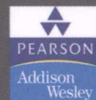
获2008 Jolt Award技术图书类生产力大奖



xUnit Test Patterns Refactoring Test Code

xUnit测试模式

——测试码重构

(美) Gerard Meszaros 著
付 勇 译

清华大学出版社

xUnit 测试模式
——测试码重构

(美) Gerard Meszaros 著
付 勇 译

清华大学出版社
北 京

Authorized translation from the English language edition, entitled xUnit Test Patterns Refactoring Test Code, 978-0-13-149505-0 by Gerard Meszaros, published by Pearson Education, Inc., publishing as Addison-Wesley, Copyright © 2007.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and TSINGHUA UNIVERSITY PRESS Copyright © 2008.

北京市版权局著作权合同登记号　图字：01-2008-0462

本书封面贴有 Pearson Education(培生教育出版集团)防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

xUnit 测试模式——测试码重构/(美)梅扎罗斯(Meszaros, G.) 著；付勇 译.
—北京：清华大学出版社，2009.1

书名原文：xUnit Test Patterns Refactoring Test Code

ISBN 978-7-302-19138-4

I. x…　II. ① 梅… ② 付…　III. 软件—测试　IV.TP311.5

中国版本图书馆 CIP 数据核字(2008)第 200977 号

责任编辑：王　军　梁卫红
装帧设计：孔祥丰
责任校对：胡雁翎
责任印制：王秀菊
出版发行：清华大学出版社　　地　址：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　邮　编：100084
社　总　机：010-62770175　　邮　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市新茂装订有限公司
经　销：全国新华书店
开　本：185×260　印　张：42.75　字　数：988 千字
版　次：2009 年 1 月第 1 版　　印　次：2009 年 1 月第1 次印刷
印　数：1～4000
定　价：86.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770177 转 3103　产品编号：027048-01

编者序

如果访问 junit.org，您就会看到我的一句名言：“在软件开发领域，如此少的几行代码，作用如此巨大，真是从未有过”。有人批评说 JUnit 并不重要，任何理性的编程人员用一个周末就可以把它创造出来。这没有错，但绝对遗漏了重点。JUnit 之所以重要并受到丘吉尔式的评价，是因为这个小工具对于许多编程人员的根本性转变至关重要：测试移动到了编程前沿和中心位置。人们以前提倡它，但 JUnit 使得它真正成为现实。

当然，不仅仅是 JUnit，我们已经为许多编程语言编写了 JUnit 端口。这个开放的工具家族(通常称为 xUnit 工具)已经从它的 Java 根源广泛蔓延开来(当然，根源实际上不是用 Java 写的——Kent Beck 数年前用 Smalltalk 写过代码)。

对于 xUnit 工具，更重要的是它们的基本原理，为编程团队提供了大量机会——编写强大回归测试套件的机会，这些套件允许团队以较小风险对代码库做出重大变更，以及使用测试驱动开发重新思考设计过程的机会。

但伴随这些机会而来的是新问题和新方法。和所有工具一样，可以很好地利用 xUnit 家族，也可能利用不好。经过深思熟虑的人已经找出各种方法来有效使用 xUnit、组织测试和数据。和早期的对象一样，使用工具的许多知识都隐藏在熟练用户的大脑中。没有这些隐含知识，就不能发挥它们的最大作用。

大概 20 年前，面向对象团体中的人们就已经意识到了对象的这个问题，并开始寻找答案。答案就是用模式的形式描述它们隐藏的知识。Gerard Meszaros 是这方面的先驱之一。我开始研究模式时，Gerard 是我学习的领路人之一。和模式世界里的许多人一样，Gerard 早期也采用极限编程，因此最早使用 xUnit 工具。因此，由它担负用模式的形式来持久保存那些专业知识的任务完全合情合理。

我第一次听说这个项目时异常兴奋(我真想发动一次突袭，从 Bob Martin 那里偷来这本书，因为我想让这本书为我的丛书增光添彩)。

和其他优秀的模式书籍一样，这本书为该领域的新手提供这些知识，最重要的是，为有经验的从业者提供词汇和基础，从而让他们能够将自己的知识传授给同事。对于许多人而言，著名的“四人帮”编写的 *Design Patterns* 一书让面向对象设计发出了夺目的光芒。这本书同样适用于 xUnit。

