

正如前面的计划清单所讲的一样，我们就从实现乘法这个功能开始。那么，我们首先需要建立什么对象呢？什么对象也不需要。记住，我们不是从建立对象开始，而是从测试开始。（我一直都在提醒自己注意这个问题，希望你也能时刻记住提醒自己。）

既然如此，那么我们首先应该进行什么测试呢？清单中的第一个测试看起来很复杂，我们需要从比较简单的开始。第二个测试不过是实现乘法功能而已，能难到哪儿去呢？我们就从它开始吧。

在编写测试的时候，我们总是为我们的操作设想最完美的接口（interface）。我们总是告诉自己这些操作在外界看来应该是个什么样子，尽管很多时候我们的设想并不能化为现实，最好是从一种尽可能优秀的应用编程接口（application program interface, API）开始，然后再倒着进行设计，这要比从一开始就把一切都搞得很复杂、拙劣而“现实”好。

下面是一个关于乘法功能的简单实例：

```
public void testMultiplication() {  
    Dollar five= new Dollar(5);  
    five.times(2);  
    assertEquals(10, five.amount);  
}
```

[我知道，我知道！这段代码有很多问题：公共域问题，副作用问题，货币金额用整数来表示的问题，等等。别急，一步一步来。我们将这些毛病记录下来，然后继续前进。显然，测试没有通过，但是我们希望测试能够尽快到达可运行状态（green bar）。^①]

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

我们刚才键入的测试程序甚至还不能通过编译。[我会在后边讲测试框架（testing framework）JUnit 的时候解释在什么地方键入以及怎样将其键入。]修改这样的测试非常简单。即便是编译后也无法运行，但为了使其能够编译通过，我们至少要做哪些工作呢？我们存在以下四个编译错误：

- 没有 Dollar 类

① Junit 测试工具运行测试时，如果测试全部运行通过，那么状态条是绿色的；如果存在没有通过的测试，那么状态条就是红色的。本书作者大量使用包含 green 或 red 的字句，我们以后统一将其译作测试运行通过或没有通过。——译者注

可以看出测试程序没有运行通过（red bar）。我们在测试框架（在该例中为 JUnit）中运行了这个作为开篇所编写的一小段代码，可以发现，尽管我们希望结果是“10”，事实上却很不幸，我们看到的结果是“0”。

没有关系，失败也是一种进步。我们已经对这次失败有了一个具体的衡量，这要比只是模糊地知道自己要失败的好。我们要解决的编程问题已经由原来的“实现多币种”转化为“让这个测试程序能够工作，然后让剩下的测试程序也能够工作”。问题已经比以前简单多了，要考虑问题的范围也小了很多。而且，我们完全可以让这个测试程序工作起来。

你也许不喜欢这个解决方案，但是现在的目的不是获得最完美的解决方案，而是让这个测试程序可以运行。我们将在做出理想的产品之前做出点牺牲。

下面是我所能想到的可以让测试程序通过的最小改动：

Dollar

```
int amount= 10;
```

图 1-2 显示了测试程序再次运行后的结果。现在测试程序运行通过，可喜可贺！

不过不要高兴得太早，致力于电脑编程的男孩女孩们，这一轮的工作还没完成呢！世界上恐怕很难找到几个输入可以让这个功能有限、风格很差、近乎弱智的测试程序运行通过。所以，我们在继续前进之前要把它一般化。记注，这一轮工作由下列的环节组成：

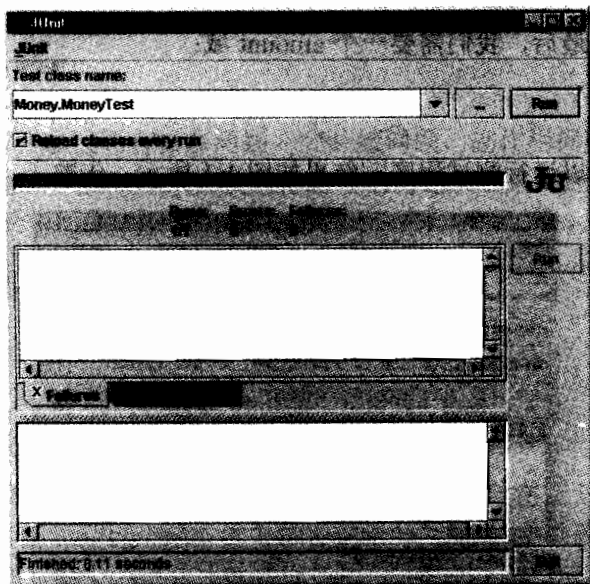


图 1-2 测试程序运行

- (1) 新增一个测试。
- (2) 运行所有的测试程序并失败。
- (3) 做一些小小的改动。

- (4) 运行所有的测试程序，并且全部通过。
- (5) 重构代码以消除重复设计，优化设计结构。

依赖关系 (dependency) 与重复设计 (duplication)

Steve Freeman 指出：测试程序与代码所存在的问题不在于重复设计（关于重复设计的概念我们还没有提到过，但我将在这段闲话说完后尽快向你解释），而在于代码与测试程序之间的依赖关系——你不可能只改动其中一个而不改动另外一个。我们的目标是编写另外一个对我们有用的测试而不必改动代码，而这对于当前实现而言是不可能的。

依赖关系是各种规模的软件开发中的关键问题。如果你让一家厂商的 SQL 具体实现散布在整个产品代码中，而又决定换成另一家厂商，那么就会发现，你的代码依赖于某家数据库厂商，你在不修改代码的情况下无法改变数据库。

如果问题出在依赖关系上，那么其表现就是重复设计。重复设计通常表现为逻辑上的重复设计——相同的表达式在代码的多个地方出现。利用各种对象可以很好地抽象出逻辑上的重复设计。

与现实生活中的大多数问题不同，在现实生活中，消除某种症状往往使问题以其他更恶劣的形式重新表现出来，消除程序中的重复设计就是消除依赖关系，这就是测试驱动开发第二条规则的由来。只有在编写下一个测试之前消除现有的重复设计，通过一处且仅仅一处改动即可让下一个测试运行通过的可能性才最大。

(1) ~ (4) 项我们都已运行过了。那么什么地方有重复设计呢？通常重复设计存在于两段代码之间，但是在这儿重复设计却存在于测试中的数据与代码中的数据之间。你看到了吗？如果我们这样写会怎样呢：

Dollar

```
int amount= 5 * 2;
```

这个 10 必然有它的来历，我们只顾在大脑中快速地做乘法以至于将这点忽略了。在这儿的 5 与 2 处于两个不同的地方，所以依照规则，在我们继续之前必须毫不留情地消除重复。

我们无法只通过一步就消除 5 和 2。既然如此，可以不在对象初始化时给 amount 赋值，而将这个过程移至 times() 方法中。

Dollar

```
int amount;
```

```
void times(int multiplier) {  
    amount= 5 * 2;  
}
```

测试仍然通过，测试程序保持在可运行状态。

这样的软件开发步伐对于你来说是否太小了？记住，测试驱动开发并非一定要采取这样一小步一小步的开发过程，而是要培养你将软件开发化为这样的一小步一小步的开发任务的能力。我日复一日都以这样小的步伐进行开发吗？当然不是。但是当情况变得有些棘手时，我很高兴我有这样的能力。选择一个简单的例子一步一步来尝试，来学习。如果你可以将软件开发分成一个个粒度比较小的开发任务，那么你自然可以将它分得大小适当。但是如果你仅仅采用较大的步伐进行开发，那么你根本不会知道较小的步伐是否合适。

言归正传。我们刚才谈到哪儿了？对，谈到如何消除测试代码和工作代码之间的重复。我们可以从哪儿得到一个 5 呢？这是传给构造函数的值，所以我们用 `amount` 变量来保存它：

Dollar

```
Dollar(int amount) {  
    this.amount= amount;  
}
```

然后我们就可以在 `times()` 函数中使用它：

Dollar

```
void times(int multiplier) {  
    amount= amount * 2;  
}
```

参数“`multiplier`”的值是 2，所以我们可以用这个参数来代替这个常量：

Dollar

```
void times(int multiplier) {  
    amount= amount * multiplier;  
}
```

为了证明我们精通 Java 的语法，我们将使用 `*=` 操作符（这确实削减了重复内容）：

Dollar

```
void times(int multiplier) {  
    amount *= multiplier;  
}
```

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“`amount`”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

现在可以说第一个测试已经完成了，下一步我们将解决那些奇怪的副作用问题。在此之前，让我们回顾一下，我们做了以下的工作：

- 创建一个清单，列出我们所知道的需要让其运行通过的测试
- 通过一小段代码说明我们希望看到怎样的一种操作

- 暂时忽略 JUnit 的一些细节问题
- 通过建立存根 (stub) 来让测试程序通过编译
- 通过一些另类的做法来让测试运行通过
- 逐渐使工作代码一般化，用变量代替常量
- 将新工作逐步加入计划清单，而不是一次全部提出

2

变质的对象

测试驱动开发的总体流程如下：

(1) 写一个测试程序。考虑一下你希望实现的操作要如何在你的代码中体现出来。你是在写一个故事 (story)。设想你希望拥有的接口 (interface)。在故事中包含任何你所能想像到的、计算出正确结果所必需的元素。

(2) 让测试程序运行。尽快地让测试程序可运行是压倒一切的中心任务。如果明显存在一个整洁、简单的解决方案，那么就将其键入。如果这个明显整洁、简单的解决方案需要耗费一分钟的时间，那么就把它记下来，再回到主要问题上来，即怎样才能让测试在几秒钟内就能运行通过。对许多有经验的软件工程师来说，这种偏离审美的举动是难以理解的。他们只知道怎样遵循良好的工程法则。快速使测试运行通过是一切行为的理由。但这只是暂时的。

(3) 编写合格的代码。现在系统已经能够工作了，抛开过去那些为了让测试运行而使用的不光彩的手法。现在我们又得重归正派的软件设计之路。消除先前引入的重复设计，使测试尽快运行通过。

我的最终目标是整洁可用的代码 (clean code that works, 感谢 Ron Jeffries 的这句言简意赅的总结)。即使是最优秀的程序员有时候也无法得到整洁可用的代码，大多数普通的程序员 (比如我) 大部分时间无法得到整洁可用的代码。怎么办？办法是分而治之。首先解决目标中的“可用”问题，然后再解决“代码的整洁”问题。这与体系结构驱动 (architecture-driven) 的开发相反，在那种开发方法中首先解决“代码的整洁”问题，然后才四处搜罗，努力把在解决“可用”问题过程中学到的内容整合到设计中去。

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

我们让一个测试运行通过了，但在这一过程中我们注意到一些奇怪的现象：当我们在 Dollar 对象上施加一个操作后，Dollar 对象改变了。假设我写的代码是：

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
    five.times(3);
    assertEquals(15, five.amount);
}
```

我想不出一一种干净漂亮的解决办法来让测试程序按照我的意图工作。第一次调用 `times()` 函数后, `five` 已经不再是 5 了, 它实际上是 10。可是, 如果我们从 `times()` 函数返回一个新对象, 那么我们可以多次对原来的 `five` 进行操作, 但不会让它发生丝毫变化。我们做这样的改变实际上是改变了 `Dollar` 类的接口, 因此我们必须相应地改动测试程序。没有关系。我们所设想的恰当接口是不大可能比对恰当实现的设想更完善的。

```
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);
}
```

除非我们改变 `Dollar.times()` 函数的声明, 否则新的测试程序不能编译。

Dollar

```
Dollar times(int multiplier) {
    amount *= multiplier;
    return null;
}
```

现在测试程序可以编译了, 但它还不能运行。有进步! 要让它运行, 需要返回一个新的带有正确的 `amount` 值的 `Dollar` 对象:

Dollar

```
Dollar times(int multiplier) {
    return new Dollar(amount * multiplier);
}
```

当瑞士法郎与美元的兑换率为 2:1 的时候, 5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将 “amount” 定义为私有

Dollar 类有副作用吗?

钱数必须为整数?

在第 1 章, 为了让一个测试程序工作, 我们从这个测试程序的伪实现 (bogus implementation) 开始逐渐将它变成真实的 (real)。在这儿, 我们把自认为是正确的实现代码键入并且祈祷 (片刻的祈祷者, 确实如此, 因为测试程序运行不过是几毫秒的事) 它能运行成功。我们很幸运, 测试程序运行通过了, 所以我们又可以从清单中划去一项。

下面是我所了解的尽快使测试程序可运行的三条策略中的两条:

- 伪实现——返回一个常量并逐渐用变量代替常量,直至伪实现代码成为真实实现的代码
- 显明实现 (Obvious Implementation)——将真实的实现代码键入

当我在实际开发中运用测试驱动开发的时候,我经常交替使用这两种实现模式。如果一切都很顺利,而且我知道该写些什么,我就会一个接一个地采用显明实现(每次都要运行测试程序,以保证对你来说想当然的事情对电脑来说也没错)。一旦测试没有运行通过,我就会退回去转而采用伪实现,重构直至得到正确的代码。当我重新恢复自信的时候,我又会再次开始使用显明实现。

测试驱动开发还有第三种方式,三角法 (Triangulation),关于三角法的内容我们将在第3章介绍。让我们回顾一下,在这一章我们做了哪些工作:

- 将一个设计缺陷(副作用)转化为一个由此缺陷导致运行失败的测试程序。
- 采用存根实现使代码迅速编译通过。
- 键入我们认为正确的代码以使测试程序能尽快工作。

将一种感觉(例如,对副作用的厌恶)转化为一个测试程序(例如,对同一个 Dollar 对象连乘两次)是测试驱动开发中常见的方式。我这样做的时间越久,我将自己对审美的判断转化为测试程序的能力就越强。如果我能够做到这一点,那么设计讨论就会变得有意思多了。首先我们讨论系统应当是这样还是那样工作。一旦就系统的行为达成一致,我们就开始谈论如何用最好的办法来实现它。我们可以一边喝酒,一边不顾现实地大谈程序要如何工整,如何严谨。但当我们编程的时候,就必须抛弃这些不切实际的空谈,从具体的案例出发。

3

一切均等

假设我有一个整数并把它加 1，但我并不希望原来的整数发生变化，而只是想使用新的数值，该怎么办？对象的行为通常不是这样的。假设我有一份契约，并且我将其保证金加 1，那么这个契约的保证金就应该发生变化（是的，是的，这是很有意思的商业规则，但在这里与我们无关）。

正如我们现在使用美元对象 Dollar 一样，我们可以把对象当作数值来使用，这样的对象称为数值对象（Value Object）。数值对象的一个要求就是一旦数值对象的实例变量值在构造函数中被指定，那么以后就再也不允许发生变化。

使用数值对象有一个巨大的优势：你不用担心别名（aliasing）问题。比如我有一张支票，将其金额设置为 5 美元，然后同样也把另一张支票的金额设置为 5 美元。当我修改第一张支票的金额却无意中更改了第二张支票的金额时，我职业生涯中最讨厌的 bug 出现了。这就是别名问题。

如果你有了数值对象，就不用担心别名问题了。如果我有一个 5 美元的对象，那么我将保证它一直是并且永远是 5 美元。如果有人想要 7 美元，那么他们就必须创建一个全新的对象。

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

数值对象的一个隐含的意思就是所有的操作都必须返回一个新的对象，就如同我们在第 2 章中所见到的一样。另一个隐含的意思是使用数值对象必须要实现 equals() 函数。因为一个 5 美元的对象跟其他 5 美元的对象几乎没有什么差别。

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗?

钱数必须为整数?

实现 equals() 函数

实现 hashCode() 函数

如果你用 Dollars 作为散列表的键值 (key), 那么如果要实现 equals() 就必须实现 hashCode()。我们也将它加入计划清单, 当有必要的时候再来解决它。

你根本就没有考虑过该如何实现 equals() 函数, 是吗? 好的, 其实我也没有。在用尺子狠狠地打了一下我自己的手背后, 现在让我们来考虑如何测试相等性 (equality)。首先, 5 美元应该等于 5 美元:

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

测试没有运行通过。equals() 的伪实现直接返回 true:

Dollar

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

你我都知道返回 true 实际上是因为 “5==5”, 事实上是 “amount==5”, 也就是 “amount==dollar.amount”。如果这些我都一步步地实现, 那么我就没办法向你说明测试驱动开发的第三种并且是最保守的实现方法: 三角法 (Triangulation)。

如果两个已知间距的接收站都能测定无线电信号的方向的话, 那么就有足够的信息计算信号的方位和范围 (无论如何, 你的三角知识要比我多)。这种计算我们称之为三角法。

与之类似, 当采用三角法的时候, 我们只有在例子达到 2 个或更多时才对代码实施一般化。我们暂时忽略测试程序和模型代码之间的重复设计。当且仅当第二个例子需要更一般化的解决方案时, 我们才开始对代码实施一般化。

因此, 要使用三角法, 我们需要第二个例子。5 美元 != 6 美元怎么样?

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

现在我们要使判等函数 equals() 一般化:

Dollar

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

当瑞士法郎与美元的兑换率为 2:1 的时候, 5 美元+10 瑞士法郎=10 美元
5 美元*2=10 美元

4

私有性

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

既然我们已经定义了相等的概念，我们就可以使用它以使我们的测试程序更加“有说服力”。从概念上讲，Dollar.times() 操作应该返回一个 Dollar 对象，这个对象的值是原对象的值乘以乘数。但我们的测试程序并没有明确这一点：

```
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);  
}
```

我们可以重写第一个断言（assertion），让 Dollar 对象之间进行比较：

```
public void testMultiplication() {  
    Dollar five= new Dollar(5);  
    Dollar product= five.times(2);  
    assertEquals(new Dollar(10), product);  
    product= five.times(3);  
    assertEquals(15, product.amount);  
}
```

这样看起来好一些，所以我们同样也重写第二个断言：

```
public void testMultiplication() {  
    Dollar five= new Dollar(5);
```



```

Dollar product= five.times(2);
assertEquals(new Dollar(10), product);
product= five.times(3);
assertEquals(new Dollar(15), product);
}

```

现在临时变量 `product` 的作用已经不大，我们可以采用内联 (`inline`) 方式将其直接插入进来：

```

public void testMultiplication() {
    Dollar five= new Dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}

```

现在这个测试程序看起来已经很清楚了，仿佛它并不是一系列的操作，而是一个关于正确与否的断言。

测试程序经过这样的一些修改，现在 `Dollar` 类是惟一个使用其实例变量 `amount` 的类，所以我们可以将 `amount` 设为私有：

```

Dollar
private int amount;

```

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

现在我们又可以将另外一项从计划清单中划去了。请注意实际上我们是在冒险。如果关于判等的测试不能准确地说明判等功能可用的话，那么关于乘法的测试也不能准确地说明乘法功能可用。这是我们在测试驱动开发中必须积极面对的一个风险。我们并非要追求绝对的完美。通过表述任何事物都从代码和测试两条线路来走，我们希望尽可能地减少缺陷，以在前进的路上增强自信。有时我们的推理并不严密，这会让某个缺陷悄悄溜进来。如果发生这种情况，我们就吸取本应该编写测试的教训，然后继续前进。其他时间我们则是在测试运行通过的指引下继续大胆前行。

作为回顾，我们在这一章做了以下的事情：

- 使用了刚刚开发的功能来改进一个测试
- 观察到如果两个测试同时失败，那么情况可就严重了
- 尽管有风险，但还是要继续前进
- 使用了测试中对象的新功能以减少测试程序与代码之间的耦合度

5

法郎在诉说

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

我们如何处理清单上的第一个测试呢？这是其中最令人感兴趣的测试。不过看起来仍然挺难的。我没有把握写一个只需一小步就能实现它的测试程序。可以看出一个先决条件是必须要有一个类似 Dollar 的对象，但这个对象代表的是法郎。如果我们能让这个法郎对象像现在的美元对象一样工作，那么距离编写和运行把两类不同类型对象混合相加的测试就更近了。

我们可以拷贝并编辑 Dollar 测试程序：

```
public void testFrancMultiplication() {  
    Franc five= new Franc(5);  
    assertEquals(new Franc(10), five.times(2));  
    assertEquals(new Franc(15), five.times(3));  
}
```

（你不会对我们第 4 章将测试程序进行简化的工作不满意吧？那样做使我们这儿的工作变得容易多了。难怪很多书中都经常采用这样一种讲解方式。事实上我这次并不是有意这样做的，但是我不能保证以后不会特意这样安排。）

采用什么快捷方式可以让我们的新测试程序运行通过呢？那就是拷贝美元 Dollar 的实现代码，并用法郎 Franc 代替美元 Dollar。

停，请等一下。我能够听到你们当中那些倾向于完美设计的人正在对此嗤之以鼻。通过拷贝粘贴来复用代码？抽象工作哪里去了？就这样抛弃整洁设计了吗？

如果这使你感到失落，那么换一口新鲜空气。从鼻子吸气……屏住！1，2，3……从嘴巴呼气。好了，记住，每个开发周期（cycle）都分不同的阶段（它们进行得很快，一般都是以秒计算的，但是它们只是不同的阶段而已）：

- (1) 写一个测试程序。
- (2) 让测试程序编译通过。
- (3) 运行测试程序，发现不能运行。
- (4) 让测试程序可以运行。
- (5) 消除重复设计，优化设计结构。

不同的阶段有不同的目的。它们需要不同的解决方式，不同的审美观。前三个阶段需要很快完成，这样我们就到达一个包含新功能的已知状态。你可以不择手段地到达这一状态，这是因为就在此短暂的一刻尽快地让测试程序运行通过比如何设计更重要。

现在我很担心。我给了你可以抛弃一切良好设计原则的自由。你可能会跑去跟你的小组同事说：“Kent 说了，所有的设计原则都无足轻重”。慢，这一开发周期还没有结束呢。这下子真是大跌眼镜！没有第五步，整个周期的前四步也没什么意义了。要适时地进行设计。该让它运行通过时就让它运行通过，该优化时就进行优化。

好了，我感觉好些了。现在，可以肯定，在消除重复代码之前，除了你的同伴以外你是不会把代码给任何人看的。我们讲到哪儿了？哦，对了，讲到为了追求速度而违反一切优秀设计的原则（后面的几章讲的都是如何为我们在这些阶段中引入的不合理的代码做出补偿）。

Franc

```
class Franc {
    private int amount;

    Franc(int amount) {
        this.amount = amount;
    }

    Franc times(int multiplier) {
        return new Franc(amount * multiplier);
    }

    public boolean equals(Object object) {
        Franc franc = (Franc) object;
        return amount == franc.amount;
    }
}
```

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

因为运行代码的时间实在是太短了，我们甚至可以跳过“让它编译”这一步。

我们现在有大量的重复设计，在编写下一个测试代码之前必须把它们消除掉。我们先着手完成 equals() 函数的一般化。无论如何，我们又可以从计划清单中划去一项了，尽管我们还得再新增三项内容。作为回顾，我们：

- 无法完成一个大的测试，所以我们首先通过一个小的测试先行动起来
- 无所顾忌地通过复制和编辑来写出这个测试程序
- 更糟糕的是，通过将整个模型代码拷贝过来并加以编辑来让测试程序工作
- 自我保证在重复设计消除之前决不回家

6

再谈一切均等

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

Wallace Stegner 的《Crossing to Safety》中有一出描写一个人物工作室的精彩一幕。一切物品都摆放的恰到好处，地面一尘不染，所有一切给人的感觉就是整洁有序。然而，这个人却从未制作出来过任何东西。“他准备工作做了一辈子。先准备好，然后再打扫干净。”（正是因为有感于这本书的结局，我在横穿大西洋的波音 747 客机的商务舱中失态地嚎啕大哭。请务必谨慎阅读。）

我们在第 5 章中避免了陷入这个怪圈。实际上我们让一个新的测试用例运行通过了，但是为使这个测试能够尽快开始工作，我们大量使用了丑陋的拷贝和粘贴代码的方法。现在是收拾烂摊子的时候了。

一种可能的方法是让其中的一个类继承另一个类，我尝试过这种方法，但是它几乎没有节省任何代码。因而我们改用另一种方法，为这两个类寻找一个共同的父类，如图 6-1 所示。（我们也试过这种方法，尽管需要花费一些时间，但结果非常理想。）

如果我们用 Money 类来统一共同的判等函数代码会怎样？让我们一点一点来做：

Money

class Money

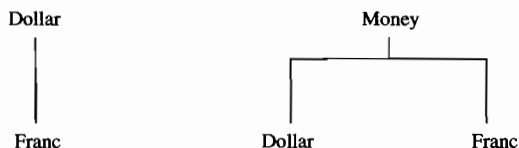


图 6-1 两个类的一个共同父类

所有的测试依然可以运行（并不是说所有的测试都有可能因此而无法运行通过，但无论如何现在是运行测试的好时机）。如果让 `Dollar` 类继承 `Money` 类，那么也不大可能会造成任何测试无法通过。

Dollar

```
class Dollar extends Money {
    private int amount;
}
```

真的不会吗？不会的，所有的测试依然全部运行通过。现在我们可以把 `amount` 实例变量上移至 `Money` 类中：

Money

```
class Money {
    protected int amount;
}
```

Dollar

```
class Dollar extends Money {
}
```

`amount` 的可见性必须由 `private` 改为 `protected` 以便子类仍然能够看见它。（如果我们想动作慢一点的话，第一步可以先在 `Money` 类中声明 `amount` 域，第二步将其从 `Dollar` 类中去掉。因为我现在感觉挺自信的。）

现在我们可以着手准备将 `equals()` 函数的代码上移了。首先我们对这个临时变量的声明进行修改：

Dollar

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

所有的测试依然可以运行通过。现在我们改变强制转化的类型：

Dollar

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

为了便于理解，我们还应当改变这个临时变量的名字：

Dollar

```
public boolean equals(Object object) {  
    Money money= (Money) object;  
    return amount == money.amount;  
}
```

现在我们可以将其从 Dollar 类移至 Money 类:

Money

```
public boolean equals(Object object) {  
    Money money= (Money) object;  
    return amount == money.amount;  
}
```

现在我们需要删除 `Franc.equals()` 函数了。首先我们注意到关于判等的测试中没有法郎对象之间的比较。我们拷贝和粘贴代码这种拙劣的做法现在开始给我们出难题了。在我们改动代码之前，我们要把这里本来一开始就应该有的测试程序写出来。

你将经常在没有足够测试程序的代码中实施测试驱动开发（至少未来十年左右）。当没有足够的测试程序时，你一定会遇到不被测试程序支持的重构。你可能犯了一个重构错误，但是测试程序依然可以运行。这可怎么办？

写出你希望拥有的测试。如果你不这样做，那么迟早会在重构时出现麻烦。然后你就会对重构产生厌烦情绪，因而不愿做太多的重构。接着你的设计就会变糟，接着你会被解雇。你的狗也会离你而去，你将不再注意自己的营养。你的牙齿会出毛病。所以，为了让你的牙齿保持健康，在重构之前你一定要注意补上需要的测试。

幸运的是，这儿的测试程序都非常容易写。我们就把 Dollar 类的测试程序拷贝过来：

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

又多了两行重复！我们也会在后面弥补这一缺陷的。

测试程序已经就位，我们现在可以让 Franc 类继承 Money 类了：

Franc

```
class Franc extends Money {  
    private int amount;  
}
```

为了使用 Money 类中的 amount，删除 Franc 类的 amount 域：

Franc

```
class Franc extends Money {  
}
```

`Franc.equals()` 函数与 `Money.equals()` 函数几乎是一样的。如果我们使它们完全一致，那么我们就可以删除 Franc 类中的实现，而不用改变程序的含意。首先我们改变这个临时变量的声明：



Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount
        && getClass().equals(money.getClass());
}
```

在模型代码中这样使用类有些不大对劲儿。我们希望使用一种在金融领域内有意义的判定条件，而不是 Java 对象域。但是我们现在没有任何类似货币的对象，而且当前情况还不能说有理由引入这样一种对象，所以我们现在不得不这样做。

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

比较法郎对象与美元对象

货币？

现在是我们除去共同的 times() 函数代码的时候了，只有这样做才能实现多币种计算。但是，在我们这样做之前，我们要回顾一下这一章所完成的非常重要的内容，我们在这一章做了以下的事情：

- 着手解决一个困扰我们的难题并将它转化为一个测试程序
- 用一种合理但并不完美的方法（getClass()）使测试程序运行通过
- 除非有更好的动机，否则不要引入更多的设计

8

制造对象

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

比较法郎对象与美元对象

货币？

times() 函数的两种实现惊人地相似：

Franc

```
Franc times(int multiplier) {  
    return new Franc(amount * multiplier);  
}
```

Dollar

```
Dollar times(int multiplier) {  
    return new Dollar(amount * multiplier);  
}
```

为使这两种实现完全一致，第一步，我们可以使它们都返回一个 Money 对象：

Franc

```
Money times(int multiplier) {  
    return new Franc(amount * multiplier);  
}
```

Dollar

```
Money times(int multiplier) {  
    return new Dollar(amount * multiplier);  
}
```

下一步该如何做就没有那么明显了。**Money** 类的两个子类并没有完成足以表明其有存在必要的工作，既然如此，我们索性消除它们。但我们不能一步到位，因为如果这样做的话，就不能很好地讲解测试驱动开发了。

好的。如果直接引用这两个子类的地方没有多少的话，我们距离消除这两个子类就更进一步了。我们可以在 **Money** 类中引入一个返回 **Dollar** 对象的工厂方法（factory method）。就像这样：

```
public void testMultiplication() {  
    Dollar five = Money.dollar(5);  
    assertEquals(new Dollar(10), five.times(2));  
    assertEquals(new Dollar(15), five.times(3));  
}
```

它的实现就是创建并返回一个 **Dollar** 对象：

Money

```
static Dollar dollar(int amount) {  
    return new Dollar(amount);  
}
```

但我们希望不再有引用 **Dollar** 的地方，所以我们需要改变测试程序中的声明：

```
public void testMultiplication() {  
    Money five = Money.dollar(5);  
    assertEquals(new Dollar(10), five.times(2));  
    assertEquals(new Dollar(15), five.times(3));  
}
```

编译器指出：**times()** 函数并不是 **Money** 类中定义的方法。因为我们现在并不打算实现它，于是我们将 **Money** 类改成抽象类（我想我们一开始就应该这样做，不是吗？）并声明 **Money.times()** 函数：

Money

```
abstract class Money  
abstract Money times(int multiplier);
```

现在我们可以改变工厂方法的声明了：

Money

```
static Money dollar(int amount) {  
    return new Dollar(amount);  
}
```

所有的测试都运行通过，所以至少我们没有造成任何破坏。现在我们可以测试程序中的各个地方都使用工厂方法了：

```
public void testMultiplication() {  
    Money five = Money.dollar(5);
```

```

    assertEquals(Money.dollar(10), five.times(2));
    assertEquals(Money.dollar(15), five.times(3));
}

public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(new Franc(5).equals(new Franc(5)));
    assertFalse(new Franc(5).equals(new Franc(6)));
    assertFalse(new Franc(5).equals(Money.dollar(5)));
}

```

现在的状况比刚才要有所好转。任何客户代码都不知道是否存在一个叫 `Dollar` 的子类。通过消除测试程序与子类存在的耦合,我们现在可以自由地改变继承关系而不会对模型代码造成任何影响。

在盲目地改动 `testFrancMultiplication` 之前,我们注意到, `Dollar` 乘法测试程序没有测试过的内容,该测试程序也都没有测试,如果我们删除这个测试,会使我们丧失对某些代码的自信吗?有点儿,所以我们先不管它,但这是个疑点。

```

public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(Money.franc(5).equals(Money.franc(5)));
    assertFalse(Money.franc(5).equals(Money.franc(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}

public void testFrancMultiplication() {
    Money five = Money.franc(5);
    assertEquals(Money.franc(10), five.times(2));
    assertEquals(Money.franc(15), five.times(3));
}

```

这个实现与 `Money.dollar()` 函数的实现非常相似:

Money

```

static Money franc(int amount) {
    return new Franc(amount);
}

```

当瑞士法郎与美元的兑换率为 2:1 的时候, 5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

`Dollar` 类有副作用吗?

