

Join the discussion @ p2p.wrox.com



Wrox Programmer to Programmer™



Professional Test Driven Development with C#

C#测试驱动开发

[美] James Bender 著
Jeff McWherter 译
贾洪峰 李菊彦 译

清华大学出版社

C# 测试驱动开发

(美) James Bender 著
Jeff McWherter
贾洪峰 李菊彦 译

清华大学出版社

北 京

James Bender , Jeff McWherter
Professional Test Driven Development with C#
EISBN : 978-0-470-64320-4
Copyright © 2011 by Wiley Publishing, Inc.
All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing , Inc. 授权清华大学出版社出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字 : 01-2011-3827

本书封面贴有 Wiley 公司防伪标签, 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

C#测试驱动开发 / (美) 本德(Bender, J.) , (美) 麦格瓦特(McWherter, J.) 著 ; 贾洪峰, 李菊彦 译.

—北京 : 清华大学出版社, 2012

书名原文 : Professional Test Driven Development with C#

ISBN 978-7-302-27971-6

. C... . 本... 麦... 贾... 李... . C 语言 - 程序设计 . TP312

中国版本图书馆 CIP 数据核字(2012)第 028902 号

责任编辑 : 王 军 于 平
装帧设计 : 牛艳敏
责任校对 : 成凤进
责任印制 :

出版发行 : 清华大学出版社

网 址 : <http://www.tup.com.cn> , <http://www.wqbook.com>
地 址 : 北京清华大学学研大厦 A 座 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 :

装 订 者 :

经 销 : 全国新华书店

开 本 : 185mm × 260mm 印 张 : 20 字 数 : 487 千字
版 次 : 2012 年 3 月第 1 版 印 次 : 2012 年 3 月第 1 次印刷
印 数 : 1 ~
定 价 : .00 元

产品编号 :

作者简介

James Bender 是 Improving Enterprises 公司的技术副总裁，从事软件开发和基础体系结构设计已有 17 年。作为开发人员和体系结构设计师，他参与过各种各样的软件设计，既有小型的单用户应用程序，也有企业规模的多用户系统。他是一位经验丰富的顾问和作家，擅长于 .NET 开发与体系结构设计、SOA、WCF、WF、云计算和敏捷开发方法。

James 致力于推动软件开发的封装，追求新的、更好的应用程序开发方式。在他的职业生涯之初，是用 C++ 在基于 SCO Unix 的系统上开发信用卡处理应用程序。在 20 世纪 90 年代后期，James 开始研究用基于 Java 的 JSP 页面和微软的 ASP 技术进行 Web 开发。他从 .NET 首次发布公共测试版就开始使用此工具。后来他继续研究 .NET 技术，重点是分布式计算范例(.NET Web 服务使之成为可能)，之后很自然地发展到对微软的 Windows Communication Foundation(WCF)极为着迷。

从 2003 年开始，James 就开始实际应用敏捷方法，包括 Scrum 和极限编程(eXtreme Programming, XP)。在研究敏捷方法的同时，James 开始探索测试驱动开发。他致力于向客户及软件开发社区的众多开发人员介绍在敏捷软件开发和测试驱动开发中用到的概念和技术。

James 是 Visual C# 方面的微软 MVP、开发社区的活跃会员和 Central Ohio .NET Developers Group(www.condg.org)的现任主席，并继续领导着 Columbus Architects Group(www.colarc.org)，还是 nplus1.org 第一方(first-party)内容的高级编辑，nplus1.org 是一个面向体系结构设计师的教育网站。James 的博客地址为 www.jamescbender.com。

Jeff McWherter 是 Gravity Works Design and Development 的合伙人和开发经理，这家公司的办公室位于密歇根州兰辛市的老城区(Old Town District)。Jeff 毕业于密歇根州大学，拥有 12 年以上的专业软件开发经验，他还在微软获得了很多证书，包括微软认证解决方案开发员(MCSD)、微软认证数据库管理员(MCDBA)、微软认证应用程序开发员(MCAD)和微软认证技术工程师(MCTS)。

2010 年，Jeff 连续第三年被评为“微软最有价值专家(MVP)”。同年，他荣获了由兰辛区域商会颁发的“Ten Over The Next Ten”奖项，这一奖项评选出会在接下来的 10 年间担负起重任的 10 位年轻专家。Jeff 还是一位作家，Wrox 出版社出版了他的 *Testing ASP.NET*

Web Applications 一书。

作为一名作家和软件开发人员，Jeff 还积极参与国内程序设计社区的开发，例如在会议上发言、组织 Lansing Give Camp 等活动，这些活动将开发人员和一些非盈利组织联合起来，共同参加一些志愿者项目。

技术编辑简介

Mitchel Sellers 擅长使用微软技术进行软件开发。他是 IowaComputerGurus 有限公司的 CEO、微软 C# MVP 和微软认证专家，出版了两本书，为很多书籍做技术编辑。经常会看到 Mitchel 与非常大型的软件开发社区进行互动，既包括活动/会议，也包括网上讨论论坛。如需有关 Mitchel 的专业经验、认证和出版作品方面的更多信息，请参阅他在 MitchelSellers.com 上的简历。

致 谢

我首先要感谢我的女朋友 Gayle (如果幸运的话,等您看到这本书时她已经是我的未婚妻了)。在编写本书期间,她非常支持和理解我,她的付出远远超出了应当付出的。

我要感谢我的父母,因为有他们才有了我,也才有了这本书。我的妈妈因为马上可以看到这本书而感到非常自豪。保佑她的心脏吧!我希望当她发现我在撒谎,这本书一点也不像 Stephen King 的小说时,还能如此自豪。

在这个充满理解和支持的部门里,我要感谢 Daniel Grey、Mark Kovacevich、Jeff Perry 和 Improving 公司的每一个成员。我还要感谢 Pete Klassen。我们想念你,兄弟!

我要感谢 Jeff McWherter 和 Michael Eaton 为本书所做的贡献。Jeff,感谢你为我分担一些工作;Mike,感谢你促使我将“非 Web”人员也包含在内。我还要感谢我的编辑 Sydney,是她让这本书看起来好像我具有丰富的写作经验。

Brian Prince,感谢你促使我加入开发社区。我本来想在这儿写点有趣的事情,但却都想不起来了。

当我有机会编写这本书时,我差点拒绝了。我要感谢 Ted Neward 说服我最终写了这本书。

我还要感谢我在 NPlus1.org Mike Wood 和 Chris Woodruff 的合作伙伴,在过去的几个月里,当我对于撰写本书有所懈怠时,他们会督促我。

要感谢的人还有很多: Brahma Ghosh、Brian Sherwin、Bill Sempf、Jeff Blankenburg、Carey Payette、Caleb Jenkins、Jennifer Marsman、Sarah & Kevin Dutkiewicz、Steve Harman、Josh Holmes。感谢 Matt Groves,他对本书的付出几乎和我一样多。我肯定漏了某个人,在这里表示歉意!

James

首先感谢我富有耐心的妻子 Carla。感谢你在我努力工作期间给予我的所有支持、耐心和理解。感谢我在 Gravity Works 的同事——Amelia Marschall、Lauren Colton、Scott Gowell 和 Dave Smith,感谢他们回答我随时碰到的各种问题。最后,还要感谢 James,感谢他的努力工作、奉献和友谊。

Jeff

前言

作为一名咨询师，我与开发人员合作过。在每一位客户那里，我都会遇到一个新团队，要了解他们是如何开发软件的。我见过非常优秀的团队，也看到过分崩离析以致从未有过一个成功项目的团队。这几年，我注意到不同的团队获得成功具有不同的特点。我开始总结，是什么使一个开发团队开发和部署的应用程序优质且能为企业提供价值。

大多数人都预计我的观察会得出这样一条结论：成功的团队拥有更聪明、更有能力的人员，他们当然能够成功了。但那些失败的团队中也有许多非常聪明的人。显然，智力不是成功的关键因素。

我在成功团队中所看到的，是他们对技术的热情，对自己作品的自豪感。他们总在学习新工具和新技术，希望能够提高软件开发速度，减少错误。而那些不太成功的团队则满足于坚持旧的工作方式，对周边发生的变化从来不感兴趣。

在我第一次遇到这些成功的、富有激情的开发团队时，他们并非都采用了测试驱动开发(test-driven development, TDD)。但在了解到这一概念之后，他们大多都会快速而迫切地锁定它。这些团队发现，在开发软件的过程中，增加测试驱动开发实践，可以马上得到非常出色的结果，提高了所交付应用程序的质量，减少了其中的缺陷。

要培养激情很难，扼杀它却很容易。在缺乏激情的团队里，通过介绍测试驱动开发，在许多情况下都能重新点起开发人员心中的激情。特别是对那些已经厌倦了日复一日地重复相同开发工作的开发人员而言，尤其如此！

除了激情之外，研究测试驱动开发还有另外一个非常有说服力的理由。有证据表明，近年来的两个最大变化可能会吸引极多的开发人员，这两大变化就是敏捷方法的兴起和测试驱动开发。这两者经常一起发展。我不相信，敏捷方法能够在不使用测试驱动开发的情况下获得长期成功，也想象不出测试驱动开发如何在一个瀑布环境中发挥作用。

敏捷方法已经比较成熟。它不再是一些小型开发部门使用的“疯狂牛仔式编码”方法了。一些大型公司在设计其 IT 部门的结构时大规模采用瀑布形式，现在也开始采用一种敏捷方法来开发越来越多的项目。即使是最官僚化的组织、政府也开始研究敏捷方法，并取得了巨大成功。这些开发清楚地表明了一个现实：那些可以在敏捷环境中工作(包括采用测

试驱动开发)的开发人员，其价值很快就能超过那些不能在此环境中工作的开发人员。

驱动测试开发并不是存在于真空中的。在过去几年里，许多团体和运动都旨在提高所开发软件的质量，并使企业参与到这一过程中。新的工作理念与方式已经向前发展，可以帮助开发人员开发出能够满足企业需求的可维护应用程序。诸如“软件工艺”和 SOLID 等术语已经进入了世界各地富有激情的开发人员的词汇表当中。一些开发人员甚至更进一步，将自己称为软件工匠或软件技工。

为了满足人们对于学习测试驱动开发及相关知识的要求，已经有了许多书籍、网站和研讨会，其中有很多是非常出色的。但也有一些只不过是將一些常见的、可移植的工作方式加以商业化运作，为其套上一层昂贵的、专用解决方案的外衣。许多聪明的、热情的开发人员都在讨论和传播测试驱动开发。但是，没有任何一种“一站式”资源能够将一位开发人员(具体来说，是.NET 开发人员)从一个新手变为一个……嗯，还是一个新手，不过是掌握了一些信息的新手。

您正在阅读本书，就凭这一点，就说明您对测试驱动开发是有兴趣的。您可能是一位开发人员，听了关于测试驱动开发的信息，但却从未真正有机会研究它。您也可能是一位很有经验的测试驱动开发人员，只是想看看这本书在讨论这一主题时与其他书有什么不同。无论是哪一种情况，就凭您正在阅读本书，就表明测试驱动开发已经成为主流，值得花时间来学习、练习和改进。

本书读者对象

测试驱动开发是一种非常有效的方式，从项目伊始就能保证应用程序的质量。测试驱动开发的相关原理与实务使您和您的团队能够快速编写出便于维护的软件，更好地满足企业的需要。如果您是一位希望提高自己技能的开发人员，那么本书就是为您准备的。

如果您刚刚接触测试驱动开发，那就从第 1 章开始学习。这样会使您了解足够的背景知识，理解测试驱动开发为什么会如此引人入胜。第 1 章还会介绍一些有关面向对象编程、SOLID 原则和重构的概念。为了实际运用测试驱动开发，这些技术都是至关重要的基础。

如果曾经研究过测试驱动开发，可以从第 3 章开始阅读，该章回顾了面向对象开发、SOLID 原则和重构。即使是非常有经验的开发人员，有时也需要复习一下这些概念与应用程序开发有什么关系。本书的其他部分(从第 4 章开始)为这些开发人员提供了测试驱动开发的形式和结构。

测试驱动开发经验非常丰富的开发人员可能希望从第 部分开始阅读。如果从这里开始，您应当已经拥有了很高程度的测试驱动开发、面向对象编程(OOP)和 SOLID 技术。这一部分主要讨论.NET 开发人员所面对的具体情景。其中包括如何在以下应用程序中采用测试驱动开发：基于 Web 的应用程序(包括 Web 窗体、ASP.NET MVC 和 JavaScript)、用“模型-视图-视图模型(MVVM)”模式在 WPF(Windows Presentation Foundation)上构建的应用程

序、用微软的 WCF(Windows Communication Foundation)构建的服务应用程序。一个应用程序中最难测试的部分就是边缘。这些章节将说明如何尽可能缩小应用程序的边缘，从而使其更容易测试。

本书内容

要使测试驱动开发在软件行业中得以繁荣兴盛，需要一些条件，本书从讨论这些条件开始。软件开发发展到今天，有其历史和特定的条件，理解这些很重要。避免重复过去的错误也很重要。在自己当前的开发实践中找出这些反面模式则更为重要。

为了支持测试驱动开发的应用，本书还全面介绍了面向对象编程、敏捷方法以及 SOLID 软件设计和编码原理。

当然，本书还介绍了测试驱动开发中一些固有的、必需的概念。首先介绍的一些测试都是很简单的，很容易理解。您会看到如何在 Visual Studio 中使用 NUnit 单元测试框架编写单元测试。

之后将介绍依赖注入模式。其中包括如何实现这一模式，以及依赖项注入框架(如 Ninject)如何帮助管理应用程序中的依赖。本书还会介绍“模拟”(mocking)与“模拟框架”(mocking framework)实践，包括对模拟框架 Moq 的介绍。

关于行为驱动开发的基础知识，也在介绍范围之内，但不会深入讨论这一主题。本书解释了行为驱动开发背后的思想，并突出介绍了命名测试的业务驱动开发风格。本书还介绍了 NBehave 测试框架。NBehave 有许多功能，但本书只用它为测试提供“句法糖”。

本书的组织结构

设计本书的内容结构花费了很大的力气，希望每一章都是以上一章的课程为基础。前几章主要提供一个基础，说明测试驱动开发的重要性以及有效使用它所需要的基础技巧。每个章节都以一个概念为基础，如依赖项注入(dependency injection)和模拟，直到使您掌握实际运用测试驱动开发所需要的全部工具和技巧为止。

第 III 部分结合前几章讲授的测试驱动开发技巧，阐述了如何利用微软的几个框架来练习测试驱动开发，这几个框架是用来为应用程序开发界面的，包括 ASP.NET MVC、WPF 和 WCF。

本书的最后是附录，列出了一些备用工具，有助于使用测试驱动开发来开发应用程序。它还列出了一些可能出现的用户情景，如果在日常工作中还没有使用测试驱动开发，可以把这些用户情景当作练习。

完成本书练习的必备条件

要完成本书给出的示例，并使用可下载的演示应用程序，需要以下工具：

- Visual Studio 2010(任意版本)
- NUnit version 2.5.2.9222 或更新版本，可从 nunit.org 获取
- Moq version 4 beta 4(build 4.0.10827.0)或更新版本，可从 code.google.com/p/moq 获取
- Ninject version 2(build 2.1.0.91)或更新版本，可从 ninject.org 获取
- NBehave version 0.4.5.183 或更新版本，可从 nbehave.org 获取
- Fluent NHibernate version 1.1 或更新版本，可从 fluentnhibernate.org 获取
- 数据库管理系统(Database Management System, DBMS)供示例应用程序使用。本书中的示例使用了 Microsoft SQL Server Developer，但任何关系数据库系统都是可以的。

源代码

使用本书的示例时，既可以采用人工方式输入所有代码，也可以使用随本书提供的源代码文件。本书中用到的所有源代码都可以从 www.wrox.com 和 <http://tupwk.com.cn/downpage> 下载。在该网站上，只要找到本书的标题(使用搜索框或标题列表)，并单击本书详细信息页中的 Download Code 链接，就可以获得本书的全部源代码。该网站提供的代码在本书中用以下图标突出显示：



列表中包含了标题中的文件名。如果只是一个代码段，可以在代码提示中找到其文件名，如下所示：

代码段文件名



提示：

由于许多书的书名类似，所以用 ISBN 进行搜索是最容易的；本书的 ISBN 为 978-0-470- 64320-4。

下载代码之后，用最喜欢的压缩工具解压即可。或者，可以进入 Wrox 的代码下载主页 www.wrox.com/dynamic/books/download.aspx，查看本书及所有其他 Wrox 书籍的可用代码。

勘误表

尽管我们已经尽力保证正文或代码中不出现错误，但是错误总是难免的。如果您在本书中找到了错误，如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，还有助于提供更高质量的信息。

请给 wkservice@vip.163.com 发电子邮件，我们就会检查您的反馈信息，如果是正确的，我们将在本书的后续版本中采用。

要在网站上找到本书英文版的勘误表，可以登录 <http://www.wrox.com>，通过 Search 工具或书名列表查找本书，然后在本书的详细信息页面单击 Book Errata 链接。在打开的页面可以查看到 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表链接，网址是 www.wrox.com/misc-pages/booklist.shtml。

P2P.WROX.COM

要与作者和同行讨论，请加入 p2p.wrox.com 上的 P2P 论坛。这个论坛是一个基于 Web 的系统，便于您张贴与 Wrox 图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当论坛上有新的消息时，可以给您传送感兴趣的论题。Wrox 作者、编辑、其他业界专家和读者都会到这个论坛上来探讨问题。

在 <http://p2p.wrox.com> 上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发您自己的应用程序。加入论坛的步骤如下：

- (1) 进入 p2p.wrox.com，单击 Register 链接。
- (2) 阅读使用协议，并单击 Agree 按钮。
- (3) 填写加入该论坛所需要的信息和自己希望提供的其他信息，单击 Submit 按钮。
- (4) 您会收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。



提示：

不加入 P2P 也可以阅读论坛上的消息，但要发布自己的消息，必须加入该论坛。

加入论坛后，就可以发表新消息，响应其他用户发表的消息。可以随时在 Web 上阅读消息。如果要让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

关于使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作情况以及 P2P 和 Wrox 图书的许多常见问题。要阅读 FAQ，可以在任意 P2P 页面上单击 FAQ 链接。

目 录

第 部分 入 门

第 1 章 通向测试驱动开发之路	3
1.1 软件开发的经典方法	4
1.1.1 软件工程简史	4
1.1.2 从瀑布到迭代和递增	5
1.2 敏捷方法简介	6
1.2.1 敏捷方法简史	6
1.2.2 TDD 的原理与实践	7
1.3 TDD 背后的概念	8
1.3.1 作为设计方法的 TDD	8
1.3.2 作为开发实践的 TDD	8
1.4 TDD 的优点	9
1.5 TDD 方法的简单示例	10
1.6 本章小结	17
第 2 章 单元测试简介	19
2.1 什么是单元测试	20
2.1.1 单元测试的定义	20
2.1.2 什么不是单元测试	20
2.1.3 其他类型的测试	22
2.2 NUnit 一览	24
2.2.1 什么是单元测试框架	24
2.2.2 NUnit 基础知识	24
2.3 与模拟对象分离	28
2.3.1 模拟为什么重要	28
2.3.2 虚拟、伪对象、存根 和模拟	29
2.3.3 最佳实践与最差实践	35
2.4 Moq 概览	35
2.4.1 模拟框架做些什么	36
2.4.2 关于 Moq	36
2.4.3 Moq 基础知识	36
2.5 本章小结	40

第 3 章 重构速览	41
3.1 为何重构	42
3.1.1 项目的生命周期	42
3.1.2 可维护性	43
3.1.3 代码度量	43
3.2 整洁代码原则	45
3.2.1 OOP 原则	45
3.2.2 SOLID 原则	49
3.3 代码异味	52
3.3.1 什么是代码异味	52
3.3.2 重复代码和相似类	52
3.3.3 大型类和大型方法	53
3.3.4 注释	55
3.3.5 不当命名	55
3.3.6 特征依赖	56
3.3.7 If/Switch 过多	57
3.3.8 Try/Catch 过多	58
3.4 典型重构	59
3.4.1 析取类或接口	60
3.4.2 析取方法	61
3.4.3 重命名变量、字段、方法 和类	66
3.4.4 封装字段	66
3.4.5 用多态替换条件	67
3.4.6 允许类型推断	70
3.5 本章小结	71
第 4 章 测试驱动开发：以测试为 指南	73
4.1 从测试开始	74
4.2 红灯、绿灯、重构	76
4.2.1 TDD 的 3 个阶段	76
4.2.2 “红灯”阶段	77
4.2.3 “绿灯”阶段	77

4.2.4 “重构”阶段	78	6.4 创建项目	126
4.2.5 重新开始	78	6.4.1 选择框架	126
4.3 重构示例	79	6.4.2 定义项目结构	128
4.3.1 第一项功能	79	6.5 本章小结	132
4.3.2 通过第一个测试	82	第 7 章 实现第一个用户情景	133
4.3.3 第二项功能	83	7.1 第一个测试	134
4.3.4 重构单元测试	85	7.1.1 选择第一个测试	134
4.3.5 第三项功能	86	7.1.2 为测试命名	135
4.3.6 重构业务代码	88	7.1.3 编写测试	136
4.3.7 纠正重构缺陷	90	7.2 实现功能	144
4.3.8 第四项功能	92	7.2.1 编写能够正常工作的最简单 代码	144
4.4 本章小结	94	7.2.2 运行可以通过的测试	153
第 5 章 模拟外部资源	95	7.2.3 编写下一个测试	153
5.1 依赖项注入模式	96	7.3 通过重构来改进代码	160
5.2 抽象数据访问层	106	7.4 多角度测试	161
5.2.1 将对数据库的关注移出业务 代码	106	7.5 本章小结	161
5.2.2 将数据与存储库模式 隔离	106	第 8 章 集成测试	163
5.2.3 注入存储库	107	8.1 早集成、常集成	164
5.2.4 模拟存储库	110	8.2 编写集成测试	165
5.3 本章小结	111	8.2.1 如何管理数据库	165
第 部分 将基础知识变为行动		8.2.2 如何编写集成测试	166
第 6 章 启动示例应用程序	115	8.2.3 端对端集成测试	185
6.1 定义项目	116	8.2.4 使各类测试保持分离	185
6.1.1 开发项目综述	116	8.3 运行集成测试的时机和 方式	185
6.1.2 定义目标环境	117	8.4 本章小结	186
6.1.3 选择应用程序技术	118	第 部分 TDD 方案	
6.2 定义用户情景	118	第 9 章 Web 上的 TDD	191
6.2.1 收集情景	118	9.1 ASP.NET Web 窗体	192
6.2.2 确定待办事项表	120	9.2 使用 ASP.NET MVC	204
6.3 敏捷开发过程	121	9.2.1 MVC 101	205
6.3.1 估计	121	9.2.2 Microsoft ASP.NET MVC 3.0	205
6.3.2 迭代工作	122	9.2.3 使用 MVC Contrib 项目	214
6.3.3 团队内部交流	124	9.2.4 ASP.NET MVC 汇总	214
6.3.4 零次迭代：第一次迭代	124	9.3 使用 JavaScript	214
6.3.5 零次迭代中的测试	124		
6.3.6 结束迭代	125		

9.4 本章小结	220	13.2 单元测试框架	274
第 10 章 测试 WCF 服务	221	13.2.1 MSTest	274
10.1 应用程序中的 WCF 服务	222	13.2.2 MbUnit	275
10.2 测试 WCF 服务	222	13.2.3 xUnit	276
10.2.1 为实现可测试性进行 重构	223	13.3 模拟框架	277
10.2.2 向服务引入依赖项注入	225	13.3.1 Rhino Mocks	277
10.2.3 编写测试	230	13.3.2 Type Mock	279
10.2.4 实现依赖项的存根	233	13.4 依赖项注入框架	281
10.2.5 验证结果	237	13.4.1 Structure Map	281
10.2.6 要留意的问题多发 区域	237	13.4.2 Unity	283
10.3 本章小结	238	13.4.3 Windsor	284
第 11 章 测试 WPF 和 Silverlight 应用 程序	239	13.4.4 Autofac	285
11.1 测试用户界面时的问题	240	13.5 其他有用工具	286
11.1.1 MVVM 模式	240	13.5.1 nCover	287
11.1.2 MVVM 如何使 WPF/ Silverlight 应用程序 可测试	243	13.5.2 PEX	287
11.1.3 将所有内容结合 在一起	255	13.6 如何向团队介绍 TDD	288
11.2 本章小结	258	13.6.1 在拒绝改变的环境中 工作	289
		13.6.2 在接受改变的环境中 工作	289
第 部分 需求和工具		13.7 本章小结	289
第 12 章 应对缺陷和新的需求	261	第 14 章 结论	291
12.1 处理修改	262	14.1 已经学到的内容	292
12.1.1 修改的发生	262	14.1.1 你是自己代码的客户	292
12.1.2 从测试开始	264	14.1.2 逐步找出解决方案	292
12.1.3 修改代码	266	14.1.3 用调试器作为手术 器械	293
12.1.4 使测试保持通过状态	269	14.2 TDD 最佳实践	293
12.2 本章小结	270	14.2.1 使用有意义的名字	293
第 13 章 有关优秀工具的争论	271	14.2.2 为一个功能单元至少编写 一个测试	294
13.1 测试运行程序	271	14.2.3 保持模拟的简单性	294
13.1.1 TestDriven.NET	272	14.3 TDD 的好处	294
13.1.2 Developer Express 测试 运行程序	272	14.4 如何向团队介绍 TDD	295
13.1.3 Gallio	273	14.5 本章小结	296
		附录 A TDD Katas	299

第 部分

入 门

- 第 1 章 通向测试驱动开发之路
- 第 2 章 单元测试简介
- 第 3 章 重构速览
- 第 4 章 测试驱动开发：以测试为指南
- 第 5 章 模拟外部资源



第 1 章

通向测试驱动开发之路

本章内容

- 软件开发如何发展到 TDD(Test-Driven Development，测试驱动开发)
- 敏捷方法是什么，它与基于瀑布的传统技术有何区别
- TDD 是什么，使用它有什么好处

测试驱动开发(TDD)已经成为现代软件开发中最重要的概念和实践之一。要理解其中的原因，需要考虑软件开发实践的历史。TDD 是通过一个演化过程逐渐形成的。它的出现主要是应对编写软件时所面临的困难与挑战，并不存在什么真实计划。这是一种非常经典的案例：一件事物的特点使其更为成功、更为强大，得以推广，而那些导致失败的特点则被丢弃。TDD 实践不是由哪一家公司或哪一个人创建的；它们是在进行无数次讨论(更准确地说，是无数次争论)之后产生的，这些争论的内容包括：过去做了什么、这些做法为什么会失败、如何做才会更好！如果 TDD 是一个结构，如一座房屋，那它的基础就是建立在失败之上的。失败的项目就是 TDD 的构建基础，因为它们的开发人员知道必须要找到一种更好的方法。

本章将介绍软件开发的历史，以及软件项目的管理方法如何从顺流直下的瀑布方式转到迭代方式，再转到敏捷方法。还会学习 TDD 实践如何成为敏捷方法的关键组成部分，以确保能够生成满足业务需要的高质量代码。本章会解释 TDD 的原则，概括其优点，并给出一个例子说明如何实现 TDD。

1.1 软件开发的经典方法

要理解 TDD 的重要性，有必要看一看它的发展过程。在过去 50 年里，软件开发实践一直致力于寻求一种平衡，也就是在业务需要、当前技术的能力和最能提高开发人员工作效率的方法之间达到一种平衡。在这条道路上曾经摔过不少跟头，但这些非常重要，因为可以用来判断哪些技术和方法最终会走进死胡同。本章就回顾一下 TDD 的发展过程。

1.1.1 软件工程简史

针对业务进行软件开发，始于大型机时期。每个软件供应商似乎都有自己独特的平台和范例，用于进行软件开发。有时这些系统非常相似，开发人员可以非常顺畅地在任务之间和平台之间进行转移。但有时，要转换任务或者转移平台，可能就需要从头做起。尽管计算的基本概念相同，但每个供应商对这些概念都有自己的理解，有时这种理解非常与众不同。当时所用的语言也非常原始，一些在今天被认为是理所当然的最简单工作，可能需要许多行代码才能完成。而且在许多时候，有些东西在一种语言或平台的一个实现中有效，换到另一个实现中可能就无法以同样的方式工作了。

大型机是一种非常昂贵的大型设备。许多公司是没有大型机的，所以就出现了“服务社”(service bureau)的概念：拥有大型机的公司可以在一定时间内向客户出租其计算机。遗憾的是，有时需要等待一段时间才能使用计算机。设想一下，今天编写了一段程序，但要等到下周一才能编译它。在这种限制下，开发人员是很难提高工作效率。假定打算在星期一进行编译，但遇到了一个错误。可以纠正这个错误，但是至少在 3 天之内是无法知道修正是否正确。由于计算资源的使用受限，所以与交付产品相比，一些不必要的测试会退居末席。

在“瀑布”开发概念出现之前也经历了一段时间。自己拥有设备的开发人员经常采用迭代方式工作：仔细研究系统的一些特定部分，完善它们，然后再添加新的特性和功能。这种方法是很有用的，因为它允许开发人员以一种符合逻辑的方式开发应用程序，这种逻辑方式使开发人员能够理解和管理开发中的事务。但是，业务用户以及什么符合逻辑的且能被用户理解的内容，并未考虑在内。

第二代微型计算机出现于 1977 年，但在 1979 年发布 VisiCalc 之前，它们并没有真正在企业中应用。VisiCalc 是第一个可供个人计算机使用的电子表格应用程序。它表明：个人计算机不只是家庭玩具，而是可以向企业提供实际价值的机器。与大型机相比，个人计算机有许多优点，第一个优点就是它要便宜得多。一家公司可能连一台专用大型机也买不起，但能够买得起几十台个人计算机。尽管个人计算机的运算速度比不上大型机，但它们的实用性使其非常适于处理那些不需要大型机功能就能完成的日常任务。开发人员可以为个人计算机编写应用程序，并能马上知道这些代码是否正确。他们不需要再等待好几天才能排定任务计划和运行任务了。

到第三代和第四代编程语言出现时，就更好了。这些语言把其原来编程语言中的一些常见任务进行抽象，开发人员只需要专注于手边的业务问题，从而进一步提高了生产效率。有了这些语言，那些不希望处理语言底层难题(如 Assembler 和 C)的人也能从事软件开发，从而扩大了用户群。企业和企业计算机行业最终选择了一些基础语言及其派生变体。由于开发人员的技能具有更强的可移植性，从而提高了他们对企业的吸引力，能够生产出更畅销的软件。

最后，企业对规划的需求产生了瀑布项目方法。瀑布背后的概念是：每个软件项目(其平均时间跨度大约两年)从一开始，就应当对于从启动到交付的每个阶段进行规划。这就需要在开始时花费很长时间收集需求信息。在收集到所有需求信息之后，就将它们“扔过墙去”，抛给体系结构设计师。体系结构设计师拿到这些需求信息之后，开始设计系统，将其最细小的细节也构建出来。他们完成自己的任务之后，把设计方案丢给开发人员，由他们来开发系统。开发人员完成自己的工作之后，把系统丢给“质量保证(QA)”部门，由他们对应用程序进行测试。只要应用程序通过验证，就向用户进行部署。

在瀑布方法中，软件测试通常需要很长时间、难度很大、效率很低、成本很高。QA 测试人员以人工方式运行各种测试脚本来对应用程序进行测试，这些脚本是一些文档，用来指示测试人员在系统中执行某一操作，并给出应当观察到的结果。有时，这些脚本会长达数百页。如果对系统进行了一处修改，可能需要测试人员花费两到三周的时间来对系统进行完整的回归测试。另外还有一个问题：这些测试脚本通常是由创建系统的开发人员编写的。在这些情况下，脚本中通常描述的是系统“将会”如何运作，而不是“应当”如何运作。

走向 TDD 的第一步是自动化 QA 测试工具的发展。这些工具记录了用户在一个用户界面(UI)上所执行的一系列操作，并允许用户在以后回放这些操作。这样可以验证用户界面是能够正常工作的。在记录了初始测试之后，QA 工具执行回归测试的速度要远远快于人工测试，而且能够重复运行。许多此类早期工具都有一个很大的缺点，那就是它们创建的测试都非常脆弱。如果用户界面的某个方面发生了变化，测试通常不能应对这些变化，从而可能中断。对于那些采用“记录/回放”模型的工具，这种情况意味着只能抛弃该测试，再生成一个新测试。这些工具的后期版本允许编写一些脚本，从而能够比较轻松地吸收这样一些更改，但这些测试仍然很脆弱。

1.1.2 从瀑布到迭代和递增

软件开发并不是凭空进行的。无论是在一个长达 18 个月的项目中创建一个应用程序，校正企业测试过程规范(Testing Procedure Specification, TPS)报告，还是为您孩子的儿童曲棍球队创建一个网站，这都不重要；重要的是你正在应用一种方法。你知道了需求，要对功能进行规划，而且还开发了应用程序。在进行测试之后，就向用户们发布该应用程序。

瀑布方法有一个问题，就是所有需求信息都是在早期收集的。而在企业中，需求通常会因为各种原因发生变化。可能是法律的修改、公司策略方向的转移，甚至是一些非常简

单的事情(如需求收集阶段的一个错误),都会对后期过程产生严重影响。虽然瀑布方法在本质上是经过策划的,但它并不能很好地对变化做出响应。在对系统的需求做出一处修改之后,系统的其他部分通常也必须再经历一遍相同的需求/设计/开发/质保过程。这样会导致连锁效应,从而使计划的其余部分无法保持准确。

要制订早期计划,必须提前对这一工作进行估计——有时,要比实际工作提前数年,而且早期计划通常不是由实际完成工作的人制订的。这就相当于用纸牌搭建了一座房子,一次错误估计可能会对项目计划的其余部分造成一场浩劫。

体系结构设计师也不是无辜的。这个时代造就了一批生活在象牙塔中的体系结构设计师,他们给出的应用程序设计在实践中可能难以实现,在某些情况下甚至根本无法实现。开发人员对这种情况也无能为力,因为他们中的大多数人只是执行体系结构设计师的设计构想,而不考虑这种设计有没有意义。很多时候,最终交付给企业的东西(距离最初提出需求,已经有两年时间了)与所希望或需求的东西完全是风马牛不相及的。

为了克服瀑布方法中的一些问题,一些开发工作室转向“迭代”或“递增”开发的概念。这一思想是在获得一个大型瀑布项目时,将它分为几个较小的瀑布项目。每个子项目拥有一个确定的范围和交付目标,在完成后,将进入较大项目的下一次迭代。这是一种进步,因为这些较小项目更容易确定范围,大大加快向用户交付软件的速度。但在最后,它实际上就是几个链接在一起的瀑布项目,只不过是较短小而已。各个子项目仍然没有一种很好的机制来应对业务与技术的持续变化。还需要再进一步。

1.2 敏捷方法简介

瀑布方法试图对软件开发的实体进行控制与约束,敏捷方法则不同,它是包容这些实体。业务的变化是不可避免的,所以软件开发方法也必须能够适应。提前完成的大型计划有一个关键性的不足之处,就是根据准确定义所做出的评估总是有误差的。如果它们完全准确,那就不需要估计了,就是“确切数字”了。迭代过程的前途看好,但这些迭代本身和整体的方法都必须是灵活的,面对变化是开放性的。

1.2.1 敏捷方法简史

2001 年 2 月,Scrum、极限编程(Extreme Programming, XP)、实用编程(Pragmatic Programming)、功能驱动开发(Feature Driven Development)和其他一些新技术的倡议者聚在一起,起草了“敏捷宣言”。内容如下:

“通过亲自及协助他人进行软件开发,我们正致力于发掘更出色的软件开发方法。通过这样的努力,我们已建立以下价值观:

- 个人与互动 重于 流程与工具
- 可用的软件 重于 详尽的文件

- 与客户合作 重于 合约协商
- 响应变化 重于 遵循计划

也就是说，虽然右侧项目有其价值，但我们更重视左侧的项目。”

“敏捷宣言”本身并不是一种开发方法。它没有规定应当如何开发软件。它只是列出了一组关键价值，可用于创建和描述更轻型、更快速的应用程序开发方法，这些方法更多地关注人员、工作软件和结果，而不是将重点放在需要多年时间才能完成的过度重视细节的项目计划和堆积如山的文档。

今天，人们正在使用许多著名的敏捷开发方法：

- Scrum
- 极限编程(XP)
- 功能驱动开发
- Clear Case
- 自适应软件开发(Adaptive Software Development)

1.2.2 TDD 的原理与实践

这些方法的实现方式完全不同，但它们有一些共同的特征：

- 它们都将团队内部的交流放在优先地位。鼓励开发人员、业务用户和测试人员经常交流。
- 它们注重项目的透明性。开发团队不是生存在一个黑盒中，不能向团队的其余人员隐藏自己的行为。它们使用非常公开的工具(看板、大型可视图表等)来保证团队成员能够获取足够的信息。
- 团队成员都是相互负责的。团队不会因为某一个人而成功或失败；他们的成功与失败是按整个团队来衡量的。
- 开发人员个体没有自己的代码基。整个团队拥有完整的代码基，每个人都对其质量负责。
- 工作是在短暂的迭代开发周期中完成的，理想情况下，在每个周期结束时会发布。
- 应对变化的能力是这种方法的基础。
- 一个系统的大致框架是提前定义的，但详细设计要等到实际安排功能开发计划时才会进行。

敏捷方法并不是万灵药。它们也与编程中的混乱(chao)或“牛仔编程”无关。事实上，敏捷方法需要更多层次的训练才能正确实行。另外，真正的敏捷方法并不存在。最终，每个团队需要采用最适合自己的方法。这可能意味着，在开始时采用一种著名的敏捷方法，然后对其进行修改，或者将几种方法结合在一起。应当不断评估这些方法，多采用有效的，少采用无效的。

1.3 TDD 背后的概念

TDD 的历史始于 1999 年，当时有一批开发人员拥护一组被称为“极限编程(XP)”的概念。极限编程是一种敏捷式方法，其基础是找出软件开发中的哪些实践是有效的，专门派一组开发人员投入时间和精力来执行这些实践，其哲学就是“好事不怕多”！极限编程的一个关键是“测试先行的编程”。TDD 是从极限编程发展而来的，因为有些开发人员发现极限编程还不能包容当时一些更激进的概念，但发现通过实施 TDD 来提高产品质量的承诺是很有吸引力的。

如前所述，敏捷方法中不包括大规模的事先设计。业务需求被提炼为系统的功能。每项功能的详细设计是在为该功能排定进度时完成的。这些功能，以及相关的库和代码，都很简短。

1.3.1 作为设计方法的 TDD

当用作一种应用程序设计方法时，如果能够鼓励业务用户参与进来，帮助开发人员定义所生成的逻辑，有时甚至更进一步，定义一组输入及其期望输出，那么 TDD 就可以发挥最佳作用。这种做法非常必要，可以确保开发人员理解自己所开发功能背后的业务需求。TDD 确保最终产品与业务需求一致。它还有助于确保坚持了功能的作用域，可以帮助开发人员了解：对于当前开发的功能而言，哪些已完成工作是真正有意义的。

1.3.2 作为开发实践的 TDD

作为一种开发实践，TDD 并不简单。在过去进行开发时，可能是坐下来首先创建窗口、网页甚至类；而在 TDD 中，可能是首先编写测试。TDD 被称为“测试先行的开发”，乍看显得有点笨拙。但是，在首先编写测试时，实际上是在为将要编写的代码提出设计需求。在与业务用户一起定义这些测试时，实际上创建了一个由测试组成的需求的可执行版本。在通过这些测试之前，所编写的代码是不能满足业务需求的。

在编写第一个测试时，应用程序未能通过该测试的第一个标志就是它没有编译成功。这是因为这个测试试图实例化一个还没有定义的类，或者它希望使用一个不存在对象的方法。第一步就是创建要测试的类，为这个类定义希望测试的类的方法(method)。这时，测试仍然会失败，因为刚刚创建的类和方法没有做任何事情。下一步是编写刚好足以通过测试的代码。应当是能够通过测试的最简单代码。目的不是根据需求中可能出现的内容来编写代码。除非是需求发生了变化，或者由于添加一个测试来暴露出某项功能的缺失，否则不会编写这样的代码。这样就可以防止开发人员编写一些过于复杂的代码，来完成一些只需要简单算法就能解决的任务。要记住，TDD 的目的之一就是生成易于理解和维护的代码。

在通过第一个测试之后，添加更多测试了。应当尽力添加足够多的测试，以确保满足对被测功能的所有需求。作为这个过程的一部分，希望确保针对多种输入组合来测试方法。

这些输入包括超出许可范围的取值。这类测试称为“负面测试”。如果需求表明计算方法应当处理的比例最多仅为 20%，那就看看在用 21%调用此方法时会出现什么情况。通常这种做法会引发某种异常。如果方法的参数应当为字符串，那向它传递一个空字符串会发生什么？如果传递 null 又会如何呢？尽管保持这些测试的合理性很重要，但从不同角度进行测试以确保代码的可靠性，也是非常重要的。在测试了所有需求，而且全部通过测试之后，任务也就完成了。

1.4 TDD 的优点

在向从来没有使用过 TDD 的开发人员、开发经理和项目经理描述 TDD 时，他们常会持怀疑态度。从理论上来说，编写这种代码看起来确实冗长而且复杂。但其中的好处也是不可忽视的：

- TDD 从一开始就保证了代码的质量。鼓励开发人员仅编写能通过测试从而满足需求的代码。一个方法的代码越少，从逻辑上来说，其中包含错误的几率就越小。
- 无论是有意设计的还是巧合，大多数 TDD 开发人员编写的代码都遵循 SOLID 原则。SOLID 原则是一组帮助开发人员确保编写出高质量软件的编程实践。由 TDD 实践生成的这些测试是极为宝贵的，而那些作为副作用得到的高质量也是 TDD 难以置信的重要好处。SOLID 原则将在第 3 章介绍。
- TDD 确保了代码与业务需求之间的高度一致性。如果需求是以测试方式给出，而且通过了所有测试，就可以很自信地说代码满足了业务需要。
- TDD 鼓励创建更简单、针对性更强的库和 API。TDD 对开发过程的改变很大，这是因为那些为库或 API 编写接口的开发人员就是这个接口的第一用户。这就提供了有关如何编写该接口的新视角，而且马上就能知道这个接口是否有意义。
- TDD 鼓励与企业沟通。要创建这些测试，需要多与业务用户交流。这样，就可以确保输入与输出的组合有意义，还可以帮助用户理解所开发的产品。
- TDD 有助于从系统中清除那些没有用到的代码。大多数开发人员在编写应用程序时，设计界面和编写方法都是以可能发现的情况为基础的。这样就会导致系统中存在大量永远不会用到的代码或功能。这种代码的成本非常高，编写它们需要花费精力，即使这些代码什么也不做，也依然必须对其进行维护。它还使事情变得混乱，分散开发人员本应专注于重要工作代码的注意力。TDD 有助于从系统中清除这种寄生代码。
- TDD 提供了内置的回归测试。在对系统和代码进行修改时，总会创建一套测试，用来确保将来的更改不会损害现在的功能。
- TDD 终止了递归错误的出现。可能遇到过这种情景：正在开发一个系统时，相同的错误总是一次又一次重复出现。您可能觉得终于抓住并终止了这个错误，但在两个星期之后发现它又回来了。而利用 TDD 方法，只要报告了一个缺陷，就会编

写一个新的测试来揭露它。如果通过了这个测试而且一直能通过，说明已经真正消除了这一缺陷。

- 如果开发应用程序时能够时刻记着可测试性，所得到的结果就是一种开放的、可扩展的、灵活的体系结构。对于 TDD 和松散耦合体系结构，依赖注入(将在第 5 章介绍)都是关键组件。这样所得到的系统，因为其体系结构而变得可靠、易于修改，并能对抗缺陷。

1.5 TDD 方法的简单示例

下面的练习将展示一个示例，说明如何用 TDD 为系统添加功能。对于这个示例，假设需要实现一种功能，计算一个字符在一个字符串中的出现次数。假定已经有了一个解决方案，其中已经有了项目结构，但还没有实现这一方法的类。对于这个例子，还假定已经在项目中引用了单元测试框架。不要着急，第 6 章将会介绍如何进行引用。现在，这个解决方案的外观如图 1-1 所示。

ChapterOne.UnitTests 项目中将包含单元测试。最后完成的类将会位于 ChapterOneExample.Utilities 项目。首先，在单元测试项目中创建一个类，它将包含单元测试，如图 1-2 所示。

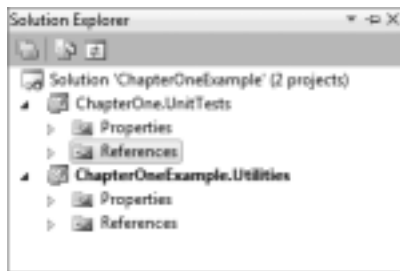


图 1-1

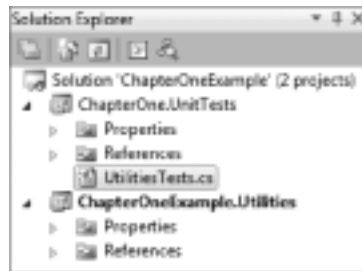


图 1-2

有多种不同方法可以在单元测试项目中安排单元测试类。有些开发人员喜欢将每个测试类放在一个独立的代码文件中。而另外一些开发人员则喜欢为一个特定功能的所有测试类创建一个代码文件。更常见的也是本书采用的做法，是为应用程序特定部分的所有单元测试类创建一个代码文件类——在本例中，这个特定部分就是实用程序项目。如果有一个业务逻辑库，其中包含几个基于业务/域的服务，那可以为每个域服务的测试类创建一个独立的代码文件。对于该例，这种做法有些大材小用了，所以只需要为整个项目使用一个测试类。

在创建 UtilitiesTest.cs 类时，Visual Studio 创建一些样板代码：

```
namespace ChapterOne.UnitTests
{
    public class UtilitiesTests
    {
```

```
    }
}
```



提示：

对于本例来说，UtilitiesTests 这个名称是可以的，但在真正的业务开发场景中，它的描述性不足以让团队中的其他开发人员了解它的目的。对于不了解技术的业务用户来说，更是没意义。本书后面的示例将采用一种特定的方法来为测试命名并设计其结构，使其更符合业务驱动开发风格。采用这一风格给出的测试名称便于人们理解，从而使非技术人员更容易理解实际测试。

现在编写第一个测试。这可能是对应于你的需求的最简单的表达式。该测试给出字符串 mysterious，并要求库计算字母 y 出现的次数。

```
using NUnit.Framework;

namespace ChapterOne.UnitTests
{
    public class UtilitiesTests
    {
        [Test]
        public void ShouldFindOneYInMysterious()
        {
            var stringToCheck = "mysterious";
            var stringToFind = "y";
            var expectedResult = 1;
            var classUnderTest = new StringUtilities();

            var actualResult =
                classUnderTest.CountOccurences(stringToCheck, stringToFind);

            Assert.AreEqual(expectedResult, actualResult);
        }
    }
}
```

此处应该翻译成特性

测试方法 ShouldFindOneYInMysterious 有一个 Test 属性，向单元测试框架表明这是一个测试。测试条件是通过定义要搜索的字符串和要在其中查找的字符设置的。它们还明确了预期结果。接着，在测试中调用该方法，并捕获实际结果。最后，由一个 Assert 语句来判断预期值与实际值是否相同。

第一个表明未通过此测试的标志就是：该应用程序未能成功编译。这表明还没有在应用程序中实现 StringUtilities 类。这是首先要完成的工作。为此，只需要向实用程序项目中添加一个名为 StringUtilities 的新类即可。Visual Studio 创建的这个类如下：

```
namespace ChapterOneExample.Utilities
{
}
```

```

    public class StringUtilities
    {
    }
}

```

该测试仍然不能通过，因为还没有为该类创建 CountOccurrences 方法。要想通过这个测试，下一步就是添加这个方法：

```

using System;

namespace ChapterOneExample.Utilities
{
    public class StringUtilities
    {
        public int CountOccurrences(string stringToCheck,
                                    string stringToFind)
        {
            throw new NotImplementedException ();
        }
    }
}

```

该方法在开始时抛出异常，因为到目前为止，该测试未能通过的唯一原因就是编译失败。这看起来显得有些傻，但在 TDD 中，不能靠想当然来进行任何假设；在编写方法之前，看到测试失败是非常重要的。这样可以确保仅编写恰好使测试得以通过的代码。在运行此测试时它会失败，如图 1-3 所示。



图 1-3

该测试失败的原因(如图 1-3 中突出文本所示)是还没有实现这个方法。下一个步骤就是编写代码，使测试通过：

```

using System;

namespace ChapterOneExample.Utilities
{

```

```

public class StringUtilities
{
    public int CountOccurrences(string stringToCheck,
                               string stringToFind)
    {
        var stringAsCharArray = stringToCheck.ToCharArray();
        var stringToCheckForAsChar = stringToFind.ToCharArray()[0];
        var occurrenceCount = 0;
        for (var characterIndex = 0;
            characterIndex < stringAsCharArray.GetUpperBound(0);
            characterIndex++)
        {
            if (stringAsCharArray[characterIndex] ==
                stringToCheckForAsChar)
            {
                occurrenceCount++;
            }
        }

        return occurrenceCount;
    }
}

```

这可能是实现该方法的最佳方式，也可能不是，但如果运行此测试，可以从图 1-4 中看到，它已经足以满足需求了。

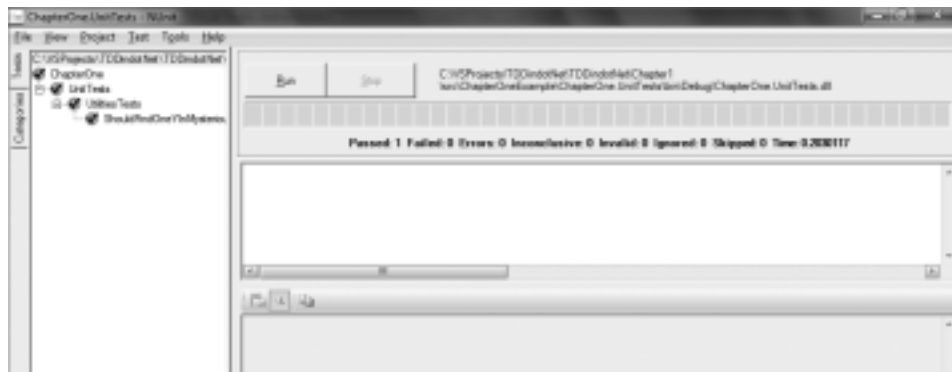


图 1-4

现在，可以看到在目标单词中找到了该字符的一个实例，所以知道该方法已经正常运行了。考虑到多方测试，需要再编写一个测试，用来验证它能找到多个实例：

```

[Test]
public void ShouldFindTwoSInMysterious()
{
    var stringToCheck = "mysterious";
    var stringToFind = "s";
    var expectedResult = 2;
}

```

```
var classUnderTest = new StringUtilities();

var actualResult = classUnderTest.CountOccurences(stringToCheck,
                                                  stringToFind);

Assert.AreEqual(expectedResult, actualResult);
}
```

在运行这两个测试时，可以看到这段代码中有个问题，如图 1-5 所示。

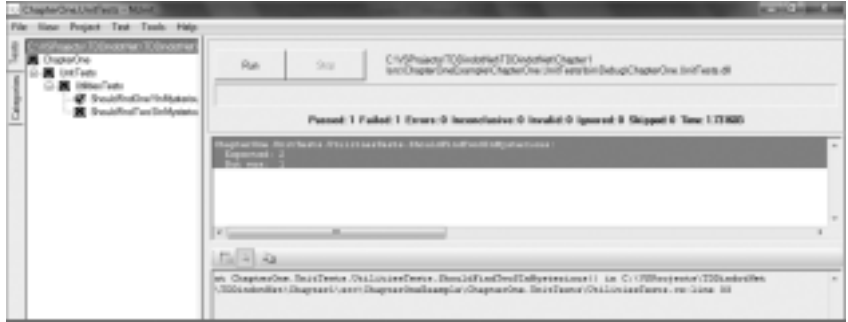


图 1-5

测试已经找到代码中的错误。具体来说，for 循环对目标字符串的循环次数比实际需要次数(string length - 1)少一次。在找到这一缺陷之后，可以对代码进行修正：

```
for (var characterIndex = 0;
     characterIndex <= stringAsCharArray.GetUpperBound(0);
     characterIndex++)
```

现在运行测试时，代码将按预期方式运行，如图 1-6 所示。

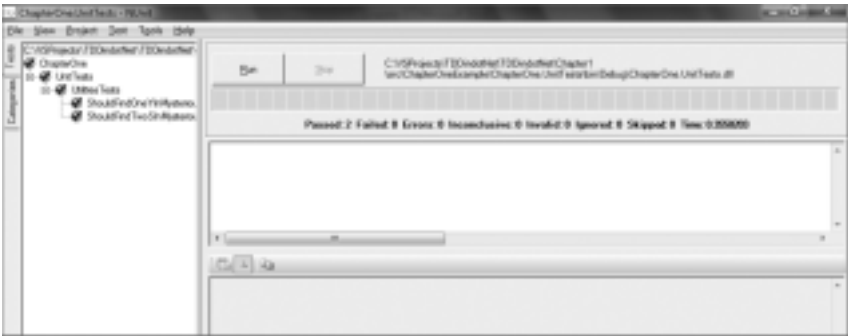


图 1-6

现在假设在继续开发字符计数方法时，又遇到了一项新需求。业务用户希望这个搜索过程不区分大小写。也就是说，该算法不应当考虑字符是大写还是小写。第一步是编写一个测试，来表达这一新需求：

```
public void SearchShouldBeCaseSenstive()
{
    var stringToCheck = "mySterious";
```

```

var stringToFind = "s";
var expectedResult = 2;
var classUnderTest = new StringUtilities();
var actualResult = classUnderTest.CountOccurrences (stringToCheck,
                                                    stringToFind);
Assert.AreEqual(expectedResult, actualResult);
}

```

图 1-7 显示，在运行这一测试时，当前的实现不满足该新需求。



图 1-7

接下来更新这个方法，以通过测试，同时确保其他两个测试不会失败。这个改变非常容易；只需要在运行这个搜索算法之前，将要搜索的字符串和字符改为大写：

```

var stringAsCharArray = stringToCheck.ToUpper().ToCharArray();
var stringToCheckForAsChar = stringToFind.ToUpper().ToCharArray()[0];

```

图 1-8 显示了再次运行该测试得到的结果。只需要进行这一修改就可以通过新的测试了，不会导致现有测试失败，所以满足了这一需求。

现在已经部署了第一版本的字符串实用类，很快就会出现第一个缺陷。当用户将 null 作为字符串搜索时，会抛出一个 null 引用异常。可以对调用代码的责任提出质疑，要求它在调用该方法之前检查所提供的值，还可以宣称允许出现 null 引用异常；毕竟，这个字符串就是 null。但事实上，好的开发人员应当意识到所有输入都是魔鬼，必须对其有效性进行独立验证。最后，业务用户可能宁愿返回 -1。编写一个测试来演示该缺陷：

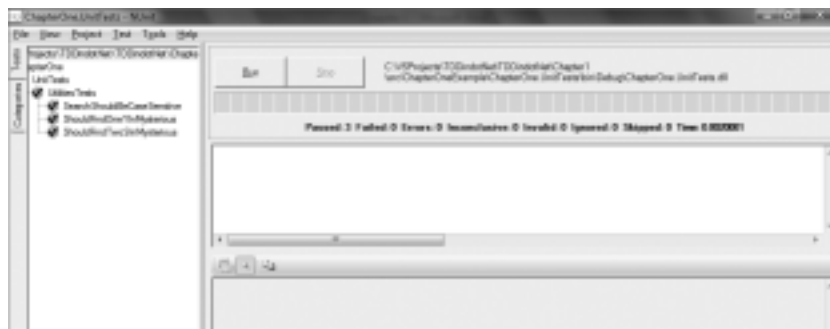


图 1-8

```
public void ShouldBeAbleToHandleNulls()
{
    string stringToCheck = null;
    var stringToFind = "s";
    var expectedResult = -1;
    var classUnderTest = new StringUtilities();

    var actualResult = classUnderTest.CountOccurences(stringToCheck,
        stringToFind);

    Assert.AreEqual(expectedResult, actualResult);
}
```

和预期一样，运行该测试会失败，如图 1-9 所示。

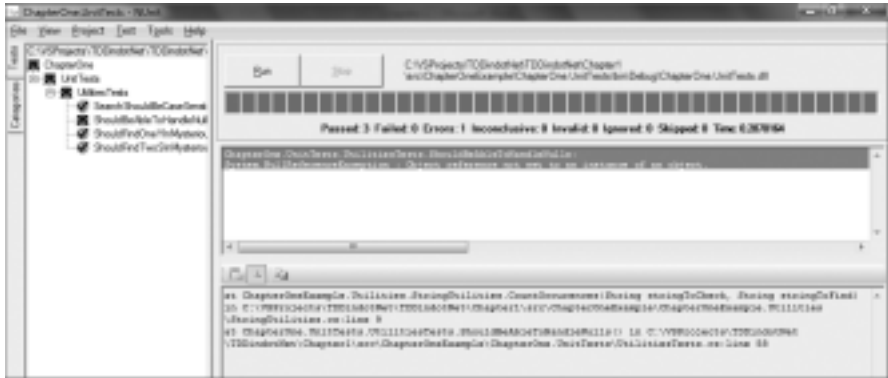


图 1-9

还需要再修改一次代码，这次验证输入参数，如果数据不能通过验证，则给出相应的响应结果：

```
public int CountOccurences(string stringToCheck, string stringToFind)
{
    if (stringToCheck == null) return -1;
    var stringAsCharArray = stringToCheck.ToUpper().ToCharArray();
```

再次运行该测试时，对代码所做的修改纠正了这个缺陷，如图 1-10 所示。

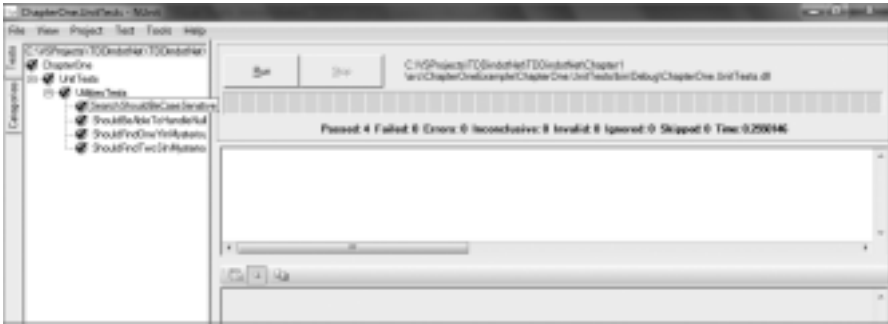


图 1-10

除了确保已经弥补了这一缺陷之外，该测试还能确保该缺陷将来不会再次出现。

1.6 本章小结

从本章可以看出，软件开发的历史已经转了整整一圈，又回到了对迭代开发的喜爱。还看到，“敏捷宣言”已经为当今的新迭代方法创建了一个框架。软件开发人员也必须了解改变的价值，找出能够调整自己作品的方式，以便能跟上业务后续部分的改变步伐。本章还介绍了一个基本示例，说明如何使用测试驱动开发(TDD)来编写易于实现和维护的可靠软件。还介绍了这些测试如何有助于避免引入新的缺陷，同时还提供了一种框架，用于在不损害当前功能的前提下添加新功能。最后，介绍了一些使用 TDD 时所需要的工具。

第 2 章

单元测试简介

本章内容

- 什么是单元测试
- 单元测试与其他类型的测试有何区别
- 单元测试框架如何有助于快速轻松地编写单元测试
- 在实践 TDD 时，在测试中模拟外部资源为何非常重要
- 简要概述 NUnit 单元测试框架和 Moq 模拟框架，这是 .NET 世界中两个非常流行的 TDD 工具