Martin Fowler

丛书编辑

ThoughtWorks 首席科学家

作者序

自测试代码的价值

在 *Refactoring* [Ref]的第 4 章，Martin Fowler 写道：

> 如果观察大多数编程人员如何花费时间，你就会发现写代码实际上只占一小部分。有些时间用于确定下一步应该做什么，有些时间用于设计，但大多数时间会用于调试。我确信每位读者都忘不了长时间的调试，通常是要忙到深夜。每位编程人员都有花费一整天(甚至更长时间)来找出缺陷的经历。修复缺陷通常很快，但找到它却非常困难。而当你修复一个缺陷之后，总有可能出现另一个缺陷，你甚至要到很久之后才会注意到它，然后你要花费几年时间来找出这个缺陷。

有些软件很难手动测试。在这些情况下，通常要求我们写测试程序。

我回想起 1996 年做过的一个项目。我的任务是构建一个事件架构，它让客户软件注册一个事件，当其他软件产生这个事件时(观察者[GOF]模式)，能够得到通知。除了写一些样本客户软件之外，我想不出还有方法能够测试这种架构。我有 20 个需要测试的不同场景，因此我用必要数量的观察者、事件和事件产生者编码每种场景。首先，我记录控制台中发生的情况，并手动扫描它，很快这种扫描就变得单调乏味。

因为我很懒散，自然而然就会寻找实现这种测试的更简单方法。对于每个测试而言，我都用接收器的名称作为值填充预期事件和预期接收器编入索引的 Dictionary。当特定接收器发现事件时，它会在 Dictionary 中查找它自己索引的条目和它刚刚接收到的事件。如果条目存在，接收器删除该条目；如果它不存在，接收器就会添加错误消息条目，说明它是意外事件通告。

运行完所有测试之后，测试程序会查看 Dictionary 并打印出其内容(如果它不是空的话)。因此，运行所有测试成本几乎为零。测试要么通过，要么产生测试失败列表。我无意间发现仿制对象(Mock Object)和测试自动化架构(Test Automation Framework)的概念非常重要！

试读结束：需要全本请在线购买： www.ertongbook.com

我的第一个 XP 项目

1999 年下半年，我参加了 OOPSLA 会议，当时我看到了 Kent Beck 的新书 *eXtreme Programming Explained*[XPE]。我习惯于进行迭代和增量开发，相信自动单元测试的价值，尽管我还没有广泛使用它。我很尊敬 Kent，在 1994 年第一届 PLoP[1] 会议上我就认识他。因为这些原因，我相信值得在 ClearStream Consulting 项目上试着应用一下极限编程。OOPSLA 之后不久，我有幸遇到了一个适合应用这种开发方法的项目——即增件应用程序，它与现有数据库交互，但没有用户接口。客户可以用不同方法开发软件。

我“依据这本书”开始进行极限编程，使用了这本书里推荐的很多方法，包括结对编程、集体所有制和测试驱动开发。当然，在理解如何测试应用程序行为的某些方面时，我遇到了一些挑战，但我还是尽量为大多数代码写测试。这样随着项目的进展，我开始注意到一种不良趋势：要花费越来越长的时间实现几乎相同的任务。

我向开发人员解释了这个问题，并要求他们在任务卡上记录写新测试、修改已有测试和写产品代码上花费的时间。很快问题出现了，写新测试和写产品代码所花费的时间大概是个常量，而修改已有测试所花费的时间不断增加，因此开发人员的估计也不断增加。当开发人员要求我配对任务，而我们花费 90%的时间修改已有测试以适应相对次要的变更时，我知道我们必须变革，而且要快！

引入新功能时我们会遇到各种编译错误和测试失败，在分析这些错误和失败之后我们发现，对被测系统(SUT)方法的变更影响了许多测试。当然，这并不奇怪。奇怪的是大多数影响发生在测试的夹具建立阶段过程中，而且这些变更不影响测试的核心逻辑。

这是一个重大发现，因为它让我们知道如何创建分散在大多数测试中的 SUT 的对象。换句话说，测试非常了解 SUT 行为的非本质部分。我说“非本质”是因为大多数受影响的测试不关注如何创建夹具中的对象；它们关心的是确保这些对象处于正确的状态。经过进一步检查我们发现，许多测试会在它们的测试夹具中创建相同或几乎相同的对象。

这个问题最明显的解决方案是将逻辑分解到一组测试实用程序方法中。有一些变体：

- 当测试需要相同对象时，我们只创建一种方法，它返回这一类准备使用的对象。现在称这些方法为“创建方法”。
- 有些测试需要给对象的某个属性指定不同值。在这些情况下，我们作为参数化创建方法(参见“创建方法”)的参数传递该属性。
- 有些测试需要创建畸形的对象以确保 SUT 拒绝它。为扰乱测试辅助签名的每个属性写单独的参数化创建方法，因此我们创建有效对象，然后取代一种坏属性(参见“派生值”)的值。