钱数必须为整数?

实现 `equals()` 函数

实现 `hashCode()` 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

比较法郎对象与美元对象

货币?

删除 testFrancMultiplication?

下一步我们将消除 times() 的重复设计。现在先回顾一下，我们在这一章做了以下的事情：

- 通过使同一方法（times()）的两个变种的签名（signature）相一致，我们朝着消除重复设计的方向又前进了一步
- 至少将这个方法的一个声明移至共同的父类中
- 通过引入工厂方法，消除测试代码与具体子类存在的耦合
- 注意到当子类被消除后，一些测试将是冗余的，但我们暂时不管它们

9

我们所处的时代

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

比较法郎对象与美元对象

货币？

删除 testFrancMultiplication？

计划清单中有哪些工作可以帮助我们消除那些讨厌而无用的子类呢？货币？如果我们引入货币的概念会怎样呢？

我们想要怎样来实现货币呢？可恶！这可不是初犯了。在惩罚的尺子打过来之前，我得改口了。现在我们想要怎样对货币进行测试呢？好了，这回暂且不罚你了。

我们或许想要用很复杂的对象来表示货币，而用次最轻量级（flyweight）的工厂方法来保证创建的对象不超过我们的需求。但就现在而言，使用字符串来表示就足够了：

```
public void testCurrency() {  
    assertEquals("USD", Money.dollar(1).currency());  
    assertEquals("CHF", Money.franc(1).currency());  
}
```

首先，我们在 Money 类中声明 currency() 操作：

Money

```
abstract String currency();
```

然后，我们分别在两个子类中实现它：

Franc

```
String currency() {  
    return "CHF";  
}
```

Dollar

```
String currency() {  
    return "USD";  
}
```

我们希望两个类中的这个方法采用相同的实现。因而我们可以在一个实例变量中保存 `currency` 并返回这个变量（为了节省时间，我将加快重构的进度。如果太快，请通知我减慢。哦，等一下，我忘了这是一本书了，也许不会快那么多的。）

Franc

```
private String currency;  
Franc(int amount) {  
    this.amount = amount;  
    currency = "CHF";  
}  
String currency() {  
    return currency;  
}
```

对 `Dollar` 来说也如法炮制：

Dollar

```
private String currency;  
Dollar(int amount) {  
    this.amount = amount;  
    currency = "USD";  
}  
String currency() {  
    return currency;  
}
```

现在，我们可以把变量的声明和 `currency()` 的实现上移，因为它们是完全一样的：

Money

```
protected String currency;  
String currency() {  
    return currency;  
}
```

如果我们将字符串常量 “USD” 和 “CHF” 移至静态工厂方法中，那么两个构造函数将全相同，因此我们可以给出一个公共实现。

首先，我们给构造函数增加一个参数：

Franc

```
Franc(int amount, String currency) {  
    this.amount = amount;  
    this.currency = "CHF";  
}
```

这次改动使得两个构造函数的调用者无法工作：

Money

```
static Money franc(int amount) {  
    return new Franc(amount, null);  
}
```

Franc

```
Money times(int multiplier) {  
    return new Franc(amount * multiplier, null);  
}
```

等一等！为什么Franc.times()调用的是构造函数而不是工厂方法呢？我们是现在就将其改为使用工厂方法，还是再等一等？教条一些的做法是先等一会儿，不打破我们手头的工作。我的做法是考虑暂且打断一会儿，但时间不能长，并且我从不再次打断已经被打断的过程（Jim Coplien教我的这个原则）。为使其更加可行，在继续之前，我们将对times()做一次清理：

Franc

```
Money times(int multiplier) {  
    return Money.franc(amount * multiplier);  
}
```

现在工厂方法可以传递“CHF”了：

Money

```
static Money franc(int amount) {  
    return new Franc(amount, "CHF");  
}
```

最后我们可以将参数赋值给实例变量：

Franc

```
Franc(int amount, String currency) {  
    this.amount = amount;  
    this.currency = currency;  
}
```

采取如此小的开发步骤又让我感受到某些人的抵触情绪了。难道我真的就建议你这样做吗？不是的，我只是建议你应该拥有这样工作的能力。我刚才所做的，是在大步工作的过程中犯了一个愚蠢的错误，于是折回头来以小步骤的工作节奏找出原因并将工作重新完成。我现在感觉好一些了，所以现在看看我们能否一口气完成对Dollar同样的修改：

Money

```
static Money dollar(int amount) {  
    return new Dollar(amount, "USD");  
}
```


Dollar

```

Dollar(int amount, String currency) {
    this.amount = amount;
    this.currency = currency;
}
Money times(int multiplier) {
    return Money.dollar(amount * multiplier);
}

```

第一次就成功了，太棒了！

采用测试驱动开发经常要进行这样的调整。感觉小步骤的工作节奏很受限制吗？把开发节奏加快一些。不知道该怎么去做吗？放慢些开发节奏试试。测试驱动开发是一种需要自己调整的过程——稍微这样一些，稍微那样一些。没有恰好适当的节奏，现在没有，将来也不会有。

现在两个构造函数完全一样了，所以我们可以将它们的实现上移：

Money

```

Money(int amount, String currency) {
    this.amount = amount;
    this.currency = currency;
}

```

Franc

```

Franc(int amount, String currency) {
    super(amount, currency);
}

```

Dollar

```

Dollar(int amount, String currency) {
    super(amount, currency);
}

```

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

比较法郎对象与美元对象

货币？

删除 testFrancMultiplication？

现在我们基本上已经做好了将 `times()` 的实现上移并消除子类的准备，但是让我们先来回顾一下我们在这一章做了哪些事情：

- 大的设计构思受阻，所以我们着手解决前面所发现的小问题
- 通过将方法变体上移至调用者（工厂方法）使两个构造函数一致
- 稍微打断一下重构过程，在 `times()` 中使用工厂方法
- 用一大步再做一次同样的重构（对 `Dollar` 做我们曾对 `Franc` 所做的工作）
- 把相同的构造函数上移

10

有趣的 Times 方法

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

比较法郎对象与美元对象

货币？

删除 testFrancMultiplication？

介绍完这一章以后，我们将使用单个类来表示 Money。times() 的两个实现很相近，但还不是完全一样：

Franc

```
Money times(int multiplier) {  
    return Money.franc(amount * multiplier);  
}
```

Dollar

```
Money times(int multiplier) {  
    return Money.dollar(amount * multiplier);  
}
```

没有一种显而易见的可以使得它们完全相同的办法。有的时候你必须以退为进，有点类



`Money@478a43>`”。这条信息对我们没有多大的帮助。我们可以通过定义`toString()`来得到一个更明确的错误信息：

Money

```
public String toString() {  
    return amount + " " + currency;  
}
```

停！有代码，无测试？你能这样做吗？我们当然可以在写`toString()`之前为其编写测试程序，不过：

- 我们是打算在屏幕上查看输出结果的
- 因为`toString()`仅仅是用来输出调试信息的，所以它失败所带来的风险不大
- 我们的测试程序还未运行通过，在这种情况下我们宁愿不写测试

这是一种特殊情况。

现在错误信息为：“`expected:<10 CHF> but was:<10 CHF>`”。比刚才的提示要好一点，但仍然让人搞不明白。我们答案中的数据是正确的，但是类是错误的——应该是`Money`而不是`Franc`。问题出在`equals()`的实现中：

Money

```
public boolean equals(Object object) {  
    Money money = (Money) object;  
    return amount == money.amount  
        && getClass().equals(money.getClass());  
}
```

我们确实应该检查货币是否是相同的，而不是类是否是相同的。

如果我们的测试处于不可运行状态，那么我们宁愿不去写一个测试程序。但我们现在打算改动模型代码，所以不能没有测试程序就修改模型代码。保守一点的做法是把导致程序不能运行的改动恢复原貌，以使测试程序重新运行通过，然后改动`equals()`的测试程序，修复实现代码中的问题，最后再重试原来的改动。

这一次，我们将采用比较保守的办法。（有时，当测试处于不可运行状态时我也会逆规划行事，编写测试程序，当然这么做时必须全神贯注。）

Franc

```
Money times(int multiplier) {  
    return new Franc(amount * multiplier, currency);  
}
```

现在我们的测试程序又可以运行通过了。前面那种出错情况的原因是：尽管我们希望`Franc(10, “CHF”)`和`Money(10, “CHF”)`相等，但事实上它们并不相等。我们就把这个作为测试：

```
public void testDifferentClassEquality() {  
    assertTrue(new Money(10, "CHF").equals(new Franc(10, "CHF")));  
}
```

正如我们所预想的一样，运行失败了。`equals()`应该比较的是货币，而不是类：

Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount
        && currency().equals(money.currency());
}
```

现在我们可以从 `Franc.times()` 返回一个 `Money` 对象，测试仍然通过：

Franc

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

同样的方法是否也适用于 `Dollar.times()` 呢？

Dollar

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

是的！现在两个实现完全相同了，因此我们可以将它们上移。

Money

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

比较法郎对象与美元对象

货币？

删除 testFrancMultiplication？

乘法已经没问题了，现在可以执行消除子类的任务了。作为回顾，我们在这一章做了以下事情：

- 为使两个 `times()` 操作一致，首先通过内联方式调用需要调用的方法，然后将常量替

换成变量

- 无须事先编写测试程序，直接编写 `toString()` 函数以便调试
- 试着做一下改动（返回 `Money` 而不是 `Franc`），让测试告诉我们改动是否可行
- 恢复试验时所做的修改，编写另外一个测试程序。测试程序运行通过了，试验代码就能运行通过

11

万恶之源

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

4 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗？

钱数必须为整数？

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

比较法郎对象与美元对象

货币？

删除 testFrancMultiplication？

两个子类 Dollar 和 Franc 只剩下构造函数了。一个只有构造函数的子类并不能说明其有存在的必要，所以我们希望删除这两个子类。

我们可以在不改变代码含意的情况下，将对子类的引用修改为对父类的引用。首先对 Franc 类进行操作：

```
Franc
static Money franc(int amount) {
    return new Money(amount, "CHF");
}
```

然后是 Dollar 类：

```
Dollar
static Money dollar(int amount) {
    return new Money(amount, "USD");
}
```


现在已经没有对Dollar的引用了,我们可以删除它。另一方面,仍然有一个地方存在对Franc的引用,就在我们刚写的测试程序中:

```
public void testDifferentClassEquality() {
    assertTrue(new Money(10, "CHF").equals(new Franc(10, "CHF")));
}
```

别的地方是否已经涵盖了这个有关判等的测试呢,让我们来看看其他的判等测试:

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(Money.franc(5).equals(Money.franc(5)));
    assertFalse(Money.franc(5).equals(Money.franc(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
```

看起来关于判等的测试已经足够周全了,事实上是过于周全。我们可以删除第三和第四个断言,因为它们与第一和第二个断言完成的工作是一样的:

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
```

当瑞士法郎与美元的兑换率为 2:1 的时候, 5 美元+10 瑞士法郎=10 美元

5 美元*2=10 美元

将“amount”定义为私有

Dollar 类有副作用吗?

钱数必须为整数?

实现 equals() 函数

实现 hashCode() 函数

与空对象判等

与非同类对象判等

5 瑞士法郎*2=10 瑞士法郎

美元 Dollar/瑞士法郎 Franc 之间的重复设计

普通判等

普通相乘

比较法郎对象与美元对象

货币?

删除 testFrancMultiplication?

我们所写的测试迫使我们比较货币而不是比较类,这只有在存在多个类的情况下才有意义。因为我们想删除 Franc 类,所以保证系统在存在 Franc 类的情况下可以工作的测试已经是多余的了,帮不上什么忙。测试 testDifferentClassEquality() 不存在了, Franc 类也随之而去。

类似地，美元乘法和瑞士法郎乘法分别都有各自的测试，我们可以看到，在 `Currency` 基础之上，现在它们的代码在逻辑上已经没有什么差别（如果存在两个类的话会有差别）。我们可以删除 `testFrancMultiplication()`，而不必担心对系统的功能有任何的影响。

整合成单个类的工作已经完成，我们可以着手处理加法问题了。首先，作为回顾，我们在这一章做了以下事情：

- 在掏空了子类的功能后将其删除
- 消除了对旧的代码结构有意义、而对新的代码结构显得多余的一些测试

12

加法，最后的部分

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

新的一天开始了，我们的计划清单已经变得有些杂乱，因此我们要把悬而未决的事项抄到一张新的清单上。（我喜欢用笔把要做的事项誊写到一张新的清单上，但是如果这些清单中有许多琐碎的项目，我则更倾向于多操点儿心而不是重抄一遍。正因为我很懒，所以这样做才不至于疏忽大意而使这些琐碎的东西越积越多。当然，请量力而行。）

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

我不知该加法的整个过程应当怎样描述，所以我从一个简单的例子开始：5 美元+5 美元=10 美元。

```
public void testSimpleAddition() {  
    Money sum= Money.dollar(5).plus(Money.dollar(5));  
    assertEquals(Money.dollar(10), sum);  
}
```

我们本来可以通过伪实现来让代码返回“Money.dollar(10)”，但是加法实现显而易见，因此我们先来尝试一下这种显然的做法：

```
Money  
Money plus(Money addend) {  
    return new Money(amount + addend.amount, currency);  
}
```

（一般而言，我将加快我的实现过程，这样一方面能够节省纸张，另一方面还能保持你的兴趣。在如何设计不够明了的地方，我将用伪实现和重构，我希望通过这些让你看到测试驱动开发是怎样帮你控制开发步伐的大小的。）

前面刚提到我要加快速度，而我又不得不立即慢下来——不是慢在运行测试上，而是慢在编写测试本身。有时有些测试需要仔细考虑，我们怎样表示多币种运算（multi-currency arithmetic）呢？这就是需要认真考虑的情况之一。

最困难的设计要求是系统中的大部分代码都不能感觉到是在（潜在的）与多种货币打交道。

一种可能的策略是立即把所有的币值转化成某种参照货币(我想让你猜猜不修边幅的美国程序员一般会选哪种货币作为参照货币)。不过这种解决方案使得汇率不易改变。

然而，我们更喜欢用另一种解决方法。这种方法允许我们方便地表示多种汇率，而且还能使许多类算术表达式看起来更像算术表达式。

对象可以帮上忙。当你所拥有的对象并没有按照你的要求行事时，那么就创建另外一种具有相同外部协议(external protocol)——一个替代者，但具体实现不同的对象。

这可能听起来似乎有点像魔术，我们怎么会想到在这里创建一个替代者呢？不瞒你说，设计灵感的闪现没有什么套路可循。Ward Cunningham 十年前就想出这个“把戏”了，直到目前为止我还没看见过谁能够独立地提出一个类似的设计，可见它是一个多么微妙的“把戏”。测试驱动开发并不能保证让我们在遇到难题时爆发出瞬间的灵感。然而，可以增强自信的测试和经过认真重构的代码不仅为我们准备了足够的洞察力，同时也教会了我们怎样运用这种洞察力。

我们的解决方案是创建一种行为像是Money类的对象，但代表了两种Money对象的和。我曾使用过多种不同的比喻来解释这种思想。其中一个比喻就是把和当作一个钱包，你可以把几种具有不同币值和币种的钞票放在同一个钱包中。

另一个比喻是表达式，诸如“(2+3)*5”，或者这个例子中的“(2美元+3瑞士法郎)*5”。Money对象是表达式中无法再继续细分的元素。通过操作形成表达式，其中之一就是Money对象的和Sum。一旦操作(例如把某种组合投资的金额累加起来)完成，在给定一组汇率后，运算的结果就能够化归为某种单一的货币。

把这个比喻运用在我们的测试中，我们就知道结果应该这样写：

```
public void testSimpleAddition() {
    ...
    assertEquals(Money.dollar(10), reduced);
}
```

把汇率代入到表达式中就得到化归后的表达式。那么在现实世界中由谁来提供汇率呢？银行。我们应该这样写：

```
public void testSimpleAddition() {
    ...
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

(混合使用银行和表达式的隐喻真有点怪异。我们先把整个故事描述清楚，而后再来看看怎样操作这些数值)。

其实在这里我们做了一个重要的决定，我们本来可以简单地写成“...reduce= sum.reduce(“USD”,bank)”，为什么要引入银行来担负这项工作呢？一个答案是，“这是第一个跳入我脑海中的东西”，这种解释没有说明多少问题。为什么我首先想到的是bank类来做这项工作而不是表达式呢？这是因为在此刻我考虑到了以下几个方面：

- 表达式看起来是我们工作的中心,我试着保持以它们为中心,尽可能不理睬其他东西,这样它们能够尽可能长时间地保持灵活性(而且还容易测试、重用和理解)。
- 我能想像表达式中涉及许多操作,如果我们把每个操作都加入到表达式中,它将会无限制地增长。

似乎没有足够的理由老是把规模开销看得很重要,但对于我来说,从这个角度出发的理由已经很充分了。若迹象表明银行无需参与化归任务,那么我完全愿意将化归的责任转移给表达式来完成。

银行在我们的简单例子中实际上不需做任何事,只要有一个银行对象就可以了:

```
public void testSimpleAddition() {
    ...
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

两个 **Money** 对象的和应该是表达式:

```
public void testSimpleAddition() {
    ...
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

至少我们确切地知道了怎样获得 5 美元:

```
public void testSimpleAddition() {
    Money five= Money.dollar(5);
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

怎样让代码编译呢? 我们需要一个表达式接口(我们也可以用类,但接口是轻量级的):

Expression

```
interface Expression
```

Money.plus()需要返回一个表达式 (**Expression**):

Money

```
Expression plus(Money addend) {
    return new Money(amount + addend.amount, currency);
}
```

这意味着 **Money** 类要实现 **Expression** (这很容易,因为 **Expression** 现在还没有任何操作):

Money

```
class Money implements Expression
```

我们需要一个空的Bank类：

Bank

```
class Bank
```

这里引出了reduce()：

Bank

```
Money reduce(Expression source, String to) {  
    return null;  
}
```

现在，我们来编译它。然而不幸的是，编译失败了。好的！有进展！虽然失败了，但我们能轻易地对它进行伪实现：

Bank

```
Money reduce(Expression source, String to) {  
    return Money.dollar(10);  
}
```

我们回到了可运行状态，并且准备重构。首先，回顾一下，我们在这一章做了以下的事情：

- 把一个大的测试削减成一个小一些的测试，仍然算是在前进（从“5 美元+10 瑞士法郎”到“5 美元+5 美元”）
- 认真思考可能与我们的计算有关的比喻
- 基于我们新的比喻，重写了前面的测试
- 让测试尽快通过编译
- 让测试运行
- 带着一丝惶恐，期待着为写出真实的实现而必须进行的重构

13

完成预期目标

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

在消除所有的重复设计之前，我们关于 5 美元+5 美元=10 美元的测试就不能算是真正完成。我们的代码之间没有重复，但我们的数据之间有重复，即伪实现中的 10 美元：

Bank

```
Money reduce(Expression source, String to) {  
    return Money.dollar(10);  
}
```

和测试中的“5 美元+5 美元”是完全一样的。

```
public void testSimpleAddition() {  
    Money five= Money.dollar(5);  
    Expression sum= five.plus(five);  
    Bank bank= new Bank();  
    Money reduced= bank.reduce(sum, "USD");  
    assertEquals(Money.dollar(10), reduced);  
}
```

以前在有伪实现时，我们知道如何倒着来完成真正的实现，只需简单地用变量代替常量就可以了。不过这次怎样倒着来完成这一工作，对我来说却不再是那么显而易见了。因此尽管有些冒失，但我们仍将继续朝前走。

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

首先，Money.plus()需要返回的是一个真正的表达式（Expression）对象——Sum，而不仅仅是一个 Money 对象。（稍后我们可能会对将两种相同货币相加的特殊案例进行优化，但那是以后的事。）

两个 Money 对象的和应该是一个 Sum 对象：


```
public void testPlusReturnsSum() {
    Money five= Money.dollar(5);
    Expression result= five.plus(five);
    Sum sum= (Sum) result;
    assertEquals(five, sum.augend);
    assertEquals(five, sum.addend);
}
```

上面的测试我并不想要一直使用。它过多涉及了操作的具体实现，而不是其外部可见行为。然而，如果我们能够使它运行通过，那么就有望向我们的目标迈进一步。为了使它能够通过编译，我们需要一个带有两个域：被加数（augend）和加数（addend）的 Sum 类。

Sum

```
class Sum {
    Money augend;
    Money addend;
}
```

运行测试会产生一个 ClassCastException，因为 Money.plus() 返回的是 Money 对象，而不是 Sum 对象。

Money

```
Expression plus(Money addend) {
    return new Sum(this, addend);
}
```

Sum 类需要一个构造函数：

Sum

```
Sum(Money augend, Money addend) {
}
```

同时 Sum 还必须是一种 Expression：

Sum

```
class Sum implements Expression
```

现在系统又能够编译通过了，但测试仍然运行失败。这次是因为 Sum 构造函数时没有设置域。（我们可以初始化域来进行伪实现，但是我说过我要加快步伐了。）

Sum

```
Sum(Money augend, Money addend) {
    this.augend= augend;
    this.addend= addend;
}
```

现在可以向 Bank.reduce() 传递 Sum 对象了。若 Sum 中的货币全都一样，而且目标货币也一样，则结果应该是一个 Money 对象，它的数目是各数目的总和：

```
public void testReduceSum() {
    Expression sum= new Sum(Money.dollar(3), Money.dollar(4));
    Bank bank= new Bank();
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(7), result);
}
```


我仔细选择会使已存在的测试不再适用的参数。当我要对一个 `Sum` 对象进行化归时，结果（在这些简化的环境下）应该是 `Money` 对象，它的数目是两种 `Money` 对象的数目之和，并且它的货币是我们希望化归成的那种货币：

Bank

```
Money reduce(Expression source, String to) {
    Sum sum= (Sum) source;
    int amount= sum.augend.amount + sum.addend.amount;
    return new Money(amount, to);
}
```

这立刻使两方面变得讨厌起来：

- 强制类型转换。这段代码应该对任何表达式都适用。
- 公共域以及对它的两级引用。

这两个问题很容易解决。首先，我们可以把方法的主体移至 `Sum` 类且去掉某些可见域。我们“确信”将来会用到 `Bank` 作参数，但这是纯粹的、简单的重构，所以我们不考虑这个。（实际上，刚刚我安置这个对象的时候也因为我“知道”我会需要它。）

Bank

```
Money reduce(Expression source, String to) {
    Sum sum= (Sum) source;
    return Sum.reduce(to);
}
```

Sum

```
public Money reduce(String to) {
    int amount= augend.amount + addend.amount;
    return new Money(amount, to);
}
```

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 `Money` 对象

`Bank.reduce(Money)`

（这就提出一个问题：当参数是 `Money` 时，我们怎样去实现，呃……测试 `Bank.reduce()`。）

让我们来编写这个测试，因为现有测试都已运行通过，而且上面的代码也没有什么明显需要修改的：

```
public void testReduceMoney() {
    Bank bank= new Bank();
    Money result= bank.reduce(Money.dollar(1), "USD");
    assertEquals(Money.dollar(1), result);
}

Bank
Money reduce(Expression source, String to) {
    if (source instanceof Money) return (Money) source;
    Sum sum= (Sum) source;
    return sum.reduce(to);
}
```

尽管这样实现很难看，但是测试运行通过，我们现在可以进行重构了。无论何时当我们需要显式地判定是哪种类才能进行下一步工作时，都要使用多态来代替。由于 Sum 实现了 reduce(String)，所以如果 Money 也实现了它，那么我们就可以把它添加到 Expression 接口中。

Bank

```
Money reduce(Expression source, String to) {
    if (source instanceof Money)
        return (Money) source.reduce(to);
    Sum sum= (Sum) source;
    return sum.reduce(to);
}
```

Money

```
public Money reduce(String to) {
    return this;
}
```

如果我们将 reduce(String) 添加到 Expression 接口中，

Expression

```
Money reduce(String to);
```

那么就能够消除所有烦人的强制类型转换和类判定：

Bank

```
Money reduce(Expression source, String to) {
    return source.reduce(to);
}
```

我对 Expression 和 Bank 中的方法名相同但却有着不同的参数类型这个事实并不十分满意。在 Java 中我从未找到一种令人满意的、解决这种问题的一般化方案。在支持键值参数 (keyword parameter) 的语言中，通过语言的语法，Bank.reduce(Expression,String) 与 Expression.reduce(String) 之间的差异可以很好地加以传达。而如果采用位置参数 (positional parameter) 参数，那么代码难以从自身来传达两者间的差异。

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

Bank.reduce(Money)

带换算的 Reduce Money

Reduce(Bank,String)

接下来，我们将真正地把一种货币兑换成另外一种货币。首先，让我们回顾一下我们在本章所做的工作：

- 因为重复设计没有完全消除，所以没有把一个测试标记为完成

- 为了知道能够如何实现，继续往前走而不是往回走
- 写一个测试，以迫使创建一个我们稍后要用到的类（Sum）
- 开始更快地实现（Sum 的构造函数）
- 在一个地方采用强制类型转换来实现代码，然后一旦测试运行通过，就把代码移到它本属于的地方
- 为了消除显式的类判定而引入多态（polymorphism）

14

变 化

当瑞士法郎与美元的兑换率为 2:1 的时候, 5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

Bank.reduce(Money)

带换算的 Reduce Money

Reduce(Bank,String)

变化值得拥抱（尤其是当你有本名为《拥抱变化》的书时）。不过，我们这里所考虑的变化形式要简单得多——我们有两法郎，但我们想得到一美元。这话听起来就是一个测试案例：

```
public void testReduceMoneyDifferentCurrency() {  
    Bank bank= new Bank();  
    bank.addRate("CHF", "USD", 2);  
    Money result= bank.reduce(Money.franc(2), "USD");  
    assertEquals(Money.dollar(1), result);  
}
```

当我把法郎兑换成美元时，要用 2 除。（我们仍有意地忽略所有那些烦人的数字问题。）我们使用一段拙劣的代码来使测试运行通过：

Money

```
public Money reduce(String to) {  
    int rate = (currency.equals("CHF") && to.equals("USD"))  
        ? 2  
        : 1;  
    return new Money(amount / rate, to);  
}
```

Money 类现在突然间就知道兑换率这个东西了。真是的，其实 Bank 类才应该是我们惟一关注兑换率的地方。我们将不得不把 Bank 作为一个参数传递给 Expression.reduce()。（看到了吗？我们就知道会用到它的，而事实证明我们是对的。用《The Princess Bride》中祖父的话来讲就是“你非常聪明……”）首先是调用者的代码：

Bank

```
Money reduce(Expression source, String to) {
```

```
    return source.reduce(this, to);
}
```

然后是实现者：

Expression

```
Money reduce(Bank bank, String to);
```

Sum

```
public Money reduce(Bank bank, String to) {
    int amount= augend.amount + addend.amount;
    return new Money(amount, to);
}
```

Money

```
public Money reduce(Bank bank, String to) {
    int rate = (currency.equals("CHF") && to.equals("USD"))
        ? 2
        : 1;
    return new Money(amount / rate, to);
}
```

方法必须是公共的，之所以如此，是因为接口中的方法必须是公共的（我确信这是出于一些很充分的理由）。

现在我们能计算 **Bank** 中的兑换率了：

Bank

```
int rate(String from, String to) {
    return (from.equals("CHF") && to.equals("USD"))
        ? 2
        : 1;
}
```

并且向 **bank** 询问正确的兑换率：

Money

```
public Money reduce(Bank bank, String to) {
    int rate = bank.rate(currency, to);
    return new Money(amount / rate, to);
}
```

恼人的 2 仍出现在测试和代码中。为了除掉它，我们需要在 **Bank** 中保存一张兑换率表并且在需要时查找出相应的兑换率。我们可以用一张散列表来将每对货币映射为兑换率。我们可以使用一个包含两种货币的二元数组作为键值吗？`Array.equals()` 会检查这些元素是否相等吗？