单元测试(Unit Testing, UT)是 TDD(测试驱动开发)的基础。当单元测试与业务需求保持一致，正确地反映业务需求时，它们几乎就是活的设计文档——只需要按下按钮就可以验证编写的代码。单元测试不难编写，不过它们的确需要对常用的软件编写方法做一点小小改动。它们还向有经验的开发人员介绍了一些新概念(如代码隔离)和一种思路：利用替代对象或模拟对象，使测试能够仅专注于被测代码。这些新概念对 TDD 而言是必需的。通过编写独立的、可重复的、有针对性的测试，可以确保代码满足业务需求，而且这些代码能够继续发展，所做的修改不会使代码偏离业务要求。NUnit 和 Moq 等工具和框架用这些概念为开发人员提供帮助，可以使开发过程更容易、更快速，而且具有更高的回报率。

2.1 什么是单元测试

多年以来，针对“单元测试”给出了许多不同的甚至是互相排斥的定义。许多人提到“单元测试”时，实际是指组件测试、集成测试，甚至是用户验收测试(UAT)。这通常会在软件开发中导致很多混淆。在 TDD 中，如果考虑到 TDD 单元测试实践处于多么核心的位置，就会发现这种混淆的严重性更是前所未有的。为了与开发人员同行进行交流，对“单元测试”给出一个统一的定义非常重要。

2.1.1 单元测试的定义

用最简单的话来说，“单元测试”就是针对一个工作单元设计的测试。在这里，“一个工作单元”是指对一个方法的一个要求。第 1 章的示例测试了一个算法，它计算一个给定字符在给定字符串中的出现次数。我们不知道字符串或字符来自哪里；对于这个给定工作单元(也就是计算给定字符在给定字符串中出现的次数)，这些细节是不重要的。其他任何事情都超出了这个工作单元的范围。

这种测试的好处是：它仅局限于一个特定的工作单元。我们可以将自己的所有精力集中于工作单元。不需要知道系统中其他参与者的细节(除非与他们有直接依赖关系)。这样，编写测试和最终代码就都很容易了。另外，如果代码中出现缺陷，导致测试失败，则可以肯定该缺陷就在这个特定的工作单元中。不必再深入到“兔子洞”里去查找该缺陷发生在哪一层代码中。如果字符计数测试失败，那就知道问题就在这里。

从编写风格上来说，可以采用各种不同方式编写单元测试，但所有单元测试都有一些共同的特征：

- 与其他代码隔离
- 与其他开发人员隔离
- 有针对性
- 可重复
- 可预测

现在已经知道了真正的单元测试的特征，2.1.2 小节将介绍应当避开的一些模式和特征。

2.1.2 什么不是单元测试

被测代码与系统其他部分之间存在着边界，初学者常希望避开那些跨界测试。这一点适用于一切内容：从同一应用程序中的其他类和服务，到数据库、Web 服务和任意其他外部依赖项。当测试开始渗透到其他类、服务和系统时，测试就会失去针对性，失败时会很难找到缺陷代码。单元测试背后的思想是，仅测试这个方法中的内容。测试失败时，不希望必须穿过几层代码、数据库表或者第三方产品的文档去寻找可能的答案。

跨界时还会产生另一个问题，特别是边界是一个共享资源时，如数据库。与团队中

的其他开发人员共享资源时，可能会污染他们的测试结果，自己的测试结果也可能会受他们的影响。如果测试向数据库中写入一个值，读取它，然后删除它，则该测试显然破坏了边界。如果系统只有一位开发人员，可能不会成为问题。但如果其他开发人员也在处理同一代码，而且他们都尝试运行相同的测试，在相同的数据库的同一个表中创建及(或)删除相同的记录，则这些测试之间非常可能相互干扰。这就形成了不稳定的测试环境。

这并不是说那些跨过被测类与方法边界的测试就没有价值。一组代码如果不能被集成到一个更大的系统中，其价值就是很有限的。集成测试是极为重要的，可以用来衡量系统的整体健康程度，确保所开发的每个组件时都能合并在一起，形成一个系统。集成测试也应当在开发期间创建。一种常见做法是先完成按特征划分的工作单元(已经完成并通过了单元测试)，然后创建集成测试，以确保刚刚创建的代码可以使用应用程序的其余部分或可以供其余部分使用。一个非常正确的座右铭就是：“早集成，常集成”。

考虑到保持测试的针对性，还希望确保这些测试只测试一件事情。根据“单一职责原则”(在第3章介绍)，每个类或方法应当只有一个修改理由。这一原则的一种合理、可行的扩展是每个类和方法应当只有一个目的，只做一件事情。结果，每个单元测试应当仅测试一组需求。在第1章的示例中，那个方法做了一件事情：计算一个特定字符在一个字符串中出现的次数。它没有说明这个字符出现在字符串中的什么位置、实例的大小写是否相同，也没有说明除了这个字符出现多少次之外的任何其他信息。根据“单一职责原则”，这些功能应当由 `StringUtility` 类中的其他方法提供。同样，`StringUtility` 类不应当提供用于处理数字、日期或复杂数据类型的功能。它们不是字符串，根据定义，`StringUtility` 类只处理字符串。

如果发现所编写的测试对一件以上的事情进行了测试，就可能违反了“单一职责原则”。从单元测试的角度来看，这意味着这些测试是难以理解的非针对性测试。另外，随着时间的推移，在向方法或类中添加了更多的不恰当功能之后，这些测试很可能会变得非常脆弱。诊断问题也会变得富有挑战性，因为这些功能可能是通过多个途径实现的。判断是其中哪一条路径中包含缺陷，也可能变得非常困难。很多时候，这些分支之间的交互可能会导致缺陷。这类缺陷也难以修正，因为对一个分支的修改可能会在其他分支中导致不可预测的行为。

还有一种违反这一概念的情景，就是将单元测试和集成测试混合在一起。记住，好的单元测试不会突破被测内容的边界。如果发现在单元测试中测试了不止一件事情，则要研究一下这个测试自身，可能对边界施加了太多压力。

单元测试应当是可预测的。在针对一组给定的输入参数调用一个类的方法时，其结果应当总是一致的。有时，这一原则可能看起来难以遵守。例如，如果正在编写一个日用品交易应用程序，黄金的价格很可能上午9点是一个值，在下午2点半是另一个值。还可能存在一些情景，需要能够在系统中创建随机处理功能。

在这些情况下，好的设计原则是将提供不可预测数据的功能抽象到一个可以在单元测试中模拟的类或方法中(关于“模拟”，将在2.3节介绍)。在这个日用品交易系统中，应当

模拟那个用来提供随时间变化的价格的服务。根据需要，可能希望在每次测试时总是返回相同值。或者，可能希望它提供一个给定时刻的预定值。在存在随机功能的情况下，一种不错的设计实践是：将这种随机功能包装在一个也可以由测试模拟的服务类中，使它总是返回相同的值或值序列。利用这些可预测的数据集，可以编写一些特定的测试，有效地检测代码，而不用担心在每次运行测试时都返回不同结果。

2.1.3 其他类型的测试

本书的重点是单元测试，但对于软件开发来说，它们并不是唯一重要的测试。单元测试验证一个系统的一些方面——内部类、方法和满足业务需求的服务，并提供必要的功能。完整的应用程序远不止这些基于业务的内部对象。

用户界面测试验证：应用程序的用户界面能够正常操作，可以供应用程序的用户使用，并且提供访问所有必要功能的方式。在每一个准则中，都需要综合考虑很多事情。这个应用程序是由用户通过 Web 浏览器访问的基于 Web 的应用程序吗？它是不是一个类似于 Microsoft Word 或 Excel 的基于 Windows 窗体的应用程序？Silverlight 怎么样呢？可用性要求包括哪些内容？是否需要使视力障碍人士也能访问所开发的应用程序？用户是谁，他们对计算机的了解程度如何？他们是公司的职员，还是外部客户？安全性如何？

用户界面测试显然不是小事。在开发高质量应用程序时，它同样很重要。许多工具有助于实现各种用户界面测试的自动化。有些针对基于 Web 的应用程序，有些则针对 Windows 窗体。最后，执行此类测试的最佳方法是雇用一個经验丰富、知识渊博的质量保障部门。一个了解业务的熟练 QA 工程师绝对是“人有所值”的。通常是由这些人作为最终的仲裁者，决定一个系统是可以部署生产了，还是必须发回返工。

集成测试是软件开发的一个重要步骤，不应忽略它直到开发结束才匆匆进行。有许多开发人员就在准备部署系统的前夜还在和一些缺陷做斗争，这些缺陷在将系统的各部分组合为一个应用程序时也会出现。如果在实现一个功能之后就将其与系统的其余部分集成在一起，则可以改善这一情况。事实上，敏捷方法的常见做法就是尽快把软件提交给用户，哪怕它还没有全部完成。显然，在集成应用程序之前，这是不可能实现的。因此可以推断：全面的集成测试是成功敏捷方法的关键组成部分。

单元测试和集成测试都是将各个功能和应用程序作为一个仅有一位用户的完整产品进行验证。但如果有 100 位用户同时访问它，会发生什么情况呢？这一问题是通过对应应用程序进行压力测试来解决的。压力测试只是创建了一些可以模拟多位用户同时与应用程序交互的条件。压力测试设计用来衡量在负载之下的响应时间，以及当一个应用程序跨越多个资源时其伸缩性如何。大多数用户界面(UI)工具都提供了某种形式的压力测试。

压力测试在传统上是推迟到要向某种 QA 环境中部署时执行。当然，在将应用程序部署到实际生产硬件中之前，很难确定基准性能的量度标准。但我仍然认为，应当在整个应用程序开发过程中以某种形式进行压力测试。在早期和非生产硬件上执行压力测试时所生成的数据，不能用于证明应用程序在生产条件下达到实际预期性能。但是，在一个应用程序

的早期开发过程中进行压力测试并持续收集数据，这是有好处的。在生成该应用程序时，开发团队马上就可以知道：新功能是否对应用程序的性能或可伸缩性产生了直接的负面影响。

例如，假定有一个处于开发过程的应用程序，当有 100 个并发用户时，平均操作需要 2s。然后开发团队添加两项新功能，当用户数目不变时，这个平均时间突然跳至 5s。显然，这两项功能中的某一项对应用程序性能产生了影响。因为一直在执行压力测试，所以必然是由这两个新完成功能之一导致的。如果将压力测试推迟到最后，这一信息就会难以查找，用户只是抱怨系统的速度很慢。通过始终执行压力测试，可以找到问题所在，解决问题的速度和效率也要高得多。

压力测试中的一个重要步骤是确定一个可供使用的性能尺度标准。系统每小时应当能够处理多少个事务？系统应当支持多少并发用户？系统的可接受响应时间是多长？在任何开发人员开发的应用程序中，都有一些遭到用户的抱怨：“它的速度就是很慢”。当然，许多系统都有进行优化的空间，但如果没有某种形式的衡量尺度来测试性能，也没有对目标性能的某种定义，很难对这种主观性很强的抱怨进行有意义的优化。需要问一下，“它变慢的体现是什么？”

大多数程序都曾经历过某种形式的用户验收测试(UAT)，通常是在应用程序开发过程的最终进行。这种测试发生在业务用户最终要在现实情景中使用此软件并评估其性能时。如果应用程序满足了他们的要求，用户就会签收应用程序，接下来通常就是生产了。

在瀑布方法中，用户验收测试通常是项目计划的最后一步，之后就要部署应用程序进行生产了。在大多数情况下，这些新应用程序的任何一部分，用户都是首次看到。由于项目团队的工作场所通常远离业务用户，所以第一轮用户验收测试通常是以用户列出一长串不喜欢的内容清单而结束的。其中有些要修正，而有些问题则被认为是优先级较低，用户被告知，他们只能容忍这些问题。从项目开始到用户验收测试，通常会有很长一段时间，应用程序甚至可能已经不能反映业务的当前需要了。

设想正在建造一幢房子。对大多数人来说，房子可能是一生中购买的最贵重东西了。如果能够拥有一个根据自己的愿望和规格建造的房子，的确是一件值得兴奋的事。但是设想一下，最后一次离开建筑设计师的办公室之后，直到房子完工之前，再也不能看到这个房子的进展，也不能对蓝图和设计进行任何修改。想象一下，你第一次看到这个房子就是入住的那天。你会认为这个房子在多大程度上反映了你的真正愿望呢？它可能反映了蓝图上的东西，但在图画与实际结构之间还是有很大区别的。令人惊讶的是，许多公司仍然在采用这种方式开发软件。

在使用敏捷开发方法的组织中，在用户验收测试之前，会多次向用户展示处于开发过程中的应用程序。这样，用户就能找出应用程序中有哪些功能或方面未达到期望效果，确保所开发的应用程序与业务需求保持一致。开发人员和开发经理也可以从这些短暂反馈循环中受益，因为在将缺陷引入系统之后，如果能很快发现，就能更快速、更轻松地修复它，而且成本也比较低。

2.2 NUnit 一览

许多工具可以帮助执行单元测试。自从 TDD 在 2005 年开始流行，框架和工具的市场就有了爆炸性发展。最流行的单元测试工具之一是单元测试框架。这些框架允许定义单元测试代码，控制测试的执行，还提供了一个应用程序来运行测试，并在成功完成测试套件中的每个测试后给出报告。NUnit 是最流行、最成熟的 .NET 单元测试框架之一。

2.2.1 什么是单元测试框架

在单元测试框架出现之前，开发人员在创建可执行测试时饱受折磨。最初的做法是在应用程序中创建一个窗口，配有“测试控制工具(harness)” 。它只是一个窗口，每个测试对应一个按钮。这些测试的结果要么是一个消息框，要么是直接在窗体本身给出某种显示结果。由于每个测试都需要一个按钮，所以这些窗口很快就会变得拥挤、不可管理。一些富有进取精神的开发人员开始使用控制台应用程序，它们可以执行每个测试并将结果输出到控制台。这是一个进步，因为开发人员增加了在测试上花费的时间，缩短了测试工具上花费的时间。这样做还有一个好处：可以很轻松地建立应用程序的服务器上自动运行测试。

这些方法在一定程度上是有效的，但它们没有为创建、执行和查询测试提供公共方法。单元测试框架(如 NUnit)希望能够提供这些功能。单元测试框架提供了一种统一的编程模型，可以将测试定义为一些简单的类，这些类中的方法可以调用希望测试的应用程序代码。开发人员不需要编写自己的测试控制工具；单元测试框架提供了测试运行程序(runner)，只需要单击按钮就可以执行所有测试。利用单元测试框架，可以很轻松地插入、设置和分解有关测试的功能。测试失败时，测试运行程序可以提供有关失败的信息，包含任何可供利用的异常信息和堆栈跟踪。

如前所述。NUnit 可能是用于 .NET 的最流行单元测试框架。之所以会这样，是有充分理由的。NUnit 是以 JUnit 为基础的，JUnit 是一种基于 Java 的单元测试工具。NUnit 性能稳定、使用简便、执行快速。NUnit 周围有一个非常活跃的用户社区。该社区为基础框架提供了新功能和改进，同时还创建了许多流行的加载项，使 NUnit 成为几乎任何单元测试需求的合适选择。NUnit 易于学习和使用，已经成为几乎所有 TDD 专业人员工具箱中的标准工具。

2.2.2 NUnit 基础知识

人们编写的大多数单元测试都有非常简单的模式：

- 执行一些操作，以建立测试。
- 执行测试。
- 验证结果。

- 必要时，重设环境。

测试本身就是一些方法，可以执行和调用要测试的语法。这些方法必须驻存于类中，该类被称为“测试容器”。下面的示例给出一个类，用 TestFixture 特性来表明它是一个测试类，用 Test 属性表明方法是用于测试的：

```
namespace NUnitExample
{
    [TestFixture]
    public class ExampleTests
    {
        [Test]
        public void TestMethod()
        {
            Debug.WriteLine("This is a test");
        }
    }
}
```

有时，在运行测试之前需要进行一些设置，如填充数据集、实例化类或设置一个环境变量。在这种情况下，可以使用 Setup 特性来定义一个将在运行测试之前执行的方法：

```
namespace NUnitExample
{
    [TestFixture]
    public class ExampleTests
    {
        private string _testMessage;

        [SetUp]
        public void SetupForTest()
        {
            _testMessage = "This is a test.";
        }

        [Test]
        public void TestMethod()
        {
            Debug.WriteLine(_testMessage);
        }
    }
}
```

本例中，在 SetupForTest 方法中对变量 _testMessage 进行初始化。当 TestMethod 执行时，将赋给变量 _testMessage 的值("This is a test.")写到“调试”控制台。当然，如果可以为测试提供设置代码，那就必须另行提供一些代码，在完成测试之后将测试所使用的资源复位。NUnit 提供了一个 TearDown 特性，用来实现这一功能：

```
namespace NUnitExample
{
    [TestFixture]
    {
```

```
public class ExampleTests
{
    private string _testMessage;
    [SetUp]
    public void SetupForTest()
    {
        _testMessage = "This is a test.";
    }

    [Test]
    public void TestMethod()
    {
        Debug.WriteLine(_testMessage);
    }

    [TearDown]
    public void TearDownAfterTest()
    {
        _testMessage = string.Empty;
    }
}
```

该示例重现了一种情景，在这种情景中可能希望或需要在完成测试后对资源进行复位。可能不需要复位实例变量，但可能希望复位环境变量或者回滚数据库事务。

现在执行测试，但它实际并没有测试任何东西。可以使用一个“断言(assert)”来改变这一情景。所谓“断言”，就是把最终测试结果告知测试运行程序的方式。在 NUnit 测试条件中有许多不用类型的断言，如两值相等，两个引用类型的变量是否指向同一对象，是否满足各种条件。本书后面的一些示例演示了如何使用名为 NBehave 的框架在测试中编写断言；其语法将在第 7 章介绍。目前，由于这些测试没有大量使用 NUnit 断言，所以将仅介绍最基本的一些。在第一个示例中，测试只是验证 _testMessage 的长度是否大于 0：

```
[Test]
public void MessageLengthGreaterThanZero()
{
    if (_testMessage.Length > 0)
    {
        Assert.Pass();
    }
    else
    {
        Assert.Fail();
    }
}
```

有一个 NUnit 断言可以验证 _testMessage 的长度大于 0，但该例只关注用于一般目的的基本断言。该测试检查 _testMessage 的长度。如果它大于 0，则此测试调用 Assert.Pass，它告诉测试运行程序：测试已通过，如图 2-1 所示。



图 2-1

为了查看 Assert.Fail 的工作情况，添加一个测试，以查看 _testMessage 中的取值长度是否大于 100：

```
[Test]
public void MessageLengthGreaterThanOrEqualTo100()
{
    if (_testMessage.Length > 100)
    {
        Assert.Pass();
    }
    else
    {
        Assert.Fail();
    }
}
```

在运行此测试时，由于调用了 Assert.Fail，所以该测试失败，如图 2-2 所示。



图 2-2

在 NUnit 中有许多其他类型的断言可供使用。对于将在后面章节生成的项目会使用 NBehave 编写断言，它所提供的语法更具业务友好性。但有了 Assert.Pass 和 Assert.Fail 的相关知识，如果了解其他类型的断言，就可以在需要时选择使用本机 NUnit。NUnit 库中包括以下一些可用断言。

- Assert.AreEqual(expected, actual)：该方法被重载，可以取任意类型的期望值或实际值，只要这两个实参的类型相同即可。也可以为每个实参传递不同类型的数值参数，因为该方法可以根据需要进行向上或向下的类型强制转换。例如，传递一个整型(integer)或 double 型不会导致错误。对于值的类型，该断言将验证期望形参和

实际形参具有相同值。在传递引用类型(对象)时,该方法将使用对象 Equals 方法的结果来判断此断言是否成功。推论断言 AreNotEqual 用来验证两个值不相等。

- Assert.AreSame(expected, actual) : 该断言以引用类型为其实参,用于判断为期望实参传递的对象与为实际实参传递的对象是否相同。这意味着两个对象占据内存中的相同空间,而不是互为副本的关系。有一个名为 Assert.AreNotSame 的推论断言用来验证两个对象不是同一对象。
- Assert.IsTrue(bool) / Assert.IsFalse(bool) : 该断言是布尔变量或逻辑条件,其求值结果为布尔类型。
- Assert.IsNull(object) / Assert.IsNotNull(object) : 该断言将检查一个引用类型,并判断它是否为 Null。
- Assert.Greater(x,y) / Assert.GreaterOrEqual(x,y)——对两个实现 IComparable 界面的值类型或引用类型进行求值,以判断 x 是否大于 y ($x > y$),或者 x 是否大于等于 y ($x \geq y$)。与 AreEqual 类似, Greater /GreaterOrEqual 可以取两个不同类型的数字,只要能对其进行适当的向上或向下强制类型转换,以进行对比即可。 NUnit 还提供了 Assert.Less 和 Assert.LessOrEqual,它们的行为方式相同,但用来验证 x 小于 y ($x < y$),或者 x 小于等于 y ($x \leq y$)。

这里列出的断言是 NUnit 中最常用的断言, NUnit 库中还有许多可用断言。如果希望获得 NUnit 中所有可用断言的完整列表,可以访问 www.nunit.org,其中给出了按版本排列的完成文档。

2.3 与模拟对象分离

精心编写的软件会尽力限制依赖,但迟早需要将各组件组合在一起构成一个更大的组件。这些组合关系在应用程序中生成一个由依赖组成的网,终结于外部资源,如数据库、Web 服务、文件系统或其他资源。模拟对象(Mock object)用来在应用程序中代表这些其他组件,它们有时代表外部资源。利用这些对象测试代码时,不需要担心与外部资源进行交互的后果。

2.3.1 模拟为什么重要

在为一个方法编写单元测试时,目的是仅测试该方法中的代码。这是有意为之的,是要隔离被测代码。这样做的理由是希望仅评估被测代码在给定条件下的有效性。对条件的定义,不只限于该方法的输入数据,还有用于执行此代码的上下文和环境。

通过隔离该代码,可以确保任何未能通过的测试都恰好指向特定方法中的问题,而不是指向位于该系统偏远角落的方法。对于该示例来说,可以假定该方法所处理的其他组件都是正确的,这些组件本身已经通过测试,可以相信这些组件及测试的质量。(第 5 章将更多地讨论如何对待第三方组件)。通常只关心当时摆在面前的方法。

单元测试还应当能够快速运行。即使是在小型应用程序中，如果正确地应用了 TDD，有数百个单元测试也是很正常的。我们的目的是让开发人员在开发过程中经常运行这些测试，以确保他们对类和方法所做的修改不会对系统的其他部分产生负面影响。如果在一次成功测试和一次失败测试之间只需要很少的工作，则只需要查看这一小部分工作以确认哪些内容出错。如果这些测试的运行需要花费几个小时，或者哪怕只是需要花费几分钟，开发人员也不会很频繁地运行它们。

一个从来都不会运行的测试是没有价值的。大多数需要很长时间来运行的测试都是这样的，这是因为它们要与外部资源(如数据库)进行交互。数据库调用可能很慢，特别是一次对一个开发数据库执行数百次操作时。模拟数据库的对象可以立刻为预定义形参返回预定义值，而不需要连接数据库和执行 SQL 查询。与直接查询数据库相比，这样的执行速度要快得多。

单元测试应当是可预测的、前后一致的。在编写方法及其相应测试时，应当能够确定：如果方法以 X 和 Y 为实参，将会返回 Z。外部资源和它们包含的数据是随时间变化的。如果方法和调用此方法的测试依赖于外部资源，则不能保证输入相同形参时总是能获得相同的结果。通过模拟实参这一资源，可以确保它们总是获得一致的可靠数据，用于测试方法。

最后一点，许多开发项目是由团队承担的，这些团队的成员显然不止一个人。此时在运行自己的单元测试时不能影响到队友的测试结果。也许测试套件中包含的一些测试会改变一个数据库表的状态，而其他测试可能依赖于该表中的数据。如果是这样，则非常可能出现以下情况：两个或更多个开发人员同时运行此测试，他们的测试都会失败。他们相互改变对方的数据，谁也不会得到所期望的数据。通过模拟这些外部资源，可以确保不会使其他开发人员运行的测试因为错误数据或预料之外的数据而发生错误。

2.3.2 虚拟、伪对象、存根和模拟

模拟(mock)是一个多少有些通用性的术语，它包含了一系列在单元测试中使用的替代对象。虚拟(dummy)对象是代替外部资源的简单模拟。它们通常会在调用一个方法时为该方法返回预定义响应，但通常不会根据输入参数而改变该响应。许多开发人员不希望因为模拟框架而产生开销，也不需要该模拟所提供的功能，在这种情况下，会在自己的测试中使用手工虚拟对象。



提示：

对于这些示例，我不会使用模拟框架。我会以手工方式实现这些模拟以说明每种模拟类型的内部情况，使您能够更好地理解每种模拟类型中发生了什么。2.4 节会介绍在本书示例中使用的模拟框架 Moq。

本节的示例包含对 `DependentClass` 类执行的测试。`DependentClass` 类依赖于一个实现

接口 IDependency 的类。DependentClass 有一个 GetValue 方法，该方法以一个字符串为形参。DependentClass 类中 GetValue 的实现调用 IDependency 实现的 GetValue 方法，这个 IDependency 实现是在创建 DependentClass 的实例时，作为构造函数实参传递给 DependentClass 的。DependentClass 和 IDependency 的定义如下：

```
internal interface IDependency
{
    int GetValue();
}

internal class DependentClass
{
    private readonly IDependency _dependency;

    public DependentClass(IDependency dependency)
    {
        _dependency = dependency;
    }

    public int GetValue(string s)
    {
        return _dependency.GetValue(s);
    }
}
```

在以下代码中有一个测试，在向它传递 DummyDependency 的一个实例时，它会测试 DependentClass 的实现，DummyDependency 是实现 IDependency 接口的虚拟对象：

```
[TestFixture]
public class DummyTestClass
{
    [Test]
    public void TestWithADummy()
    {
        var dependency = new DummyDependency();
        var dependentClass = new DependentClass(dependency);
        const string param = "abc";
        const int expectedResultOne = 1;
        var resultOne = dependentClass.GetValue(param);
        Assert.AreEqual(expectedResultOne, resultOne);
    }
}

public class DummyDependency : IDependency
{
    public int GetValue(string s)
    {
        return 1;
    }
}
```

该测试方法创建 DummyDependency 类的实例,并将其作为 DummyDependency 的实现进行传输,在实例化 DependentClass 时会需要这一实现。该测试调用 DependentClass 实例的 GetValue,并向它传递"abc"作为参数。

DummyDependency 类有一个 GetValue 实现,它只是返回数值 1。这样就可以满足测试条件了;取值 1 是所期望的结果,虚拟对象 DummyDependency 满足这一要求。

但可以很容易地看出虚拟对象的局限性。无论测试传递什么值,虚拟对象都只能以一种方式做出反应——返回数值 1。在某些情况下,这种限制不是问题。但对于大多数验证业务领域逻辑的测试来说,有必要采用更可靠的模拟方式。

相对于虚拟对象来说,伪(fake)对象和存根(stub)前进了一步,因为它们可以根据输入参数改变其响应。例如,数据库存根可以针对用户 ID 61 返回姓名 Rick Nash,针对用户 ID 1 返回姓名 Steve Mason。除此之外,不再调用其他逻辑。存根通常不能跟踪方法的调用次数,也不能跟踪对一方法序列的调用顺序。下面给出一个存根的示例:

```
[TestFixture]
public class StubTestClass
{
    [Test]
    public void TestWithASTub()
    {
        var dependency = new StubDependency();
        var dependentClass = new DependentClass(dependency);
        const string param1 = "abc";
        const string param2 = "xyz";
        const int expectedResultOne = 1;
        const int expectedResultTwo = 2;

        var resultOne = dependentClass.GetValue(param1);
        var resultTwo = dependentClass.GetValue(param2);
        Assert.AreEqual(expectedResultOne, resultOne);
        Assert.AreEqual(expectedResultTwo, resultTwo);
    }
}

public class StubDependency : IDependency
{
    public int GetValue(string s)
    {
        if (s == "abc")
            return 1;
        if (s == "xyz")
            return 2;
        return 0;
    }
}
```

上面的示例生成了一个新类 StubDependency,该类实现了 IDependency 界面。与 Dummy-

Dependency 不同的是，StubDependency 的 GetValue 实现有一个逻辑，可以根据不同输入形参返回不同值。该存根能够以不同的指定方式对不同激励做出响应。与虚拟对象相比，它提供的模拟方式要可靠得多。

模拟是在伪对象和存根的基础上又前进了一步。模拟提供的功能与存根相同，但更复杂。可以为它们定义一些规则，规定必须以何种顺序调用它们 API 的方法。大多数模拟可以跟踪一个方法被调用了多少次，并根据这一信息做出反应。模拟通常知道每个调用的上下文，并可以在不同情景下做出不同反应。因此在进行模拟时，需要对被模拟的类有所了解。

下面的模拟示例向 IDependency 界面和 DependentClass 类添加了一些成员：

```
internal interface IDependency
{
    int GetValue(string s);
    void CallMeFirst();
    int CallMeTwice(string s);
    void CallMeLast();
}

internal class DependentClass
{
    private readonly IDependency _dependency;

    public DependentClass(IDependency dependency)
    {
        _dependency = dependency;
    }

    public int GetValue(string s)
    {
        return _dependency.GetValue(s);
    }

    public void CallMeFirst()
    {
        _dependency.CallMeFirst();
    }

    public void CallMeLast()
    {
        _dependency.CallMeLast();
    }

    public int CallMeTwice(string s)
    {
        return _dependency.CallMeTwice(s);
    }
}
```

IDependency 接口和 DependentClass 类包括 3 个方法：CallMeFirst、CallMeTwice 和 CallMeLast。在开发人员使用的许多 API 中，有些方法必须以指定顺序调用，有些方法需

要被调用指定次数。从 IDependency 接口和 DependentClass 类的新方法名称就可以看出，CallMeFirst 方法必须被首先调用，CallMeTwice 方法必须被调用两次，CallMeLast 方法必须是一个特定事务最后调用的方法。

为了执行这些规则，需要编写一个比前面两个示例稍微高级和复杂的模拟类：

```
public class MockDependency : IDependency
{
    private int _callMeTwiceCalled;
    private bool _callMeLastCalled;
    private bool _callMeFirstCalled;

    public int GetValue(string s)
    {
        if (s == "abc")
            return 1;
        if (s == "xyz")
            return 2;
        return 0;
    }

    public void CallMeFirst()
    {
        if ((_callMeTwiceCalled > 0) || _callMeLastCalled)
            throw new ArgumentException("CallMeFirst not first method called");
        _callMeFirstCalled = true;
    }

    public int CallMeTwice(string s)
    {
        if (!_callMeFirstCalled)
            throw new ArgumentException("CallMeTwice called before CallMeFirst");
        if (_callMeLastCalled)
            throw new ArgumentException("CallMeTwice called after CallMeLast");
        if (_callMeTwiceCalled >= 2)
            throw new ArgumentException("CallMeTwice called more than twice");
        _callMeTwiceCalled++;
        return GetValue(s);
    }

    public void CallMeLast()
    {
        if (!_callMeFirstCalled)
            throw new ArgumentException("CallMeLast called before CallMeFirst");
        if (_callMeTwiceCalled != 2)
            throw new ArgumentException(
                string.Format("CallMeTwice not called {0} times",
                    _callMeTwiceCalled));
        _callMeLastCalled = true;
    }
}
```

为了确保 DummyClass 所用 IDependency 实现的方法能够正确应用，有必要构造一个模拟，它不只是返回值，还封装该 API 的所有规则。每个方法都必须检查，确保它们的调用顺序正确。对于 CallMeTwice 而言，该方法必须验证其调用次数是正确的。

在展示一个基于 MockDependency 模拟类的方法之前，先声明几点。显然，根据该示例中的代码，人工模拟的效率很低、很费时间，并会使基本代码变得脆弱。正因如此，大多数开发人员都选择采用模拟框架而避免人工设计模拟。

其次，在该例中可以看出：MockDependency 类需要较好地了解 DependentClass 如何使用它。而所需知识的数量和类型在大多数情况下都明显违反了面向对象编程和优秀编码实践的许多规则。在某些情况下必须使用拥有这一知识水平的模拟，但如果发现自己需要频繁进行此类模拟，就应当再研究一下应用程序的设计和代码。

下面给出一个测试，它测试 DependentClass 实现，该实现使用一个基于 MockDependency 的 IDependency 实现：

```
[TestFixture]
public class MockTestClass
{
    [Test]
    public void TestWithAMock()
    {
        var dependency = new MockDependency();
        var dependentClass = new DependentClass(dependency);

        const string param1 = "abc";
        const string param2 = "xyz";
        const int expectedResultOne = 1;
        const int expectedResultTwo = 2;

        dependentClass.CallMeFirst();
        var resultOne = dependentClass.CallMeTwice(param1);
        var resultTwo = dependentClass.CallMeTwice(param2);
        dependentClass.CallMeLast();

        Assert.AreEqual(expectedResultOne, resultOne);
        Assert.AreEqual(expectedResultTwo, resultTwo);
    }
}
```

既然模拟有这么多选择，自然就会问一个很常见的问题：使用哪种模拟呢？答案是：视情况而定。一般情况下，可以优先使用伪对象和存根。如果需要实现与组件的复杂交互，那模拟要有用一些，它们所需要的配置和开销通常超过了大多数单元测试。采用存根还可以确保正在设计的系统是松散耦合的；如果要求调用方法知道 API 的大量使用规则，那就不是松散耦合了。最后，如果模拟资源的方法没有常规输出，则需要使用监视(spy)。

2.3.3 最佳实践与最差实践

在 TDD 中使用模拟时，应当了解以下概念和规则。

- 依赖项注入——TDD 之所以能成为可行的软件编写方式，模拟是关键概念。为了有效使用模拟，应用程序应当使用“依赖项注入”(Dependency Injection)。简而言之，这意味着：在对类进行内部实例化时，不是静态创建该类所依赖的对象，而是向其提供这些对象的一些实例，这些实例符合该依赖项所需要的接口。在创建对象时，应当将这些实例作为构造函数实参传递给对象。这样就可以很轻松地用模拟对象来代替实际对象，供完全集成的应用程序使用。依赖项注入将在第 5 章详细介绍。
- 为接口设计，而不是为实现设计——在将另一个类或资源用作依赖项时，应当关注的不是它如何执行自己的任务，而是关心其接口是什么。与此类似，在设计和构建服务的类时，应当使用接口来抽象 API 中的功能。这种设计不仅可以提高代码的可靠性、使其更易于扩展，还可以更轻松、更高效地实现模拟。
- 尝试限制依赖项——大多数代码都需要依赖于某种东西。可能依赖数据库来存储和检索数据，也可能依赖 Web 服务来验证用户，或者依赖应用程序中的另一个域服务。有鉴于此，应当采取措施来限制代码所依赖的东西。大量的依赖项不仅会意味着一个脆弱的系统，而且这样的系统也很难进行有效的模拟和测试。
- 不要模拟私有方法——如果正在编写一个测试，而且代码依赖于另一个类，应当仅模拟公共方法。即使这样，也应当仅模拟那些直接使用的公共方法；不要过多地进行模拟。模拟私有方法需要了解所模拟服务的内部功能，而根据封装规则，我们是不应当了解这些的。作为服务的使用者，应当仅关心公共接口上的方法；而看不到受保护的私有方法。
- 不要欺骗——在继续 TDD 实践时，有时可能会受到诱惑：通过模拟来走一些捷径。这可能是由于一个特定依赖项所需要的模拟要比通常使用的稍微复杂一些。也可能是存根方法必须返回一个复杂数据或对象图，而不喜欢创建它。不要让自己掉进陷阱。测试是应用程序的质量基础。测试依靠模拟和存根来确保它们能够正确地与系统中所包含的各种依赖项进行交互。如果用模拟走捷径，将会损害测试，进而使软件受到损害。

2.4 Moq 概览

.NET 有许多模拟框架。多年来，Rhino Mocks 曾经是用于 .NET 的模拟框架，许多应用程序都在测试套件中广泛使用它。除了 Rhino Mocks 之外，其他模拟框架(如 NUnit Mocks、TypeMock 和 Easy Mock)也都比较普及。它们都是非常优秀的框架，所提供的各种功能会在一定程度上吸引不同的开发人员和开发团队。一些开发人员仍然喜欢人工编写自己的模

拟测试，但随着模拟框架变得更为普遍、功能更全、使用更简单，这种人工编写模拟的情景会越来越少。模拟框架绝对要比自己编写模拟要好用得多，它可以使 TDD 体验更加高效。Moq 正在快速成为 .NET 最流行的模拟框架之一。它的功能丰富、使用简单，社区支持基础坚实，这些特点确保了 Moq 功能的持续改进与开发。

2.4.1 模拟框架做些什么

模拟框架提供了快速创建和使用存根与模拟的工具。利用模拟框架 API，可以创建自己的模拟，并在运行时注入自己的测试功能。模拟框架还提供了域特定语言(Domain-Specific Language, DSL)，用于为所模拟的资源定义执行规则。

2.4.2 关于 Moq

Moq 是一种根据 BSD 许可发布的开源 .NET 模拟框架。Moq 利用了 .NET 3.5 和 C# 3.0 中的新语言功能——特别是 lambda。Moq 的语法更具描述性，从而避开了许多模拟框架的记录/回放功能。Moq 是一种轻量级框架，其使用非常简单。Moq 拥有一个非常活跃的用户与支持社区，几乎每天都会检查基本代码，进行更新和错误修正。

2.4.3 Moq 基础知识

Moq 的使用非常简单。下面的示例说明如何根据该类和接口为测试创建模拟和存根：

```
public interface ILongRunningLibrary
{
    string RunForALongTime(int interval);
}

public class LongRunningLibrary : ILongRunningLibrary
{
    public string RunForALongTime(int interval)
    {
        var timeToWait = interval*1000;
        Thread.Sleep(timeToWait);
        return string.Format("Waited {0} seconds", interval);
    }
}
```

该方法的目的是模拟一个长时间运行的过程，如数据库查询。在该例中，首先展示一个长时间运行的过程如何使测试过程变得非常缓慢，因为测试必须要等待很长的时间才能执行。在该示例的后面，替换了这个长时间运行的过程，用一个代表数据库调用的模拟来充当数据库访问，实际上并没有访问数据库。该示例说明如何在不使用存根的情况下运行它：

```
[TestFixture]
public class MoqExamples
```

```

{
    private ILongRunningLibrary _longRunningLibrary;

    [SetUp]
    public void SetupForTest()
    {
        _longRunningLibrary = new LongRunningLibrary();
    }

    [Test]
    public void TestLongRunningLibrary()
    {
        const int interval = 30;
        var result = _longRunningLibrary.RunForALongTime(interval);
        Debug.WriteLine("Return from method was '{0}' ", result);
    }
}

```

可以看到,该测试只是实例化了 LongRunningLibrary ,并调用 RunForALongTime 方法。该测试随后把该方法的结果写到调试控制台。如果在测试运行程序中运行该测试,测试会一直等待该方法执行,然后将输出写到调试控件台,如图 2-3 所示。



图 2-3

这个测试最终得以通过,但等待 30 秒的时间还是太长了。设想一下,如果要运行 10 个甚至是 100 个这种测试呢。还要受到 RunForALongTime 方法的实现以及 LongRunningLibrary 类其余部分的左右。如果该类会发生变化或存在缺陷,测试的通过与否可能就没有什么意义了。

首先,必须为变量 _longRunningLibrary 声明一个新类型。不是生成具体实例,而是要求 Moq 根据 ILongRunningLibrary 接口创建一个模拟实例:

```

private Mock < ILongRunningLibrary > _longRunningLibrary;

[SetUp]
public void SetupForTest()
{
    _longRunningLibrary = new Mock < ILongRunningLibrary > ();
}

```

因为 _longRunningLibrary 现在是一个模拟了,所以在测试方法中处理它要稍有不同。要访问模拟实例的方法,需要使用模拟对象的 Object 属性:

```

[Test]
public void TestLongRunningLibrary()

```

```

{
    const int interval = 30;
    var result = _longRunningLibrary.Object.RunForALongTime(interval);
    Debug.WriteLine("Return from method was '{0}' ", result);
}

```

如果现在执行这一代码，不会得到所期望的结果，如图 2-4 所示。



图 2-4

该测试失败了，因为提供的是 LongRunningLibrary 的模拟实例，而不是模拟方法的实现。这一点很容易纠正：

```

_longRunningLibrary
    .Setup(lrl => lrl.RunForALongTime(30))
    .Returns("This method has been mocked!");

```

该例告诉 Moq 在用实参 30 调用 LongRunningLibrary 模拟实例的 RunForALongTime 方法时，只是希望它返回字符串 “ This method has been mocked! ”。如果现在在测试运行程序中运行该测试，将会通过测试，如图 2-5 所示。



图 2-5

现在，存根被设置为以数值 30 作为其输入形参。如果再次运行该测试，但这次向它传递数值 100，则从存根方法中获取的响应会与预期不同，如图 2-6 所示。

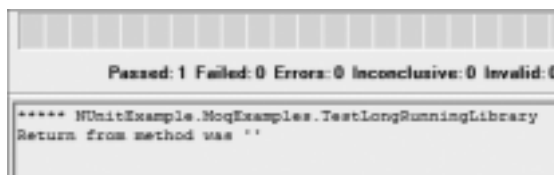


图 2-6

通常，在模拟方法时应当使用静态值。在进行测试时，希望确保对于预定输入能够得到预期的输出。但有些时候这种想法是不太切合实际的。此时 Moq 可以提供一些选择。

第一种选择是使用 Moq 的 It.IsAny 方法来指定 尽管存根希望获取一个指定类型的值，

但这种类型的任意值也都是可以接受的。例如，如果对存根方法设置代码进行如下修改：

```
_longRunningLibrary
    .Setup(lrl => lrl.RunForALongTime(It.IsAny<int>()))
    .Returns("This method has been mocked!");
```

然后运行同一测试，这次传递数值 100 作为间隔，而不是 30，将会在测试运行程序中得到如图 2-7 所示的响应。



图 2-7

可以接受指定类型的任意值是一项强大的功能，不要滥用。传统上，这类存根有一个问题：因为大多数存根都返回一个静态值，所以无法在另一端验证是否向存根传递了一个有效值。Moq 提供了一种工具，可以用来访问存根方法的输入形参，并在返回值中任意使用它们。在本例中，将该值附加在返回值的最后：

```
_longRunningLibrary
    .Setup(lrl => lrl.RunForALongTime(It.IsAny<int>()))
    .Returns((int s) =>
        string.Format(
            "This method has been mocked! The input value was {0}", s));
```

现在，在测试运行程序中运行测试时，可以看到输入形参已经包含在返回字符串值中了，如图 2-8 所示。

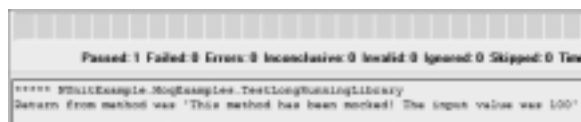


图 2-8

在某些情况下，可能希望验证方法会在一组指定条件下抛出异常。对于该例来说，假定 0 是无效间隔，应当抛出异常 `ArgumentException`。可以为存根添加一个设置，规定：在以 0 为实参时，应当由存根抛出异常 `ArgumentException`：

```
_longRunningLibrary = new Mock<ILongRunningLibrary>();
_longRunningLibrary
    .Setup(lrl => lrl.RunForALongTime(It.IsAny<int>()))
    .Returns((int s) =>
        string.Format(
            "This method has been mocked!
            The input value was {0}", s));
_longRunningLibrary
```

```
.Setup(lrl => lrl.RunForALongTime(0))  
.Throws(new ArgumentException("0 is not a valid interval"));
```

调用设置方法的顺序非常重要。Moq 会把这些规则放在堆栈中，因此，如果首先设置有关以 0 为形参的规则，那它会被有关所传递任意整数的规则所重写。这样就不会发生异常。一般来说，希望在堆栈中首先放置针对性最弱的设置，然后是针对性最强的。可以把这个堆栈看作为一种滤网；希望在合适的级别捕获各种可能出现的组合。如果前面的级别过于严格，那就会过于频繁地捕获各种条件，而且处于不恰当的级别。创建这些堆栈可能需要一些技巧，但多加练习之后就会变得非常简单了。

将间隔修改为 0 并重新运行此测试时，将会得到图 2-9 所示的结果。



图 2-9

Moq 是一种功能强大的模拟框架，它为各种不同的情景提供了模拟和存根。本章仅介绍了很小一部分 Moq 功能。在本书后续章节中，将根据需要介绍 Moq 的其他功能。但在创建存根的绝大多数情景中，有两种能力就够用了：一种是创建存根来响应值实参、返回静态或动态值，另一种是抛出异常。

2.5 本章小结

单元测试用于隔离和验证一小部分代码。如果测试中需要涉及对多个方法的调用，则要考虑这些测试和类的设计。集成测试很重要，但它们与单元测试的目的不同。不要将单元测试与其他类型测试的目的混淆。要使单元测试保持小型、简单，能够反映业务需要。在编写单元测试时，一定要仅依赖类的公共接口，不受限制地使用这些接口。

NUnit 和 Moq 是有效实践 TDD 的关键工具。NUnit 是一种针对.NET 的简单易用、非常流行的单元测试框架。它提供了一些工具，可以用来编写一些能够由测试运行程序所执行的单元测试，快速给出测试结果。利用 Moq 可以为外部资源和应用程序内的依赖项生成模拟和存根，从而可以仅关注特定的被测代码。Moq 允许创建一些代表这些资源的对象，为被测代码提供某一受到限制的功能，用来验证被测代码的功能。

第 3 章

重 构 速 览

本章内容

- 重构应用程序代码为什么重要
- 为什么 OOP 和 SOLID 等清洁代码原则有助于开发可靠的应用程序
- 如何识别并修正应用程序开发中最常见的设计和编码错误

没有代码是完美的。对于开发人员来说，接受这一事实是一件非常放松的事情。总是有那么一些事情原本可以做得更好，只是当时可能没有意识到这一点。看一下几年前编写的代码，可能会发现一些非常细小、微妙的内容，如果对它们进行一些修改，可以使代码执行更快、更能满足业务需要或者更容易维护。有时我们可能还会说：“如果我那时知道现在所知道的内容，我会采用一种完全不同的方法来完成它。”

重构就是改变类或方法的内部实现，其目的是提高代码的可读性和可维护性。重构还可以降低代码的整体复杂性，而不会改变类或方法的外部行为。这些改变可能非常简单，例如改变一个方法或变量的名称，从而将方法从一个类移动到另一个类；或可能是将一些大型类分割为几个较小的类。利用重构，可以持续不断地修改和改善所编写的代码。

本章解释了代码重构之所以重要的原因，回顾了面向对象编程和 SOLID 原则。本章还给出了一些常见的编码与设计问题，用通俗的话讲，就是“代码异味(code smell)”，并介绍了一些用于从代码中清除这些问题的常见模式。

3.1 为何重构

在使用 TDD 时，最初编写方法或创建类时的目的只是为了通过测试，再无其他了。这时不需要寻求什么风格，也不需要使代码精致和能重复使用，只需要尽力使所有测试都亮绿灯即通过所有测试。在达到这一目的之后，接下来就是改进代码。在实践中，这是一项非常注重实际经验的方法。许多开发人员花费了大量的时间，希望第一次就使自己的代码变得精致、漂亮。最终会遗漏一些非常重要的业务功能，不得不回头再将它们加到代码中。在开始时应当寻求完整的业务功能，然后再寻求使其变得完美，这样才能确保首先满足最高优先级(有效的业务代码)，然后再考虑其他方面。单元测试有助于确保：无论以重构的名义进行了哪些修改，这些代码都仍然能够满足业务需求。这就是“无畏重构(fearless refactoring)”一词的出处。

3.1.1 项目的生命周期

对许多开发人员来说，TDD 涉及大量概念，如模拟和依赖项注入(在第 5 章介绍)。但与 TDD 的一个主要原则(甚至是首要原则)相比，这些概念理解起来比较简单，也容易适应，这个原则就是：在编写代码之前编写测试。TDD(请记住，第一个 D 表示“驱动”)背后的思想就是将测试变成业务需求(有时还包含非功能性和技术性需求)的鲜活的可执行版本。利用这种“测试先行”的思想，可以确保不会编写那些对应用程序没有任何价值的代码。第 6 章将重点介绍一个采用“测试先行”开发方式的示例。

拥有测试之后，接下来的目标就是编写出仅能够通过测试的代码。这些代码既不要多，也不要少。策略应当是如何沿最短、最直的线路由 A 点(测试失败)到达 B 点(通过所有测试)。这时不要操心系统的其他方面。

另外，也不要操心那些未在测试中涉及的功能，重点考虑客户一定需要的功能。例如，有一个功能要求一个方法以整数为输入形参。我们只知道该输入形参的值不大于 100。测试应当“仅”对数值进行测试；而不要涉及限制这些数值不大于 100 的功能。在将这一功能排入日程，并为它创建测试之前，它不是需求之一。如果该功能永远没有被实际排入日程，那会怎么样呢？最好的结果是：编写了一些没有任何价值的代码，但必须与其他基本代码一起维护。而最糟糕的情况是：使一项本来应当非常简单的功能变得过度复杂了。有一个软件开发术语 YAGNI，它的意思就是“你不会用到它的(You Aren't Going to Need it)”。如果该功能被排入开发日程，那时可以编写这些测试和必要的代码，使它们通过测试。而现在，只需要考虑那些当前开发功能的测试即可。

如果已经通过测试，而且未破坏应用程序的任何其他内容(通过了所有测试)，就可以看看有哪些潜在的改进之处了。也许变量名或方法名不再能反映所提供值或功能的意图。也许在新代码中重复提供了某一功能。也许某一个函数稍长了一些，对其进行分解可以提高其可读性。无论是哪种情况，现在都该使用重构了。

首先对代码进行一点小小的改进，如重命名变量。在完成之后运行测试，验证没有改变类或方法的外部功能。不要仅对刚刚编写的代码运行这些测试；要运行所有测试。这一点是非常必要的，要确保没有对基本代码的其他内容产生不良的副作用。如果非常满意地发现所做的更改没有任何副作用，可以改进代码中的下一个方面，并再次运行这些测试。根据需要一直重复这一过程。这一实践方式被称为“红灯、绿灯、重构”。也就是说：开始是一个未能通过的测试，因为还没有实现能够通过这一测试的逻辑；然后实现此测试，使其得以通过；最后重构代码，对其加以改进，同时不会影响测试结果。

3.1.2 可维护性

大家已经接受这样一个观点：在采用 TDD 方式时，编写代码时的最初目的就是让测试能够通过。随着时间的推移，其支持的应用程序和业务向前发展时，该应用程序必须随之改变，要能够支持新功能或改变当前业务逻辑执行其功能的方式。能够快速而轻松地进行此类修改非常重要。为提高可读性而重构代码是管理应用程序的一个重要步骤。提高可读性不只是为了其他开发人员，也是为了我们自己。每个开发人员都曾经有过这样的不幸经历：在阅读自己过去编写的代码时，怎么也想不起这个代码要做什么、如何做、又是因为为什么要这样做。

为提高可维护性而进行的重构应当尽可能对代码进行简化。这包括：确保方法简短、控制结构简单，变量和方法名称能够明确地描述其用途。可以把为提高可维护性而进行的重构看成为将来的自己留下了一条提示，以便届时能够想起最初编写代码的动机。

3.1.3 代码度量

衡量一段基本代码的质量是一项比较主观的任务。关于质量是由什么构成的，有多少开发人员差不多就有多少种不同的观点。不过，一些细节透露出应用程序中到底有些什么。

代码测试覆盖率(代码覆盖率)衡量的是单元测试套件运行了多少基本代码。该度量是有争议的，因为这个数字只能说明问题的一部分。覆盖率小于 100%的基本代码并不一定意味着其质量较低。覆盖率为 100%的基本代码只能确保它的质量与执行该代码的测试相同。

在查看代码覆盖率时，首先要问的问题就是正在衡量基本代码的哪些部分。在大多数情况下，我发现更重要的是要根据源代码的功能部分来编写测试。这些功能部分几乎包含了除以下对象之外的所有内容：实体对象、数据传输对象，以及任何以包含数据为唯一目的的对象或结构。其原因在于，测试仅针对功能；而保存数据的结构中没有可变部分。因此，建议这些部分不应当包含在代码覆盖率度量中，因为它们可能会使测试结果值偏低。

另一方面，TDD 的目标是编写仅使测试通过所必需的代码。如果数据结构的某些字段在测试中不会用到，为什么还需要它们呢？代码覆盖率可以快速揭示哪些内容没有使用，可以检查是否需要这些字段。当然在某些情况下，数据结构会与使用这些字段的外部系统

交互。最终，在查看数据实体的代码覆盖率数字时，必须运用一些智慧，并准备做一些研究。

幸运的是，有关应用程序中功能代码的问题要更简短一些。对于新手来说，在代码覆盖率度量中应当始终包含所有功能代码。这些代码存在的唯一理由就是有测试需要它存在；因此，它的代码覆盖率应当很高。对于未被覆盖的所有此类代码都应当进行检查。检查结果必为以下两种结果之一：遗漏了一个测试；此代码不是必需的，应当删除。

遗漏了测试显然是一个问题，应当立即解决。它还揭示了在过多依赖代码覆盖率作为代码质量指标时的一个重要问题：代码覆盖率的良好性和可靠性不会超过所执行的测试。如果测试的质量很低，即使代码覆盖率为 100% 也没有任何意义。

如果有涵盖需求的完整测试集合，但仍然有一些代码未被单元测试所覆盖，那这部分代码可能不是必需的，可以删除。注意，是删除，不是将其变成注释加以忽略。没有用到的代码就像是基本代码中的寄生虫；它们不会提供任何价值，但必须维护，就好像是应用程序的有效组成部分一样。这些代码的唯一贡献就是干扰，所以应当删除它们，使开发过程更简单。如果正在使用版本控制系统(绝对应当使用这样一个系统)，在以后用到这些代码时还可以将它们再放回原处(这时一定要针对这些代码编写一个测试)。所以这些代码并不是真正永远消失了，但它们不属于工作代码。

近年来，有一个度量有些“回暖”(但与过去的使用方式不再一样)，那就是代码行数。过去，人们对这个数字的期望过高，有些开发人员和经理经常拿这一数字吹嘘，因为它表明开发人员做了大量工作。但是，这种观点有点落后了，开发人员应当努力用更少的代码来完成更多的任务。

事实上，方法或类中的代码越多，出现缺陷的几率就越大。使方法更简短、针对性更强，可以降低代码的复杂性和出现错误的可能。编写的代码越多，编写错误代码的机会也越多。

还有一种可能：如果编写了更多的代码行，很可能是重复提供了在应用程序其他地方已经存在的功能。在软件开发中有一个“不要重复自己(DRY)”的理念：每个功能单元应当仅在基本代码中出现一次。3.2.2 小节“1. 单一职责原则”部分将更详细地讨论这一理念。

“圈(Cyclomatic)复杂度”是很久之前就已经出现的度量。但直到最近，由于度量工具的涌入，它才在业务领域被广泛作为代码度量标准加以使用。圈复杂度测量的是一个方法、类或应用程序中存在的不同路径数。其数字越大，表明代码中可能存在的不同路径越多，复杂程度也就越高。

代码是复杂的，这是不争的事实。有时，代码对一项条件求值，然后以分支方式转到另一个方向。系统的复杂度越低，就越容易理解和维护。对于圈复杂度较高的代码，应当进行重构，使其变得更简单。这里有几种方法：找到具有多个不同路径的方法，将一些分支逻辑提取到其他方法中；利用多态(在本章后面介绍)删除那些只用于确定对象具体类型的控制结构。

3.2 整洁代码原则

一场仍在发展的软件设计运动已经开始支持这样一种观点：开发人员应当持续研究自己的技巧，以期能够编写出整洁代码。总而言之，这场运动的要点和哲学就是：开发人员应当努力学习，根据一套预先确定的实践方式与原则，尽最大可能编写出最佳代码。

3.2.1 OOP 原则

OOP(Object-oriented programming, 面向对象编程)是一种方法,用于将现实世界的对象抽象为可供代码使用的类。其思想是：如果可以在代码中建立业务问题的模型，那就可以更轻松地创建能够正确解决这些业务问题的应用程序，而且其解决方式能够更好地反映现实世界。大多数现代开发语言都支持 OOP。利用 OOP 可以更轻松地将现实世界概念化，开发出满足业务需要的应用程序，同时使各个代码单元短小，能够重用。拥有坚实的 OOP 基础，那就可以很轻松地理解和使用 TDD 中用到的一些概念。

笼统来说，OOP 有 3 个主要原则。过去几年，许多人一直坚持还有其他一些原则。这些人也不一定是错的。但如果对这些主张稍做分析，马上就会发现，这些附加原则要么是想吹捧一种语言相对于另一种语言的优势，要么是想贬低一种语言或平台，说它不是真正的 OOP。还有一种可能，就是它们组成了一个愿望清单，开发人员希望在自己选择的语言中见到这些功能。这些附加原则不一定是无效的；绑定和消息传送都是重要的概念，在某些语言中，它们和这里讨论的原则同等重要。但 OOP 的三条一般原则是：封装、继承和多态。

1. 封装

“封装原则”要求：创建的类应当是黑盒。换句话说，类的内部不允许客户端访问。客户端只有一种方式可以与类交互或改变其状态，那就是通过类的公共接口。客户端在任何情况下都不能也不需要知道一个类是如何执行其行为的。客户端只知道利用哪些方法及/或属性来获取它所需要的信息和交互。

真的不需要在意类的客户端如何完成其工作，只要获取所需要的结果即可。还需要确保类能够自由执行其任务，而不存在未经许可修改其内部状态的风险。在设计类时，非常重要的一点就是正确设定方法、属性和变量的作用域。只有那些作为公共接口组成部分的成员才应当声明为公共成员。只有那些必须供类的后代使用的成员才应当声明为受保护的。所有其他成员都应当是私有的。对于新接触这一概念的开发人员来说，有一条很不错的经验法则，那就是把所有成员都设置为私有的，在需要时再将某一成员变为公共成员。另外，永远都不要允许成员变量设置为私有之外的内容：如果一个后代类需要使用该变量，可以让它通过一个受保护的属性来访问。如果需要使该变量成为公共接口的一部分，应当将它包装在一个公共属性中。永远都不要简单地允许对内部变量进行不加监视或不加控制的访问。

2. 继承

在应用程序中构建类时，很快可以发现，一些类之间存在很自然的分层关系。这些类可能非常相似，但它们的功能并不是完全一样。在这些情况下，可以将公共功能放在一个基类中，然后再从该基类中派生出目的性更强的类来。这就是“继承”，是 OOP 中一个非常重要的概念。

假定正在编写一个工资发放应用程序。不同员工的计薪方式不同，有按小时计薪、按月计薪和按季度计薪，针对不同员工，应用程序中均有相应实体与之对应，如图 3-1 所示。

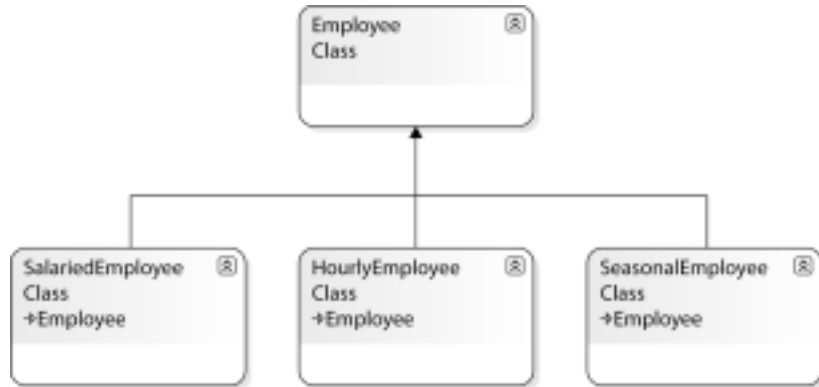


图 3-1

业务规则指出，对于每种员工，其薪水(工资、津贴、保险)计算方式不同。但是，所有这 3 类员工又有许多相同的内容。每个员工都有一些简历信息(姓名、地址、工作简历、税务信息)，有一些特定的过程对于各类人员也都是相同的，如从数据库中获取员工信息的过程。