我们已经发现会成为[2]我们第一个测试自动化模式的东西。

后来，测试开始失败，因为数据库不接受我们插入有相同键码(该键码有唯一性约束)的另一个对象，因此我们添加代码以便从程序上生成唯一键。我们将这种变体称为“匿

1 程序模式语言会议(The Pattern Languages of Programs conference)。
2 从技术上说，在三个独立项目小组发现它们之前，它们不是真正的模式。

名创建方法”(参见“创建方法”)来表示存在添加的行为。

确定现在称为“脆弱测试”的这个问题是项目的重要事件，其解决方案模式的后续定义防止项目产生可能的失败。如果没有这个发现，我们最多放弃已经构建的自动化单元测试。最糟糕的是，测试会大大降低生产力，让我们不能将承诺提交给客户。事实证明，我们能够高质量地提交承诺。是的，测试人员[3]还会在我们的代码中找到缺陷，因为我们肯定遗漏了某些测试。然而，在我们弄清楚遗漏的测试之后，引入所需的变更来修复这些缺陷就是一个相对简单的过程。

我们乐此不疲。自动单元测试和测试驱动开发真的不错，从那时起我们就一直使用它们。

在后续项目上应用这些实践和模式时，我们也遇到了新的问题和挑战。在每种情况下，我们像“剥洋葱”一样寻找根本原因，并提出解决它的方法。当这些方法成熟之后，我们就将它们添加到自动化单元测试的方法库中。

我们在 XP2001 上提交的论文中首次描述了这些模式。在与那一届以及后续会议参与者的讨论中我们发现，许多同行使用相同或类似的方法。这将我们的方法从“实践”提升为“模式”(对上下文中复发问题的复发解决方案)。那届会议上还首次发表了有关测试味道[RTC]的论文，该论文建立在[Ref]中首次描述的代码味道的概念之上。

我的动机

我完全相信自动化单元测试的价值。我在没有使用它的条件下进行了 20 年的软件开发，但我知道我的职业生活因为有它而更精彩。我相信 xUnit 架构和它支持的自动化测试在软件开发中是真正的进步。当我看到有些公司试图采用自动单元测试但由于缺乏关键信息和技能而失败时，我感到很遗憾。

作为 ClearStream Consulting 的软件开发顾问，我见过许多项目。有时别人在项目初期让我帮助客户确定他们“做得对”。然而，通常是出现了问题才想起我。因此我见过许多导致测试味道的“最坏实践”。如果我够幸运并且通知我的时间足够早，我可以帮助客户从错误中恢复过来。如果不是这样，客户可能会在对 TDD 和自动化单元测试的运行方法不太满意的情况下混过去——自动化单元测试是浪费时间这句话已经过时了。

在事后才知道，只要在正确的时间有正确的知识，就很容易避免大多数错误和最坏实践。但如果不犯错误，又如何获得这些知识呢？背负自私自利的名声，雇用有这些知识的人是学习新实践或技术最节省时间的方法。依据 Gerry Weinberg 的“Raspberry Jam 法则”[SoC][4]，上课或读书是没有效率的选择(尽管这些方法更便宜)。我希望通过写下这些错误并提出避免它们的方法，能够减少项目上的失误，不管它是完全敏捷还是比过去稍微更加敏捷——“Raspberry Jam 法则”经不起考验。

3 测试功能有时称为“质量保证”。严格来说，这种用法是错误的。
4 Raspberry Jam 法则：“摊得越大就越薄”。

本书的读者对象

本书主要面对的是想写更好的测试的软件开发人员(编程人员、设计人员和结构师)，以及管理人员和教练——他们需要理解开发人员在做什么。这里重点关注的是使用 xUnit 自动化的开发人员测试和客户测试。另外，有些高级模式也可应用于使用其他方法(除了 xUnit 之外)进行自动化的测试。Rick Mugridge 和 Ward Cunningham 合著了一本关于 Fit [FitB]的书，他们支持许多相同的实践。

开发人员可能想一页一页地阅读这本书，但他们应该着重略读相关章节而不是逐字阅读它们。重点应该放在整体把握存在哪些模式以及它们如何运行上。在需要的时候，再返回查找特定模式。每种模式的前几个部分应该提供这种概述。

管理者和教练可以着重阅读第 I 部分“总述”和第 II 部分“测试味道”。他们也应该阅读第 18 章“测试策略模式”，因为这些是他们需要理解的决策，当他们努力实现这些模式时，会给开发人员提供支持。管理者至少应该阅读第 3 章“测试自动化的目标”。

前　　言

虽然前面已经介绍过，但这里还是需要重复一下：编写没有缺陷的软件非常困难。实际系统正确性的证明超出了我们的能力，行为规范也同样具有挑战性。预知未来需要或不需要成为可能——如果我们擅长于此，应该会在股市上发财而不是还在构建软件系统。