```
public void testArrayEquals() {
    assertEquals(new Object[] { "abc" }, new Object[] { "abc" });
}
```

不行。测试失败了，因此我不得不创建一个真实的对象作为键值。

Pair

```
private class Pair {
```

```
private String from;
private String to;

Pair(String from, String to) {
    this.from= from;
    this.to= to;
}
}
```

因为要把 **Pair** 用作键值，所以我们不得不实现 `equals()` 和 `hashCode()`。我们不打算为这些内容编写测试程序，这是因为我们是在重构的时候写的这些代码。如果我们已经品尝到了这些重构所带来的好处，而且所有的测试都运行通过的话，那么我们希望这些代码已在实践中成功运用了。如果我是在同某个还不十分清楚我编写这段代码的意图的人一起编程，或者是代码逻辑略显一丝复杂的话，我就会为其编写单独的测试程序。

Pair

```
public boolean equals(Object object) {
    Pair pair= (Pair) object;
    return from.equals(pair.from) && to.equals(pair.to);
}

public int hashCode() {
    return 0;
}
```

这里的 0 是一个糟糕的散列值，但它有容易实现的优点，并且它可以让我们很快通过测试。这时的货币查找就像是线性查找。以后，当有多种货币时，我们可以根据实际的应用数据采取更彻底的解决问题的方法。

我们需要找个地方来存储兑换率：

Bank

```
private Hashtable rates= new Hashtable();
```

并在需要的时候设置兑换率：

Bank

```
void addRate(String from, String to, int rate) {
    rates.put(new Pair(from, to), new Integer(rate));
}
```

在有请求时查找兑换率：

Bank

```
int rate(String from, String to) {
    Integer rate= (Integer) rates.get(new Pair(from, to));
    return rate.intValue();
}
```

等一下！测试没有运行通过。哪儿出乱子了？经过稍微仔细的检查发现，如果我们询问从 USD 到 USD 的兑换率，期望值是 1。因为这是个意外，所以让我们写一个测试表述出来：

```
public void testIdentityRate() {  
    assertEquals(1, new Bank().rate("USD", "USD"));  
}
```

现在还有三个错误，但是有望经过下面的修改全部得以更正：

Bank

```
int rate(String from, String to) {  
    if (from.equals(to)) return 1;  
    Integer rate= (Integer) rates.get(new Pair(from, to));  
    return rate.intValue();  
}
```

成功了！

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

Bank.reduce(Money)

带换算的 **Reduce Money**

Reduce(Bank,String)

下一步，我们将会实现最后一个大的测试，5 美元+10 瑞士法郎。不知不觉中，我们在本章已经运用了几种重要的技术。现在，我们回顾一下，在这一章我们做了哪些事情：

- 用了很短的时间，增加了一个希望会用到的参数
- 分离代码与测试程序间的数据重复
- 编写了一个测试程序（testArrayEquals）来核实一个我们有关 Java 操作的推断
- 引入了一个私有的帮助类（helper class），但并未专门为其提供测试程序
- 在重构中犯了一个错误，我们选择使用简陋的解决办法尽快处理通过，并通过编写测试程序来单独考虑这一问题

15

混合货币

当瑞士法郎与美元的兑换率为 2:1 的时候, 5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

Bank.reduce(Money)

带换算的 Reduce Money

Reduce(Bank,String)

现在我们终于可以增加这个主要的测试了, 我们前面所做的一切都是因它而起, 5 美元+10 瑞士法郎:

```
public void testMixedAddition() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

这才是我们想写的代码。不幸的是存在大量的编译错误。在我们将 **Money** 类一般化为 **Expression** 类时, 我们四处遗留下了许多还没有完成或是没有正确完成的内容。对此我一直很担心, 然而我也不想让你为此揪心。可是, 现在是该操这个心的时候了。

我们无法让上面的测试很快通过编译。我们先修改一个地方, 而它又会引起其他连锁的改动。我们面前有两条路。我们可以写一个比较具体的测试, 使它很快地运行起来, 然后进行一般化, 或者就让我们相信编译器, 让它来指导我们不犯错误。我听你的——还是让我们慢慢来吧 (在实践中, 我可能一次仅修改连锁变动中的一个)。

```
public void testMixedAddition() {
    Money fiveBucks= Money.dollar(5);
    Money tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```


测试没有通过。我们得到的是 15 个 USD，而不是 10 个 USD。好像 `Sum.reduce()` 压根儿就没有对传进来的参数进行化归。它确实没有：

Sum

```
public Money reduce(Bank bank, String to) {  
    int amount= augend.amount + addend.amount;  
    return new Money(amount, to);  
}
```

如果我们对两个参数都进行化归，那么测试应该通过：

Sum

```
public Money reduce(Bank bank, String to) {  
    int amount= augend.reduce(bank, to).amount  
        + addend.reduce(bank, to).amount;  
    return new Money(amount, to);  
}
```

确实如此。现在我们可以开始一点点地处理掉本应该是 `Expression` 的 `Money`。为了避免连锁反应，我们将从外围开始，然后倒着一直推进到测试用例。例如，`augend` 和 `addend` 现在可以是 `Expression` 了：

Sum

```
Expression augend;  
Expression addend;
```

`Sum` 构造函数的参数也可以是 `Expression`：

Sum

```
Sum(Expression augend, Expression addend) {  
    this.augend= augend;  
    this.addend= addend;  
}
```

（`Sum` 现在开始让我想起 `Composite` 了，但我还不想现在就一般化到那种程度。到了我们想要 `Sum` 接受的参数不是两个的时候，我们就需要对它进行某种转换了。）对 `Sum` 来说要修改的就这么多——那么对 `Money` 又做怎样的处理呢？

`plus()` 的参数可以是 `Expression`：

Money

```
Expression plus(Expression addend) {  
    return new Sum(this, addend);  
}
```

`times()` 可以返回一个 `Expression`：

Money

```
Expression times(int multiplier) {  
    return new Money(amount * multiplier, currency);  
}
```

这意味着 `Expression` 应当包含 `plus()` 及 `times()` 两种操作。对 `Money` 所做的修改就这么多。我们现在可以改变测试用例中 `plus()` 的参数了：

```
public void testMixedAddition() {
    Money fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

当把 fiveBucks 改成 Expression 类型时，我们必须做几处改动。值得庆幸的是，我们现在只需集中注意力解决编译器指出的错误。首先，我们先进行修改：

```
public void testMixedAddition() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

编译器礼貌地告诉我们，Expression 中没有定义 plus()。于是我们就来定义它：

Expression

```
Expression plus(Expression addend);
```

接着我们必须将其增加到 Money 和 Sum 中。Money?是的，在 Money 类中它必须是公共的：

Money

```
public Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

我们只是在 Sum 中提供了存根（stub）实现，并把这一问题加入到我们的计划清单中。

Sum

```
public Expression plus(Expression addend) {
    return null;
}
```

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

~~Bank.reduce(Money)~~

~~带换算的 Reduce Money~~

~~Reduce(Bank,String)~~

Sum.plus

Expression.times

现在程序编译通过，所有的测试都运行通过。

我们准备结束将 Money 一般化为 Expression 的工作。但首先，让我们回顾一下我们在这

一章做了哪些事情：

- 编写我们想要的测试，然后暂且通过简陋的方法让它很快运行通过
- 先从“叶子”再到“根”（这里是指测试用例）来实施一般化（使用更抽象一些的声明）
- 根据编译器的提示进行修改（Expression fiveBucks），这一改动引起一系列的连锁修改（把 plus() 增加到 Expression 中）

16

抽象，最后的工作

当瑞士法郎与美元的兑换率为 2:1 的时候，5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

Bank.reduce(Money)

带换算的 Reduce Money-

Reduce(Bank,String)

Sum.plus

Expression.times

为了实现 `Expression.plus`，我们需要实现 `Sum.plus()`，接着需要实现 `Expression.times()`，继而整个例子就完成了。下面是 `Sum.plus()` 的测试：

```
public void testSumPlusMoney() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Expression sum= new Sum(fiveBucks, tenFrancs).plus(fiveBucks);
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(15), result);
}
```

我们本可以通过 `fiveBucks` 与 `tenFrancs` 相加来创建一个 `Sum` 对象，然而采用上面这种显式的创建对象的形式表述起来更为直接一些。我们写这些测试不仅使我们的编程经历更加有趣和有意义，还能让后代像欣赏罗赛塔石一样欣赏我们的才华。哦，还是想想读者们吧。

这个案例中的测试要比代码长。这里的代码和 `Money` 中的代码是一样的。（我很久以前不是听谁提到过要用抽象类的吗？）

Sum

```
public Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

当瑞士法郎与美元的兑换率为 2:1 的时候, 5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

Bank.reduce(Money)

带换算的 Reduce Money

Reduce(Bank,String)

Sum.plus

Expression.times

当用测试驱动开发时, 你最终可能大致要编写出与模型代码数量相同的测试代码。要想使 TDD 符合经济利益, 你要么得每天编写是以前两倍数量的代码, 要么只花费一半的代码量来实现同样的功能。你得思量 and 观察一下 TDD 对你的实践过程有什么样的影响。不过一定要把调试、模块集成和解释问题的时间考虑进来。

当瑞士法郎与美元的兑换率为 2:1 的时候, 5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

Bank.reduce(Money)

带换算的 Reduce Money

Reduce(Bank,String)

Sum.plus

Expression.times

如果能够让 Sum.times() 工作, 声明 Expression.times() 就是简单的一步了。测试代码如下:

```
public void testSumTimes() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Expression sum= new Sum(fiveBucks, tenFrancs).times(2);
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(20), result);
}
```

测试又一次比代码长。(你们当中 JUnit 的呆子们会知道怎样纠正这一局面的——其他的人还是阅读这些成型的东西吧。)

Sum

```
Expression times(int multiplier) {
    return new Sum(augend.times(multiplier), addend.times(multiplier));
}
```

因为我们在上一章中把 augend 和 addend 抽象为 Expression 类型, 所以现在我们需要在 Expression 中声明 times() 以使代码能够通过编译:

Expression

```
Expression times(int multiplier);
```

这样就迫使我们上移 `Money.times()` 及 `Sum.times()` 出现的位置:

Sum

```
public Expression times(int multiplier) {
    return new Sum(augend.times(multiplier), addend.times(multiplier));
}
```

Money

```
public Expression times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

当瑞士法郎与美元的兑换率为 2:1 的时候, 5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

Bank.reduce(Money)

带换算的 Reduce Money

Reduce(Bank,String)

Sum.plus

Expression.times

运行通过。

惟一需要完善的地方就是, 当计算 5 美元+5 美元时, 我们要对返回 Money 对象进行试验。

测试如下:

```
public void testPlusSameCurrencyReturnsMoney() {
    Expression sum= Money.dollar(1).plus(Money.dollar(1));
    assertTrue(sum instanceof Money);
}
```

这个测试看起来不是很顺眼, 因为它测试的是内部的实现机制而不是对象的外部可见行为。然而, 这样能够驱使我们去执行所需要的改动, 况且它毕竟只是一个试验而已。下面就是要让它运行而必须修改的代码:

Money

```
public Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

当瑞士法郎与美元的兑换率为 2:1 的时候, 5 美元+10 瑞士法郎=10 美元

5 美元+5 美元=10 美元

从 5 美元+5 美元返回一个 Money 对象

Bank.reduce(Money)

带换算的 Reduce Money

Reduce(Bank,String)

Sum.plus

Expression.times

当且仅当参数是一个 **Money** 对象时，没有明显的、简洁的途径（无论如何对我别无他法，我确信你能想出某些途径）来检查到底是哪种货币。试验失败了，我们删除了测试（怎么说我们也不大愿意这样），不管它了。

回顾一下本章我们所做的工作：

- 编写了一个考虑未来读者需求的测试
- 建议做一个试验，将测试驱动开发与你的现行编程方式作比较
- 声明变更又一次在系统中触发了一系列连锁修改，与前面一样，根据编译器的指示进行修改
- 尝试了一次简短的试验，然后当它无法工作时放弃掉

17

资金实例回顾

让我们回过头再去看看 **Money** 的例子，看看我们所使用的过程方法及其结果。我们将会讲解：

- 下一步做什么？
- 比喻——比喻对设计结构产生的神奇的效果
- JUnit 的用法——什么时候运行测试和怎样使用 JUnit？
- 代码统计——最终代码的数字抽象
- 过程——我们说过程为测试无法通过/测试运行通过/重构，但每一步投入多大的工作量呢？
- 测试质量——根据常规测试衡量标准，怎样组织测试驱动开发的测试？

下一步是什么？

代码完成了吗？没有。**Sum.plus()**与 **Money.plus()**之间还存在让人讨厌的重复设计。若我们当初把 **Expression** 作为一个类而不是一个接口（一般不这样教你，因为更多的时候类会成为接口）的话，那么那些共同代码就会有一个很自然的去处。

我并不相信有“完成”这一说。测试驱动开发可以被用做一种力争完美的方法，但是其最有效的用途不在此。如果你手头有一个大型的系统，那么你经常改动的部分应该是绝对坚实的，因此你才能自信地进行每日的修改。当你渐渐需要触及外围系统，触及那些不经常改动的部分时，测试代码便可能疤痕累累，而相关设计也更显拙劣，但这并不妨碍你对修改结果充满自信。

当做完所有这些显而易见的任务后，我喜欢运行一下代码评测系统，例如 **Smalltalk** 的 **SmallLint**。评测系统所提出的许多问题都是我已经获知的或不敢苟同的。自动评测系统不会忘记任何问题，因此如果我没有把作废的实现删除，我无须强调这一点，自动评测系统就会指出的。

另一个“下一步是什么”的问题是“我需要什么别的测试吗？”有时你认为测试不会运行通过，但却通过了，这时你就要去找原因。有时应该无法运行的测试确实没有运行通过，你可以将其作为一个已知限制或以后要完成的工作记录下来。

最后，当计划清单上所列的所有任务都已完成时，就是回顾设计的好时候了。词语和概念是否相符？依照当前的测试是否存在难以祛除的重复设计？（久拖不去的重复设计往往是设计存在潜在缺陷的征兆。）

比 喻

在资金实例的编码过程中，最令我惊奇的是最终结果大相径庭。根据我的记忆，我至少已经为制作部门编写过三遍 `Money` 了。在别的书中将其作为例子引用，也有六次之多。我在讲台上现场（放松点，并不像听起来那么刺激）编写它也有十五次了。为了准备写这本书（我把第一部分撕了，根据以前的审校重写了它），我又编了三到四次。接着，正在我写的时候，我想起使用表达式（`Expression`）作为比喻，而设计与以前完全不同了。

我真的没有料到比喻的力量竟然如此强大。比喻仅仅是名字的源泉，难道不是吗？当然。

Ward Cunningham 把“可能是由不同货币混在一起的一堆钱”比作向量，如同数学向量一样，只不过它的系数是货币而不是 x^2 。我有一段时间用的是“`MoneySum`”，然后是“`MoneyBag`”（这很好并且很自然），最终是“`Wallet`”（就大部分人们而言，这样更通俗）。所有这些比喻都暗示了这样一组 `Money` 之间的关系是平直的、没有层次的（flat）。例如“`2 USD+5 CHF+3 USD`”应当是“`5 USD+5 CHF`”。同种货币的两个值将被合并。

关于表达式的比喻把我从一堆令人心烦的同种货币合并问题中解脱出来。代码比以前更为简洁、清晰。我担心 `Expression` 的性能。我很乐意在了解了一些应用过程中的统计数据之后，再开始对其进行优化。

如果在我编写了 20 遍之后一切又都需要重写怎么办？我每次都需要这样来找到灵感吗？有没有在编程的时候就需要时刻提醒自己的指导原则？这样我就能一次榨出需要三遍编程才能领悟的内涵。一次搞定？

JUnit 的用法

我在编写 `Money` 的代码时，让 JUnit 保留了日志。我足足点击了 JUnit 的测试运行按钮 125 次。因为我一边写作一边编程，所以各次运行的时间间隔不具代表性。但是当我专注于编程的时候，大约每分钟就运行一次测试。也只有在那段时间里，我才在成功或失败之中不断地得到意外，同时那也是一次仓促的重构。

图 17-1 是一个测试运行时间间隔的直方图。那些数值较大的时间间隔极有可能是我在花时间在写作。

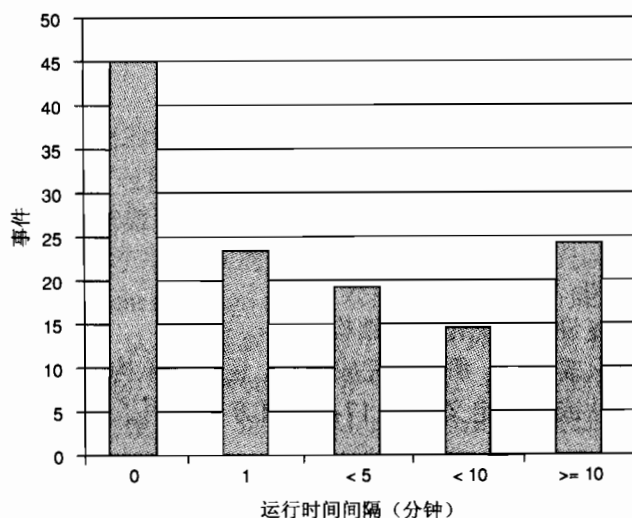


图 17-1 测试运行时间间隔的直方图

代码统计

表 17-1 给出了一些代码的统计数据。

表 17-1 代码统计

	功能代码	测试代码
类	5	1
函数 ^①	22	15
行数 ^②	91	89
程序复杂性 ^③	1.04	1
行数/函数	4.1*	5.9**

① 因为我们并没有实现所有的 API，所以不能估计出函数的绝对数目，或者每个类中的函数数目，或者每个类的代码行数。但其比例关系是很有意义的。测试代码和功能代码的行数及函数数目大致相等。

② 可以通过将那些公共的固定代码抽出来削减测试代码的行数，然而模型代码与测试代码之间粗略的行数对应关系仍将保持不变。

③ 程序复杂性（cyclomatic complexity）是一种常用的衡量流程复杂性的方法。测试代码的复杂性为 1，这是因为测试代码中没有分支和循环。功能性代码复杂性比较低，这是因为多态的大量使用代替了显式的流程控制。

* 统计数据包括了函数头和结尾处的大括号。

** 测试中每个函数的代码行都很多，这是因为我们没有把公共的固定代码部分分离出来。这在“JUnit 的用法”一节中解释过。

过程

测试驱动开发的过程如下：

- 加入一个小的测试
- 运行所有测试，运行失败
- 适当修改
- 运行测试且成功
- 重构，消除重复设计，优化设计结构

假设写一个测试是只需一步，那么要完成编译、运行和重构需要多少改动呢？（这里所说的改动是指改变一个方法或类的定义。）图 17-2 的直方图表明了你所看到的每个货币测试的改动数目。

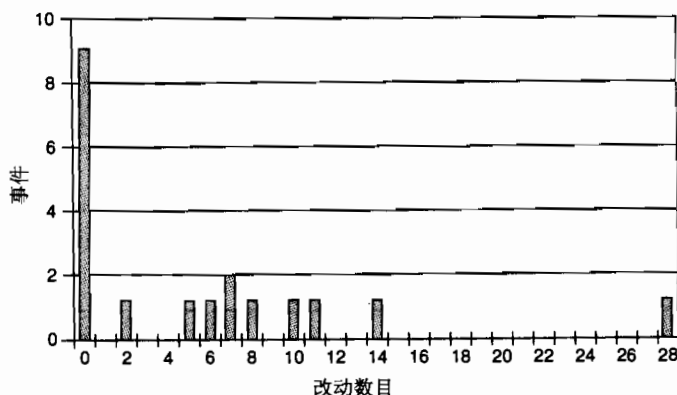


图 17-2 每次重构的改动数目

如果采集的是一个大型工程的数据，那么我们期望为了编译和运行而做的修改数目很小。（如果编程环境理解测试程序所要表达的意图的话——例如自动创建存根程序，那么这个数字甚至会更小。）然而，（这至少是一篇硕士论文）每次重构的改动数目应当遵循“胖尾巴形”或“山峰尖形”，它像一条钟型曲线，但相对标准钟型曲线来说，它带有更多的大幅变化。自然界中的许多测量数据都遵循这种规律，例如股票市场的价格波动^①。

测试质量

测试是测试驱动开发自然而然的副产品，只要系统在运行，保持测试代码的运行就是相当有益的。不要指望它们代替其他类型的测试，如：

① Mandelbrot, Benoit, ed. 1997. *Fractals and Scaling in Finance*. New York: Springer-Verlag. ISBN: 0387983635.

- 性能测试
- 压力测试
- 可用性测试

然而, 如果测试驱动代码的缺陷密度 (defect density) 足够低的话, 那么专业测试的地位将不可避免地“从大人的监督过程”转变为某种更像放大器的东西。这种放大器是那些知道系统应做什么和那些编写系统来做的人之间交流用的。未来专业测试的发展趋势是一个漫长而有趣的话题, 在此我们列举几个广泛认同的衡量手段, 用以衡量我们以上所编写的测试代码。

- 语句覆盖量 (statement coverage) 当然不是一种充分的衡量测试质量的方法, 但它是一个很好的出发点。认真遵循测试驱动开发的人理应实现 100% 的语句覆盖。JProbe (www.sitraka.com/software/jprobe) 的报告显示, 我们的测试代码只有一个方法中的一行没有被测试用例覆盖到, 这就是 `Money.toString()`。我们是作为辅助调试手段而不是模型代码增加上去的。
- 缺陷插入 (defect insertion) 是另一种评估测试质量的途径。其思想很简单: 改变一行代码的意思, 测试应该不能通过。你可以手工这样做, 或者使用工具, 例如 Jester (jester.sourceforge.net)。Jester 报告显示, 只有一行代码能够在改变意思后不会导致运行失败, 即 `Pair.hashCode()`。我们采用伪实现使之仅返回 0。返回一个不同的常量实际上不改变程序的意思 (任何捏造的数字都是一样的), 因此插入的缺陷并不是一个真正的缺陷。

这本书的审阅人 Philip 提出了一个有关测试覆盖的论点, 值得在这里重复一下。一种粗略的覆盖度量是测试程序不同位置的测试数量除以需要进行测试的位置的数目 (逻辑复杂度)。一种提高覆盖率的方法就是多写测试, 因此测试驱动开发者为代码所写的测试数量与传统的专业测试者所写的测试数量相差很大。(第 32 章中有一个具体的例子, 其中我编写了 6 个测试程序, 而有个测试者为同一个问题写了 65 个测试。) 提高覆盖率的另一途径是采用一系列固定的测试并简化程序逻辑。代码重构经常就有这样的效果——条件判定被消息取代, 或者从根本上取消这个条件。用 Philip 的话来说, “我们不是通过增加测试覆盖, 使其覆盖输入的所有排列组合情况 (更确切地说是一个有效的所有可能排列的简化), 而是让代码缩减, 从而使用相同的测试就能覆盖到各种排列组合情况。”

最后一次回顾

讲授测试驱动开发过程中意外地反复出现的三个方面是:

- 让测试利落运行的三种方法——伪实现、三角法以及显明实现
- 把消除代码与测试间的重复设计作为驱动设计的一种手段
- 控制测试间隙的能力, 当道路光滑时增大摩擦力, 在状况解除时就开快一些

第二部分

xUnit 实例

下面探讨如何实现测试驱动开发的工具。

用 Python 语言来实现 xUnit 非常自然，所以第二部分我将转向 Python 语言。先别着急，为了不曾接触过它的读者着想，我会在适当的地方加上些评注的。当学完这些内容以后，你会对 Python 有所了解，你将有能力编写自己的测试框架，而且还会看到一个更复杂的 TDD 例子——这就是一举三得。

18

步入 xUnit

通过先编写测试来开发一种测试工具，而编写测试的工具就是这件要开发的工具本身似乎有点像给自己做大脑外科手术。（“别碰那些运动神经中枢——噢，糟糕，一切都完了。”）这些内容有时会显得晦涩难懂。因此，测试框架的逻辑比第一部分功能较弱的货币例子复杂多了。本章是迈向采用测试驱动开发“真正”软件的一步，也可以把它作为自参考编程方面的计算机科学读物来读。

首先，我们需要能够创建一个 `TestCase` 并且运行一个测试方法。例如：`TestCase("testMethod").run()`。我们有一个第一步怎么走的问题：我们是在为一个将被用来编写测试用例的测试框架编写测试它的测试程序。因为我们目前还没有框架，所以将不得不通过手工来验证开始的一小步。幸运的是，我们放松休息的不错，因此不大可能会犯什么错误。这也是我们以极小的步伐前进、从各个方面进行验证的原因。下面是我所能想到的测试框架的测试清单。

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

我们仍然是先进行测试。就最初的第一个测试程序而言，我们需要这样一个小程序。它在一个测试方法被调用时输出“true”，否则输出“false”。如果我们有这样一个测试用例，它在测试方法中设置一个标志位，这样在完成之后，我们可以打印标志位并确保它是正确的。一旦我们手工验证了这一点，那么就可以采用自动化的方式处理这一过程。

我们最初的测试程序的策略是这样的：创建一个带有标志位的 `TestCase`。在测试方法运行前，标志位被置为“false”。测试方法将设置这个标志位。在测试方法运行后，标志位被置为“true”。我们为 `TestCase` 类起名为 `WasRun`，因为它是一个报告方法是否运行的测试用例。标志位也叫做 `wasRun`（可能会有些混淆，但这是一个非常棒的名字）。因此我们最终可以这样写：

`assert test.wasRun` (`assert` 是 Python 的内建设施)。

Python 能够边读取文件边执行其中的语句, 因此我们可以手工调用测试方法:

```
test= WasRun("testMethod")
print test.wasRun
test.testMethod()
print test.wasRun
```

我们期望在方法运行以前这会打印出“None”(Python 中的 None 就像 `null` 和 `nil`, 与 0 和其他几个对象一样均代表“false”)并在方法运行之后输出“1”。结果不是这样, 因为我们目前还没有定义类 `WasRun` (测试优先, 测试优先嘛)。

WasRun

```
class WasRun:
    pass
```

(关键字 `pass` 在类或方法还没有实现时使用。)现在编译器告诉我们需要一个 `wasRun` 属性。我们需要在创建类实例时创建这个属性 (构造函数被称做 `__init__`)。我们在构造函数中把 `wasRun` 标志设置为“false”。

WasRun

```
class WasRun:
    def __init__(self, name):
        self.wasRun= None
```

现在运行, 程序忠实地输出“None”, 接着告诉我们需要定义方法 `testMethod`。(如果你的 IDE 注意到这一点, 就会为你提供存根实现, 并打开一个编辑器让你对这一实现进行编辑, 岂不更棒? 是的, 这样最好不过了。)

WasRun

```
def testMethod(self):
    pass
```

现在当我们执行这个文件时, 得到的是“None”和“None”。而我们想得到的是“None”和“1”, 可以在 `testMethod()` 中设置标志位来达到这一目的。

WasRun

```
def testMethod(self):
    self.wasRun= 1
```

现在我们得到了正确答案——可运行状态。万岁! 我们有一堆的重构要做, 但是只要我们保持测试的可运行状态, 我们就知道我们是在进步。

下一步, 我们需要使用真正的接口——`run()`, 而不是直接调用测试方法。测试修改如下:

```
test= WasRun("testMethod")
print test.wasRun
test.run()
print test.wasRun
```

此刻我们把代码硬性写成:

WasRun

```
def run(self):
    self.testMethod()
```

我们的测试又可以输出正确的数值了。许多重构有这种感觉——分成两部分，这样你就能独立地对其进行操作。当完成时，如果它们又整合到了一起，那也没什么；如果没有整合到一起，那么也就不管了。在这个用例中，我们期望最终创建一个超类 `TestCase`，但首先，我们必须区分例子中的各个部分。这里可能同细胞的有丝分裂有奇妙的相似之处，但我的细胞生物学知识不足以解释这个。

下一步是动态调用 `testMethod`。Python 最酷的特性之一就是可以把诸如类和方法的名字这样的条目当作函数对待（请看上面 `WasRun` 的调用）。当我们取得与测试用例名称相对应的属性时，返回的是一个对象，当把它作为函数调用时，我们调用的就是方法^①。

WasRun

```
class WasRun:
    def __init__(self, name):
        self.wasRun = None
        self.name = name
    def run(self):
        method = getattr(self, self.name)
        method()
```

这里有另外一种一般性的重构模式：将在一个实例中能够工作的代码中的常量替换成变量，使其一般化，这样就能在许多实例中工作了。这里的常量是硬性写死的代码，并不是指数据值，但是其基本原理是一样的。测试驱动开发通过良好地运行实例可以让你据其很好地实施一般化，而不是纯粹通过自己的推断来进行一般化工作。

现在我们的小小的类 `WasRun` 正在做两项不同的工作：其一是跟踪方法是否被调用，其二是动态调用方法。现在是细胞有丝分裂时间了。首先，我们创建一个空的超类 `TestCase`，而且让 `WasRun` 成其子类：

TestCase

```
class TestCase:
    pass
```

WasRun

```
class WasRun(TestCase): ...
```

现在，我们把 `name` 属性上移到超类：

TestCase

```
def __init__(self, name):
    self.name = name
```

WasRun

```
def __init__(self, name):
    self.wasRun = None
    TestCase.__init__(self, name)
```

① 感谢 Duncan Booth 纠正了我这个 Python 新手犯的错误，并指出了 Python 的解决之道。

最终，`run()`方法用到的属性仅仅来自于超类，因此它可能属于超类。（我总是注意把操作放在靠近数据的地方。）

TestCase

```
def __init__(self, name):
    self.name= name
def run(self):
    method = getattr(self, self.name)
    method()
```

执行完每一步后，我都运行测试，以确保得到同样的答案。

我们已经厌倦了每次都看到打印输出“None”和“1”。用我们刚刚创建的机制，现在可以编写以下代码：

TestCaseTest