利用继承，可以创建一个 Employee 基类，用它来封装这 3 类员工共同拥有的所有功能。然后分别为按小时计薪、按月计薪和按季度计薪的员工创建类。这 3 种目的性更强的类是从 Employee 基类派生或继承而来的。这些派生类利用它们与 Employee 基类的祖先/后代关系，可以访问其功能。

从基类继承而来的类可以采用不同的方式来处理基类所提供的功能。它们可以允许客户端针对其实例化对象调用这些方法，就好像这些派生类本身包含这些功能一样。这些派生类还可以选择重写基类的实现，在其自己的实现中调用基类的实现并加以扩展，或者干脆忽略基类提供的功能。

关于派生类如何使用基类的功能，基类也可以做出一些规定。基类可以把它的部分或全部方法声明为抽象方法。这就意味着派生类必须根据基类中定义的接口来提供其自己的实现。

继承大大增强了开发人员重用现有代码的功能。但是，确保不会过度应用继承这一概念也是非常重要的。继承树过深，可能会导致新的复杂度。如果分层结构中间层的某个逻辑单元必须加以修改，会导致整个代码变得脆弱、难以修改。在某些情况下，应当优选使

用“复合”而不是“继承”。例如，在 Employee 分层结构中已经插入了一个名为 PayableEmployee 的实体，如图 3-2 所示。

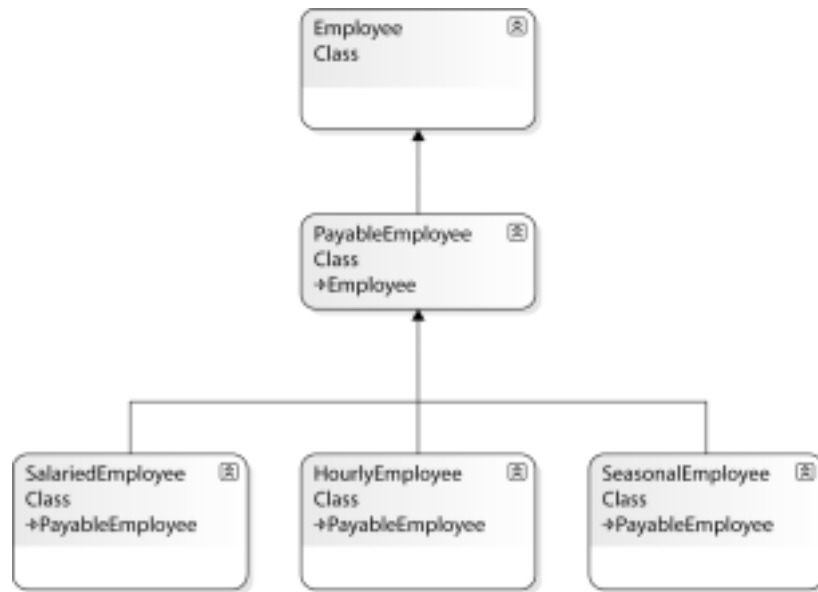


图 3-2

在这个层次结构中，计算员工报酬的逻辑被封装在 PayableEmployee 类中。但如果公司决定根据不同算法为每类员工支付报酬，又会发生什么情况呢？应当向类层次结构中引入更多类吗？这样可能会解决手边的问题，但如果不同类之间部分地共享了支付算法中的一些逻辑，又会如何呢？难道还要再添加一个继承层来应对这一问题吗？即使是像本例这样简单的类层次结构，也会很快变得过于复杂，让人感到绝望。

针对这一问题，较好的策略是使用复合。不是将为员工支付报酬的逻辑放在 Employee 层次结构的一个类中，而是为报酬计算模块定义一个名为 IEmployeePayrollService 的标准化接口。Employee 类中将包含一个受保护的成员变量，其中容纳一个实现 IEmployeePayrollService 接口的类，如图 3-3 所示。

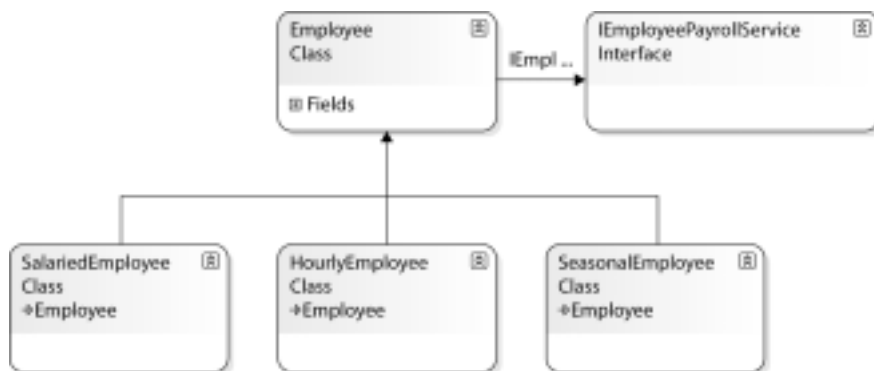


图 3-3

现在，每个员工都可以使用任意类型的类来计算他们所需要的报酬，只要它实现了这个标准化的员工报酬服务接口即可。

3. 多态性

多态性(Polymorphism)可能是 OOP 中最容易误解的原则了。“多态性”这一概念是指：尽管两个类可能共享某一行为，但它们可能采用完全不同的方式来实现这一行为。多态性多少与继承及它所生成的类之间的关系有关。

例如，在刚刚讨论的 Employee 示例中，Hourly、Salary 和 Seasonal 员工类都可以从 Employee 基类继承抽象方法 ComputePay(“抽象成员”是指其接口必须在基类中声明，但派生类必须自己提供其实现的成员)。ComputePay 方法可以调用 IEmployeePayrollService 接口的类实例的适当方法。

尽管对于所有 3 个派生类来说，这个方法的接口是相同的，但它们用于判断员工报酬的方法非常不同。按月计薪的员工可能有一个根据其月薪设定的数值。而按小时和按季度计薪的员工可能有“小时费率”，在计算报酬时会将这一费率与上个计薪周期内的工作小时数相乘。但在本例中，按小时计薪的工作有一个设定的工作时间，超过这一工作时间后会另加一半，而按季度计薪的工人就没有这一设置。每种员工都有 ComputePay 方法，每个方法都会返回该员工报酬支票的金额，但它们是使用不同方法来获取这一数字的。

如果考虑到：由 Employee 类派生的所有类都可以看作一个员工，而不考虑其具体实现方式，就能认识到多态性的真正威力了。在该例提到的公司中，到了发薪日，薪酬支付系统需要为所有员工发放支票。因为有了多态性，该系统只需要对一个员工列表进行迭代，并针对每位员工调用 ComputePay。它不需要知道是在为哪种员工计算支票金额；每个具体类型都拥有它所需要的具体实现方式。应用程序只要记着收集该方法返回的数值即可。

如果没有多态性，这个系统就必须做以下两件事情之一。第一件事是将所有员工分到三个组中(按月计薪、按小时计薪和按季计薪)，并以不同的方式处理每一组。另一件事是对所有员工进行迭代，并对每位员工做出判断，以确定如何为该员工付薪水。无论是哪件事情，都会需要更多的代码，这样不仅会使处理过程缓慢，还会使它更加复杂，引入更多的错误点。利用多态性可以让计算机完成这一工作，具体示例参见 3.4.5 小节。

例如，有一个如图 3-4 所示的动物层次结构：

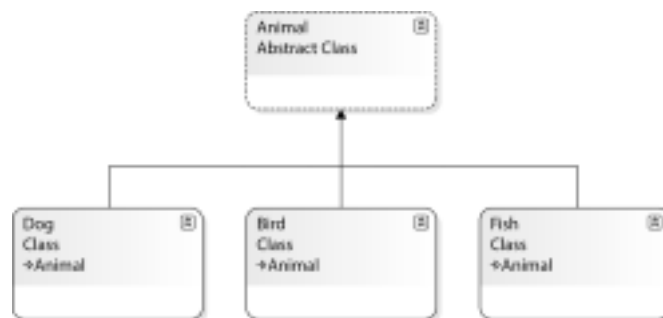


图 3-4

它们除了都是动物之外，还都能移动。但它们的移动方式不同；狗是跑、鸟是飞、鱼是游。因为它们都是从 `Animal` 继承而来的，所以可以将每一种子类型(狗、鸟和鱼)都称为动物。但如果希望让这些动物移动，那该怎么办呢？为 `Dog` 实现一个 `Run` 方法，为 `Bird` 实现一个 `Fly` 方法，为 `Fish` 实现一个 `Swim` 方法，这是最佳处理方式吗？这种做法可能看起来比较直观，但有非常严重的局限性。假定在代码中引用了某种类型的 `Animal`。怎样才能让它移动呢？要调用哪个方法呢？为了回答这个问题，必须首先判断正在处理哪种具体类型的 `Animal`。这种做法的效率很低，会使动物分层结构失去意义。

更好的方式是为 `Animal` 类创建一个抽象方法 `Move`。从该层次结构继承而来的每个动物都会为这一方法创建一种实现，提供自己特有的移动方法。在本例中，不需要知道所拥有的是哪种具体类型的 `Animal`，而是知道可以调用 `Move` 方法，这个动物会采取适当的移动方法。

3.2.2 SOLID 原则

在 21 世纪的前几年里，“Uncle Bob” Robert Martin 引入了用 OOP 开发软件的五条原则，其目的是设计出更易于维护的高质量系统。无论是设计新应用程序，还是重构现有基本代码，这些 SOLID 原则都成为开发人员的地图。

1. 单一职责原则

单一职责原则(Single Responsibility Principle, SRP)指出，每个方法或类应当有且仅有一个改变的理由。这意味着每个方法或类应当做一件事情，或者只有一项职责。在所有的 SOLID 原则中，这是大多数开发人员感到最能完全理解的一条。严格来说，这也可能是违反最频繁的一条原则了。

再来看一下 3.2.1 小节的 `Employee` 示例。3 个员工子类(`Hourly`、`Salary` 和 `Seasonal`)各有一个 `ComputeVacationTime` 方法。这看起来可能符合逻辑；一个员工工作并积累假期时间。这位员工去度假并享用假期时间。但考虑一下 `Employee` 类中的其他内容。员工的简历信息也存储在这些类中。现在可能会因两个已知原因来修改这些类：员工的简历信息的类型或结构发生变化；员工的假期时间计算方式发生变化。

根据单一职责原则，需要去除其中一项信息。在本例中，相信简历信息应当保留，因为是由这些信息来确定一位员工的。假期时间的计算应当被抽取到一个域服务中，因为目的是确定一位员工，而为员工计算假期时间的方法与此无关。最后得到一个 `Employee` 类，它现在只有一个改变理由，还有一个“假期时间计算服务”，它可能也只有一个改变理由。

2. 开放/封闭原则

开放/封闭原则(Open/Close Principle, OCP)是指软件(方法、类等)应当开放扩充且关闭修改。如果觉得它非常类似于继承的 OOP 原则，那就对了。它们之间的关系非常密切。事

实上，在.NET 中 OCP 就是依赖于继承的。

OCP 的要点在于：作为开发人员，别人偶尔会向我们提供基类，偶尔也会为其他开发人员生成基类框架，供其使用。这些使用者应当仅能使用这些基类，但不能对其进行修改。这一点是必要的，因为其他使用者也可能依赖于由基类提供的功能。如果允许使用者修改这些基类，可能会导致连锁反应，不仅会影响到应用程序中的各方面，还会影响到企业内的应用程序。还有一个问题，使用者有时可能会收到基类的升级版本。使用者在升级之前，必须找出一种方法用来处理其对该基类先前版本中所做的自定义。

于是，问题变为：“那么，如果我需要修改这个基类的工作方式，那应当怎么做呢？”OCP 的另一部分中给出这一答案；基类应当开放，可进行扩充。在这里，扩充是指创建一个由此基类继承而来的派生类，它可以扩充或重载基类功能，以提供使用者所需要的特定功能。这样，使用者就能使用类的修改版本，而不会影响到类的其他使用者。使用者还可以在将来更轻松地使用基类的升级版本，因为他们不用担心丢失自己的修改内容。

3. 里氏替换原则

继承对于 OCP 就相当于多态性对于里氏替换原则(Liskov Substitution Principle ,LSP)。LSP 规定：用超类代替应用程序中使用的对象时，应当不会破坏应用程序。这通常也被称为“契约式设计(design by contract)”。

回想前面的多态性示例，ComputePay 方法使用了 Employee 类型的列表，其中 Employee 就是基类型(超类型)。Salary、Hourly 和 Seasonal 类都是从 Employee 继承而来，因此它们是 Employee 的子类型。

根据 LSP，即使已经将列表声明为 Employee 的列表，也仍然可以用 Salary、Hourly 和 Seasonal 的具体实例来填充它。因为有了继承，它们都支持 Employee 声明的相同契约(公共的方法集或 API)。应用程序可以对该列表进行迭代，并调用那些在列表中各个项目的 Employee 上定义的方法，不需要知道或特别关心它们都是什么类型。如果它们支持契约，该调用就是合法的。

4. 接口分离原则

到目前为止，已经在示例中使用了基于类的继承，但还没有过多地讨论接口。回想一下，接口就是在代码中定义的契约，而类同意实现这一契约。这份协议要求类来为接口中定义的所有方法提供实现。至于如何实现方法，则由这个类来决定，只要它遵守契约，支持接口中的定义即可。接口是.NET 中功能非常强大的功能；它们对继承和多态的支持方式与类相同。

接口分离原则(Interface Segregation Principle ,ISP)规定，不应当强制客户端依赖于其不使用的接口。例如，银行系统可能有一个用于评估信用申请的服务。为便于讨论，假定该服务不仅处理有质押信用(车船贷款、抵押)，也处理无质押信用(信用卡、信用证、股票信用额度)。如果正在开发一个客户端，用于帮助从事汽车代理的金融专员为其客户获得汽

车贷款，则只需要关注汽车贷款的申请即可，无需考虑有关这一服务的任何其他事情。如果没有 ISP，应用程序可能必须了解其他方法。

尽管乍看起来这并没有什么，但它至少是增加了应用程序的复杂性，因为据以进行开发的 API 中会有许多方法，远远超出所需要的。这样可能会导致混淆，调用错误的方法还可能会导致潜在的错误。还有一种可能，API 中未被应用程序用到的部分可能会改变，而这又会导致对终端的改变。这样，因为没用到、没想用、甚至是根本就不关心的一些功能，而增加了应用程序的维护成本。这种情况还存在安全风险。该应用程序是专用于汽车贷款的。如果不道德的开发人员利用这个过于庞大的 API 来允许利用这一申请担保其他类型的信用，又该怎么办呢？这种问题的严重性就不仅仅是代码瘫痪、不可维护那么简单了。

这一问题的解决方案就是专门针对客户端的需要，为该服务创建几个更小的、更精细的接口。对于该示例应用程序，专门设计一个针对汽车贷款的接口是比较适当的做法。应用程序可以用同一实现访问同一个类，但这一次它使用了一个特定的接口，其中仅有实际服务的一部分方法。这样就降低了复杂性，将应用程序与 API 其他部分的修改隔离开来，还有助于堵塞安全漏洞。



提示：

在代码中应用 ISP 并不一定意味着服务就是绝对安全的。仍然需要采用良好的编码实践，以确保正确的验证与授权。

5. 依赖倒置原则

在完美世界里，应用程序的组件之间没有耦合关系或绑定关系。开发人员也能够改变自己希望改变的任何东西，而不需要担心在应用程序的其他地方出现缺陷，或者“不希望存在的负面影响”。令人悲伤的是，我们并不是生活在完美世界里。因此，组件需要相互绑定在一起，或者在某一点耦合，以构成实际应用程序。

依赖倒置原则(Dependency Inversion Principle, DIP)规定：代码应当取决于抽象概念，而不是具体实现；这些抽象不应当依赖于细节；而细节应当依赖于抽象。类可能依赖于其他类来执行其工作(Employee 服务可能依赖于数据访问组件向数据存储中保存和检索员工信息)。但是，它们不应当依赖于该类的特定具体实现，而应当是它的抽象。也就是说，Employee 服务不知道(或不关心)正在使用哪个具体的数据访问组件——只有它的抽象或代码契约(或接口)支持那些用于保存和检索员工所需要的方法。

显然，这一概念会大大提高系统的灵活性。如果类只关心它们用于支持特定契约而不是特定类型的组件，就可以快速而轻松地修改这些低级服务的功能，同时最大限度地降低对系统其余部分的影响。在第 6 章，还会看到如何利用这一概念来模拟这些依赖项，以进行测试。有时，需要向类中提供这一低级服务的具体实现，以便这个类能够完成自己的工作。最常见的做法，特别是在 .NET 中使用 TDD 的开发人员，就是依赖项注入(DI)模式。

依赖注入模式将在第 6 章详细介绍。

3.3 代码异味

再说一次，没有完美的代码。这是生活中的一个事实。但采用 TDD 的开发人员仍然尽最大努力使其代码达到最佳状态。有一个重要的技能有助于做到这一点，那就是在不用运行应用程序的情况下，能够评估代码，快速、轻松地查出常见的潜在问题点。这些常见问题被称为“代码异味(code smells)”。

3.3.1 什么是代码异味

在开发应用程序的这许多年里，开发人员总是需要解决代码中反复出现的一系列常见问题。针对这些问题，最终找到了一系列常见的、广为人们熟知的、得以广泛应用的解决方案。这些解决方案被称为“模式”。同样，这些年开发人员总是反复犯一些常见错误。这些错误，以及它们导致的问题，被称为“反模式”。“代码异味”就是一组人们熟知的、经常出现的代码反模式。本节介绍了一些较为常见的代码异味。3.4 节主要介绍纠正这些问题的步骤。

3.3.2 重复代码和相似类

考虑以下代码：

```
public class WidgetService
{
    private const double PricePerWidget = 1.5;

    public double GetQuoteForWidgets(int quantity)
    {
        return PricePerWidget*quantity;
    }

    public string PlaceOrderForWidgets(int quantity)
    {
        var invoice = new Invoice
        {
            TotalPrice = PricePerWidget*quantity
        };
        return invoice.InvoiceNumber;
    }
}
```

这个简单的 WidgetService 为特定数量的小饰品提供报价，并在客户端下订单时创建发货单。但是，该段代码有一个重要缺陷。如果查看 GetQuoteForWidgets 和 PlaceOrderForWidgets 方法，会看到用于确定饰品订单总金额的功能重复出现了($X = \text{PricePerWidget} \times \text{数量}$)。

这显然违反了 SRP :已经有人取得用于相同目的的逻辑,并在该应用程序中加以重复。假设引入了一个新的业务需求,需要改变饰品订单价格的计算方式。开发人员必须确保找到了代码中所有存在这一逻辑的位置,并对其进行更新,因此,必须对所有代码路径进行充分的递归测试。

这一代码异味并不只限于重复相同类中的码。例如:

```
public class WidgetService
{
    private const double PricePerWidget = 1.5;

    public string PlaceOrderForWidgets(int quantity)
    {
        var invoice = new Invoice
        {
            TotalPrice = PricePerWidget*quantity*1.15
        };
        return invoice.InvoiceNumber;
    }
}

public class DoDadService
{
    private const double PricePerDoDad = 2.25;

    public string PlaceOrderForDoDad(int quantity)
    {
        var invoice = new Invoice
        {
            TotalPrice = PricePerDoDad*quantity*1.15
        };
        return invoice.InvoiceNumber;
    }
}
```

这家商店扩大了生意,现在也销售一些小玩意。为了适应这一新的经营范围,开发团队已经创建了一个 DoDadService。查看 PlaceOrderForWidgets 和 PlaceOrderForDoDad 的方法,它们几乎一模一样。这表明又一次出现代码重复,这一次是跨类重复,将来可能会引发维护和质量问题。

3.3.3 大型类和大型方法

更大并不等于更好。例如:

```
public string PlaceOrderForWidgets(int quantity, string customerNumber)
{
    var invoice = new Invoice
    {
        InvoiceNumber = Guid.NewGuid().ToString(),
```

```
        TotalPrice = PricePerWidget*quantity,
        Quantity = quantity
    };

    var customer = _customerService.GetCustomer(customerNumber);
    invoice.CustomerName = customer.CustomerName;
    invoice.CustomerAddress = customer.CustomerAddress;
    invoice.CustomerBillingInformation = customer.CustomerBillingInformation;

    double tax;
    switch (invoice.CustomerAddress.State.ToUpper())
    {
        case "OH":
            tax = invoice.TotalPrice*.15;
            break;
        case "MI":
            tax = invoice.TotalPrice*.22;
            break;
        case "NV":
            tax = invoice.TotalPrice*.05;
            break;
        default:
            tax = 0.0;
            break;
    }

    var shippingPrice = invoice.TotalPrice * .1;
    invoice.TotalPrice += shippingPrice;
    invoice.TotalPrice += tax;

    var paymentAuthorizationCode = _paymentProcessingService.ProcessPayment(
        invoice.TotalPrice,
        customer.CustomerBillingInformation);
    invoice.Approved = ! string.IsNullOrEmpty(paymentAuthorizationCode);
    _invoiceService.Post(invoice);
    return invoice.InvoiceNumber;
}
```

这段代码有许多问题——最明显的就是它有多长。当我看到一段这样长的代码时，要做的第一件事就是在其中查找违反 SRP 的地方。在本例中，至少有 5 个不同的业务函数。该方法生成了一个发货单，将该发货单与一位客户相关联，根据客户所在的州确定税额确定发货成本并授权支付。这意味着至少有 5 种不同的理由修改这一方法。这是不能接受的。

方法较长的另一个常见问题是，它们可能很复杂，难以理解。任何富有经验的 ASP.NET 开发人员都可能编写过 Page_Load 方法，这一方法长至数百行，拥有复杂的逻辑分支，试图涵盖可能呈现该页的所有可能情况。我曾经见过一个超过 1400 行代码的方法。这还只是一个方法，而不是整个类！这些方法如此之大，迫使开发人员必须要在其中设置区域，以跟踪方法的各个部分。而区域是另一个代码异味，因为它们模糊了正在处理的代码，而且表

明类或方法太长，针对性不强。

这些长方法和大型类通常是应用程序中的主要麻烦来源。它们难以维护，也不太可能完全理解，它们的规模使它们成为孕育错误的温床。一定要使类和方法保持短小精干。与其在应用程序中使用 15 个长达数千行代码的大类，不如使用 100 个小类。越小越容易维护，越小越容易理解，越小越容易处理。

3.3.4 注释

有一种存在争议的观点(我认可这一观点)是：除非是编写设备驱动程序，或者其他某种“不再修改的”代码，否则代码中的注释也是一种代码异味。事实上，尽管大多数注释都是出于一片好心创建的，但它们最终并没有为代码添加任何价值。几乎在完成编写的那一刻，它们就过时了。它们通常是不准确的，当开发人员按照自己的经验加以理解，并利用这一知识对基本代码进行修改时，很可能会导致实质性损害。最后，注释只不过是一些行间干扰。人们希望利用注释达到的所有目的，都可以使用其他工具或技巧更好地完成。

有一些注释是为了帮助人们理解代码，对于此类注释，其解决方案很简单，只要提高代码本身的可理解性即可。不要用没有意义的名字和过于复杂的控制结构来模糊代码的意图。优秀的代码应当是整洁而易于理解的。注释成为判定不整洁代码的标志。

有些注释是用来跟踪谁修改了基本代码中的哪些内容，对于这一目的，源控制系统要比注释更得心应手。源控制系统，如 Team Foundation Server、Subversion 和 Git，可以跟踪开发人员向库中提交了什么内容，这一点要比开发人员留下注释要出色得多。开发人员经常会忘记进行注释。源控制系统从来不会忘记。在本例中，该工具已经在做这一工作，为什么还要开发人员来重复它呢？尤其是开发人员做的还比不上这些工具。

源控制系统还保留有代码的历史记录。许多开发人员注释掉大段当前未使用的代码，而不是直接加以删除，实际上他们很少会再用到这些代码。如果当前不用这些代码，直接删除即可。它们没有任何价值，完全可以删除它们。如果以后真的用到这些代码，可以很轻松地源控制系统中提取。

有时被称为“三斜线注释”的类或方法注释标题也不例外。许多开发人员是出于好心给出这些注释的，但结果与前面一样。这些注释很快就会变得不够准确和完整。阅读这些注释的开发人员会根据其表面意思加以理解，也就会根据这些错误信息做出判断，从而对基本代码产生极大的破坏。注释块还会产生大量的代码行干扰，而这些只会分散开发人员对代码的注意力。最后，如果方法和类非常简短、有较强的针对性，而且编写出色，则注释块是完全不必要的。

3.3.5 不当命名

清除代码注释的一个关键要素就是要消除变量、方法和类的不当命名。应用程序的一切内容都应当拥有一个有意义的、具有描述性的名字。

考虑下面的代码：

```
public double GetValue(int a, int b)
{
    var answer = (a*a)*b*P;
    return answer;
}
```

该方法是人们非常熟悉的一个数学公式。但它是什么呢？该方法没有提供任何线索，变量和常量 P 也没有提供任何帮助。如果必须猜一猜，那你会说什么呢？(答案在本章后面给出)。

这看起来是一个很小的细节，但代码中变量、方法或类的不当命名可能会极大地提高复杂程度。简而言之，描述性很差的名称只会使代码变得费解，许多时候，甚至让自己也难以理解。某个方法在现在看起来似乎很明了，但过一个月再来看看。是否还能想起参数 a 表示什么？如果自己都记不起来，那团队的其他人又怎么可能知道呢？3.4.3 小节中将更详细地讨论有关恰当命名的内容。

3.3.6 特征依赖

再来看看大型类、大型方法示例中的代码：

```
public string PlaceOrderForWidgets(int quantity, string customerNumber)
{
    var invoice = new Invoice
    {
        InvoiceNumber = Guid.NewGuid().ToString(),
        TotalPrice = PricePerWidget*quantity,
        Quantity = quantity
    };

    var customer = _customerService.GetCustomer(customerNumber);
    invoice.CustomerName = customer.CustomerName;
    invoice.CustomerAddress = customer.CustomerAddress;
    invoice.CustomerBillingInformation = customer.CustomerBillingInformation;

    if (customer.LoyaltyProgram == "Super Adamantium Deluxe")
    {
        invoice.TotalPrice = invoice.TotalPrice*.85;
    }

    double tax;
    switch (invoice.CustomerAddress.State.ToUpper())
    {
        case "OH":
            tax = invoice.TotalPrice*.15;
            break;
        case "MI":
            tax = invoice.TotalPrice*.22;
            break;
    }
}
```

```

        case "NV":
            tax = invoice.TotalPrice*.05;
            break;
        default:
            tax = 0.0;
            break;
    }

    var shippingPrice = invoice.TotalPrice * .1;
    invoice.TotalPrice += shippingPrice;
    invoice.TotalPrice += tax;

    var paymentAuthorizationCode = _paymentProcessingService.ProcessPayment(
        invoice.TotalPrice,
        customer.CustomerBillingInformation);
    invoice.Approved = ! string.IsNullOrEmpty(paymentAuthorizationCode);
    _invoiceService.Post(invoice);
    return invoice.InvoiceNumber;
}

```

除了前面曾提到的该方法的长度之外，这段代码还有另外一个问题：它几乎在每一行都使用了 Invoice 类的属性或方法。如果一个类过多地使用另一个类的方法，那就说它有些“特征依赖(feature envy)”。在该例中，PlaceOrderForWidgets 方法显然有些过于依赖 Invoice 方法。在这样的情况下，可以看到这一功能显然希望存在于其他某个地方，可能是在另一个类、方法或模块中。特征依赖所导致的主要问题就是代码非常脆弱。由于过于依赖于 Invoice 类，开发人员已经使这个方法成为它的附属。修改 Invoice 很可能也意味着修改该方法。要降低该方法对 Invoice 类的依赖性。3.4.2 小节将会介绍如何纠正这一问题。

3.3.7 If/Switch 过多

应用程序在本质上是要计算数据，并根据计算结果来执行某种功能。完成这一任务最常见的控制结构就是 If ...Then Else 结构和 Switch...Case 结构。在应用程序中大量使用这些结构当然不是什么稀奇的事情，但考虑下面这个方法：

```

public double CalculatePrice(int quantity, string customerState,
    string customerStatus)
{
    var basePrice = quantity*PricePerWidget;
    switch (customerState)
    {
        case "OH":
            if (quantity >= 1000 && quantity < 9999)
            {
                basePrice = basePrice*.95;
            }
            else if(quantity >= 10000)
            {

```

```

        basePrice = basePrice*.90;
    }
    break;
case "MI":
    switch (customerStatus)
    {
        case "Premier":
            basePrice = basePrice*.85;
            break;
        case "Preffered":
            basePrice = basePrice*.90;
            break;
        case "Standard":
            basePrice = basePrice*.95;
            break;
    }
    break;
default:
    if (quantity > 10000)
    {
        basePrice = basePrice*.95;
    }
    break;
}

return basePrice;
}

```

代码中大量的求值代码(If 和 Switch 代码块)在很大程度上增加了复杂性,降低了代码的可读性。即使这个示例中的简单代码暴露出一个缺陷,也可能很难轻松地诊断和纠正它。事实上,实际产品定价的算法可能要复杂得多。该方法还有另外一个问题,它违反了 SRP:该方法中包含 3 种不同的定价算法。通过限制方法中的 If 和 Switch 块的数量,可以降低这种复杂性和脆弱性。3.4.2 小节将会介绍如何修复这段代码。

3.3.8 Try/Catch 过多

处理异常是应用程序开发过程中非常重要的一项任务。我的开发团队有一些标准规则:永远都不应当让用户遇到未经处理的异常。但并不是说每行代码都应当放在 Try 代码块中——只有那些可能发生异常的代码块才应当放在里面。这些内容的详尽列表已经超出了本书的范围。应当放在 Try/Catch 代码块中的常见任务包括连接到一个数据库或与其交互、处理文件、调用 Web 服务。

与 If 和 Switch 代码块非常类似的是,将不必放在 Try/Catch 代码块中的东西移走也很容易:

```

public Customer GetCustomer(string customerId)
{

```



```

try
{
    var command = new SqlCommand();
    var reader = command.ExecuteReader();
    var customer = new Customer();
    while (reader.Read())
    {
        customer.CustomerId = customerId;
        customer.CustomerName = reader["CustomerName"].ToString();
        customer.CustomerStatus = reader["CustomerState"].ToString();
        customer.LoyaltyProgram = reader["CustomerLoyaltyProgram"].ToString();
    }
    return customer;
}
catch (Exception exception)
{
    _logger.LogException(exception);
    var customer = new Customer {CustomerStatus = "unknown"};
    return customer;
}
}

```

除了使用 ADO.NET 的最佳实践之外，可以看出，该方法也有与前面示例相同的一些问题。对于初学者来说，该方法不仅创建和执行了用于从数据库中检索客户的命令，还包含了一些代码，用于处理可能出现的异常。这样做虽然很巧妙，但在不同程度上违反了 SRP。

Try 代码块中的代码正在执行两件互相分离的任务：创建 ADO.NET 命令，从数据库中获取客户；然后将来自读取器的数据映射到客户对象。这看起来好像是一项任务，但它实际上代表两条改变理由。如果用于获取客户的存储过程改变其输入参数，用于建立此命令的代码也必须改变。如果从存储过程返回的客户数据集的数据结构发生了变化，或者是客户类本身的数据结构发生了变化，那就是另一个理由了。

Catch 代码块中还有另外一处违反 SRP 的地方。该方法包含了处理一种异常的代码，该异常可能是因为调用数据库而导致的。这是一个改变理由。对于在检索客户时所出现的异常，当前的处理算法是记录该异常，并返回一个“未知”状态的客户对象。明天，该过程可能会变得完全不同。这意味着出现另一个修改此方法的理由。

除了大量违反 SRP 之外，Try/Catch 代码块的过度膨胀也使该方法显得长了一些，引入了一些不必要的复杂度。根据 SRP，Try/Catch 代码块(调用一个工作单元，捕获被引发的异常)的结构足以作为其内部及本身进行修改的理由，不应当与业务逻辑或其他基础结构逻辑混合在一起。3.4.2 小节将纠正这一方法中的问题。

3.4 典型重构

复习一下，代码异味就是人们熟知的一系列编码反模式。因为它们为人们所熟知，而

且分布广泛,所以开发人员能够找到一些众所周知的常见方法来处理和修正这些代码异味。这些方法被称为“重构模式”。本章将重点介绍一些常见的重构模式,解释如何以及何时在代码中应用它们。

3.4.1 析取类或接口

希望将一个类分割为更小、针对性更强的类,或者从中析取出一系列更精细的接口,原因有很多。如果一个类非常大,则可能要做太多的事情。如果一个类违反了 SRP,则应考虑将其分割。如果由于设计或技术原因,类必须作为一个整体,或者类虽然很小但仍然在做太多的不同事情,析取接口是非常不错的选择,还是另外一个应当考虑从类中析取接口的好时机,就是当客户端只考虑该类的一部分公共接口时。考虑下面的类:

```
public class InvoiceService
{
    public string CreateInvoice(Invoice invoice) {...}

    public string ProcessPayment(Invoice invoice, double amount) {...}

    public double GetAmountOwed(Invoice invoice) {...}

    public double GetTotalAmountInvoicedLastFY(Customer customer) {...}

    public double GetTotalAmountPaidLastFY(Customer customer) {...}
}
```

尽管人们可能不认为这是个大型类,但它的确执行了几个相关但却互相分离的功能,供系统的不同部分使用。CreateInvoice 方法可能在客户下订单时供订单设置系统或电子商务网站使用。ProcessPayment 和 GetAmountOwed 可能供收款的账户或客户服务系统使用。GetTotalAmountInvoicedLastFY 和 GetTotalAmountPaidLastFY 可能供会计服务或年终报表服务使用。除了类的大小之外,该类的各个使用者显然不需要了解它们没有使用的方法。在本例中,通过析取该类的接口,对其进行重构是有意义的:

```
public interface IInvoiceCreatingService
{
    string CreateInvoice(Invoice invoice);
}

public interface IInvoicePaymentService
{
    string ProcessPayment(Invoice invoice, double amount);
    double GetAmountOwed(Invoice invoice);
}

public interface IInvoiceReportingService
{
    double GetTotalAmountInvoicedLastFY(Customer customer);
    double GetTotalAmountPaidLastFY(Customer customer);
}
```

```

public class InvoiceService : IInvoiceCreatingService, IInvoicePaymentService,
    IInvoiceReportingService
{
    public string CreateInvoice(Invoice invoice) {...}

    public string ProcessPayment(Invoice invoice, double amount) {...}

    public double GetAmountOwed(Invoice invoice) {...}

    public double GetTotalAmountInvoicedLastFY(Customer customer) {...}

    public double GetTotalAmountPaidLastFY(Customer customer) {...}
}

```

这一重构没有将任何功能移出类，但它的确改变了客户端使用该类的方式。有的具体类中有一些客户并不关心的方法，有时根本就不应当允许客户访问这些方法，对此，上面的这种重构给出了一个类的实例，它支持类所关心的接口。具体实现方式可以是任何样式的，这无关紧要。只要以接口方式支持代码契约，客户端就能像预期的那样使用该对象。

3.4.2 析取方法

现在再来看看 3.3.3 小节中的 PlaceOrderForWidgets 示例：

```

public string PlaceOrderForWidgets(int quantity, string customerNumber)
{
    var invoice = new Invoice
    {
        InvoiceNumber = Guid.NewGuid().ToString(),
        TotalPrice = PricePerWidget*quantity,
        Quantity = quantity
    };

    var customer = _customerService.GetCustomer(customerNumber);
    invoice.CustomerName = customer.CustomerName;
    invoice.CustomerAddress = customer.CustomerAddress;
    invoice.CustomerBillingInformation = customer.CustomerBillingInformation;

    double tax;
    switch (invoice.CustomerAddress.State.ToUpper())
    {
        case "OH":
            tax = invoice.TotalPrice*.15;
            break;
        case "MI":
            tax = invoice.TotalPrice*.22;
            break;
        case "NV":
            tax = invoice.TotalPrice*.05;
            break;
        default:
            tax = 0.0;
    }
}

```

```
        break;
    }

    var shippingPrice = invoice.TotalPrice * .1;
    invoice.TotalPrice += shippingPrice;
    invoice.TotalPrice += tax;

    var paymentAuthorizationCode = _paymentProcessingService.ProcessPayment(
        invoice.TotalPrice, customer.CustomerBillingInformation);
    invoice.Approved = ! string.IsNullOrEmpty(paymentAuthorizationCode);
    _invoiceService.Post(invoice);
    return invoice.InvoiceNumber;
}
```

前面曾经提到，该方法执行 5 项不同任务，因此违反了 SRP。通过识别和隔离方法中的特定任务，并将它们析取到另一个方法中，就可以纠正这一问题。

其中最明显的可被析取的功能就是计算订单中税负的代码(尽管它出现在该类的中间部分)。可以将这一代码析取出来，放到一个名为 `CalculateTaxForInvoice` 的方法中，再从 `PlaceOrderForWidgets` 方法中调用它：

```
public string PlaceOrderForWidgets(int quantity, string customerNumber)
{
    var invoice = new Invoice
    {
        InvoiceNumber = Guid.NewGuid().ToString(),
        TotalPrice = PricePerWidget*quantity,
        Quantity = quantity
    };

    var customer = _customerService.GetCustomer(customerNumber);
    invoice.CustomerName = customer.CustomerName;
    invoice.CustomerAddress = customer.CustomerAddress;
    invoice.CustomerBillingInformation = customer.CustomerBillingInformation;

    var tax = CalculateTaxForInvoice(invoice);

    var shippingPrice = invoice.TotalPrice * .1;
    invoice.TotalPrice += shippingPrice;
    invoice.TotalPrice += tax;

    var paymentAuthorizationCode = _paymentProcessingService.ProcessPayment(
        invoice.TotalPrice, customer.CustomerBillingInformation);
    invoice.Approved = ! string.IsNullOrEmpty(paymentAuthorizationCode);
    _invoiceService.Post(invoice);
    return invoice.InvoiceNumber;
}

private double CalculateTaxForInvoice(Invoice invoice)
{
    double tax;
    switch (invoice.CustomerAddress.State.ToUpper())
    {
```

```

        case "OH":
            tax = invoice.TotalPrice*.15;
            break;
        case "MI":
            tax = invoice.TotalPrice*.22;
            break;
        case "NV":
            tax = invoice.TotalPrice*.05;
            break;
        default:
            tax = 0.0;
            break;
    }
    return tax;
}

```

现在，该方法已经开始看着好得多了。它更简短、更容易理解。将用于计算税额的功能析取到 `CalculateTaxForInvoice` 方法中，就移除了 `PlaceOrderForWidgets` 代码的一项职责。税额仍然可以计算，只是在其他地方计算而已。通过特定的方法名称可以判断出在哪里进行这一计算，但不需要在每次查看该方法时都查看计算税额的代码——只在需要对税额计算方式做些什么时才查看。

另一项可以析取的功能就是创建并填充 `Invoice` 对象的代码。尽管发货单是下饰品订单的一个组成部分，但它是一项独立的分离任务(实际上是一项子任务)，应当被提取出来。换一种方式来考虑：用于下饰品订单的过程可能会改变，但不一定改变生成发货单的方式。与此类似，公司可能会在某个时间进入另一业务范围，需要修改 `Invoice` 类。订购饰品的过程没有变化(饰品订单中没有使用该新字段)，但 `PlaceOrderForWidgets` 方法中的代码可能需要改变，以适应 `Invoice` 中的变化。

在析取该方法之前，必须在方法中的其他地方进行修改。检查用于获取 `payment Authorization-Code` 的代码：

```

var paymentAuthorizationCode = _paymentProcessingService.ProcessPayment(
    invoice.TotalPrice, customer.CustomerBillingInformation);

```

这段生成订货单的代码需要一个 `Customer` 对象。在该方法中唯一用到 `Customer` 的其他位置就是该代码行。通过研究发现，该方法只需要 `CustomerBillingInformation` 来获取支付批准。而生成发货单的代码已经收集了这一信息：

```

invoice.CustomerBillingInformation = customer.CustomerBillingInformation;

```

在该方法的其他任何地方都没有修改这一数据。那么，还有必要再专门从这个客户对象获取它吗？在一些应用程序中，特别是正在运行多个线程时，可能确实如此。但对于该应用程序来说，并不一定如此。所以，第一步是修改调用 `ProcessPayment` 方法的代码：

```

var paymentAuthorizationCode = _paymentProcessingService.ProcessPayment(
    invoice.TotalPrice, invoice.CustomerBillingInformation);

```

完成这一修改之后，析取生成发货单的功能就是小事一桩了，从下面最终完成的方法中可以看出这一点：

```
public string PlaceOrderForWidgets(int quantity, string customerNumber)
{
    var invoice = GetInvoice(quantity, customerNumber);
    var tax = CalculateTaxForInvoice(invoice);

    var shippingPrice = invoice.TotalPrice * .1;
    invoice.TotalPrice += shippingPrice;
    invoice.TotalPrice += tax;

    var paymentAuthorizationCode = _paymentProcessingService.ProcessPayment(
        invoice.TotalPrice, invoice.CustomerBillingInformation);
    invoice.Approved = ! string.IsNullOrEmpty(paymentAuthorizationCode);
    _invoiceService.Post(invoice);
    return invoice.InvoiceNumber;
}

private Invoice GetInvoice(int quantity, string customerNumber)
{
    var invoice = new Invoice
    {
        InvoiceNumber = Guid.NewGuid().ToString(),
        TotalPrice = PricePerWidget*quantity,
        Quantity = quantity
    };

    var customer = _customerService.GetCustomer(customerNumber);
    invoice.CustomerName = customer.CustomerName;
    invoice.CustomerAddress = customer.CustomerAddress;
    invoice.CustomerBillingInformation = customer.CustomerBillingInformation;
    return invoice;
}
```

新方法 `GetInvoice` 绝对不是完美的(在实际应用中，会重新排列一些语句的顺序，使它变得更简短，更易于理解)，但与之相比已经迈出一大步了。现在把注意力转向 `PlaceOrderForWidgets` 方法，显然，仅仅析取了两个功能单元之后，该方法就大大提高了可读性、降低了复杂性，变得更易于维护。

在实践中会继续重构这一方法，析取交付价格计算、总价格计算、支付授权功能和批准功能。在析取这些方法之后，可能会出现更多可以析取的功能组。也可能会发现在从 `PlaceOrderForWidgets` 析取的方法仍然违反 SRP，所以还要从它们当中析取方法。

现在来看 3.3.8 小节中的示例：

```
public Customer GetCustomer(string customerId)
{
    try
    {
        var command = new SqlCommand();
        var reader = command.ExecuteReader();
```

```

        var customer = new Customer();
        while (reader.Read())
        {
            customer.CustomerId = customerId;
            customer.CustomerName = reader["CustomerName"].ToString();
            customer.CustomerStatus = reader["CustomerState"].ToString();
            customer.LoyaltyProgram = reader["CustomerLoyaltyProgram"].ToString();
        }
        return customer;
    }
    catch (Exception exception)
    {
        _logger.LogException(exception);
        var customer = new Customer {CustomerStatus = "unknown"};
        return customer;
    }
}

```

前面曾经提到，在 Try/Catch 代码块中放入业务代码违反了 SRP。将每个代码块中的这些方法析取到外部方法可以使该方法变得更简单，更易于理解：

```

public Customer GetCustomer(string customerId)
{
    try
    {
        return GetCustomerFromDataStore(customerId);
    }
    catch (Exception exception)
    {
        return HandleDataStoreExceptionWhenRetrievingCustomer(exception);
    }
}

private static Customer GetCustomerFromDataStore(string customerId)
{
    var command = new SqlCommand();
    var reader = command.ExecuteReader();
    var customer = new Customer();
    while (reader.Read())
    {
        customer.CustomerId = customerId;
        customer.CustomerName = reader["CustomerName"].ToString();
        customer.CustomerStatus = reader["CustomerState"].ToString();
        customer.LoyaltyProgram = reader["CustomerLoyaltyProgram"].ToString();
    }
    return customer;
}

private Customer
    HandleDataStoreExceptionWhenRetrievingCustomer(Exception exception)

```

```
{
    _logger.LogException(exception);
    var customer = new Customer { CustomerStatus = "unknown" };
    return customer;
}
```

和前面的示例类似，也可以在这里做一些方法析取工作。一定要记住：方法析取是一个迭代过程。应当不断查看违反 SRP 的功能和应当独立完成任务的功能。

3.4.3 重命名变量、字段、方法和类

下面来回忆一下 3.3.5 小节中的代码示例：

```
public double GetValue(int a, int b)
{
    var answer = (a*a)*b*P;
    return answer;
}
```

这是前几页给出的一个例子，还能想起它是做什么的吗？

现在再来看看同一方法，但它的变量都有非常合适的名称，方法名称的描述性也很强：

```
public double GetVolumeOfACylinder(int radius, int height)
{
    var volumeOfACylinder = (radius*radius)*height*Pi;
    return volumeOfACylinder;
}
```

现在很清楚该方法是做什么的了。即使它没有一个描述性的方法名，也没有变量名称 `volumeOfACylinder`，也可以从执行这一计算的代码中推断出：该代码用来判断一个圆柱体的体积。这是一个简单的示例，但只要稍微关注一下命名问题，就可以使代码清晰得多，也更容易理解。

名称应当是清楚的、经过深思熟虑的，并且要使用大家熟悉的日常语言。不要试图使用缩写，除非它们在你的公司、行业或软件开发中为人熟知。不要害怕使用长名称。描述性很强的长名称比不能提示其意图的短名称要好得多。Visual Studio 和大多数现代开发环境一样，提供了一些自动完成的形式。尽可以使用它。不要让“这个名称太长”成为不当命名的借口。

3.4.4 封装字段

类使用成员变量来跟踪和维护其内部状态。根据封装规则，这些成员变量的作用域被设置为私有，不能供外部客户端直接使用。这是一种保护性措施，用于确保类的运行时状态不受破坏。

有时，客户端需要访问类的一些字段。例如，在这里给出的 `Widget` 类，模型成员显然

是该类的客户端需要能够访问的：

```
public class Widget
{
    private object _internalWidgetState;
    public string _widgetModelNumber;
}
```

表面上来看，除了命名问题之外，这似乎是一个非常合理的代码。但是，是什么来阻止客户端改变 `_widgetModelNumber` 的值呢？它被声明为公共的，这意味着任意外部对象都可以改变成员变量，无法阻止。

处理此类情景的一种较好方法是将成员变量封装在一个字段中(有时被称为一个属性或“getter 和 setter”)：

```
public class Widget
{
    private object _internalWidgetState;
    public string ModelNumber { get; private set; }
}
```

对于初学者来说，`ModelNumber` 名称要更好一些，比 `_widgetModelNumber` 更符合命名标准。但一个字段的真正能力在于：尽管仍然可以从外部访问 `ModelNumber`，但它是只读状态的。该集合的作用域被设置为私有的，这意味着只有 `Widget` 对象本身可以设置其取值。

3.4.5 用多态替换条件

在下面的示例中，需要编写软件来向不同位置的各种不同交通工具派遣加油车。为了知道要发送多少燃料，需要知道每个交通工具的油箱中目前有多少燃料。该应用程序支持 3 种类型的交通工具，如下面的各个类所示：

```
public class Car : Vehicle
{
    public int AmountOfFuelNeededToFillTank()
    {
        return 12;
    }
}

public class Airplane : Vehicle
{
    public int AmountOfFuelNeededToFillLeftFuelTank()
    {
        return 3;
    }

    public int AmountOfFuelNeededToFillRightFuelTank()
    {
        return 4;
    }
}
```

```
    }  
}  
  
public class Boat : Vehicle  
{  
    public int AmountOfFuelNeededToFillFrontFuelTank()  
    {  
        return 8;  
    }  
  
    public int AmountOfFuelNeededToFillRearFuelTank()  
    {  
        return 10;  
    }  
}
```

该应用程序必须检查加油车路线上的所有车辆以确定要在该加油车内装多少燃料。完成这一任务的代码如下：

```
public class FuelingStation  
{  
    public int AddFuelToTruck(List < Vehicle > vehiclesOnRoute)  
    {  
        var amountOfFuelToLoadOnTruck = 0;  
        foreach (var vehicle in vehiclesOnRoute)  
        {  
            if (vehicle is Car)  
            {  
                amountOfFuelToLoadOnTruck += ((Car) vehicle).  
                    AmountOfFuelNeededToFillTank();  
            }  
            else if(vehicle is Airplane)  
            {  
                amountOfFuelToLoadOnTruck +=((Airplane) vehicle).  
                    AmountOfFuelNeededToFillLeft FuelTank();  
                amountOfFuelToLoadOnTruck +=((Airplane) vehicle).  
                    AmountOfFuelNeededToFillRight FuelTank();  
            }  
            else if(vehicle is Boat)  
            {  
                amountOfFuelToLoadOnTruck +=((Boat) vehicle).  
                    AmountOfFuelNeededToFillFront FuelTank();  
                amountOfFuelToLoadOnTruck +=((Boat) vehicle).  
                    AmountOfFuelNeededToFillRear FuelTank();  
            }  
        }  
        return amountOfFuelToLoadOnTruck;  
    }  
}
```

加油车路线上的每种交通工具用 vehiclesOnRoute 列表中的一种交通工具表示。为了判

断要发送多少燃料，这一代码必须对整个列表进行迭代，将填充每个交通工具的油箱所需要的空间加在一起。对于 Car 来说，这是相当简单的；一辆轿车(通常)只有一个油箱。飞机有两个油箱(每个机翼上一个)，它们能够以不同速度消耗油料。船艇可以有前后油箱，也可以存放不同数量的燃料。因此，用于获取每种交通工具所需燃料数量的逻辑是不同的。

AddFuelToTruck 方法中的代码尝试通过以下方式来处理这一情况：在每次循环迭代中，判断正在查看的是哪种具体类型的交通工具，然后将 vehicle 转换为该特定类型，以便能够调用这种类型的专用方法，从而获取燃料信息。

这种转换不仅效率很低，而且 AddFuelToTruck 方法还必须知道有关每种交通工具类型的特定细节，而它实际上并不应当知道或不应当关注这些信息。如果添加了一种新的交通工具类型 或者现有交通工具类型发生了变化 那这种方法也必须改变。这又违反了 SRP。

可以使用多态性对这一方法进行重构，以避免再用到有关交通工具类型的隐密信息。首先，创建一个名为 IFuelable 的接口：

```
public interface IFuelable
{
    int GetTotalAmountOfFuelNeededForVehicle();
}
```

因为已经应用了良好的命名标准，所以 IFuelable 接口有且仅有一个方法 GetTotalAmountOfFuelNeededForVehicle 返回了填满交通工具的油箱所需要的总燃料数量。下一步是针对 Vehicle 类实现该接口：

```
public class Car : Vehicle, IFuelable
{
    public int GetTotalAmountOfFuelNeededForVehicle()
    {
        return AmountOfFuelNeededToFillTank();
    }
}

public class Airplane : Vehicle, IFuelable
{
    public int GetTotalAmountOfFuelNeededForVehicle()
    {
        return AmountOfFuelNeededToFillLeftFuelTank() +
            AmountOfFuelNeededToFillRightFuelTank();
    }
}

public class Boat : Vehicle, IFuelable
{
    public int GetTotalAmountOfFuelNeededForVehicle()
    {
        return AmountOfFuelNeededToFillFrontFuelTank() +
            AmountOfFuelNeededToFillRearFuelTank();
    }
}
```

为清晰起见，从该代码清单中删除了 `AmountOfFuelNeeded...` 方法，但它们仍然存在于代码中。由于向声明中添加了 `IFuelable` 接口，所以必须向每个交通工具类中添加 `GetTotalAmountOfFuelNeededForVehicle`。这样，开发人员就能封装用于计算燃料数量的算法，这一数量就是完全填满每种交通工具类型所需要的数量。在重构 `AddFuelToTruck` 方法以利用这一接口时，它变得更简短、更高效、更易于读取：

```
public int AddFuelToTruck(List < IFuelable > vehiclesOnRoute)
{
    var amountOfFuelToLoadOnTruck = 0;
    foreach (var vehicle in vehiclesOnRoute)
    {
        amountOfFuelToLoadOnTruck +=
            vehicle.GetTotalAmountOfFuelNeededForVehicle();
    }
    return amountOfFuelToLoadOnTruck;
}
```

此外，`AddFuelToTruck` 方法也不再需要知道每种交通工具所需要的总燃料数量，它只需要关注要装载的燃料数量总和即可。

利用 .NET 中内置的 LINQ 功能，可以将它重构为更小一些的方法：

```
public int AddFuelToTruck(List <IFuelable> vehiclesOnRoute)
{
    return vehiclesOnRoute
        .Sum(vehicle => vehicle.GetTotalAmountOfFuelNeededForVehicle());
}
```

由于使用了多态性，所以这个方法现在可以利用 `List` 对象的 `Sum` 方法，从而更简短，更易于理解。

3.4.6 允许类型推断

.NET 平台上的大多数语言都是静态类型化的。这意味着开发人员必须将每个变量和方法声明为某一特定类型。C# 3 向 .NET 引入了类型推断的概念。也就是说，尽管 C# 仍然是一种静态类型化的语言，但开发人员可以允许编译器根据一个变量的初始化方式来判断它应当是什么类型。再说清楚一点，这种类型推断不会在运行时进行；如果编译器不能在编译应用程序时判断类型，会生成一条错误。

C# 中的类型推断是使用 `var` 关键字完成的。不是将变量声明为某一特定类型，而是将其声明为 `var`：

```
Customer declaredAsCustomer = new Customer(); //declared with a specific type
var typeInferredCustomer = new Customer(); //declared using the var keyword
```

`declaredAsCustomer` 和 `typeInferredCustomer` 都是 `Customer` 类的实例，它们的行为方式与使用方式都完全相同。两者之间的区别就是不需要专门向编译器说明 `typeInferredCustomer`

是一个 Customer 对象。

var 关键字的好处在于它生成的代码更容易读懂；一些开发人员将简单的变量声明看成代码行干扰。var 关键字将干扰降至最低。在将变量声明为 var 时，更容易进行重构，因为编译器可以判断应当将变量声明为什么类型。如果开发人员改变一个变量的初始化方式，编译器可以自动修正其变量声明。

var 关键字的使用是有争议的。一些开发人员把这种做法看成一种不良实践方式，因为这样就难以判断正在处理哪种类型。这一点可能是对的，但需要指出的是，只有在方法过长和违反 SRP 的情况下它才会成为问题。如果开发人员能够保持其他代码的整洁性，var 关键字的使用不会成为负担。

3.5 本章小结

开发人员永远不可能在第一次就编写出完美的代码。在实践中，如果试图在第一次就编写出“完美方法”，那绝对是一个错误。只需要尽力使应用程序正常工作，能够通过测试就可以了。有了用于验证代码的一组单元测试，就可以放心地进行重构。

重构之于代码，就相当于编辑之于书籍或文章。不要对自己的代码心存慈悲。重构可以提高其可维护性和可测试性，还能确保它遵循了 3 条 OOP 原则：封装、继承和多态性。重构可以确保遵守了 SOLID 原则：单一职责原则、开放/封闭原则、里氏替换原则、接口分离和依赖项注入。所有这一切，不仅能够提高代码的质量，还能够锻炼开发技巧，使你成为更好的 TDD 实践者。

代码异味是代码中的反模式。要学会识别它们。让自己熟悉重构模式，便能够快速、轻松地处理这些代码异味。一定要使用单元测试来验证在重构时没有破坏代码的业务功能。

最后，继续研究 OOP 原则和 SOLID 原则。这些概念不是最终目标，而是过程。我们总有机会来改进编写代码的方式。不要停止学习！

第 4 章

测试驱动开发：以测试为指南

本章内容

- TDD 开发人员如何让测试推动自己开发的代码
- 从需求到工作代码的 TDD 工作流
- 在 TDD 中为一项功能编写代码的 3 个阶段
- 好的测试如何保证代码的质量
- 如何定义“完成”这一概念

第 3 章曾经提到，在 TDD 的各个方面中，开发人员最容易纠结的是应当用测试来推动最初开发过程这一理念。很多年以来，开发人员都把测试看作是在完成代码、做好交付准备之后，用于对代码进行验证的方法。使用测试的工作流是：取得一组需求，将这些需求转换为工作代码，然后使用测试验证这些代码。

这种范例的问题在于将测试延迟到整个过程的最后。即使最小的应用程序也有一些缺陷。没有测试，就无法快速、轻松地判断缺陷在代码中的出现位置。而且针对已有代码设计和编写单元测试也会变得困难，因为在编写代码时，可测试性可能并没有被放在优先考虑的位置。重构也很难，因为没有一种快速、轻松的方法来验证重构后的代码仍然像预期的那样工作。

解决这一问题的最简单方法是把整个过程前后颠倒过来。本章将会介绍如何在编写第一行代码之前，就根据应用程序的业务要求编写测试，从而确保所编写的全部代码都会由一个测试来检验。采用测试先行的开发过程还有助于确保不会编写多余的代码，而只编写

应用程序为执行其指定任务所需要的代码；不用再浪费力气来维护那些对于应用程序没有建设性贡献的代码。

TDD 中的 workflow 经常被描述为“红灯、绿灯、重构”。这一说法描述了 TDD 开发的一系列步骤：首先以一个未能通过的测试开始，随后开发人员仅编写足以通过该测试的代码，然后再努力改进此代码。4.2 节将会介绍这些步骤的重要性，以及如何通过这些步骤得到一个能够满足业务需要、缺陷较少的可信赖高效代码。

4.1 从测试开始

TDD 中的第一个 D 是“驱动(driven)”的意思。这意味着在 TDD 实践中，必须以测试来驱动开发。这看起来似乎是一个非常简单的概念。但当刚接触 TDD 的开发人员开始尝试以测试驱动方式工作时，很容易再走上原来已经非常习惯的老路子。

大多数传统的开发工作流都是从收集需求开始。业务经理坐下来，通常会和项目经理和体系结构设计师一起粗线条地勾画出一个系统。这个过程会经过几次反复，以改进设计和增加细节。最后得到一组技术规范，用于创建新应用程序或修改现有应用程序。

对于那些没有使用 TDD 的开发团队或开发人员来说，下一步就是编写代码了。作为开发人员，会使用这些由业务需求得到的规范来定义实体、服务类和应用程序工作流程。这时，系统的质量取决于开发人员对这些书面技术规范和业务需求的理解。在完成代码编写之后(或者达到一种可以认为是“完成”的程度之后)，就可以将应用程序发送给 QA 部门进行测试。而 QA 部门总是可以找出功能中的一些缺陷。

现在该由开发人员来纠正这些缺陷了。如果没有单元测试来帮助查找和诊断缺陷时，大多数开发人员会采用调试器，在他们认为代码中可能出现缺陷的位置设置断点。有时将这种做法称为“霰弹枪方法”。这种方法的针对性很差，效率也比较低。如果应用程序有许多服务层和抽象层，可能需要花费大量时间来查遍所有各层，以找到问题的来源。最终的结果是，查找缺陷所用的时间比实际修改这些代码所需要的时间还要多。这不是一种高效利用开发时间的工作方式。

雪上加霜的是，如果没有事先考虑可测试性这一概念，那在为代码编写单元测试会非常困难。TDD 强制我们来考虑如何设计和构建软件，才能使其更灵活，降低其耦合程度。许多没有使用 TDD 的开发人员认为自己正在编写松散耦合的灵活软件。但如果没有外界因素来让开发人员保持诚实(在本例中，这种外界因素就是单元测试)，开发人员倾向于抄近路。这很自然，因为开发人员多年来一直接受的训练就是“优化、优化、再优化！”；没人教给他们要考虑可维护性。

最终，这一过程导致仅修改一个微小缺陷也要花费无数的时间。如果有一些现成的方式来确定缺陷来自哪里，就可以节省大把时间。如果更好一些，能有一些自动方式来确保这些代码反映了业务需求和技术规范，缺陷可能根本就不会出现。