软件行为的自动验证是最近几十年开发方法方面最大的进展之一。开发人员友好的实践对增强生产力、提高质量、防止软件变得脆弱等方面具有很大帮助。现在很多开发人员出于自愿进行这种实践，这一点也将说明其效能。

本章将介绍使用各种工具(包括 xUnit)进行测试自动化的概念，说明这样做的原因，并阐述更好地进行测试自动化所面临的困难。

I.1　反馈

在许多活动中，反馈都非常重要。反馈可以说明我们的动作是否产生了正确的效果。获得反馈越快，反应也就更快。这种反馈的一个好的示例是许多高速公路中主路面与路肩之间的隆声带。是的，驶离路肩给我们提供一个反馈，说明我们已经离开主路面。但获得反馈越早(当车轮刚刚进入路肩时)，就可以给我们提供越多的时间来更正线路，从而减少冲出道路的可能性。

测试就是获得软件的反馈。因此，反馈是“敏捷”或“精益”软件开发最本质的元素之一。在开发过程中包含反馈循环可以提高对自己所写软件的信心。它让我们能够更快运行，同时又不会变得偏执。当我们暂停旧功能时，它通过测试告诉我们，从而让我们可以重点关注要添加的新功能。

I.2　测试

“测试”的传统定义来自质量保证领域。要测试软件是因为我们确信它里面有缺陷。

因此我们测试、测试、再测试，直到不能证明软件里有缺陷为止。从传统上说，这种测试出现在软件完成之后。因此，它是衡量质量的方法，而不是建立产品质量的方法。在许多组织中，测试由软件开发之外的人完成。这种类型的测试所提供的反馈非常有价值，但它在开发周期中出现得太晚了，以至于让它的价值大打折扣。它也会产生令人讨厌的效果：随着发现的问题被传递回来重做(紧接着是另一个测试周期)，它会延缓进度。因此，软件开发人员应该进行什么类型的测试以尽早获得反馈呢？

I.3 开发人员测试

几乎没有软件开发人员相信自己写的代码“第一次就能运行，每次都能运行”。实际上，我们大多数人发现它第一次就能运行时都会觉得非常奇怪(我希望没有粉碎其他非开发人员读者的幻想)。

因此开发人员也会进行测试。我们想证明，软件会像预期的那样运行。有些开发人员进行测试的方法与测试者进行测试的方法一样：作为单个实体测试整个系统。然而，大多数开发人员喜欢逐个单元地测试自己的软件。“单元”可能是大粒度组件，也可能是单个类、方法或函数。这些测试与测试者写的测试之间的主要不同是，要测试的单元是软件设计的结果，而不是需求的直接转换[1]。

I.4 自动测试

自动测试已经出现几十年了。20 世纪 80 年代早期，当我在加拿大北电研发中心(Nortel R&D)下属的贝尔北方研究中心研究电话交换系统时，我就对自己构建的软件/硬件进行过自动回归测试和负载测试。这种测试主要在“系统测试”组织上下文中，使用利用测试脚本编程的专用硬件和软件完成。测试机器连接到被测试的开关，好像它就是一捆电话和其他电话开关，它通过打电话，检验大量电话特征。当然，这种自动测试基础组织不适合单元测试，通常也不提供给开发人员，因为它包含太多硬件。

最近几十年出现了一些多用途测试自动化工具，我们可以通过它们的用户接口测试应用程序。其中有些工具使用脚本语言来定义测试，更吸引人的工具使用测试自动化的“机器人用户”或“记录与回放”隐喻。遗憾的是，使用后面这些工具的早期经历让测试人员和测试管理员不太满意。原因就是由“脆弱测试”问题导致的高测试维护成本。

1 小部分的单元测试可以直接对应需求和客户测试中描述的业务逻辑，但大部分会测试业务逻辑周围的代码。

I.4.1 "脆弱测试"问题

使用商业"记录与回放"或"机器人用户"工具的测试自动化在这些工具的早期用户中名声狼籍。使用这种方法的自动化测试通常因为看起来不太重要的原因而失败。重要的是要理解这类测试自动化的局限性，以免落入与它相关的陷阱——即行为敏感性、接口敏感性、数据敏感性和上下文敏感性。

1. 行为敏感性

如果系统的行为发生改变(例如，如果需求改变，并且系统被修改以满足新需求)，重放时，执行修改后功能性的测试很可能失败[2]。不管使用什么测试自动化方法，这就是测试的基本事实。实际问题是，通常需要使用这种功能性将系统设置到正确状态来启动测试。因此，行为变更对测试过程的影响比人们预期的更大。