```
class TestCaseTest(TestCase):
    def testRunning(self):
        test= WasRun("testMethod")
        assert(not test.wasRun)
        test.run()
        assert(test.wasRun)
TestCaseTest("testRunning").run()
```

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

测试的主体仅仅由打印语句变成了断言，因此你刚才所看到的只是析取方法（`Extract Method`）的一种复杂形式。

要告诉你一个小秘密。我查看了刚才向你展示的开发步伐的大小，看起来有点荒唐。另外，我尝试了使用更大的步伐，总共大约花费了六小时（我不得不花许多时间查找与 Python 有关的知识），推倒重来了两次，这两次我都是以为代码能够工作了，而事实并非如此。对 TDD 来说这可能是最糟糕的情况了，因为我们努力想熬过最初的一步。

没有必要工作的这么细致。一旦你掌握了 TDD，就能通过不同的测试用例来实现功能上大的跨越。然而，要想精通 TDD，你应该能够在需要时有能力小步前行。

接下来，在运行测试之前，我们将先调用 `setUp()`。回顾一下，在本章

- 在经历了几次自大的错误以后，我们终于搞清楚了如何以非常小的一步来开始
- 实现了功能，首先硬性写死代码，然后通过用变量代替常量来使它更通用
- 使用了插入选择器（`Pluggable Selector`），我们曾保证至少四个月不会再用它，因为它使得很难对代码进行静态分析
- 完全都以极小的步伐，为我们的测试框架做好了第一步

19

设置表格

当你开始写测试时，会发现一种常见模式（Bill Wake 将它总结为 3A 原则）。

- (1) 安排（Arrange）——创建一些对象
- (2) 行动（Act）——激活它们
- (3) 断言（Assert）——检查结果

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

对于不同的测试，第一步，安排，经常是差不多的，而第二步和第三步，行动与断言，则是不同的。我有一个 7 和一个 9。若它们相加，我期望得到的是 16；若它们相减，我期望得到的是 -2；若它们相乘，我期望得到的是 63。操作及期望的结果是变化的，但 7 和 9 是不变的。

如果这一模式以不同规模重复（它确实如此），则我们面对的问题是：创建用于测试的新对象的频繁程度。有这样两条相互矛盾的约束：

- 性能——我们更愿意让测试尽可能快地运行，即若在几个测试中用相似的对象，我们将一次创建所有的对象来测试。
- 隔离——我们希望一个测试的成功与失败与另外的测试不相干。若测试共用对象且一个测试改变了对象，那么接下来的测试可能会改变它们的结果。

测试耦合显然有不良的后果，因为如果一个测试存在问题，那么即使后十个测试的代码是正确的，它们也会失败。在测试顺序有关的情况下，测试耦合会有一种微妙而讨厌的影响。若测试 A 先于测试 B 运行，则它们皆通过，但若测试 B 先于测试 A 运行，则测试 A 失败。或者甚至更令人气愤，测试 B 的代码错了，但因为测试 A 先运行，它反而通过了。

测试耦合——不要涉足其中。我们暂且假定我们能相当快地创建对象。这种情况下，测试

运行的每时每刻我们都可以创建对象。在 `WasRun` 中，我们已看到了这种情况，只不过不太明显而已。其中，在运行测试前，我们想要把标志设置为“false”。我们将逐步实现这些功能，首先我们需要有关测试：

`TestCaseTest`

```
def testSetUp(self):
    test= WasRun("testMethod")
    test.run()
    assert(test.wasSetUp)
```

运行它（在文件中加上最后一行 `TestCaseTest("testSetUp").run()`），Python 礼貌地通知我们这里没有 `wasSetUp` 属性。当然没有，我们还没有设置它。这个方法应该设置它：

`WasRun`

```
def setUp(self):
    self.wasSetUp= 1
```

如果我们调用它，上面的代码就可以了。调用 `setUp` 是 `TestCase` 的工作，因此我们要对那里进行修改：

`TestCase`

```
def setUp(self):
    pass
def run(self):
    self.setUp()
    method = getattr(self, self.name)
    method()
```

这是使一个测试用例运行的两个步骤，在这些比较棘手的情况下算是太多了。我们来看看它是否能够运行通过。是的，它通过了。不过如果你想学到些东西的话，那么就思考一下如何在每次修改不超过一个方法的情况下让测试运行通过。

我们马上就可以使用我们的新工具来削减测试规模了。首先，我们可以通过设置 `setUp` 中的 `wasRun` 标志位来简化 `WasRun`：

`WasRun`

```
def setUp(self):
    self.wasRun= None
    self.wasSetUp= 1
```

我们必须简化 `testRunning`，让它在运行测试前不检查标志位。我们愿意放弃对代码如此多的自信吗？只有在 `testSetUp` 准备就绪时我们才敢这样。这是一个常见的模式——一个测试当且仅当另一个测试准备就绪且正确地运行时才能简化：

`TestCaseTest`

```
def testRunning(self):
    test= WasRun("testMethod")
    test.run()
    assert(test.wasRun)
```

我们也可以简化测试本身。在这两个用例中，我们都创建了 `WasRun` 的实例，完全依照我

们早先所谈论的固定设置。我们可以在 `setUp` 中创建 `WasRun`，并在测试方法中加以使用。每个测试方法都是在一个干净的 `TestCaseTest` 实例中运行的，因此这两个测试没有任何机会产生耦合。（我们假定对象没有以某种难以置信的丑陋方式进行交互，如设置全局变量，但我们不会那样做的，至少不会在读者的眼皮底下这么做的。）

`TestCaseTest`

```
def setUp(self):
    self.test = WasRun("testMethod")
def testRunning(self):
    self.test.run()
    assert(self.test.wasRun)
def testSetUp(self):
    self.test.run()
    assert(self.test.wasSetUp)
```

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

下一步，我们将在测试方法后运行 `tearDown()`。回顾这一章，我们

- 此刻明确了编写简单的测试比测试的执行性能更重要
- 测试并实现了 `setUp()`
- 用 `setUp()` 简化了测试用例范例
- 用 `setUp()` 简化了用来检查测试用例范例的测试用例（我以前给你们说过，这就像自己给自己的大脑动外科手术）

20

后期整理

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

有些时候测试程序需要在 `setUp()` 中分配外部资源。如果我们希望测试程序保持独立性，那么分配外部资源的测试程序需要在运行完以前释放资源，比如在 `tearDown()` 方法中释放。

要想为释放已分配资源编写测试程序，一种简单的方法是再引入另外一个标志位。所有这些标志位开始讨人嫌起来，而且它们遗漏了这些方法的一个重要方面：`setUp()` 必须在测试方法运行前调用，而 `tearDown()` 方法则在运行完后调用。我打算改变测试策略，简单记录下曾经被调用过的方法。通过这样不断地在日志中追加记录，我们就可以将各个方法的调用顺序保留下来。

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

在 `WasRun` 的日志中记录信息串

`WasRun`

```
def setUp(self):
    self.wasRun= None
    self.wasSetUp= 1
    self.log= "setUp "
```

现在我们可以改动 `testSetUp()` 方法，让它来查看日志信息而不是标志位：

`TestCaseTest`

```
def testSetUp(self):
```

```
self.test.run()
assert("setUp " == self.test.log)
```

下面我们可以删除 `wasSetUp` 这个标志位了，我们还可以做测试方法的运行记录：

```
WasRun
def testMethod(self):
    self.wasRun= 1
    self.log= self.log + "testMethod "
```

这使 `testSetUp` 没有运行通过，因为日志中实际包含的信息是“`setUp testMethod`”。下面我们来改变期望值：

```
TestCaseTest
def testSetUp(self):
    self.test.run()
    assert("setUp testMethod " == self.test.log)
```

现在的这个测试做了原来两个测试的工作，所以我们可以删除 `testRunning` 并且重命名 `testSetUp` 方法：

```
TestCaseTest
def setUp(self):
    self.test= WasRun("testMethod")
def testTemplateMethod(self):
    self.test.run()
    assert("setUp testMethod " == self.test.log)
```

遗憾的是，我们使用的是 `WasRun` 的同一个实例，所以我们必须退回去，扔掉那个聪明的 `setUp` 技巧：

```
TestCaseTest
def testTemplateMethod(self):
    test= WasRun("testMethod")
    test.run()
    assert("setUp testMethod " == test.log)
```

根据前面几处的用法进行重构，不久又会撤消所做的工作，这是十分常见的。一些人不愿意作撤销的工作，所以他们一直等到有三四个用例时才进行重构。对我自己来说，我更喜欢在设计的时候多多思考，所以我只是闷头进行重构，并不担心后面会很快撤销所做的工作。

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

在 `WasRun` 的日志中记录信息串

现在我们已经做好了实现 `tearDown()` 方法的准备。知道了！首先我们准备对 `tearDown` 进行测试：

TestCaseTest

```
def testTemplateMethod(self):
    test= WasRun("testMethod")
    test.run()
    assert("setUp testMethod tearDown " == test.log)
```

这个测试失败了。要让它工作很简单：

TestCase

```
def run(self, result):
    result.testStarted()
    self.setUp()
    method = getattr(self, self.name)
    method()
    self.tearDown()
```

WasRun

```
def setUp(self):
    self.log= "setUp "
def testMethod(self):
    self.log= self.log + "testMethod "
def tearDown(self):
    self.log= self.log + "tearDown "
```

令人吃惊的是，我们得到一个错误，这个错误不在 **WasRun** 里，而在 **TestCaseTest** 里。这是因为 **TestCase** 中没有 **tearDown()** 的无操作（no-op）实现：

TestCase

```
def tearDown(self):
    pass
```

这一次，我们使用同一个正在开发的测试框架得到了数值。不需要进行重构。这种显明实现在经历了一次小的失败之后可以工作了，而且代码清晰整洁。

调用测试方法

调用测试方法之前先调用 **setUp** 方法

调用测试方法之后调用 **tearDown** 方法

即使测试方法失败也同样调用 **tearDown** 方法

运行复合测试

报告集成结果

在 **WasRun** 的日志中记录信息串

下一章我们将报告显式运行测试的结果，而不再像以前那样，当出问题的时候，让 Python 的内部错误处理与报告系统用断言来通知我们。最后回顾一下本章的内容，本章我们：

- 重新调整测试策略，以日志代替标志位
- 使用新的日志测试并实现了 **tearDown()** 方法
- 发现一个问题，大胆地修复它，而不是退回去重新开始（这是不是个好主意？）

21

计 数

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

在 `WasRun` 的日志中记录信息串

我打算引入一种实现以确保即使测试方法中出现了异常，`tearDown()`仍然会被调用。不管怎样，为了使测试能够工作，我们需要捕获异常。（我知道，我刚才尝试过这么做，失败后回退了。）如果我们在实现这种机制的时候犯了错误，那么就不能够看到错误，因为这些异常是不会报告出来的。

一般来说，测试实现的顺序十分重要。当选择下一个要实现的测试时，我会选择一个能够教会我某些东西而且是我有信心能够使其工作的测试。如果我能让那个测试工作，但却在下一个测试上卡壳了，那么我就会考虑一下回退两步。如果编程环境能够在每次所有的测试都运行通过的情况下帮我为代码设置一次检查点（`checkpoint`）的话，那可就帮上大忙了。

我们想要看到任意数量测试运行的结果——“5 个运行，2 个失败，`TestCaseTest.testFooBar`——`ZeroDivideException`，`MoneyTest.testNegation`——`AssertionError`”。这样，如果测试没有被调用执行，或者结果没有报告出来，那么至少我们应该有机会捕获这种错误。让这样一个对任何测试用例都一无所知的测试框架报告这些内容似乎有点不可思议，至少对第一个测试来说是这样。

我们将让 `TestCase.run()`方法返回一个 `TestResult` 对象，这个对象记录了测试运行的结果（目前测试是个单数，但我们后面会解决的）。

TestCaseTest

```
def testResult(self):
    test= WasRun("testMethod")
    result= test.run()
    assert("1 run, 0 failed" == result.summary())
```


我们将从一个伪实现开始：

TestResult

```
class TestResult:
    def summary(self):
        return "1 run, 0 failed"
```

而且作为 `TestCase.run()` 运行的结果，我们将返回一个 `TestResult` 对象：

TestCase

```
def run(self):
    self.setUp()
    method = getattr(self, self.name)
    method()
    self.tearDown()
    return TestResult()
```

现在测试运行通过，我们可以一次一点地来完成（就像“美梦成真”一样）`summary()`的实现。首先，我们可以用一个符号常量来代表测试的数目：

TestResult

```
def __init__(self):
    self.runCount = 1
def summary(self):
    return "%d run, 0 failed" % self.runCount
```

（操作符“%”是 Python 语言的 `printf`。）然而，`runCount` 不应该是一个常量；它应该通过点查测试的运行数目计算得出。我们可以将其初始化为 0，然后每当一个测试运行时就将其加一：

TestResult

```
def __init__(self):
    self.runCount = 0
def testStarted(self):
    self.runCount = self.runCount + 1
def summary(self):
    return "%d run, 0 failed" % self.runCount
```

我们必须实际调用这个时髦的新方法：

TestCase

```
def run(self):
    result = TestResult()
    result.testStarted()
    self.setUp()
    method = getattr(self, self.name)
    method()
    self.tearDown()
    return result
```

就像我们实现 `runCount` 一样，我们可以将记录失败测试次数的符号常量“0”转换为一个变量，但目前的测试并没有这样的要求，所以我们另外再写一个测试：

TestCaseTest

```
def testFailedResult(self):
```

```
test= WasRun("testBrokenMethod")
result= test.run()
assert("1 run, 1 failed", result.summary)
```

其中:

WasRun

```
def testBrokenMethod(self):
    raise Exception
```

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

在 `WasRun` 的日志中记录信息串

报告失败的测试

我们注意到的第一件事就是我们并没有捕获由 `WasRun.testBrokenMethod` 抛出的异常。我们希望捕获这个异常并且在结果中注明这个测试失败了。我们将暂时不执行这个测试。

作为回顾,在这一章,我们

- 写了一个伪实现,并且通过将常量转换为变量逐步将这个实现变成真实的
- 另外写了一个测试
- 当那个测试失败时,为使失败的测试能够工作,采用更小的规模又编写了另外一个测试

确性，但我现在不想这么麻烦，因为咖啡起作用了)，那么打印输出的结果就是正确的：

TestResult

```
def summary(self):
    return "%d run, %d failed" % (self.runCount, self.failureCount)
```

现在如果我们能够正确地调用 `testFailed()`，那么我们就将能够得到所期望的结果。什么时候调用它呢？是在我们捕获到测试方法中的异常的时候：

TestCase

```
def run(self):
    result = TestResult()
    result.testStarted()
    self.setUp()
    try:
        method = getattr(self, self.name)
        method()
    except:
        result.testFailed()
    self.tearDown()
    return result
```

在这个方法中隐藏了一个微妙的问题。问题在于代码的书写方式，如果 `setUp()` 运行时出现了灾难性的错误，那么我们就无法捕捉到异常。这可不是我们想要的——我们希望我们的测试相互之间独立运行、互不干扰。不管怎样，在修改代码以前，我们需要编写另外一个测试。（我最大的女儿 Bethany，当她十二岁的时候，我就把 TDD 作为她的第一种编程方式教给她。她认为除非存在无法运行通过的测试，否则你就不能编写代码。而我们这些人只有通过不断地提醒自己编写测试才行。）我将把下一个测试及其实现作为练习留给读者（我的手又敲累了）。

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

在 `WaaRun` 的日志中记录信息串

报告失败的测试

捕捉而且报告 `setUp` 中的错误

下一章我们将致力于把若干个测试放在一起运行。回顾这一章，我们

- 使我们的小规模测试获得通过
- 再次引入了那个大规模的测试
- 使用经过小规模测试论证的机制，使得大规模测试很快地运行通过
- 注意到了一个问题，没有立即着手解决，而是把它记录到计划清单上

23

如何组成一组测试

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

在 `WasRun` 的日志中记录信息串

报告失败的测试

捕捉而且报告 `setUp` 中的错误

我们不能在没有讲解 `TestSuite` 的情况下就抛开 `xUnit` 这个话题。在文件的末尾，我们调用了所有的测试，看上去乱糟糟的：

```
print TestCaseTest("testTemplateMethod").run().summary()
print TestCaseTest("testResult").run().summary()
print TestCaseTest("testFailedResultFormatting").run().summary()
print TestCaseTest("testFailedResult").run().summary()
```

重复设计总不是什么好事，除非你将其看做是寻找遗漏设计要素的动力。现在我们需要的是把测试组合到一起并成批运行的能力。（如果我们只是每次运行一个测试的话，那么费那么大劲来使测试相互独立、互不干扰地运行就显不出有多大的好处了。）实现 `TestSuite` 的另外一个理由是它提供给我们一个纯粹的关于合成（`Composite`）的例子——我们希望将单个测试与组合测试等同对待。

我们希望能够创建一个 `TestSuite`，向其中添加一些测试，然后运行这个 `TestSuite` 而得到综合性的测试结果：

TestCaseTest

```
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.add(WasRun("testBrokenMethod"))
    result= suite.run()
    assert("2 run, 1 failed" == result.summary())
```



```

result= TestResult()
test.run(result)
assert("1 run, 1 failed" == result.summary())
def testFailedResultFormatting(self):
    result= TestResult()
    result.testStarted()
    result.testFailed()
    assert("1 run, 1 failed" == result.summary())

```

我们注意到：每个测试都分配了一个 `TestResult`，这正好是要由 `setUp()` 解决的问题。我们可以在 `setUp()` 中创建 `TestResult` 来简化这些测试（代价是使得这几个测试的可读性变差）：

TestCaseTest

```

def setUp(self):
    self.result= TestResult()
def testTemplateMethod(self):
    test= WasRun("testMethod")
    test.run(self.result)
    assert("setUp testMethod tearDown " == test.log)
def testResult(self):
    test= WasRun("testMethod")
    test.run(self.result)
    assert("1 run, 0 failed" == self.result.summary())
def testFailedResult(self):
    test= WasRun("testBrokenMethod")
    test.run(self.result)
    assert("1 run, 1 failed" == self.result.summary())
def testFailedResultFormatting(self):
    self.result.testStarted()
    self.result.testFailed()
    assert("1 run, 1 failed" == self.result.summary())
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.add(WasRun("testBrokenMethod"))
    suite.run(self.result)
    assert("2 run, 1 failed" == self.result.summary())

```

调用测试方法

调用测试方法之前先调用 `setUp` 方法

调用测试方法之后调用 `tearDown` 方法

即使测试方法失败也同样调用 `tearDown` 方法

运行复合测试

报告集成结果

在 `WasRun` 的日志中记录信息串

报告失败的测试

捕捉而且报告 `setUp` 中的错误

从 `TestCase` 类中创建 `TestSuite` 对象

所有这些“self”看上去有点不顺眼，但这是 Python 语言规定的。如果这是一种面向对象语言的话，那么就会默认提供 self，而引用全局变量也要求指定限定。然而，这是一种脚本语

言,增加了对对象的支持(当然支持得很好),所以默认为全局引用,并且必须显式地指明 `self`。

我将把清单中余下的任务交给你,请用新学到的 TDD 技能来完成。

作为回顾,在本章,我们

- 为 `TestSuite` 写了一个测试
- 完成了一部分实现,但并没有能让测试运行通过。这是在违反游戏规则。如果你在当时就发现了这一问题,那么可以花费些精力,编写两个测试用例。我确信存在一种简单的伪实现可以让这个测试用例运行通过,这样我们就能够在测试运行通过的状态下进行重构,但是我此刻还想不出到底是怎样的一种实现
- 改变了 `run` 方法的接口,以使单项以及单项的组合可以以相同的方式运行,最终让测试运行通过
- 把公共的 `setup` 代码分离出来

24

xUnit 回顾

如果有一天你开始实现自己的测试框架,那么本书第二部分所描述的过程可以作为你行动的指南。实现的具体细节并没有测试用例更重要。如果你能够提供类似于本书中给出的那样一组测试用例的话,那么就能够编写出互相隔离并能组合使用的测试,从而踏上有能力执行测试优先开发的征途。

在写本书的时候, xUnit 已经被移植到了三十多种编程语言上。你所使用的语言或许已经有 xUnit 的实现了。但是即便存在可用的版本,仍有下面几个自己来实现 xUnit 的理由:

- 完全控制——xUnit 的精髓就在于简单。Martin Fowler 说过,“软件工程有史以来从来没有如此众多的人大大受益于如此简单的代码。”一些实现对我来说有点过于复杂了。通过自己来实现这种工具将会给你一种一切都在自己控制之下的感觉。
- 探索——当我开始面对一种从未接触过的编程语言时,我就会实现 xUnit。到了有八至十个测试能够运行的时候,我就已经探索出其中许多将会在今后的日常编程中用到的功能了。

当你开始使用 xUnit 的时候,你会发现,当测试运行时失败的断言与其他类型的错误存在着非常大的差别。失败的断言所需要的调试时间要长得多。正因为如此,大多数 xUnit 实现都对各种失败加以区分,即把断言失败和错误区分开来。GUI 用不同的方式呈现它们,经常是把错误显示在上面。

JUnit 声明了一个简单的 Test 接口, TestCase 和 TestSuite 都实现了这一接口。如果你想让 JUnit 工具能够运行你的测试,那么你也可以实现 Test 接口。

```
public interface Test {  
    public abstract int countTestCases();  
    public abstract void run(TestResult result);  
}
```

乐观(动态)的编程语言甚至无需声明它们所要履行的接口——它们只需实现这些操作就行了。如果你要编写一种测试脚本语言,那么 Script(脚本)就可以实现 countTestCases()来返回 1,并能够在运行失败时通知 TestResult,而你就可以同普通的 TestCases 一起运行你的脚本了。

第三部分

测试驱动开发的模式

接下来我们要介绍的是 TDD（测试驱动开发）中“最重要的”模式。其中有些模式是 TDD 技巧，有些是设计模式，还有一些是重构方法。如果你熟悉其中的某个技巧，那么这里所描述的模式将向你展示这些内容是如何与 TDD 一起工作的。如果你一无所知，那么本书将提供充足的素材，足以让你理解书中的这些例子，足够引起你到别处寻找与之相关的资料的兴趣。

25

测试驱动开发模式

在开始讨论怎样进行测试的具体细节之前，我们需要先回答一些基本的策略问题：

- 测试是什么意思？
- 什么时候进行测试？
- 如何选择要测试的逻辑？
- 如何选择要测试的数据？

测试（名词）

你怎样测试自己的软件？编写自动测试程序。

测试（test）是一个动词，意思是“评估”。没有一个软件工程师会在没有测试的情况下公开甚至是最微小的改动，那些极其自信的或极其毛糙的工程师除外。既然你已经读到这里，我假定你二者都不是。虽然你会对改动进行测试，但是对改动进行测试与装备有测试可不是一回事。测试还是一个名词，意思是“导致最终是接受还是不接受的过程。”为什么测试作为名词时（指一个自动运行的过程）与作为动词时（就是按几个键，然后观察屏幕上的显示结果）让人感觉如此不同？

下面是一张影响图（influence diagram），即 Gerry Weinberg 的优秀软件管理。两个节点之间的箭头表示当第一个节点增长时意味着第二个节点也会作相应的增长。带有圆圈的箭头表示第一个节点增长时，第二个节点会相应减少。

当压力水平上升时会出现什么情况呢？

这是一个正反馈环。你感觉到的压力越大，你所做的测试就越少。你做的测试越少，你将犯的错误就越多。你犯的错误越多，你感觉到的压力就越大。如此循环往复。

如何跳出这样一个循环呢？要么引进一个新的元素，将现有元素中的一个替换掉，要么改变箭头。就我们的情况而言，我们将用自动测试来替代测试。

“我的这次改动是不是会对什么造成破坏呢？”图 25-1 展示了现实工作中的这种动态过程。如果采用自动测试，那么当我感觉到压力的时候，我就会运行测试程序。测试是程序员的

试金石，可以将对压力的恐惧变为平日的琐事。“是的，我没有破坏任何东西，测试仍然运行通过。”我越是感到紧张，越是更多次地运行测试。运行测试马上会给我一种良好的感觉，而且会减少我出错的次数，继而减少我所感觉到的压力。

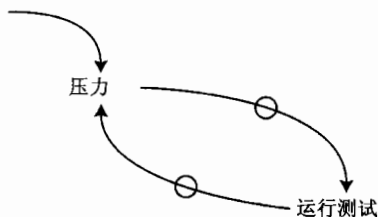


图 25-1 “没时间测试”的死循环图

“我们没有时间去运行测试了，发布软件吧！”第二种情况我们不能保证，如果压力达到一定的程度，它会崩溃的。但是，采用自动测试，你就有机会选择恐惧的程度。

你应该在编写完代码之后运行测试吗？即使你知道这个测试必然失败？不，不必了。举个例子，我曾经和几个非常聪明的年轻程序员一起工作，一起来实现一种内存级的事务（这是每种编程语言都应该具备的一种非常酷的技术）。问题是这样的：如果我们启动一个事务，修改了几个变量，然后让垃圾回收机制回收这个事务，那么我们如何实现回滚（rollback）呢？测试工作很简单，年轻人。往后站，看看大师是怎么做的。测试程序写好了。现在如何实现它呢。

两个小时以后——挫折重重的几个小时，因为一个实现如此低级功能的错误把整个开发环境都搞崩溃了——我们回滚到起初开始的地方。编写测试程序，也没想那么多就开始运行测试。运行通过了，天啊！事务机制的整个关键所在就是在事务被提交以前变量并没有被真正修改。好了，我想你可以继续做下去了，如果需要的话可以运行一下这个新的测试程序。

相互独立的测试（Isolated Test）

所运行的各种测试应该怎样互相影响呢？没有任何相互影响。

当我还是一个年轻程序员的时候——那是很久很久以前，那时我们不得不把那些小东西从雪地里挖出来，光着脚用沉重的桶把它们扛回我们的小隔间，留下可以让狼尾随而至的血红的小脚印……对不起，我又开小差了。我最初的自动测试的经历是，有一大批运行时间很长的（需要一晚上时间运行）基于 GUI 的测试（你知道的，就是记录下击键和鼠标事件，然后进行回放），这是为我当时正在开发的调试器编写的测试。每天早上我进来的时候，我的椅子上总是整齐地放着一堆纸，描述了昨晚测试程序运行的情况。运气好的话，那儿只有一张纸，报告一切正常。运气不好的话，那儿会有许多张纸，每张纸都记录着一个失败的测试。当我看到这么一大堆纸在我椅子上时，我就知道恐怖的一天开始了。

从这次经历我吸取了两个教训。第一，就是让测试程序尽可能快地运行，这样我自己就可

以运行它们了，而且可以经常运行。这样我就能够在任何人看到这些错误以前捕获到它们，而且我在早上再也不用害怕进办公室了。其次，不久以后我注意到一大堆纸经常并不意味着有一大堆问题。更多的时候是由于某个测试早早中断了，而下一个测试运行时系统处于一种不确定的状态。

我尝试着在两个测试之间重新启动系统来避开这个问题。但这耗时太长了，我又学会了尽量在小范围内进行测试，而不是对整个应用进行测试。但是我所得到的主要教训还是测试相互之间必须互不干扰。如果一个测试失败了，那么我希望对应的是一个问题；如果有两个测试失败了，那么我希望对应的是两个问题。

相互独立的测试意味着所有的测试都是不依赖于顺序的。如果我想从这些测试中挑出一部分来运行，那么我就不用担心一个测试会因为前导测试不存在而失败。

人们经常引用性能来作为测试要共享数据的理由。相互独立的测试的第二个含义是，你必须努力，有时候要相当努力，让你的问题分解为一些彼此正交（orthogonal）的小问题，使得为每个测试建立环境更加简单而快捷。独立测试鼓励你根据高内聚、松耦合的对象组合出解决方案。我经常听到别人说这是个好办法，而且当我实现了这一点时我也非常高兴。但直到我开始编写独立的测试的时候，我才确切地知道如何获得高内聚与松耦合。

测试列表（Test List）

你应该测试什么呢？在开始写测试之前，写一个包含所有你认为必须要编写的测试的清单。我们应对编程压力的第一步就是在知道下一步该怎么走之前不要贸然把工作向前推进。当我们坐下来开始进行新一轮的编程时，要考虑清楚我们这一轮编程打算完成什么样的工作。

一种记录我们所要努力完成的工作的策略就是全部用脑子来记。我尝试着这样做有很多年了，发现我陷入了一种正反馈环。我的经验积累得越多，我所知道的有可能要完成的事情就越多。我所知道的可能要完成的事情越多，我就对手头所做的工作关注得越少。我对手头的工作关注得越少，我完成的工作就越少。我完成得越少，我知道需要完成的工作就越多。

不管清单中所罗列的项目多么杂乱，只是凭一时冲动编写程序看来不能够打破这种循环。

我养成了这样的习惯，把我想在以后几个小时之内完成的任何事情都记录在电脑旁的一张纸上。我还有一张这样的清单，我把它钉在墙上，不过这张单子是这一周或者这个月的工作计划。一旦把所有要完成的事情都写在清单上，我就知道我是不会忘记去做这些事情了。当又有新条目出现时，我会迅速而有意识地判断这个条目是属于“现在的”清单还是“以后的”清单，还是它根本就不需要去做。

将其运用到测试驱动开发上，那么我们记录到列表上的就是我们要去实现的测试。首先，把你所知道需要实现的每种操作的范例都记录在清单上。其次，对于那些目前还不存在的操作，将其空（null）版本记录到清单上。最后，列出所有你认为在这一轮编程结束后为了获得整洁的代码而必须要完成的重构。

我们可以不用勾勒出各种测试的框架，而是直接做这些测试具体的实现。存在两种原因使得集中实现这些测试不适合我。首先，每个你所实现的测试都会在进行重构的时候存在一定的惯性。如果使用自动重构工具的话（比如说，可以有一个菜单项，它能够对声明以及所有使用某个变量的地方进行重命名），这可能不成问题。但是如果你实现了十个测试后突然发现参数必须以相反的顺序排列时，你是不大可能重新进行整理的。其次，如果你有十个测试都失败了，那么离绿色状态条出现的时候（测试全部运行通过的时候）还早着呐。如果你希望很快让一切测试都运行通过，那么就必须扔掉这十个测试。如果你想让所有的测试都运行通过，那么就得准备在很长时间内看到的一直是红色状态条（即仍存在没有通过的测试）。如果你完全是一个无法摆脱绿色状态条引诱的人，那么如果状态条是红色的话，你就不能去洗个澡什么的，那么这段时间就显得更加漫长了。

保守的登山家有这样一条规矩，登山时，不管任何时候四肢之中必须有三个附在物体上。如果你松开两只以上的手或脚进行移动，危险立刻就大多了。在纯粹的 TDD 模式中，从这一点上讲，导致你远离绿色状态条的改变只能有一个，这很像登山运动中的四肢之中有三个必须依附在物体上的规矩。

在你让测试运行起来的时候，实现会引入新的测试，你需要把这些新的测试记在清单上。对于所带来的重构也要如此为之。

“这样列表变得乱糟糟的。”

“唉，把它加到列表上，我们会在下次处理它。”

在这一轮编程结束的时候，需要好好保管列表上剩余的项目。如果其中的某项功能你确实完成了一半的话，那么后面你还可以使用同一张列表。如果你发现了超出目前所能完成任务的更大的重构需求，那么就把这些项目移到“以后的”列表上去。我不记得自己是否干过把测试用例转移到“以后的”清单上去的事了，如果我认为某个测试可能无法运行通过的话，那么让这个测试工作起来要比发布代码更重要。

测试优先 (Test First)

你应该在什么时候编写测试呢？在你编写要被测试的代码之前。

你不能事后再进行测试。作为一个程序员，你的目标是使功能能够实现。然而，你需要一种考虑设计的方式，你需要一种控制规模的方法。

让我们来考虑一下那个常见的将压力与测试关联起来的影响图（这里不是压力测试，那是不同的概念）：压力节点画在上面，消极地作用于下面的测试节点，测试节点也消极地作用于压力节点（见本章前面的“测试（名词）”一节）。你感觉到的压力越大，你就越不想去做足够的测试。当你知道你做的测试不够时，这又增加了你的压力。这是一个正反馈环。同样，我们必须找到一个跳出这个循环的办法。

如果我们采用这种测试优先的规则会怎样呢？我们可以改变这张影响图，使其进入一种良

性循环：测试优先在上面消极地作用于压力，而压力在下面消极地作用于测试优先。

如果我们先进行测试，我们的压力就会减小，这样使得我们更乐意去进行测试。有其他很多因素会增加压力，无论如何，测试应该处于一种良性循环当中，否则当压力增加到足够大的时候，测试将被抛弃。但是测试所带来的立竿见影的好处，即一种设计和控制规模方法，表明我们能够采用这种方法，即使在中等压力下我们也能继续运用它。

断言优先（Assert First）

我们什么时候应该写断言？试着一开始就编写断言。你难道不喜欢自身相似性（self-similarity）吗？

- 我们应该从哪儿开始构建一个系统？从我们对最终系统的描述开始。
- 我们应该从哪儿开始编写一项功能？从我们希望最终代码能够通过的测试开始。
- 我们应该从哪儿开始写一个测试？从测试完成时能够通过的断言开始。

是 Jim Newkirk 向我介绍的这种技术。当我使用断言优先进行测试时，我发现它有一种强大的简化效果。当你在写一个测试的时候，你是在同时解决好几个问题，即使你不再需要考虑具体的实现。

- 这一功能性属于哪一部分呢？是通过修改现有方法得到，还是在现有类中构造一个新的方法？或者是在一个新的地方来实现现有的方法？或是实现一个新的类来得到？
- 名字应该怎么起呢？
- 你怎样去检查结果的正确性？
- 什么是正确的结果？
- 这个测试是否需要其他的测试？

像我这么笨的人肯定不能同时解决所有这些问题。可以从上面的列表轻易分离出来的两个问题是“什么是正确的结果？”和“你怎样去检查结果的正确性？”

这儿有一个例子。假设我们希望通过一个套接字（socket）与另外一个系统通信。在通信完成以后，套接字必须被关闭，而且我们应当已经从中读出了发送过来的字符串 abc。