采用 TDD 的开发人员仍然是从一组业务需求开始。TDD 与传统开发方式的分歧是在

下一步。在为某一功能编写任何代码(包括创建一个包含该代码的新类)之前，应先根据当前正在处理的需求来编写一个单元测试。

单元测试是从业务需求和技术规范得出的。因此，单元测试以鲜活的可执行方式代表这些需求和规范。它们不再是文档中很快就会被开发人员忘掉的内容。它们成为可以运行的代码，可以用来证明正在创建该业务所需要的功能。这些需求变成了开发人员能够与之交互的有形东西。

创建代表需求和规范的单元测试之后，就可以开始编写代码了。这里的目标是编写一些最简单、最少量的代码，以通过该测试。如果代码看起来不够完善或没有完成，不要担心；如果它能通过当前的单元测试(该单元测试是从需求得出的)，那它就是完整的。

由此推知，不要编写还没有为其准备单元测试的代码。不要费心编写一些代码来满足“可能”的或甚至根本不会出现的需求；只有将需求列于文档中并为其生成测试之后，这些需求才是真实存在的。只编写测试需要的代码即可。

在针对需求创建功能时，可能会发现应用程序中其他一些需要设计和建立的部分。这种情况在采用敏捷方法时尤为常见。在大多数此类方法中，可以定义高级别的体系结构，但没有进行大型的、详细的先期设计阶段。详细设计一直被推迟到将一项功能或要求实际排入日程。这样可以避免开发团队为一些永远不会实际开发的功能和特性进行详细设计。

新的子系统和其他功能部分出现时，也要确保为这些代码创建测试。例如，可能接到任务，设计一项“根据 Person ID 析取 Person 对象”的功能。要实现该测试，可以首先创建一个 Person 业务域服务，可以取名为 PersonService。该服务表示一种机制，应用程序的组成部分可以通过这一机制根据 Person ID 找到一个人。PersonService 的目的是根据一组业务规则提供一个 Person 对象。但是，根据 SRP，它不能实际负责从数据库中检索 Person 数据。PersonService 的一个修改理由是关于如何检索人员的业务规则(例如，不按 Person ID 检索人员，而是根据数据库表中的其他字段来检索)。PersonService 的另一个修改理由可能是如何实际从数据存储检索数据(例如，Person 数据最初存储在数据库中，但在某一时间，它可能被重新放置在一个 Web 服务背后的数据存储中)。

为了满足对分离要求的关注，可以创建一个 PersonRepository 类，它负责从数据库中获取表示人员的数据。这个新类只有一个修改理由，就是从数据库中检索人员的方式是否发生变化。

有关创建人员信息的业务规则存在于 PersonService 中，所以 PersonRepository 只需要关注数据访问。

这个新类代表一个需要编写的新测试。记住，单元测试与其正在测试的具体类和方法相隔离。这意味着针对 PersonService 的测试不应当负责验证 PersonRepository 的功能。事实上，单元测试所使用的 PersonService 具体实例不应当使用 PersonRepository 专用的具体实例。它应当使用一个模拟对象，该模拟对象实现与 PersonRepository 一样的接口(有关“模拟”的内容在第 5 章详细介绍)。

这意味着除了针对 PersonService 的单元测试之外，还必须编写针对 PersonRepository

的单元测试。由于现在只处理 `PersonService`，所以只需要为 `PersonRepository` 定义接口，以及根据 `Person ID` 从数据存储中获取人员信息的方法。由于要为 `PersonService` 使用 `PersonRepository` 的一个模拟，因此在尝试使针对 `PersonService` 的测试通过时，并不需要针对数据库的专有功能。

在针对 `PersonService` 的测试通过之后，应当将注意力转到 `PersonRepository`。要求是：`PersonRepository` 应当能够根据 `Person ID` 从数据库中获取人员数据。任务是仅编写能够根据给定 `Person ID` 返回相关人员数据的代码。记住，针对资料库类的单元测试就像针对任意其他类的单元测试一样，需要能够将它们与具体的依赖项隔离。大多数数据库持久框架都提供了一种模拟数据上下文的机制，允许为资料库编写隔离的单元测试。有关具体细节参阅所用框架的文档。

在完成针对 `PersonService` 和 `PersonRepository` 的单元测试之后，下一步就是集成它们。在本例中，希望测试能够突破界限；该测试应当能够调用 `PersonService` 具体实例的方法，`PersonService` 反过来又能调用 `PersonRepository` 具体实例的方法，而后者又能访问其中存储有个人数据的数据存储。编写整体测试的要点在于：确保 `PersonService` 和 `PersonRepository` 能够一同工作。经常会出现这样一种情况：有两个分别创建的组件，最终发现它们不能互相支持。尽早将应用程序的较小部分集成，就可以确保当前正在创建的组件能够在以后合并到较大的应用程序中。

4.2 红灯、绿灯、重构

TDD 有许多习惯用语。前面已经介绍了 OOP、SOLID 和 DRY。这 3 个不仅对 TDD 非常重要，对于一般的良好软件开发实践也很重要。但如果要用一个词组或咒语来总结 TDD 的核心信念，那就是“红灯、绿灯、重构”。它会提醒你想起 TDD 的工作流程：从需求转到测试，然后再转到代码。它还设置了可能会进行重构的预期。不熟悉 TDD 的开发人员和管理员经常会听到“重构”一词，并认为它的含义就是修正那些在开始时未能正确完成的内容。这种对重构的认识简单但不真实。编写代码是一个迭代过程。重构是对代码的持续提炼，尽量使其变得最简单、可读性最强，达到最佳状态。

4.2.1 TDD 的 3 个阶段

“红灯、绿灯、重构”明确了开发人员在实施 TDD 时所要遵循的工作流。前面曾经说明，在编写代码时的步骤顺序非常重要。遵循这一步骤顺序(需求、测试、代码)可以确保正在编写的代码都有相应的测试，并能确保只会编写已经拥有针对性测试的代码。

“镀金”一词是指向应用程序中添加用户或企业并没有请求的一些功能，而这些功能仍然需要花钱来开发和维护。所添加的功能可能看起来是个好主意。但这些不需要和用不到的功能需要耗费资源来创建和维护，这是一种浪费。定义系统功能与特性的应当是需求，而不是开发人员的一时兴起。

4.2.2 “红灯”阶段

在开始使用 TDD 时，许多开发人员都会问：“我怎么能为不存在的代码编写测试呢？”事实上，许多测试都是针对当前并不存在的类或方法的。这意味着这些测试甚至不能编译通过，其效果基本上与失败测试相同。这没问题；记住，正是因为有了这些测试，才需要有相应的代码存在。

创建了第一个测试后，下一步就是采取一个或多个支持操作。在应用程序开发的早期，下面要做的事情就是在 Visual Studio 解决方案中创建一个项目。如果真是这样做的，那就需要从测试项目中添加对新项目的引用。

可能已经有了这个项目，并从测试项目中引用了它。下一步可能就是为单元测试所需要的服务创建一个新的接口。测试不会编译成功，所以在创建该接口之前无法通过这一测试。此时单元测试可能仍然无法编译成功；在没有服务的具体实现时，会得到一个编译错误。因此，需要为这个新服务创建一个具体实现。

或者，接口和服务都已经存在，但对于正在编写的测试，还没有方法可用。通过首先编写这个测试，并成为该方法的第一个使用者，可以在一定程度上了解开发人员如何处理这一服务。如果不采用 TDD，开发人员通常会用自己方便的方式为这些服务创建公共接口，根本不会考虑将会如何使用这些接口。尽管在编写这些服务时可以求助于 API，但该服务的使用者可能难以理解和使用它们。使用自己的接口编写测试可以体验一下服务的使用者如何使用此服务，这有助于定义一些简单的接口，便于最终用户的使用和理解。

最后一个明显的改变是：我们已经有接口和类，要测试的方法也已经存在，但正在测试针对该方法的一项新需求。该测试还是会失败，因为还没有编写任何能使这一测试得以通过的代码。在这种情况下，下一步就是编写代码来使新测试通过，同时不会导致已有测试失败。

在任何情况下，第一次编写测试时，无论出于什么原因它都是会失败的。可能是未能编译通过；也可能是在测试编译成功后，由于未能从被测方法中获取期望结果而失败。那现在就处于“红灯”阶段。现在的目标是进入“绿灯”阶段。

4.2.3 “绿灯”阶段

如果测试失败，那就是处于“红灯”阶段。到达“绿灯”阶段的关键在于仅编写适量的代码，使新测试通过而不会导致任何其他测试失败。“仅编写适量的”代码以通过测试，这是新接触 TDD 的开发人员有时需要努力克服的一个哲学障碍。

例如，假定有一项需求，需要向现有服务中添加一个新方法。该需求的其余部分规定：当给定输入值 61 时，新方法应当返回 False。许多没有用过 TDD 的开发人员都认为：为通过该测试，至少要编写以下数量的代码：

```
public bool MyMethod(int inputParameter)
{
```

```
        if (inputParameter == 61)
        {
            return false;
        }
        return true;
    }
}
```

有理由假定：这就是该方法要做的工作，但这并不是使该测试通过的最少代码。对于该方法在什么条件下返回 True，前面的要求和测试并没有给出任何规定；这一代码应当仅说明：当 inputParameter 等于 61 时，它应当返回 False。这里给出的代码事实上像是这个方法的最终版本。但在创建测试以验证返回 true 值的条件之前，不应当做任何假定。

下面的代码是为了通过该测试所需要的最少代码：

```
public bool MyMethod(int inputParameter)
{
    return false;
}
```

该实现看起来似乎不完整。它看起来当然还没有完成；该方法要求为 inputParameter 输入值，但却没有使用它。该练习的要点在于确保只编写能够使该测试通过的“适量”代码，从而满足业务需求。如果文档中目前列出的唯一需求就是该方法在 inputParameter 值为 false 时返回 false，那上面的例子刚好满足这一业务需求。在为这一特性或功能充实更多的需求时，将会创建更多的测试。增加的这些测试会让我们扩展该方法的功能。随着这一方法的发展，比较适当的做法可能是向 if 语句添加更多分支。另一方面，也可能需要更复杂的算法。关键在于，不要向代码中引入不必要的复杂性。

在编写了足够的代码，使测试通过之后，就可以进入重构阶段了。

4.2.4 “重构”阶段

一直到现在，目标仅限于使单元测试能够通过。注意，到目前为止还没有考虑可维护性、可读性或整体代码质量。已经创建了一些验证业务要求的单元测试，现在应当转移注意力，使代码具备这三种特性了。这一实践步骤称为“重构”。

无论是否使用 TDD，开发人员都会发现需要调整和修补其代码。这基本上也就是重构代码时所做的。用 TDD 进行重构的好处在于，有一套单元测试可以用来验证代码仍然满足业务需求。定期检查业务代码和单元测试，并寻求通过重构强化代码的机会。进行一些微小更改来重构代码，然后使用单元测试来验证重构后的代码仍然能够正常工作。只要所有测试都能通过，代码就是正确的。

4.2.5 重新开始

现在已经有有了一个能够通过的单元测试，也有一个方法，其中包含了使该测试通过的最简短代码。还有机会重构该代码，以确保它是可读的、可维护的。下一步就是再来一次。

如果回想 4.2.3 小节的代码示例，满足该测试的最少量代码就是仅返回 `false`。尽管这样足以使测试通过，并满足当前测试要求，但从业务的角度来看它可能是不完整的。随着开发过程的继续，会有更多的需求和功能被排入开发日程。这些要求和功能要转换为新的测试。这些测试的完整集合(测试套件)反映了希望该应用程序具备的所有业务功能。这样就创建了一组测试，只要使所有这些测试能够通过，就能确保应用程序满足业务需要。

在使用 TDD 时，非常重要的一点是要记住：所编写的代码及其生成的测试结果，只能达到与测试一致的水平。前面讨论了重构代码，还要大胆地对单元测试进行重构。如果测试不再反映需求，就重构它。如果发现测试中存在缺口，还要重构它。如果发现测试错误地表达了一个需求，重构它。要对测试进行维护，保持其整洁性。它们对应用程序的重要性和其他代码一样。

4.3 重构示例

假定您的一位朋友正在开发一个“闯关(tic-tac-toe)”游戏。他已经创建了前端，允许用户管理下子顺序，并选择要在其中放置记号的方块。对该示例，他要求你创建一个服务可以让他用来判断是否有一方玩家获胜，所谓获胜是指玩家将记号放在 3 个相邻的方块内，形成水平、垂直或对角行。

你的朋友以需求的方式告诉你，他计划创建一个 `char` 数据类型的 3×3 多维数组，用来表示游戏板，`x` 轴表示水平行，`y` 轴表示垂直列。该数组作为输入参数传递给你要编写的方法。你的朋友希望该库能够返回胜方的标记(`X` 或 `O`)，如果没有玩家获胜，则返回一个空格字符。



提示：

为了练习这里给出的示例，请从 www.wrox.com 下载代码。

4.3.1 第一项功能

为完成这一任务，在 Visual Studio 中创建一个名为 `TicTacToe` 的空白解决方案。向该解决方案中添加一个名为 `TicTacToe.UnitTests` 的类库，它包含一个名为 `GameWinnerServiceTests` 的类(因此，该类的完全限定名为 `TicTacToe.UnitTests.GameWinnerServiceTests`)。还添加对 `NUnit` 的引用，用作单元测试框架(如图 4-1 所示)。

你的朋友要求你实现的第一项功能是：如果任何一个玩家都未能在一行中放入 3 个标记(这里说的一行，包括水平、垂直和对角)，就没有获胜，该方法应当返回一

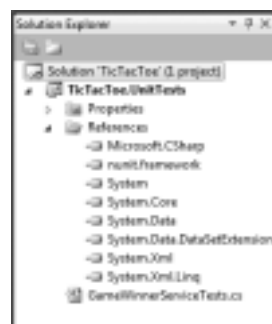


图 4-1

个空字符。第一个测试相当简单。传递一个空数组(即没有任何一个玩家在一行中放入了 3 个标记),并期望返回一个空字符(这是对任何玩家都未能在一行中放入 3 个标记的反应):



可从
wrox.com
下载源代码

```
[TestFixture]
public class GameWinnerServiceTests
{
    [Test]
    public void NeitherPlayerHasThreeInARow()
    {
        const char expected = ' ';
        var gameBoard = new char[3,3] { { ' ', ' ', ' ' },
                                         { ' ', ' ', ' ' },
                                         { ' ', ' ', ' ' } };
        var actual = gameWinnerService.Validate(gameBoard);
        Assert.AreEqual(expected, actual);
    }
}
```

GameWinnerService.cs

马上就可以注意到,这一尝试针对变量 `_gameWinnerService` 调用 `Validate` 方法,但还没有声明它呢。由于需要正在创建的 `GameWinnerService` 的一个实例,声明该变量(为清晰起见,省略了对 `GameWinnerService` 测试类的类声明):



可从
wrox.com
下载源代码

```
[Test]
public void NeitherPlayerHasThreeInARow()
{
    IGameWinnerService gameWinnerService;
    const char expected = ' ';
    var gameBoard = new char[3,3] { { ' ', ' ', ' ' },
                                     { ' ', ' ', ' ' },
                                     { ' ', ' ', ' ' } };
    var actual = gameWinnerService.Validate(gameBoard);
    Assert.AreEqual(expected, actual);
}
```

GameWinnerService.cs

事实上,该测试是不会通过的,因为还没有定义 `IGameWinnerService`。在创建该测试之前,不需要 `IGameWinnerService`,所以它没有存在的理由。既然现在有一个测试需要定义 `IGameWinnerService`,则将其作为下一步要做的。首先在解决方案中创建一个新类库 `TicTacToe.Services`,并把它创建的 `Class1.cs` 文件重命名为 `GameWinnerService.cs`。但不要允许 Visual Studio 重命名该文件内部的类,因为会删除它(如图 4-2 所示)。

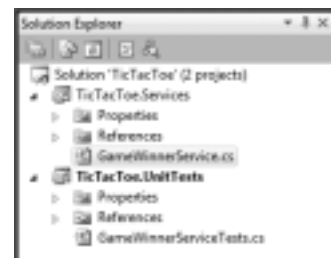


图 4-2

在 `GameWinnerService.cs` 文件中，删除对 `Class1` 的声明，用 `IGameWinnerService` 接口的声明代替它：



```
namespace TicTacToe.Services
{
    public interface IGameWinnerService
    {
    }
}
```

GameWinnerService.cs

返回单元测试，需要在 `TicTacToe.UnitTests` 类库中添加对新 `TicTacToe.Services` 类库的引用。在添加这一引用之后，将 `gameWinnerService` 声明为 `IGameWinnerService` 的类型就不会再导致编译错误。但是，由于该接口还没有方法，因此对 `gameWinnerService.Validate` 的调用仍然会导致编译错误。因此，下一步是向 `IGameWinnerService` 接口添加 `Validate` 方法的声明：



```
public interface IGameWinnerService
{
    char Validate(char[,] gameBoard);
}
```

GameWinnerService.cs

在完成这一工作后，代码仍然不能编译。现在，试图在没有生成具体实例的情况下使用 `gameWinnerService`。为了使该测试通过，下一步就是为支持 `IGameWinnerService` 接口的类创建具体实例。在 `GameWinnerService.cs` 文件中，创建一个名为 `GameWinnerService` 类，它将实现 `IGameWinnerService` 接口：



```
public class GameWinnerService : IGameWinnerService
{
    public char Validate(char[,] gameBoard)
    {
        throw new NotImplementedException ();
    }
}
```

GameWinnerService.cs

严格来说，该方法没有实现方式。事实上，它会抛出表明这一事实的异常。但这样做是可以的；只是编写了刚好足以通向下一步的代码。当在 `NeitherPlayerHasThreeInARow` 方法中向单元测试添加代码以创建 `GameWinnerService` 的具体实例，并将它指定给 `gameWinnerService` 变量时，该应用程序就会编译成功。这刚好是所要关注的全部内容。

向 `NeitherPlayerHasThreeInARow` 方法添加代码，以实例化 `GameWinnerService` 的具体实例：



```
[Test]
public void NeitherPlayerHasThreeInARow()
{
    IGameWinnerService gameWinnerService;
    gameWinnerService = new GameWinnerService();
    const char expected = ' ';
    var gameBoard = new char[3,3] { { ' ', ' ', ' ', ' ', ' ' },
                                     { ' ', ' ', ' ', ' ', ' ' },
                                     { ' ', ' ', ' ', ' ', ' ' } };

    var actual = gameWinnerService.Validate(gameBoard);
    Assert.AreEqual(expected, actual);
}
```

GameWinnerService.cs

这段代码编译成功。在运行测试时，它会失败（这一点与预期相同），因为还没有实现 Validate 方法，如图 4-3 所示。



图 4-3

4.3.2 通过第一个测试

同样，这样做是可以的。有许多不同的理由需要看到测试失败。我们需要确保这些测试真正测试某些东西；永远都不会失败的测试并未真正测试代码。尽管在这一阶段还不太可能发生，但需要验证没有编写过多的代码。还需要确保没有重复创建功能。如果现在不需要再编写任何代码就能使测试通过，很可能是做了重复工作。由于目标是尽可能保持代码为 DRY 的，所以需要确保没有重复实现已经存在的东西。

现在该开始考虑进入“绿灯”阶段了。现在的目标是仅编写适量的代码，以使测试通过。这是非常容易实现的：



```
public char Validate(char[,] gameBoard)
{
    return ' ';
}
```

GameWinnerService.cs

看到这段代码，可能会想：“这并不是‘闯三关’游戏的规则”，你想的没错。但目前的要求不是实现“闯三关”的所有规则，而只是实现其中的一条规则——如果任何玩家

都未能占有一行中的 3 个方块，就无人获胜。这一需求并没有给出应当返回获胜者的任何条件，也没有给出任一玩家在游戏板上的符号信息。这些需求目前还不存在。从图 4-4 可以看出已经通过了这一测试，这表明 Validate 方法的这一实现方式满足了该需求。



图 4-4

当前的测试已经通过。由于业务代码(GameWinnerService)和单元测试(GameWinnerServiceTests)中的东西都不需要重构，现在可以转到下一个业务需求了。

4.3.3 第二项功能

第二项功能规定：如果一位玩家的符号出现在最上方水平行的所有 3 个单元格中，则返回这位玩家的符号，将其作为获胜符号。下面是这一需求的单元测试：



```
[Test]
public void PlayerWithAllSpacesInTopRowIsWinner()
{
    IGameWinnerService gameWinnerService;
    gameWinnerService = new GameWinnerService();

    const char expected = 'X';
    var gameBoard = new char[3, 3]
        { {expected, expected, expected},
          { ' ', ' ', ' ' },
          { ' ', ' ', ' ' } };
    var actual = gameWinnerService.Validate(gameBoard);
    Assert.AreEqual(expected.ToString(),
        actual.ToString());
}
```

GameWinnerService.cs

它与上一个测试非常类似。它构造了一个多维数组充当游戏板，并将它作为参数传递给 GameWinnerService 的实例 gameWinnerService 的 Validate 方法。区别在于，这一次填充的是顶行，变量 expected 的值为 'X'，现在将它用作玩家的符号。当该测试调用 Validate 方法时，它获取值为 'X' 的 char，作为 Validate 的返回值。在运行此测试时，它未能通过，如图 4-5 所示。



图 4-5

下一步就是仅实现使此测试通过的最少量功能：



可从
wrox.com
下载源代码

```
public char Validate(char[,] gameBoard)
{
    var columnOneChar = gameBoard[0, 0];
    var columnTwoChar = gameBoard[0, 1];
    var columnThreeChar = gameBoard[0, 2];
    if (columnOneChar == columnTwoChar & &
        columnTwoChar == columnThreeChar)
    {
        return columnOneChar;
    }
    return ' ';
}
```

GameWinnerService.cs

再次指出，这不一定是“闯三关”游戏最有效或最完整的逻辑；它只是使测试通过所需要的内容。在本例中，所关心的全部内容就是确保在顶行出现相同符号时，从此方法返回该符号。如图 4-6 所示，这一测试得以通过。



图 4-6

在转向下一个测试之前，看看现在 `GameWinnerServiceTests` 类中的单元测试：



可从
wrox.com
下载源代码

```
[TestFixture]
public class GameWinnerServiceTests
{
    [Test]
    public void NeitherPlayerHasThreeInARow()
    {
        IGameWinnerService gameWinnerService;
        gameWinnerService = new GameWinnerService();
        const char expected = ' ';
        var gameBoard = new char[3,3] { {' ', ' ', ' '},
                                          {' ', ' ', ' '}
}
```

```

        { ' ', ' ', ' ' } };
        var actual = gameWinnerService.Validate(gameBoard);
        Assert.AreEqual(expected, actual);
    }

[Test]
public void PlayerWithAllSpacesInTopRowIsWinner()
{
    IGameWinnerService gameWinnerService;
    gameWinnerService = new GameWinnerService();
    const char expected = 'X';
    var gameBoard = new char[3, 3] { {expected, expected, expected},
                                     { ' ', ' ', ' ' },
                                     { ' ', ' ', ' ' } };
    var actual = gameWinnerService.Validate(gameBoard);
    Assert.AreEqual(expected.ToString(), actual.ToString());
}
}

```

GameWinnerService.cs

4.3.4 重构单元测试

可以看到，尽管现在仅编写了两个测试，但也应当对单元测试代码进行一点重构了。对于初学者来说，这两个测试都使用了 `GameWinnerService` 的实例。由于这些测试使用了该类完全相同的具体实现，并通过相同的抽象接口访问它，因此可以从各个测试中移除 `IGameWinnerService` 的声明，使之成为一个成员变量：



可从
wrox.com
下载源代码

```
IGameWinnerService _gameWinnerService;
```

GameWinnerService.cs

不仅可以从各个单元测试中析取声明，还可以从中析取 `GameWinnerService` 具体实例的创建部分以及将该对象向 `_gameWinnerService` 赋值的部分。这可以通过以下方式完成：为 `GameWinnerServiceTests` 类创建一个设置方法，并添加创建 `GameWinnerService` 具体实例的代码。



可从
wrox.com
下载源代码

```

[SetUp]
public void SetupUnitTests()
{
    _gameWinnerService = new GameWinnerService();
}

```

GameWinnerService.cs

这时，运行单元测试，以确保所做的重构工作没有破坏任何东西，如图 4-7 所示。



图 4-7

这些测试现在看起来已经非常符合 DRY 特性了，但这两个单元测试中还使用了其他构造——游戏板。也可以很轻松地将它析取到一个成员变量：



```
private char[,] _gameBoard;
```

GameWinnerService.cs

和析取 `_gameWinnerService` 一样，用于创建 `_gameBoard` 成员变量的具体实例的部分也可以移出各个单元测试，将其放在 `SetupUnitTests` 方法中：



```
[SetUp]
public void SetupUnitTests()
{
    _gameWinnerService = new GameWinnerService();
    _gameBoard = new char[3, 3]
    {
        { ' ', ' ', ' ', ' ', ' ' },
        { ' ', ' ', ' ', ' ', ' ' },
        { ' ', ' ', ' ', ' ', ' ' }
    };
}
```

GameWinnerService.cs

现在已经完成了这些修改，再次运行单元测试，确保它们仍然能够和原来一样正常工作(如图 4-8 所示)。

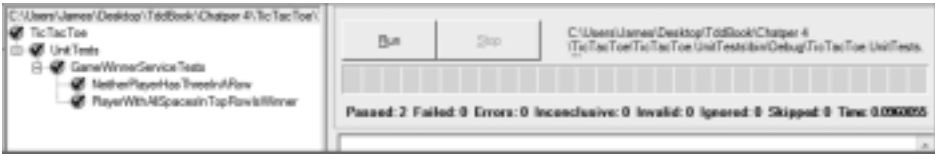


图 4-8

4.3.5 第三项功能

在完成对单元测试的重构之后，就可以开始第三项功能了。新需求规定：如果一位玩家的符号占据了第一列的所有三行，则应当从 `Validate` 方法中返回他的符号，表明他已经获胜。这一点与上一个测试非常类似，从执行这一要求的单元测试代码中可以看出这一点：



```
[Test]
public void PlayerWithAllSpacesInFirstColumnIsWinner()
{
    const char expected = 'X';
    for (var columnIndex = 0; columnIndex < 3; columnIndex++)
    {
        _gameBoard[columnIndex, 0] = expected;
    }
    var actual = _gameWinnerService.Validate(_gameBoard);
    Assert.AreEqual(expected.ToString(), actual.ToString());
}
```

GameWinnerService.cs

运行该测试，从图 4-9 可以看出，该测试未能通过。



图 4-9

和前面的失败测试一样，现在在 Validate 方法中编写能使该测试通过的最少量代码：



```
public char Validate(char[,] gameBoard)
{
    var columnOneChar = gameBoard[0, 0];
    var columnTwoChar = gameBoard[0, 1];
    var columnThreeChar = gameBoard[0, 2];
    if (columnOneChar == columnTwoChar && columnTwoChar == columnThreeChar)
    {
        return columnOneChar;
    }
    var rowTwoChar = gameBoard[1, 0];
    var rowThreeChar = gameBoard[2, 0];
    if (columnOneChar == rowTwoChar && rowTwoChar == rowThreeChar)
    {
        return columnOneChar;
    }
    return ' ';
}
```

GameWinnerService.cs

运行这些测试，可以看出这段代码不仅足以使新测试通过，但仍然能使其他两个测试

继续保持通过状态(见图 4-10)。

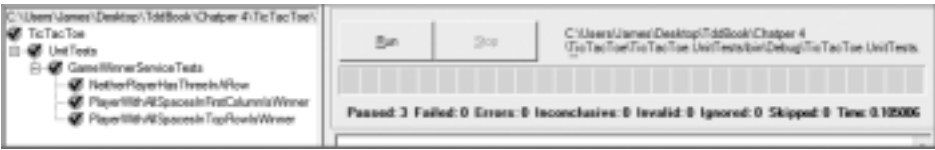


图 4-10

4.3.6 重构业务代码

在进入下一项需求之前,希望再进行一些重构。这一次将注意力转向 GameWinnerService 类中的 Validate 方法:



```
public char Validate(char[,] gameBoard)
{
    var columnOneChar = gameBoard[0, 0];
    var columnTwoChar = gameBoard[0, 1];
    var columnThreeChar = gameBoard[0, 2];
    if (columnOneChar == columnTwoChar && columnTwoChar == columnThreeChar)
    {
        return columnOneChar;
    }

    var rowTwoChar = gameBoard[1, 0];
    var rowThreeChar = gameBoard[2, 0];
    if (columnOneChar == rowTwoChar && rowTwoChar == rowThreeChar)
    {
        return columnOneChar;
    }
    return ' ';
}
```

GameWinnerService.cs

在查看该方法时,可能很快就会注意到两件事:它有点长,且违反了“单一职责原则(SRP)”。这两种代码异味出现在一起并不罕见。在本例中,这两个问题可以用同一组重构步骤来纠正。首先来看看第一处违反 SRP 的地方。

该方法做了 3 件事情。它检查顶行中 3 个单元格;检查第一列的 3 个单元格;检查是否前两个条件均不满足。首先,把检查第一行的代码析取到一个独立方法中:



```
private static char CheckForThreeInARowInHorizontalRow(char[,] gameBoard)
{
    var columnOneChar = gameBoard[0, 0];
    var columnTwoChar = gameBoard[0, 1];
    var columnThreeChar = gameBoard[0, 2];
```

```

        if (columnOneChar == columnTwoChar && columnTwoChar == columnThreeChar)
        {
            return columnOneChar;
        }
        return ' ';
    }
}

```

GameWinnerService.cs

Validate 方法中调用该方法的代码现在应当如下所示。



```

public char Validate(char[,] gameBoard)
{
    var currentWinningSymbol = ' ';
    currentWinningSymbol = CheckForThreeInARowInHorizontalRow(gameBoard);

    var rowOneChar = gameBoard[0, 0];
    var rowTwoChar = gameBoard[1, 0];
    var rowThreeChar = gameBoard[2, 0];
    if (rowOneChar == rowTwoChar && rowTwoChar == rowThreeChar)
    {
        currentWinningSymbol = rowOneChar;
    }

    return currentWinningSymbol;
}

```

GameWinnerService.cs

可以看到，需要对 Validate 方法的逻辑进行较大修改，但该方法看起来已经好得多了。这样，该逻辑更简单，方法更容易理解，也更容易了解后续的发展。现在需要运行单元测试来验证没有改变(破坏)Validate 方法的任何外部行为(如图 4-11 所示)。

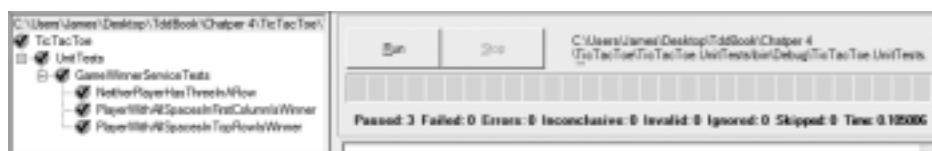


图 4-11

所有测试都通过了，这意味着对 Validate 方法的重构是成功的。接下来，将检查第一竖列中 3 个单元格的代码析取到一个方法中，就像前面检查水平列中 3 个单元格的代码一样：



```

private static char CheckForThreeInARowInVerticalColumn(char[,] gameBoard)
{
    var rowOneChar = gameBoard[0, 0];
    var rowTwoChar = gameBoard[1, 0];
    var rowThreeChar = gameBoard[2, 0];
}

```

```
        if (rowOneChar == rowTwoChar & &
            rowTwoChar == rowThreeChar)
        {
            return rowOneChar;
        }
        return ' ';
    }
```

GameWinnerService.cs

在重构 Validate 方法中的代码，以析取检查第一列中 3 个单元格的代码之后，Validate 方法应当如以下所示。



```
public char Validate(char[,] gameBoard)
{
    var currentWinningSymbol = ' ';
    currentWinningSymbol = CheckForThreeInARowInHorizontalRow(gameBoard);
    currentWinningSymbol = CheckForThreeInARowInVerticalColumn(gameBoard);
    return currentWinningSymbol;
}
```

GameWinnerService.cs

这段代码可能看起来是正确的，但运行单元测试后发现，它改变(破坏)了 Validate 方法的外部功能，如图 4-12 所示。



图 4-12

4.3.7 纠正重构缺陷

再来看看这段代码，好像是如果 CheckForThreeInARowInHorizontalRow 方法返回获胜符号，就调用 CheckForThreeInARowInVerticalColumn 来重写它。该方法会返回一个空字符，因为在测试情况下，两位玩家都没有占据第一垂直列中的 3 个单元格。

该示例应当明确重构时的单元测试值。马上就可以找到代码中的一个错误，因为只做了很少一点修改，所以可以很轻松地判断代码在哪里出现了问题，以及为什么会出现问题。基于单元测试提供的安全性，可以在第一时间进行重构。不用担心在我们不知晓的情况下因为这些修改而在代码中导致错误。单元测试清晰地描述了整个基本代码的健康程度。

在进入下一功能之前，需要修正这一错误：向 Validate 方法中添加代码，查看调用

CheckForThreeInARowInHorizontalRow 是否返回一个获胜符号。如果的确如此，则立即从 Validate 方法返回，甚至不用花费力气去调用 CheckForThreeInARowInVerticalColumn 方法：



可从
wrox.com
下载源代码

```
public char Validate(char[,] gameBoard)
{
    var currentWinningSymbol = ' ';
    currentWinningSymbol = CheckForThreeInARowInHorizontalRow(gameBoard);
    if (currentWinningSymbol != ' ')
        return currentWinningSymbol;
    currentWinningSymbol = CheckForThreeInARowInVerticalColumn(gameBoard);
    return currentWinningSymbol;
}
```

GameWinnerService.cs

运行单元测试，确保这些更改确实使原来失败的测试得以通过，并没有破坏其他两个测试，如图 4-13 所示。



图 4-13

在将检查第一行和第一列是否匹配的逻辑析取到它们自己的方法中之前，这段代码中仅有一个位置使用了空字符值(' '). 在当时的情况下，代码中存在这一字符是可以的，因为仅使用了它一次。而现在将在 4 个不同地方和 3 个不同方法中使用它。这时应当将该文字值从代码中析取出来，将它声明为一个私有常数：



可从
wrox.com
下载源代码

```
private const char SymbolForNoWinner = ' ';
```

GameWinnerService.cs

如果在将来的某一时间，需要改变这个用于表示没有胜者的取值，则只需要在一个地方修改它，而不是在 4 个地方修改。

在继续之前，还需要改变最后一件事情。Validate 方法的当前版本看起来如下所示：



可从
wrox.com
下载源代码

```
private const char SymbolForNoWinner = ' ';
```

GameWinnerService.cs

这里不需要为 currentWinningSymbol 指定默认值，因为无论如何都重写它。事实上，可以在一行代码中声明 currentWinningSymbol，并将它赋给 CheckForThreeInARowInHorizontalRow 的返回值：



```
public char Validate(char[,] gameBoard)
{
    var currentWinningSymbol = SymbolForNoWinner;
    currentWinningSymbol = CheckForThreeInARowInHorizontalRow(gameBoard);
    if (currentWinningSymbol != SymbolForNoWinner)
        return currentWinningSymbol;
    currentWinningSymbol = CheckForThreeInARowInVerticalColumn(gameBoard);
    return currentWinningSymbol;
}
```

GameWinnerService.cs

不需要为 `currentWinningSymbol` 赋默认值，因为，总归要重写该值。事实上，可以声明 `currentWinningSymbol`，并在一行代码中将其赋给 `CheckForThreeInARowInHorizontalRow` 的返回值



```
public char Validate(char[,] gameBoard)
{
    var currentWinningSymbol = CheckForThreeInARowInHorizontalRow(gameBoard);
    if (currentWinningSymbol != SymbolForNoWinner)
        return currentWinningSymbol;
    currentWinningSymbol = CheckForThreeInARowInVerticalColumn(gameBoard);
    return currentWinningSymbol;
}
```

GameWinnerService.cs

这样就去除了不必要的代码行。这看起来似乎没什么大不了的，但如果考虑到在第 3 章曾经指出，编写的代码越多，编写错误代码的机会也就越大，那去掉这一行就很有意义了。通过删除这一行不必要的代码，就降低了 `Validate` 方法中存在错误的机会。

4.3.8 第四项功能

现在已经完成了重构工作，可以转向下一项功能了。该新需求规定：如果一个用户占据了 diagonal 上的 3 个单元格，也就是从左上角开始，穿过中心单元格，结束于右下角，他就是胜者。该需求的单元测试如下所示：



```
[Test]
public void PlayerWithThreeInARowDiagonallyDownAndToRightIsWinner()
{
    const char expected = 'X';
    for (var cellIndex = 0; cellIndex < 3; cellIndex++)
    {
        _gameBoard[cellIndex, cellIndex] = expected;
    }
}
```

```

var actual = _gameWinnerService.Validate(_gameBoard);
Assert.AreEqual(expected.ToString(), actual.ToString());
}

```

GameWinnerService.cs

运行该新单元测试，会发现它没有通过，如图 4-14 所示。

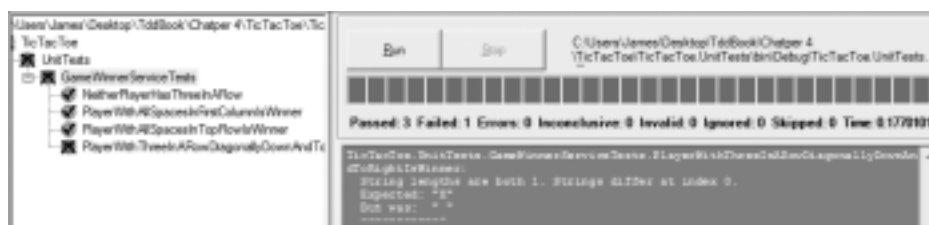


图 4-14

和前面 3 个单元测试一样，下一步的任务就是仅编写使这一测试通过的最少量代码。因为重构了 `Validate` 方法，所以知道查看一位在对角行中占据三个单元格的胜者将是改变 `Validate` 方法的新理由。首先创建一个名为 `CheckForThreeInARowDiagonally` 的私有方法，可以从 `Validate` 方法调用它：



可从
wrox.com
下载源代码

```

private static char CheckForThreeInARowDiagonally(char[,] gameBoard)
{
    var cellOneChar = gameBoard[0, 0];
    var cellTwoChar = gameBoard[1, 1];
    var cellThreeChar = gameBoard[2, 2];
    if (cellOneChar == cellTwoChar && cellTwoChar == cellThreeChar)
    {
        return cellOneChar;
    }
    return SymbolForNoWinner;
}

```

GameWinnerService.cs

现在只需要从 `Validate` 方法中调用该新方法即可：



可从
wrox.com
下载源代码

```

public char Validate(char[,] gameBoard)
{
    var currentWinningSymbol = CheckForThreeInARowInHorizontalRow(gameBoard);
    if (currentWinningSymbol != SymbolForNoWinner)
        return currentWinningSymbol;
    currentWinningSymbol = CheckForThreeInARowInVerticalColumn(gameBoard);
    if (currentWinningSymbol != SymbolForNoWinner)
        return currentWinningSymbol;
    currentWinningSymbol = CheckForThreeInARowDiagonally(gameBoard);
}

```

```
        return currentWinningSymbol;  
    }  
}
```

GameWinnerService.cs

运行这些单元测试表明，不仅新测试得以通过，而且所有其他单元测试也仍然能够通过(如图 4-15 所示)。



图 4-15

GameWinnerService 现在还绝对没有完成。该项目可从 Wrox 网站(www.wrox.com)下载。强烈建议下载该项目，并练习在本章学习的工作流程和第 3 章介绍的重构技巧，自行完成 GameWinnerService。这是练习和熟悉“红灯、绿灯、重构”这一 TDD 工作流的绝佳方式。

4.4 本章小结

使用 TDD 的开发人员会使用几种大家可能还不太熟悉的框架和工具。但是，相对于学习新工具和框架来说，更难克服的障碍可能是改变思考和编写软件的方式。在编写要测试的代码之前首先编写测试，初看起来似乎有些不够自然，所以可能要花费一些时间来忘掉过去编写代码的知识。

“红灯、绿灯、重构”这一口诀确定了练习使用 TDD 的工作流。一定要先编写测试，尽管它无法编译。仅编写使测试通过的最少量代码。不要担心它不够完美，或者可能感觉没有完全涵盖企业所关心的所有基础内容；这不是现在优先考虑的内容。只需要担心当前摆在面前的需求即可。如果这些测试满足当前的业务需求，而且能够通过，那就够了！在编写了足够的代码，完成业务功能之后，就可以对代码进行重构，增强其可读性和可维护性。

记住，测试也是基础代码的组成部分。代码的出色程度顶多和测试相同。不要仅测试那些“幸福路径”。一定要从多个角度进行测试，以暴露代码中的弱点。将测试看作业务逻辑的组成部分；不要让代码从内部坏掉。

下载本章的“闯关” GameWinnerService 示例，并完成它。提出自己的一些功能，以定义游戏规则。编写测试，然后添加到 GameWinnerService 中，使这些测试得以通过。不断重构业务代码和单元测试，使它们保持可靠性和针对性。

第 5 章

模拟外部资源

本章内容

- 为什么静态绑定组件会创建脆弱的系统？
- 依赖项注入模式如何通过动态绑定这些组件而提供帮助？
- 依赖项注入框架如何帮助管理应用程序内部的依赖项？
- 如何用库模式有效地模拟应用程序的数据访问代码？

作为业务应用程序开发人员，所编写的大多数应用程序都是由各种组件组成的(有些组件是自己创建的；有些来自第三方)，这些组件组合在一起执行一项任务。有时，这些组件表示外部资源，如数据库、Web 服务、文件系统甚至物理设备。应用程序中的组件需要有这些外部资源，但我们经常需要能够在不采用这些外部资源的情况下隔离测试这些组件。采用 TDD 的开发人员是通过在运行时为这些外部资源提供替身而做到这一点的。这些替身称为“模拟”。

为了使用模拟，并真正地隔离各个被测组件，需要一种替代方法来代替静态绑定各个组件的方法。依赖项注入模式可用于在运行时或测试期间注入这些组件的具体实现。依赖项注入框架帮助我们管理依赖项的 Web，并确保根据使用组件的上下文提供正确的具体实现。

5.1 依赖项注入模式

在使用 OOP 设计的传统应用程序中，创建服务和实体类来抽象现实世界中的过程和实体。应用程序便由各种这样的服务和实体类组成。这一组成关系在应用程序的类之间创建了依赖关系。表示和管理这些依赖关系的最明显方法是在代码中静态创建它们，作为类初始化过程的一部分(该例可以在 www.wrox.com 中所下载代码的 Before 文件夹中找到)。



可从
wrox.com
下载源代码

```
public class BusinessService
{
    private readonly string _databaseConnectionString =
        ConfigurationManager.ConnectionStrings["MyConnectionString"]
            .ConnectionString;
    private readonly string _webServiceAddress =
        ConfigurationManager.AppSettings["MyWebServiceAddress"];
    private readonly LoggingDataSink _loggingDataSink;

    private DataAccessComponent _dataAccessComponent;
    private WebServiceProxy _webServiceProxy;
    private LoggingComponent _loggingComponent;

    public BusinessService()
    {
        _loggingDataSink = new LoggingDataSink();
        _loggingComponent = new LoggingComponent(_loggingDataSink);
        _webServiceProxy = new WebServiceProxy(_webServiceAddress);
        _dataAccessComponent = new DataAccessComponent(_databaseConnection
            String);
    }
}
```

BusinessService.cs

使用这一方法处理类之间的依赖关系时，会出现几个问题。最明显的问题是，该代码被静态绑定到编写该类时所使用的特定具体实现。这样，如果不打开该代码，就无法改变这些依赖关系中所使用的具体对象。因为是静态绑定，也不可能再用模拟对象来替代任何这类依赖项，也就不可能再为单元测试隔离该类的方法中的任何代码。

而依赖关系都不是基于契约的这一事实又使问题进一步复杂化。它们依赖于具体的类定义。依赖于具体类而不是抽象接口，就会限制替换这些服务的能力。要开放该类以允许修改(根据开放/关闭原则, OCP)，必须小心地用那些支持相同 API 的类来替代这些依赖项，否则就要准备对该类进行大幅修改。

该方法最重要的问题在于：BusinessService 类需要非常了解其所依赖的类，而这些了解仅仅是为了生成这些类的具体实例。它必须知道 DataAccessComponent 需要一个连接字符串，WebServiceProxy 需要一个服务地址，而 LoggingComponent 需要为它的消息准备一

个数据接收器。这都违反了 SRP，因为创建这些类所需要的知识一定存在于应用程序的其他某个地方。事实上，可以肯定：在使用这些依赖类的应用程序中，一定重复出现了这些相同代码。这就意味着，如果这些类中有任何一个改变了该类，就有可能要改变更多的内容。

依赖项注入模式提供了一种框架，可用于在创建类时向该类提供依赖对象的具体实例。注入这些依赖项的最常见、最简单方法是将它们作为构造函数实参传递给一个对象。如果将前面的示例重构为依赖项注入模式，将会产生的类如下所示(该示例可以在 www.wrox.com 所下载代码中的 After 文件夹中找到)：



可从
wrox.com
下载源代码

```
public class BusinessService
{
    private IDataAccessComponent _dataAccessComponent;
    private IWebServiceProxy _webServiceProxy;
    private ILoggingComponent _loggingComponent;

    public BusinessService (IDataAccessComponent dataAccessComponent,
                           IWebServiceProxy webServiceProxy,
                           ILoggingComponent loggingComponent)
    {
        _loggingComponent = loggingComponent;
        _webServiceProxy = webServiceProxy;
        _dataAccessComponent = dataAccessComponent;
    }
}
```

BusinessService.cs

实现依赖项注入模式已经使 BusinessService 变得更易于理解和维护。限制依赖项仍很重要。拥有太多构造函数实参的类本身就是一种代码异味，但依赖项注入模式在很大程度上简化了对该类所含依赖项的管理。

除了提高 BusinessService 的可读性和可维护性之外，现在可以在调用该类时注入这些依赖项的任何实现方式，只要它们实现了所规定的接口即可。可以根据规则或配置值提供不同的实现方式。这种做法对于以下情景是非常有帮助的：希望该应用程序的实例在 QA 环境中使用测试数据库或 Web 服务，在准备将此应用程序交付生产时，再转换到这些资源的生产实例。对于 TDD 开发人员来说，最大的好处是这些服务的模拟实例可以注入该类，进行测试。这样就可以提供替代实现，而这些替代实现可以提供封装好的响应，以隔离该类中的代码，从而进行测试。

使用依赖项注入框架

前面的示例表明如何使用构造函数参数来注入类的依赖项，它创建了 BusinessService 更整洁的实现。但为这些依赖对象创建具体实例的逻辑并没有消失；还必须从基础代码中的其他位置清除它。但在哪里呢？

许多 OOP 开发人员都熟悉一或多种工厂(factory)模式,如工厂、抽象工厂、构造器和原型。它们都是一种基本模式的派生内容,用于在应用程序中创建类。所有工厂模式基本上都描述了一种方法,用于获得实现指定接口的对象实例。请求方只知道它需要哪一种接口。工厂可以根据所请求的接口,以及工厂开发人员所提供的任意配置设置或基于上下文的规则,决定向该请求方提供哪种特定类。

工厂模式对于抽象那些创建依赖对象所需的逻辑和细节是非常有用的,这样它们就不必出现在应用程序的业务域代码中。但是对于大型应用程序会出现一个问题。向应用程序添加的服务和依赖项会越来越多,管理和维护这些工厂的任务也随之变得非常困难和费时。

依赖项注入(DI)框架为开发人员提供了代替传统工厂模式的方式。利用它们可以快速、轻松地在系统内定义依赖项,以及根据这些依赖项创建相应的具体对象的规则。这些规则和设置可用于构建整个类树,这样就不需要知道那些依赖于这些依赖项的对象。简而言之,DI 框架做了工厂类的所有工作,而开发人员只需要做很少一点工作。本书的示例使用了 Ninject DI framework 2.1.0.91 版,可以从 www.ninject.org 下载。

以下代码回顾了前面的示例。已经将 `DataAccessComponent`、`WebServiceProxy` 和 `LoggingComponent` 的声明类型改变为相应的接口,并向 `LoggingDataSink` 类添加了一个 `ILoggingDataSink` 接口。还删除了所有引发 `NotImplementedException` 异常的代码行:



```
public class BusinessService
{
    private readonly string _databaseConnectionString =
        ConfigurationManager.ConnectionStrings
            ["MyConnectionString"].ConnectionString;
    private readonly string _webServiceAddress =
        ConfigurationManager.AppSettings["MyWebServiceAddress"];
    private readonly ILoggingDataSink _loggingDataSink;
    private DataAccessComponent _dataAccessComponent;
    private WebServiceProxy _webServiceProxy;
    private LoggingComponent _loggingComponent;
    public BusinessService()
    {
        _loggingDataSink = new LoggingDataSink();
        _loggingComponent = new LoggingComponent(_loggingDataSink);
        _webServiceProxy = new WebServiceProxy(_webServiceAddress);
        _dataAccessComponent = new DataAccessComponent(_databaseConnection
            String);
    }
}
```

BusinessService.cs

接口对于发挥依赖项注入的效力非常重要。将成员变量声明为上面的接口，就是隐含地表明我不在乎每个服务采用哪种特定的具体实现，我只在乎它实现了所声明的接口。这样，就能将判断具体类型的任务委托给依赖项注入框架完成。依赖项注入框架随后变成这些规则和逻辑的唯一储藏室，用以判断何时使用哪种具体实例。接口的使用使之成为可能。

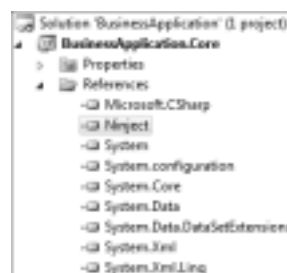


图 5-1

该示例说明如何使用 Ninject DI 框架来负责依赖项及其创建过程。要使用 Ninject，必须向项目中添加对 Ninject 框架程序集(Ninject.dll)的引用，如图 5-1 所示。

Ninject 需要类来存储其用于生成依赖项具体实例的规则。这些类被称为模块。下一步是创建一个新的类，用于保存 BusinessApplication.Core 项目的规则。将它称为 CoreModule。为了能够供 Ninject 使用，该类必须从 NinjectModule 继承而来，而且为 Load 方法提供一种实现。



可从
wrox.com
下载源代码

```
public class CoreModule : NinjectModule
{
    public override void Load()
    {
        throw new NotImplementedException ();
    }
}
```

CoreModule.cs

通过为抽象类 NinjectModule 提供具体实现，CoreModule 可以根据客户的请求，为 Ninject 提供其创建具体类时所需要的信息。在被重写的 Load 方法中定义一些规则，用于创建自己的类。首先是一个比较容易的类：LoggingDataSink：



可从
wrox.com
下载源代码

```
public override void Load()
{
    Bind < ILoggingDataSink > ().To < LoggingDataSink > ();
}
```

CoreModule.cs

选择 LoggingDataSink 作为第一个示例，是因为它没有输入形参，实际上被用作其他具体类的输入形参。该示例使用 Ninject 的 Bind 命令指定：向 Ninject 请求一个实现 ILoggingDataSink 接口的类时，应当返回 LoggingDataSink 的具体实例，这是用 To 扩展方法指定的。

下一个要创建的绑定是 LoggingComponent 的绑定。创建这一绑定的语法与 LoggingDataSink 绑定的语法完全相同：



```
public override void Load()
{
    Bind < ILoggingDataSink > ().To < LoggingDataSink > ();
    Bind < ILoggingComponent > ().To < LoggingComponent > ();
}
```

CoreModule.cs

记住，LoggingComponent 以 ILoggingDataSink 的实例作为构造函数形参。但该示例没有 Ninject 中指定一条规则，说明如何构建这一关系或者 LoggingComponent 如何获取 ILoggingDataSink 正确的具体实例。DI 框架的好处是不需要指定这一规则。Ninject 可以检查 LoggingComponent，并看到它有一个构造函数。它还看到该构造函数有一个 ILoggingDataSink 类型的形参。由于 Ninject 有一条关于如何创建 ILoggingDataSink 的规则，所以它能够推断：在创建 LoggingComponent 的实例时，还需要创建一个类的实例（该类实现 ILoggingDataSink，在本例中为 LoggingDataSink），并将它作为构造函数形参传递给 LoggingComponent。我们不需要做其他任何事情，可以免费获得这一功能。无论需要多少层，这一功能都可以进行相应扩展。例如，如果 LoggingDataSink 有一个 Ninject 可以满足的构造函数实参（也就是说，Ninject 拥有的信息可以供其创建一个与该构造函数形参一致的对象），则 Ninject 也会负责其创建任务。除了指定附加规则之外，不需要做其他任何事情。

DataAccessComponent 要比 LoggingComponent 更复杂一些。它有一个依赖项和一个构造函数形参，但对于 DataAccessComponent 来说，该形参是一个来自应用程序配置文件的字符串，而不是另一个类。

总会存在这样一些情况：类在构造函数形参中所需要的是 Ninject 所不能提供的。从应用程序配置文件中为 DataAccessComponent 提取出来的连接字符串就属于这一类。有些实例化逻辑更为复杂，不仅是将接口映射到类，为了处理这种逻辑，Ninject 允许为特定接口创建提供程序。

提供程序也是一种类，可以从模块中的代码抽象出复杂的创建逻辑。要创建提供程序，可以创建一个类，它应当继承自 Ninject 附带的抽象 Provider 类：



```
public class DataAccessComponentProvider : Provider < IDataAccessComponent >
{
    protected override IDataAccessComponent CreateInstance(IContext context)
    {
        throw new NotImplementedException ();
    }
}
```

CoreModule.cs

向提供程序基础提供的泛型就是该提供程序应当被绑定到的接口——在本例中，就是 IDataAccessComponent。需要为 CreateInstance 方法提供实现，用于返回类的实例，该实例

实现 `IDataAccessComponent` 接口。对于该示例应用程序，它就是 `.DataAccessComponent` 的一个实例，将来自应用程序配置文件的连接字符串值提供给它：



可从
wrox.com
下载源代码

```
protected override IDataAccessComponent CreateInstance(IContext context)
{
    var databaseConnectionString =
        ConfigurationManager.ConnectionStrings
            ["MyConnectionString"].ConnectionString;
    return new DataAccessComponent(databaseConnectionString);
}
```

CoreModule.cs

用于从应用程序的配置文件中检索连接字符串并返回 `DataAccessComponent` 实例的代码很简单，也易于理解。将这一代码放在它自己的提供程序类中，会使我们能够从 `CoreModule` 的 `Load` 方法中所列的更简单规则中抽象出 `DataAccessComponent` 创建逻辑。现在需要添加对 `CoreModule` 类的 `Load` 方法的引用，使 `Ninject` 知道在请求 `IDataAccessComponent` 时如何创建 `DataAccessComponent`：



可从
wrox.com
下载源代码

```
public override void Load()
{
    Bind < ILoggingDataSink > ().To < LoggingDataSink > ();
    Bind < ILoggingComponent > ().To < LoggingComponent > ();
    Bind < IDataAccessComponent > ().ToProvider(new DataAccessComponent
        Provider());
}
```

CoreModule.cs

这就是所需要的全部内容。从 `Ninject` 请求 `IDataAccessComponent` 的一个实例时，将会得到一个正确创建的 `DataAccessComponent`。

为 `WebServiceProxy` 创建提供程序的代码基本相同：



可从
wrox.com
下载源代码

```
public class WebServiceProxyComponentProvider : Provider < IWebServiceProxy >
{
    protected override IWebServiceProxy CreateInstance(IContext context)
    {
        var webServiceAddress = ConfigurationManager.AppSettings
            ["MyWebServiceAddress"];
        return new WebServiceProxy(webServiceAddress);
    }
}
```

CoreModule.cs

为了向 `Ninject` 通知这一新提供程序，需要修改 `CoreModule`，这一修改和前面添加 `DataAccessComponentProvider` 的修改一样简单。



可从
wrox.com
下载源代码

```
public override void Load()
{
    Bind < ILoggingDataSink > ().To < LoggingDataSink > ();
    Bind < ILoggingComponent > ().To < LoggingComponent > ();
    Bind < IDataAccessComponent > ().ToProvider(new DataAccessComponent
        Provider());
    Bind < IWebServiceProxy > ().ToProvider(new WebServiceProxyComponent
        Provider());
}
```

CoreModule.cs

完成 CoreModule 后，可以重构 BusinessService 类，以清除创建依赖对象所需要的逻辑：



可从
wrox.com
下载源代码

```
public class BusinessService
{
    private ILoggingComponent _loggingComponent;
    private IWebServiceProxy _webServiceProxy;
    private IDataAccessComponent _dataAccessComponent;

    public BusinessService(ILoggingComponent loggingComponent,
        IWebServiceProxy webServiceProxy,
        IDataAccessComponent dataAccessComponent)
    {
        _loggingComponent = loggingComponent;
        _webServiceProxy = webServiceProxy;
        _dataAccessComponent = dataAccessComponent;
    }
}
```

BusinessService.cs

BusinessService 的实现要比上一版本整洁得多。清除创建和维护依赖对象所需要的逻辑可以防止违反 SRP。这是因为，如果创建这些依赖对象的例程发生变化，代码中只有一个位置需要改变。BusinessService 类不再需要关心这些对象如何创建，只要它接收的对象实现了正确的接口就足够了。

那么，如何创建 BusinessService 的一个实例呢？在使用 DI 框架时，不能像过去习惯的那样用 new 方式来创建对象：

```
var businessService = new BusinessService();
```

如果没有构造函数形参，它就不可能编译成功。所以需要向 Ninject 请求 BusinessService 的一个实例。大多数 DI 框架都提供了一种存储库类，使人们能够从该框架中请求对象。在 Ninject 中，该存储库称为“内核”。为了演示从内核中请求对象的过程，要创建一个单元测试。第一步是创建一个包含单元测试的项目，如图 5-2 所示。

除了 nunit.framework 程序集之外,添加对 Ninject 程序集的引用,需要用它来创建内核。该内核是 Ninject StandardKernel 类的实例,并将它创建为 Ninject 框架的接口。将通过调用 Get 方法并指定需要哪个接口的实例,来请求 StandardKernel 的这一实例。StandardKernel 利用在构造过程提供的规则来判断应当返回什么样的具体实例,并向调用方法返回一个完整的对象图。

接下来创建一个单元测试,以测试从 Ninject 内核获取 BusinessService 实例的过程:

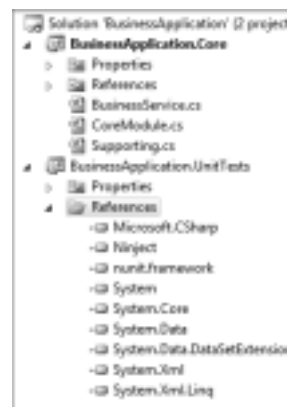


图 5-2



可从
wrox.com
下载源代码

```
[TestFixture]
public class BusinessServiceTests
{
    [Test]
    public void ShouldBeAbleToGetBusinessServiceFromNinject()
    {
        BusinessService actual;
        Assert.IsNotNull(actual);
    }
}
```

BusinessServiceTests.cs

运行该测试的尝试将会失败,因为代码现在还不能编译,如图 5-3 所示。



图 5-3

要使这一测试能够运行,需要从 Ninject 内核请求一个对象,赋给 actual。要使用这个 Ninject 类,需要向单元测试的开头添加针对 Ninject 程序集的 using 语句:



可从
wrox.com
下载源代码

```
using Ninject;
```

BusinessServiceTests.cs

现在需要向单元测试中添加代码,以生成 Ninject 内核的实例,并从中请求 BusinessService 的实例:



可从
wrox.com
下载源代码

```
[Test]
public void ShouldBeAbleToGetBusinessServiceFromNinject()
{
    BusinessService actual;
    var kernel = new StandardKernel(new CoreModule());
```

```

        actual = kernel.Get < BusinessService > ();

        Assert.IsNotNull(actual);
    }