2. 接口敏感性

通过用户接口测试被测系统(SUT)内的业务逻辑不是一个好主意。甚至是对接口的细小变更都可以导致测试失败，虽然人类用户可能说测试仍然应该通过。在过去几十年内，这种无意识的接口敏感性给测试自动化工具带来了不好的名声。不管使用哪种用户接口方法，总是会出现问题，有些类型的接口似乎更糟糕。图形用户界面(GUI)是与系统内业务逻辑交互的特别具有挑战性的方法。最近使用基于Web(HTML)的用户界面让测试自动化的某些方面变得更容易，但也产生了另一个问题，因为HTML内的可执行代码需要提供丰富的用户经验。

3. 数据敏感性

所有测试都要假设某个起始点，这个起始点称为测试夹具。测试上下文有时称为测试的"前置条件"或"之前状态"。通常依据系统中已有的数据定义测试夹具。如果数据改变，测试就可能失败，除非努力让它们对使用的数据不敏感。

4. 上下文敏感性

系统的行为可能受到系统之外事物状态的影响。这些外部因素可能包括设备(例如，打印机、服务器)的状态、其他应用程序或系统时钟(例如，测试执行的时间和/或日期)。如果没有控制上下文，受该上下文影响的所有测试都很难确定地重复。

I.4.2 克服四种敏感性

不管使用哪种方法自动化测试，都会存在这四种敏感性。当然，有些方法能够绕开这些敏感性，而其他的则要求我们通过特殊路径。测试自动化架构的 xUnit 家族提供了

---

2 行为可能出现变更，因为系统要做的事情不同，或者因为它使用不同的计时或排序方法做相同的事情。

试读结束：需要全本请在线购买： www.ertongbook.com

深层控制，我们只要学习如何有效使用它就行了。

I.5 使用自动测试

到目前为止，大多数讨论都集中于应用程序的回归测试。当修改现有应用程序时，这是一种非常有价值的反馈形式，因为它有助于捕获我们无意引入的缺陷。

I.5.1 作为规范测试

测试驱动开发(TDD)中会使用自动化测试另一种完全不同的用法，它是像极限编程这样的敏捷方法的核心实践之一。自动测试的这种用法不仅与回归测试有关，更多的则是与要写的软件的行为规范有关。TDD 的效能来源于它让我们将对软件的思考分为两个单独阶段：它应该做什么和它应该怎么做。

等一下！敏捷软件开发的支持者不是回避瀑布式开发吗？的确是这样。敏捷支持者喜欢按照逐个特性设计和构建系统，让每一步都可以得到工作软件，以此证明在继续开发下一种特性之前各特性都正常。这并不意味着我们不能进行设计，而是意味着应该进行“连续设计”！将它推向极端会导致“紧急设计”——几乎没有提前进行任何设计。但不必像那样进行开发。可以按照一个特性一个特性的方式，将前面的高级设计(或体系结构)与详细的设计结合起来。不管哪种方式，捕获该行为应该是什么形式的可执行规范时，将思考如何实现专用类或方法的行为延迟几分钟很有用。毕竟，我们大多数人在每次集中关注一件事时会有困难，同时也会放弃一些事情。

完成写测试并且验证它们如期望的那样失败之后，就可以将我们的注意力转换为着重关注让它们通过。现在测试就是进度度量。如果增量地实现功能性，当写更多代码时，就会发现测试会一个接一个通过。运行时，我们将以前写的所有测试作为回归测试运行，以确保我们的变更没有任何意料之外的副作用。这是自动单元测试真正的价值所在：它能够“约束”SUT 的功能性，以便该功能性不会意外改变。这样我们就能睡个好觉！

I.5.2 测试驱动开发

最近有很多书在讨论测试驱动开发这个主题，因此本书不会用大量篇幅来讨论它。本书着重讨论测试中代码的形式，而不是写测试的时机。谈到测试如何产生，最关注的是何时研究测试重构以及学习如何将使用一种模式写的测试重构为使用具有不同特性模式的测试。

本书中我尽量支持“开发过程不可知论者”，因为自动化测试有助于所有团队，不管其成员是进行 TDD、测试优先开发还是测试最后开发。同时，人们学习如何自动化“最后测试”环境中的测试之后，他们很可能会倾向于试验“测试优先”方法。而且，我们的确探索了开发过程的某些部分，因为它们影响进行测试自动化的难易程度。这项研究主要包括两个方面：(1)全自动测试与开发集成过程和工具之间的相互影响；(2)开发过程