```
testCompleteTransaction() {  
    ...  
    assertTrue(reader.isClosed());  
    assertEquals("abc", reply.contents());  
}
```

reply 是从哪儿来的？当然是从套接字过来的：

```
testCompleteTransaction() {  
    ...  
    Buffer reply= reader.contents();  
    assertTrue(reader.isClosed());  
    assertEquals("abc", reply.contents());  
}
```


那么套接字从哪儿来的呢？是我们通过连接到一个服务器创建的：

```
testCompleteTransaction() {  
    ...  
    Socket reader= Socket("localhost", defaultPort());  
    Buffer reply= reader.contents();  
    assertTrue(reader.isClosed());  
    assertEquals("abc", reply.contents());  
}
```

但在此之前，我们需要打开一个服务器：

```
testCompleteTransaction() {  
    Server writer= Server(defaultPort(), "abc");  
    Socket reader= Socket("localhost", defaultPort());  
    Buffer reply= reader.contents();  
    assertTrue(reader.isClosed());  
    assertEquals("abc", reply.contents());  
}
```

现在也许我们必须根据它们的实际用途来调整它们的名字了。但我们已经通过很多微小的步骤构建出了测试的轮廓，在很短的时间内根据反馈做出每一个决定。

测试数据（Test Data）

我们将在测试优先的测试里面使用什么数据呢？使用那种容易让人理解的数据。你是在给大家写测试。不要为了编排数据而编排数据。如果数据必须取不同的值，那么这种取法必须要有意义。如果 1 和 2 之间没有概念上的差别，那么就使用 1。

测试数据并不是增加自信的通行证，如果你的系统需要处理多项输入，那么你的测试就必须能够反映出多项输入。然而，如果包含三条数据项的列表就能指明你的设计和实现构思的话就没必要使用包含十条输入数据项的列表。

在测试数据中的一个诀窍是永远不要用同一个常量来表达多种意思。如果我在测试一个 `plus()` 方法，可以尝试着测试 `2+2`，因为这是加法的经典例子，或者测试 `1+1`，因为它更加简单。如果我们的参数顺序颠倒了怎么办呢？（好了，行了，就 `plus()` 而言这个不成问题，但是你的思路是正确的。）比方说，如果我们将 2 作为第一个参数，那么我们应该用 3 作为第二个参数。（在过去，当启动一个新的 Smalltalk 虚拟机时，`3+4` 就是一个测试用例的分水岭。）

另一种测试数据是真实数据（Realistic Data），是你在真实世界里使用的数据。真实数据在下列这些情况下是非常有用的：

- 在使用根据实际运行所采集到的外部事件序列来测试实时系统时
- 在将目前系统的输出与以前系统的输出进行匹配时（平行测试）
- 当你对某种仿真系统进行重构而期望在完成时得到完全相同的结果时，特别是在浮点精度有可能存在问题的时候

显然数据 (Evident Data)

你如何表达数据的意图？让测试自身包含预期的和实际的结果，并且努力使它们之间的关系明显化。你不仅仅是给电脑写测试，还要让其他人容易读懂。后来的人会问自己：“这家伙到底在想什么啊？”你应该留下尽可能多的线索，尤其是那个倒霉的读者就是你的时候。

这儿有个例子。如果我们将一种货币兑换成另外一种货币时，每笔交易要收取 1.5% 的手续费。如果 USD 和 GBP 的兑换率为 2:1，那么如果我们要兑换 100 美元，我们将得到 50 GBP-1.5%=49.25 GBP。我们应该像这样写测试：

```
Bank bank= new Bank();
bank.addRate("USD", "GBP", STANDARD_RATE);
bank.commission(STANDARD_COMMISSION);
Money result= bank.convert(new Note(100, "USD"), "GBP");
assertEquals(new Note(49.25, "GBP"), result);
```

或者我们可以努力让计算更明白一些：

```
Bank bank= new Bank();
bank.addRate("USD", "GBP", 2);
bank.commission(0.015);
Money result= bank.convert(new Note(100, "USD"), "GBP");
assertEquals(new Note(100 / 2 * (1 - 0.015), "GBP"), result);
```

我能够读懂这个测试，能够看到输入的数字和用来计算预期结果的数字之间的关系。

显然数据有利的一面是它使编程更容易。只要我们在断言中把表达式写清楚了，就知道要写什么程序了。我们必须通过某种方法让程序计算一道除法和一道乘法。我们甚至可以用伪实现 (Fake It) 来逐步发现这一操作的归属。

显然数据看上去是“不要在代码中直接使用魔数 (magic number)”这条规则的例外。就单个方法而言，这些数字间的关系是很明显的。如果我有已经定义好的符号常量，那么，我将会用符号形式。

26

不可运行状态模式

这些模式是关于你什么时候写测试，在哪里写测试，以及什么时候停止写测试的。

一步测试（One Step Test）

你将从计划列表里面选择编写哪一个测试呢？选择那个具有指导意义而且你有把握实现的测试。

每个测试都应当代表迈向总体目标的一步。看看下面这个测试列表（Test List），那么下一步你会选择哪个测试呢？

Plus

Minus

Times

Divide

Plus like

Equals

Equals null

Null exchange

Exchange one currency

Exchange two currencies

Cross rate

这没有正确答案。我以前从来没有实现过这些对象，或许你的经验更丰富一些，我这里的一步将只是你一步的十分之一。如果你觉得这个清单里的每个测试都不能代表一步的话，那么可以增加一些能够导向其中这些条目的新测试。

当我检查测试列表的时候，我是这样想的：“这个做起来太容易了，那个怎么做不明摆着的吗，这个我不知道怎么办，啊，这个我能做”，这样最后一个测试就成了我要实现的下一个测试。表面上看不出这个测试怎么做，但是我有信心让它运行起来。

从类似这样的测试逐步构造出来的程序看上去似乎是用自顶向下的方法编写的,因为你是从编写一个代表整个计算过程的简单用例的测试开始的。一个根据测试程序构造产生的程序看上去也像用自底向上的方法来完成的,因为你是从小块内容开始,然后逐步聚集成越来越大的程序的。

其实自顶向下也好,自底向上也好,都没有真正有助于这一过程的描述。首先,这种垂直的描述是对程序随着时间而改变的一种过于简单的观察。程序的构造过程暗含了某种自相似的反馈环,环境影响程序而且程序反过来也影响环境。其次,如果你需要一种指导性的描述的话,那么从已知到未知(known-to-unknown)是个有助于理解的描述。从已知到未知意指我们已经掌握了一些知识和经验,我们要根据它们来完成下一步的工作,而且我们期望在开发过程中进行学习。把这两部分合起来,然后我们就可以让程序从已知到未知逐步增长。

启动测试 (Starter Test)

我们应该从哪个测试开始?从测试某个实质上不做任何工作的操作开始。

对一个新的操作,你必须搞清楚的第一个问题是“这个操作隶属于哪儿?”除非你搞清楚了这个问题,不然你是不知道该为这个测试写些什么的。在一次只解决一个问题的原则下,你如何能够只回答这个问题而不想其他的呢?

如果你一开始就编写一个真实的测试,那么你会发现自己同时要解决一连串的问题:

- 这个操作隶属于哪儿?
- 正确的输入是什么?
- 有了这些输入以后正确的输出是什么?

从一个真实的测试开始将使你在很长时间内得不到反馈。不可运行/可运行/重构,不可运行/可运行/重构,你肯定希望这样的循环只花费几分钟的时间。

你可以通过选择极易发现的输入和输出来缩短这一循环。举个例子,在一个有关极限编程的新闻组里面,有人发帖子询问如何通过测试优先的方式来做一个 Polygon Reducer。输入是由许多多边形组成的网格,而输出也是许多多边形的网格,精确覆盖了相同的表面,但使用尽可能少的多边形。“我如何用测试来驱动这个问题?难道做测试还需要去阅读哲学博士的论文吗?”

启动测试提供了一种答案:

- 输出应当与输入一样。一些多边形的配置是已经规格化了的,不能够做更多的缩减。
- 输入应该尽可能的少,比如只有一个多边形,甚至是一个空的多边形列表。

我的启动测试像下面这样:

```
Reducer r= new Reducer(new Polygon());  
assertEquals(0, reducer.result().npoints);
```

好了,第一个测试开始运行了。现在我们着手完成列表里面剩下的其他测试……

一步测试适用于启动测试。选择一个有指导意义而且肯定能够很快工作的启动测试。如果一个应用是你第 n 次实现了，那么就挑选一个有一两步操作的测试。你有自信能够让它工作起来。如果你正在实现的东西困难又复杂，而且还是第一次，那么你需要的是立刻得到一点点激励。

与下面的测试相比，我发现我的启动测试通常比较高阶，更像是一个应用测试。举个例子，我经常采用的测试驱动开发范例是一个简单的基于套接字的服务器。它的启动测试是这样的：

```
StartServer
Socket= new Socket
Message= "hello"
Socket.write(message)
AssertEquals(message, socket.read)
```

其他的测试就单独写在服务器里面了，“假设我们收到一个像这样的字符串……”

说明测试（Explanation Test）

如何拓展自动测试呢？我们利用测试来请求及提供说明解释。

如果自己是团队中惟一采用 TDD 进行开发的人，那么这将是令人很不耐烦的。但不久，你就会注意到测试过的代码中出现的集成问题和缺陷报告都减少了，你的设计也将变得简单且易于解释。甚至会出现大家对测试和测试优先产生极大的热情的现象。

要小心呵护这些新近转向 TDD 的人的热情。没有什么比强迫更能够阻止测试驱动开发的推广了，如果你是管理者或团队的领导，那么你不能强迫任何人改变他的工作方式。

那我们能做些什么呢？我们可以采用这样一种简单的方法，那就是开始让大家通过测试来请求说明：“让我们看看我是不是理解了你所说的。比如说，我有一个这样的 Foo 和一个那样的 Bar，那么结果应该是 76？”一种伴随技术（companion technique）就是渐渐地利用测试来给出说明：“它是这么工作的，当我有一个这样的 Foo 和一个那样的 Bar 时，答案是 76。如果我有一个那样的 Foo 和一个这样的 Bar，那么我认为结果应该是 67。”

你可以在更高的抽象层次上这样做。如果有人向你说明一个序列图（sequence diagram），那么你可以向他请示自己可否将这张图转换成某种更为熟悉的记法。接着你键入一个包含图中所有外部可见对象与消息的测试用例。

学习测试（Learning Test）^①

你什么时候为外部软件编写测试呢？在你第一次准备使用这个包中的某项新功能的时候。假设我们准备基于 Java 的 MIDP 库开发一些东西。我们希望在 RecordStore 中储存一些数

① 感谢 Jim Newkirk 和 Laurent Bossavit 分别提供了这个模式。

据而且能够检索到这些数据。我们直接编写代码，然后期待这些代码能够工作吗？这算是一种开发方法。

另一种方法是在发现我们将使用一个新类里的一种新的方法时，不直接用它来编写程序，而是编写一个小测试来验证这个 API 的工作是否符合我们的期望。所以，我们可以这样写：

```
RecordStore store;

public void setUp() {
    store= RecordStore.openRecordStore("testing", true);
}

public void tearDown() {
    RecordStore.deleteRecordStore("testing");
}

public void testStore() {
    int id= store.addRecord(new byte[] {5, 6}, 0, 2);
    assertEquals(2, store.getRecordSize(id));
    byte[] buffer= new byte[2];
    assertEquals(2, store.getRecord(id, buffer, 0));
    assertEquals(5, buffer[0]);
    assertEquals(6, buffer[1]);
}
```

如果我们对这个 API 的理解是正确的，那么这个测试应该能够一次通过。

Jim Newkirk 根据某个项目写过一份报告。在这个项目中，编写学习测试乃是例行工作。在新发行的软件包到手后，首先是运行测试（如果有必要的话还要进行修改）。如果测试不能运行，那么运行应用也就没有什么意义了，因为它肯定运行不了。一旦测试能够运行了，那么应用随时都能运行。

另外的测试

如何才能让技术讨论不跑题呢？当出现某种与当前讨论话题并不直接相关的想法时，那么就在列表里面增加一个测试然后重新回到论题上来。

我喜欢漫无边际的讨论（你现在已经读了这本书的大半了，所以极有可能已经自己得出了这个结论）。在开会过程中保持一种严肃的对话气氛很容易扼杀一些聪明的想法。你从这跳到那儿，然后又跳到另外的地方，我们怎么讨论到这儿了呢？管他呢，这样讨论的感觉真棒。

有些时候编程依赖于重要的突破。不过，绝大多数编程还是按部就班的。我有十个单项要实现，到第四个单项的时候我想歇一歇，我的一种逃避工作的办法就是重新加入到乱哄哄的讨论中去（也许担忧也会如影随形）。

整日没有任何成效的日子教会了我有时最好还是集中注意力为好。当我有这样的感觉时，我虽然注重那些新的想法，但是我不会让它们转移我的注意力。我把它们写进列表里，然后回到我正在进行的工作上来。

回归测试 (Regression Test)

当一个错误被发现时,你想做的第一件事是什么呢?写一个尽可能小的会失败的测试,一旦运行,我们就对其加以修缮。

回归测试是这样的一种测试,试想当你最初能够预见一切情况的时候,你会怎么编写代码。每次当你需要编写回归测试的时候,思考一下你如何才能当时编写出这样的测试。

你同样能够通过对整个应用的测试来学到有用的东西。针对应用的回归测试会给你的用户一个告诉你问题出在哪里而他们期望的又是什么的机会。小规模回归测试是一种改进测试的方法。比如错误报告说出现了奇怪的很大的负数。那么你所得到的教训就是在写测试列表的时候需要把整数溢出测试加进去。

在你能够轻松地隔离缺陷之前你也许需要重构系统,在这个事例中系统用缺陷告诉你,“你还没有把我完全设计好。”

休息

当你感到累的时候怎么办?休息一下。

喝上一杯,散散步,打个盹儿。抛开那些刚才所做的情绪化的决定和键入的代码。

通常情况下,这样的放松会使你大脑的思维得到解放。说不定你会突然想到解决办法而高兴得跳起来,“我还没有将参数倒过来试试呢!”总之休息一下。给自己几分钟时间,你的点子是不会跑掉的。

如果你并没有什么点子,那么回想一下这一轮编程的目标。它们仍然是真实可达的吗?或者你是不是应该选择一个新的目标了?我们所要完成的东西是不是不可能完成了?如果是这样,那么对团队意味着什么呢?

Dave Ungar 称之为淋浴法 (Shower Methodology)。如果你知道要敲入什么代码,那么把它敲进去。如果你不知道要敲入什么,那么去洗个淋浴,待在水龙头下面,直到自己知道要写什么为止。许多团队如果听从他的建议的话,那么团队会更加活泼,更加高产,而且健康向上。TDD 是对 Ungar 的淋浴法的一种改进,如果你知道要写什么,那么就键入这个明显的实现。如果你不知道要写什么代码,那么做一个伪实现。如果正确的设计仍没有清晰地表达出来,那么实施三角法。如果你还不知道要写什么,那你可以去冲个澡了。

图 26-1 显示了休息和工作的动态关系。你感觉累了,所以你意识到自己已经疲惫的能力也有所下降,所以你继续工作而且越发感觉到累。

跳出这种循环的方法是引入外界的其他元素。

- 如果是几小时一轮的话,就在键盘旁边放上一瓶水,这样生理上的需要就会驱使你规律地休息一下。
- 如果是每天一轮的话,那么几小时的艰苦工作之后的放松有助于你停下来睡一会儿。

- 如果是每周一轮的话，那么周末的放松有助于你卸下包袱（我妻子说我最好的想法总是在周五的晚上产生）。
- 如果是每年一轮的话，那么强制假期的政策会帮助你彻底放松。法国在这方面做得很好——两个星期的假期远远不够。你用第一个星期减轻工作压力，第二个星期又要准备回到工作中去了。因此，要想在一年中剩下的其他时间精力充沛，还是应该休息三个星期，最好四个星期。

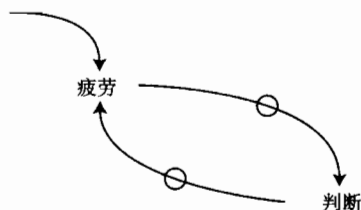


图 26-1 疲劳影响判断而判断反过来影响疲劳

休息也会产生副作用。有时当遇到一个棘手的问题时，我们要做的就是咬咬牙挺过去。但是，编程文化受强者文化意识影响很深——“我的健康会每况愈下，我会和家庭越来越疏远，如果可能的话我会自杀的”——我不会就这样的想法提供任何建议。当你发现你对咖啡因上瘾而工作没有进展的话，那么你不应该休息这么多次，间或出去散散步就行了。

重新开始

当你感到迷失方向时怎么办？扔掉原来的代码，重新开始。

你迷失方向了。你已经休息过了，尝试了一些休息的手段，休假也休了，但是仍然摸不到方向。那些一个钟头前还运行得好好的代码现在搞得一团糟，实在想不出让下一段测试代码运行的办法，而且你必须要实现二十多个这样的测试例程。

我写这本书的时候这样的情况发生了多次。我把代码搞得有点混乱。“但是我不得不写完这本书，孩子们嗷嗷待哺，那些要账的人又在敲门了。”我的本能反应会使我梳理代码，使它能够继续下去。停下来吃一些点心，然后重新开始会更有意义。当我点击忽略的时候，我就扔掉了 25 页的手稿，因为这些是基于一个有明显错误的语法决定的。

我最喜欢举的一个重新开始的例子是 Tim Mackinnon 告诉我的。他在面试一个人的时候叫她对他写的程序进行连调。最后，他们实现了好几个新的测试程序。但是，一天就这样结束了，当他们完成的时候感到很疲倦，所以他们放弃了以前做的工作。

在连调程序时使用交换合作伙伴的方法是一个激励我们重新开始的好方法。你会花上好几分钟时间向你的新合作伙伴解释这个复杂的程序，而你的新合作伙伴对于你犯的错误根本不会深入调查，他会拿过鼠标，指着删除按钮说：“非常抱歉，我们这样重新开始怎么样？”

便宜的桌子，舒适的椅子

为了测试驱动开发你需要怎样的物理条件呢？买个舒适的椅子，至于其他办公用具能省点就省点。

如果你坐得不舒服的话，是编不好程序的。当然，一个只愿意每个月给你的开发小组 100000 美元的公司是不会花 10000 美元来买象样点儿的椅子的。

我的解决办法是用便宜而难看的折叠桌子来放电脑，但是买我能找到的最好的椅子。我的桌子有很多空间，而且还能收拾出更多空间出来，我会神清气爽，准备好下午和早上的编程。

调程序的时候尽量把自己搞得舒服一点。把桌面上清理一下，这样你可以把键盘挪来挪去，而且每个合作伙伴都能在你的键盘前坐得舒适一点。我最喜欢讲授的一个诀窍就是工作的时候把键盘放到适合你输入的地方。

Manfred Lange 指出资源分配也应考虑电脑硬件。用便宜、老的、慢的电脑来收发个人邮件、上网，用最新的电脑作为共享开发工作站。

27

测试模式

以下模式是编写测试代码的技术细节。

子测试 (Child Test)

如果测试例程太大，你又如何能让它运行起来呢？写出能够代表部分大的测试例程的小测试例程，让这些小的测试例程运行通过。然后再处理大的测试例程。

不可运行/可运行/重构的节奏对于持续的成功非常重要，所以当你处在失去这种节奏的危险边缘时，付出额外的努力保持这种节奏是值得的。我经常会遇到这种情况，特别是当我写的测试突然要求几处变化以正常工作时。甚至十分钟的不可运行状态也让我心惊肉跳。

当我写的测试太大时，我首先试图回答这几个问题：为什么它会这么大？我应该采取什么不同的方法使它变得小一些？现在我感觉如何？

问题的抽象已经完成，我删掉麻烦的测试代码，重新开始。“让这三个东西同时工作起来太难了。如果我分别使 A、B 和 C 工作起来，那么使整个东西工作起来只是小菜一碟。”有时我真地删掉测试代码，有时我只是将它的名字改为用 x 开始，这样它就不会被运行了。（我是否该告诉你一个秘密？有时我甚至不会删除麻烦的测试。我同时使用两个，“做两遍？”，对一个问题做两分钟的不完整测试，同时做子测试。这样做可能会犯错。两个不完整测试可能和我旧的错的版本是一样的。）

自己试试这两种方法。看看当你有两个不完整测试时是否会感到不同，是否会编出不同的代码，并做出适当的反应。

模拟对象 (Mock Object)

如何测试一个依赖于昂贵且复杂的资源的对象？创建一个这些资源的模拟版本。

关于模拟对象有很多值得一看的材料^①，而本书只是作为一个介绍。典型的例子是数据库。数据库的建立很耗时间，难于保持一致，而且如果数据库位于远程的服务器上，这将使你的测试被绑定于网络中的物理地址。数据库也是开发过程中错误产生的温床。

解决之道是在大多数时间不使用真正的数据库。大多数测试写一个像数据库一样的对象，但它仅仅驻留在内存中。

```
public void testOrderLookup() {
    Database db= new MockDatabase();
    db.expectQuery("select order_no from Order where cust_no is 123");
    db.returnResult(new String[] {"Order 2" ,"Order 3"});
    ...
}
```

如果 `MockDatabase` 没有收到它希望的询问，将产生一个异常。如果询问正确，它将返回一些看起来像结果集的东西，其实这些是由字符串常量组成的。

除了性能和可靠性之外，模拟对象的另一个优点是可读性。你可以阅读前一个测试程序。如果使用的全是现实数据的数据库，那么当你看到一个询问返回的结果是 14 时，你不知道为什么 14 是正确的结果。

如果要使用模拟对象，那么就不能在全局变量中存储开销昂贵的资源[即使仿真成单例模式(Singleton)也不行]。如果这样做，你将不得不把全局作为一个模拟对象，运行测试程序，一定要在完成后再重设全局变量。

我有好多次都对这个限制感到愤怒。Massimo Arnoldi 和我做过一些依赖于存储在全局变量中的一系列交换速率的编码工作。每个测试都需要数据的不同子集，有时需要不同的交换速率。在做了很多使全局变量工作的尝试后，一天早上(对我来说早上通常是做出一些勇敢的设计决定的时候)我们决定，只在我们需要的地方使用 `Exchange`。我们认为我们将修改上百个方法。最后，我们对十或十五个方法添加了参数，并且用这种方法修改了设计的其他方面。

模拟对象鼓励你仔细思考个个对象的可见性，减少设计中的耦合。这也给项目增添了风险——如果模拟对象和现实对象的行为不一样怎么办？可以通过同样能够对实际对象适用的一系列测试来测试模拟对象，从而减少这种危险。

自分流 (Self Shunt)

如何测试对象间是否正常交互？让测试对象与测试用例而不是期望的对象进行交互。

假设我们想要动态更新测试用户界面的绿色状态条。如果将一个对象同测试结果连接起来，这个对象就可以被告知何时测试运行，何时测试失败，何时全套测试开始和结束，等等。当我们得知一个测试运行时，就可以更新界面。下面就是这样的测试：

^① 例如，参见 www.mockobjects.com。

ResultListenerTest

```
def testNotification(self):
    result= TestResult()
    listener= ResultListener()
    result.addListener(listener)
    WasRun("testMethod").run(result)
    assert 1 == listener.count
```

此测试需要一个对象对 Notification 进行计数:

ResultListener

```
class ResultListener:
    def __init__(self):
        self.count= 0
    def startTest(self):
        self.count= self.count + 1
```

等等, 对于 listener 来说我们为什么需要一个独立的对象呢? 我们只要使用测试用例自身就可以了。TestCase 自己就可以是一个模拟对象。

ResultListenerTest

```
def testNotification(self):
    self.count= 0
    result= TestResult()
    result.addListener(self)
    WasRun("testMethod").run(result)
    assert 1 == self.count
def startTest(self):
    self.count= self.count + 1
```

使用了自分流写出的测试比没有使用的可读性高。前述的测试就是个很好的例子。count 的初始值为 0, 然后变为 1。可以在测试程序中看到这个语句序列。它是如何变成 1 的呢? 因为 startTest() 被调用。startTest() 是如何被调用的呢? 运行此测试程序时就会被调用。这是另一个对称的例子——测试方法的第二个版本中, 同一个地方 count 有两个值, 而在第一个版本中, count 在一个类中被置 0, 在另一个类中被加 1。

自分流要求使用提取接口 (Extract Interface) 来实现接口。你将决定到底提取接口简单还是对已有的类进行黑匣测试简单。但是, 我注意到使用分流的接口提取将会使其子实现更加迅速。

使用自分流测试的结果是, 对 Java 的测试将实现各种各样奇异的接口。在优化类型的语言中, 测试例程的类只需实现实际运行测试时所需的操作。但是, 在 Java 中需要实现接口定义的所有操作, 有可能大多数这样的实现为空。因此, 你可能希望接口越少越好。这些实现可能返回合理的值, 也有可能抛出一个异常, 这取决于你是否想知道一个不希望的操作被调用。

日志字符串 (Log String)

如何测试才能使消息调用序列是正确的? 将日志保存在字符串中, 当调用一个消息时, 就

向字符串尾追加相应的信息。

以 xUnit 服务为例。在模板方法（Template Method）中我们调用 `setUp()`，一个测试方法，和 `tearDown()`。通过实现这个方法在字符串中记录我们所调用的方法，这样各测试程序读起来非常赏心悦目。

```
def testTemplateMethod(self):
    test= WasRun("testMethod")
    result= TestResult()
    test.run(result)
    assert("setUp testMethod tearDown " == test.log)
```

实现也很简单：

WasRun