```

BusinessServiceTests.cs

在创建 Ninject 的 StandardKernel 的实例时,需要向它提供 CoreModule 的实例。Ninject 通过这一途径来了解创建对象所需要的规则和步骤。在这里给出的示例中,StandardKernel 仅以一个模块(CoreModule)作为其规则集。在该例中,这样是可以的,因为只有一个正在使用的库(BusinessApplication.Core)。但如果有几个程序集呢?可以将其中一个作为主模块,其中包含解决方案中全部内容的创建规则,但这不是个好主意。它要求主模块所在的项目非常了解其他项目和程序集。它还限制了在解决方案中各种组件的重用。如果 B 库中有一个组件,但它的创建规则包含在 A 库中的一个模块中,现在就要依赖于 A 库及其所有依赖项。为了解决这个问题,可以在解决方案中为每个项目创建一个模块,仅为该库中的类和接口提供创建规则。这意味着降低这些库之间的相互依赖,使重用变得容易了许多。

重写 StandardKernel 的构造函数,使其包含一个模块数组。这样就可以为应用程序创建单个内核(在实践中,该内核应当仅有一个),它拥有在该应用程序中的任何地方创建任何对象所需要的全部信息。创建这样一个内核的语法非常简单:

```

var kernel = new StandardKernel(new ModuleA(),
                                new ModuleB(),
                                new ModuleC());

```

运行这些测试表明,仍然存在一个问题,如图 5-4 所示。



图 5-4

堆栈跟踪表明,在 CoreModule.cs 文件的第 32 行有问题:



```

var databaseConnectionString =
    ConfigurationManager.ConnectionStrings
        ["MyConnectionString"].ConnectionString;

```

CoreModule.cs

当单元测试从应用程序配置文件(web.config 或 app.config)请求信息时,它需要自己拥有该文件的实例。如果该文件已经被添加到该测试所在的 Visual Studio 项目中,或者从另一个项目中复制而来,则没问题。它不能访问另一个项目中的配置文件。为了解决这个问题,可以向 BusinessApplication.UnitTests 项目中添加一个 app.config 文件,并用示例连接字符串填充它:



```
< ?xml version="1.0" encoding="utf-8" ? >
< configuration >
  < connectionStrings >
    < add name="MyConnectionString" connectionString=
      "SampleConfiguration"/ >
  < /connectionStrings >
< /configuration >
```

app.config

现在运行测试时,Ninject 已经创建了 BusinessService 的实例,并拥有其所有依赖项目,如图 5-5 所示。



图 5-5

前面创建 CoreModule 时,没有指定 BusinessService 的创建规则。由于向 Ninject 请求指定具体类的实例,而不是满足接口的对象,因此 Ninject 就将它所拥有的规则用于依赖 BusinessService 的类,并返回 BusinessService 类的具体实例。CoreModule 中的示例显示的是将接口绑定到具体类,但也可以将具体类绑定到类:

```
Bind < BusinessService > ().To < BusinessService > ();
```

在将一个类绑定到其自身时,Ninject 用以下语法提供了一种快捷方式:

```
Bind < BusinessService > ().ToSelf();
```

当然,如果要创建一个 IBusinessService 接口,并让 BusinessService 实现该接口,可以绑定它们两个,并让测试从 Ninject 内核请求 IBusinessService 实例:

```
IBusinessService actual;
var kernel = new StandardKernel(new CoreModule());
actual = kernel.Get < IBusinessService > ();
```

本书的其余实例更全面地应用了 Ninject,并演示了此框架的更多功能。

5.2 抽象数据访问层

大多数业务应用程序都会使用某种数据存储。可能是传统的数据库管理系统(DBMS)、文本文件或二进制文件、Web 服务，或者其他用于持久保存数据的机制。数据管理与持久化是业务开发人员大多数工作的焦点。但在编写单元测试时，隔离被测代码是非常重要的，即使是那些涉及数据访问的代码也是如此。利用存储库模式，可以将应用程序中负责从业务逻辑中访问外部数据存储的组件抽象出来，从而为业务逻辑创建隔离测试。

5.2.1 将对数据库的关注移出业务代码

在发布 .NET 1.0 时，它对微软的 ActiveX Data Object (ADO) 框架进行了更新，称之为 ADO.NET。ADO.NET 不只是 ADO 的 .NET 端口；它添加了许多功能，专门设计用来帮助开发人员快速、轻松地处理类和对象。这样，ADO.NET 的早期用户采用了一种“智能对象”体系结构，在这种体系结构中，为每个类都赋予了自持久化功能。这一功能往往在类本身嵌入一些 ADO.NET 代码，使用数据读取器或数据集，以向数据库中存储数据或从中检索数据。

尽管这样的确创建了一些基本代码，可以供开发人员快速、轻松地开发自持久化对象，但也引发了许多问题。在应用程序中到处分布数据访问逻辑，会产生许多维护问题，因为在改变数据库对一个类的处理方式时，很少能与该类完全保持隔离。因此，不能围绕数据库访问构建单一故障点。数据访问方式的变化和缺陷有时需要对应用程序基本代码进行大幅改造。

如果系统中的类之间存在复杂关系，也会产生问题。有些情况下，类如果不实例化几个子对象，就不能正确地实例化其自身。这样就出现了一个与上面类似的问题，在上面的问题中，有关如何创建具体对象的逻辑和规则都分散在整个应用程序中。这样就不可能不违反 SRP，也就不可能拥有一个有效的工作系统。

最后，这一模式使隔离单元测试的编写变成梦魇，甚至根本就不可能完成。这种紧密嵌入的 ADO.NET 代码不可能快速、轻松地与业务逻辑相分离，从而就很难为这些数据库连接点提供模拟对象，甚至不可能提供。

5.2.2 将数据与存储库模式隔离

最终，开发人员决定：尽管智能对象可能很有帮助，但将数据访问逻辑从业务实体中分离出去的好处要更多些。这样可以解除耦合，从而可以使业务开发能够独立于数据访问和数据存储开发而继续向前发展。这种独立性意味着后端数据存储可以发生变化，有时甚至能够以动态方式变化，而业务领域的逻辑和实体是与这些变化相隔离的。这种松散耦合使开发人员能够更轻松地为业务逻辑编写隔离单元测试。

简而言之，存储库模式要求所有数据访问都封装在一个存储库对象中，业务域类将用它来执行所有持久化工作(如图 5-6 所示)。

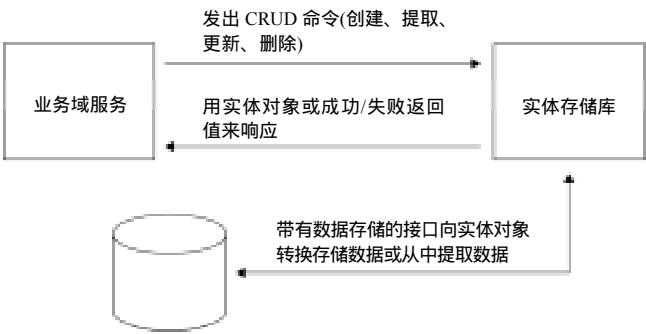


图 5-6

存储库在业务域与数据存储之间创建了一层垫片。存储库封装了如何以简化格式存储数据以及如何将数据从这种简化格式转换到实体对象的相关知识。每个实体应当拥有其自己的存储库。但是，不同类型实体的存储库应当拥有类似的接口和共同的基类，以帮助数据访问代码保持 DRY，从而可以更轻松地了解如何使用存储库。

存储库可以与任意数目的数据访问技术一起使用，包括 ADO.NET、Web 服务和平面文件存储。许多对象关系映射器(ORM)框架简化了存储库的创建。在使用框架(如实体框架或 nHibernate)管理持久性时，有可能编写一个存储库，以某一实体类型为泛型，将这种类型专属的工作交给框架完成。如需更多信息，参阅 ORM 的文档。

5.2.3 注入存储库

可以创建一个 Person 实体，对前面示例的解决方案进行一些补充。



```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

person.cs

PersonService 类用作业务域类，从执行业务规则的角度使用 Person 对象，并在业务工作流中使用 Person。下面是 PersonService 的接口：



```
public interface IPersonService
{
    Person GetPerson (int personId);
}
```

personService.cs

PersonService 使用 PersonRepository 的实例进行数据持久化。该示例仅实现了 GetPerson 方法，它没有使用数据库后端，而是使用了一个 IList 成员变量来保存 Person 数据：



```
public interface IPersonRepository
{
    Person GetPerson(int personId);
}

public class PersonRepository : IPersonRepository
{
    private readonly IList < Person > _personList;

    public Person GetPerson(int personId)
    {
        return _personList.Where(person => person.Id == personId).
            FirstOrDefault();
    }

    public PersonRepository()
    {
        _personList = new List < Person >
        {
            new Person {Id = 1, FirstName = "John", LastName = "Doe"},
            new Person {Id = 2, FirstName = "Richard", LastName = "Roe"},
            new Person {Id = 1, FirstName = "Amy", LastName = "Adams"}
        };
    }
}
```

personRepository.cs

现在已经定义了应用程序中的相关部分，可以实现 PersonService 了：



```
public class PersonService : IPersonService
{
    public PersonService()
    {
    }

    public Person GetPerson (int personId)
    {
        return new Person();
    }
}
```

personService.cs

现在 PersonService 有了一个空构造函数。需要注入 PersonRepository，以便 PersonService 能够访问数据存储，并改变 GetPerson 方法的实现：



可从
wrox.com
下载源代码

```
public class PersonService : IPersonService
{
    private readonly IPersonRepository _personRepository;

    public PersonService(IPersonRepository personRepository)
    {
        _personRepository = personRepository;
    }

    public Person GetPerson (int personId)
    {
        return _personRepository.GetPerson(personId);
    }
}
```

personService.cs

最后，需要定义在 CoreModule 中创建 PersonService 和 PersonRepository 的规则：



可从
wrox.com
下载源代码

```
Bind < IPersonRepository > ().To < PersonRepository > ();
Bind < IPersonService > ().To < PersonService > ();
```

CoreModule.cs

为了验证所有一切都被正确连接在一起，需要编写一个单元测试，以验证可以从 Ninject 获取 PersonService 的实例以及一个 Person，它指出可能调用 PersonRepository：



可从
wrox.com
下载源代码

```
[TestFixture]
public class PersonServiceTests
{
    [Test]
    public void ShouldBeAbleToCallPersonServiceAndGetPerson()
    {
        var expected = new Person {Id = 1, FirstName = "John", LastName = "Doe"};
        var kernel = new StandardKernel(new CoreModule());
        var personService = kernel.Get < PersonService > ();
        var actual = personService.GetPerson(expected.Id);

        Assert.AreEqual(expected.Id, actual.Id);
        Assert.AreEqual(expected.FirstName, actual.FirstName);
        Assert.AreEqual(expected.LastName, actual.LastName);
    }
}
```

PersonServiceTests.cs

运行此单元测试，将会看到它顺利通过，这表明正确地创建了 PersonService 和 PersonRepository，如图 5-7 所示。



图 5-7

5.2.4 模拟存储库

该单元测试能够正确运行，但仍然存在一个严重问题。由于它突破了 PersonService 和 PersonRepository 之间的边界，因此并不是真正的单元测试。要使它成为真正的单元测试，需要提供一个模拟对象，以代表 PersonService 中的 PersonRepository。该示例说明如何使用 Moq 框架为 PersonRepository 生成模拟：



可从
wrox.com
下载源代码

```
var personRepositoryMock = new Mock < IPersonRepository > ();
personRepositoryMock
    .Setup(pr => pr.GetPerson(1))
    .Returns(new Person {Id = 1, FirstName = "Bob", LastName = "Smith"});
var personService = new PersonService(personRepositoryMock.Object);
```

PersonServiceTests.cs

第 2 章介绍了 Moq 的基础知识，但这里会通过解释这段代码所做的工作来快速回顾一下这些内容。这段代码段的第一行要求 Moq 框架根据 IPersonRepository 创建一个名为 personRepositoryMock 的模拟对象或替代对象。目前，personRepositoryMock 是一个实现 IPersonRepository 接口的空类；它有方法，但在为这些方法提供实现方式之前，它们不会实际做任何事情。

之后的一行完成了提供这些方式的任务：在 Moq 中使用 Setup 和 Returns 来扩展方法，告诉 personRepositoryMock，当以实参 1 调用 GetPerson 时，它应当返回在 Returns 扩展方法中描述的 Person 对象。最后，手工创建了 PersonService 的实例(没有再使用 Ninject)，并将它作为构造函数实参传递，以注入 PersonRepository(personRepositoryMock)的模拟版本。

下面是整个测试，其中包括用于创建 PersonRepository 模拟版本的代码：



可从
wrox.com
下载源代码

```
[Test]
public void ShouldBeAbleToCallPersonServiceAndGetPerson()
{
    var expected = new Person {Id = 1, FirstName = "John", LastName = "Doe"};
    var personRepositoryMock = new Mock < IPersonRepository > ();
    personRepositoryMock
        .Setup(pr => pr.GetPerson(1))
        .Returns(new Person {Id = 1, FirstName = "Bob", LastName = "Smith"});
    var personService = new PersonService(personRepositoryMock.Object);
    var actual = personService.GetPerson(expected.Id);

    Assert.AreEqual(expected.Id, actual.Id);
}
```

```

        Assert.AreEqual(expected.FirstName, actual.FirstName);
        Assert.AreEqual(expected.LastName, actual.LastName);
    }

```

PersonServiceTests.cs

注意，在设置 personRepositoryMock 时，Person 对象与预期不匹配。所以该测试应当会失败，如图 5-8 所示。

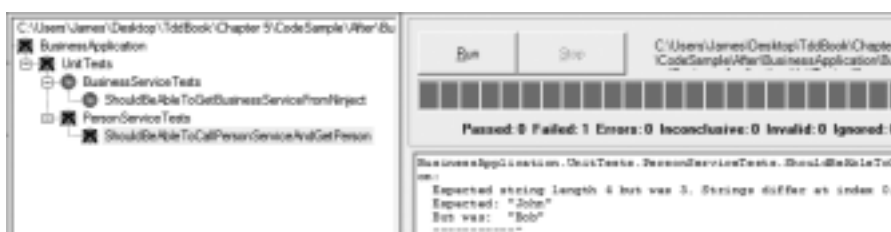


图 5-8

这个未能通过的测试以及有关名字不匹配的消息表明：PersonService 的这个测试版本使用了 PersonRepository 的模拟版本，而不是该类的实际实现方式。

5.3 本章小结

作为业务开发人员，工作中编写的大多数应用程序都依赖于某类外部资源。这些资源可能是数据库、Web 服务、文件系统，甚至是某种物理设备。除此之外，仍然需要确保能够隔离所编写的单元测试，以便在不需要任何依赖项的情况仅执行正在测试的特定方法。用静态生成依赖项的传统方法创建的应用程序非常脆弱，而且难以维护。

依赖项注入模式允许根据接口而不是具体实现来定义这些依赖项。这些接口的具体实现可以在运行时提供或注入到对象中。这样就可以根据使用该对象的上下文来改变这些依赖项的具体实现。在运行时，应用程序可以根据环境(生产环境、QA 环境和开发环境)来改变具体实现。对于单元测试，可以注入模拟对象，以代替实际实现。这样就可以使测试保持隔离和可预测性。

数据访问是业务应用程序常见的一项要求。许多开发人员都尝试在实际类定义中嵌入代码，以执行持久化操作。这样会导致数据访问代码被分散在应用程序的各个位置，增加了维护难度。

通过利用储存库模式，可以将这一功能与业务逻辑隔离。这样就在商业域逻辑和执行持久化操作所需要的基础架构之间创建明确的界限。它还简化了单元测试中对数据访问的模拟，因为这一代码不再与业务域代码紧密绑定在一起。

第 部分

将基础知识变为行动

- 第 6 章 启动示例应用程序
- 第 7 章 实现第一个用户情景
- 第 8 章 集成测试



第 6 章

启动示例应用程序

本章内容

- 业务需求如何驱动定义单元测试这一功能？
- 在开始开发应用程序之前采取什么步骤？
- 如何决定应用程序的技术问题
- 用户情景如何驱动敏捷应用程序开发方法？
- 项目组织为何很重要，在创建项目时需要记住什么？

在开始编写应用程序之前，必须完成许多任务，否则一行代码也不能写。一个应用程序是由一组功能性业务需求和一组非功能性技术要求确定的。功能性需求描述了要满足业务的需求，应用程序应做什么。非功能需求则描述了一组条件或形参，应用程序必须满足这些条件和形参。收集这些信息之后，利用功能性需求来创建用户情景，这些用户情景被分解为一些功能，并将其分配给开发人员来构建。非功能性需求用于确定应用程序平台和结构，以及应当在应用程序中使用哪些框架(如果使用)。

本章将介绍在编写应用程序之前需要执行的各种步骤和做出的决定，和所使用的敏捷实现。还会介绍在开始讨论技术主题之前要做出哪些决定，以及在做出这些决定之后如何选择正确的技术。最后，将会介绍如何开始启动项目：在 Visual Studio 解决方案中创建示例应用程序所需要的基本项目。

6.1 定义项目

大多数开发人员喜欢编写代码。在大多数情况下，这是使我们成为开发人员的第一事务。大多数开发人员尝试用代码解决给定问题。许多开发人员用代码交流。一些开发人员甚至用代码进行思考！如果可能，我相信大多数开发人员会尝试用代码进行所有交流、解决所有问题。遗憾的是，还有一些业务不能用代码交流，更不要说用代码思考了。

没有需求就不会编写任何应用程序。即使是要编写一个用于记录所收集 DVD 的小应用程序，也需要需求收集阶段，当然它可能不是非常正式的。在编写应用程序之前，需要知道应用程序要用来做什么。除了知道应用程序要做什么之外，还必须知道如何建立应用程序，以及它将必须在什么条件下运行。所有这些决定都应当在开始开发之前做出。

应用程序的业务功能由业务需求定义。应用程序的技术基础由目标环境以及构建应用程序的技术定义。在理想情况下，所选的技术选择应当有益于业务，而不应当让业务去屈从于技术。遗憾的是，理想情况极少见。因此，重要的是尽早考虑业务需求，由它们来驱动技术需求。

目标环境定义了应用程序执行任务所需要的资源，以及应用程序必须在什么限制条件下工作。这一环境的定义非常重要，可以确保应用程序在部署时拥有必要的资源，而且其设计与建立方式使它不会试图超出它的目标。选择用于构建应用程序的技术也非常重要。很多情况下，可能已经做出了决定；我们所在的公司可能已经设立了标准：要使用.NET 的某个版本和一些应用程序框架。在其他一些情况下，需要考虑新的框架，或者升级到.NET 的最新版本。这些因素对于如何设计和建立应用程序有着非常重要的影响，这些决定越早做出越好！

6.1.1 开发项目综述

作为业务开发人员，大多数时间都用来建立那些能够满足特定业务需要的应用程序。该应用程序可能用于简化或自动化一个现有过程，也可能是帮助公司利用新的收入来源。任何情况下，都必须收集一组高级需求，以标识应用程序的业务需求。

项目综述是对应用程序业务需要的高级描述。它描述了应用程序的整体目标，部分关键工作流，以及应用程序的主要用户角色和类型。该综述用于提供业务需求的骨架结构。这里没有进行详细设计，也没有创建具体的用户情景，其思想就是简单标识该应用程序的主体部分，以便能够在下一阶段对其进行分解。

接下来的几章将描述一个办公室供货管理系统的构建过程，其代码可以从 www.wrox.com 下载。该应用程序简称为 OSIM。作为 OSIM 项目综述的一部分，我已经标识了该应用程序的主要业务需求：

- 该应用程序必须跟踪当前库存。

- 该应用程序必须允许用户向库存列表中添加新类型的物品。
- 该应用程序必须允许用户记录向库存中补充的物品。
- 该应用程序必须允许用户记录从库存中提出供使用的物品。
- 用户必须通过验证。

注意，该列表并没有指出应当如何完成这些任务。它没有解决任何仅与技术相关的问题。事实上，可以用一个锁在文档柜里的记事本来满足所有这些业务需要。项目综述的重点不是开始使用技术来解决业务问题。其目的是标识业务需求，同时从客观的、不偏不倚的角度来看待将要采用的技术。如果在项目的这一阶段涉及技术，其缺点是所涉及的人员会在潜意识中使需求屈从于心中已有的某一技术或解决方案。这与应当做的事情正好相反：技术应当为业务需要服务，而不是相反。

6.1.2 定义目标环境

业务需要经常会对应用程序的目标环境产生影响。考虑到应用程序的部署环境时，需要做的不只是考虑服务器将要运行哪种操作系统，还需要考虑将会如何使用应用程序。它是一个基于 Web 的应用程序吗？它是在用户的桌面上运行吗？它可能在移动平台上运行。是否需要将 Web 服务部署为应用程序的一部分？

还有一点也非常重要，就是考虑将如何构建应用程序以及将针对它收集哪些类型的基础设施架构需求。该应用程序是否有多个层？希望在峰值时有多少并发用户？应用程序需要提供哪种响应时间？应用程序是否需要扩展？希望它进行垂直扩展(转移到一个更强大的服务器)还是水平扩展(将应用程序部署到更多服务器上)，或者同时在两个方向上扩展？这些需要如何影响应用程序的体系结构？

应用程序的目标环境应当在应用程序的非功能需求中描述。有些情况下，目标环境事先已经定义好，应用程序的设计必须量体裁衣。有些情况下，则可以创建一种新的环境，但也可能存在其他一些约束，如预算、公司硬件及/或软件标准。定义应用程序的目标环境时，所有这些都应当考虑在内。

对于 OSIM 应用程序，目标环境必须能够容纳一个多层应用程序。作为技术需求的一部分，业务拥有者已经规定：他希望应用程序有一个基于 Web 的前端、一个基于 Windows 客户端的前端、一个 Web 服务接口(供给外部贸易合作伙伴和各种现有企业报表工具使用)。该应用程序包括用户接口层(包括所有应用程序外部的 Web 服务)、业务逻辑层和数据存储层。由于这是一个在部门之间使用的小型应用程序，因此可以预期并发用户不会超过 2~4 个；但是，这种分层的体系结构为该应用程序提供了扩展空间。基于此，将目标环境定义为 Microsoft Windows 服务器，在其上运行 IIS 7 或更新版本(不一定是专用服务器)和 Microsoft SQL Server 2008 R2。

6.1.3 选择应用程序技术

定义目标环境之后，必须确定应用程序平台。有许多平台可供选择。J2EE、.NET、Ruby on Rails 和 PHP 是用于创建 Web 应用程序的最流行平台。在定义目标环境时所做的选择为选择应用程序平台提供了一些指导，因为有些服务器操作系统更擅长支持特定的应用程序平台。

在选择应用程序平台时，除了操作系统，还需要考虑其他一些问题。对于特定平台，开发人员是否已经拥有足够的知识和经验？是否已经对特定平台进行了大量投入？哪种平台最能满足所构建应用程序的需要？最后，希望选择一种拥有美好未来并能持续支持的应用程序平台。

在选择好平台之后，仍然要面对许多决策。应用程序是否使用对象关系映射器(Object Relational Mapper, ORM)？如果使用，使用哪一种？应当使用哪种依赖项注入(DI)框架？单元测试框架呢？哪一种最适合自己的需要？Web 框架又怎么样呢？对这些问题的回答非常类似于对应用程序平台相关问题的回答：找出一种最适合自己需要的、拥有良好支持、具有美好前景的工具。

该 OSIM 应用程序将使用几个框架。对于本书，主要关注的框架是 nUnit、nBehave、Moq 和 Ninject。出于说明性目的及保持一定程度的完整性，也列出了其他一些框架。使用这些框架不需要有先验知识。

6.2 定义用户情景

在敏捷开发方法中，用户情景定义了应用程序的规则和工作流。这些用户情景由业务需要驱动，成为应用程序的功能性需求。在收集用户情景时，它们被分解为应用程序功能，再分配给开发人员，开发人员用它们来定义测试，以驱动开发工作。如需有关敏捷方法的背景知识，请参阅 1.2 节。

6.2.1 收集情景

用户情景表示应用程序的业务需求。它们应当描述业务用户与应用程序之间的预期交互。收集用户情景的操作大体与 6.1.1 小节中描述的行为相关。因此，这时从广义上定义用户与系统之间的交互时，应当尽量不要确定技术。

收集创建这些情景所需要的信息，然后将其与业务比对以验证它们，这可能是应用程序开发项目中最困难的任务之一了。开发人员倾向于首先考虑技术。业务用户对技术的精通程度通常不如开发团队。为了与业务用户互动，非常重要的一点是，在与他们交流时，所采用的语言、设置和方式应当使他们感到自在。因此，如果团队拥有业务分析师(BA)，强烈建议利用他的技巧来收集用户情景。

**提示：**

遗憾的是，并不是所有开发团队都有业务分析师。在这些情况下，需要由架构设计师、首席开发人员，甚至是开发团队的普通成员来与项目经理(PM)合作，以与业务人员沟通，并定义用户情景。在这一阶段，最重要的就是以某种方式与业务人员沟通。

有许多技术可用于与业务人员交流并获得创建这些用户情景所需要的信息。一些技术是非常主动的，有些则比较被动。3种最常见的技术(从主动到被动)为：跟班、客户会谈、联合应用程序设计(JAD)会话。

客户会谈很有用，它们允许开发人员(或项目经理)一对一地与业务用户交流。客户会谈只是一个机会，可以与业务用户一起坐下来，在不会分神的自由环境中(或者是尽可能不会分神的环境中)，向他们询问有关应用程序的问题。这些会谈可以是一对一的，也可以是集体讨论。客户会谈可以是正式的提问-答复会谈，也可以是非正式交谈，也可能是现有系统的业务用户进行一些演示。在许多情况下，他们会给开发人员提供机会，让他们看看业务用户当前是如何工作的。这样可以为开发人员提供一些非常宝贵的信息，在他们创建用户情景时会很有帮助。客户会谈还有一点好处：对业务用户的影响相对较低(记住，除了帮助你创建用户情景之外，他们都还有自己的工作要做)，在日程安排方面也有很大的灵活性。过多地依赖客户会谈也有一个问题，那就是经常会从不同用户那里获得互相冲突的信息。在这种情况下，要与提供冲突信息的业务用户进行再次接洽，以尝试达成一种解决方案，这一点非常重要。

客户会谈在收集用户需要方面是一种很好的媒介。与之相对，可以在一段时间内跟随业务用户的工作。所谓“跟班”就是在一段时间内跟着用户，观察他们如何完成工作。通过“跟班”，可以学到很多有关业务问题领域的知识；有时开发人员甚至能了解到在客户会谈中永远不会遇到的东西。另一方面，“跟班”对于业务用户和开发人员可能都是极不方便的。另外，如果允许开发人员跟随业务用户会产生安全风险，使业务用户必须执行一些操作，那就会出现安全问题。例如，护士会处理病人的隐私信息，并为病人提供治疗。根据HIPAA规则，不得将开发人员或任何其他人员置于这种可能在无意间知晓病人隐私信息或病人疾病及治疗的场所。“跟班”可以在极端情况下使用，但别忘了它可能导致的不利影响。

还有一种方式就是在大型的JAD会议中一次性聚集所有业务用户并收集一组用户情景。这些JAD会议可能会持续几天，涉及所有业务用户，如果是大型系统，则可能是所有业务用户群的代表。JAD会议通常会在办公场所之外举行，以尽量避免分神。JAD会议的好处在于所有的不同业务单位都会出席，在客户会谈或跟班之后仍然可能存在的冲突，可以马上得到解决。该过程还有助于业务用户理解业务其他部分的需求，以及这些部分与自己所属部分有什么关系。JAD会议对业务用户的影响可能非常大，因为可能需要他们花

费几天的时间与开发团队合作，以确定用户情景，在此期间无法从事自己的工作。由于业务用户的具体工作不同，这种做法有时可能是不合实际的。有一种方法可以部分缓解这一问题，那就是分段进行 JAD 会议，而不是连续举行。

用户情景本身应当是用来描述用户与应用程序的交互。应当尽量使各个用户情景之间相互隔离。但为了保持用户情景的简短和全面，在一个用户情景结束或分支的地方开始另一个用户情景，也是可以接受的。总而言之，用户情景应当便于业务用户和开发人员理解。它们应当增加系统的业务价值，当开发团队完成了满足用户情景所必需的功能之后，应当立即可供用户测试。

所有这 3 种方法的目标都是收集一组用户情景。例如，OSIM 应用程序有一种用户情景，称为“登入现有物品的新库存”，其内容如下：

- 在接收到新的库存并对其计数之后，经过认证的用户使用 OSIM 登录页面登入 OSIM 系统。用户登入之后，选择导航菜单中的“登入新库存”选项。应用程序将用户带到添加库存页面。用户从物品类型列表中选择库存类型，并在数量文本框中输入收到的数量。随后单击“保存”按钮，库存数更新，页面控件被清空，以便用户输入更多库存。
- 如果物品类型列表中没有该物品类型，则必须添加新的物品类型。参见“添加新物品类型”用户情景。如果用户试图在数量字段输入一个非数字值，应用程序应当拒绝。如果用户单击“保存”按钮，但还没有从列表中选择物品类型或没有添加大于 0 的数量，则该页面应当通知用户：信息无效，并引导他们纠正错误，再次单击保存按钮。

6.2.2 确定待办事项表

简而言之，“待办事项表”(product backlog, PB)就是一个列表，列出了为完成该应用程序需要做的工作。一旦创建了定义业务需求的用户情景，它们就被分解为一些可以分配给开发人员来完成的功能。一些开发团队选择将各个功能放到一个待办事项表中。而其他团队则将所有用户情景放在待办事项表中，允许开发人员为用户情景开发自己认为最合适的各个功能。方法没有对错之分，只是对于任何给定项目来说，哪些工作最适合开发团队而已。

一项功能就是一个工作单元，通过满足功能性需求且不违反任何非功能性需求而向系统提供价值。多个用户情景可能引用同一功能。例如，多个用户情景可能都涉及用户登录应用程序的功能。在页面上创建日志可能是一个由这些用户情景派生的功能，但每个用户情景应当并不需要在页面上拥有自己的日志。对于每个涉及页面登录的用户情景，页面登录功能都可以满足其需求。不需要为每个用户情景都创建独立功能。

功能的另一个重要方面是可以对它们进行测试。应当有某种方式可用来验证这一功能能够正常工作，而不需要用户在调试器中运行该应用程序；或者运行一个 SQL 脚本，以确保一个实体中的值是否发生了变化。一些不会直接影响到用户对系统理解的基础任务必须

在应用程序中执行。应当尽可能多地在“零次迭代(将在 6.3 节描述)”中执行这些任务。

在项目经理和体系结构设计师的指导下，将待办事项表中的工作按业务优先级分配给用户。在大多数敏捷过程中，待办事项表中的功能是用情景或功能卡表示的，项目经理保存这些功能的主列表。尽管有许多工具可供用于维护待办事项表，但 Microsoft Excel 仍然是最流行的可用工具。待办事项表的管理将在 6.3 节介绍。

6.3 敏捷开发过程

所有敏捷过程都不同。有各种不同品牌的敏捷方法可供使用。但是，许多开发工作室已经发现，最好的做法是以这些方法作为起点，并根据自己的具体需要对过程进行修改，然后使任何单一的预打包方法保持完整。事实上，大多数品牌方法都鼓励这种做法，它们提供了一组核心价值，还提供了“追溯”机制，用于修改围绕这些核心价值的过程。最后，每种成熟的敏捷过程都会采用某种不同的方式来适应使用该方法的开发人员、企业或项目需要。

记住，本书不是有关敏捷开发的。这里讨论敏捷开发是为了给 TDD 的使用提供框架和上下文，因为大多数敏捷开发方法在很大程度上依赖于 TDD 提供的好处。

所有敏捷过程都稍有不同，但公共的主题和价值已经渗透到所有敏捷方法。有些原则定义了敏捷实务。尽管我们鼓励改变和调整敏捷过程，但最好不要太过偏离这些核心原则。

大多数敏捷过程都特别重视一条思想，那就是在很短的迭代过程中完成易于管理和理解的小型工作单元。这些工作单元应当着重于系统的一项可测试的特定功能。这种可测试性非常重要。敏捷开发中的另一项核心原则是，尽可能快速、频繁地向用户提交软件。如果一项功能不能由 QA 测试员或业务用户测试，则该功能的价值就是有问题的。

6.3.1 估计

在本质上，估计总是错误的。敏捷方法并不是要模糊或否认这一事实，而是包容这一事实。在许多早期的方法中，估计通常是由项目经理或体系结构设计师完成的。有些情况下，首席开发人员可能会参与其中，但大部分估计工作不是由实际执行任务的人员完成的。大多数人都难以准确地估计自己的生产能力。估计其他人的生产力更是毫无意义的工作。敏捷方法发展了这一思想：应当由执行任务的人来提供这些估计结果。

此外随着时间的推移，也可以根据业务问题领域及应用程序本身知识的增长对这些估计进行调整。在 1981 年，Barry Boehm 提出了一种概念，它后来成为“不确定性锥形”现象的基础。这一概念最早由 Steve McConnell 在 1997 年出版的 *Software Project Survival Guide*(Microsoft Press, ISBN: 9781572316218) 一书中用于软件开发。用于软件开发时，“不确定性锥形”指出：在项目开始时所做工作评估的不准确因素可能为 4。这意味着完成一项功能所需要的实际工作可能是实际估计值的 25%，也可能是估计值的 400%。随着项目的继续，完成一项功能所需工作的估计值会变得越来越准确。但只有在完成这一功能之后，

它才可能完全准确。

对于软件开发团队来说，这意味着在项目开始时对一个给定功能所做的估计可能会偏离高达 400%。在大多数方法中，这些极为不准确的估计值被列入项目计划中，当作福音，以后会用作衡量标准，来衡量该项目是成功还是失败。而实际上，这种估计，特别是在项目开始时所做的估计，是开发人员根据当时拥有的信息做出的最佳猜测。在大多数情况下，它是不完善的、不准确的，很可能会发生变化。

一种更符合逻辑的做法是定期对待办事项表中的功能进行重新估计。这些新估计值的基础是开发团队在一段时间内从事该应用程序所获取的实际知识。尽管这些估计值可能仍然不很准确，但要比原来的估计值更接近实际值。这一重新评估使项目经理能够改进项目计划，从而更准确地判断开发团队是否仍然在正确的轨道上。

6.3.2 迭代工作

本节的剩余部分描述我在团队中使用的敏捷方法。该过程不是我自己创建的，也不是在一夜之间创建的，了解这一点非常重要。它是许多不同开发人员和开发团队(包括项目经理、业务分析师和客户)多年参与众多不同项目的结果。该过程由于许多人的输入而发展、演进，在本书出版之后可能还会继续发展。并不是说这里介绍的过程是可用的最佳流程。而且它不是最佳流程的可能性还很大。我是公司里的一位咨询师，专门针对大量不同行业中的许多不同项目类型，为各种客户定制应用程序开发项目工作。该过程已经针对这一目标进行了调整。如果它所描述的内容与您的工作环境不一致，该流程就不适合您。它只是使用敏捷方法的一种方式，这里提供这一方式，是为了介绍敏捷方法大概是什么样子。

敏捷方法鼓励采用短时迭代。我倾向于将自己的迭代保持在一周或两周时间范围内。这一规则的例外称为“零次迭代”。“零次迭代”是第一次迭代，通常是用在收集了主要的非功能需求并定义了用户情景之后，准备结束需求收集阶段。“零次迭代”是设置迭代。在这一期间，我或团队的某位成员创建开发环境。其中包括设置团队将会用到的应用服务器，设置 QA 环境并配置连续迭代(CI)服务器。“零次迭代”中完成的其他任务包括：在 Visual Studio 中设置基本解决方案和项目，并创建一个源控件库以存储它们。作为这一过程的组成部分，还在相应的 Visual Studio 项目中引用了任意第三方案集。这样，在组成开发团队之后，可以确认团队成员拥有适当的凭据，能够在开发期间访问所需要的系统。如果有一些可用的基本屏幕布局，我会与用户界面(UI)开发人员合作，先为团队创建一些模板网页或窗体，以帮助其他开发人员快速开始开发工作。

“零次迭代”并没有固定的时间范围。有时，它可能需要几个星期的时间。而对于一些小型项目，也许在几个小时内就能把一切准备好。重要的是，在“一次迭代”(第一次严格意义上的开发迭代)开始时，已经为开发团队准备好了基础体系结构和服务。

在第一次迭代开始之前，我和项目经理与业务拥有人会谈。这里所说的业务拥有人并不是指所有业务用户；它是指业务方的一两位项目发起人或利益相关人。这种会谈的目的是计划下一次迭代的工作。除最后一次迭代之外，这种会谈会在每一次迭代结束时重复举行。

业务拥有人在项目经理和体系结构设计师或首席开发人员的建议和指导下，选定要在下一次迭代中完成的工作。业务拥有人将获知项目团队下一迭代的“生产时间”。所谓生产时间，是指开发人员编写代码所需的小时数。我是通过以下方式得出每位开发人员的生产时间数的：取得开发人员正常工作安排中的小时数(例如，如果持续两周迭代一个，则开发人员的正常工作时间应当是 80h)，然后减去预计这位开发人员不编写代码的时间。这些时间可能包括节假日、会议或在执行该项目期间的任意其他时间开销。另外，如果开发人员是新手或者刚接触该项目，考虑到其学习曲线，还会从他的生产时间数中扣除一些时间。

客户不要将非生产时间等同于被浪费的时间，这一点非常重要。对于我来说，由于我的业务拥有人就是客户，所以我可以向他们解释，不用为节假日埋单。我还提醒他们，尽管开发人员的主要目的是编写代码，但有关项目的其他一些不涉及编写代码的任务也是必不可少的。对于需要有学习了解过程的开发人员来说，我会解释：业务拥有人为这些资源较少付费。可以合理地推测，由于这些开发人员的经验较少，所以与资深的开发人员(当然收费也较高)相比，他们完成一项任务需要的时间要长一些。

只有当业务拥有者拥有可供使用的生产时间时，才允许他安排功能或情景的日程。例如，如果团队共有 100h 的生产时间，而业务拥有人已经安排的功能时间为 95h，他们可以选择待办事项表的任意功能组合，只要不使总安排时间超过 100 即可。这意味着他们不能选择超过 6h 的功能，使总时长达到 101h。开发人员已经接到要求，给出自己对这些功能的最准确估计值。这时应当由项目经理和技术领导人为团队提供支持，在这些数字上保持强硬立场。记住，团队的责任是完成这一迭代中安排的所有任务；不要因为安排过多的任务而妨碍他们。

经常发生的情况是：业务拥有人无法找到一些替换功能，使生产时间数恰好合适。这并不意味着这些时间就会浪费。业务拥有人可以选择一些放在“停车场”的功能，即这些功能本来并没有安排在这一迭代周期内完成，但如果有富余时间，业务拥有人希望开发人员能够处理该“停车场”中的功能。放在“停车场”中的功能应当是小型的、优先级较低，因为在当前迭代周期中可能无法完成它们(事实上，它们可能需要经过几个迭代周期才能完成)。业务拥有人需要理解：尽管在迭代周期中安排完成的功能是开发人员的任务，但放在“停车场”中的项目却不是。

我的迭代周期总是从星期一开始。由于我希望让我的迭代周期为一周或两周，所以如果一个迭代从周一开始、到周五结束，那对它的管理会更轻松一些。如果星期一是假日，迭代也仍然从星期一开始，但会把假日考虑在内，从每个开发人员的工作时间中扣除一天的小时数。在迭代开始之前的星期五，我会召开一次开发团队短时会议(不长于 15min)，简要查看一下要在下一迭代安排哪些功能，以及会分配给谁来完成。这种会议的目的不是发现和解决与这些功能相关的问题；只是要让每个人都了解下一迭代周期中安排了哪些内容，应当由谁来完成。与具体功能或情景相关的问题、观点或关注事项等，都在会后讨论，使非相关人员可以回去继续工作。

6.3.3 团队内部交流

本节重点介绍了这一过程的一个关键部分：与团体会议相对的个人交流。大多数开发人员都会畏惧会议。这些会议被认为是浪费时间，尤其是将一大堆人召集在一起，讨论一个他们认为实际上只与一两个人相关的主题时。对于这一问题，更快速、更高效的处理方法是鼓励两三个对这一主题感兴趣或受其影响的个人，进行一次专门对话，并尝试让他们认为有必要参与讨论的人加入讨论。

有一种观点认为，一个团队中的所有成员都应当知道任意时刻的团队动态。但我不同意这种观点。首先，即使是一个很小的团队，也不太可能让每个人都了解每个用户情景、功能、非技术需求、所做决定的全部细节。期望开发人员能够在任何时间都记得所有这些信息是不合情理的。我发现一种更好的做法，就是为项目准备一个“维基”(wiki)。

维基是一个可以由用户更新的网站，可以快速、轻松地将它用于存储项目的有关信息。当开发人员集体开会和做出决定，或者为问题制定解决方案时，会将自己的发现和决定放在维基上。由于维基是可搜索的，团队中的其他开发人员在应用程序开发期间遇到问题或者希望了解做出决策的原因时，可以查询维基。小范围的讨论再加上维基的使用，可以显著减少低效冗长的会议。

6.3.4 零次迭代：第一次迭代

在迭代首日的早晨，开发人员选择一项分配给自己的功能或用户情景开始工作。分配给开发人员的功能或情景可能不止一项，究竟按什么顺序来完成它们则取决于个人。在安排任务进度时，会花费很大的精力将潜在瓶颈降至最低，但有时仍然会出现这种瓶颈。在这些情况下，鼓励开发人员之间讨论完成任务的顺序，以确保不会有人过长地等待其他人。有些情况下，如果开发人员之间的关系非常密切，可以选择结对完成工作。

在开发人员选择了要开始处理的功能之后，下一步就是阅读用户情景，并与业务分析师、项目经理或其他向开发团队展示业务的相关专家会谈。向开发团队宣讲业务的角色非常关键，不应空缺。开发人员必须与这个人会谈，对功能进行回顾，以确保自己能够从业务的角度理解功能或用户情景。应用程序中有许多缺陷的出现，只是因为开发人员认为自己理解了业务需求，但实际上并没有理解。

当相关业务专家和开发人员就如何从业务的角度来开发功能或用户情景达成一致时，开发人员与该项目的体系结构设计师和首席开发人员会谈。这种会议的目的是让开发人员解释他计划如何设计和开发用于实现该功能的代码，并确保它能够在该应用程序的更大型版本中使用。无论谁来编写代码，重要的是应用程序的代码和功能要保持一致。这种讨论的目标就是帮助确保这种一致性。

6.3.5 零次迭代中的测试

在开发人员与相关业务专家、体系结构设计师或首席开发人员会谈并理解了功能背后的业务需求，以及如何让这些功能容纳于更大型的应用程序之后，就开始进行开发。在使

用测试驱动开发时，他首先为自己的功能编写单元测试。开发人员编写单元测试之后，编写出能使该测试通过的最少代码。如果测试通过，开发人员继续编写单元测试，然后再编写代码使测试通过，直到他使该功能所有需求的单元测试都通过为止。在此过程中，开发人员应当运行整套单元测试，以确保新代码没有破坏应用程序的任何已有功能。最后，开发人员编写集成测试，以确保他为自己分配到功能所编写的组件能够与应用程序的其余部分正确集成。只要通过了所有测试，开发人员就将自己的代码交付给主开发部门，标志着这一功能已经完成，可以让质量保证部门进行测试了。

在我的开发工作室中，有一个 CI 服务器，用来监看向主开发部门的提交操作。在发生提交操作时，CI 服务器生成应用程序，并运行所有单元测试和集成测试。如果有任何单元测试或集成测试失败，则该次建立被标记为失败，开发团队会得到通知。当建立过程失败时，首要的事情是修复代码，使建立内容和过程变好。开发团队可以切换优先级，共同协作来解决任何问题，尽可能快速地将建立过程恢复为成功状态。

如果建立过程成功，则将编译后的应用程序部署到质量保证环境。质量保证团队会得到通知，知道新的建立版本中修正了什么功能或缺陷。由一位质量保证测试员验证这一功能是完整或正确的，或者验证缺陷得到了纠正。如果功能不完善，或者缺陷仍然存在，质量保证测试员会向开发人员通报自己的发现。

如果质量保证测试员感到满意，认为这项功能是完整的、正确的，或者所关注的缺陷已经被修复，则将该功能标记为已完成，或者将缺陷标记为关闭。此建立结果将被提交给客户质量保证环境(可以供客户评估应用程序的环境)。只有质量保证测试员可以将功能标记为完成或将缺陷标记为关闭状态。项目管理员或体系结构设计师都不能改写这一决定。如果出现分歧，相关各方可以讨论相关事宜。但是，质量保证人员保留决定哪些完成、哪些未完成的权力。

6.3.6 结束迭代

由于我的迭代周期为一两周，而且它们总是从星期一开始，所以我知道星期五是迭代周期的最后一天。通常会在迭代结束后的星期一或星期二安排一场有业务人员参加的“展示 - 告知”会议。这是向客户展示在上一个迭代周期内所完成工作的机会。对于开发团队来说，这还是一个非常好的机会，可以与业务用户互动，以更好地理解业务需求。

这种会议对于项目的成功极为重要，不应忽略。敏捷方法的主要支柱之一就是由业务用户提供的快速反馈。有一点非常重要，就是让业务用户知道他们看到的不是已经完成的应用程序，而是一项正在发展中的工作。让业务人员查看这样一个还未完成的应用程序，主要是判断开发团队是否已经正确地将业务需求转换为应用程序。如果应用程序不满足业务需求，那就可能出现两种结果：缺陷和修改。

缺陷是指所规定的业务需求与应用程序行为之间的差异。缺陷可能有各种不同情况，既可能非常微小，比如网页上的标签拼写错误，也可能非常大，遗漏了业务工作流程中的一步。相对于在 6 个月之后才发现的缺陷，甚至更糟糕的、在应用程序已经投入生产之后才

发现的缺陷,那些能够马上发现的缺陷在纠正时要容易得多,成本也要低得多。在这种“展示 - 告知”会议上发现的缺陷总处于最高优先级;开发团队在继续后续开发之前需要先修正这些缺陷。

“改变”是指应用程序反映了业务需求,但业务需求因为某种原因不够准确的情况。这可能是需求收集阶段的缺陷所导致的。也可能是因为在收集需求之后业务发生了改变。还有可能是业务部门在创建业务需求时没有将某些因素考虑在内。无论是哪种情况,这些改变都可能创建新的用户情景,或者改变现有用户情景。这些新的或修改过的用户情景需要验证,分解为功能,然后还必须预计其工作量。这些新的用户情景和功能被添加到待办事项表中,安排在随后的迭代周期中完成。

只要客户或业务用户还有剩余工作和资源来继续开发,迭代周期就仍然会继续。作为咨询师,我发现客户经常会在待办事项表中仍有一些未实现功能时选择停止开发。这是因为在为应用程序收集需求时,业务用户,甚至包括一些开发人员,所创建的需求文档与其说是实际业务需求列表,不如说是愿望列表。许多功能如果能有当然非常好,但对于应用程序的功能来说又显得太苛刻了。在要求将这些功能排入日程并将预算提交给客户时,许多客户都选择放弃这些功能。他们可能没有看到这些功能的价值;或者可能希望将这一功能推迟到下一预算年度;也可能只是认为这一功能不值得花费那么多钱去创建。所有这些理由都是可以的。敏捷方法的强大之处就在于能够将重点放在向应用程序中添加了哪些价值,并对那些不能增加价值的内容做出合理的、深思熟虑的判断。

6.4 创建项目

不应当把 Visual Studio 中的解决方案及项目的结构认为是理所当然的。好的组织可以给你充分的自由,而确保开发资产的组织方式合乎逻辑且前后一致则是很重要的一点。项目组织是应用程序体系结构的第一道防线,因为项目结构可以帮助定义应用程序的组成部分,并驱动其发展。项目的组织结构应当符合逻辑,它应当知道指定组件存在于何处。这种清晰性可以帮助开发人员了解如何划分组件,以及如何找到应用程序中由其他人创建的组件。

6.4.1 选择框架

软件开发涉及许多重复性任务。其中有许多任务,尽管对于应用程序的功能和成功非常关键,但实际上不认为它们会增加业务价值。数据持久化、登录和依赖项注入等功能都是应用程序所依赖的内容。但从业务的角度来看,他们并不关心编写有关数据访问的代码时投入了多少精力,因为这一工作并不直接解决业务问题。这种代码被称为“管道代码(plumbing code)”,开发人员中流传的“不要做管道工”一说就是由此而来。

为了完成那些重复性、非常费时或难以实现的任务,开发人员通常会依赖于框架。框架就是一个或一组库,可以在应用程序中利用它(们)来完成各个应用程序中都比较类似的

任务。例如，大多数.NET 应用程序都会用到某种数据库。访问数据库和处理数据持久化的代码不会增加任何业务价值，而编写这类代码可能需要花很长时间。这些时间本来可以让开发人员编写一些能够向应用程序增加业务价值的代码。我们可以不再重复编写相同的模板数据访问代码，而是使用持久化框架来处理数据持久化等琐碎事务。

有数以百计的框架可以用来处理各种开发任务。如果发现自己总是要编写相同代码来解决项目中一个又一个相同问题，则可以用框架来完成这一任务。框架可以用在应用程序的任何适当位置。尽管框架通过自动实现常见任务而提供了强大功能，但它们通常是为完成一件具体事情而设计的。不要让一个登录框架去做持久化框架所做的事情，那不是它要做的。

在为应用程序选择框架时，必须记住几个要点。你正在考察的框架是否提供了所需要的功能？这看起来似乎是一个很明显的问题，但许多开发人员在使用框架时，用来完成的任务都只是与其设计用途“类似”。有时这样做是可以的；大多数框架都是开源的，可以对其进行扩展或修改，以满足自己的需要。但如果能够尽可能确保框架中准备使用的类与自己的需要非常接近，则会使改进工作更容易。

选择提供支持的框架。许多框架都是开源项目。对于这些开源框架，要确保有大型的、活跃的社区参与添加新功能、修改缺陷和提供培训。如果框架是购买的第三方工具，则确保它来自一家信誉良好的公司，能够提供参考和一些在线教程或支持。

最后要考虑学习曲线。您可能有一些熟悉 DI 框架结构图的团队成员。如果是这种情况，则不要坚持让他们使用 Ninject。团队已经有了相关知识，使用它就是了。如果他们对这两种工具都不熟悉，则让团队的一些高级成员来研究每种框架，并告诉你他们认为哪一种方法更易于团队使用。没有什么行为能比被迫使用自己不喜欢、不理解的工具更打击团队士气了。

不选择某一特定框架是有理由的。如果选择某一框架的主要论据是“它是新的，而且很酷”，那可能需要重新评估自己的决策。新框架通常是令人兴奋的，会提供旧框架所不具备的一些功能。当然有许多新框架都非常好用。但要确认，这种框架除了能够让你少一些无聊或者在简历上多写一项技能之外，还能满足你的实际需求。

对于 OSIM 应用程序的开发，我已经选择使用.NET 4.0 作为应用程序框架。我采用这些框架来辅助开发以下内容：

- **ASP.NET MVC** 为.NET 设计的 Web 框架，允许使用模型/视图/控制器(MVC)模式来创建网站。
- **Automapper** 一种开源框架，可以让我们很轻松地将应用程序内部使用的实体对象映射到将由前端使用的数据传递对象(DTO)。
- **WPF** 用于开发 Microsoft Windows 客户端应用程序的框架。
- **WCF** 用于在.NET 中生成 Web 服务的框架。
- **Fluent NHibernate** ORM，充当应用程序中的持久化(数据访问)层。
- **nUnit** 单元测试框架。

- **nBehave** 业务驱动开发(BDD)命名库，可以让我们以更流畅、与业务用户更友好的方式编写测试。
- **Moq** 模拟框架。
- **Ninject** DI 框架。

该列表包括许多非微软框架和工具。这里使用这些框架的原因是：尽管它们不是来自微软，但都很流行，在 TDD 实践中非常可能遇到。微软提供了许多很好的示例，说明如何将这些框架结合在一起。在人们熟悉的众多资源中，只有少数几个可以用来解决大量非微软框架的集成问题。

另外，尽管微软在创建 ORM 框架(实体框架)、单元测试框架(MS Test)和依赖项注入框架(Unity)等方面非常出色，但我选择在这里及日常开发中使用的框架都有超越微软相应产品的优势，而且在我的开发工作室中能够更好地工作。如果你希望使用与这些框架相对应的微软产品，在开发软件或实践 TDD 时也没问题。尽管本书中的示例没有一一转换为使用微软框架，但其理念仍然是适用的。

6.4.2 定义项目结构

本小节描述如何定义 Visual Studio 项目的结构。和前面描述的方法类似，这个结构是许多开发人员从事许多项目之后得到的产物。我们的团队提出减少问题的思路，进行尝试，然后评估结果。需要经历多个项目，经过数年的时间才使这一结构达到了现在的精炼程度，而且和我们的方法一样，但仍然会根据需要对它进行修改。

许多开发人员没有意识到的是：项目结构实际上存在于两个地方：Visual Studio 解决方案中项目、文件和对象的结构；项目在文件系统中的安排形式。项目在文件系统如何安排与如何定义 Visual Studio 解决方案的结构一样重要，因为许多文件和资产都是项目的组成部分，但不位于 Visual Studio 解决方案中。如果正在开发一个具有许多客户图片的营销网站，可能会选择不把这些图片存储为 Visual Studio 解决方案的一部分，以降低内存使用率。第三程序集也可能成为难以管理的资产。如果能够采用一种一致方式来安排所有项目文件，并使源代码库反映这一结构，就能使开发人员很轻松地找到文件和其他资产。在它的帮助下，一旦将应用程序从源控件库转移到本地驱动器之，开发人员就能快速进行设置并运行它。

1. 组织项目文件夹

在我的本地硬盘上有一个 DevProjects 文件夹，用来保存我的所有应用程序，既包括 .NET 项目，也包括其他平台。作为咨询师，我有许多客户，在 DevProjects 文件夹中，我为每位客户准备了一个子文件夹。在开始一个新项目时，我会进入 DevProjects 文件夹中该客户的子文件夹，为该项目创建一个文件夹。将我的解决方案绑定到源控件服务器时，我会在该项目文件夹级别完成这一工作，使它里面的所有子文件夹都包含在源代码库中，保持被跟踪状态。在项目文件夹中，我总是至少另行创建两个文件夹，名字分别为 src 和 libs。

src 文件夹包含该项目的源代码。我告诉 Visual Studio，到时候就在该文件夹中创建它的项目。该文件夹的结构将通过 Visual Studio IDE 管理。我将在我的 Visual Studio 项目中创建文件夹，Visual Studio 将在文件系统上创建文件夹。Visual Studio 项目的所有组成部分，也就是我可以通过 Visual Studio “解决方案资源管理器”访问的文件或资产，都放在 src 文件夹中。

libs 文件夹包含了所有第三程序集及它们可能需要的支持文件。当我需要向 Visual Studio 项目中添加第三程序集时，首先将它和它所需要的支持文件复制到 libs 文件夹中。接下来，进入 Visual Studio，添加一个引用，指向我刚刚复制到 libs 文件夹中的程序集。由于该项目是在项目级别绑定到源控件库的，因此 libs 文件夹中的一切内容都被包含在源控件中，进行跟踪。因为 Visual Studio 项目中对程序集的引用使用相对路径，所以我知道无论该项目放在我硬盘上的什么位置，它都会使用其 libs 文件夹中的程序集。

当开发人员第一次将项目拖到一台计算机上时，甚至是从源控件库刷新其当前源时，他肯定能够获得建立和运行该应用程序所需要的所有第三程序集。他还知道他所拥有的程序集版本就是应用程序根据设计所要使用的准确版本。开发人员不用再担心是否安装了给定框架的正确版本，也不需要再从头至尾查看系统以找到项目正在查找的程序集，它已经在那儿了。在某些情况下，在不同 Visual Studio 解决方案中会有一些不同的项目，使用同一框架的不同版本。将程序集与使用它们的源代码放在一起，就不再需要担心应用程序会使用给定框架的错误版本。

利用这种范例，的确会拥有同一程序集的多个副本。这些程序集也的确必须存储在版本控制系统中。一些开发人员不喜欢这样做。但我发现，使这些程序集接近源代码所带来的好处，要远远胜于保存这些文件的多个副本或者将它们保存在源控制库中所带来的忧虑与关注。

有时，我会在项目文件夹中创建其他文件夹。如果该项目没有使用文档协作系统(它应当使用这种系统)，我会创建一个 docs 文件夹来保存项目文档。将这些文档放在这个文件夹中的好处在于它们与代码的距离很近，而且能够受益于源代码控制系统的版本控制功能。如果项目使用桌面数据库引擎(如 Microsoft Access)，而不是使用数据库管理系统(DBMS)，或者如果需要在项目中包含 SQL 脚本作为其组成部分，那可以创建一个 db 文件夹，用来存储这些资产。

这是两种极端情况。一般来说，项目应当使用文档协作服务器(如 SharePoint)来存储项目文档。如果应用程序使用数据库管理系统，那就降低了对 db 文件夹的需求。我反对创建这些及任意其他文件夹，因为文件夹的这种滋生状态通常是不良应用程序开发习惯的标记。

2. 创建 Visual Studio 解决方案

创建文件夹结构之后，就应该创建 Visual Studio 解决方案了。同样，Visual Studio 解决方案中有条理的结构也很重要。解决方案中的项目最终会变为应用程序中的各程序集。如果不能正确地区分这一功能，可能会生成一些过于庞大的程序集，它们不恰当地将功能混合在一起，使得难以部署和扩展应用程序。

我首先创建一个空的 Visual Studio 解决方案，如图 6-1 所示。这样做的理由是：在创建项目时，我希望它们的名称能够反映程序集所代表的命名空间。如果在创建项目时允许 Visual Studio 为我们创建解决方案，该解决方案的名称就是由项目的名称派生而来，就不能足够灵活地为各个项目命名。

恰当地给项目命名对于解决方案的组织非常重要。一个项目名称应当反映它所代表的命名空间，并描述最终程序集将提供哪些特性和功能。所有应用程序都需要一个类库，用于容纳核心或业务域服务和实体。这通常是创建的第一个项目。在本书的示例中，我创建了一个名为 OSIM.Core 的类库，尽管 OSIM.Domain 等名字也是可接受的。在创建该项目之后，删除了 Visual Studio 用以预填充该项目的 Class1.cs，并在项目中创建两个文件夹：Entities 和 DomainServices(如图 6-2 所示)。



图 6-1



图 6-2

除了保持名称的清晰之外，项目和文件夹的命名约定也反映了命名空间。例如，如果要在 Entities 文件夹中创建一个 Person 类，该类的完全限定名将是 :OSIM.Core.Entities.Person。

一家讲授 TDD 的学校说，我应当等到需要时才创建这个项目。TDD 的确在推行一种观念：不要提前编写任何代码，而应当等到为了通过一个测试而需要这段代码时才编写。推广一下，可以说：只有在需要一个项目时才创建它，这一点可能是正确的。但是，在我的团队里，我希望主要项目能够准备就位，以便在“一次迭代”的第一天，开发团队可以立即开始编写代码。这是一个有点灰色的区域，是贯彻测试先行的理念呢，还是务实一些呢，可能会有一些争执。最终，我们必须估量自己的情况，最适合自己的团队和项目的选择才是最好的。

说到测试，我知道接下来在所有解决方案中总会用到的两个项目是为单元测试和整体测试准备的。接下来就创建这两个项目，如图 6-3 所示。

在我的测试项目中，我将为解决方案中的每个项目都创建一个文件夹，以使测试保持良好的结构。当前只有 OSIM.Core 项目，所以我将为项目创建文件夹，如图 6-4 所示。

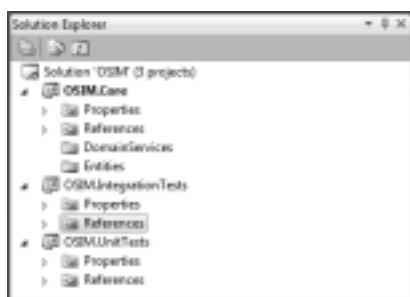


图 6-3

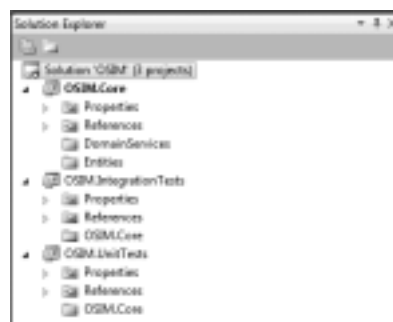


图 6-4

根据 OSIM 应用程序目前的高层设计，我知道该应用程序将有 3 个前端：一个基于 Web 的前端，在 ASP.NET MVC 2 框架上开发；一个使用 WPF 开发的 Windows 客户端前端；一个使用 WCF 的 Web 服务界面。不错，Web 服务是一种前端。它们代表一个访问点，供给外部用户与应用程序交互，这一点与网页或 Windows 窗体相同。唯一的区别在于不是由人直接与界面交互，而是需要某种 Web 服务代理来使用该界面。我们将为其中的每一个界面创建适当的项目类型，如图 6-5 所示。

现在我的解决方案增加了三个项目，应当在测试项目中为它们添加文件夹了，如图 6-6 所示。



图 6-5



图 6-6

现在，根据最初的高层设计，应用程序所需要的所有项目已经都有了。以后，我可能会在项目中发现还需要一些用于支持目的的其他项目。在创建这些项目时，第一个要问的问题就是“它真的属于一个独立项目吗？”许多开发人员会以提高精细粒度的名义说服自己创建新项目。有些情况下，其结果是得到两个高度相关的程序集，单靠其中一个几乎提供不了什么价值。这种情况下经常会出现错误，因为在尝试用其中一个程序集部署应用程序时，可能会误用另一个程序集。如果两个或多个类确实需要彼此才能发挥作用，就不应将它们分割到不同项目中。

如果真的需要创建一个独立的程序集，要确保命名约定仍然能够表明这个新项目正在支持哪个项目，以及如何支持。例如，如果希望创建一个项目以包含 Windows 客户端项目

的主题，比较好的名字是 OSIM.WinClient.Themes.

到现在，该项目结构已经完成了。我知道随着项目的进展，还要添加结构元素，但不希望在刚开始启动应用程序开发项目时就使环境过于复杂。通常，都希望项目结构尽可能保持简单、整洁，只有在需要时才进行修改和补充。

6.5 本章小结

在应用程序开发项目中，开始编写代码之前必须完成许多任务。第一个任务就是收集业务和技术需求。在敏捷方法中，这包含高层项目概述，用来确定应用程序的规模和范围。敏捷方法不再沿袭大型前期设计的概念，而只是获得其大体构思。详细设计将被推迟到为用户情景或功能安排日程时进行。

收集高层需求之后，必须定义一个目标环境。这个环境将用于确定所开发应用程序的类型，有助于排除许多非功能性技术需求。定义这个环境之后，必须选择各种应用程序技术。

业务需求用于创建用户情景，它定义了应用程序的业务价值来自何处。用户情景应当是不依赖于具体技术，而且是可测试的。否则，这些需要可能会为了适应这些技术而被扭曲，这是与其目的相悖的。这些用户情景被分解为功能，并将它们添加到产品待办事项中，然后在开发项目迭代期间分配给开发人员完成。

在使用敏捷过程开发应用程序时，迭代周期应当比较短，其最终的结果应当可以提交给业务部门。确保业务部门了解：这不是一个已经完成的系统，而是为了对目前已经完成的工作收集反馈。如果存在缺陷，就立即纠正。如果需要改变，就创建新的用户情景，并将它们放在待办事项表中。在开发期间纠正应用程序要比在生产期间纠正容易得多。

无论是在文件系统中还是在 Visual Studio 中，应用程序的结构都很重要。它应当清晰、简单、一致。让开发人员能够很轻松地理解和使用这一结构，有助于正确地划分项目的资产，并使新接触该项目的开发人员能够更轻松地了解体系结构和基础代码。

第 7 章

实现第一个用户情景

本章内容

- 如何将用户情景分解为功能
- 在开发时，如何在编写业务代码之前先编写单元测试
- 业务驱动开发(business-driven development, BDD)命名风格是什么样的
- BDD 如何帮助编写清晰、易于理解的测试
- 单元测试如何帮助开发人员重构代码，而不用担心破坏特性或功能
- 为什么三角测试对于保证代码质量非常重要

现在，办公室供货管理(office supply inventory management, OSIM)应用程序的整体范围和高层设计已经定义好，可以开始构造应用程序了。就像我在敏捷方法中概括的那样，整个构造过程始于业务拥有者选择要在第一迭代中开发的用户情景或功能。在项目经理和应用程序体系结构设计师的指导下，业务拥有者决定从以下用户情景开始：

用户应当能够向应用程序中添加新类型的物品。

从这里开始设计符合逻辑，因为这是一个库存管理系统。要管理库存(和使用可能是由其他用户情景提供的功能)，用户需要拥有一张库存物品表。

要开始工作，需要将较大的用户情景分解为较小的功能。功能应当短小、简单、独立、可测。对于当前的用户情景，我提供了以下功能列表：

- 一个物品类型实体(entity)和一个可以将该物品实体存储到数据存储中的持久化层
- 一个物品类型域服务，可以提供数据存储中的物品类型列表

- 一个用户界面，可以让业务用户列出数据存储中的已有物品类型

该功能列表列出了应用程序的一些层和组成部分，非技术业务用户需要用它们来验证该用户情景是否完整。如果该用户情景完整，非技术业务用户就能够使用应用程序创建新的物品类型，然后通过物品类型列表中查看它们来验证是否完整。这个过程不需要技术人员参与。业务用户不需要开发人员或 DBA 查询数据存储，来验证实际创建好的业务类型。业务用户通过自己的工作就能验证用户情景是完整的，能够正常工作。

现在已经定义了功能，接下来需要确定按照什么顺序来建立它们。应用程序和洋葱、多层冰淇淋果冻和 OGRE 一样，都是建立在分层思想之上的。因为应用程序是分层建立的，所以应用程序的构造应当从核心开始，然后在核心之上构建各层，再在各个层的功能之上构建后续层。和大多数业务应用程序一样，在 OSIM 的核心是数据存储。和大多数应用程序一样，OSIM 的最外层是界面，它可能是一个网页、Windows 客户端应用程序或服务层。因此，按照下面的过程来开发这些功能比较合理：

- (1) 物品类型实体和持久化层。
- (2) 物品类型业务域服务。
- (3) 前端用户界面，将允许业务用户与物品类型业务域服务进行交互。

该示例中使用 TDD 来构建应用程序，所以需要决定第一个测试应当是什么样的。

7.1 第一个测试

对于任何功能来说，选择第一个要编写的测试都是非常重要的任务。它设置了该功能的方向和基调。它还能确保该功能所需要的组件和方法按照正确的顺序构建和分层。第一个测试通常是针对所构建功能的最基本需求。同样，它也表明，在继续进行开发和重构时，永远都不会过远地偏离该特性需要提供的基本功能。

7.1.1 选择第一个测试

在选择第一个要编写的测试时，需要确保真正理解了用户情景以及它背后的业务需求。技术实现顺应业务需求而来。在大多数情况下，清晰而符合逻辑的过程决定了应当如何构建每个用户情景的功能。一般来说，每个应用程序都会在业务域层的上方有某种类型的用户界面层，用一个持久化层与位于核心的某种数据存储通信。在一些大型应用程序中，会对这些层中的一部分进行细分，或者使用外部资源，这些外部资源会创建某种分割或子分层。

无论一个应用程序的体系结构多么复杂，首先创建那些最接近应用程序核心的特性，然后向外围扩展，通常会更容易一些。这样可以使应用程序的核心尽可能简单。在该核心上方构建的层有一个简单的 API 集合可用，这样又有助于简化它们的接口。有了定义良好的核心，可以快速、轻松地开发新特性，而不用重复大量代码。

有些开发人员选择首先开发用户界面层，然后再向内层推进。这种方法在本质上也没

有什么问题。但是在为当前开发的层编写测试时，需要为当前层下方的层定义 API，并用存根方式表示它。这可能是开发的一个好方法，因为它所创建的 API 更多的是由这个库的用户根据自己的需求驱动的，而不是由那些认为自己知道下游客户需要什么功能的库开发人员来驱动。我经常使用这种方法。成功的关键在于能够回溯，为那些为之创建了存根的方法编写测试。与其说这是一个技术问题或过程问题，不如说它是一个规则问题。在制定了 TDD 中所需要的规则之后，当然可以尝试这一方法。对于一些主要由单一用户界面驱动的系统来说，其效率可能要更高一些。如果要开发的库集合将会供给大量用户使用，还是自下而上方法的效率更高一些。在任何情况下，都可以自由尝试这两种方法，选择对自己最有效的方法。

对于目前这个功能，即能够将一种物品类型存储到数据存储中，第一个测试就是要确保一个有效的物品类型实体会被保存到数据存储中。如果能够确保从持久化层返回一个有效的物品类型 ID 号，就能对此进行验证。

7.1.2 为测试命名

测试就是类的方法。因此，可以为测试命名任何名字，只要它是 .NET 中的有效方法名。但是，为了保持 TDD 的灵魂，我发现为测试起一个有意义的、具有描述性的名字会更好。为此，我借鉴了另外一种驱动方法：行为驱动开发(behavior-driven Development, BDD)。

BDD 是对 TDD 理念的扩展。BDD 强调有利害关系的技术团体和非技术团队都要参与到软件开发过程中。可以把它看成一种强调团体间合作的敏捷方法。自然，大多数采用某种敏捷方法的团队最终都会遵循 BDD 的许多原则。BDD 本身是一个很宽泛的主题，所以在本书中不详细讨论。

对于我编写的单元测试，我希望使用 BDD 风格为测试命名。这种风格强调使用完整的、描述性的、便于业务用户理解的名称，避免使用技术术语。BDD 希望类、方法和变量的名称尽可能反映日常英语。

使用这一风格时，我把对测试的描述看作一个句子。类名中包含了测试及其实现的所有基类，应当说清楚运行该测试的一组条件。对于当前特性来说，首先创建一个类，用来描述这些前提条件的最基本内容：处理物品类型库。该类将名为 `when_working_with_the_item_type_repository` 放在用于初始化可测试 `ItemType` 库的代码处。因此，`when_working_with_the_item_type_repository` 成为所有对 `ItemType` 库进行测试的单元测试的基类。

这里使用的命名风格非常重要。许多开发人员不喜欢在类或方法的名字中使用下划线字符。如果这一说法是针对最后仅供其他开发人员和技术人员阅读的生产代码，我同意。但是 TDD 的目标是确保测试反映了业务需要、需求和规则。因此在编写单元测试时，让非技术业务用户参与编码实践是非常重要的。大多数技术人员在阅读以混合大小写形式表示的类名和方法名方法(如 `GetItemsFromDatabaseByCustomerId`)，都有多年的经验。他们非常习惯这种表示方法，可以很容易地读取这些名称。但对于大多数业务用户就不是这么回事了。对于他们来说，如果名字中的单词以下划线分割会更容易阅读和理解。由于我们位于

业务的技术端，需要向业务人员传达相关内容，所以重要的是要尝试满足他们，至少也要部分满足。因此，在为测试命名时，必须采用一种更令人愉悦、更便于业务人员理解的方式。还有一点非常重要：测试类和方法的名称中要避免使用业务用户不能理解的技术术语。

基类 `when_working_with_the_item_type_repository` 不会实际包含单元测试。它只是对可执行操作的不完整描述。它的名称表明，我们正在处理物品类型库，但并没有表明用它来做什么。要完成这组条件，需要创建一个从 `when_working_with_the_item_type_repository` 继承的类，它的名称和代码都描述了其余的初始条件。这个从 `when_working_with_the_item_type_repository` 继承而来、实现单元测试的类名为 `and_saving_a_valid_item_type`。该类的名称加上它由其继承而来的基类，明确地定义了该测试的一个开始条件：在处理物品类型库并保存一个有效物品时，将会发生一些事情。这些“事情”就是单元测试文件的名称。

在为实际单元测试方法命名时，要尽力用类的名称将所创建的句子补充完整。如果要向一位业务用户描述测试的要点，可能这样说：“在处理物品类型库并保存一个有效物品类型时，应当返回一个有效的物品类型 ID。”利用这一模型，很容易就能看到单元测试方法的名称应当是 `then_a_valid_item_type_id_should_be_returned`。

7.1.3 编写测试

为了帮助自己以 BDD 风格编写测试，我使用了一个名为 NBehave 的框架(可以从 nbehave.org 下载它)。NBehave 是一个用于 .NET 的 BDD 框架，允许开发人员创建技术规范(用日常英语撰写的业务规则)，并将它们链接到可执行测试代码。这实际上创建了一种“领域特定语言(domain-specific language, DSL)”，它允许非技术业务用户根据框架来编写测试，该框架是由 NBehave 创建的 DSL 提供的。

用 NBehave 编写和执行这些测试 DSL 的功能非常强大。但是在本书中，我只是使用 NBehave 提供了一些“句法糖”，这样可以使我用 BDD 风格编写的单元测试具有更好的可读性和可管理性。在本书后面继续介绍示例时，会指出在哪里使用了 NBehave 的这些句法特性。

在第一组测试的命名机制和高层类布局准备就绪之后，就可以编写这些测试了。但在为这一功能创建测试类之前，还需要做一点“家务活”。要使用 NBehave 框架，需要添加对 NBehave 程序集的引用。具体来说，需要引用两个程序集：一个是 `NBehave.Spec.Framework`，它是基础 NBehave 程序集；另一个是 `NBehave.Spec.NUnit`，它提供“句法糖”，使 NBehave 能够处理 NUnit。还需要添加对 NUnit 程序集 `nunit.framework` 的引用，以能够编写和运行单元测试，如图 7-1 所示。

接下来需要添加 Specification 基类。该示例中的 Specification 类是由我以前工作过的一个团队开发的。它为以 BDD 风格编写的单元测试提供基类。它确保所有单元测试类都是从 NBehave 类 `SpecBase` 派生而来，所有派生类都被标记为 NUnit 的单元测试。在 `OSIM.UnitTests` 项目中创建 Specification 类，并使用下面的代码来定义该类：



可从
wrox.com
下载源代码