影响设计易测性的方式。

I.6 模式

在准备写这本书的过程中，我阅读了许多有关基于 xUnit 的测试自动化方面的会议文献和书籍。每位作者似乎都有特别感兴趣的领域和最喜欢的方法，这并不奇怪。我不一定同意他们的实践，但我总是尽量理解为什么这些作者会使用特殊方法完成任务，以及什么时候使用他们的方法比使用我的方法更合适。

这种层次的理解是示例与只解释方法和模式“如何”的散文之间的主要差异之一。这种模式有助于读者理解实践背后的原因，允许他们在可选模式之间做出明智选择，从而避免将来意料之外的不良结果。

软件模式已经出现了 10 年，因此大多数读者至少应该知道这个概念。模式是“复发问题的解决方案”。有些问题比其他问题严重，以至于用单个模式不足以解决它。这时就出现了模式语言，这种模式集合(或语法)将读者从整个问题一步步引导到详细的解决方案。在模式语言中，有些模式必然有更高的抽象级别，而其他的则专注于低级细节。为了便于使用，模式之间必须有联接，以便我们可以从高级“策略”模式逐步过渡到更详细的“设计模式”和更详细的“编码习语”。

I.6.1 模式、原则与味道

本书包括三种模式。最传统的一类模式是“复发问题的解决方案”，本书中的大多数模式属于这种通用类别。下面分析三种不同级别之间的差异：

- “策略”级模式有深远结果。使用共享夹具取代新鲜夹具的决定让我们走上了另一条道路，导致产生了不同的测试设计模式的集合。在本书的“策略模式”章节中，对每种策略模式都有详细的介绍。
- 为专用功能性开发测试时通常使用测试“设计”级模式。它们着重关注如何组织测试逻辑。大多数读者熟悉的示例之一是仿制对象模式。每种测试设计模式都有各自的介绍，本书的相关章节按照主题(例如测试替身模式)对这些模式进行了分组。
- 测试“编码习语”介绍了编码特定测试的不同方法。许多习语都是语言专有的，比如在 Smalltalk 使用代码块闭包表示预期异常结果，在 Java 中使用匿名内部类表示仿制对象。有些习语，例如简单成功测试(参见“测试方法”)，相当通用，因为它们在各种语言中都有类似词语。在“测试设计模式”的介绍中，这些习语通常作为实现方式变体或示例列出。

通常，有几种可选择模式可用于各个级别。当然，我总是有使用某些模式的偏好，但一个人的“反模式”可能是另一个人的“最佳实践模式”。因此，本书也包含了一些我

不一定提倡的模式。本书描述了这些模式的优缺点，让读者自己决定是否使用它们。在每种模式描述以及引言中，我介绍了这些可选项之间的联接。

有关模式的理想情况是，它们提供足够信息以便读者能够在几种可选项之间做出明智决定。选择的模式可能受我们确定的测试自动化目标的影响，这些目标描述了测试自动化努力的预期结果。这些目标由一些原则支持，这些原则构建有关什么让自动测试“不错”的信任系统。本书中，第 3 章“测试自动化的目标”介绍了测试自动化的目标，第 5 章“测试自动化的原则”介绍了测试自动化的原则。

最后一类模式是反模式[AP]。这些测试味道描述了复发问题，模式有助于我们依据观察到的这些症状及其根本原因来解决这些问题。在 Martin Fowler 的书[Ref]中首次普及了代码味道的概念，并在 XP2001 [RTC]发表的论文中，作为测试味道应用于基于 xUnit 的测试。测试味道与可用来消除它们的模式以及可能导致它们的模式[3]前后对照[4]。第 II 部分“测试味道”详细讨论了测试味道。

I.6.2 模式形式

本书包括我对这些模式的描述。在我对它们进行分类之前，模式本身就已经存在，至少三位不同的测试自动化人员各自发明了它们。我大胆地写下它们，目的是为了使知识方便分类。但要这样做，我必须选择一种模式描述形式。

模式描述有许多形式和大小。有些模式描述的结构非常严谨，由许多有助于读者找到各种章节的标题所定义。有些则更像文学，而很难作为参考资料使用。尽管如此，但所有模式都提供了通用的核心信息。

1. 我的模式形式

我很喜欢阅读 Martin Fowler 的著作，我的许多快乐都来源于他使用的模式形式。正如谚语所说“模仿是最真诚的欣赏”。我妄自对他的格式做了一些小小修改。

模板首先是问题语句、概述语句和示意图。斜体的问题语句概述了模式处理的问题的核心。它通常表现为“如何…?”这个问题。黑体概述语句可以抓住一到两个句子中模式的本质，而示意图是模式的可视表示法。示意图之后的无标题文本概述了只在一些句子中使用这种模式的原因。它详细阐述了问题语句，包括传统模式的“问题”和“上下文”部分。读者通过略读这一部分就可以知道是否需要进一步阅读。