```
def setUp(self):
    self.log= "setUp "
def testMethod(self):
    self.log= self.log + "testMethod "
def tearDown(self):
    self.log= self.log + "tearDown "
```

日志字符串在你实现 `Observer` 或是希望以一定的序列得到通知时很有帮助。如果你希望得到特定的通知，但不关心执行顺序，那么可以使用字符串集合，并在断言处使用集合比较。

日志字符串和自分流搭配使用效果很好。通过添加日志并返回合理值可以在分流的接口中实现日志字符串的方法。

清扫测试死角（Crash Test Dummy）

如何测试到不大可能被调用到的错误代码呢？使用一种特殊的对象调用它，这个对象抛出一个异常而不做任何实际工作。

没有被测试到的代码是不会正常工作的。这看起来是安全的假设。那么你怎么对付那些奇怪的错误情况呢？是否也要测试它们呢？只要我想要它们正常工作，就要测试。

假设我们想要测试当文件系统满了以后我们的应用会发生什么。我们可以花费很多时间创建许多大文件而填满整个文件系统，还可以采用伪实现（Fake It）。伪实现听起来似乎不那么正规，是不是？我们可以模拟这种情况的发生。

下面是我们清扫文件测试死角的代码：

```
private class FullFile extends File {
    public FullFile(String path) {
        super(path);
    }
    public boolean createNewFile() throws IOException {
        throw new IOException();
    }
}
```

现在我们可以编写针对期望异常（Expected Exception）的测试了：

```
public void testFileSystemError() {
    File f= new FullFile("foo");
    try {
        saveAs(f);
        fail();
    } catch (IOException e) {
    }
}
```

清扫测试死角就像模拟对象（Mock Object），只是你不用模仿整个对象。Java 的匿名内部类（inner class）会摧毁我们希望执行模拟错误的方法。我们可以在测试用例中重载想要的方法，使得测试用例更易读：

```
public void testFileSystemError() {
    File f= new File("foo") {
        public boolean createNewFile() throws IOException {
            throw new IOException();
        }
    };
    try {
        saveAs(f);
        fail();
    } catch (IOException e) {
    }
}
```

不完整测试（Broken Test）

当你独自编程时，如何能离开编程工作一段时间呢？使剩下的测试不完整。

Richard Gabriel 交给我一个小窍门。当你又回到电脑前，看着半截句子，必须能够回想起当时你写这一段代码时想的是什么。一旦你接上了头绪，你就会接着写下去。如果没有这样的句子留着等待完成，那么你每次都要花费几分钟时间四处寻找接下来的工作是什么，然后你会想起当时脑子里的情景，最后才开始键入未完成的内容，而这样连续多次下来浪费的时间是很多的。

我在个人的项目中尝试过类似的技术，效果很好。独自写一段测试用例，运行它并确保它不能运行。当你接着编码时，你会有明显的地方重新开始。明显的标记会帮助你回忆起你原来的想法，使得测试工作加快不少，你就会在成功的道路上越跑越快。

我曾经觉得留着一段不完整的测试过夜会给我带来麻烦。但是这并没有带来麻烦，我想是因为我知道程序还没有完成。一段不完整的测试用例不会使程序显得更加不完整，它只是标志了程序的完成状态。离开几星期以后还能迅速找回头绪进行快速开发是很值得的，尽管离开时留着红色状态条会有点痛苦。

提交前保证所有测试运行通过

当你在团队中编程时如何结束一段编码工作？让所有的测试运行起来。

当你对团队中的其他人负有责任时，情况就完全不同了。当你开始进行团队编程工作时，你不会知道代码究竟是什么样子。你需要以一种自信和确定的方式开始工作。因此，在你提交你的代码前，确保所有的测试都已通过。

在你提交时运行的测试套件可能比你在开发过程中运行的某个测试程序更具广泛意义。（不要放弃运行整个测试程序，除非已经慢得不行了。）在你试图提交时，你可能会发现集成测试套件中有很多未通过的测试，怎么办呢？

最简单的办法是把做过的工作推倒重来。这些未通过的测试强烈地表明你对于你刚刚编码的东西没有充分地了解。如果开发团队采用了这种方法，将会有频繁提交的趋势，因为第一个提交的人不会冒丢失任何工作的风险。频繁提交可能是件好事。

一些比较自由的方法使得你有机会改正错误，重新尝试。不要支配集成的资源，你应该学会几分钟后就放弃，并且重新开始。或许这根本不用说，但是我还是要讲一下，注释掉一些测试代码使套件通过是要严格禁止的。

28

可运行模式

一旦出现未能通过的测试，你就必须去处理。如果你把不可运行状态当作是尽快处理的一个条件，那么你会发现很快能得到可运行状态的。使用这些模式使代码通过测试（即使那些结果是你最不愿意看到的）。

伪实现（直到你成功）

测试不能通过时首先应该执行什么？返回一个常量。一旦你能使测试运行起来，那个常量就会逐渐转换成用变量表示的表达式。

在 xUnit 的实现中有一个简单的例子：

```
return "1 run, 0 failed"
```

变成：

```
return "%d run, 0 failed" % self.runCount
```

变成：

```
return "%d run, %d failed" % (self.runCount, self.failureCount)
```

伪实现（Fake It）就像登山时在头的上方钉一个登山用的钢锥。实际上你没有到达那儿（测试到了那个地方，但代码是错误的）。但是，当你到达那儿时，你就知道你是安全的（测试还将继续进行下去）。

伪实现事实上会惹恼一些人。为什么要去做一些自己明知道是错误的事情呢？因为有一些东西运行起来总比没有什么可以运行要好，特别是在你通过测试进行证明的时候。Peter Hansen 曾讲过这样一个故事：

这些事情仿佛就发生在昨天。作为刚接触测试驱动开发的两个新手，我和我的搭档坚定地依照规则行事，为了使测试更快运行通过而使用了各种“丑陋”的做法。在这一过程中，我们意识到当初没有正确地实现测试，于是我们又退回去重新修改，使代码再一次运行起来。再次运行时，第一次运行的代码已经不复存在了。我们彼此对视了一会儿，说道：“哈……请看那儿！”因为那种方法

教给了我们一些未知的东西。

虽然我不知道他们怎样从一次伪实现中得知他们的测试写错了,但是可以肯定他们一定很高兴没有花时间寻找真正的解决方法。

使伪实现强有力的因素有两个:

- 心理因素——得到可运行状态与得到不可运行状态的感觉截然不同。当状态是可运行的时候,你就知道了你所处的位置,就能从那儿充满自信地开始重构。
- 范围控制——程序员们往往擅长于想像各种各样将来的问题。以一个具体的例子为起点并在那里开始归纳,就能使你免于过早地被无关紧要的担忧所困惑。因为聚焦于一点,你就能更好地解决直接的问题。当你转去执行下一个测试用例时,你也就能聚焦在那一个上,并且确定前一个测试可行。

那么伪实现是否违反了不产生没必要代码的规则呢?我不这样认为,因为在重构的阶段你就删除了测试用例和代码之间的重复部分。当我写下^①

```
assertEquals(new MyDate("28.2.02"), new MyDate("1.3.02").yesterday());
```

MyDate

```
public MyDate yesterday() {  
    return new MyDate("28.2.02");  
}
```

时,在测试和代码中就有了重复。可以修改如下:

MyDate

```
public MyDate yesterday() {  
    return new MyDate(new MyDate("31.3.02").days()-1);  
}
```

但这仍然有重复设计。于是,我就消除了其中的数据重复(在我的测试意图中, **this** 就相当于 **MyDate("31.3.02")**),修改如下:

MyDate

```
public MyDate yesterday() {  
    return new MyDate(this.days()-1);  
}
```

你能使用三角法 (Triangulation) 的方法,直到你感到厌倦之后开始使用伪实现或者显实现 (Obvious Implementation)。诡辩并不能使所有的人都信服上述说法。

当我使用伪实现时,就会联想到驾驶汽车长途旅行,孩子们坐在后排。我编写了第一个测试,使用一些“丑陋”的方法使测试很快就运行通过,这时:“不要让我停车写另一个测试。如果必须得停在路边的话,你会后悔的。”

“好了,行了,爸爸,我会整理代码。你没必要发这么大火。”

^① 感谢 Dierk König 提供了这个例子。

三角法 (Triangulation)

怎样可以更适当地利用测试推动抽象呢？只有当你有两个或两个以上的例子时，你才能进行抽象。

有这样一种情况。假设要写一个返回两个整数的和的函数，我们这样写代码：

```
public void testSum() {
    assertEquals(4, plus(3, 1));
}

private int plus(int augend, int addend) {
    return 4;
}
```

如果我们用三角法做正确的设计，我们就必须写如下代码：

```
public void testSum() {
    assertEquals(4, plus(3, 1));
    assertEquals(7, plus(3,4));
}
```

当我们有另一个例子时，我们就能抽象出 `plus()` 的实现：

```
private int plus(int augend, int addend) {
    return augend + addend;
}
```

三角法吸引人的地方在于它的规则看起来十分清楚。而伪实现的规则则依赖于我们在测试用例和假装实现之间对于重复的认知来推动抽象，看起来就有点模糊且必须经过解释。尽管三角法的规则看起来简单，却产生了一个无限的循环。一旦我们有两个断言，且抽象出了 `plus()` 的正确的实现，我们就能当场删掉其中一个相对于另一个而言完全冗余的断言。然而，如果我们那样做了，可以使 `plus()` 的实现简化成恰好只返回一个常量，却要求我们再添加一个断言。

只有当我实在是不能确定关于计算的正确抽象时，我才使用三角法的方法。否则，我就依赖于显明实现或者伪实现。

显明实现 (Obvious Implementation)

怎样实现简单的操作呢？直接实现。

伪实现和三角法都是很细微的步骤。有时你确信知道怎样实现一个操作，那就干吧。举个例子，我真的用伪实现来实现像 `plus()` 这样简单的操作吗？通常不。我只是在显明实现方式下直接敲进去。当我意外地进入不可运行状态时，我才进一步细化。

当然，如果使用伪实现和三角法半途而废，那么这些方法也不会有优点。如果你知道你要写些什么，并且能够很快地完成，那么赶快动手吧。然而如果只使用显明实现的话，你就是在

苛求自身的完美^①。从心理学角度，这是危险的一步。如果你写的程序不是使测试正常通过的最简单的修改办法怎么办呢？如果你的搭档给了你一个更简单的方法呢？那你就失败了！你的世界一下子崩溃了！你完了。你没话说了吧。

既要保证它们正常运行又要编出整洁的代码一下子还真让人受不了。要想尽快做到的话，还是让它们先能正常运行吧，这样才能从容地得到整洁的代码。

注意记录在使用显明实现时你有多少次会进入不可运行状态。我在显明实现下写程序的过程中常常受阻，程序不能正常运行。如今我确信自己知道该写些什么，我也是如此做的。程序还是不能正常运行。那么现在该怎么办……这尤其是在出现下标越界错误和积极的或消极的错误时容易发生。

保持不可运行/可运行/重构的编程习惯。显明实现只是你的第二选择。但是当你的脑子和手反应不协调时你就得准备换换方式了。

从一到多（One to Many）

怎样实现一个作用于对象集合体的操作呢？首先在非集合体中实现，然后使之作用于集合体。

比如说，假设我们要写一个求数组和的函数，可以这样开始：

```
public void testSum() {  
    assertEquals(5, sum(5));  
}
```

```
private int sum(int value) {  
    return value;  
}
```

（我是在 `TestCase` 类中实现 `sum()` 的，这样可以避免为同一种方法写一个新类。）

接着要测试 `sum(new int[] {5,7})`，我们先传一个数组参数给 `sum()`：

```
public void testSum() {  
    assertEquals(5, sum(5, new int[] {5}));  
}
```

```
private int sum(int value, int[] values) {  
    return value;  
}
```

我们可以把这一步当作一个隔离变化（*Isolate Change*）的例子来看。一旦我们向测试用例中传递了参数，就可以在不影响测试用例的情况下自由地改变实现。

现在我们用集合体来代替单个的值：

^① 感谢 Laurent Bossavit 对此发表的高见。

```
private int sum(int value, int[] values) {  
    int sum= 0;  
    for (int i= 0; i<values.length; i++)  
        sum += values[i];  
    return sum;  
}
```

接着就删除没用的单个参数:

```
public void testSum() {  
    assertEquals(5, sum(new int[] {5}));  
}
```

```
private int sum(int[] values) {  
    int sum= 0;  
    for (int i= 0; i<values.length; i++)  
        sum += values[i];  
    return sum;  
}
```

前述步骤也是一个隔离变化的例子, 其中我们修改了代码, 这样就能在不影响代码的情况下改变测试用例。现在我们可以按计划丰富测试用例了:

```
public void testSum() {  
    assertEquals(12, sum(new int[] {5, 7}));  
}
```

29

xUnit 模式

本章所讲的是 xUnit 家族中的一种测试框架的应用模式。

断言 (Assertion)

怎样检验测试是否正确工作呢？写一个布尔表达式对代码是否工作自动作出判断。

要想使测试完全自动化，对结果的判断必须去除任何人工成分。我们只需要按一个键，就能让所有验证代码是否正常工作的必要的决定都由计算机作出。这就意味着：

- 这个决定必须是布尔型的——`true` 通常表示一切正常，而 `false` 则表示一些意外的事情发生了。
- 计算机必须能通过调用一些由 `assert()` 派生出来的方法来检查布尔型的状态。

我曾经看过这样的一个断言：`assertTrue(rectangle.area() != 0)`。返回任何非空的值就能满足这一测试，因此它并不实用。应该更具体一点。如果那个面积必须是50，那么就说它必须是50：`assertTrue(rectangle.area() == 50)`。很多xUnit实现对测试等式有专门的断言。测试等式是十分常见的，并且，如果你知道你正在测试等式，你就能写出丰富的出错信息。期望值通常排在最前面，因而在JUnit中我们可以这样写：`assertEquals(50, rectangle.area())`。

把对象当作黑盒看待是困难的。如果我们有一个Contract，其中包含一个既可以是Offered也可以是Running类型的实例的Status，那么我们可能基于自己所期望的实现编写出如下的测试：

```
Contract contract= new Contract(); // Offered status by default
contract.begin(); // Changes status to Running
assertEquals(Running.class, contract.status.class);
```

这个测试过分依赖于status的当前实现。即使status的表示变换成为一个布尔值，这个测试也应该仍然能够运行通过。也许一旦status转变为Running，即有可能请求一个实际的开始日期：

```
assertEquals(...,contract.startDate()); // Throws an exception if the status is Offered
```

有一种潮流，坚持所有的测试只能使用公共的规程来写，我知道我正在逆流而行。更有甚

者，有一种在JUnit基础上扩展形成的软件包JXUnit，它允许测试变量的值，甚至是那些被声明为私有的。

希望使用白盒测试不是一种测试问题，而是一种设计问题。每当我想使用一个变量作为检验代码是否运行正确的一种方法时，我就获得了改进设计的机会。如果我因为害怕而屈服，仅仅检查变量，那么我就失去了这个改进设计的机会。那就是说，如果没有达到设计意图，就不可能再达到了。我将做的就是擦干眼泪，检查变量，并做好标记以便于在我才思敏捷的某一天返回，然后继续前进。

最初的SUnit（测试框架最初的Smalltalk版本）只有简单的断言。如果出现错误，就会弹出一个调试程序，你只需更正代码，然后就不用管了。因为Java的集成开发环境（IDE）并没有那么复杂，且由于构造基于Java的软件通常出现在批量处理环境中，所以，在断言中加入提示信息，使其在出错时就能显示出来，这是十分有意义的。

在JUnit中，就是利用头一个可选参数^①来获得这样的提示。测试运行时，如果你写了`assertTrue("Should be true", false)`，你就将看到一个这样的出错信息：“Assertion failed: Should be true”。这就提供了足够的信息使你找到代码中的出错源。一些程序开发小组沿用了惯例，要求所有的断言必须附加丰富的出错信息。两种方法都尝试一下，看看在出错信息上的投入是否值得。

固定设施（Fixture）

怎样创建几个测试都需要的通用对象呢？把测试中的局部变量转变成实例变量，重载`setUp()`并初始化这些变量。

如果我们想要删除模型代码中的重复余部分，是否也要从测试代码中将其删除呢？可能需要吧。

问题是这样的：通常在你写代码的时候，代码更多地用在适当的对象初始化上，而不是操纵对象和检查结果上。对对象进行的初始化对多个测试来说都是相同的（这些对象就是测试的固定设施，也被称为脚手架）。存在这样的重复是不好的，原因如下：

- 它要花费一定的时间去写，甚至去复制、粘贴，而我们期望的是快速编写出测试来。
- 如果我们想要手动修改一个接口，那么在若干个测试中我们都应该进行修改（这正是我们所说的重复）。

然而相同的重复也有其优点。当设置代码（初始化代码）在测试中与断言逻辑安放在一起时，从上到下都具有可读性。如果我们把设置代码分离出来，组成单独的方法，那么我们就必须记住这种方法被调用了，且在写余下的测试时记住该对象的特征。

xUnit中支持两种编写测试的风格。如果你并不期望读者能很轻松地记住固定设施对象，

① 可选参数一般出现在最后，但是将解释性字符串放在一开始的地方有助于增强可读性。

你就可以把固定设施创建代码与测试写在一起。然而，你也可以把公共的固定设施创建代码放入一个叫`setUp()`的方法中，在那里为测试中用到的对象建立实例变量。

这儿有一个例子，虽然过于简单，还不能显示出把公共设置代码分离出来的价值，但对这本书而言足够了。我们这样写：

EmptyRectangleTest

```
public void testEmpty() {
    Rectangle empty= new Rectangle(0,0,0,0);
    assertTrue(empty.isEmpty());
}

public void testWidth() {
    Rectangle empty= new Rectangle(0,0,0,0);
    assertEquals(0.0, empty.getWidth(), 0.0);
}
```

（这同样演示了需要一定误差精度的`assertEquals()`的浮点版本。）我们像下面这样除去测试中的重复部分：

EmptyRectangleTest

```
private Rectangle empty;

public void setUp() {
    empty= new Rectangle(0,0,0,0);
}

public void testEmpty() {
    assertTrue(empty.isEmpty());
}

public void testWidth() {
    assertEquals(0.0, empty.getWidth(), 0.0);
}
```

我们已经把公共代码提取出来作为一种方法，在调用测试方法之前，测试框架将保证调用这一方法。测试方法进一步简单化，但是要想理解这些测试方法我们必须知道`setUp()`中都有些什么。

我们究竟应该使用哪种方式呢？两种都用。我几乎总是把公共设置代码分离出来，但我对细节有很强的记忆力。读我的测试的人有时就抱怨有太多的东西需要记忆，因此，可能我应该少分离出一些来。

`TestCase`的子类和这些子类的实例之间的关系是`xUnit`中最容易混淆的部分之一。每一种新的固定设施都应当是`TestCase`的一个新的子类。每一个新的固定设施都在那个子类的一个实例中创建，使用一次，然后被清除。

在上述例子中，如果我们想要为一个非空的`Rectangle`写测试，那么我们就需要创建一个新类，可能叫`NormalRectangleTest`，并且在`setUp()`中将每一个不同的变量初始化一个非空的`Rectangle`类型。总之，如果我发现自己需要一种稍有不同的固定设施，那么就会创建一个新的`TestCase`子类。

这表明测试类和模型类之间不是简单的关系。有时一个固定设施用于测试若干个类（尽管很少见）。有时单独一个模型类就需要两到三个固定设施。实际上，最终测试类的数目和模型类的数目常常大致相同，但那并不是因为对于每个模型类，你要写一个且只能写一个测试类。

外部固定设施（External Fixture）

如何在固定设施中释放外部资源呢？覆盖`tearDown()`，然后释放资源。

记住：每个测试的目的就是使运行前和运行后的状态完全一致。例如，如果你在测试中打开了一个文件，在测试完成之前你就必须关闭它。可以这样写：

```
testMethod(self):
    file= File("foobar").open()
try:
    ...run the test...
finally:
    file.close()
```

如果那个文件在若干个测试中都用到了，那么就必须使其成为公共固定设施中的一部分：

```
setUp(self):
    self.file= File("foobar").open()
testMethod(self):
    try:
        ...run the test...
    finally:
        self.file.close()
```

首先，令人讨厌的重复的`finally`子句告诉我们在设计中还缺少某些东西。其次，使用这种方法容易出错，因为很容易忘记编写`finally`子句，或者把关闭文件一并忘了。最后，测试中有三处不优美的地方——`try`、`finally`及`close`本身，它们对运行测试来说都不是主要逻辑。

`xUnit`能保证在测试方法之后运行一个叫做`tearDown()`的方法。不论测试方法中发生了什么，`tearDown()`总要被调用（如果`setUp()`失败了，`tearDown()`不会被调用）。我们能把上面的测试修改为：

```
setUp(self):
    self.file= File("foobar").open()
testMethod(self):
    ...run the test...
tearDown(self):
    self.file.close()
```

测试方法（Test Method）

怎样表示一个单一的测试用例呢？把它看作一种方法，并且其名字以“`test`”开头。

在你的系统中，将有成百上千、成千上万的测试。你怎样记录所有这些测试呢？

异常测试 (Exception Test)

怎样测试期望的异常呢？捕获每个期望的异常，然后忽略它们，只有当没有抛出异常时才报错。

让我们来看看，假设我们正在写一些查找一个值的代码。如果这个值没有找到，那么我们就必须抛出一个异常。查找的测试十分简单：

```
public void testRate() {
    exchange.addRate("USD", "GBP", 2);
    int rate= exchange.findRate("USD", "GBP");
    assertEquals(2, rate);
}
```

异常的测试可能不是很明显。下面是我们的做法：

```
public void testMissingRate() {
    try {
        exchange.findRate("USD", "GBP");
        fail();
    } catch (IllegalArgumentException expected) {
    }
}
```

如果 `findRate()` 没有抛出一个异常，我们将调用 `fail()`，这是 xUnit 的一个方法，能报告测试失败。注意，我们只关心捕获我们所期望的特定异常，因此，当一个非期望异常（包括断言失败）被抛出时，我们也能够得到通知。

全部测试 (All Tests)

怎样一次性执行所有的测试呢？把所有测试套件 (suite) 合成一个套件——每个包一个，而整个应用是一个集成的测试包。

假想你把一个 `TestCase` 的子类加入一个包中，并且在那个类中加入了一个测试方法。那么，当你下次运行所有测试时，那个测试方法也应该运行。（这属于测试驱动的一部分——前面所说的只是个测试提纲，如果不是忙于写书的话，我也许会实现给大家看的。）因为大多数 xUnit 实现或 IDE 都不支持这一功能，所以每个包都应当声明一个 `AllTests` 类，这个类实现一个返回 `TestSuite` 的静态方法 `suite()`。下面是资金例子的 `AllTests`：

```
public class AllTests {
    public static void main(String[] args) {
        junit.swingui.TestRunner.run(AllTests.class);
    }
}
```

```
public static Test suite() {  
    TestSuite result= new TestSuite("TFD tests");  
    result.addTestSuite(MoneyTest.class);  
    result.addTestSuite(ExchangeTest.class);  
    result.addTestSuite(IdentityRateTest.class);  
    return result;  
}
```

你可以给 AllTests 一个 main() 方法，这样该类能直接在 IDE 或者命令行中运行。

30

设计模式

模式的一个主要优点是尽管看起来我们一直在处理完全不同的问题，但我们处理的大多数问题是由我们所使用的工具产生的，而不是正在处理的外部问题产生的^①。正是因为如此，即使外部问题的处理环境差别迥异，我们仍然能够可望找到（事实上已经找到了）常见问题的通用解决方案。

利用对象来组织计算是一种用常见的、可预测的方法来解决内部所产生的常见子问题的最佳例证之一。设计模式的巨大成功见证了采用对象编程的程序员们所看到的通用性^②。而《Design Patterns》这本书的成功之处则在于它统一了这些模式的表述。该书似乎有点倾向于把设计看成是开发中的一个环节。当然它并不赞成把重构看作是一种设计活动。测试驱动开发中的设计则要求从略微不同的角度来看待设计模式。

我们这里所讲的设计模式并不详尽，它们只是一些足以让我们理解书中例子的设计。这些模式概括如下：

- 命令（Command）——表示把计算作为一个对象而不是消息来调用。
- 值对象（Value Object）——通过创建其值一经创建便永远不改变的对象来避免别名问题。
- 空对象（Null Object）——表示一种对象计算的基本情形。
- 模板方法（Template Method）——使用可以通过继承来具体化的抽象方法来表示计算序列中不变的内容。
- 插入式对象（Pluggable Object）——通过调用另一个具有两种或两种以上实现的对象来表示变化的内容。
- 插入式选择器（Pluggable Selector）——通过动态调用不同实例的不同方法来避免不必要的子类。

① Alexander, Christopher. 1970. *Notes on the Synthesis of Form*. Cambridge, MA: Harvard University Press. ISBN: 0674627512.

② Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. 1995. *Design Patterns: Elements of Reusable Object Oriented Software*. Boston: Addison-Wesley. ISBN: 0201633612.

- 工厂方法 (Factory Method) ——通过调用方法而不是构造函数来创建对象。
- 冒名顶替 (Imposter) ——通过引入现有协议的另一种实现来引入变化。
- 递归组合 (Composite) ——使用一个对象来表示一组对象的行为的组合。
- 收集参数 (Collecting Parameter) ——来回传递一个用来汇集源于不同对象的计算结果的参数。

根据这些模式在 TDD 中应用场所的不同，表 30-1 列举如下。

表 30-1 测试驱动开发中设计模式的使用

模 式	编写测试	重 构
命令	X	
值对象	X	
空对象		X
模板方法		X
插入式对象		X
插入式选择器		X
工厂方法	X	X
冒名顶替	X	X
递归组合	X	X
收集参数	X	X

命令

当要调用一个比简单方法调用要复杂得多的计算时，你该怎么做呢？为那个计算创建一个对象，然后调用它。

发送消息是一种很好的方法。编程语言语法上的支持使得发送消息很容易实现；而编程环境又使得对消息的操纵变得很容易（例如，能够自动地重命名消息的重构）。然而，有时仅仅发送消息是不够的。

例如，假设我们想要把记录消息被发送出去的情况记录下来 (logging)，那么我们可以通过增加语言功能[包裹方法 (wrapper method)]来实现，但只有记录很少能够满足要求，而人们往往更看中语言的简单性，所以我们一般不这么做。另外，假设我们稍后要调用一个计算。我们可以启动一个线程，然后立即把它挂起，稍后再重新启动，但是那样的话，我们还会有让人兴奋的并发问题需要处理。

复杂化的计算调用需要昂贵的处理机制，但在大多数情况下，我们不需要这么高的复杂性，我们不愿花费这么高的代价。当我们所需要的调用仅仅比消息稍微具体一点、稍稍更具可操作

性一些时, 对象能给我们答案。创建一个对象来表示调用, 赋予它计算所需要的所有参数。当我们做好一切准备时, 就可以使用通用的协议, 如 `run()` 来调用它。

Java 的 `Runnable` 接口就是一个极好的例子:

`Runnable`

```
interface Runnable
    public abstract void run();
```

你可以在 `run()` 的实现中填入你想做的任何事情。遗憾的是, Java 在语法上没有创建和调用 `Runnable` 的轻量级方法, 因此, 它们并不像其他语言中的等效部件一样用得那么多——如 `Smalltalk/Ruby` 或者 `LISP` 中的 `blocks` 或 `lambda`。

值对象

怎样设计可以被广泛共享、但并不关心其具体身份的对象呢? 在创建它们的时候就对其状态进行设定, 以后永远不要改变它。对这种对象的操作往往返回一个新的对象。

在这里, 我可以说对象真棒, 是不是? 对象是一种非常棒的组织逻辑的方法, 有助于事后的理解和扩展。但是对象还有一个小问题(好了, 不止这一个, 但就目前而言, 这个问题就足够说明问题了)。

假设我(一个对象)有一个 `Rectangle` 对象, 我根据 `Rectangle` 计算出了一个值, 比如说是它的面积。后来有人礼貌地向我索要我的 `Rectangle`, 而我也不想表现出不合作, 就给了他们。几分钟后, 瞧瞧吧, 那个 `Rectangle` 对象已经在我不知情的情况下改变了。之前我所计算出来的面积已经过时了, 而我却没有办法知道这一切。

这是一个经典的别名(aliasing)问题。如果两个对象共享指向第三个对象的引用, 且如果一个对象改变了那个共享对象, 那么另一个对象最好还是不要依赖于该共享对象的状态为好。

这儿有几种解决别名问题的方法。一种解决办法就是不要把你所依赖的对象给别人使用, 而采用拷贝的方法来解决这一问题。这样做在时间和空间上开销昂贵, 而且没有考虑你需要通过共享对象让其他对象共享改变的时间。另一种解决方案就是使用观察器(`Observer`), 在那里, 你显式地在那些你所依赖的对象中进行注册, 并且期望在它们改变时能通知你。观察器将使控制流很难跟踪, 而且用于建立和删除依赖的逻辑会变得丑陋不堪。

还有一种方法就是不把对象当对象看。对象具有会随时间改变的状态。如果可以的话, 我们可以消除这种随时间的改变。如果我有一个对象, 并且知道它不会改变, 那么我就可以随意地到处传递指向这个对象的引用, 我知道别名在此不成问题。如果对象不发生变化的话, 那么共享对象就不存在不为他人所知的变化了。

记得当初学习 `Smalltalk` 的时候, 我对整数感到困惑。如果我想把位 2 改成 1, 为什么所有的 2 都没有变成 6 呢?