```
[TestFixture]
public class Specification : SpecBase
{
}
```

Specification.cs

现在已经完成了设置任务，可以开始为该功能编写单元测试了。首先在 OSIM. UnitTests 项目的 OSIM.Core 文件夹中创建一个类文件，用来保存该单元测试，将这个文件命名为 ItemRepositoryTests.cs，如图 7-2 所示。

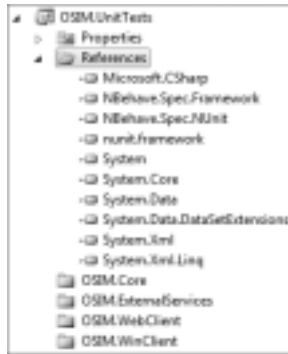


图 7-1

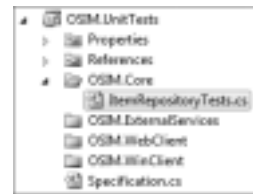


图 7-2

删除 Visual Studio 在该文件中放置的 ItemRepositoryTests 类定义，留下一个仅包含一个命名空间定义的.cs 文件：



可从
wrox.com
下载源代码

```
namespace OSIM.UnitTests.OSIM.Core
{
}
```

ItemRepositoryTests.cs

现在已经为创建单元测试类做好了准备。首先从 when_working_with_the_item_type_repository 基类开始：



可从
wrox.com
下载源代码

```
public class when_working_with_the_item_type_repository : Specification
{
}
```

ItemRepositoryTests.cs

目前，该类基只是从 Specification 继承而来，因此只能向 NUnit 表明自己是一个可能包含单元测试的类。它还继承了 NBehave SpecBase 类提供的功能。一旦开始为该示例及其他从其继承的示例编写测试，就将设置测试环境的公共代码放在该类中，其他的测试类再来继承它。

接下来，创建一个包含当前功能的单元测试的类 `and_saving_a_valid_item_type`：



```
public class and_saving_a_valid_item_type :
    when_working_with_the_item_type_repository
{
}
```

ItemRepositoryTests.cs

通常，我将一个特定类的所有单元测试保存在同一个.cs 文件中。许多开发人员更喜欢在一个文件中保存一个类。对于生产代码，我赞同这一做法(尽管它没有扩展到接口，稍后对此进行详述)。但是，对于单元测试来说，我发现将针对应用程序特定部分的所有测试按顺序放在同一个文件中要更容易。对于初学者来说，业务用户会发现当这些测试都放在相同位置时更容易浏览。这样做是有道理的，因为大多数业务用户都习惯于处理文档。如果他们在处理合约，不会把每个段落都放在一个独立的文件中，而是都放在一个地方。业务用户就是这样来考虑信息的。要部分满足业务用户的要求，需要做很多工作，而将这些测试合并在一个文件中所付出的代价最小。另外，因为计划拥有大量测试(在许多项目中，单元测试的数量会达到几百甚至几千个)，所以要将每个单元测试都放在自己的文件中是不现实的。这种做法会使 Visual Studio 中的“解决方案资源管理器”变得非常缓慢。如果单元测试的.cs 文件已经变得过大而难以管理，就要想法按某种合理方式对其进行分割。但我总是选择建立少量的大型单元测试文件，而不是大量小文件。

构造该单元测试的下一步是创建一个实际测试方法 `then_a_valid_item_type_id_should_be_returned`。将该方法添加到 `and_saving_a_valid_item_type` 类中，它的属性将其定义为一个可执行的单元测试：



```
public class and_saving_a_valid_item_type :
    when_working_with_the_item_type_repository
{
    [Test]
    public void then_a_valid_item_type_id_should_be_returned()
    {
    }
}
```

ItemRepositoryTests.cs

现在已经设置了测试类，可以将注意力转向编写测试逻辑。每个测试都由 3 个基本组成部分：设置起始条件和环境，执行被测代码，评估结果。设置初始环境似乎是业务人员的第一条命令。但要记住，TDD 的关键理念在于仅编写最基本的代码，一定要等到确实需要时才编写代码。这一惯例更多地被应用于以 TDD 编写的业务逻辑，但它可以、也应当适用于所编写的测试。在这里，希望编写的第一部分代码就是用来调用所创建功能的代码。

将这一测试分解为 3 项任务(设置环境、执行被测代码、评估结果)之后,现在需要至少 3 个方法,每个方法执行其中的一项任务。现在已经有了 `then_a_valid_item_type_id_should_be_returned` 方法,把调用被测代码和衡量结果这两项任务放在该方法中完成似乎也是合理的。但要记住 SOLID 原则,包括单一职责原则(SRP),也应当以一种注重实效的方式应用于这一测试中。因此,我们希望一个方法用于设置将在其中运行测试的上下文,一个方法用于执行被测代码,一个方法评估结果。

为了保持类和方法名的 BDD 风格,使其意义明确、不含技术术语,并为测试的各个部分创建类似于句子的结构,可能希望执行被测代码的方法尽可能具有很强的描述性。但是,由于这些方法大概会出现在几乎所有测试中,所以还需要它具有通用性。幸运的是,NBehave 提供了所需要的方法: `Because_of`。在 `and_saving_a_valid_item_type` 类中,可以创建该方法的一个重载实现方式(NBehave 类 `SpecBase` 中出现的基本方法):



```
protected override void Because_of()
{
    base.Because_of();
}
```

ItemRepositoryTests.cs

在大多数情况下,不需要调用该方法的基类版本,所以可以去除对 `Because_of` 基类实现的调用。而是希望添加的代码能够调用被测代码,并保存 `then_a_valid_item_id_should_be_returned` 方法的评估结果:



```
protected override void Because_of()
{
    _result = _itemTypeRepository.Save(_testItemType);
}
```

ItemRepositoryTests.cs

在成员变量 `_result` 中保存返回值。将被测库的实例保存在 `_itemTypeRepository` 成员变量中,并传递一个保存在 `_testItemType` 成员变量中的物品类型。目前, `and_saving_a_valid_item_type` 类如下所示:



```
public class and_saving_a_valid_item_type :
    when_working_with_the_item_type_repository
{
    private int _result;
    private IItemTypeRepository _itemTypeRepository;
    private ItemType _testItemType;

    protected override void Because_of()
    {
        _result = _itemTypeRepository.Save(_testItemType);
    }
}
```

```

[Test]
public void then_a_valid_item_type_id_should_be_returned()
{
}

```

ItemRepositoryTests.cs

现在已经定义了执行被测代码的代码，可以向 `then_a_valid_item_type_id_should_be_returned` 方法中添加评估返回结果的代码，这一结果保存在 `_result` 成员变量中：



```

[Test]
public void then_a_valid_item_type_id_should_be_returned()
{
    _result.ShouldEqual(_itemTypeId);
}

```

ItemRepositoryTests.cs

这段代码给出了另外一个示例，它利用 NBehave 提供的句法糖，使业务用户更容易阅读单元测试代码。ShouldEqual 方法只是编写第 4 章所述资产的另外一种方式。在本例中，业务用户不必理解资产是什么以及它的语法是如何发挥作用的，而是可以阅读像句子一样的代码。例如，如果大声地读出一个代码行，可能会说出这样的内容：“结果应当等于物品类型 ID。”将这一内容与从左向右读取资产时所说出的内容对比，它像是一个句子，显然要比另一个更易于理解。

现在试图运行该测试将会失败。事实上，该代码甚至无法编译。该测试执行一个接口 (IItemRepository) 的方法，而这个接口是不存在的；还试图传递一个类的类型 (ItemType)，而这个类型也是不存在的。如果已经按照顺序在 Visual Studio 中编写这一代码，现在肯定会看到很多红灯。下一步就是创建一些业务类来修正这些问题。

关于按什么顺序来创建业务类，在一本书中是说不清楚的。实际上，要视情况而定。在本例中有一个 IItemRepository，它有一个 Save 方法，以 ItemType 类型的对象为实参。在本例中，为了使涉及物品类型库的开发更容易，可以首先创建 ItemType，这样它在编写库代码时已经存在了。

在 OSIM.Core 项目中，是一个名为 Entities 的文件夹。在这个文件夹中，创建一个名为 ItemType 的类，如图 7-3 所示。



图 7-3

现在，关于哪种类型的数据应当包含 ItemType，并没有任何具体要求。但 ItemType 类必须是可持久化的(即可以将它保存到数据库)。要将一个实体保存到数据库，该实体必须有一个唯一的 ID。这样可以让应用程序以后在数据库中找到这个指定的实体。因此，以下假定是合理的：ItemType 类应当定义了一个 int 类型的、名为 Id 的公共属性。另外，Fluent NHibernate(FNH)、Object Relational Mapper (ORM)(将在第 8 章用它来处理数据持久化)需要将该属性声明为虚拟的，所以也需要进行这一声明。在第 8 章讨论 FNH 映射时，解释为什么需要虚拟关键字。



```
public class ItemType
{
    public virtual int Id { get; set; }
}
```

ItemType.cs

回到该测试，需要在 IRepositoryTests.cs 文件的顶部添加一个 using 语句：



```
using OSIM.Core.Entities;
```

IRepositoryTests.cs

该测试现在应当能够解析 ItemType 类了。下一步是创建 IRepository 接口。要完成这一任务，需要在 OSIM.Core 项目中创建一个名为 Persistence 的新文件夹，如图 7-4 所示。

该文件夹中保存了 OSIM 应用程序的持久化(数据访问)逻辑。需要创建一个文件来存储 IRepository 和具体的 IRepository 类，所以需要向 OSIM.Core 项目中的 Persistence 文件夹中添加一个名为 IRepository 的新类，如图 7-5 所示。



图 7-4

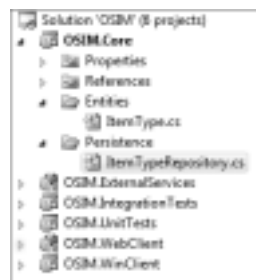


图 7-5

由于将它创建为类，因此 Visual Studio 自动包含了一个默认实现：



```
namespace OSIM.Core.Persistence
{
    public class IRepository
    {
    }
}
```

IRepository.cs

SOLID 原则指出，在开发代码时，应当关注的是合约和接口，而不是具体实现。在测试时，从 IRepository 接口引用 IRepository。因此，需要添加一个名为 IRepository 的接口，该接口将由 IRepository 类实现：



```
namespace OSIM.Core.Persistence
{
    public interface IItemTypeRepository
    {
    }

    public class ItemTypeRepository : IItemTypeRepository
    {
    }
}
```

ItemTypeRepository.cs

注意，该接口和库位于相同的.cs 文件中。这不是错误。传统的常识指出：这些实体应当放在不同文件中。但如果一个接口只是由一个类来实现，为什么还要把它们放在不同文件中呢？我发现，在大多数情况下，将接口和具体类放在同一文件中，会使这些实体的管理更轻松，不会约束处理接口或具体类的方式。有些情况下，则适合把接口放在不同文件中。事实上，有时的确应当把接口放在分离项目中的。但我发现这都是一些极端情况。如果确实出现了这类情况，将接口移到另一个文件或另一个项目中是很简单的事情。因此，在开始时，可以先为常见的主要情景进行开发，将接口和具体类放在同一文件中。

再回到 OSIM.UnitTest 项目，可以在 ItemRepositoryTests.cs 文件中添加一个 using 语句，指向 OSIM.Core.Persistence 名称空间，以允许 and_saving_a_valid_item_type 单元测试类来解析 IItemRepository 接口。下面是 ItemRepositoryTest.cs 文件的完整内容：



```
using NBehave.Spec.NUnit;
using NUnit.Framework;
using OSIM.Core.Entities;
using OSIM.Core.Persistence;

namespace OSIM.UnitTests.OSIM.Core
{
    public class when_working_with_the_item_type_repository : Specification
    {
    }

    public class and_saving_a_valid_item_type :
    when_ working _with_the_item_type_repository
    {
        private int _result;
        private IItemTypeRepository _itemTypeRepository;
        private ItemType _testItemType;
        private int _itemTypeId;

        protected override void Because_of()
        {
            _result = _itemTypeRepository.Save(_testItemType);
        }
    }
}
```

```

[Test]
public void then_a_valid_item_type_id_should_be_returned()
{
    _result.ShouldEqual(_itemTypeId);
}
}
}

```

ItemRepositoryTests.cs

在编译和运行该测试之前，最后一道业务程序是向 `IItemRepository` 接口添加 `Save` 方法。这样还需要向 `ItemRepository` 类添加一个 `Save` 方法：



```

using System;
using OSIM.Core.Entities;

namespace OSIM.Core.Persistence
{
    public interface IItemTypeRepository
    {
        int Save(ItemType itemType);
    }

    public class ItemTypeRepository : IItemTypeRepository
    {
        public int Save(ItemType itemType)
        {
            throw new NotImplementedException ();
        }
    }
}

```

ItemTypeRepository.cs

现在，`ItemRepository` 类的 `Save` 方法的实现没有做任何工作。这没什么问题。记住，就目前来说，目的是仅编写能够使测试通过的代码。到现在，还不能编译此代码，所以也还没有看到测试失败。在知道有关编写代码的需求之前，不希望编写任何代码。

当然，编译和运行该测试肯定会显示测试失败，如图 7-6 所示。



图 7-6

7.2 实现功能

尽管花费了一点时间才出现这一结果，但重要的是看到测试失败了。如果该测试得以通过，那可能有几种含义。如果通过了测试，可能表明代码没有正确地反映需求或当前正在开发的功能。测试失败很重要，因为它可以帮助确认该测试正确地引用了目前应用程序中尚未实现的功能。而如果测试通过，可能表明正在处理的功能是某种重复，或者当前用户情景的这一特定功能已经从过去的需求、功能或用户情景中得到了满足。无论是哪一种情况，对于一个没有编写任何应用程序代码就能通过的测试都要加以研究，这一点非常重要。

7.2.1 编写能够正常工作的最简单代码

因为测试失败了，所以现在的目的就是仅编写足够数量的代码使测试通过。在某些情况下这非常容易，只需要编写或修改一行代码即可。这没有什么问题。重要的是只满足当前的业务需求，不要试图猜测以后会需要什么。还有一点也非常重要，那就是让代码尽可能简单。简单代码中包含缺陷的可能性较小，以后也更容易维护。

在该示例中，为了实现这个功能，最终要做的事情比只编写或修改一行代码要稍微麻烦一些。即使我们知道自己要做很多事情才能使测试通过，也应当循序渐进、不断测试，只有在需要时才编写代码，而不是提前编写，这一点仍然是很重要的。经常执行测试(不只是自己的测试，而是所有测试)，即使是知道它不会通过也要执行，这样才能确保开发当前功能的行为没有破坏应用程序其他部分的功能。有时会影响到应用程序的其他代码。这并不总是坏事，有时甚至可能是必需的。尽快知道这一后果，特别是当这一后果与预期不同并会产生负面影响时，这有助于了解自己编写的代码如何与应用程序的其他部分交互。如果对其他功能产生了不希望出现的影响，那在知道这一后果后立即加以修正要容易得多，因为可能导致这一错误的更改列表相对较短。

根据我的结果分析，在 `ItemRepositoryTests.cs` 文件的第 22 行出现了一个 `Null` 对象错误。这一行代码在 `and_saving_a_valid_item_type` 测试类的 `Because_of` 方法中对 `ItemTypeRepository` 进行调用的部分：



```
protected override void Because_of()
{
    _result = _itemTypeRepository.Save(_testItemType);
}
```

ItemRepositoryTests.cs

测试会失败，是因为尽管已经定义了 `_itemTypeRepository`，但还没有用一个对象来填充这个变量。第 2 章介绍了 NUnit 提供的 `Setup` 属性，用于在运行每个测试之前设置执行

上下文。由于该示例使用 BDD 命名风格，并且 NBehave 为 BDD 风格的测试提供框架，因此还有另外一种选择。最终，类是从 NBehave 类 SpecBase 继承而来的，它提供了一个名为 Establish_context 的虚拟方法。Establish_context 方法提供了创建上下文的位置，测试将在该上下文中运行。因为我倾向于将单元测试类创建为由类组成的层，每一层为测试定义一个先决条件，所以比较方便的做法是在类分层结构中设置一个公共方法，用以创建执行上下文。我可能希望向一些业务人员展示我的测试，他们也比较容易理解这种设置。

为了使这个最初失败的测试得以通过，只需要创建一个 `ItemTypeRepository` 类型的对象，并把它存储到 `itemTypeRepository` 成员变量中：



```
protected override void Establish_context()
{
    base.Establish_context();

    _itemTypeRepository = new ItemTypeRepository();
}
```

ItemRepositoryTests.cs

完成这一任务之后，应当再次运行这些测试。在这种情况下，该测试仍然可能失败，但应当保留经常运行测试的习惯。最低限度是，在实现功能时，每完成一个小任务之后都要运行它们，如图 7-7 所示。



和预料中一样，测试仍然失败。但这次的失败出现在不同位置，原因也不一样。这表明已经有了进步。新的错误是因为在 `ItemTypeRepository` 的 `Save` 方法中出现了 `System.NotImplementedException` 异常，具体来说，就是在 `ItemTypeRepository.cs` 文件的第 15 行：



```
public class ItemTypeRepository : IItemTypeRepository
{
    public int Save(ItemType itemType)
    {
```

```
        throw new NotImplementedException ();
    }
}
```

ItemTypeRepository.cs

我使用 Resharper by JetBrains (www.jetbrains.com)来扩充 Visual Studio 开发环境。在由 Resharper 创建该方法时，它会自动添加这个引发 NotImplementedException 的实现。如果没有使用 Resharper ,自己创建这一初始实现仍然不错 ,用来提醒自己还没有实现该代码。要完成这一功能，需要实现该方法。现在只需要编写最简单的有效代码即可：



```
public class ItemTypeRepository : IItemTypeRepository
{
    public int Save(ItemType itemType)
    {
        return 1;
    }
}
```

ItemTypeRepository.cs

无可否认，这是能够满足测试需求的最简单代码。当然，这样一行代码不可能成为这一功能或相关用户情景的解决方案。但根据目前在这一功能生命周期中所处的阶段，把它作为下一步是合理的。再次运行测试，将会得到另一个未能通过的测试，如图 7-8 所示。



图 7-8

在本例中，从库方法返回的数字不是测试预期的返回数据。显然需要补充一些上下文。要更全面地定义上下文，需要告诉测试，它希望调用 ItemTypeRepository 的 Save 方法能得到什么样的 ItemType Id 值。如果正在编写的测试用于检查向数据存储保存数据的过程，而且在返回结果中得到了某种类型的 Id 数字，我希望我的单元测试在执行时随机选择这一数字：



```
protected override void Establish_context()
{
    base.Establish_context();

    var randomNumberGenerator = new Random();
    _itemId = randomNumberGenerator.Next(32000);

    _itemTypeRepository = new ItemTypeRepository();
}
```

ItemRepositoryTests.cs

从表面上来看，可以查看该单元测试，认为我引入了一定程度的不可预测性。但事实并非如此。我期望 `ItemTypeRepository` 类的 `Save` 方法总是以相同方式做出响应：它应当为保存到数据存储的实体返回 ID 号。我创建了一个条件，在此条件下，在每次运行该测试时，从 `ItemTypeRepository` 返回的 ID 号都不同，但在每次执行这一测试的作用域内，该号码是相同的。因为这个随机 ID 号在为测试创建上下文的过程中确定的，所以它不会向测试引入任何不可预测性。我只是改变了所返回 ID 号的值；返回该 ID 号的步骤并没有改变。为避免这种区别还不够明显，在后面演示如何为 `ItemTypeRepository` 创建模拟依赖项时还会再次谈到它。

再次运行该测试，将会看到它会失败，如图 7-9 所示。

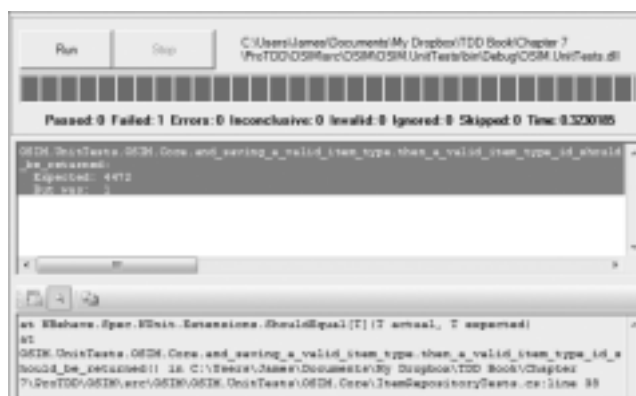


图 7-9

在本例中，该测试希望返回 4472 作为 `ItemType Id` 的值。`ItemTypeRepository` 类的 `Save` 的当前实现返回 1。

现在可以采取下一步骤使测试通过了。在这一阶段，考虑应用程序的设计是有帮助的。具体来说，该讨论如何实现数据持久化了。

该应用程序使用 Object Relational Mapper(ORM)框架 Fluent NHibernate (FNH)进行数据持久化。第 5 章解释了 ORM 背后的概念，在第 8 章将介绍如何用它来处理该示例中的数据持久化。.NET 世界提供了各种可供选择的 ORM 框架。我为该示例选择 FNH 是因为它是最熟悉的框架。这并不意味着 FNH 对于你的项目来说也是最佳 ORM。如果你或你的

团队对于某种 ORM(如实体框架 ,Entity Framework)拥有比较丰富的经验 ,而且这一工具也能满足需求 ,那它可能就是适合的 ORM。本书中使用 FNH 的示例都是可移植的 ,所以应当能够用你和你的团队决定使用的任意 ORM 来实现相同实务。



提示：

如果不熟悉 FNH ,那也不用着急。该示例没有使用 FNH 框架的任何高级功能。对于该示例中仅与 FNH 相关的步骤 ,还会进行一些简单解释。

和大多数现代 ORM 一样 ,FNH 鼓励使用第 5 章讨论的库模式。因为我经常使用 FNH ,所以我已经开发了一个基本的泛型库 ,可供采用 FNH 的项目使用。和 Specification 类一样 ,我在大多数项目中都使用该库。该库将包含在可供下载的代码示例中 ,但该示例演示了如何从头构建库。

FNH 提供了一个 Session 类 ,用于处理与数据存储之间的所有交互。从应用程序代码中调用该库时 ,最后总是对 FNH Session 对象的一个或多个调用。Session 对象是由 FNH 接口类型 ISessionFactory 的实例创建的。要在 ItemTypeRepository 中使用这些类型 ,需要添加对 FluentNHibernate 和 NHibernate 程序集的引用 ,如图 7-10 所示。

在 ItemTypeRepository 能够识别 FNH 框架中的类之前 ,需要向 ItemTypeRepository.cs 文件添加一条 using 语句：

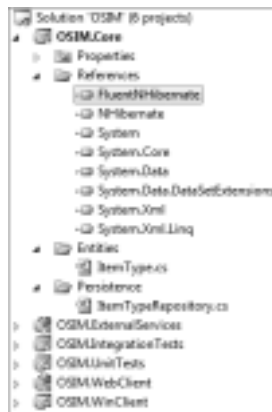


图 7-10



可从
wrox.com
下载源代码

```
using NHibernate;
```

ItemTypeRepository.cs

现在需要向 ItemTypeRepository 类中添加一个成员变量 ,用于存储 ISessionFactory 对象：



可从
wrox.com
下载源代码

```
public class ItemTypeRepository : IItemTypeRepository
{
    private ISessionFactory _sessionFactory;

    public int Save(ItemType itemType)
    {
        return 1;
    }
}
```

ItemTypeRepository.cs

最后 ,需要向 ItemTypeRepository 类添加构造函数 ,这样在创建 ItemTypeRepository 时允许提供一个实现 ISessionFactory 的类：



```
public ItemTypeRepository(ISessionFactory sessionFactory)
{
    _sessionFactory = sessionFactory;
}
```

ItemTypeRepository.cs

为便于检查，下面给出了当前 ItemTypeRepository 类的完整列表：



```
public class ItemTypeRepository : IItemTypeRepository
{
    private ISessionFactory _sessionFactory;

    public ItemTypeRepository(ISessionFactory sessionFactory)
    {
        _sessionFactory = sessionFactory;
    }

    public int Save(ItemType itemType)
    {
        return 1;
    }
}
```

ItemTypeRepository.cs

现在是尝试运行测试的好时机。遗憾的是，应用程序不再能编译了。创建 ItemTypeRepository 的构造函数，以允许客户为 ISessionFactory 注入一个值时，清除了默认构造函数(没有形参)。要使用 ItemTypeRepository，需要为实现 ISessionFactory 的构造函数提供一个值。因为这是一个单元测试，而且需要使 ItemTypeRepository 中的代码保持隔离，所以需要提供模拟对象，而不是提供有效的 FNH 会话工厂(Session Factory)。

回想一下第 2 章的内容，模拟框架 Moq 允许创建模拟对象或存根对象，以注入到正在测试的类中，用以确保该代码与外部依赖项相隔离。要使用 Moq 框架，首先需要向 OSIM.UnitTests 对象中添加对它的引用，如图 7-11 所示。

下一步是向 and_saving_a_valid_item_type 类的 Establish_context 方法中添加代码，以根据 FNH ISessionFactory 接口创建模拟对象：

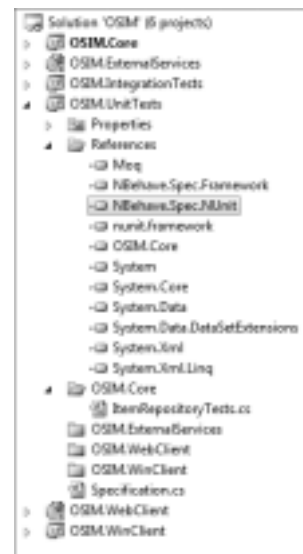


图 7-11



```
protected override void Establish_context()
{
    base.Establish_context();

    var randomNumberGenerator = new Random();
    _itemId = randomNumberGenerator.Next(32000);
    var sessionFactory = new Mock < ISessionFactory > ();

    _itemTypeRepository = new ItemTypeRepository();
}
```

ItemRepositoryTests.cs

我们会注意到，Visual Studio 指出它没有关于 ISessionFactory 的定义。这是因为 OSIM.UnitTests 项目没有引用 FNH 库。需要添加这些引用(如图 7-12 所示)。

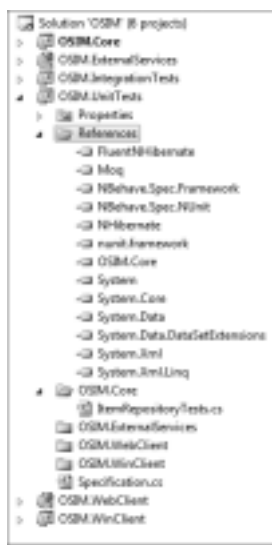


图 7-12

然后需要向 ItemRepositoryTests.cs 文件添加一条 using 语句，以包含所需要的 FNH 命名空间：

```
using NHibernate;
```

现在可以将模拟的会话工厂用作 ItemTypeRepository 的构造函数形参：



```
protected override void Establish_context()
{
    base.Establish_context();

    var randomNumberGenerator = new Random();
    _itemId = randomNumberGenerator.Next(32000);
    var sessionFactory = new Mock < ISessionFactory > ();
```

```

        _itemTypeRepository = new ItemTypeRepository(sessionFactory.Object);
    }

```

ItemRepositoryTests.cs

对于调用 FNH API 的 `ItemTypeRepository` 方法，现在应该创建其 `Save` 方法的实现了。前面曾经提到，与 FNH 进行的所有数据库访问都是通过 `Session` 类完成的。从一个实现 FNH `ISessionFactory` 接口的对象实例中获取该 `Session` 类的实例。下面是 `ItemTypeRepository` 中 `Save` 方法的实现，它利用 `Session` 类将 `ItemType` 保存到数据库：



```

public int Save(ItemType itemType)
{
    int id;
    using (var session = _sessionFactory.OpenSession())
    {

        id = (int) session.Save(itemType);
        session.Flush();
    }
    return id;
}

```

ItemTypeRepository.cs

如果您从来没有用过 FNH 框架，那需要对 FNH 特有的这些命令进行一点解释。首先，从 `SessionFactory` 中获取 `Session`。它本质上是与数据库的连接。在创建 `SessionFactory` 时，为其配置了连接数据库所需要的信息。这一主题将在第 8 章详细介绍。第一个调用是对 `Session` 对象 `Save` 方法的调用。该方法将实体保存到数据库中，返回主键值或 ID 值。在映射中指定了实体中的哪个字段是所需要的 ID，这一内容将在第 8 章中介绍。接下来调用 `Flush`，以确保对数据库的任意挂起操作都是完整的。当到达 `using` 代码块的结尾时，`Session` 关闭。

再次运行测试，将会看到另外一次失败，如图 7-13 所示。



图 7-13

ItemTypeRepository.cs 文件的第 26 行是对 Session 类的 Save 方法的调用。Null-Reference-Exception 告诉我们，在调用 OpenSession 方法时，SessionFactory 返回一个 Null 对象。这意味着需要向测试中添加另一个模拟对象用来代表 Session ,还需要为 SessionFactory.OpenSession 添加一个存根方法，它返回模拟的 Session 对象：

```
protected override void Establish_context()
{
    base.Establish_context();
    var randomNumberGenerator = new Random();
    _itemId = randomNumberGenerator.Next(32000);
    var sessionFactory = new Mock < ISessionFactory > ();
    var session = new Mock < ISession > ();
    sessionFactory.Setup(sf => sf.OpenSession()).Returns(session.Object);
    _itemTypeRepository = new ItemTypeRepository(sessionFactory.Object);
}
```

ItemRepositoryTests.cs

这一语法与 2.4.3 小节的代码类似。它向代表 SessionFactory 的模拟发出命令：在调用 SessionFactory 对象的 OpenSession 方法时，返回为 Session 创建的模拟。这时运行测试会发现，仍没有完成自己的工作(如图 7-14 所示)。



图 7-14

最后一个障碍是为 Session 模拟提供 Save 方法的存根实现。在本例中，希望 Save 方法返回在测试类的 Establish_context 方法中生成的随机 ItemType Id：



```
session.Setup(s => s.Save(_testItemType)).Returns(_itemId);
```

ItemRepositoryTests.cs

下面是 Establish_context 的完整实现：

```
protected override void Establish_context()
{
```

```

base.Establish_context();

var randomNumberGenerator = new Random();
_itemTypeId = randomNumberGenerator.Next(32000);
var sessionFactory = new Mock < ISessionFactory > ();
var session = new Mock < ISession > ();

session.Setup(s => s.Save(_testItemType)).Returns(_itemTypeId);
sessionFactory.Setup(sf => sf.OpenSession()).Returns(session.Object);

_itemTypeRepository = new ItemTypeRepository(sessionFactory.Object);
}

```

ItemRepositoryTests.cs

7.2.2 运行可以通过的测试

如图 7-15 所示，测试现在可以通过了。

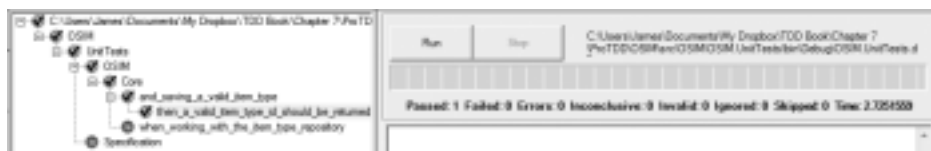


图 7-15

获得通过的测试表明已经完成了该功能。在工作环境中，可能会在之后再返回来做一些重构工作，但这个得以通过的测试证明：就需求而言已经完成了这一功能，可以将它与基本代码的其余部分集成在一起。

如果你和我认识的大多数开发人员一样，那会迫切希望仔细调整自己的代码，并添加一些内容，以使其变得“更好”。一定要按捺住这种愿望。添加并未请求的功能(有时称为“镀金”)所导致的问题可能要多于所解决的问题。这一添加过程要花费时间、增加基本代码的复杂性，产生维护开销，还可能影响需求范围之内的系统其他部分。不要掉入这样的陷阱——“嗯，业务可能需要它”。在公司明确要求添加功能之前，它就是不需要的。这与懒惰无关；只是向公司提供它所需要的功能的同时，尽可能使基本代码和应用程序保持简单。在当前示例中，已经给出了一个明文列出的业务需求。该需求已经被转换为单元测试。这段代码满足了该测试。这就是“完成任务”的定义。

7.2.3 编写下一个测试

在完成该测试后，可以再来研究用户情景或功能，并决定是否需要编写更多测试。在本例中，希望再编写一个测试，以确保在有人尝试向数据库中保存 Null 对象时，库会抛出异常。和任何其他功能或需求一样，第一步也是编写测试。如果 ItemRepositoryTests.cs 文

件还没有打开则打开它，并向该基类中添加一个新的测试类，它是从 `when_working_with_the_item_type_repository` 基类继承的：



```
public class and_saving_an_invalid_item_type :
    when_working_with_the_item_type_repository
{
}
```

ItemRepositoryTests.cs

接下来添加一个方法，用于验证这个测试的结果是否与预期相同——在本例中，是指抛出 `ArgumentNullException` 异常：



```
public class and_saving_an_invalid_item_type :
    when_working_with_the_item_type_repository
{
    private Exception _result;

    [Test]
    public void then_an_argument_null_exception_should_be_raised()
    {
        _result.ShouldBeInstanceOfType(typeof(ArgumentNullException));
    }
}
```

ItemRepositoryTests.cs

这段代码将 `_result` 声明为 `Exception` 类型，它是 `ArgumentNullException` 的基本类型。这是因为尽管希望调用 `ItemTypeRepository` 时引发异常，但想验证所抛出异常的类型。如果将 `_result` 声明为所希望的指定类型(在本例中，为 `ArgumentNullException`)，抛出了某种完全不同的东西，如 `NullReferenceException` 类型的异常，则测试在计算结果之前会出现运行时错误。而将 `_result` 声明为 `Exception` 类型的变量，就可以存储所抛出的异常，而不用考虑其具体类型。

那么，如何填充 `_result` 呢？前面的示例使用了 `Because_of` 方法的重写实现，它调用被测方法，并将返回结果存储在 `_result` 成员变量中：



```
protected override void Because_of()
{
    _result = _itemTypeRepository.Save(_testItemType);
}
```

ItemRepositoryTests.cs

在本例中，由于需要捕获在调用 `ItemTypeRepository` 的 `Save` 方法时所抛出的异常，需要以稍微不同的方式来设计 `Because_of` 方法的实现结构。对于该测试来说，希望把对 `Save`

的调用放在 Try/Catch 代码块中，并在 Catch 部分将异常对象存储在 _result 成员变量中：



```
protected override void Because_of()
{
    try
    {
        _itemTypeRepository.Save(null);
    }
    catch (Exception exception)
    {
        _result = exception;
    }
}
```

ItemRepositoryTests.cs

如果要尝试运行该测试，将会得到编译错误。单元测试类 `and_saving_an_invalid_item_type` 没有对成员变量 `_itemTypeRepository` 的声明。该成员变量是在前面的测试中是作为 `and_saving_a_valid_item_type` 类的一部分创建和实例化的。也可以在 `and_saving_an_invalid_item_type` 中做同样的事情。但这一代码已经编写了一次。尽管可以接受单元测试中的某些代码重复，但仍然希望尽可能使单元测试代码保持 DRY(不要重复自己, don't repeat yourself)。`and_saving_a_valid_item_type` 和 `and_saving_an_invalid_item_type` 类是从 `when_working_with_the_item_type_repository` 基类继承而来的。为了减少代码重复，移动声明和代码，在基类中为 `_itemTypeRepository` 创建模拟(mock)：



```
public class when_working_with_the_item_type_repository : Specification
{
    protected IItemTypeRepository _itemTypeRepository;
    protected override void Establish_context()
    {
        base.Establish_context();
        _itemTypeRepository = new ItemTypeRepository(sessionFactory.Object);
    }
}
```

ItemRepositoryTests.cs

从 `and_saving_a_valid_item_type` 类和实例化 `ItemTypeRepository` 实例的代码行中删除对 `_itemTypeRepository` 的声明：



```
public class and_saving_a_valid_item_type :
when_working_with_the_item_type_repository
{
    private int _result;
    private ItemType _testItemType;
    private int _itemId;

    protected override void Establish_context()
```

```

    {
        base.Establish_context();
        var randomNumberGenerator = new Random();
        _itemTypeId = randomNumberGenerator.Next(32000);
        var sessionFactory = new Mock < ISessionFactory > ();
        var session = new Mock < ISession > ();
        session.Setup(s => s.Save(_testItemType)).Returns(_itemTypeId);
        sessionFactory.Setup(sf => sf.OpenSession()).Returns(session.Object);
    }
}

```

ItemRepositoryTests.cs

我们已经移动了对 `ItemTypeRepository` 模拟的声明和实例化，但 `ItemTypeRepository` 模拟仍依赖于对 `SessionFactory` 的模拟，而后者又依赖于 `Session` 对象的模拟。现在需要创建这些模拟，作为 `when_working_with_the_item_type_repository` 类的实例变量：



```

public class when_working_with_the_item_type_repository : Specification
{
    protected IItemTypeRepository _itemTypeRepository;
    protected Mock < ISessionFactory > _sessionFactory;
    protected Mock < ISession > _session;

    protected override void Establish_context()
    {
        base.Establish_context();
        _sessionFactory = new Mock < ISessionFactory > ();
        _session = new Mock < ISession > ();
        _itemTypeRepository = new ItemTypeRepository(_sessionFactory.Object);
    }
}

```

ItemRepositoryTests.cs

接下来，需要将实现 `SessionFactory` 模拟的 `CreateSession` 方法存根的代码行从 `and_saving_a_valid_item_type` 类的 `Establish_context` 方法移动到 `when_working_with_the_item_type_repository` 类的 `Establish_context` 方法：



```

protected override void Establish_context()
{
    base.Establish_context();
    _sessionFactory = new Mock < ISessionFactory > ();
    _session = new Mock < ISession > ();
    _sessionFactory.Setup(sf => sf.OpenSession()).Returns(_session.Object);
    _itemTypeRepository = new ItemTypeRepository(_sessionFactory.Object);
}

```

ItemRepositoryTests.cs

这一改变使 `and_saving_a_valid_item_type` 类的 `Establish_context` 方法看起来类似如下：



```
protected override void Establish_context()
{
    base.Establish_context();

    var randomNumberGenerator = new Random();
    _itemTypeId = randomNumberGenerator.Next(32000);

    session.Setup(s => s.Save(_testItemType)).Returns(_itemTypeId);
}
```

ItemRepositoryTests.cs

需要做的最后一处修改是让 `and_saving_a_valid_item_type` 类的 `Establish_context` 方法使用 `when_working_with_the_type_repository` 类的成员变量 `_session`：



```
protected override void Establish_context()
{
    base.Establish_context();

    var randomNumberGenerator = new Random();
    _itemTypeId = randomNumberGenerator.Next(32000);
    _session.Setup(s => s.Save(_testItemType)).Returns(_itemTypeId);
}
```

ItemRepositoryTests.cs

移动这些声明不仅可以在这两个测试中重用大量代码，还可以在其他许多需要编写的测试中重用。让 `_session` 变量一直作为基类 `when_working_with_the_item_type_repository` 的受保护成员，可以分别提供每个测试类方法的存根。`when_working_with_the_item_type_repository` 中的 `Establish_context` 方法负责实现会话工厂的存根，所以不需要告诉它为每个测试返回 `Session` 的模拟。由于向该类中添加了更多的测试，所以这是一种有效的组织方式。

在完成测试类的重构之前，希望重新运行 `and_saving_a_valid_item_type` 测试类中的 `then_a_valid_item_type_id_should_be_returned` 测试，如图 7-16 所示。

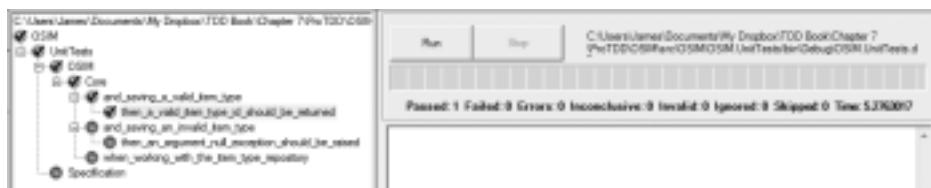


图 7-16

NUnit 单元测试运行程序已经选择了新的 `then_an_argument_null_should_be_raised` 单元测试。但现在，应当仅考虑现有的 `then_a_valid_item_type_id_should_be_returned` 测试。现

在之所以应当更多地关注它，是希望确保对单元测试类(`and_saving_a_valid_item_type` 和 `when_working_with_the_item_type_repository`)的重构没有对测试造成任何破坏。在重构之前这一测试已经通过，如果没有对业务代码进行任何更改，则有理由相信没有对测试造成损坏。在重构单元测试时，非常重要的一步是运行已有测试以确保它们仍然都能通过，没有对它们造成任何破坏。不要忽略这一步。

由于现在已经完成了对单元测试类的重构，所以可以去研究新的测试类(`and_saving_an_invalid_item_type`)，并首次运行该测试(如图 7-17 所示)。

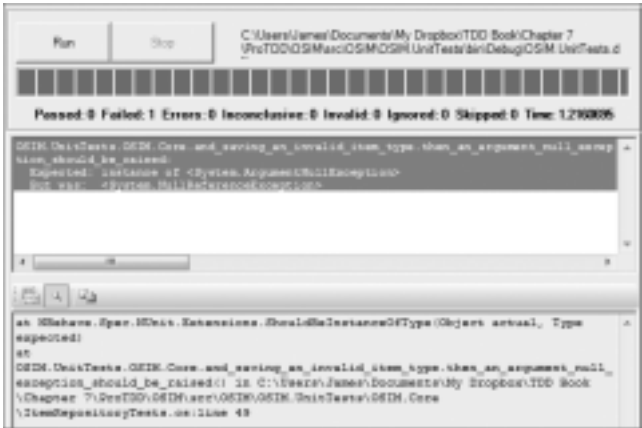


图 7-17

在调用 `ItemTypeRepository` 类的 `Save` 方法时会产生预料之外的异常。该例很好地说明了将 `_result` 声明为宽泛的 `Exception` 类型，而不是非常狭隘的 `ArgumentNullException` 类型有多重要。该测试之所以会失败，是因为 `Session` 类的模拟没有提供 `Save` 方法的存根，用于对 `Null` 值作出反应。由于为 `SessionFactory` 和 `Session` 对象创建模拟的机制都在 `when_working_with_the_item_type_repository` 类中处理，因此只需要为 `Save` 方法声明存根：



```
protected override void Establish_context()
{
    base.Establish_context();
    _session.Setup(s => s.Save(null)).Throws(new ArgumentNullException());
}
```

ItemRepositoryTests.cs

在 `Session` 对象设置了 `Save` 方法的存根之后，再次运行该测试，如图 7-18 所示。



图 7-18

既然现在 Session 模拟已经抛出了正确的异常，所以该测试已经通过。下面是完整的 ItemRepositoryTests.cs 文件：



```
using System;
using Moq;
using NBehave.Spec.NUnit;
using NHibernate;
using NUnit.Framework;
using OSIM.Core.Entities;
using OSIM.Core.Persistence;

namespace OSIM.UnitTests.OSIM.Core
{
    public class when_working_with_the_item_type_repository : Specification
    {
        protected IItemTypeRepository _itemTypeRepository;
        protected Mock <ISessionFactory> _sessionFactory;
        protected Mock <ISession> _session;

        protected override void Establish_context()
        {
            base.Establish_context();

            _sessionFactory = new Mock < ISessionFactory > ();
            _session = new Mock < ISession > ();

            _sessionFactory.Setup(sf => sf.OpenSession()).Returns
                (_session.Object);

            _itemTypeRepository = new ItemTypeRepository
                (_sessionFactory.Object);
        }

        public class and_saving_an_invalid_item_type :
            when_working_with_the_item_type_repository
        {
            private Exception _result;

            protected override void Establish_context()
            {
                base.Establish_context();

                _session.Setup(s => s.Save(null)).Throws
                    (new ArgumentNullException());
            }

            protected override void Because_of()
            {
                try
                {
                    _itemTypeRepository.Save(null);
                }
            }
        }
    }
}
```

```
        catch (Exception exception)
        {
            _result = exception;
        }
    }

    [Test]
    public void then_an_argument_null_exception_should_be_raised()
    {
        _result.ShouldBeInstanceOfType(typeof(ArgumentNullException));
    }
}

public class and_saving_a_valid_item_type :
    when_working_with_the_item_type_repository
{
    private int _result;
    private ItemType _testItemType;
    private int _itemTypeId;

    protected override void Establish_context()
    {
        base.Establish_context();

        var randomNumberGenerator = new Random();
        _itemTypeId = randomNumberGenerator.Next(32000);

        _session.Setup(s => s.Save(_testItemType)).Returns(_itemTypeId);
    }

    protected override void Because_of()
    {
        _result = _itemTypeRepository.Save(_testItemType);
    }