接下来三节介绍了模式的实质内容。“运行原理”这一节介绍了如何构造模式以及它的本质。如果有几种方法实现该模式的某些重要方面，它还包含有关“导致结果的上下文”的信息。该节对应于传统模式形式的“解决方案”或“所以”这两节。“使用时机”这一节描述应该考虑使用模式的环境。这一节对应于传统模式模板的“问题”、“要求”、“上下文”和“相关模式”这几节。它还包括有关“导致结果的上下文”的信息，如果

---

3 有些人将这些模式称为“反模式”。只是模式通常有不好的结果，但并不意味着该模式总是不好。因此，我不将这些模式称为反模式，我只是不经常使用它们而已。

4 有些情况下，模式和味道都有类似的名称。

该信息可能影响是否要使用这种模式的话。我还介绍了所有"测试味道"，这些味道建议你应该使用哪种模式。"实现方式说明"这一节介绍了实现这些模式的具体细节。这一节的次级标题表示模式的主要组件或实现这种模式的变体。

大多数具体模式包括三个附加部分。"启发示例"这一节提供了一些示例，用来说明应用该模式之前测试码的状态。"示例：{模式名称}"这一节显示应用模式之后测试的状态。"重构说明"这一节更详细地说明了如何从"启发示例"转换为"示例：{模式名称}"。

如果要在其他地方补写模式，描述还包括"高级阅读"这一节。如果这些应用有一些特别有意思的东西，就会附加"已知使用"这一节。当然，许多系统中都存在大多数模式，因此挑选三种用法来代替它们比较武断，也没意义。

如果有许多相关方法存在，通常将它们表示为具有一些变体的单个模式。如果变体是实现相同功能模式的不同方法(也就是说，用相同的通用方法解决相同问题)，"实现方式说明"这一节就会列出这些变体以及它们之间的差异。如果变体是使用模式的主要不同原因，"使用时机"这一节就会列出这些变体。

I.6.3　过去用过的模式和味道

在打算提出模式和味道的简明列表又想尽可能保持它们过去用过的名称时，我经历了激烈的思想斗争。我通常作为别名列出模式或味道过去用过的名称。有些情况下，最好将模式过去用过的版本当成更大模式的特定变体。在这些情况下，我通常在"实现方式说明"这一节作为命名变体包含过去用过的模式。

许多过去用过的味道没有通过"可闻性测试"——也就是说，味道描述根本原因而不是症状[5]。当过去用过的测试味道描述原因而不是症状时，我选择将它作为一种名为"原因"的特定类型的变体移入相关基于症状的味道中。神秘访客(参见"模糊测试")就是一个好的示例。

I.6.4　引用模式和味道

我努力提出引用模式和味道的好方法，特别是引用过去用过的模式和味道的方法。我既想在合适的时候使用过去用过的名称，也想使用新的集合的名称，总之是要使用更合适的那一个名称。我也想让读者能够明白哪个是哪个。在本书的联机版本中，超链接就是这个目的。然而，对于印刷版本而言，我将这种链接表示为引用的页码注释，同时又不会让引用打乱整个文本。几次尝试之后我采取的解决方案是，在章节、模式或味道中第一次引用模式或味道时，注释可以查找到它的页码。如果引用到模式的变体或味道的原因，第一次会包含集合模式或味道名称。注意，第二次引用到模糊测试的神秘访客原因时，没有显示味道名称，而引用到模糊测试的其他原因，例如不相关信息(参见"模糊测试")时，包含的是集合味道名称而不是页码。

---

5 "可闻性测试"来源于[Ref]中的尿布故事，当时 Kent Beck 问 Grandma Beck，"我怎么知道要更换尿布？""如果它有臭味，就更换它。"她回答说。味道基于"臭味"而不是臭味的原因而命名。

I.7 重构

在软件开发中，重构是相对较新的概念。人们总是需要修改现有代码，而重构是改变设计而不会改变代码行为的严谨方法。它与自动化测试紧密相关，因为如果没有自动化测试的安全网来证明在重新设计过程中没有中断任何进程，就很难进行重构。

许多现代集成开发环境(IDE)内置对重构的支持。它们大多数至少自动化 Martin Fowler 在书[Ref]中描述的一些重构的重构步骤。遗憾的是，这些工具没有告诉我们何时或为什么要使用重构。因此我们不得不复制 Martin 的书！关于这个主题另一个必读材料是 Joshua Kerievsky 写的[RtP]一书。