```

a := 2.
b := a.

a := a bitAt: 2 put: 1.
a => 6
b => 2

```

整数其实是伪装成对象的值。在 Smalltalk 中，对于小整数来说它们确实是数值，而对于无法在单个机器字中容下的整数而言它们是用对象模拟的。当我对那一位进行设置时，我所得到的的是一个置位后的新对象，而不是对原来的对象进行置位。

在实现值对象时，所有的操作都必须返回一个新对象，而保持原来的对象不变。用户必须知道他们正在使用的是一个值对象，并把结果保存起来（如前所述）。所有这些对象分配都会产生性能问题，就像在你使用了真实的数据集、真实的应用模式、剖析数据之后出现的性能问题以及随即而来的对性能的抱怨一样，我们应该像处理其他的性能问题一样来进行处理。

每当碰到类似于代数的问题时——如几何形状的交、并，具有数值并且能执行符号算术运算的数值单位，我都倾向于采用值对象。无论何时，只要采用值对象有一丝意义，我都会尝试一下，因为值对象使得代码的阅读和调试都非常容易。

所有的值对象都必须实现等值操作（equality）[在很多语言中这意味着要实现散列运算（hashing）]。如果两个对象具有不同的合约（contract），那么它们是不同的，即不相等。然而，如果我这儿有 5 法郎，那儿也有 5 法郎，那就无所谓它们是不是同一个 5 法郎，5 法郎就是 5 法郎，它们应该是相等的。

空对象

怎样用对象来表示特殊情形呢？创建一个代表特殊情形的对象，让它具备和普通对象相同的协议。

inspired by java.io.File

```

public boolean setReadOnly() {
    SecurityManager guard = System.getSecurityManager();
    if (guard != null) {
        guard.canWrite(path);
    }
    return fileSystem.setReadOnly(this);
}

```

在 java.io.File 中对 guard != null 有 18 处检查。尽管我十分欣赏他们为保证文件安全所作的努力，但我还是心存疑虑。他们总是能够认真地检查 getSecurityManager() 的结果是不是 null 吗？

另一种可供选择的方法就是创建一个叫 LaxSecurity 的新类，它绝不会抛出异常。

LaxSecurity

```

public void canWrite(String path) {
}

```


如果有人请求一个 `SecurityManager`, 并且这儿没有一个可用的, 那么我们就可发送一个 `LaxSecurity` 类来代替:

`SecurityManager`

```
public static SecurityManager getSecurityManager() {  
    return security == null ? new LaxSecurity() : security;  
}
```

现在我们就不必担心有人忘记检查空值了。原来的代码整洁了很多:

`File`

```
public boolean setReadOnly() {  
    SecurityManager security = System.getSecurityManager();  
    security.canWrite(path);  
    return fileSystem.setReadOnly(this);  
}
```

我和Erich Gamma曾经在一次OOPSLA的讲座上对于在JHotDraw中的某个地方采用空对象是否恰当展开了争论。正当我略占上风时, Erich计算得出引入空对象需要10行代码, 而我们用这10行代码去掉一个条件判定。我讨厌这种最后回合遭受技术击倒的感觉(TKO, technical knock-out)。(由于没有组织好, 我们给观众留下了极差的印象。显然他们不知道卓有成效的技术讨论是一种困难但可以学习掌握的技能。)

模板方法

怎样表示计算中的不变顺序步骤(invariant sequence)同时又能提供将来可以进行细化的设施呢? 编写一个方法, 而这个方法完全是根据其他方法来实现的。

编程中充满了经典的顺序步骤:

- 输入/处理/输出
- 发送消息/接收应答
- 读取命令/返回结果

我们希望能够清晰地表述出这些步骤的普适性, 而同时又在每一步的具体实现中引入变化。

通过继承(inheritance), 对象语言提供了一种或许有限但却简单的表达这种普适序列的机制。一个超类可以包含一个完全用其他方法来编写的方法, 而子类则可以以不同的方式来实现这些方法。例如, JUnit 运行一个测试的基本的顺序是这样的:

`TestCase`

```
public void runBare() throws Throwable {  
    setUp();  
    try {  
        runTest();  
    }  
    finally {  
        tearDown();  
    }  
}
```


子类可以随心所欲地实现 `setUp()`、`runTest()` 和 `tearDown()`。

在编写模板方法时需要考虑的一个问题就是是否应该为子方法编写一种缺省的实现。在 `TestCase.runBare()` 中，所有三个子方法都有缺省实现。

- `setUp()` 和 `tearDown()` 是没有操作。
- `runTest()` 基于测试用例的名称动态地查找并调用测试方法。

如果不填入子步骤计算就没有任何意义的话，就要使用编程语言所提供的方法加以标注。

- 在 Java 中，要声明子方法为 `abstract`（抽象的）。
- 在 Smalltalk 中，要在方法实现中抛出一个 `SubclassResponsibility` 错误。

最好是在实践中发现模板方法，而不是一开始就设计采用模板方法。每当我对自己说“哦，顺序步骤是这样的，而具体细节是那样”的时候，后来我总是发现自己以内联（`inline`）的方式键入方法的细节，并重新提取出其中真正的变化部分。

当你发现在两个子类中存在某一顺序步骤的两个变体时，你需要一点一点地缩小它们的差异。一旦从其他方法中提取出了相异的部分之后，那么剩下的就是模板方法了。然后，你就可以把该模板方法挪到超类中，并消除重复。

插入式对象

怎样表达变化呢？最简单的方法就是显式地使用条件判定：

```
if (circle) then {
... circley stuff...
} else {
... non circley stuff
}
```

很快你就会发现这种显式的判定开始扩散开来。如果你在一个地方采用显式的判定语句来区分圆与非圆，那么这个判定语句就可能会扩散到其他的地方。

测试推动开发中的第二个重要思想就是要消除重复（`duplication`）。因此，你必须将这种类似瘟疫的显式条件判定消灭在萌芽状态。当你第二次碰到某个条件判定时，就到了采用最基本的对象设计技法——插入式对象的时候了。

通过简单地消除重复而采用插入式对象有时显得不够直观。我和 Erich Gamma 发现了一个这样的例子，它是我最喜爱的一个料想不到会采用插入式对象的例子。在编写图形编辑器的时候，选择操作有点复杂。当你在一个图形上点击鼠标键时，接下来移动鼠标将移动这个图形，而松开鼠标键，该图形处于被选中状态。如果鼠标没有处于任何一个图形上面，那么你将要选择的一组图形，接下去的鼠标移动通常会延展出一个矩形，该矩形用于选择多个图形。松开鼠标键，矩形内的所有图形均处于被选中状态。最初的代码如下所示：

SelectionTool

```
Figure selected;
public void mouseDown() {
```

```
selected= findFigure();
if (selected != null)
    select(selected);
}
public void mouseMove() {
    if (selected != null)
        move(selected);
    else
        moveSelectionRectangle();
}
public void mouseUp() {
    if (selected == null)
        selectAll();
}
```

这里存在丑陋的重复判定（我已经讲过，它们将像疾病一样地蔓延）。解决办法就是创建一个叫做 **SelectionMode** 的插入式对象，该对象具有 **SingleSelection** 和 **MultipleSelection** 两种实现。

SelectionMode

```
SelectionMode mode;
public void mouseDown() {
    selected= findFigure();
    if (selected != null)
        mode= SingleSelection(selected);
    else
        mode= MultipleSelection();
}
public void mouseMove() {
    mode.mouseMove();
}
public void mouseUp() {
    mode.mouseUp();
}
```

在具有显式接口的语言中，你必须要实现两个（或者两个以上）插入式对象的接口。

插入式选择器^①

你怎样针对不同的实例调用不同的行为呢？存储方法的名字，然后动态调用这个方法。

如果出现了一个类中有10个子类且每个子类仅仅实现一种方法的情况该怎么办呢？使用子类来捕获如此少的变化，有点大材小用。

```
abstract class Report {
    abstract void print();
}
```

^① 更多的细节请见 K. Beck 的《The Smalltalk Best Practice Patterns》，pp.70~73, Prentice-Hall, 1997, ISBN 013476904X。引用自己作品的形式不好，但正如著名的哲学家 Phyllis Diller 所说，“当然我为自己的笑话而笑。陌生人是不能相信的。”

```
}  
  
class HTMLReport extends Report {  
    void print() { ...  
    }  
}  
  
class XMLReport extends Report {  
    void print() { ...  
    }  
}
```

另一种解决方法就是使用一个包含switch语句的类。根据某个字段值的不同，调用不同的方法。然而，方法名要出现在三个地方：

- 创建实例的地方
- switch 语句中
- 方法本身

```
abstract class Report {  
    String printMessage;  
  
    Report(String printMessage) {  
        this.printMessage= printMessage;  
    }  
  
    void print() {  
        switch (printMessage) {  
            case "printHTML" :  
                printHTML();  
                break;  
            case "printXML" :  
                printXML();  
                break;  
        }  
    };  
  
    void printHTML() {  
    }  
  
    void printXML() {  
    }  
}
```

每次添加一种新的打印种类，你必须确保增加打印方法，并修改switch语句。

插入式选择器方案使用反射（reflection）机制来动态调用方法：

```
void print() {  
    Method runMethod= getClass().getMethod(printMessage, null);  
    runMethod.invoke(this, new Class[0]);  
}
```

现在，在报告的创建者与打印方法名之间还存在着丑陋的依赖关系，但至少你不用使用case语句了。

插入式选择器很容易被滥用。最大的问题就是跟踪代码来检查某个方法是否被调用。只有

当你在整理某种一组子类中每个类只有一种方法的较为简单的情形时才使用插入式选择器。

工厂方法

如果你在创建对象时需要一定程度的创建新对象的灵活性时该怎么做呢？使用方法而不是由构造函数来创建对象。

构造函数表述直观。在使用的时候，你可以明确看到创建一个对象。但是，构造函数，尤其是 Java 中的构造函数缺乏清晰的表达性和灵活性。

在资金例子中，我们所需要的一种灵活性就是在创建对象的时候能够返回不同类的对象。我们有如下测试：

```
public void testMultiplication() {  
    Dollar five= new Dollar(5);  
    assertEquals(new Dollar(10), five.times(2));  
    assertEquals(new Dollar(15), five.times(3));  
}
```

如果无法摆脱只能创建 Dollar 实例的事实，我们就无法引入 Money 类。通过一个方法引入一层间接（indirection）后，我们就获得了在不改变测试的情况下返回不同类实例的灵活性：

```
public void testMultiplication() {  
    Dollar five = Money.dollar(5);  
    assertEquals(new Dollar(10), five.times(2));  
    assertEquals(new Dollar(15), five.times(3));  
}
```

Money

```
static Dollar dollar(int amount) {  
    return new Dollar(amount);  
}
```

这种方法被称做工厂方法，因为它创建了对象。

采用工厂方法的缺点恰恰就是这层间接。尽管它不像一个构造函数，但你仍要切记这种方法确实会创建一个对象。只有当需要这种灵活性的时候，才去使用工厂方法。不然，对于创建对象来说，构造函数足以胜任了。

冒名顶替

怎样在计算中引入一种新的变化？引入一种所用协议与现有对象相同但具体实现不同的新对象。

在面向过程的程序中引入变化需要添加条件逻辑。正如我们所看到的插入式对象那样，这些逻辑往往迅速扩张，需要一组设计良好的多态消息来去掉这些重复代码。

假设某种结构已经成型，而且现存有这样一个对象。而你现在想要系统完成一些不同的工

作。如果有明显的可以插入if语句的地方,而且你要编写的逻辑不会与别处重复,那么这样做就行了。然而实际情况往往是引入这种变化需要对多个方法进行修改。

测试驱动开发中有两种情形决定使用这种模式。有时你要编写一个表述新用例的测试案例,而现有的对象没有一个能够表述你想要表述的内容。假设我们正在测试一个图形编辑器,而且我们已经有了能够正确进行绘制的矩形:

```
testRectangle() {  
    Drawing d= new Drawing();  
    d.addFigure(new RectangleFigure(0, 10, 50, 100));  
    RecordingMedium brush= new RecordingMedium();  
    d.display(brush);  
    assertEquals("rectangle 0 10 50 100\n", brush.log());  
}
```

现在我们想显示一个椭圆。在这种情况下,采用冒名顶替模式显而易见——将 `RectangleFigure` 替换成 `OvalFigure`。

```
testOval() {  
    Drawing d= new Drawing();  
    d.addFigure(new OvalFigure(0, 10, 50, 100));  
    RecordingMedium brush= new RecordingMedium();  
    d.display(brush);  
    assertEquals("oval 0 10 50 100\n", brush.log());  
}
```

通常,第一时间观察出使用冒名顶替模式需要具备一定的洞察力。当 Ward Cunningham 意识到一组 `Money` 与一个 `Money` 作用一样时就是这样一种情形。你认为它们是不同的,但现在却能够把它们看成是一样的东西。

下面是在重构过程中所出现的两个冒名顶替模式的例子:

- 空对象——你能像对待存在的数据一样对待并不存在的数据。
- 递归组合——你能像对待单个对象一样对待对象集。

就像所有的重构是由消除重复推动的一样,在重构中寻找冒名顶替模式也是由它推动的。

递归组合

怎样实现一个对象,该对象的行为是一列其他对象行为的组合?使该对象成为组成对象的冒名顶替者(Imposter)。

我最喜欢的例子也是不适合使用递归组合的例子——`Account` 和 `Transaction`。`Transaction` 用来存放值的增量(实际上它们要复杂、有趣得多):

```
Transaction  
Transaction(Money value) {  
    this.value= value;  
}
```

`Account` 通过累加 `Transaction` 的值来计算其余额:

Account

```
Transaction transactions[];
Money balance() {
    Money sum= Money.zero();
    for (int i= 0; i < transactions.length; i++)
        sum= sum.plus(transactions[i].value);
    return sum;
}
```

它们看起来十分简单。

- Transaction 有一个值。
- Account 有一个余额。

接着，有趣的问题出现了。一个客户有一堆账户（Account），他想要查看所有账户的总余额。最明显的方法就是把 OverallAccount 作为一个新类来实现，该类用于累加一系列 Account 余额。重复设计！重复设计！

要是 Account 和 Balance 实现同一个接口会怎么样呢？让我们称之为 Holding，因为此刻我也想不出比这更好的名字来。

Holding

```
interface Holding
    Money balance();
```

Transaction 可以通过返回它们的值来实现 balance()：

Transaction

```
Money balance() {
    return value;
}
```

现在 Account 可以是由 Holding 而不是 Transaction 组成的了：

Account

```
Holding holdings[];
Money balance() {
    Money sum= Money.zero();
    for (int i= 0; i < holdings.length; i++)
        sum= sum.plus(holdings[i].balance());
    return sum;
}
```

现在与 OverallAccount 有关的问题消失了。一个 OverallAccount 就是包含多个 Account 的 Account。

上面所述只是递归组合模式的一点皮毛。在现实世界中，Transaction 并没有余额。使用递归组合模式是程序员惯用的技巧，其他人通常不好理解。然而，它对于程序设计的裨益是巨大的，因而与真实概念的脱离常常是值得的。Folder（文件夹）包含 Folder、TestSuite 包含 TestSuite、Drawing 包含 Drawing，所有这些很难在外部世界中找到对等的概念，但它们却使代码变得如此简单。

我必须反复尝试很长时间才知道哪里可以使用递归组合模式，哪里不能使用。从这里的讨

论可以看出,我仍然不能清楚地表述出什么时候一个对象集合仅仅是一个对象集合,什么时候一个对象集合应当是一个递归组合。值得欣慰的是,由于我们越来越擅长于重构,所以当重复出现的时候,你就可以引入递归组合并目睹程序复杂性的消失。

收集参数

怎样收集一个散布于若干对象中的操作的结果呢?在操作中加一个参数,在那里收集结果。

一个简单的例子就是 `java.io.Externalizable` 接口。`writeExternal` 方法将一个对象及其引用的所有对象写出。因为所有对象都必须以较松散的合作方式才能写出,所以该方法传递了一个 `ObjectOutput` 参数作为收集参数:

```
java.io.Externalizable
public interface Externalizable extends java.io.Serializable {
    void writeExternal(ObjectOutput out) throws IOException;
}
```

增加收集参数是使用递归组合模式后的一种常见现象。在开发 JUnit 时,直到我们有了若干个测试后,才需要使用 `TestResult` 来收集多个测试的结果。

随着期望结果复杂性的增加,你可能会发现引入收集参数的必要性。例如,假设我们要打印 `Expression`。如果我们想要的只是一个普通的字符串,那么字符串连接就足够了:

```
testSumPrinting() {
    Sum sum= new Sum(Money.dollar(5), Money.franc(7));
    assertEquals("5 USD + 7 CHF", sum.toString());
}

String toString() {
    return augend + " + " + addend;
}
```

但是,如果我们想要缩进树形式的表达式,那么代码将会像下面这样:

```
testSumPrinting() {
    Sum sum= new Sum(Money.dollar(5), Money.franc(7));
    assertEquals("+\n\t5 USD\n\t7 CHF", sum.toString());
}
```

如下所示,我们必须引进一个收集参数:

```
String toString() {
    IndentingStream writer= new IndentingStream();
    toString(writer);
    return writer.contents();
}

void toString(IndentingWriter writer) {
    writer.println("+");
}
```

```
writer.indent();  
augend.toString(writer);  
writer.println();  
addend.toString(writer);  
writer.exdent();  
}
```

单例模式（Singleton）

在没有全局变量的语言中怎样提供全局变量呢？不要使用全局变量。多考虑考虑你的设计思路吧，这样对编程是有好处的。

31

重 构

这些模式用来描述怎样小步地改变系统的设计，甚至是大幅度的修改。

在测试驱动开发中我们以一种有趣的方式使用重构^①。通常在任何情况下，重构是不能改变程序的语义的。在测试驱动开发中，我们所关心的只是那些已经通过的测试，所以，举个例子来说，在测试驱动开发中可以用变量来取代常量，我们完全可以认为这种操作就是一种重构，因为它并不改变已经通过的测试。惟一保留语义的情况或许就是我们的一个测试用例。任何其他以前通过的测试用例都可能失败。然而我们到目前为止还没有那样的测试，所以我们现在还不用担心这一点。

这种“观察性的等价 (observational equivalence)”要求你有足够多的测试，这样一来，在你的认知范围内，至少在完成时，依靠这些测试所进行的重构与就所有可能的测试而进行的重构是等价的。你没有理由这样说：“我知道这儿有一个问题，但是所有的测试都通过了，于是我就把代码提交上去了 (check in)。”那就编写更多的测试。

调和差异 (Reconcile Differences)

怎样统一两段看起来相似的代码呢？一点一点地减少它们之间的差异，只有当它们完全一致的时候才去统一它们。

重构是非常伤脑筋的工作。简单的东西一目了然。如果我们提取一种方法而且是机械地正确地完成的话，那么改变系统行为的几率是很小的。但是有一些重构要求你要仔细检查控制流和数据值。一长串的推理将使你相信你将做的改变不会使任何现有答案发生改变。这些就是那种能够提高你处理问题能力的重构。

这种超出可信度的重构恰恰是我们力图通过小步骤和具体反馈的策略所要避免的。尽管我们不能避免所有这种重构，但却可以减轻其带来的影响。

^① Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley. ISBN 0201485672.

这种重构可以在各种规模层次上进行。

- 如果两个循环结构类似，那么通过使其完全一致，我们可以将二者合并。
- 如果一个判定语句的两个分支类似，那么通过使其完全一致，我们可以去掉这个条件判定。
- 如果两个方法类似，那么通过使其完全一致，我们可以去掉其中一个方法。
- 如果两个类类似，那么通过使其完全一致，我们可以去掉其中一个。

有时你需要从反方向来解决差异的调和问题，也就是说考虑如何才能使最后一步改变最轻松易行，然后倒过来考虑如何才能到达最后这一步。例如，如果你想去掉若干个子类，那么最简单的最后一步就是子类不包含任何内容的情形，这样超类就能够取代子类而不改变系统的行为。为了清空子类，我们需要使子类中的方法与超类中的方法完全一致。我们一个接一个地清空子类，然后当它们被清空时，将所有指向它们的引用换成指向超类的引用。

隔离变化 (Isolate Change)

怎样改变具有多个部分的方法或对象的某一部分呢？首先，对要修改的部分进行隔离。

外科手术的图像出现在我的脑海里：除了要动手术的地方，整个病人被用布盖起来，这就给医生留下了固定的一块地方。现在，我们也许可以长篇大论地讨论将病人抽象成其腹部左下四分之一部分是否是一种良好的医疗措施。但是在手术时，医生能够集中注意力还是令人欣慰的。

一旦隔离了变化部分并进行了修改，你会发现其对结果的影响是如此之小以至于我们可以取消隔离。如果发现我们需要做的只是在 `findRate()` 函数中返回一个实例变量，那么我们可以考虑在用到这个方法的地方以内联的方式插入 `findRate()` 的代码并删除这一方法。但是，不要不假思索地做这些改动，要权衡一下添加一种方法与在代码中显式地增加某一概念的价值。

一些可能的隔离变化的方法有提取方法 (Extract Method) (最常用的)、提取对象 (Extract Object) 和方法对象 (Method Object)。

数据迁移 (Migrate Data)

怎样从一种数据表示形式迁移至另外一种形式呢？临时制作冗余数据。

怎样实现？

这是一种由内而外的迁移方式，即改变内部的表示形式，然后再改变外部的可见接口。

- 增加以新的数据格式表示的实例变量。
- 在所有设置旧格式变量 (format variable) 的地方设置新的格式变量。
- 在任何使用旧格式变量的地方使用新的格式变量。
- 删除原先的格式。

- 改变外部接口以反映这种新的格式。

然而有时你想要先改变 API，那么就应该这样做：

- 增加一个以新数据格式表示的参数。
- 将新数据格式的参数转换成内部原先的数据格式表示。
- 删除旧格式的参数。
- 将使用旧格式的地方换成使用新格式。
- 删除旧格式。

为什么？

一到多（One to Many）情形每次都会产生数据迁移问题。假设我们想以从一到多的方式来实现 TestSuite 类，我们这样开始：

```
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.run(self.result)
    assert("1 run, 0 failed" == self.result.summary())
```

通过如下代码实现（在一到多的“一”中实现）：

```
class TestSuite:
    def add(self, test):
        self.test= test
    def run(self, result):
        self.test.run(result)
```

现在我们开始制作冗余数据。首先初始化测试集：

```
TestSuite
def __init__(self):
    self.tests= []
```

在每个设置 test 的地方，我们都将其添加到集合中：

```
TestSuite
def add(self, test):
    self.test= test
    self.tests.append(test)
```

现在我们用测试列表来取代单个测试。就当前测试用例的目的而言，这是一种重构（它保留了语义），因为集合体中只有一个元素。

```
TestSuite
def run(self, result):
    for test in self.tests:
        test.run(result)
```

删除现在不使用的实例变量 test：

```
TestSuite
def add(self, test):
    self.tests.append(test)
```

就像从 Java 的 Vector/Enumerator 迁移至 Collection/Iterator 一样，当以不同的协议在相同格式之间迁移数据时，你也可以使用逐步迁移数据的方法。

提取方法（Extract Method）

怎样使一个既长又复杂的方法比较容易读懂呢？将其中的一小部分转变成一个单独的方法并调用这个新的方法。

怎样实现？

提取方法实际上是一种比较复杂的原子级（atomic）重构。这里我将举一个典型的例子。幸运的是，它也是最常以自动化方式完成的重构，因此不必手工实现。

（1）在方法中寻找一块可以单独独立出来形成方法的区域。循环体、整个循环以及条件语句的分支都是常见的可供提取的部分。

（2）保证在要提取的区域内不存在向在本区域外声明的临时变量赋值的语句。

（3）将这部分代码从原来的方法中复制到新方法中，然后编译。

（4）对于在新方法中用到的原方法中的每个临时变量或参数，都为新方法增加一个参数。

（5）在原来的方法中调用新方法。

为什么？

当我试图去理解复杂的代码时，我会使用提取方法。“这一段代码在这里完成了这样的工作，起个什么名字好呢？”半个小时后，代码变得更易读了，你的搭档会意识到你确实能帮上忙，并且你也能更清楚地理解正在进行的工作。

当我看到两种方法中有些部分相同，有些部分不同时，我就会采用提取方法来删除重复的部分。我把相似的地方提取出来。（Smalltalk 中的重构浏览器甚至能检查出你正在提取的方法是否与你的已有方法等价，并提出让你使用现有的方法而不是创建一个新方法。）

把方法拆分成小段有时会走过了头儿。当我不知道下一步怎么走时，常常会使用内联方法（Inline Method）（很方便，是下一种重构），把所有的代码放到一个地方，这样我就能重新发现应该提取些什么。

内联方法（Inline Method）

如何简化变得过于繁琐或分散的控制流程呢？在调用方法的地方把方法本身内联起来。

怎样实现？

（1）复制方法。

(2) 在调用语句处粘贴该方法。

(3) 将所有形参 (formal parameter) 替换成实参 (actual parameter)。例如, 如果你要传递参数 `reader.getNext()` (这是一种会产生副作用的表达式), 那么就得注意把它赋值给一个局部变量。

为什么?

本书的一个审校人员曾就本书第一部分使用 `Bank` 来将 `Expression` 化归成单个 `Money` 提出意见。

```
public void testSimpleAddition() {  
    Money five= Money.dollar(5);  
    Expression sum= five.plus(five);  
    Bank bank= new Bank();  
    Money reduced= bank.reduce(sum, "USD");  
    assertEquals(Money.dollar(10), reduced);  
}
```

“这太复杂了。你为什么不让 `Money` 来对其自身进行化归呢?” 那么我们怎样针对这一问题来进行试验呢? 将 `Bank.reduce()` 的实现内联进来, 看看会是什么样子:

```
public void testSimpleAddition() {  
    Money five= Money.dollar(5);  
    Expression sum= five.plus(five);  
    Bank bank= new Bank();  
    Money reduced= sum.reduce(bank, "USD");  
    assertEquals(Money.dollar(10), reduced);  
}
```

你或许更喜欢第二个版本, 也许不。这里所要指出的是你可以使用内联方法来调整控制流程。我在重构时, 脑海里就会出现系统的一幅图像, 逻辑片段和控制流在各种对象之间流淌。当我认为发现了好的思路时, 我就通过重构来进行试验, 并查看结果。

在编程进入白热化时, 我时常聪明反被聪明误。当出现这样的情况时, 内联方法是我的退路: “我把这个发送到那, 发送到那……遏, 可恶! 这里出了什么问题?” 我把多层抽象以内联的方法直接插入, 看看究竟将发生什么, 然后我就可以根据实际需要而不是自己的主观臆断来重新对代码进行抽象。

提取接口 (Extract Interface)

在 Java 中, 怎样引进操作的另一种实现呢? 创建一个包含该共享操作的接口。

怎样实现?

(1) 声明一个接口。有时一个现有类的名字就应当是这个接口的名字, 在这种情况下, 你首先要重命名这个类。

- (2) 让现有的类实现这个接口。
- (3) 向接口中添加必要的方法，必要的话，在类中扩展方法的可见性。
- (4) 在可能的地方将类型声明从类更改为接口。

为什么？

有时在你需要提取接口时，实际上你是在从一种实现转移到另一种实现。你有一个 `Rectangle` 类，并且想要添加一个 `Oval` 类，因此，你就创建了一个 `Shape` 接口。尽管有时你得绞尽脑汁才能找到一个恰当的比喻，但是通常情况下，为接口取名是很简单的。

有时在你需要提取接口时，你要引入清扫测试死角（`Crash Test Dummy`）或者其他的模拟对象（`Mock Object`）。在这种情况下，命名一般是比较困难的，因为目前你手头上只有一个是真的例子。有时我也管不了那么多了，于是更倾向于给接口取名为 `IFile`，类的名字就是 `File`。我教会了自己在这种时候要稍停片刻，看自己是否对目前的工作理解得还不够深。也许这个接口应当被称做 `File` 而类被称做 `DiskFile`，因为那个类假设数据是存在磁盘上的。

转移方法（Move Method）

怎样把一个方法转移到它所归属的地方？把它添加到它所归属的那个类中，然后调用它。

怎样实现？

- (1) 复制方法。
- (2) 在目标类中粘贴方法，并适当命名，然后编译。
- (3) 如果方法引用了原来的对象，那么就加一个参数来传递原来的对象。如果方法引用了原对象中的变量，那么就把这些变量作为参数传递进来。如果原对象中的变量是设置好的，那么你就应该放弃这么做了。
- (4) 将原方法体替换成对新方法的调用。

怎样实现？

下面是我最喜欢的在咨询时演示的重构，因为它很好地暴露了没有根据的设想所带来的危害。`Shape` 的任务就是计算面积：

`Shape`

```
...
int width= bounds.right() - bounds.left();
int height= bounds.bottom() - bounds.top();
int area= width * height;
...
```

在一个方法中，每当有超过一条以上的消息发送给另一个对象时，我就会怀疑可能存在问

题。在这个例子中，我发现有四条消息发送给 `bounds`（它是一个 `Rectangle`）。是转移这个方法的时候了：

Rectangle

```
public int area() {  
    int width= this.right() - this.left();  
    int height= this.bottom() - this.top();  
    return width * height;  
}
```

Shape