    [Test]
    public void then_a_valid_item_type_id_should_be_returned()
    {
        _result.ShouldEqual(_itemTypeId);
    }
}
}
```

ItemRepositoryTests.cs

7.3 通过重构来改进代码

在开始使用 TDD 编写代码时，唯一的目标就是使测试通过。不要担心编写的代码是否精致、完美，或者只是功能性的，确保达到了主要开发目的——满足业务需要即可。这些

测试本身是以业务需求为基础的，通过这些测试即表明满足了业务需要。在满足了业务需求之后，就应当考虑重构了。

如第4章所述，重构是指在不影响代码外部行为的前提下改进现有代码，使其效率更高、可读性更好、更易于维护。拥有一组已经通过的测试，就可以放心地重构代码，而且准确地知道什么时候代码不再能满足业务需求。现在是时候戴上编辑帽子，认真、仔细地研究代码了。它的可读性怎么样？团队的新成员能否仅通过查看代码就可以快速理解它的功能，并知道代码是如何实现这一功能的？代码是否符合 SOLID 原则？是否有许多重复代码？是否有效地使用了接口和抽象？是否注入了依赖项，以保持耦合的松散性？现在是解决这些问题的时候了。努力让代码更完美一些，但要保证仍然能够通过这些测试。

7.4 多角度测试

“幸福路径”总是最容易测试的。如果单元测试总是停留在幸福路径上，或者进行测试的输入值属于可能输入的中间范围，那是不完整的。实际上，还可能有更坏的结果：它们会向你撒谎。它们会告诉你，你的代码满足了业务需要，而且质量很高。但事实并非如此。

问题在于所有的输入都是魔鬼。如果没有构建一些能够探查应用程序边界及其以外范围的测试，实际上就是在祈求好运，寄希望于用户永远不会突破应用程序的边界。用户只是想要能够正常工作的软件。他们并不知道内存大小、出界错误、Null 对象异常等概念，也不了解当所用取值高于或低于代码期望取值范围时所出现的问题。也不应当要求他们知道这些；这是应用程序应当考虑的问题。

在编写单元测试时，一定要确信它们对方法的预期边界进行了检查。如果方法希望获得一个 1~10 之间的取值，则编写一个传递数值 5 的测试，然后编写传递数值 1 和 10 的测试，再编写传递 0 和 11 的测试。确保代码知道如何处理预期之外的输入。这就是 `and_saving_an_invalid_item_type` 测试类及其 `then_an_argument_null_exception_should_be_raised` 测试的要点。我希望确保 `ItemTypeRepository` 类能够处理以下情景并相应做出响应：客户端用 Null 对象调用 `Save`。从多个角度进行测试，以确保代码的质量。

7.5 本章小结

为应用程序开发的第一项用户情景或功能非常重要。第一工作单元的成功会设置好基调，定义项目的价值。在计划工作时，以合理方式安排各项功能的开发顺序：从应用程序的核心开始，向外层发展。在将用户情景分解为功能时，一定要让它们保持小型化、相互隔离、可以测试。

在创建单元测试时，采用一致的标准为类、方法和变量命名。为它们起的名字应当是有意义的——不仅对开发人员如此，对于获邀参加测试验证的非技术业务用户也应当如此。

研究 BDD 命名风格和约定，并利用这一知识为类、方法和变量命名。

在编写任何代码之前，会看到测试失败。要确保测试正在测试不存在的功能，这一点非常重要。如果没有编写任何代码就通过了测试，那可能意味着很多事情。可能说明这是一个重复功能。也可能是在创建以前的功能时，作为副产品创建了现在所需要的功能。还有一种可能——测试没有测试正确的功能。找出原因；不要只是假定以这种方式通过测试之后就意味着一切 OK，可以继续后面的开发了。

仅编写能使测试通过的最少量代码。在为了使测试通过而实现业务代码时，总是尽力做最简单的有效工作。测试通过，也就完成了任务。不要进行“镀金”。如果有关一项功能的特性、缺陷或要求没有出现在用户情景或功能列表中，就不要建立它。等到需要这项功能时再建立；不要提前执行这个过程。

一定要从多个方向进行测试。测试最佳情景或“幸福路径”对于确保代码质量没有任何帮助。要测试方法的边缘情景：输入那些位于期望边界的参数、超出边界的取值，以及预期之外的取值和条件。记住，所有输入都是魔鬼。

在拥有一套获得通过的测试之后，就知道应用程序已经满足了业务需求。在满足这些需求之后，可以对代码进行重构，使它更优化、可读性更强、更易于维护。在重构代码时，要时刻牢记 SOLID 原则，把它们当作行动指南。要使这些测试保持通过，这样就知道代码仍然能够满足业务需要了。

最后，研究一下 ORM 框架(如 NHibernate 和 Entity Framework)，以加快开发速度。将数据访问等功能交由这样一种框架来完成，可以加快开发速度，而且能降低出现缺陷的可能性。

第 8 章

集成测试

本章内容

- 集成测试与单元测试有什么区别？集成测试为什么重要？
- 为什么尽早编写和运行集成测试非常重要？
- 如何用 NUnit 编写自动化集成测试？这一实践从哪里开始偏离使用 NUnit 编写单元测试的过程？
- 端对端测试是什么？为什么它们对应用程序开发工作的成功很关键？
- 如何管理集成测试和端对端测试所用的外部资源？
- 应当在什么时候运行集成测试和端对端测试，如何运行？

在 TDD 中，单元测试的主旨在于推动应用程序中分离组件和组成部分的开发。为此，非常重要的一点是要让单元测试及执行这些测试的代码与其他组件或外部资源隔离。事实上，测试相同功能单元的测试也应当相互隔离。这意味着这些测试可以采用任意顺序或组合方式运行，并产生相同的可预测结果。这一点是非常必要的，可以确保采用一种简单的、松散耦合的、完整的方式来开发应用程序的各个组件。通过隔离单元测试，还可以更轻松地诊断和纠正导致测试失败的缺陷。

但是，总有必须将应用程序各个组件合并到一起的时候。而确保这些不同部分能够正确地组合在一起，正是集成测试的工作。集成测试用来验证正在开发的各个应用程序组件和外部资源能够正确地协同工作。

8.1 早集成、常集成

在使用 TDD 时，类和方法都是根据测试开发的。具体来说，它们是根据测试反映技术规范及功能需求的思想而开发的。编写了足以使所有测试通过的代码之后，类和/或方法就满足了所有技术规范和需求。应用程序的其他类、外部资源和其他组件都被模拟或实现存根，以在隔离状态下测试代码。这种测试保证了单个类和组件的质量。

由于有了这一隔离概念，真正的 TDD 单元测试从定义上就只是保证当前类或方法范围内的质量。这样就在应用程序作为整体使用的类、组件和外部资源之间形成了一系列的接合部。这些接合部是由各个类、组件和外部资源提供的接口定义的。即使一个接口的意图非常明确，在将两个或更多个软件组合在一起构成应用程序时，也仍然可能出现质量问题。应当确保正确地实现了这些依赖项的模拟和存根。但是，仍然有可能引入错误和缺陷。向一个方法提供的输入可能不是预期之中的。方法的输出也可能不是使用该方法的类所需要的。在许多情景下，类、组件和资源是经过精心设计的，都有全部能够通过的经过精心编写的单元测试。但是，这些类、组件和外部资源链接在一起时，不一定能够如预期一样发挥功能。通过良好的设计和交流可以减少其中许多问题。但即使是在最优情况下，仍然可能出现集成问题。只有通过对这些组件、类和外部资源的整体进行测试，才能确保所开发应用程序的接合部是可靠的、可依赖的。

集成测试的设计目标就是针对应用程序中的各种接合部，确保应用程序的各个部分能够正确地协同工作。它们与单元测试的相似之处在于：相同的自动化单元测试框架可以(也应当可以)用于创建这些集成测试。与单元测试的区别在于：单元测试使用模拟来隔离具体的被测方法与类，而集成测试则覆盖了其测试目标(各方法)之间的所有应用程序代码，还会测试所有通向系统最底层(通常指数据存贮)的通道。

由于集成测试如此重要，所以许多开发人员情愿把它留到项目的最后再进行，这是错误的。许多在 TDD 出现之前拥有丰富经验的软件开发人员(甚至包括其他许多开发人员)都可以至少向你讲述一个悲惨的故事：一个应用程序马上要启动或者要部署给质量保证部门了，却要提前花费整整一个晚上的时间来试图查明，为什么将大型系统的两个或多个组件放在一起它们拒绝工作。显然，这种“分裂”行为是出乎意料的，很多情况下，开发团队必须都变成夜猫子，以诊断和纠正这一问题。很多时候，其解决方案是提供一个补丁或垫片。我曾经看到一个在比萨饼盒背面设计和制定的解决方案。这些补丁和垫片通常可以清除症状(在大多数情况下，这些补丁和垫片的创建没有任何相关文档进行描述)。但大多数开发人员没有意识到或者不愿意谈论的是，底层的问题仍然存在。它只是隐藏起来了，回头面对它的，将是一个被派来维护该应用程序的初级开发人员。

好消息是：不一定非得这样。事实上，也不应当是这样。如果开发团队在项目的早期就开始自动化集成测试，并在整个开发周期中对其进行维护，就可以很轻松地避免这些连夜召开的紧急会议。

在开发周期的早期创建自动化集成测试，并定期运行它们，有助于查找和纠正正在组成

应用程序的各个类、组件和外部资源接合部之间出现的错误。如果自动化集成测试失败，就可以马上知道在应用程序的一个接合部出现了问题，而不用等到产品部署了两个月才知道。定期运行这些集成测试，将其看作 CI 过程的一部分，在向应用程序引入缺陷时，可以缩短可疑对象的列表。

在从事一个没有现有单元测试的“褐色地域(brownfield)”开发项目时(即维护、扩展或改进现有应用程序)，如果事先不能在应用程序的设计或编码实践中采用依赖项注入等方式，那集成测试可能是唯一可用于自动化测试的选项。在这些较早的项目中，特别是当它们拥有大量基础代码或者已经投入生产时，从经济的角度来看通常不太可能对基础代码进行重新改造，以引入支持真正单元测试的依赖项注入。在这些情况下，记住，任何测试(即使是集成测试)都比根本就没测试要好。

在继续开发和增强这些大型现有应用程序时，应当尝试引入一些有可能实现应用程序单元测试的技术和方法。全面改写应用程序是不太可能的，但只要可能，就应当努力改进应用程序。

8.2 编写集成测试

集成测试的编写与单元测试的编写相类似。事实上，在编写集成测试时使用的大量框架都与编写单元测试时使用的框架相同。区别在于，在编写集成测试时，其目的不一定是测试各代码单元。集成测试的目的是测试各个代码单元之间的接合部。这就意味着前面详细讨论的隔离规则不再适用。这样说也不完全正确；这些测试是由它们所测试的接合部隔离的，但它们会与不同的类和组件交互——甚至还可能与外部资源交互。集成测试的目的是确保这些不同组件能够协同工作。

在实践中，这意味着集成测试有时必须与外部资源交互。测试可能从数据库读取数据，也向其中写入数据。测试还可以调用一个 Web 服务，或者与硬盘上的文件或文件夹交互。因此，非常重要的一点就是能够访问一个环境，其中拥有所有这些资源的测试版本，而且可以对这些测试版本采取控制措施。在每次测试之前，需要重置环境以使其保持一致的测前状态。如果添加或改变了数据库中的数据，则需要删除它或者将其恢复原貌。如果调用了 Web 服务，需要该服务的测试版本能够根据传递给它的输入进行正确一致的响应。要确保能够将文件系统返回有序状态，撤消这些测试所引发的所有修改。

8.2.1 如何管理数据库

单元测试、集成测试和端对端测试有一件事情是相同的，就是它们都希望自己的环境在测试之前处于指定状态。对于单元测试，这一点很容易，因为通过依赖项注入和模拟消除了对外部资源的依赖。而对于集成测试和端对端测试，就有点复杂了，因为外部资源是这些测试的组成部分。

集成测试和端对端测试最常用到的外部资源就是数据库。对于集成测试和端对端测试

来说，拥有一致的开始环境意味着，在运行这些测试之前，数据库中的数据必须处于某一特定状态。在本章的示例中，这是一件很简单的任务；Fluent NHibernate 为该集成测试所使用的配置会在每次运行测试之前销毁并重建数据库。这种方法的效率是有争议的，但可以确保在每次运行测试之前数据库总是处于一致状态。

如果没有使用 ORM，或者销毁/重建方法不太现实，则需要在测试中设置一些回滚操作，以确保测试总是在一致环境中运行。这一过程可能采用一些不同的方式。很多情况下，执行这一数据库清理工作的合理位置和时机是在测试的初始化期间。这就确保了环境总是处于执行测试的正确状态。

在某些情况下，在测试结束时执行一个复位脚本或存储过程可能更容易、更实用。在这些情况下，需要确保测试的结构设计能够在每次运行测试时都会运行这一脚本或过程，而无论测试成功还是失败。在这些情况下，检索用于验证测试的数据之后、开始实际验证之前，应当马上运行该脚本。这是因为，NUnit 和大多数单元测试框架都是在第一个失败断言之后终止执行测试。这意味着，如果想等到所有断言都完成之后再执行数据库清理工作，在测试失败时可能就不会进行这一清理工作。

8.2.2 如何编写集成测试

第 7 章开始开发一些核心内容，这些核心内容最终演变为 OSIM 应用程序的数据访问层。别忘了，编写目前的实现只是为了使测试通过(这里的测试是指 `and_saving_an_invalid_item_type.then_an_argument_null_exception_should_be_raised` 和 `and_saving_a_valid_item_type.then_a_valid_item_type_id_should_be_returned`)。

这是一些单元测试，仅关注一个实现 `IItemRepository` 接口的类的 `Save` 方法。`IItemRepository` 接口的这一特定实现使用 Fluent NHibernate 来执行其数据访问。结果，`Save` 方法最终依赖于一个实现 Fluent NHibernate 的 `ISessionFactory` 接口的对象实例，以及一个实现 Fluent NHibernate 的 `ISession` 接口的对象实例。为了使单元测试保持隔离(确保它们仅测试该 `IItemRepository` 实现的 `Save` 方法)，可以传递实现 `ISessionFactory` 和 `ISession` 接口的模拟对象。我们将演示，如何根据需要利用由这些接口所定义的方法的存根，提供这些模拟对象。最终，对 `Save` 方法中的代码进行测试，但还没有验证：`IItemRepository` 的这一实现可以将 `ItemType` 类的实例保存到数据库中。

1. 回顾 `IItemRepository`

事实上，`IItemRepository`(`IItemRepository` 的实现)并没有将 `ItemType` 类的实例保存到数据库中。回顾一下，下面是 `IItemRepository` 当前实现的代码：



```
using System;
using NHibernate;
using OSIM.Core.Entities;

namespace OSIM.Persistence
```

```

{
    public interface IItemTypeRepository
    {
        int Save(ItemType itemType);
        ItemType GetById(int id);
    }

    public class ItemTypeRepository : IItemTypeRepository
    {
        private ISessionFactory _sessionFactory;

        public ItemTypeRepository(ISessionFactory sessionFactory)
        {
            _sessionFactory = sessionFactory;
        }

        public int Save(ItemType itemType)
        {
            int id;
            using (var session = _sessionFactory.OpenSession())
            {
                id = (int) session.Save(itemType);
                session.Flush();
            }
            return id;
        }

        public ItemType GetById(int id)
        {
            using (var session = _sessionFactory.OpenSession())
            {
                return session.Get < ItemType > (id);
            }
        }
    }
}

```

ItemRepositoryTests.cs

可能已经注意到,与第7章相比,这里添加了一个新方法。GetById 使 ItemTypeRepository 的使用者能够通过提供 ID 来检索 ItemType 实例,它是数据库中 ItemType 数据表的主键。为一个实体指定主键的操作将在本章后面有关 Fluent NHibernate 映射一节中介绍。在上面给出的代码中,用于调用 Fluent NHibernate 会话以根据 ID 获取 ItemType 的代码非常简单,这里不准备花费时间来介绍它。Fluent NHibernate 的确拥有一些根据非主键字段搜索实体的功能,但这超出了本书的范围。有关搜索条件的详情,参阅 Fluent NHibernate 文档。

现在,ItemTypeRepository 只有一个构造函数的实现:



```
public ItemTypeRepository(ISessionFactory sessionFactory)
{
    _sessionFactory = sessionFactory;
}
```

ItemRepositoryTests.cs

该构造函数以 `ISessionFactory` 实例为唯一形参。现在，它正被用于传递 `ISessionFactory` 的模拟实例。为了进行集成，需要提供一个对象，它是 `ISessionFactory` 接口的一个有效实现。有各种方法可以在创建时向 `ItemTypeRepository` 提供 `ISessionFactory` 接口的实现。我喜欢使用在第 5 章介绍的 `Ninject`，以处理类实例化和依赖项解析。在创建 `ItemTypeRepository` 时，完全可以用它来向每个 `ItemTypeRepository` 提供正确创建和配置的 `ISessionFactory` 实例。

2. 为依赖项注入添加 Ninject

为了使用 `Ninject` 创建 `ItemTypeRepository` 及其依赖项(包括 `ISessionFactory` 的一个实例)，首先需要为 `OSIM.IntegrationTests` 项目创建一个模块类。调用该新类 `IntegrationTestsModule`，如图 8-1 所示。

要使用 `Ninject`，需要在 `OSIM.IntegrationTests` 项目中添加对 `Ninject` 程序集的引用，如图 8-2 所示。

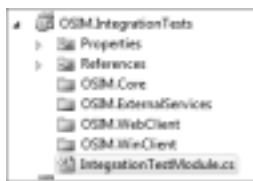


图 8-1

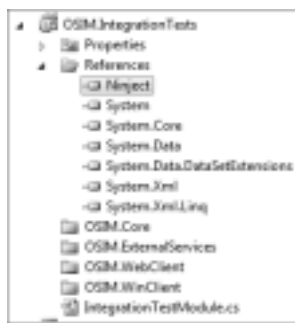


图 8-2

在向 `OSIM.IntegrationTests` 项目添加了 `Ninject` 库之后，可以开始使用 `IntegrationTestsModule` 类了。第一步是向 `IntegrationTestModule.cs` 文件中添加针对 `OSIM.Core.Persistence` 名称空间的 `using` 语句：



```
using OSIM.Core.Persistence;
```

IntegrationTestModule.cs

接下来需要向 `OSIM.IntegrationTests` 项目添加对 `NHibernate` 程序集的引用，如图 8-3 所示。

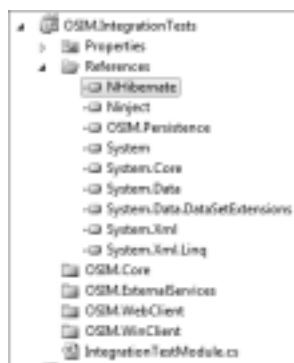


图 8-3

还需要添加对 NHibernate 名称空间的引用：



```
using NHibernate;
```

IntegrationTestModule.cs

最后，需要为 Ninject.Modules 名称空间添加 using 语句，并将 IntegrationTestModule 的定义修改为继承 Ninject 提供的 NinjectModule 基类。在此之后，IntegrationTestModule.cs 文件的内容应当如下所示：



```
using NHibernate;
using Ninject.Modules;
using OSIM.Core.Persistence;

namespace OSIM.IntegrationTests
{
    public class IntegrationTestModule : NinjectModule
    {
    }
}
```

IntegrationTestModule.cs

现在，IntegrationTestModule 不能编译。NinjectModule 有一个抽象方法 Load 需要实现：



```
public class IntegrationTestModule : NinjectModule
{
    public override void Load()
    {
        throw new NotImplementedException ();
    }
}
```

IntegrationTestModule.cs

该应用程序现在可以编译通过了，但如果尝试使用 `IntegrationTestModule` 来建立 `ItemTypeRepository` 的实例——或者任何相关内容，都会抛出 `NotImplementedException` 异常。需要向这个模块添加一些绑定规则，以便在向 Ninject 要求一个 `ItemTypeRepository` 的实例时，它能够创建一个 `ItemTypeRepository` 类。第一个规则很简单；在向 Ninject 要求一个 `IItemTypeRepository` 实例时，应当返回 `ItemTypeRepository` 的实例：



```
public override void Load()
{
    Bind < IItemTypeRepository > ().To < ItemTypeRepository > ();
}
```

IntegrationTestModule.cs

下一步更复杂。我们不能仅将 `SessionFactory` 类的实例绑定到 `ISessionFactory` 接口。对于在代码中实例化的 `ISessionFactory`，Fluent NHibernate 不会提供用于实现它的类的实现方式。我们需要调用 `Fluent NHibernate` 静态类 `Fluently`，方法是向它提供一组配置信息，可供 `Fluent NHibernate` 用于创建实现 `ISessionFactory` 接口的对象的实例。

在该示例中，我希望使用 Ninject 提供程序。提供程序就是一个类，它的任务是创建和返回实现指定接口的对象。提供程序还使开发人员能够实例化对象，以定义一些实例化规则，这些规则远比“如果我要求 X，就给我 Y”复杂得多。用于实例化 `ISessionFactory` 配置实例的规则就绝对属于这一类别。

第一步是向 `IntegrationTestModule` 的 `Load` 方法添加一条绑定规则，它指示 Ninject 利用一个指定的提供程序，满足对相关接口的请求：



```
public override void Load()
{
    Bind <IItemTypeRepository> ().To <ItemTypeRepository> ();
    Bind <ISessionFactory> ().ToProvider
        (new IntegrationTestSessionFactoryProvider());
}
```

IntegrationTestModule.cs

我们需要创建 `IntegrationTestSessionFactoryProvider` 的实例。为了简单起见，可以仅在 `IntegrationTestModule.cs` 文件中创建这个类，并将它放在 `IntegrationTestModule` 的定义之后。从技术上来说，`IntegrationTestSessionFactoryProvider` 类需要实现 Ninject 提供的 `IProvider` 接口。但要构建这样一个实现 `IProvider` 接口的类，更简单的方法是继承 `Provider<T>` 基类，这也是由 Ninject 提供的。要使用这个基类，需要向 `IntegrationTestModule.cs` 文件添加另一条 `using` 语句：



```
using Ninject.Activation;
```

IntegrationTestModule.cs

还需要向 OSIM.Core 项目添加一个引用，并添加一条 using 语句，以便向 ItemType 所在位置添加名称空间：



```
using OSIM.Core.Entities;
```

IntegrationTestModule.cs

接下来，需要定义 IntegrationTestSessionFactoryProvider 类：



```
public class IntegrationTestSessionFactoryProvider : Provider <ISessionFactory>
{
}
}
```

IntegrationTestModule.cs

Provider 基类定义了一个名为 CreateInstance 的抽象方法，需要为它提供实现：



```
public class IntegrationTestSessionFactoryProvider : Provider <ISessionFactory>
{
    protected override ISessionFactory CreateInstance(IContext context)
    {
        throw new NotImplementedException ();
    }
}
```

IntegrationTestModule.cs

IntegrationTestModule 将为 Ninject 提供必要的规则和信息，用于创建这些类的实例，并完成运行集成测试的完整对象图。在生产时，希望使用 Ninject 模块来创建实际生产依赖项。我在 OSIM.Core 项目的 Persistence 文件夹中包含了一个文件，其中有一个示例，显示了生产 Ninject 模块的大致情况。

3. 创建 Fluent NHibernate 配置

我们需要调用 Fluent NHibernate 的 Fluently 静态类，并提供在访问数据库时所必需的配置信息：



```
public class IntegrationTestSessionFactoryProvider : Provider < ISessionFactory >
{
    protected override ISessionFactory CreateInstance(IContext context)
    {
        var sessionFactory = Fluently.Configure()
            .Database(MsSqlConfiguration.MsSql2008
```

```

        .ConnectionString(c => c.Is(ConfigurationManager.AppSettings
            ["localDb"])).ShowSql())
        .Mappings(m => m.FluentMappings.AddFromAssemblyOf< ItemTypeMap >())
        .ExportTo(@"C:\Temp")
        .ExposeConfiguration(cfg => new SchemaExport(cfg).Create(true, true))
        .BuildSessionFactory();
    return sessionFactory;
}
}

```

IntegrationTestModule.cs

这段代码值得解释一下。在第 7 章曾经提到, Fluent NHibernate 使用了一个实现 `ISession` 接口的对象来访问数据库。`ISession` 的这一实现是由 `ISessionFactory` 提供的。要获得一个实现 `ISessionFactory` 的对象的实例, 需要使用 Fluent NHibernate `Fluently.Configure` API。幸运的是, 这些 API 调用是作为扩展方法实现的, 使该 `ISessionFactory` 实现的创建过程要稍微容易一些。

在这个过程的开始, 首先调用 `Fluently.Configure` 方法。将对扩展方法 `DataBase` 的调用添加到对 `Configure` 的调用。`Database` 方法告诉 Fluent NHibernate 该数据库位于何处, 它正在使用哪种数据库管理系统(DBMS)。在本例中, 使用 Microsoft SQL Server 2008 的一个实例。连接字符串存储在配置文件中 `localDb` 键下的 `AppSettings` 部分中。

接下来需要告诉 Fluent NHibernate, 对实体对象的映射位于何处。本章后面会介绍 NHibernate 映射, 但在该例中告诉 Fluent NHibernate, 这些映射位于定义 `ItemTypeMap` 类的程序集中。`ItemTypeMap` 类是 `ItemType` 类的 Fluent NHibernate 映射类。该类还不存在, 将在 `OSIM.Core` 项目中创建它。

`ExportTo` 方法指示 Fluent NHibernate, 将这一映射信息导出到 C 盘上 `Temp` 目录中的一个文件。如果 Fluent NHibernate 生成的“对象—数据”表映射不正确或者与预期不符, 该文件可能会很有帮助。

我利用对 `ExposeConfiguration` 方法的调用来销毁和重新生成数据库架构, 它是通过调用一个新创建的 `SchemaExport` 对象的 `Create` 方法来管理的。它对于集成测试很有帮助, 因为在开发期间经常会改变对象结构, 如果必须人工修改数据库表以存储这些类的数据, 会非常麻烦。它还确保我们总是使用新的、洁净的数据库架构实例。这意味着在重设测试环境时可以少做一些工作。在转向生产时, 要确保清除了对 `ExposeConfiguration` 的调用, 否则可能存在删除生产数据库的风险。`ExposeConfiguration` 方法和 `SchemaExport` 类的功能非常强大。除本例之外, 要想全面了解 Fluent NHibernate 的应用, 可参阅其文档。

`BuildSessionFactory` 是最后一个调用。它根据提供的配置信息, 开始 `ISessionFactory` 实例的创建过程。

由于集成测试将数据存储在数据库中，因此需要创建一个开发数据库。该项目使用 Microsoft SQL Server 2008 R2 作为数据库管理系统(DBMS)。创建一个名为 OSIM.Dev 的数据库，并接受其所有正常的创建默认选项，如图 8-4 所示。



图 8-4

4. 创建 Fluent NHibernate 映射

下一步是为 ItemType 类创建“对象-数据表”映射。和所有 ORM 框架一样，Fluent NHibernate 需要某种类型的映射，告诉 ORM 如何将类的公共字段转换为数据表的字段。传统的 NHibernate 使用 XML 文件，它可能很难理解，而且容易产生缺陷。Fluent NHibernate 是 NHibernate 框架的扩展，允许开发人员用 C#或 VB.NET 创建这些映射。这是它相对于传统 NHibernate XML 映射的主要优点。首先，大多数开发人员更熟悉、更习惯使用 C#或 VB.NET。使用自己更习惯、更熟练的编程语言总是更可取的做法。由于这些映射文件会提前编译，而不是在运行时读取，因此在 Fluent NHibernate 映射中很少出现错误。

这些映射的位置非常重要。尽管它们依赖于 OSIM 应用程序中的实体类型，但从技术上来说，它们是持久化层的组成部分。因此，希望把它们与 OSIM.Core 项目的库和其他持久化逻辑放在一起。

提示：

现在，你可能会问自己这样一个问题：“为什么要把持久化逻辑放在核心域项目呢？”作为开发人员，我们受到的教育是把应用程序的所有部分都分散在不同的应用程序集中。之所以需要进行这种分离，是因为在对一个程序集进行修改时，可以很轻松地转入另一个程序集中。事实上，程序集特有的这种“即插即用”功能有些夸大了。在许多情况下，无论向应用程序基本代码中引入多大程度的分离和抽象，对一个程序集进行修改后也总是需要对大多数甚至所有其他程序集进行修改。



这 and 把持久化逻辑保存在核心域项目中有什么关系呢？答案就是“简单性”。是的，应用程序的体系结构设计应当使不同的功能保持分离，如数据访问功能与表示功能。但这种设计和所有体系结构设计选项一样都是有成本的。我发现，将表示逻辑与域服务分离的价值要大于分离所产生的成本(主要是指复杂度增大)。对于一个小型部门应用程序，就像该示例中的 OSIM 应用程序一样，我发现将域服务和持久化逻辑保持分离的好处并没有高于相应的成本(同样是指复杂性)。

当然，可以将该应用程序的体系结构设计为将持久化逻辑保存在自己的独立程序集中。但这样就需要在体系架构中进行其他修改，以避免出现循环依赖。在大型应用程序中，不同组件需要以不同比例缩放，所以使它们保持分离是很重要的。但大多数应用程序都是一些小型部门应用程序，即使是在大型企业内，它们也只是部署给一小部分用户使用。在这些情况下，实用主义就胜出了。如果需要将持久化层或任意其他相关层提取到自己的程序集中，总是可以随时重构这一应用程序。

还会看到一种观点：这样做时，会增大从一个持久框架转换到另一个持久框架的难度。我认为这是站不住脚的；如果有人找到我，并希望这样做，我需要有很好的理由来支持这一转换。主要问题不在于持久化层与应用程序其余部分的分离程度，改变持久框架不像更换遥控器电池那样简单。必然要进行一些大幅变动。不只是提供新的 API，还要处理框架之间的一些变化，如事务支持和连接池等。在面对这些问题时，我经常向我的客户和开发人员提醒一句古老的谚语：小车不倒尽管推！

首先在 OSIM.Core 项目的 Persistence 文件夹中创建一个名为 Mappings 的文件夹，如图 8-5 所示。

接下来在这个 Mappings 文件夹中创建一个类，如图 8-6 所示。



图 8-5

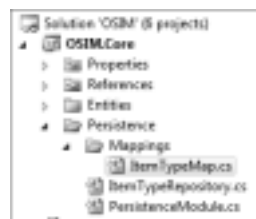


图 8-6

Visual Studio 在 ItemTypeMap.cs 文件中创建一个存根类。



```
namespace OSIM.Core.Persistence.Mappings
{
    public class ItemTypeMap
    {
    }
}
```

ItemTypeMap.cs

为了使它成为 Fluent NHibernate 映射类, 需要为 OSIM.Core.Entities 名称空间(ItemType 声明所在位置)和 FluentNHibernate.Mapping 名称空间(FluentNHibernate 映射类的定义所在位置)添加 using 语句。然后, 需要改变 ItemTypeMap 类的声明, 使它继承 Fluent NHibernate 基类 ClassMap<T>。ClassMap<T>是一个泛型类, 需要向它传递一个类型作为泛型类型参数, 也就是希望这个类成为的类型(映射)——在本例中, 为 ItemType:



```
using FluentNHibernate.Mapping;
using OSIM.Core.Entities;

namespace OSIM.Persistence.Mappings
{
    public class ItemTypeMap : ClassMap<ItemType>
    {
    }
}
```

ItemTypeMap.cs

Fluent NHibernate 希望在每个映射类中都有一个默认构造函数, 提供该映射类所映射实体的映射信息。下面说明如何把它添加到 ItemTypeMap:



```
public class ItemTypeMap : ClassMap<ItemType>
{
    public ItemTypeMap()
    {
    }
}
```

ItemTypeMap.cs

在为 ItemType 创建映射之前, 再来看看该类的定义。下面是 ItemType 当前的定义:



```
namespace OSIM.Core.Entities
{
    public class ItemType
    {
        public virtual int Id { get; set; }
    }
}
```

```
    }
}
```

ItemTypeMap.cs

现在没有更多要说的了。事实上，ItemType 中的唯一字段是 Id 字段，作为该实体的主键。这个定义多少显得有些缺少活力，这是因为当前的单元测试不需要 ItemType 中的任何其他字段。不过，为了实现本示例的目的，添加了一个名称字段，因为这样可以使该例的意义更明显一些：



```
namespace OSIM.Core.Entities
{
    public class ItemType
    {
        public virtual int Id { get; set; }
        public virtual string Name { get; set; }
    }
}
```

ItemTypeMap.cs

这时可能会奇怪，为什么这些字段都声明为虚字段。比较简短的解释是：Fluent NHibernate 在创建数据表架构时使用了反射作为其过程的一部分，并向数据库中存储数据和从中读取数据。如果不将这些字段声明为虚字段，Fluent NHibernate 就不能“变自己的戏法”了。

即使使用了一段时间的 Fluent NHibernate，有时也会不可避免地忘记这一需求。有一条很好的经验法则：如果出现一个似乎没有什么意义的映射错误，那首先要进行检查，确保正在映射的类的所有公共成员都被声明为虚成员。这通常就是问题所在。

终于可以为 ItemType 创建映射了。返回到 ItemTypeMap 类，向 ItemTypeMap 的默认构造函数中添加 Id 和 Name 的映射规则：



```
public class ItemTypeMap : ClassMap<ItemType>
{
    public ItemTypeMap()
    {
        Id(x => x.Id);
        Map(x => x.Name);
    }
}
```

ItemTypeMap.cs

Id 和 Map 是两个最常用的 Fluent NHibernate 映射命令。Id 指定实体类中的哪个或哪些字段被映射到数据表的主键。Map 命令只是告诉 Fluent NHibernate，将实体类中的字段映

射到数据表中一个相同名称和类型的字段。该映射类的结果就是创建一个名为 ItemTypes 的表，它有两个列：Id(主键)和 Name。对于关系和索引等，存在一些扩展，在项目后面将会用到它们。另外，一些复杂命令允许完成一些稍微复杂的操作，例如在同一个表中存储两个相似的类型，创建复杂关系等。甚至当数据库中的表或字段名称与应用程序代码中的类或字段名称不匹配时，也可以将数据存储到一些表或字段中。所有这些内容都在 Fluent NHibernate 文档中介绍。

5. 创建集成测试

下一步是将该数据库的连接字符串放在配置文件中。首先，向 OSIM.IntegrationTests 项目添加 app.config 文件，如图 8-7 所示。

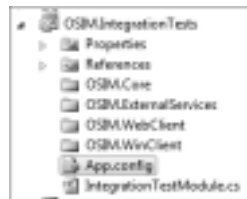


图 8-7

在 OSIM.IntegrationTests 项目中创建这个文件的原因在于：运行该集成测试时，OSIM.IntegrationTests 程序集变成了可执行上下文。因此，.NET 运行时通过 NUnit 测试运行程序查找 OSIM.IntegrationTests 程序集的配置文件。如果它不能找到这个配置文件，将会产生运行时错误。

接下来，在 App.config 文件中创建 appSetting 部分，将一个名为 localDb 的键放在 appSetting 部分，其中包含对 OSIM.Dev 数据库的连接字符串：



```
< ?xml version="1.0" encoding="utf-8" ? >
< configuration >
  < appSettings >
    < add key="localDb" value="Data Source=JAMES-PC;Initial Catalog=OSIM.Dev;
      Integrated Security=True" / >
  < /appSettings >
< /configuration >
```

App.config

现在所有数据库和 ORM 工作都已完成，终于可以考虑编写测试了。与单元测试不一样的是，对于要首先看到测试失败这一点，不是特别重视；事实上，在许多情况下这些测试都是立即通过的。这是因为单元测试是用来驱动开发的(它们演示需要编写的代码)，但集成测试的目的则是验证(我们预计这些代码已经存在，而且可以正常工作；只是希望验证一下这一预期)。

如果在 OSIM.IntegrationTests 项目中还没有 OSIM.Core.Persistence 文件夹，那就马上添加一个，如图 8-8 所示。

现在已经为 ItemTypeRepository 测试准备了位置，要创建一个新的 cs 文件，用来保存测试类，如图 8-9 所示。

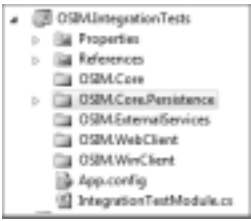


图 8-8



图 8-9

Visual Studio 在 ItemRepositoryTests.cs 文件中创建其正常的存根类：



```
namespace OSIM.IntegrationTests.OSIM.Persistence
{
    public class ItemTypeRepositoryTests
    {
    }
}
```

ItemTypeRepositoryTests.cs

根据本项目采用的 BDD 命名风格 ,将 ItemTypeRepositoryTests 类的名称修改为 when_using_the_item_type_repository，使它继承 Specification 基类。为使该类编译通过，还需要向 OSIM.UnitTests 项目以及 NBehave.Spec.NUnit 和 NBehave.Spec.Framework 程序集添加引用。还需要添加一个 using 语句，以包含 OSIM.UnitTests 名称空间。



```
using OSIM.UnitTests;

namespace OSIM.IntegrationTests.OSIM.Persistence
{
    public class when_using_the_item_type_repository : Specification
    {
    }
}
```

ItemTypeRepositoryTests.cs

和第 7 章为单元测试创建的 when_working_with_the_item_type_repository 类相似，该类也有两个目的。它的定义有助于为测试建立 BDD 风格的名称，完整、准确地描述正在测试的操作。它还提供了一个基类，在这个类中可以执行 ItemTypeRepository 的所有集成测试所需要的公共设置步骤。现在，离开 when_using_the_item_type_repository 类，建立将包含实际集成测试的类。

集成测试和单元测试之间的另一个区别是，单元测试一次应当仅关注一个步骤或一部分功能，而集成测试一次测试多个步骤是完全可以接受的。现在这个测试就是一个很好的例子。我们希望验证：可以使用 ItemTypeRepository 向数据库中写入数据并能从中读取数

据。在一个自动化测试中,无法验证可以在不添加任何东西的情况下从数据库中读取内容;如果不回读取值,也无法验证可以正确地向数据库中写入数据,因为这样无法确保正确保存写入值。

因此,集成测试的测试名称,包括要为 ItemTypeRepository 编写的测试,其针对性要弱于单元测试的名字。本例中,测试类的名称(and_attempting_to_save_and_read_a_value_from_a_datastore)描述了为确保 ItemTypeRepository 能够正确工作而必须执行的两个操作:



```
public class and_attempting_to_save_and_read_a_value_from_the_datastore :
    when_using_the_item_type_repository
{
}
```

ItemTypeRepositoryTests.cs

测试方法名称则描述了这两个步骤的预期输出:



```
[Test]
public void then_the_item_type_saved_to_the_database_should_equal_the_item_type
    _retrieved()
{
}
```

ItemTypeRepositoryTests.cs

由于没有在 OSIM.IntegrationTests 项目中引用 NUnit 程序集,因此现在来完成这一工作,如图 8-10 所示。

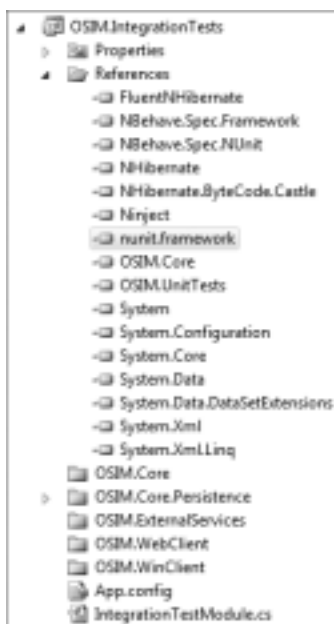


图 8-10

还需要添加 using 语句, 以将 NUnit.Framework 名称空间添加到 ItemTypeRepositoryTests.cs 文件中:



可从
wrox.com
下载源代码

```
using NUnit.Framework;
```

ItemTypeRepositoryTests.cs

执行集成测试的步骤要比执行单元测试的步骤稍多一些, 也要稍微复杂一些。这是合理的, 因为集成测试覆盖了更广泛的应用程序功能。在本质上, 我们知道希望创建和填充 ItemType 对象, 将它存储在数据库中, 将相同物品类型检索到 ItemType 的不同实例中, 然后验证这些字段是互相匹配的。then_the_item_type_saved_to_the_database_should_equal_the_item_type_retrieved 方法的代码实际上相当简单。我们只是希望对所检索的 ItemType(实际)实例的字段进行迭代, 并确保这些值与 ItemType 对象存储(期望)实例中的相应值匹配:



可从
wrox.com
下载源代码

```
[Test]
public void
then_the_item_type_saved_to_the_database_should_equal_the_item_type_retrieved()
{
    _result.Id.ShouldEqual(_expected.Id);
    _result.Name.ShouldEqual(_expected.Name);
}
```

ItemTypeRepositoryTests.cs

为完成这一代码, 使其能够编译通过, 需要向 and_attempting_to_save_and_read_a_value_from_the_datastore 类添加 _expected 和 _result 的声明。还需要为 NBehave.Spec. NUnit 和 OSIM.Core.Entities 名称空间添加 using 语句。下面是 ItemTypeRepository.cs 文件到目前为止的完整内容:



可从
wrox.com
下载源代码

```
using NBehave.Spec.NUnit;
using NUnit.Framework;
using OSIM.Core.Entities;
using OSIM.UnitTests;

namespace OSIM.IntegrationTests.OSIM.Persistence
{
    public class when_using_the_item_type_repository : Specification
    {
    }

    public class and_attempting_to_save_and_read_a_value_from_the_datastore :
        when_using_the_item_type_repository
    {
        private ItemType _expected;
        private ItemType _result;

        [Test]
```

```

        public void
then_the_item_type_saved_to_the_database_should_equal_the_item_type
_retrieved()
    {
        _result.Id.ShouldEqual(_expected.Id);
        _result.Name.ShouldEqual(_expected.Name);
    }
}

```

ItemTypeRepositoryTests.cs

下一步是为测试编写 `Because_of` 方法。在本例中, 需要使用 `ItemTypeRepository` 将 `ItemType` 类的实例保存到数据库中。然后, 使用同一个 `ItemTypeRepository`, 将刚刚保存到数据库中的相同数据(使用由 `ItemTypeRepository` 返回的 `Id` 值)检索到 `ItemType` 的一个不同实例中, 它是由 `_result` 成员变量表示的:



```

protected override void Because_of()
{
    var itemTypeId = _itemTypeRepository.Save(_expected);
    _result = _itemTypeRepository.GetById(itemTypeId);
}

```

ItemTypeRepositoryTests.cs

还需要添加 `using` 语句, 以导入 `OSIM.Core.Persistence` 名称空间:



```

using OSIM.Core.Persistence;

```

ItemTypeRepositoryTests.cs

接下来的全部工作就是编写 `Establish_context` 方法。`Establish_context` 方法中的第一项任务就是创建 `ItemTypeRepository` 的一个实例, 并将它指定给 `_itemTypeRepository` 成员变量。因为 `Ninject` 是本例中的依赖项注入框架, 所以需要创建 `Ninject` 标准内核的一个实例, 并向它提供 `IntegrationTestModule` 类的实例。它包含了为集成测试创建 `ItemTypeRepository` 的规则:



```

private StandardKernel _kernel;

protected override void Establish_context()
{
    base.Establish_context();
    _kernel = new StandardKernel(new IntegrationTestModule());
}

```

ItemTypeRepositoryTests.cs

要使用 `Ninject` 标准内核, 需要为 `Ninject` 名称空间添加 `using` 语句:



可从
wrox.com
下载源代码

```
using Ninject;
```

ItemTypeRepositoryTests.cs

现在需要向 Ninject 提出要求，让其提供一个实现 `IItemTypeRepository` 接口的类的实例。在为 `_kernel` 成员变量实例化 `StandardKernel` 对象时，为它提供了 `IntegrationTestModule` 类的一个实例。`IntegrationTestModule` 类拥有所有规则和步骤，供 Ninject 创建实现 `IItemTypeRepository` 接口的类的实例，以进行集成测试。只需让 `_kernel` 对象提供实现 `IItemTypeRepository` 接口的对象。Ninject 负责所有创建步骤：



可从
wrox.com
下载源代码

```
protected override void Establish_context()
{
    base.Establish_context();

    _kernel = new StandardKernel(new IntegrationTestModule());
    _itemTypeRepository = _kernel.Get < IItemTypeRepository > ();
}
```

ItemTypeRepositoryTests.cs

在继续前进之前，停下来从实用的角度来看目前的代码。`ItemTypeRepository.cs` 文件包含了 `when_using_the_item_type_repository` 类，`and_attempting_to_save_and_read_a_value_from_the_datastore` 就是继承自该类。进行以下假定是比较合理的：所有为 `ItemTypeRepository` 执行集成测试的类都会以某种方式继承 `when_using_the_item_type_repository` 类。还可以合理地假定：对 `ItemTypeRepository` 类执行集成测试的大多数(甚至全部)类都需要一个实现 `IItemTypeRepository` 接口的对象的实例，该接口与其他的 `ItemTypeRepository` 测试类相同。因此，下面的做法也是合理的：将 `_kernel` 和 `_itemTypeRepository` 成员变量的声明以及实例化代码移到 `when_using_the_item_type_repository` 基类中。

在移动了这些声明，并在 `when_using_the_item_type_repository` 类中创建了 `Establish_context` 方法之后，`when_using_the_item_type_repository` 和 `and_attempting_to_save_and_read_a_value_from_the_datastore` 类应当类似如下：



可从
wrox.com
下载源代码

```
public class when_using_the_item_type_repository : Specification
{
    protected IItemTypeRepository _itemTypeRepository;
    protected StandardKernel _kernel;

    protected override void Establish_context()
    {
        base.Establish_context();

        _kernel = new StandardKernel(new IntegrationTestModule());
        _itemTypeRepository = _kernel.Get < IItemTypeRepository > ();
    }
}
```

```

public class and_attempting_to_save_and_read_a_value_from_the_datastore :
when_using_the_item_type_repository
{
    private ItemType _expected;
    private ItemType _result;

    protected override void Establish_context()
    {
        base.Establish_context();
    }

    protected override void Because_of()
    {
        var itemTypeId = _itemTypeRepository.Save(_expected);
        _result = _itemTypeRepository.GetById(itemTypeId);
    }

    [Test]
    public void then_the_item_type_saved_to_the_database_should
        _equal_the_item_type_retrieved()
    {
        _result.Id.ShouldEqual(_expected.Id);
        _result.Name.ShouldEqual(_expected.Name);
    }
}

```

ItemTypeRepositoryTests.cs

运行测试前的最后一步是为 `_expected` 成员变量创建 `ItemType` 类的一个实例，需要用它将数据保存到数据库中：



```

protected override void Establish_context()
{
    base.Establish_context();

    _expected = new ItemType {Name = Guid.NewGuid().ToString()};
}

```

ItemTypeRepositoryTests.cs

我喜欢使用 `Guids` 作为字符串测试数据。每次可以很容易地获得一个随机值。因为它总会得到一个新的不同值，所以不需要担心数据库中的旧数据会因为某种原因而没有清理干净，从而破坏测试结果。当然，要使用 `Guid`，还需要添加(如果还没有添加)一个用于包含 `System` 名称空间的 `using` 语句：



```
using System;
```

ItemTypeRepositoryTests.cs

现在剩下的就是运行测试。从图 8-11 可以看出，没有另外编写任何代码，这一测试就

通过了。



图 8-11

如果希望进一步验证，我只是需要使用我最喜欢的数据库管理工具(对该例来说，使用 SQL Management Studio)，并查看数据库和表中的内容(如图 8-12 所示)。



图 8-12

我在编写自己的单元测试时，在编写代码之前先编写了测试。这里当然可以采用相同的方法。事实上，我要编写的唯一代码就是先把一个实现 `ISessionFactory` 的对象注入 `ItemTypeRepository` 类的代码，用于处理创建该 `ItemTypeRepository` 类的 `Ninject` 模块，以及连接到开发数据库的 `Fluent NHibernate` 配置。前面刚刚提到的这 3 个行为和驱动单元测试的用户情景及功能之间的区别在于：为了使集成测试通过，需要实现的功能不是增加业务价值的代码，而是为增加业务价值的代码提供支持的代码。

这并不是说，这一代码和功能不重要。如果用户不能将其数据保存在数据存储并在以后检索，那应用程序本身就不能用。但如果编写一个应用程序用来处理抵押支付或者跟踪处方药的库存，那业务用户更多的是关注应用程序中实现的业务规则，而不是用什么框架来将这些信息存储到数据库，或者使用什么模式在系统中创建这些对象。

我倾向于首先编写集成测试的另一个原因，坦白来说就是习惯。作为咨询师，我在日常工作中要监督大量年轻、经验不足的开发人员，甚至是一些经验丰富的老手也没有接触过 `ORM` 和 `IOC` 容器等的概念。在我作为首席开发人员的日子里，我希望确保坚持使用这些工具和概念。如果把编写集成测试的任务随便交给一位没有任何 `IOC` 容器经验的开发人员，他很可能会手工创建某种对象工厂，甚至创建静态绑定的成员变量(这更糟)。在这种情况下及另一情况下(集成测试随单元测试的不同而变化，至少对我来说是这样的)，我希望开发人员，特别是那些对这些新工具和新技术没有太多经验的开发人员，能够记住这些工具和技术的用法。当他们编写单元测试时，我宁可选择相反的做法；他们应当仅考虑满足功能或用户情景确定的直接需求。任何其他考虑都是次要的，可以在集成时通过重构来处理。在与系统中的其他部分组合在一起时，我希望我的团队确保自己使用了正确的黏合剂。

8.2.3 端对端集成测试

单元测试的目的是通过创建一些可执行的自动化测试来驱动代码的开发，这些测试可以验证所开发的应用程序满足了由功能和用户情景定义的业务需求。集成测试的目的是确保应用程序的各个组件能够协同工作，构成更大的应用程序。端对端测试用来验证所开发的应用程序满足了功能或用户情景的所有需求，而且其功能是完整的，被正确地集成在一起。单元测试覆盖了业务功能的一个特定单元，而集成测试则主要针对组件之间的接合部。端对端测试覆盖了整个系统，既包括最接近前端或用户界面的部分，也包括后端数据存储、Web 服务或该应用程序可能拥有的任意其他依赖项。

端对端测试是非常有用的，其原因包括：它验证了应用程序在所有层上都实现了完整集成；还从更宏观的角度确保满足了功能或用户情景的要求。

和单元测试、集成测试一样，端对端测试也有自己的一些特点。与单元测试和集成测试相比，任何一个给定测试套件中的端对端测试都是相对较少的。单元测试涵盖的是具体的小型业务任务，而端对端测试针对的是完整的业务 workflow。因此，单元测试的针对性较强，而端对端测试则是面向更广的功能。与集成测试不同的是，端对端测试并不关心具体的接合部。事实上，它们应当完全不知道这些接合部的存在。大多数端对端测试应当首先创建一个尽可能接近用户界面的访问点的实例。如果该用户界面可以使用，例如在 ASP.NET MVC 应用程序中、用 WPF 或 Silverlight 编写的采用 MVVM 模式的应用程序中，或者是 WCF/ASMX 服务中，那就应当以该用户界面作为端对端测试的起点。有一些功能是层叠在其他层中的类上，端对端测试只不过就是允许对象调用这一类功能。端对端测试并不关心使用了 `ItemTypeRepository`。它关心的是当用户提供了正确的信息，并调用输入窗体上 Save 按钮提供的代码时，就可以保存数据，并返回正确的结果。

8.2.4 使各类测试保持分离

应用程序的全套测试包括单元测试、集成测试和端对端测试，这些测试分别用于不同目的。开发人员应当经常运行单元测试。运行集成测试的频繁程度要低一些，而运行端对端测试的频繁程度则还要更低一些。为了很好地控制运行哪种测试及其运行频繁程度，使各类测试相互保持分离非常重要。为此，我经常在每个应用程序的 Visual Studio 解决方案创建两个测试项目：一个单元测试项目和一个集成测试项目(它还包含我的端对端测试)。将它们放在不同项目中，从而放在不同的程序集中，可以更好地控制何时运行这些测试。在 8.3 节还将更详细地对此进行讨论。

8.3 运行集成测试的时机和方式

集成测试，特别是端对端测试，其运行时间可能要比单元测试长得多，这是因为它们需要与应用程序的外部依赖项交互。如果两个或更多个开发人员尝试同时运行这些测试，

还可能因为需要与外部依赖项交互而导致问题。因此，集成测试和端对端测试不会也不应当像单元测试那样频繁运行。

开发人员在运行集成测试和端对端测试时，应当已经完成或接近完成一项功能或用户情景。为了确保不会向应用程序的当前功能中引入任何缺陷，这一要求是必要的。有些开发人员或开发团队会制定一条实践规范：开发人员不需要运行与其当前工作、函数或用户情景无关的集成测试和端对端测试。如果对应用程序进行了分割，能够很轻松在判断哪些测试是相关的，哪些测试是不需要运行的，那这种规范就是一种很好的做法。至少，在向主代码库中提交任何代码之前，必须运行这些测试。

集成测试和端对端测试还应当作为 CI 过程的一部分加以运行。CI 过程的一个目的就是确保在向 QA 人员发布应用程序之前，确保应用程序的所有组件和部分正确地集成在一起，能够正常工作。这样就可以确保 QA 人员不需要花费时间来测试已经被标记为不能正常工作的应用程序。作为 CI 过程的一部分来运行这些测试的另一个好处是，在运行这些测试时不会占用开发资源。作为建立过程的一部分，CI 服务器可以执行需要长时间运行的测试，并在有测试失败时，报告是哪些测试失败。与此同时，开发人员可以继续处理其他功能或用户情景。如果应用程序的集成测试和端对端测试的运行需要很长时间，这种做法对于其开发团队是很有吸引力的。不用说，在 CI 过程中失败的所有测试都应当成为应用程序开发团队的优先解决对象。所有其他行为都应当被暂停，直到这一建立过程再次处于“全绿”状态为止。

8.4 本章小结

集成测试对于保证应用程序的质量非常重要。单元测试确保各个组件满足在功能和用户情景中列出的业务需求。集成测试确保这些独立组件可以正确地一同工作，创建一个功能应用程序。

在编写集成测试时，希望主要关注应用程序中各个组件之间的接合部。除了内部各组件之间的接合部之外，还希望考虑应用程序与外部资源之间的接合部，这些外部资源包括数据库、Web 服务、文件系统和其他任何不专属于应用程序组成部分的资源。

和单元测试一样，集成测试希望在可预测状态的环境中执行。单元测试依赖于被注入的依赖项对象和模拟(mocking)来提供存根(stubbed)环境，而集成测试与之不同，它必须使用实际外部资源对象的测试版本。作为集成测试套件的一部分，需要确保在运行集成测试之前，使测试执行环境返回一种一致状态。

单元测试用来测试应用程序各个组件的内部功能，而集成测试用来测试应用程序中各个组件之间的接合部。端对端测试是一种特殊的集成测试，用来验证应用程序代码可以根据应用程序的用户情景来执行整个业务 workflow。应用程序测试套件将包含许多单元测试和集成测试，它们中的每一个会关注一小部分功能。与此相对，应用程序测试套件中将包含较少的端对端测试，而每个端对端测试都将涵盖较大的功能范围。

与单元测试相比，集成测试和端对端测试的执行需要花费较长时间，这是因为它们需要使用非模拟组件和外部资源。由于其运行时间较长，所以建议开发人员不要像运行单元测试那样频繁地运行集成测试和端对端测试。在开发一项功能或用户情景期间，应当定期运行集成测试。在开发人员将代码提交到主源代码库之前，应当执行端对端测试。集成测试、端对端测试和单元测试，都应当作为开发团队 CI 过程的一部分加以运行。只要建立过程中存在未能通过的测试，就应认为这一建立过程是失败的，开发团队的首要任务就是使这些测试得以两次通过，并完成建立过程。总体目标就是向用户提交高质量的软件。而这一点只能通过成功的 CI 过程来实现。

第 部分

TDD 方 案

- 第 9 章 Web 上的 TDD
- 第 10 章 测试 WCF 服务
- 第 11 章 测试 WPF 和 Silverlight 应用程序



第 9 章

Web 上的 TDD

作者：Jeff McWherter

本章内容

- 采用 Web 窗体的 TDD
- 采用 MVC 的 TDD
- 采用 JavaScript 的 TDD

前面各章已经讨论了一些工具和理论，描述了(测试驱动开发)到底是什么。现在应当将这些知识在实例中加以运用，并把 TDD 带到许多软件开发人员的生活世界——Web 中。

你可能在想，Web 框架(主要是 ASP.NET Web 窗体)没有很好地区分各种关注事项。在应用 TDD 时，要时刻注意不要违反单一职责原则。从混合在 ASPX 文件中的 C#或 Visual Basic 代码，到页面之间复制的公共 JavaScript 函数，许多 Web 框架都没有遵循前面各章提到的大量规则。本章要讨论一些模式，可用于更轻松地测试 Web 应用程序，其中包括那些没有很好地区分关注事项的框架。本章将首先从 ASP.NET Web 窗体入手，然后转向相对较新的微软框架——ASP.NET MVC，其中将介绍一些技巧，要提高 Web 应用程序的可测试性，并使用 TDD 创建它们，会用到这些技巧。

关于如何针对 Web 进行测试，已经出版了一些专门书籍，如 Jeff McWherter 和 Ben Hall 编写的 *Testing ASP.NET Web Applications*(Wrox, 2009, ISBN: 978-0-470-49664-0)。本章概括了有关这一主题的要点。到本章结束时，你将会明白如何使用 TDD 来开发 Web 应用程序。

9.1 ASP.NET Web 窗体

2002 年 1 月, ASP.NET 发布, 其 .NET Framework 的版本为 1.0。这次发布对于使用微软技术的 Web 开发人员来说, 是一个重要的里程碑, 使他们能够利用 Visual Basic .NET 和 C# 等语言开发出功能强大的 Web 应用程序。ASP.NET 中包含一些可用于快速开发 Web 应用程序的工具, 在开发过程中, 可以利用一些内置控件为开发人员考虑基本的创建、读取、更新和删除(CRUD)操作。随 ASP.NET 框架一起发送的组件使我们能够把控件拖放到应用程序中。利用这一功能可以加快开发进度。但在这个过程中, 很多应用程序的体系结构和可测试性从来未被考虑。当开发人员转到不同公司或不同项目之后, 这些 Web 应用程序仍然留在原处, 所以需要为以这种方式开发的应用程序一次又一次地打补丁, 其维护成本也不断上升。

尽管到今天仍然存在这些问题, 但可以使用一些技术和模式来对 ASP.NET Web 窗体应用程序进行单元测试。

Web 窗体组织结构


Web 窗体由两部分组成。ASPX 文件包括 HTML、CSS、JavaScript 和 ASP.NET 标记。第二部分是代码隐藏文件, 这是一个 CS 文件或 VB 文件, 其中包含了可执行代码。现在花点时间来讨论一下这些文件中应当包含什么内容。

1. ASPX 文件

ASPX 文件是用户看到的 Web 应用程序页的视图。因此, 在这些文件中应当不包含业务逻辑。ASPX 文件应当清洁、简单, 便于 Web 和图片设计师阅读。不是所有人都可以将 ASPX 文件发送给设计人员, 但如果你能这样做, 那一定希望这些设计人员能够理解这些文件。如果 ASPX 文件中存在访问数据库的逻辑, 不仅会违反有关分离关注事项的规则, 还会使别人难以阅读自己的代码。ASPX 文件中不应包含 <% 标记。

2. 代码隐藏文件

可以把代码隐藏文件看作视图呈现器的控制器。代码隐藏文件挂钩到 ASP.NET 页的生存周期中, 与 ASP.NET 运行时有非常紧密的依赖关系, 从而使这些文件的测试变得很困难。因此, 应当使这些代码隐藏文件尽可能保持“单薄”, 使其中仅包含“黏合代码”。Visual Studio 和 ASP.NET Web 窗体目前的默认工作方式鼓励开发人员在这些文件中存储大量逻辑, 这实际上是与 TDD 要求的模式相悖的。



提示：

“黏合代码”对于应用程序的功能没有什么贡献。黏合代码的目的只是将在通常状态下可能不兼容的代码部分“黏合”在一起。由于黏合代码非常简单，所以大多不需要测试。

在 ASP.NET Web 窗体中，为保持代码隐藏文件的整洁，并很好地区分关注事项，最佳方法之一就是遵循“模型-视图-呈现器(MVP)”设计模式，如图 9-1 所示。

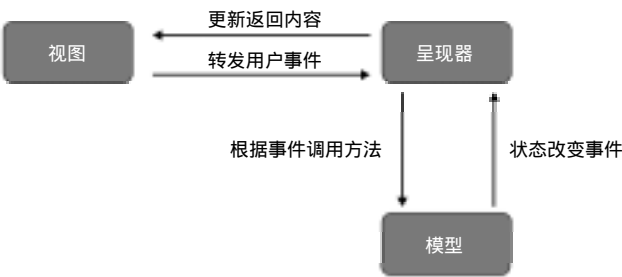


图 9-1

从其核心来看，呈现器充当中间层，类似于“模型-视图-控制器(MVC)”模式(本章后面会介绍)中的控制器对象。呈现器检索数据、实现其持久化，然后设置其格式，以在视图中显示它。视图是显示数据的界面，模型则表示数据或域模型。MVP 模式从 20 世纪 90 年代早期就已经出现，是一种遵循 SOLID 设计原则的流行方法。

3. 用 MVP 和 Web 窗体实现测试驱动开发

该示例创建一个应用程序，用来列出最近阅读过的书籍。我们关注如何向用户显示这一数据。图 9-2 所示为要建立的内容。



图 9-2

项目布局

在开始创建第一个测试之前,应当首先确定项目的组织结构。创建一个 ASP.NET Web 应用程序,并将其命名为 Wrox.BooksRead.Web。这个名字是从标准的 Company.ProjectName.Component 命名架构创建的。要使项目的组织结构保持清晰、整洁有助于后期维护。在为项目命名时,其关键在于保持一致性。Wrox.BooksRead.Web 项目中没有添加任何新奇的东西;它目前就是一个默认的 ASP.NET Web 应用程序。

还需要添加另一个 Windows 类库项目 Wrox.BooksRead.Tests。该项目将保存为这个 Web 应用程序准备的测试。该应用程序中包括一个 Lib 文件夹,其中保存用于测试 Wrox.BooksRead.Web 应用程序的外部二进制文件。还需要添加 NUnit 和 Rhino Mocks(一种流行的模拟框架)。这时,图 9-3 中显示的所有项目都应当编译通过,现在可以开始创建第一个测试了。

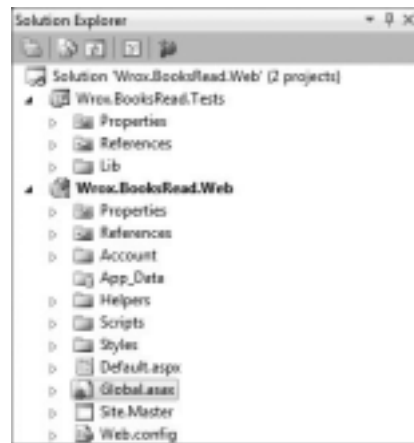


图 9-3

第一个测试

在使用 MVP 或 MVC 等模式时,通常难以选择从哪里开始。控制器或呈现器通常是最佳开始位置,这是由每个项目中包含的应用程序逻辑数目决定的。也可以首先处理有关模型的测试,但处理控制器/呈现器有助于确定模型应当是什么样的。下面是第一个测试:



可从
wrox.com
下载源代码