重构测试有点不同于重构产品代码，因为我们没有自动化测试的自动化测试。如果测试在测试重构之后失败，是因为在重构过程中出现了错误才导致失败吗？测试在测试重构之后通过，我们就能够确信在适当时候它不会失败吗？为了解决这个问题，许多测试重构是非常保守的“安全重构”，它们让将行为改变引入测试的可能性减到最低限度。我通过采用适当的测试策略(如第 6 章“测试自动化策略”所述)，尽量避免对测试进行重要重构。

本书着重介绍重构的目标而不是这种努力的技巧。附录 A 对重构进行了简要小结，而重构过程不是本书讨论的重点。模式本身都很新，我们也没有时间来统一它们的名称、内容或适用性，更不用说能够在重构到它们的最好方法方面达成一致了。更复杂的是，每个重构目标(模式)都有许多潜在的起点，提供详细的重构说明会让这本已经很长的书变得更冗长。

I.8 假设

在写这本书的时候，我假设读者熟悉一些对象技术(也称为“面向对象编程”)，对象方法似乎是自动单元测试变得普及的先决条件。这并不说明我们不能用过程语言或函数语言完成测试，但使用这些语言可能更具有挑战性(至少会很难)。

不同的人有不同的学习风格。有些人需要从“全局”抽象内容开始，逐步学习“刚好足够”的细节。其他人只需要理解细节，不需要“全局”了解。有些人通过听或阅读单词就能学得很好；有些人需要图片来帮助他们让概念形象化。还有些人通过阅读代码来学习编程概念。我尽可能提供小结、详细描述、代码样本和图片，尽量满足所有这些学习风格。对于不能从该学习风格中受益的读者而言，这些条目应该是可略过的部分[PLOPD3]。

I.9 术语

本书的术语来源于两个不同的领域：软件开发和软件测试。因此有些读者难免不熟悉某些术语。读者如果遇到自己不理解的术语，可以参考词汇表。然而，这里我要指出一到两个术语，因为熟悉这些术语是理解本书大多数资料的基本要求。

I.9.1 测试术语

软件开发人员可能发现自己对术语“被测系统”(本书中缩写为 SUT)不太熟悉。它是“要测试的所有东西”的简称。写单元测试时，SUT 就是要测试的类或方法；写客户测试时，SUT 可能是整个应用程序(或者至少是它的主要子系统)。

还要求我们构造的应用程序或系统没有包含在 SUT 中的部分运行测试，因为 SUT 会调用它，或者因为我们执行它时，它建立 SUT 会使用的必要数据。前面一种类型的元素称为依赖组件(DOC)，这两种类型都是测试夹具的一部分。如图 I-1 所示。

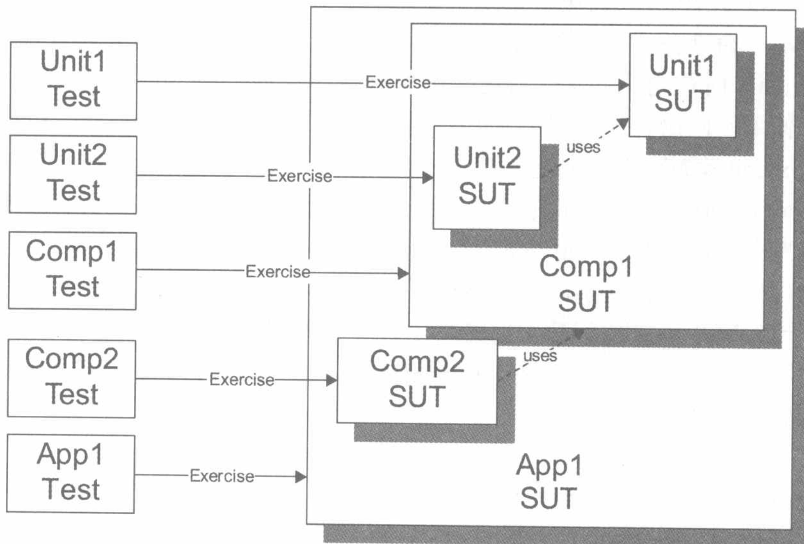


图 I-1　一组测试，每个测试都有自己的 SUT。应用程序、组件或单元只是与特定测试相关的 SUT。Unit1 SUT 担任 Unit2 Test 的 DOC(夹具的一部分)这一角色，并且是 Comp1 SUT 和 App1 SUT 的一部分

I.9.2 语言专用 xUnit 术语

虽然本书包括用各种语言和 xUnit 家族成员所写的示例，但 JUnit 表现更为突出。JUnit 是大多数人在某种程度上很熟悉的语言和 xUnit 架构。JUnit 到其他语言的许多转

试读结束：需要全本请在线购买： www.ertongbook.com