```
...  
int area= bounds.area();  
...
```

下面是转移方法的三大特点：

- 即使对代码的意思理解不深，也能很容易看出哪里需要进行方法的转移。每当你看到两个或者两个以上的消息发给一个不同的对象时，就要移动方法了。
- 这种方法迅速且安全。
- 结果通常具有启发性。“但是 `Rectangle` 没有做任何计算……哦，我知道了，那样更好。

有时你想要转移的仅仅是方法的一部分，可以先提取一个方法出来，然后转移整个方法。把该方法（现在只要一行）内联到原来的类中。也许你能够搞清楚怎样一遍就能完成这种动作。

方法对象（Method Object）

怎样表示一个复杂的需要若干参数和局部变量的方法？为该方法创建一个对象。

怎样实现？

- 使用与方法一样的参数来创建一个对象。
- 把局部变量变成对象的实例变量。
- 创建一个叫做 `run()` 的方法，使其内容与原来的方法内容相同。
- 在原方法中，创建一个新的对象，然后调用 `run()`。

为什么？

在准备把一种新的逻辑加入系统时，方法对象十分有用。例如，当你根据成分资金流计算资金流的时候要完成多个方法调用。当你想要开始计算资金流的当前净利时，可以首先根据第一种计算形式创建一个方法对象。然后，你就可以在更小范围内、用自己的测试编写新的计算形式。而插入新的计算形式只需简单的一步。

方法对象对于提取方法无能为力的方面也颇有成效。有时你会发现一段代码有大量的临时变量和参数，且每次你试图提取其中的一段时，你就必须随提取的方法提供 5 到 6 个临时变量。

和参数。提取的方法与原来的代码相比好不了多少，因为方法签名太长了。而创建一个方法对象给了你一个新的名字空间，在那里你可以提取方法而不用传递什么。

添加参数 (Add Parameter)

怎样把参数添加到方法中？

怎样实现？

- (1) 如果方法在一个接口中，那么先在该接口中添加参数。
- (2) 添加参数。
- (3) 使用编译时的出错信息来指导你需要修改哪些调用代码。

为什么？

添加参数通常算是一种扩展。你使第一个测试用例在不需要这一参数的情况下运行通过，但是在增加了参数的新情况下，为了使计算准确无误，你必须要考虑更多的东西。

添加参数也算是把一种数据表示迁移至另一种数据表示的一部分。首先添加参数，然后删除所有使用旧参数的地方，接着再删除旧参数。

把方法中的参数转变为构造函数中的参数

怎样把一个参数从一个或多个方法中转移到构造函数中呢？

怎样实现？

- (1) 在构造函数中添加一个参数。
- (2) 添加一个名称与该参数名相同的实例变量。
- (3) 在构造函数中设置变量。
- (4) 一个接一个地把对“parameter”的引用改为对“this.parameter”的引用。
- (5) 当不存在对参数的引用时，从方法和所有的调用者中删除这个参数。
- (6) 从引用中删除现在多余的“this.”。
- (7) 正确地重命名变量。

为什么？

如果你将同一个参数传递给同一对象中的多个不同的方法，那么你可以通过只传递一次参数（消除重复）来简化 API。如果你发现某个实例变量只在一个方法中使用了，那么你可以反方向应用这种重构。

32

掌握 TDD

我希望在这里提出一些问题，供你们在将测试驱动开发（TDD）集成到自己的开发实践过程中时思考。其中有一些是小问题，而有一些则是大问题。有的问题提供了答案，或者至少有相应的提示，而有些问题则是留给你自己去探讨的。

步伐应该有多大？

这里暗指两个问题：

- 每个测试程序覆盖的方面有多大？
- 重构时中间步骤有多少？

你可以编写使每个都对应一行逻辑代码和少数重构的测试。你也可以编写每个都对应上百行逻辑代码和数小时重构的测试。你应当选择哪一种方式呢？

回答这一问题的部分答案是不管哪种方式，你都应该有能力做到。不过，经过一段时间，测试驱动开发人员都会明显倾向于采用小步骤进行开发。然而有人正在尝试单独利用应用级别的测试或者结合我们所编写的程序员级别的测试来驱动开发。

首先，当你重构时，你应当做好采用大量微小步骤的准备。手工重构很容易犯错，而犯的错误越多而且解决这些错误越靠后，你继续进行重构的可能性就越小。一旦你通过细致的步骤进行了 20 遍重构，那么就可以尝试着省去其中的一些步骤。

自动重构能够大大提高重构速度。曾经要手工花费 20 步才能完成的重构现在只要点击一个菜单项就行了。当量变达到一定程度时通常会产生质变，自动重构也是如此。一旦你有了优秀工具的支持，在重构时便会更加野心勃勃、积极进取，就会尝试多得多的试验以了解怎样对代码结构进行组织。

在我编写这本书的时候，Smalltalk 的重构浏览器（Refactoring Browser）仍然是最好的重构工具。很多 Java 集成开发环境中都出现了对 Java 重构的支持，而对重构的支持肯定会很快传播到其他语言和编程环境中去。

什么可以不必测试？

Phlip 所提供的简略答案是：“写测试，一直写到恐惧转变为厌倦时为止。”然而这是一个

带反馈的循环，答案需要由你自己来找。可是你之所以阅读本书是因为想在此找到答案而不是问题（如果是后者，那你可读错了东西，不过相互递归参照的文献可读够了），所以请参考下面这个清单。你应该测试：

- 条件部分
- 循环部分
- 操作部分
- 多态性

但只测试你编写的代码。除非有理由不信任，否则就不要测试其他来源的代码。有时，准确的描述外部代码的细节（我的意思是“内部的 bug”）更多的是要编写出自己的逻辑代码。请参照上面的清单，看看有哪些内容需要测试。有时，为了谨慎起见，我会用测试来标注外来代码中的某种不寻常的行为特征，当这个 bug 被修正以后，我的这个测试就会无法运行通过，但是原来代码中的那种不良行为却不存在了。

怎样知道自己的测试没有疏漏呢？

测试就像是煤矿中的金丝雀一样，它们的不良反映揭示了周围可能出现的致命气体。下面是一些预示着设计存在缺陷的特征。

- 过长的设置代码——如果为了一个简单的断言，需要花费上百行代码创建对象，那么肯定有哪儿不对劲儿。对象太大，需要分割。
- 冗余的设置代码——如果你无法为公共设置代码找到一个存放它的公共场所的话，那么就表明有太多的对象过于紧密地联系在一起了。
- 过长的测试运行时间——测试驱动开发中运行时间太长的测试就不会经常被运行，常常是过一段时间才运行一次，也许这些测试已经无法工作了。更糟糕的是，它们暗示着对应用的方方面面进行测试是困难的。这种测试困难是一种设计问题，并且需要在设计时就被提出来。（十分钟的测试套件等同于 9.8m/s^2 的重力加速度。运行时间比十分钟还长的测试套件必然会被抛弃，或者对应用进行调整，使测试套件运行时间重新又回到 10 分钟以内。）
- 脆弱的测试——意外中断的测试说明应用的某一部分出人意料地存在对另一部分的影响。你需要对系统进行设计，要么打破联系，要么将两部分合并，直到这种影响消失为止。

测试驱动开发是怎样引领框架的形成的？

有一种看似矛盾实则正确的说法：不为代码的将来考虑，你的代码反而更有可能适应将来的需要。

而我从书本上学到的却恰好相反：“编码为今天，设计为明天。”而测试驱动开发看起来已彻底推翻了这一论点：“为明天编码，为今天设计。”下面是现实中出现的一些实际情况。

- 要求实现软件的第一个功能。该功能的实现简单明了，很快就完成了工作，而且瑕疵很少。
- 要实现的第二个功能是第一个功能的变种。将两项功能重复的代码放在同一个地方，将不同的部分放在不同的地方（放在不同的方法或类中）。
- 要实现的第三个功能是头两个的变种。也许稍加改动，我们仍有可能沿用目前共有的逻辑。而互不相同的逻辑部分一般都有自己明显的归属，要么在不同的方法中实现，要么在不同的类中实现。

这样，我们逐步满足了开放/封闭原则的要求（对象应当对扩展开放，对进一步的修改封闭），而该原则恰恰适合实践中所出现的各种各样的变种。TDD 留给你的是擅长表述那种变化的框架，尽管这种框架可能不擅长于表述那种没有发生（或尚未发生的）的变化。

那么，如果三年后，一种非同寻常的变化出现了会怎样？设计将在恰好需要的地方迅速进化而适应这种变化。我们只会在很短的时间内违反开放/封闭原则，但是你有测试在手，它给了你不会破坏任何现有功能的自信，所以违反这一原则根本不会付出高昂的代价。

在极端情况下，当你引进变化的速度非常快时，TDD 与前期的设计阶段是一样的。我曾经在几个小时内开发出了一个报告框架（reporting framework），而旁观者肯定觉得这里有诀窍，“他肯定是在脑子里有了成型的框架以后才开始的”。不，你错了。我只是采用测试驱动的方式进行开发的时间太久了，以至于在你们还没有意识到以前，我就已经从大部分的错误中调整过来了。

你需要多少反馈？

你应该编写多少测试呢？这儿有一个简单的问题：有三个整数，分别代表一个三角形三条边的长度，要求返回：

- （1）这个三角形是否是等边三角形
- （2）这个三角形是否是等腰三角形
- （3）这个三角形是否是不等边三角形

而且当这三条边不能构成三角形时，抛出异常。

来，试着解决一下这个问题（我的 Smalltalk 解决方案就列在这个问题的末尾）。

我写了 6 个测试（这话的意思有点像“定个基调”，诸如，“对付这个问题，我只要编写四个测试”，“直接编写解决这个问题的代码”）。Bob Binder 在其讲解全面的《Testing Object-Oriented Systems》^①一书中为同一个问题编写了 65 个测试。你得通过自己的经验和头脑来判断要编写多少个测试。

当我考虑要编写多少个测试的时候，就想起了平均无故障时间（MTBF）。例如，Smalltalk 整数的行为就像整数，而不像一个 32 位的计数器，因此测试 MAXINT 没有意义。不过，整数

^① Binder, Bob. 1999. Testing Object-Oriented Systems: Models, Patterns, and Tools. Boston: Addison-Wesley. ISBN 0201809389. 这是一本有关测试的比较全面的参考书。

确实有一个最大值，但那与你的内存容量有关。我有必要编写用巨大的整数填满整个内存空间的测试吗？编写此种测试与否对程序的 MTBF 会有怎样的影响呢？如果我从来不打算提供一个接近那种大小的三角形的话，那么编写与不编写这样的测试对程序的健壮性而言没有什么分别。

一个测试是否值得编写取决于你对 MTBF 衡量的仔细程度。如果你想让起搏器程序的 MTBF 从 10 年延伸至 100 年的话，那么对那些极不可能发生的条件和条件组合进行测试就是有意义的，除非你可以用别的方法证明这种条件不会出现。

测试驱动开发对测试的观点就是注重实效。在测试驱动开发中，测试从某种意义上说是一种达到目的的手段——达到充满自信地编写代码的目的。如果我们对实现有充分了解，不用测试就能拥有自信的话，那么就没有必要编写测试了。在黑盒测试中，我们有意识地抛开实现细节，这样做有一定的益处。通过不考虑具体的实现细节，黑盒测试展示了一种不同的价值系统——单单测试这块儿就是有价值的。考虑某些环境因素是一种适当的态度，但那与测试驱动开发有所不同。

TriangleTest

testEquilateral

```
self assert: (self evaluate: 2 side: 2 side: 2) = 1
```

testIsosceles

```
self assert: (self evaluate: 1 side: 2 side: 2) = 2
```

testScalene

```
self assert: (self evaluate: 2 side: 3 side: 4) = 3
```

testIrrational

```
[self evaluate: 1 side: 2 side: 3]
```

```
on: Exception
```

```
do: [:ex | ^self].
```

```
self fail
```

testNegative

```
[self evaluate: -1 side: 2 side: 2]
```

```
on: Exception
```

```
do: [:ex | ^self].
```

```
self fail
```

testStrings

```
[self evaluate: 'a' side: 'b' side: 'c']
```

```
on: Exception
```

```
do: [:ex | ^self].
```

```
self fail
```

evaluate: aNumber1 side: aNumber2 side: aNumber3

```
| sides |
```

```
sides := SortedCollection
```

```
with: aNumber1
```

```
with: aNumber2
```

```
with: aNumber3.
```

```
sides first <= 0 ifTrue: [self fail].
```

```
(sides at: 1) + (sides at: 2) <= (sides at: 3) ifTrue: [self fail].  
^sides asSet size
```

什么时候应该删除测试？

测试越多越好，但如果两个测试互为冗余，你应该保留它们两个吗？这就取决于下面两个标准。

- 第一个标准就是自信。如果删除一个测试降低了你对整个系统功能的信心，那么就不要删除。
- 第二个标准就是沟通。如果你有两个测试，走的是同一条路，但对读者来说讲述的是不同的情形的话，那么就应该原封不动地保留。

也就是说，如果你有两个测试，它们就自信和沟通而言都是冗余的，那就删掉其中用处最少的那个。

编程语言和环境怎样影响测试驱动开发呢？

尝试在 Smalltalk 中使用重构浏览器进行测试驱动开发，在 C++ 中用 vi 来进行测试驱动开发，感觉有何不同呢？

在测试驱动开发循环（测试 / 编译 / 运行 / 重构）难以实现的编程语言和环境中，你可能倾向于采用更大的步骤：

- 每个测试覆盖更多内容。
- 重构时使用较少的中间步骤。

这样做会使进度更快还是更慢？

在测试驱动开发循环丰富的编程语言和环境中，你更有可能尝试更多的试验。这样有助于更快地进行开发，或者获得更好的解决方案？不然的话还不如像以前的惯例那样，单独划出时间来进行纯粹的思考（代码审查或编写程序）？

你能采用 TDD 来开发大型系统吗？

测试驱动开发能够适应非常大系统的开发吗？你必须要编写哪些新的测试呢？需要什么新型的重构呢？

我所涉足的最大的、完全采用测试驱动的系统是在 LifeWare (www.lifeware.ch) 开发的。历经 4 年和 40 人/年的工作，该系统包含用 Smalltalk 编写的大约 250000 行的功能代码和 250000 行的测试代码，有 4000 个能够在 20 分钟内执行完毕的测试。整个测试集每天要运行多遍。系统众多的功能似乎与测试驱动开发的效果无关。在消除了重复设计以后，你往往会创建出更加小巧的对象，我们可以不用考虑应用的大小而单独对这些对象进行测试。

能使用应用级的测试来驱动开发吗？

用小规模测试（我称之为“单元测试”，但它们与大家认同的单元测试定义又不完全相同）

错扼腕叹息，但是不要去动那块代码。

其次，我们必须打破测试与重构之间的僵局。除了测试以外我们还可以通过其他方法得到反馈，比如说和同伴一起更加仔细地工作。我们可以获得粗粒度的反馈，例如尽管我们知道系统级测试并不充分，但它可以帮助我们获得一定程度的自信。有了这些反馈，我们就能够让那些必须要修改的地方更便于修改。

随着时间的流逝，系统中一直在改变的部分将会看起来像是测试驱动的。偶尔我们会走进没有灯光的黑暗的小巷，身陷困境，这使我们想起过去要做这些修改时动作会有多么缓慢。然后我们会减慢速度，打破僵局，再次继续前行。

测试驱动开发是供什么人使用的？

每种编程实践都直接或间接地蕴涵着一种价值系统。测试驱动开发也不例外。如果你乐于胡乱将代码堆砌在一起，觉得只要能工作就万事大吉了，那么测试驱动开发则不是供你使用的。测试驱动开发基于这样一种天真的呆子哲学：如果你编写的代码越好，你就越成功。测试驱动开发帮助你在适当的时间关注适当的问题，这样你的设计就更加清晰，同时能够在学习过程中对设计进行进一步的优化。

我说“天真”可能有点夸张。说其天真是因为它假定清晰的代码是通向成功的惟一道路。良好的工程实践可能只占到项目成功因素的 20%，糟糕的工程实践当然会导致项目的失败，但是只要其他的 80% 操作到位，普普通通的工程实践也能使项目获得成功。从这一点来看，测试驱动开发简直太棒了。它能使你写出的代码几乎没有错误，并且与该领域中通常的设计相比更加清晰。然而，对于那些灵魂经过优雅的洗礼的人来说，他们会在 TDD 中发现做对就能做好。

测试驱动开发非常适合那些对代码情有独钟的“呆子们”。我年轻时软件工程生活的一项最令人痛楚的事情就是满怀热情地开始一个项目，然后看着代码基随着时间的流失逐渐腐烂。一年后我一心只想丢掉已经变味儿的代码，转道开发别的项目去。测试驱动开发能够让你随着时间的流逝对代码依然信心依旧。随着测试的累积（测试的改进），你对系统的行为充满自信。在你改进设计的时候，有可能进行越来越多的改动。我的目标是在一年之后对项目拥有更透彻的把握，而不像刚开始时满脑子充满不切实际的幻想。测试驱动开发帮助我实现了这一切。

测试驱动开发对初始条件敏感吗？

似乎存在某种可以让工作更顺畅的编写测试的顺序：不可运行/可运行/重构/不可运行/可运行/重构。你可以采用不同的顺序来编写相同的测试，那样一来似乎没有办法实施小步推进。是不是真的存在一种顺序，实现起来可以相比其他顺序快上或容易一个数量级？是不是还有更好的推进顺序还没有被设计出来呢？测试中是否有某种东西能够告诉我们特定的处理顺序？如果测试驱动开发对初始条件的微小变化敏感的话，那么可以用以小见大的方式来预测吗？（同样，密西西比河中的小漩涡是不可预测的，但是在河口处测量出的每秒 2000000 立方英尺的流

量还是可以信赖的。)

测试驱动开发与模式的关系?

我所有的技术性的写作都在尝试找到可以产生类似专家行为的基础规则。这部分是源于我的学习方式——我首先效仿专家的行为特征,然后逐渐找出内在的原因。我当然不是寻找那些可以生搬硬套的规则,不过那些思维僵硬的人可能会机械地诠释我的论断。

我的大女儿(Bethany,你好!我说过我会在这儿提到你的——还好,没提什么让你不好意思的)花了好几年来学习更快地做乘法。而我和我的妻子则都能快速地完成乘法运算、学得快而自豪。这是怎么一回事呢?原来每次 Bethany 遇到 6×9 时,她都会把 6 加 9 次(或者我想是把 9 加 6 次)。计算乘法的速度不算慢,实际上是她的加法更快一些。

我发现存在这样一种效果:如果把重复的行为化归成一定的规则,那么应用这些规则直接生搬硬套就行了,期望别人也能注意到这一点。这比干什么都从第一定律开始快得多。当出现异常,或者出现现有规则无法处理的问题时,你就有更多的时间和精力来发挥和应用自己的创造能力。

我在写《Smalltalk Best Practice Patterns》时就有这样的亲身经历。当时我决定只按照自己的规则写作。开始时比较慢,因为要花费时间查找规则或者停下来编写新的规则。但是一个星期后,我发现代码从指尖飞速滑出,而这在以前是需要一个停笔思考的过程的。这给了我更多的时间和精力来考虑设计和分析方面更重要的问题。

测试驱动开发与模式的另一种联系就是前者可以作为一种模式驱动设计(pattern-driven design)的实现方法。比如说,我们决定在设计某样东西时要引入 Strategy 模式。我们为第一种变化形式编写了测试并将其作为一种方法予以实现。然后我们有意识地为第二种变化形式编写测试,期望重构阶段能够驱使我们采用 Strategy 模式。我和 Robert Martin 对这种形式的测试驱动开发进行了一番研究。问题在于设计往往让你意想不到。合情合理的设计思想往往最终被证明是错误的,最好是只考虑你希望系统所完成的功能,让好的设计逐渐自己“浮出水面”。

测试驱动开发为什么有效?

准备离开银河系。我们此刻先假设测试驱动开发帮助团队有效地构造松耦合、高内聚的系统,同时还具有瑕疵率低,维护成本低的特点。(我通常不会断言存在这样的系统,但是我相信你们能够想像出这种不可能存在的系统的样子来。)怎样才能开发出这样的系统呢?

我们所说的这种效果部分来源于减少出错。你越早发现并处理一个错误,成本就越低,通常效果十分显著(可以去问问火星探路者旅行器)。错误的减少继而可以带来大量心理和社会方面的影响。在我开始采用测试驱动开发以后,我自己在编程时的压力就减轻了不少,不再需要一时间为每件事情操心。我可以先让单个测试运行,然后才是剩余的部分。与同伴的关系变得更加积极了。我不再使程序构建(build)过程无法工作,而人们也可以依赖于我的软件来进行下一步的工作了。我所开发的系统的用户也变得更加积极了。发行的新版系统仅仅意味着

更强的功能，而不是随之而来的与老 bug 混在一起的许多新的错误（bug）。

错误率降低！你从哪里得出的这个论点呢？有科学证明吗？

没有。没有哪种研究明确地证实了测试驱动开发与任何一种其他的开发方法在质量、效果或者趣味性上存在差异。但是大家的亲身经历很能说明问题，况且 TDD 所带来的后续影响也不会有错。程序员们确实精神放松，团队真正提高了信任度，而且用户也真正学会了期待新的版本。尽管我还没有看到过负面的影响，但我还是会说：“总体而言就是这样的状况。”不同的人能力也各不相同，但是你得通过自己尝试才能感受到 TDD 的力量。

测试驱动开发的另一个能看出其效果的优点在于它能缩短设计决策的反馈循环。实现决策的反馈循环明显地缩短了——缩短到几秒或几分钟，接着我们每天要重复运行几十或者几百次测试。设计决策贯穿着设计思想——也许 API 应该是这个样子，也许我们应该那样比喻——而第一个例子就是一个包含设计思想的测试。我们不是先进行设计，并在等了几个星期或者几个月以后再让人们体会到设计的失败与成功之处，采用 TDD，在你努力将思想变成可行的接口时，几秒或者几分钟之内就能获得反馈。

对于“测试驱动开发为什么有效？”的一个不可思议的回答源于对复杂系统的狂热想像。无以伦比的 Philip 这样说：

将“吸引”正确代码的编程实践方法作为限定函数而不是绝对值来采纳。如果你为每个功能都编写单元测试，在每步之间都通过重构来简化代码，而且每次增加一种功能并且是在所有的测试都通过以后增加的话，那么你就创造出了一种数学家称之为“吸引子（attractor）”的东西。这是一个在状态空间中所有流聚集的点。随着时间的推移，你的代码会越来越好，而不是越变越糟；吸引子作为一种限定函数来解决代码正确性问题。

这是一种几乎所有程序员都采用类似方式获得的“正确性”（当然，医学或者宇航软件除外）。但是，清楚地理解吸引子这个概念总比否认它或者漠视其重要性要好。

TDD 这个术语有什么含义？

- 开发（Development）——过去那种与软件开发有关的阶段性思考方式日益衰落，因为时间所分隔的各项决策之间的反馈是困难的。这种意义上的开发意味着一种在分析、逻辑设计、物理设计、实现、测试、审阅、集成和部署之间复杂的跳转。
- 驱动（Driven）——我以前称呼测试驱动开发为“测试优先编程”。然而，与“优先”相对就是“最后”了，而很多人是在程序写好后才测试的。有一种命名规则说：一个名字的反义词应当至少是模模糊糊、不令人满意的（结构化编程的魅力就在于没人想要非结构化的东西）。如果你不用测试驱动开发，那么你用什么来驱动开发？推测（speculation）？还是规格说明书（specification）？（你注意到那两个词是源于同一个词根了吗？）
- 测试（Test）——自动、具体、切实的测试。按一个键就可以让测试运行。具有讽刺意味的是测试驱动开发不是一种测试技术（Cunningham Koan）。它是一种分析技术、

Darach 的挑战

Darach Ennis 对 TDD 的适用性的拓展提出了挑战。他说：

在各类工程部门和各种工程人员当中存在这样一种谬见，说这本书能够帮助读者解决 TDD 开发中存在的某些问题，其中的一些问题列举如下：

- 无法对 GUI（图形用户界面）应用实施自动测试（例如 Swing，CGI，JSP/Servlets/Struts 应用）
- 无法对分布式对象实施自动单元测试（例如 RPC 和消息传送方式，或者 CORBA/EJB 以及 JMS 应用）
- 无法采用测试优先的方法来开发数据库模式（例如 JDBC）
- 没有必要对第三方代码或外部工具生成的代码进行测试
- 无法采用测试优先的方法从 BNF 开发出产品质量的语言编译器/解释器

我不能肯定他是对的，也不能肯定他是错的。但他向我提供了一些在思考能在多大程度上推广 TDD 时需要仔细考虑的问题。

A

影响图

本书含有很多影响图（influence diagram）的图例。影响图的理念源于 Gerald Weinberg 所著的绝佳的《Quality Software Management》系列图书，尤其是第一册：《Systems Thinking》^①。影响图的目的是展示系统中一个因素对其他因素的影响。

影响图有三个要素：

- 活动，用单词或短语标记
- 正连接，在两个活动之间用有向箭头标记；表示源活动越多，所造成的目的活动也越多，或者是源活动越少，所造成的目的活动也越少
- 负连接，在两个活动之间用带有圆圈的有向箭头标记；表示源活动越多，所造成的目的活动越少，或者是源活动越少，所造成的目的活动越多

我们用了一大堆文字说明了一个简单的概念。图 A-1~A-3 提供了一些示例。



图 A-1 两个表面上不相关的活动

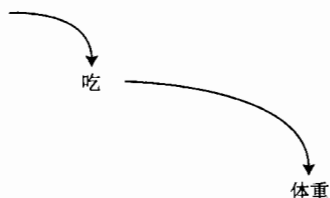


图 A-2 正连接的活动

^① Weinberg, Gerald. 1992. *Systems Thinking. Quality Software Management*. New York: Dorset House. ISBN: 0932633226.

吃得越多，体重越重。吃得越少，体重越轻。当然，影响个人体重的因素要比这一系统复杂得多。影响图是一种有助于你理解系统某一方面的模型，它无法帮助你全面、彻底地理解和控制系统。

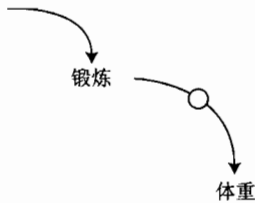


图 A-3 负连接的活动

反馈

影响并不只在一个方向起作用。通常一个活动的影响将反过来改变该活动，要么是正面要么是负面，如图 A-4 所示。

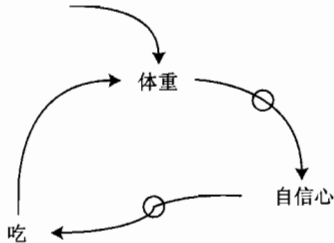


图 A-4 反馈

如果我的体重增加，那么我的自信心将下降，这会使我吃得更多，体重进一步增加，如此循环。只要影响图中有回路，就会有反馈。

有两种反馈——正反馈和负反馈。正反馈使得系统产生越来越多的活动。你可以通过查点回路中负连接的个数来找到正反馈环。如果存在偶数个负连接，那么得到的就是一个正反馈环。图 A-4 所示的反馈环就是一个正反馈环。它将使你的体重不断增加直到有其他活动介入为止。

负反馈抑制或减少活动。具有奇数个负连接的回路是负反馈环。

系统设计的关键是：

- 创建良性循环，使回路中的正反馈环促进有益活动的增长
- 避免死循环，回路中的正反馈会促使低效和具破坏性的活动增长
- 创建防止滥用有益活动的负反馈环

系统控制

当选择一种软件开发实践系统时,你一般喜欢那种相互支持的实践方法,这样即使处在压力下,一定数量的活动也能进行。图 A-5 演示了一种会导致测试不充分的系统实践活动。

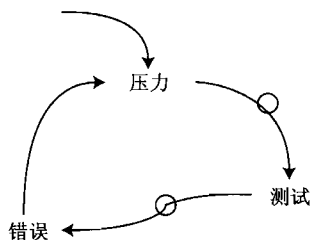


图 A-5 测试时间不足导致可用时间减少

在时间的压力下,你会减少测试的次数,这样会导致更多的错误,从而使时间更为紧张。最后,一些外部活动(如“对开发经费不足的担忧”)插足进来,导致盲目地发布软件。

当你的系统无法正常工作时,你确实还有其他的回旋余地:

- 改变一个正反馈环的方向。如果测试与自信度之间有回路,并且测试的失败已经使自信度减少,那么你可以通过让更多的测试工作起来来增强你對自己有能力让更多的测试工作起来的自信。
- 引入负反馈环来控制增长过多的活动。
- 创建或断开连接来消除无用的循环。

B

斐波纳契数列

作为对本书的一位审阅者所提问题的解答,我提供了一份采用测试驱动开发的斐波纳契数列。好几位审阅者都评论说该例子使得他们茅塞顿开,真正理解了 TDD 是如何工作的。然而这个例子的篇幅还不够长,而且对 TDD 的技术关键演示得还不够。如果你在看完本书主要的例子后认识还是很模糊的话,那么请到这里看看。

第一个测试显示 `fib(0)=0`。实现返回一个常量。

```
public void testFibonacci() {
    assertEquals(0, fib(0));
}

int fib(int n) {
    return 0;
}
```

(我仅仅以 `TestCase` 类作为代码的出发点,因为我们要开发的对象只是一个函数。)

第二个测试显示 `fib(1)=1`。

```
public void testFibonacci() {
    assertEquals(0, fib(0));
    assertEquals(1, fib(1));
}
```

我把第二条断言放在同一个方法中,因为编写一个 `testFibonacciOfOneIsOne` 实在没有什么太大的意义。

让这个测试运行有好几种方法。我选择把 0 当作特殊情况对待:

```
int fib(int n) {
    if (n == 0) return 0;
    return 1;
}
```

测试用例中的重复开始令人生厌了,再增加新的测试用例,只会使情况变得更糟。通过使用一张包含输入和期望输出值的表来驱动测试,我们可以分离出断言的公共结构。

```
public void testFibonacci() {
    int cases[][] = {{0,0},{1,1}};
    for (int i = 0; i < cases.length; i++)
```

```
    assertEquals(cases[i][1], fib(cases[i][0]));  
}
```

现在增加一个测试用例只需六次击键并且不用再增加其他的行。

```
public void testFibonacci() {  
    int cases[][] = {{0,0},{1,1},{2,1}};  
    for (int i = 0; i < cases.length; i++)  
        assertEquals(cases[i][1], fib(cases[i][0]));  
}
```

战战兢兢中，测试程序运行通过了。无独有偶，常量 1 也适合这个测试。现在继续下一测试：

```
public void testFibonacci() {  
    int cases[][] = {{0,0},{1,1},{2,1},{3,2}};  
    for (int i = 0; i < cases.length; i++)  
        assertEquals(cases[i][1], fib(cases[i][0]));  
}
```

万岁，终于失败了！如法炮制（把小的输入当作特殊情况），我们这样写：

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    return 2;  
}
```

现在我们可以进行一般化了。我们写 2，但真的用意并不是 2，而是 1+1。

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    return 1 + 1;  
}
```

第一个 1 是 $\text{fib}(n-1)$ 的一个实例：

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    return fib(n-1) + 1;  
}
```

第二个 1 是 $\text{fib}(n-2)$ 的一个实例：

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    return fib(n-1) + fib(n-2);  
}
```

整理一下，同样的结构也适用于 $\text{fib}(2)$ ，因此我们加强了第二个条件：

```
int fib(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fib(n-1) + fib(n-2);  
}
```

如此我们得到了整个斐波纳契数列，完全来源于测试。