```
public class when_using_the_books_read_controller : Specification
{
}

public class and_getting_a_list_of_books : when_using_the_books_read_controller
{
}

public class and_when_calling_getdata_bind_should_be_called :
    and_getting_a_list_of_books
{
    IDisplayBooksReadView _view;
    DisplayBooksReadController _controller;

    protected override void Establish_context()
    {
        base.Establish_context();
        _view = MockRepository.GenerateMock < IDisplayBooksReadView > ();
        _view.Expect(v => v.Bind());
        _controller = new DisplayBooksReadController(_view);
    }

    protected override void Because_of()
```

```

    {
        _controller.GetData(this, EventArgs.Empty);
    }

    [Test]
    public void then_getdata_should_call_bind()
    {
        _view.VerifyAllExpectations();
    }
}

```

DisplayBookReadControllerTests.cs

在查看该测试时，应当能够找到一些熟悉的部分。在刚看到这个测试时，可能会奇怪这段代码和 Web 窗体应用程序有什么关系呢？在该例中，我们尝试创建一个虚拟视图和一个真正的控制器。然后尝试使用控制器中的方法加载数据，并进行检查，以确保该虚拟视图拥有来自控制器的数据。记住，这里我们正在测试的是控制器的交互，而不是视图。这就是为什么我们可以使用虚拟视图。和预料中一样，该类将不会编译通过，因为还缺少很多内容。

在实现此代码时，在一个文件中实现起来要更容易一些，可以使用诸如 Resharper 或 Refactor 等重构工具，然后选择 Move Type To File 菜单命令，如 9-4 所示。

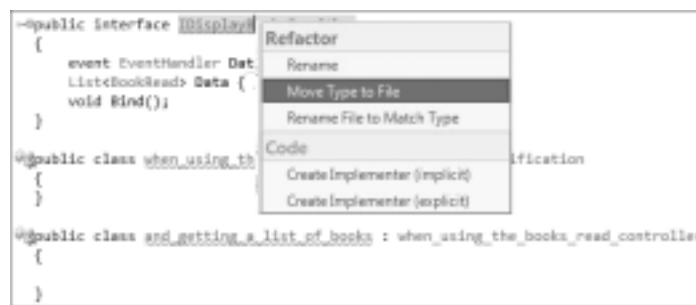


图 9-4

要是使该测试只是能够编译通过，需要添加以下组件：

- IDisplayBooksReadView
- The BookRead object
- DisplayBooksReadController

IDisplayBooksReadView 是视图需要遵守的合约。该视图需要知道将要显示的数据，这些内容是在添加到视图的 Data 字段中处理的。在这种情况下，它刚好是 BookRead 对象的 List，而现在也并不真的关心它看起来是什么样的。Bind 方法将这个数据绑定到视图上的控件，视图使用 DataRequested 事件向控制器请求数据。IDisplayBooksReadView 应当类似于如下：



```
public interface IDisplayBooksReadView
{
    event EventHandler DataRequested;
    List <BookRead> Data { get; set; }
    void Bind();
}
```

IDisplayBooksReadView.cs

这里给出的 BookRead 对象是一个模型。这时，并不需要定义在该对象中包含的内容；只需要创建它，使测试得以通过：



```
public class BookRead
{
}
```

BookRead.cs

DisplayBooksReadController 是该应用程序的呈现器。呈现器负责视图与模型之间的通信。同样，呈现器有一个字段，其中包含了当前正在处理的视图，最终会将它传递给呈现器对象的构造函数，另外还需要一个函数，它能够从所选择的任意类型的数据源中获取数据。下面是 DisplayBooksReadController 的代码：



```
public class DisplayBooksReadController
{
    public IDisplayBooksReadView View { get; set; }

    public DisplayBooksReadController(IDisplayBooksReadView view)
    {
    }

    public void GetData(object sender, EventArgs e)
    {
    }
}
```

DisplayBooksReadController.cs

如图 9-5 所示，该测试仍然失败了，但至少它能够编译了。在试图计算控制器传递给视图的数据项的数字时，会得到一个空引用异常。有两种方法可以解决这一问题，可以使用模拟框架来模拟此数据，也可以直接硬编码一些对象，并说它就是正确的(也称为存根)。该例显示如何实现 ReadBook 数据的存根。在本章后面，将会处理测试、数据库和模拟对象之间的关系。

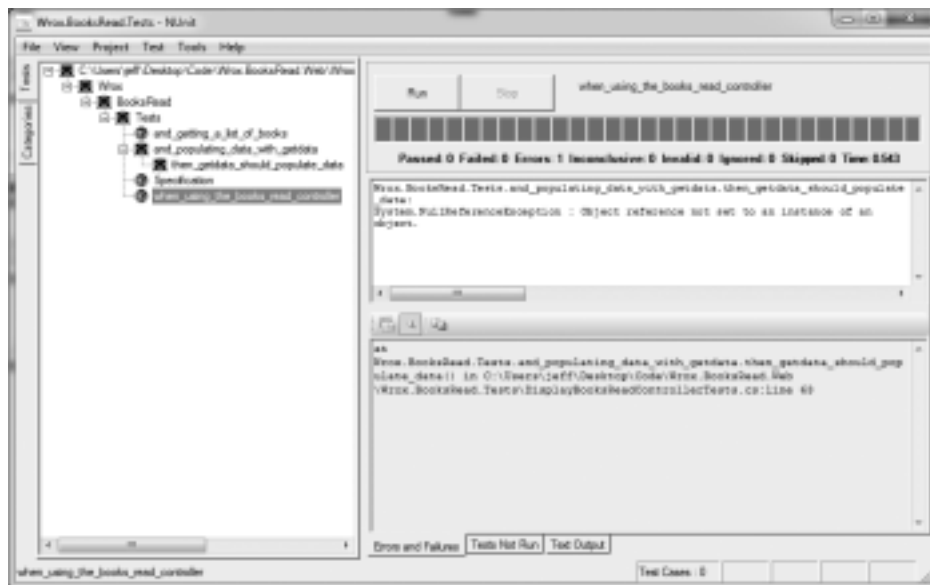


图 9-5

为了完成这一任务，需要实现控制器，代码如下：



可从
wrox.com
下载源代码

```
public class DisplayBooksReadController
{
    public IDisplayBooksReadView View { get; set; }

    public DisplayBooksReadController(IDisplayBooksReadView view)
    {
        View = view;
        View.DataRequested += GetData;
    }

    public void GetData(object sender, EventArgs e)
    {
        View.Data = new List < BookRead > {
            new BookRead{ReadBookId = 1, Name = @" Testing ASP.NET Web
            Applications", Author = "Jeff McWherter/Ben Hall", ISBN =
            "978-0470496640", StartDate = new DateTime(2010, 2, 1),
            EndDate = new DateTime(2010, 2, 12), Rating = 10,
            PurchaseLink = "http://www.amazon.com" },
            new BookRead{ReadBookId = 2, Name = @" Test Driven Development:
            By Example", Author = "Kent Beck", ISBN = "978-0321146533",
            StartDate = new DateTime(2010, 2, 1), EndDate =
            new DateTime(2010, 2, 12), Rating = 9, PurchaseLink =
            "http://www.amazon.com" },
            new BookRead{ReadBookId = 3, Name = "Test2", Author =
            "Author3", ISBN = "22222222", StartDate = new DateTime(2010, 3,
            1), EndDate = new DateTime(2010, 3, 12), Rating = 3,
            PurchaseLink = "http://www.msu.edu" }
        };
    }
}
```

```

    };
    View.Bind();
}
}

```

DisplayBooksReadController.cs

首先从 `DisplayBooksReadController` 类的构造函数入手，将视图属性设计为被传递给控制器的视图，然后将 `DataRequested` 事件连接到 `GetData` 方法。`GetData` 返回 3 个用存根表示的 `ReadBook` 对象。实现该控制器意味着现在可以像下面这样定义模型：



```

public class BookRead
{
    public int ReadBookId { get; set; }
    public string Name { get; set; }
    public string Author { get; set; }
    public string ISBN { get; set; }
    public DateTime StartDate { get; set; }
    public DateTime EndDate { get; set; }
    public int Rating { get; set; }
    public string PurchaseLink { get; set; }
}

```

BookRead.cs

Web Forms MVP 项目的这个第一测试应当可以通过。由于采用了 MVP 模式，因此可以将该类从 ASP.NET 生命周期中隔离开来，并创建一些测试，用来确保该类的功能能够正常使用。

创建更多测试

现在已经完全实现了控制器。由于该例是从控制器入手的，因此现在还没有任何有关视图的测试。需要进行测试，以确保该视图的构造函数连接了所期望的全部事件，目前只有一个事件——`DataRequested` 事件。可以为该视图创建一个 Mock 对象，将它注入到控制器中，然后验证 `DataRequested` 事件已经被连接。记住，`DataRequested` 事件是在控制器中被连接到 `GetData` 函数的。由于已经实现了这一功能，因此该测试应当能够通过，不需要添加更多代码。



```

public class and_wireing_the_events_by_the_constructor :
    and_getting_a_list_of_books
{
    IDisplayBooksReadView _view;

    protected override void Establish_context()
    {
        base.Establish_context();
        _view = MockRepository.GenerateMock < IDisplayBooksReadView > ();
        _view.Expect(v => v.DataRequested += null).IgnoreArguments();
    }
}

```

```

        protected override void Because_of()
        {
            new DisplayBooksReadController(_view);
        }

        [Test]
        public void then_the_constructor_should_wire_up_events()
        {
            _view.VerifyAllExpectations();
        }
    }

```

DisplayBookReadControllerTests.cs

在 GetData 函数和 DataRequested 事件触发之后，控制器调用视图的 Bind 方法。该视图中的 bind 方法就是数据被绑定到 ASPX 文件中控件所在的地方。在该方法中会看到 ListBox1.DataSource = data 等代码，但现在说这些还有点为时过早。首先，需要确保从控制器中的 GetData 函数调用了 bind 方法。为此，创建该视图的另一个模拟，然后验证调用了 Bind 方法。在实现所有这一逻辑之后，它应当能够通过。

```

public class and_when_calling_getdata_bind_should_be_called :
    and_getting_a_list_of_books
{
    IDisplayBooksReadView _view;
    DisplayBooksReadController _controller;

    protected override void Establish_context()
    {
        base.Establish_context();
        _view = MockRepository.GenerateMock < IDisplayBooksReadView > ();
        _view.Expect(v => v.Bind());
        _controller = new DisplayBooksReadController(_view);
    }

    protected override void Because_of()
    {
        _controller.GetData(this, EventArgs.Empty);
    }

    [Test]
    public void then_getdata_should_call_bind()
    {
        _view.VerifyAllExpectations();
    }
}

```

DisplayBookReadControllerTests.cs

现在已经实现了为在 ASP.NET Web 窗体中显示最近所读书籍需要的全部“管道”代码。在介绍用于连接所有这些内容的黏合代码之前，先简要地介绍一下帮助器的概念。

帮助器

帮助器方法帮助其他方法执行任务。通常，它们是在应用程序其他方法之间共享的一些通用任务。例如，在 Web 中，通用的一种做法是将文本截断为指定长度，然后在其末尾加上一个省略号，表明该文本已被截断。这种方法并不是 Read Book 视图所特有的；在该应用程序中的任何其他视图中都可能用到它，所以可以创建一个包含截断方法的 HTMLHelper 类，以帮助我们以这种方法来显示文本。如果前面没有说过下面这句话，那现在请听我说：TDD 是与“重复”相关的。首先为 Truncate 方法创建一个测试：



```
public class when_using_the_html_helper_to_truncate_text : Specification
{
    protected string _textToTruncate;
    protected string _expected;
    protected string _actual;
    protected int _numberToTruncate;
    protected override void Because_of()
    {
        _actual = HTMLHelper.Truncate(_textToTruncate, _numberToTruncate);
    }
}

public class and_passing_a_string_that_needs_to_be_truncated :
    when_using_the_html_helper_to_truncate_text
{
    protected override void Establish_context()
    {
        _textToTruncate = "This is my text";
        _expected = "This ...";
        _numberToTruncate = 5;
    }

    [Test]
    public void then_the_text_should_be_truncated_with_ellipses()
    {
        _expected.ShouldEqual(_actual);
    }
}
```

HTMLHelpersTests.cs

接下来，实现足以使这一测试通过的代码：



```
public static string Truncate(string input, int length)
{
    if (input.Length <= length)
        return input;
    else
        return input.Substring(0, length) + "...";
}
```

HTMLHelpers.cs

现在再回过头来创建更多的测试：



```
public class and_passing_a_string_that_does_not_need_to_be_truncated :
    when_using_the_html_helper_to_truncate_text
{
    protected override void Establish_context()
    {
        _textToTruncate = "This is my text";
        _expected = "This is my text";
        _numberToTruncate = 0;
    }

    [Test]
    public void then_the_text_should_not_be_truncated()
    {
        _expected.ShouldEqual(_actual);
    }
}

public class and_passing_a_string_that_is_less_than_what_needs_to_be_truncated :
    when_using_the_html_helper_to_truncate_text
{
    protected override void Establish_context()
    {
        _textToTruncate = "This is my text";
        _expected = "This is my text";
        _numberToTruncate = 50;
    }

    [Test]
    public void then_the_text_should_not_contain_ellipses()
    {
        _expected.ShouldEqual(_actual);
    }
}

public class and_pass_a_null_value : when_using_the_html_helper_to_truncate_text
{
    protected override void Establish_context()
    {
        _textToTruncate = null;
        _expected = string.Empty;
        _numberToTruncate = 50;
    }

    [Test]
    public void then_the_text_should_return_empty_string()
    {
        _expected.ShouldEqual(_actual);
    }
}
```

HTMLHelpersTests.cs

然后对这一实现方式进行重构，使这些测试得以通过：



```
public static string Truncate(string input, int length)
{
    if (length == 0)
        return input;
    if (String.IsNullOrEmpty(input))
        return string.Empty;
    if (input.Length <= length)
        return input;
    else
        return input.Substring(0, length) + "...";
}
```

HTMLHelpers.cs

将所有内容黏合在一起

现在，我们已经创建了显示 Read Books 所需要的全部逻辑。剩下的就是把这个逻辑黏合到 Web 窗体中。下面的代码是 Read Books Web 窗体应用程序的默认页面。要注意的第一件事情就是 Default 页面需要实现 IDisplayBooksReadView。添加模块级别的变量，以用来持久保存控制器和数据。还应当为 DataRequested 定义一个事件。在 Page_Load 函数中，只需要创建一个新的控制器，然后触发 DataRequested 事件。

触发 DataRequested 事件的结果就是，Bind 函数被调用，数据被绑定到屏幕上。只要理解了概念，下面的代码就非常容易理解了：

```
public partial class _Default : System.Web.UI.Page, IDisplayBooksReadView
{
    private DisplayBooksReadController Controller;
    public event EventHandler DataRequested;
    public List < BookRead > Data { get; set; }

    protected void Page_Load(object sender, EventArgs e)
    {
        Controller = new DisplayBooksReadController(this);
        DataRequested(sender, e);
    }

    public void Bind()
    {
        rptBooksRead.DataSource = Data;
        rptBooksRead.DataBind();
    }
}
```

Default.aspx.cs

作为开发人员,您可能没有考虑当设计人员在收到你的代码时,必须仔细检查些什么。就像代码隐藏文件中的所有内容一样,目标应当是使标记尽可能保持清洁。下面的示例使用了一个重复器。你希望尽力避免使用 DataGrid,因为它们会添加一些非常“丑陋”的代码,这些代码不仅难以测试,而且设计人员处理起来也比较困难。使<%标记的数量保持最少;只在需要显示一个字段时才实现它们。利用这些策略,可以创建一些能够让大多数设计人员或开发人员都能轻松理解的标记。



```
< asp:Repeater ID="rptBooksRead" runat="server" >
  < ItemTemplate >
    < div class="readBook" >
      < div class="bookInfo" >
        < h2 > < %#Wrox.BooksRead.Web.Helpers.HTMLHelper
          .Truncate(((Wrox.BooksRead.Web.BookRead)
            Container.DataItem).Name, 25) % >
        < /h2 >
        < p > < %# ((Wrox.BooksRead.Web.BookRead)
          Container.DataItem)
            .Author % > < /p >
        < p > ISBN: < %# ((Wrox.BooksRead.Web.BookRead)
          Container.DataItem).ISBN
          % > < /p >
        < p > Date Finished: < %# ((Wrox.BooksRead.Web.BookRead)
          Container
            .DataItem)
              .EndDate.ToString("MM/dd/yyyy")% >
        < /p >
        < a href=' < %#
          ((Wrox.BooksRead.Web.BookRead)Container.DataItem)
            .PurchaseLink % > 'target="_blank" > Purchase < /a >
      < /div >
      < div class="ratingContainer" >
        < span class="rating" > < %#((Wrox.BooksRead.Web.BookRead)
          Container.DataItem)
            .Rating % >
        < /span >
      < /div >
      < div style="clear:both;" / >
      < hr / >
    < /div >
  < /ItemTemplate >
< /asp:Repeater >
```

Wrox.BooksRead.Web.Default.aspx

从图 9-6 可以看出，一共显示了 3 本已经读过的书籍。

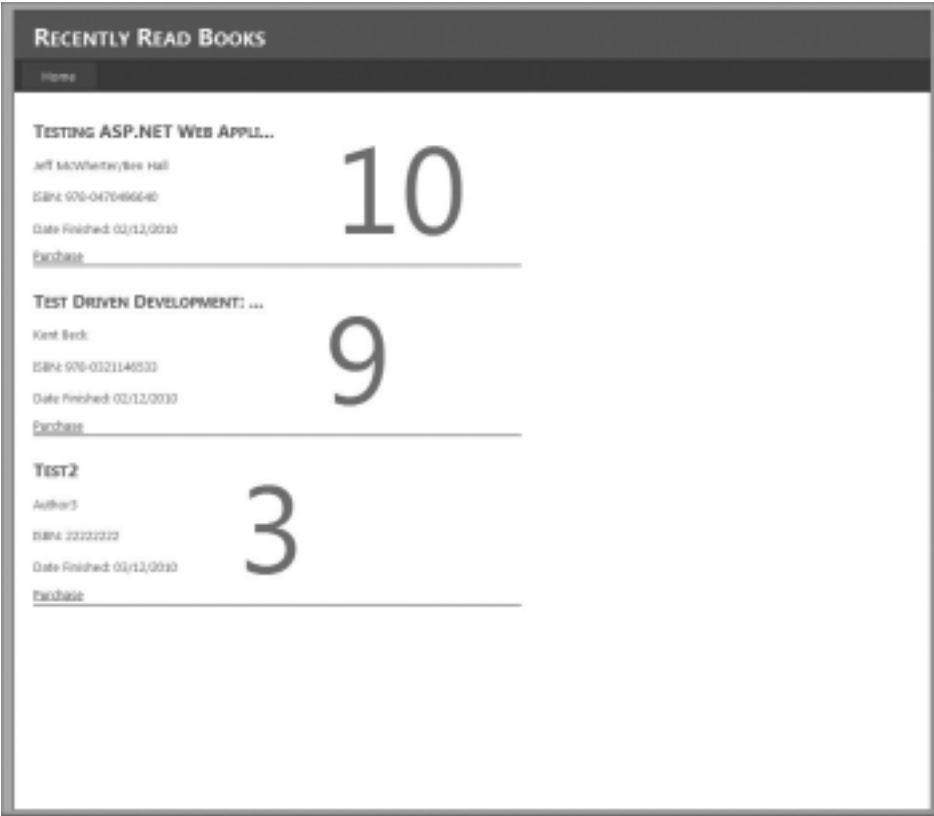


图 9-6

在该 MVP 示例中，展示了如何隔离对象和创建可测试的 Web 窗体。MVP 模式从来没有被用作开发 ASP.NET Web 窗体的主流方法，但如果不得不继续使用 ASP.NET Web 窗体开发，并且不能选择 ASP.NET MVC 等新框架，那以这种模式作为开始还是一种不错的选择。

这是一个简单的示例，但在 ASP.NET 框架的更低层次上工作并且需要启动模拟对象时(如 HTTPResponse 对象)，测试 MVP 模式就会变得非常痛苦。微软开发人员近来需要提交一种具有高度可测性的框架，于是就出现了 ASP.NET MVC 框架。

9.2 使用 ASP.NET MVC

1979 年，挪威计算机科学家 Trygve Reenskaug 在 Xerox PARC 工作时发表了一篇文章，题为“*Applications Programming in Smalltalk - 80: How to Use Model - View Controllers*”。就是在这篇论文中，Reenskaug 正式阐述了这一软件设计模式。尽管 MVC 设计模式的出现已经有 30 多年了，但它直到最近才因为 Ruby on Rails 的成功变得流行起来。Ruby on Rails 向开发人员表明可以创建出高度可测试的 Web 应用程序。使用 Ruby on Rails 的开发人员

总是夸耀其代码的可测试性是多么多么好。其他 Web 平台也实现 MVC 框架只是时间问题。在 2011 年，几乎所有编程语言，从 Cold Fusion 到 PHP，都拥有了 MVC 框架。其代码看似不同，但每个框架都是以 Reenskaug 的工作为基础。MVC 并非只能用于 Web。许多 GUI 框架，包括 Cocoa(Mac OS X 的编程环境)，也从 Reenskaug 的 Small Talk 框架中获得了灵感。在 Cocoa 中，鼓励开发人员使用这一模式。

2009 年 3 月，微软向公众发布了 ASP.NET MVC Framework 1.0。根据微软公共许可，其代码是开源的，所以如果你想知道该框架在后台做了些什么，可以去看一看。ASP.NET MVC Framework 团队是敏捷的，总是尽早、频繁发布代码，以从社区获取反馈。如果能够看到提交给这个团队的反馈得以实施，那是很棒的一件事情。微软明确表明：ASP.NET MVC 框架无意于替代 ASP.NET Web 窗体。它的创建是为了给开发人员多提供一种选择。

在编写本书时，我们数了数市面上的 ASP.NET MVC 书籍，共有 16 本，而以某种方式提到 ASP.NET MVC 的其他书籍则有无数本。我们并不是想让你成为 MVC 专家，但我们的确希望让你很好地掌握如何利用已经学习的 TDD 技术来创建 MVC Web 应用程序。

9.2.1 MVC 101

Reenskaug 声明，“他创建的‘模式-视图-控制器’模式，是为了很好地完成一项常见任务——使用户能够从多个角度控制自己看到的信息”。图 9-7 所示为 MVC 设计模式。



图 9-7

对于本章，将 MVC 定义如下：

- **模型**——模型管理应用程序中的数据。存储和操作数据库的类与业务逻辑合并在一起。
- **视图**——视图将模型呈现在一个界面中，用户可以与该界面交互。HTML、JSON 和 XML 都是常见的视图形式。
- **控制器**——控制器协调模型与视图之间的通信。控制器从视图接收输入，并从模型启动响应。控制器根据所请求的视图决定呈现哪个视图——HTML、XML 或 JSON。验证逻辑也存在于控制器中。

9.2.2 Microsoft ASP.NET MVC 3.0

前面各章使用 WCF 等不同技术实现了 OSIM 示例。Visual Studio 2010 带有 ASP.NET MVC 2.0，但该示例使用了 ASP.NET MVC 3.0，可以从 <http://www.asp.net/mvc/mvc3> 下载。

MVC 框架依赖于一些配置约定，只要遵循这些约定就会“诸事顺利”。路由是 ASP.NET MVC 的一个重要方面。简单来说，MVC 框架中的路由指出了应当如何处理 HTTP 请求。

过去，使用 ASP.NET Web 窗体时，几乎无法控制请求如何工作。请求与处理该请求的页面紧密耦合在一起。下面的代码是在创建 ASP.NET MVC 应用程序时添加到 Global.asax 文件中的默认代码：



```
routes.MapRoute("Default", // Route name
    "{controller}/{action}/{id}", // URL with parameters
    new { controller = "Home", action = "Index", id =
        UrlParameter.Optional });
```

Global.asax.cs

注意到这一点是非常重要的，因为许多新接触 ASP.NET MVC 的开发人员会忽略这一事实。大体来说，这段代码指出：假定在控制器类中有一个名为 home 的索引方法，URL 类似于 http:// url / controller / action / id 或 http:// localhost/ home / index。

1. 创建 ASP.NET MVC 项目

创建 ASP.NET MVC 项目就像选择 MVC 3 Web 应用程序一样简单，在创建新项目时，可以在 Web 项目类型下面找到它，如图 9-8 所示。

在选择了 MVC 3.0 项目类型之后，可以从两种默认模板中选择——Empty(空模板)或 Internet Application (Internet 应用程序)。在选择了 Internet Application 模板之后，将会创建一个包含默认 HTML/CSS 样式的新项目。该项目还包含一个首页的默认视图和控制器，以及用于验证的基本内容。为了跟随该示例的步骤，选择 Empty 模板，如图 9-9 所示。

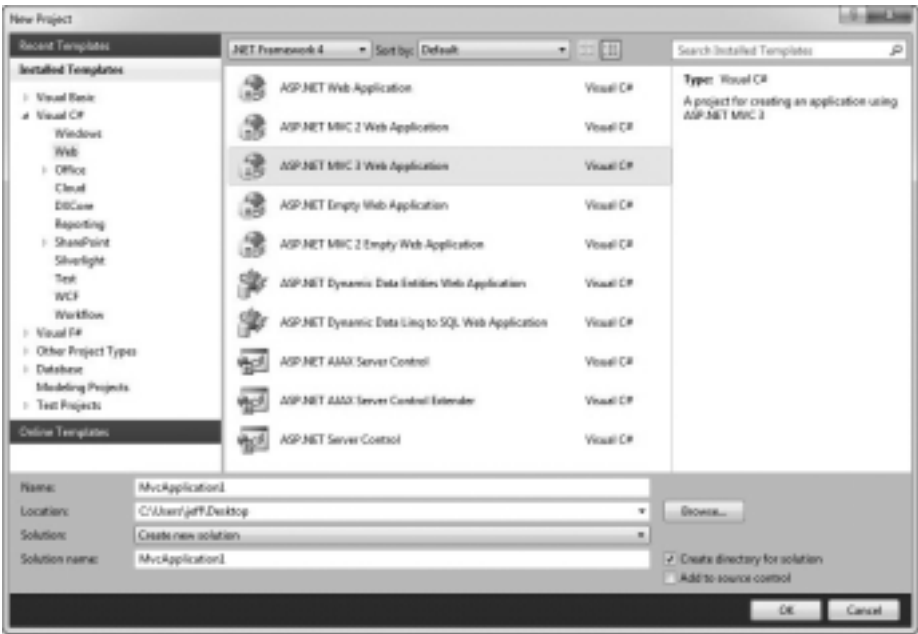


图 9-8

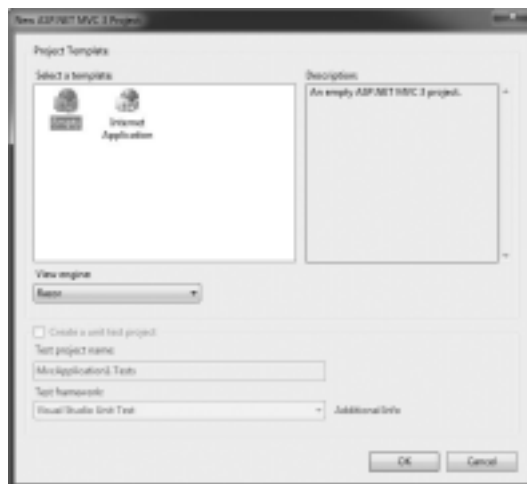


图 9-9

图 9-10 给出了该框架创建的默认结构。它包含以下文件夹：

- Content 包含 HTML、CSS 和图片文件。
- Controllers 包含控制器类。
- Models 包含模型类。
- Scripts 包含应用程序需要的 JavaScript。
- Views 包含视图。每个控制器在该目录下都有一个子目录，其中包含了用于 CRUD 操作的视图。



图 9-10

2. 创建第一个测试

根据前面的示例，该示例实现了 OSIM 项目中使用的相同功能，再次涉及物品类型概念。我们列出、创建和编辑物品类型。

在许多情况下，开发人员会启动一个已经存在数据模型的项目。该项目就是其中一个，因为前面各章已经深入讨论了对数据层的测试。我们知道，创建 OSIM.Core 模块(该模块中包含数据层)的开发人员已经拥有相关的全套测试，所以不需要再为该程序集创建测试。因为将使用物品类型，所以首先来为控制器创建一个测试，以确保该控制器能够获得要显示的物品类型数据。

首先对测试进行设置。需要做的第一件事情是设置物品类型库对象。还将设置一个模拟(mock)，以在调用 GetAll 方法时返回一个列表，其中包含这个库的 3 个物品类型：

```
public class when_working_with_the_item_type_controller : Specification
{
    protected Mock < IItemTypeRepository > _itemRepository
        = new Mock < IItemTypeRepository > ();
    protected ItemType _itemOne;
    protected ItemType _itemTwo;
```



```

protected ItemType _itemThree;

protected override void Establish_context()
{
    _itemOne = new ItemType { Id = 1, Name = "USB drives" };
    _itemTwo = new ItemType { Id = 2, Name = "Nerf darts" };
    _itemThree = new ItemType { Id = 3, Name = "Flying Monkeys" };
    var itemTypeList = new List < ItemType >
    {
        _itemOne,
        _itemTwo,
        _itemThree
    };
    _itemRepository.Setup(x => x.GetAll)
    .Returns(itemTypeList);
}
}

```

ItemTypeRepositoryTests.cs

在为物品类型库准备好设置逻辑之后，就可以创建第一个测试了。将要创建的第一个页面列出物品类型。我们希望进行测试，以确保控制器能够正确地处理库。下面的测试确保了在创建控制器时，能够正确地从库中设置模型：

```

public class and_trying_to_load_the_index_page :
    when_working_with_the_item_type_controller
{
    Object _model;
    int _expectedNumberOfItemsInModel;

    protected override void Establish_context()
    {
        base.Establish_context();
        _expectedNumberOfItemsInModel = _itemRepository.Object.GetAll.Count;
    }

    protected override void Because_of()
    {
        _model = ((ViewResult)new ItemTypeController
            (_itemRepository.Object).Index()).ViewData.Model;
    }

    [Test]
    public void then_a_valid_list_of_items_should_be_returned_in_the_model()
    {
        _expectedNumberOfItemsInModel.ShouldEqual
            (((List < ItemType > )_model).Count);
        _itemOne.ShouldEqual(((List < ItemType > )_model)[0]);
    }
}

```

ItemTypeControllerTests.cs

3. 使第一个测试通过

现在已经有了测试，可以创建自己的控制器了，将其命名为 `ItemTypeController`。右击 `Controllers` 文件夹，并选择 `Add | Controller` 命令，打开一个对话框。在该对话框中可以创建控制器，如图 9-11 所示。

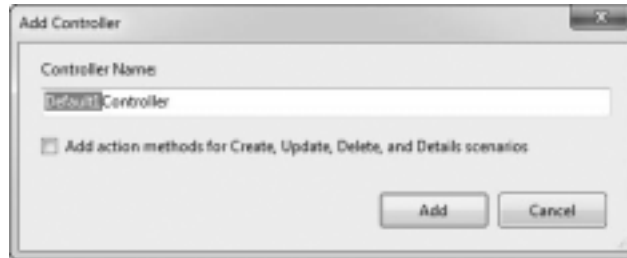


图 9-11

可以选择让 Visual Studio 来创建所有操作方法，用于创建和编辑等，但由于还没有为它们准备测试，因此希望在以后自己处理它们。现在，只实现 `Index` 方法。创建了 `Controller` 类之后，用以下代码填充空白区域：

```
public class ItemTypeController : Controller
{
    IItemTypeRepository _itemTypeRepository;

    public ItemTypeController(IItemTypeRepository itemRepository)
    {
        _itemTypeRepository = itemRepository;
    }

    public ActionResult Index()
    {
        ViewData.Model = _itemTypeRepository.GetAll();
        return View();
    }
}
```

ItemTypeController.cs

`ItemTypeController` 代码中最值得注意的是添加了一个构造函数，它接收 `Item` 库。这就是获取库实例的方式。有了这个逻辑，测试就可以通过，但网站还不能像预期的那样发挥功能。

4. 创建第一个视图

我们需要创建一个视图，用来显示物品类型。我们希望在 Views 目录下创建一个名为 ItemType 的子目录，使一切保持井然有序。在创建子目录之后，右击新创建的 ItemType 目录，并选择 Add | View 命令，会打开类似于图 9-12 所示的对话框。

在用 ASP.NET MVC 3.0 创建视图时，关于视图引擎有两个默认选项。深入讨论视图引擎超出了本书范围，现在只要知道视图引擎是呈现页面的标记即可。每种视图引擎呈现标记的方式稍有不同，而且都有自己的优点。默认情况下，Microsoft ASP.NET MVC 3.0 装配有 Razor 和 ASPX 视图引擎。也可以使用其他一些开源视图引擎，如 Spark 和 NHaml。本例使用了 Razor 视图引擎。

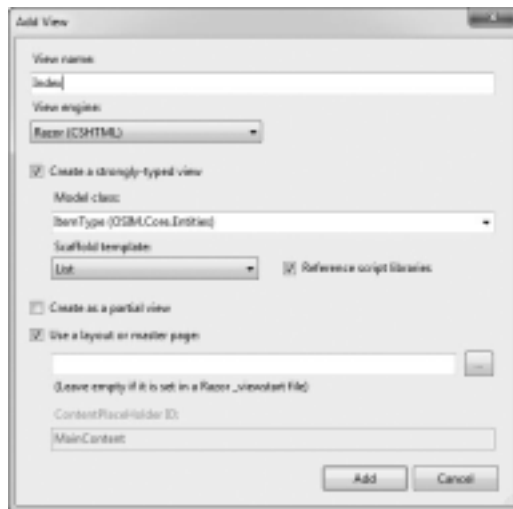


图 9-12

选中 Create a strongly-typed view 复选框，并从 OSIM.Core.Entities 名称空间中选择 ItemType 类，使 ASP.NET Framework 为此视图生成一个支架(scaffold)。支架就是一个包含标记的模板，用于在视图中呈现模型。支架使你快速由点 A 到达点 C。对于支架模板有许多不同选项。在本例中，正在屏幕上列举物品类型，所以应当选择 List 模板。下面是 Index 视图的呈现代码：



可从
wrox.com
下载源代码

```
@model IEnumerable< OSIM.Core.Entities.ItemType >

@{
    ViewBag.Title = "ItemType";
}

< h2 > ItemType < /h2 >

< p > @Html.ActionLink("Create New", "Create") < /p >
< table >
    < tr >
        < th > < /th >
        < th > Name < /th >
    < /tr >
    @foreach (var item in Model) {
        < tr >
            < td > @Html.ActionLink("Edit", "Edit", new { id=item.Id }) | < /td >
            < td > @item.Name < /td >
        < /tr >
    }
< /table >
```

Index.cshtml

为 ItemType 目录索引中的视图命名非常重要。这是因为 MVC 是以约定为基础的，当控制器(Index 方法)中为默认设置时，MVC 框架将查看具有这一名字的文件。图 9-13 给出了当框架不能找到正确文件时的情况。



图 9-13

这并不是说就不能创建自定义路由。尽管可以随心所欲地为视图命名，但通常不鼓励这种做法。

5. 把所有内容黏合在一起

有了 Index 视图之后，距离网站应用程序的建成和运行只有一步之遥了。还需要添加一点黏合代码，用来处理依赖项注入。和前面各章的示例一样，该例说明如何使用 Ninject 向控制器中注入依赖项。还记得向物品类型控制器中添加的用来接收物品类型库的构造函数吗？该逻辑将存储库的依赖项注入到控制器中。下面是 Global.asax.cs 文件中的代码。



```
public class MvcApplication : NinjectHttpApplication
{
    protected override void OnApplicationStarted()
    {
        base.OnApplicationStarted();

        AreaRegistration.RegisterAllAreas();
        RegisterGlobalFilters(GlobalFilters.Filters);
        RegisterRoutes(RouteTable.Routes);
    }

    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }
}
```

```

public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "Default" // Route name
        "{controller}/{action}/{id}", // URL with parameters new
        { controller = "ItemType", action = "Index", id =
        UrlParameter.Optional });
}

protected override IKernel CreateKernel()
{
    return new StandardKernel(new PersistenceModule(), new
    CoreServicesModule());
}
}

```

Global.asax.cs

该示例中最值得注意的是,它不是从 System.Web.HttpApplication 继承而来的 Global.asax,而是继承自 NinjectHttpApplication,这就需要引用 Ninject.Web.Mvc 程序集。为了完成 Web 应用程序并使其运行,所需的全部黏合代码就是继承 NinjectHttpApplication,并实现 CreateKernel

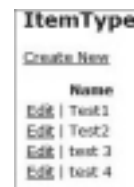


图 9-14

方法,以映射依赖项。可能还注意到,在 RegisterRoutes 方法中,该示例是将默认控制器由 Home 改为 ItemType,使它默认指向所创建的唯一控制器。如果没有进行这一修改,那 Web 应用程序将会查找一个不存在的首页控制器。运行该 Web 应用程序,将会呈现一个类似于图 9-14 所示的网站。

现成已经实现了第一项功能,可以返回来创建一些与物品类型的创建和编辑相关联的测试了。我将放弃为这些功能创建控制器和视图,只给出以下测试:

```

public class and_trying_to_create_a_new_valid_item_type :
    when_working_with_the_item_type_controller
{
    ItemType _newItemType;
    ItemTypeController _controller;
    RedirectToRouteResult _result;
    string _expectedRouteName;

    protected override void Establish_context()
    {
        base.Establish_context();
        _expectedRouteName = "Index";
        _newItemType = new ItemType() { Id = 99, Name = "New Item" };
        _controller = new ItemTypeController(_itemRepository.Object);
    }
}

```

```

        protected override void Because_of()
        {
            _result = _controller.Create(_newItemType) as RedirectToRouteResult;
        }

        [Test]
        public void then_a_new_item_type_should_be_created_and_
            the_redirected_to_the_correct_view()
        {
            _result.ShouldNotBeNull();
            _result.RouteValues.Values.ShouldContain(_expectedRouteName);
        }
    }

    public class and_trying_to_create_a_new_invalid_item_type :
        when_working_with_the_item_type_controller
    {
        ItemType _newItemType;
        ItemTypeController _controller;
        ViewResult _result;
        string _expectedRouteName;

        protected override void Establish_context()
        {
            base.Establish_context();
            _expectedRouteName = "create";
            _newItemType = new ItemType() { Id = 99, Name = "New Item" };
            _controller = new ItemTypeController(_itemRepository.Object);
            _controller.ModelState.AddModelError("key", "model is invalid");
        }

        protected override void Because_of()
        {
            _result = _controller.Create(_newItemType) as ViewResult;
        }

        [Test]
        public void then_a_new_item_type_should_not_be_created()
        {
            _result.ShouldNotBeNull();
            _result.ViewName.ShouldEqual(_expectedRouteName);
        }
    }

    public class and_trying_to_edit_an_existing_item :
        when_working_with_the_item_type_controller
    {
        string _expectedRouteName;
        ItemTypeController _controller;
        ViewResult _result;

        protected override void Establish_context()
        {

```

```
        base.Establish_context();
        _expectedRouteName = "edit";
        _controller = new ItemTypeController(_itemRepository.Object);
        _result = _controller.Edit(_itemOne.Id) as ViewResult;
    }

    protected override void Because_of()
    {
    }

    [Test]
    public void then_a_valid_edit_view_should_be_returned()
    {
        _expectedRouteName.ShouldEqual(_result.ViewName);
    }
}
```

ItemTypeControllerTests.cs

9.2.3 使用 MVC Contrib 项目

开源 MVC Contrib 项目可以在 <http://mvcontrib.codeplex.com/> 下载。该项目向 ASP.NET MVC 框架中添加了许多功能，对于 MVC 开发人员来说非常有用。MVC Contrib 项目包含了一些可以用来帮助完成单元测试的功能。MVC Contrib 项目填充了 ASP.NET MVC 框架中的一些缝隙。对于任何一个希望开发 ASP.NET MVC 应用程序的人员来说，MVC Contrib 项目都是必须的。

9.2.4 ASP.NET MVC 汇总

在微软堆栈上开始 Web 应用程序时，ASP.NET MVC 需要一种不同的思考方式。在刚开始时，由于即使开发最简单的 Web 应用程序也需要掌握大量新的工具和概念，所以可能会感到有些难以应付。但在学习了基础知识之后，就会意识到：使用 ASP.NET MVC 框架会迫使你和你同事编写出易于阅读、测试和维护的清洁代码。

9.3 使用 JavaScript

许多 Web 开发人员对 JavaScript 都是爱恨交加。在大多数情况下，Web 开发人员不会从本质上憎恨 JavaScript；他们恨的是大多数 Web 浏览器中的文档对象模型(Document Object Model, DOM)。近年来，jQuery 和 Prototype 等 JavaScript 框架已经减轻了 DOM 为 Web 开发人员带来的痛楚。利用这些框架可以更轻松地使用 DOM，并减少测试 JavaScript 所需要创建的测试数目。记住，不需要对该框架进行测试。

谈到 JavaScript，Web 开发人员最常犯的错误之一就是没有将它们的逻辑抽象到不同

文件中。许多开发人员将所有内容都放在一个脚本文件中，或者重复许多不同 Web 页的功能。他们没有意识到存在 JavaScript 的测试框架，或者没有意识到可以使用 TDD 方法创建 JavaScript。

JavaScript 是一种真正的语言，其代码也应当得到同样对待。前面介绍的所有 SOLID 相关规则同样适用于 JavaScript。当采用这种方式创建 JavaScript 代码时，可以很轻松地对其进行测试和维护。一些 Web 开发人员抱怨一个需要下载多个 JavaScript 文件的 HTML 页面加载过程非常缓慢。在创建过程中使用合并和缩小等技巧可以解决这些缓慢的页面加载问题。测试性能是另一个主题；现在，将重点讨论创建可测试代码。

JavaScript 测试框架

和大多数语言一样，有很多框架可以用来测试 JavaScript 代码。qUnit、Screw Unit 和 jsUnit 等框架都是开源测试框架，它们与 NUnit 非常类似，但是为测试 JavaScript 设计的。这里的示例使用了 qUnit——Query 团队已经选择这种测试框架来测试它非常流行的 JavaScript 框架。qUnit 的出现已经有一段时间了，由于 jQuery 项目的成功，它拥有一大批追随者。当开源项目拥有大批追随者时，这就意味着可以找到文档和文件错误，并可以经常获得软件更新。

在这些结合 JavaScript 使用 TDD 的示例中，有些内容与本章前面所介绍的所有应用程序中测试的 JavaScript 相比，要稍为复杂一些。在这些应用程序中添加的大多数 JavaScript 可以看成黏合代码，在集成测试或功能测试阶段会对其进行测试。

下面的示例说明如何创建一个表示空账户的 JavaScript。该对象保存当前的余额，还有一个函数用于在两个账户之间转移资金。第一步是设置 qUnit 框架，这一过程非常简单。图 9-15 显示了为该应用程序设置的目录。它包括了 jQuery、qUnit JavaScript 和 qUnit 样式表。在 js 文件夹中创建自己的测试。

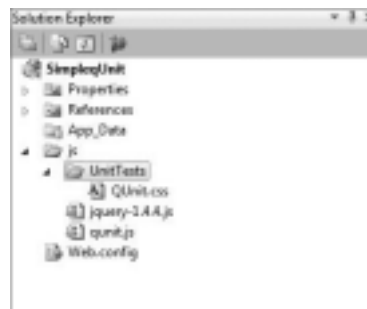


图 9-15

该框架准备就绪之后，就可以编写第一个测试了。qUnit 测试是用 JavaScript 编写，运行程序包含在 HTML 中。就像在 C# 或 Visual Basic 中进行测试一样，应当为正在测试的每个对象创建独立文件。在该例中，正在测试账户对象，所以创建 AccountTests.htm，它将包含针对所创建账户对象的所有测试。

和其他测试框架一样，希望编写测试，描述测试希望完成什么。在该示例中，只是希望创建一个账户对象，它有一个账户号和一个余额：

```
< script type="text/javascript" >
test('Should_Create_An_Account_Object_With_Balance', function() {
    var checkingAccount = new Account(564858510, 600);
    equals(checkingAccount.Balance, 600);
});
< /script >
```


要实现该测试，需要添加测试运行程序的实现代码。以下代码给出了将要运行的测试的完整实现。



```
< html xmlns="http://www.w3.org/1999/xhtml" >
  < head >
    < title > Account Tests < /title >
    < link rel="stylesheet" href="QUnit.css" type=
      "text/css" media=" screen" / >
    < script type="text/javascript" src="../../jquery-1.4.4.js" >
    < /script >
    < script type="text/javascript" src="../../qunit.js" > < /script >
    < script type="text/javascript" src="../../Account.js" > < /script >

    < script type="text/javascript" >
      test('Should_Create_An_Account_Object_With_Balance',
        function() {

          var checkingAccount = new Account(564858510, 600);
          equals(checkingAccount.Balance, 600);

        });

    < /script >
  < /head >
  < body >
    < h1 id="qunit-header" > Account Unit Tests < /h1 >
    < h2 id="qunit-banner" > < /h2 >
    < div id="qunit-testrunner-toolbar" > < /div >
    < h2 id="qunit-userAgent" > < /h2 >
    < ol id="qunit-tests" > < /ol >
    < div id="qunit-fixture" > test markup < /div >
  < /body >
< /html >
```

AccountTests.htm

在测试和运行逻辑准备就绪之后，就可以运行该测试了。应当可以预料到该测试会失败，因为还没有实现该账户对象。图 9-16 显示了这一情况。



图 9-16

在测试失败之后，就可以实现账户 JavaScript 了：



```
function Account(accountNumber, balance)
{
  this.AccountNumber = accountNumber;
  this.Balance = balance;
}
```

Account.js

如图 9-17 所示，该测试现在通过了。



图 9-17

现在来满足账户对象在对象之间转移资金的需求。在一个名为 `Should_Transfer_From_Checking_To_Savings_Successfully` 的测试中，创建两个带有余额的账户，然后从其中一个账户向另一个账户转移资金。在运行该测试时，预计它会失败，如图 9-18 所示。

可从
wrox.com
下载源代码

```
test('Should_Transfer_From_Checking_To_Savings_Successfully',
function() {
    var checkingAccount = new Account(564858510, 600);
    var savingsAccount = new Account(564858507, 100);
    checkingAccount.Transfer(savingsAccount, 100);
    equals(savingsAccount.Balance, 200);
    equals(checkingAccount.Balance, 500);
});
```

AccountTests.htm



图 9-18

测试失败之后，可以实现传递资金所需要的代码。所有的测试都得以通过，如图 9-19 所示。

可从
wrox.com
下载源代码

```
function Account(accountNumber, balance)
{
    this.AccountNumber = accountNumber;
    this.Balance = balance;
}

Account.prototype.Transfer = function(toAccount, amount)
{
    toAccount.Balance += amount;
    this.Balance = this.Balance - amount;
}
```

Account.js



图 9-19

我们只是实现了为使 JavaScript 账户对象得以通过所需要的最少内容，但还没有完工。还需要测试业务规则和边界条件。首先测试当一个账户尝试转移的数目大于本身余额时会发生什么。编写测试来看一下：



```
test('Should Not Transfer From When The From Account Has
_Inufficient_Funds',function () {
    var checkingAccount = new Account(564858510, 600);
    var savingsAccount = new Account(564858507, 100);

    checkingAccount.Transfer(savingsAccount, 700);
    equals(savingsAccount.Balance, 100);
    equals(checkingAccount.Balance, 600);
});
```

Account.js

如图 9-20 所示，该测试将会失败。



图 9-20

所编写的测试预料，当尝试从一个仅有 600 美元的账户中转移 700 美元时，这些账户的余额将保持不变，即没有发生转移。如果仔细看一下如图 9-20 所示的错误，可以看到这次转移的确发生了。该应用程序有一处错误，应当在别人注意到之前修改它。只需添加一个安全语句，查看一下所转移的数目是否大于当前余额，就可以快速解决这一问题。如果大于该数目，就不执行转移，直接返回：



```
Account.prototype.Transfer = function (toAccount, amount) {
    if (this.Balance < amount) {
        return;
    }

    toAccount.Balance += amount;
```

```

    this.Balance = this.Balance - amount;
}

```

Account.js

图 9-21 显示所有测试现在都已通过。

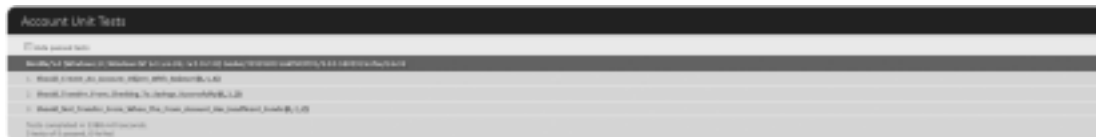


图 9-21

现在应当开始感受到 TDD 的重复了,这一点是非常重要的。在现实情况下,应当拥有覆盖所有情景的测试。前面各章讨论了到底要测试哪些内容,但对于我们来说,这个例子就够了。

在为整个对象创建了测试之后,就可以在生产代码中实现新开发的对象了。下面的代码是这个 JavaScript 账户对象的实现。已经为该账户对象编写了一组测试,这里给出的代码可以看作黏合代码或 jQuery 框架代码,不需要在这一级别对其进行测试:



```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title> Simple qUnit Test </title>
  <script type="text/javascript" src="js/account.js"> </script>
  <script type="text/javascript" src="js/jquery-1.4.4.js"> </script>
</head>
<body>
  <div>
    Savings Balance: <span id="SavingsBalance"> </span>
  </div>
  <div>
    Checking Balance: <span id="CheckingBalance"> </span>
  </div>
</body>

<script type="text/javascript">
  var checkingAccount = new Account(564858510, 600);
  var savingsAccount = new Account(564858507, 100);

  savingsAccount.Transfer(checkingAccount, 100);

  jQuery("document").ready(function () {
    jQuery("#SavingsBalance").text(savingsAccount.Balance);
    jQuery("#CheckingBalance").text(checkingAccount.Balance);
  });
</script>
</html>

```

Default.aspx

图 9-22 所示为该项目的最终组织结构。



图 9-22

9.4 本章小结

Web 上的测试驱动开发有时可能非常麻烦。本章介绍了一些框架和概念，可以用来开始创建一些高度可测试的 Web 应用程序。在你工作过的一些环境中，开发团队可能不希望尝试新的概念，但本章介绍的许多概念可以利用团队已经掌握的现有工具来慢慢向团队灌输。

对于许多微软 Web 开发人员来说，ASP.NET MVC 是一种新的思考方式。微软已经声明，ASP.NET Web 框架不会在短时间内消失。所以 MVC 框架是可以选择使用的一种额外工具。

和前面讨论的所有 TDD 实践一样，学习这些的最佳方法是走出去，与那每天使用这些实践的其他开发人员会谈。一些免费会议如 Day of .NET 和 Code Camps，都是与人会谈并提高自己掌握新技能的好方法。

第 10 章

测试 WCF 服务

本章内容

- 了解 WCF(Windows Communication Foundation , Windows 通信功能)服务为什么是应用程序的重要组成部分。
- 将 WCF 服务看作接口代码，类似于 Windows 窗体或网页
- 重构 WCF 服务，以采用依赖项注入
- 理解传输在测试 WCF 服务时所扮演的角色

与网页或桌面窗口类似，服务是通向应用程序的接口。和应用程序的其他接口一样，也需要对应用程序的服务进行测试，以确保它们满足作为应用程序接口所应当具有的所有条件。对服务进行测试尤其重要，因为它们与 Windows 窗体或网页不同，它的错误和缺陷很容易被服务层模糊或淹没，从而使缺陷诊断变得极为困难。

WCF 是用 .NET 创建服务的标准框架。但是，即使是 WCF 的一些基础知识，也让开发人员感到害怕。结果，即使是那些希望学习 WCF 的开发人员也发现服务测试是一件令人退缩的提议。学习正确测试 WCF 服务的技巧并不难。本章提供了测试 WCF 服务所需要的工具和知识。



提示：

本章假定你拥有一些基本的 WCF 知识和技巧。如果不熟悉 WCF，有许多相关主题的书可供阅读。我的博客中提供了一个正在进行的 WCF 培训系列，它非常适合从来没有用过 WCF 的开发人员，博客地址为 www.jamescbender.com。

10.1 应用程序中的 WCF 服务

服务对于应用程序非常重要。作为一种体系结构工具，服务允许我们开发出一些应用广泛、适应性很强的系统，仅使用静态绑定或局限于单个实体机器的应用程序是不可能开发出这种系统的。服务增强了可扩展性，提高了灵活性。有人可能会宣称：服务是在寻求松散耦合系统中的最新形式。移动设备的日益流行、希望通过 JavaScript 和 AJAX 调用来开发出能够快速响应的 Web 应用程序、Silverlight 等框架、REST 的复活，都为体系结构概念带来了新的生命。直到几年之前，人们还认为只有有关在象牙塔中的体系结构设计师和面向服务的体系结构(SOA)狂们才会关注这一概念。由于这些原因，越来越多的应用程序开始提出某种类型的服务接口需求。

服务也是代码

服务中的实际代码应当非常简单。换句话说，这些代码几乎不应当做什么事情。服务依靠自己执行的最复杂操作应当是在数据传输对象(data transfer object ,DTO)和域实体对象之间转换数据。服务也会执行一些简单的数据验证，类似于在网页或 Windows 窗体上完成的操作。除此之外，服务应当仅调用域服务或某种业务模型对象的方法。

尽管服务的代码应当非常简单，但服务仍然是代码。因此，它仍然很可能存在缺陷，对于应用程序其他部分中所做的更改非常敏感。需要确保服务使用了正确的域服务或模型。对验证规则必须加以检验。必须对数据转换进行定期复查，以确保对实体域类所做的修改不会影响服务的操作。服务是应用程序的外部入口点，也就是说它们支持数据的自动输入；非常类似于供用户使用的网页或 Web 窗体。这就意味着也必须对服务的有效性和验证进行核实，以确保应用程序的安全性。出于这些原因，对服务进行的测试应当和对应用程序其他界面的测试一样多，这一点非常重要。

10.2 测试 WCF 服务

WCF 服务不同于域服务或实体，我们不能直接控制 WCF 服务的实例化。这使人们相信：WCF 服务不能利用依赖项注入，因此也就不能进行单元测试。这是一种谬论。当今可用的大多数流行依赖项注入框架都提供了一些工具，用于向 WCF 服务中注入依赖项。我

们做的全部工作就是修改一个服务实现的构造方式，对于如何配置 WCF 项目进行一些微小的调整。

10.2.1 为实现可测试性进行重构

和域服务与实体类似，在理想情况下，对于 WCF 服务也应当是先编写测试再编写代码。测试先行是 TDD 的一个主要原则(记住，第二个 D 表示“驱动”)。本章从一个现有 WCF 服务入手，对其进行重构，使其变为可测试的。之所以进行这一演示有多个原因。我们非常有可能遇到一些基础代码，在其中已经针对可测试性设计了域专用代码，甚至是 ASP.NET MVC 或 WPF 界面，具体来说，这里的可测试性就是指接受依赖项的能力。WCF 服务所处的地位非常独特，可以比较轻松地对其进行重构，以接受依赖项。这种服务就是一个接口。因此，它处于应用程序的边缘，也就是说在创建一个新的构造函数以接受依赖项时，应当不会破坏引用关系。另外，由于对服务的访问是通过代理来完成的，所以调用方没有参与服务类的实例化，可以随意修改实例提供程序，几乎不存在引入破坏性变化的风险。

最后，一些开发人员在体验 WCF 时需要学习曲线。之所以选择从一个现有服务入手，是你在向现有 WCF 服务中引入依赖项注入时具有更强的自信心。如果尝试建立一个服务，并同时引入依赖项注入，可能会信心不足。应当将本章学到的所有技巧都应用于以后的 WCF 服务开发中，在习惯这些框架之后，就采用一种测试先行的范例。

已经向 OSIM 应用程序中添加了一个简单的 WCF 服务，它以字符串数组形式返回 ItemTypes 列表(该示例和本书的所有其他示例一样，都可以从 wrox.com 下载)。我将该服务称为 InventoryService，并在 OSIM.ExternalServices 项目中创建它，如图 10-1 所示。

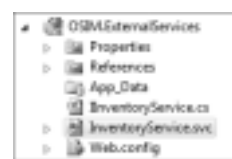


图 10-1

InventoryService 服务的实现只是创建了 ItemTypeService 域服务类的一个实例(创建该类就是为了支持这个服务)。它还会返回对 ItemTypeService 的 GetItemTypes 方法的调用结果，调用结果被平摊到一个字符串数组中。



可从
wrox.com
下载源代码

```
using System.Linq;
using Ninject;
using OSIM.Core.Services;
using OSIM.Persistence;

namespace OSIM.ExternalServices
{
    public class InventoryService : IInventoryService
    {
        public string[] GetItemTypes()
        {
            var kernel =
```



```

        new StandardKernel(new PersistenceModule(), new
CoreServicesModule());
        var itemTypeService = kernel.Get < IItemTypeService > ();

        var itemTypeList = itemTypeService.GetItemTypes()
            .Select(x => x.Name)

            .ToArray();
        return itemTypeList;
    }
}
}

```

InventoryService.svc.cs

要实现 `InventoryService` 的 `GetItemTypes` 方法，创建 `Ninject StandardKernel` 类的一个实例，并向它提供 `PersistenceModule`(其中包含用于创建 `ItemTypeRepository` 并将其连接到数据库的规则和逻辑)和 `CoreServicesModule`(其中包含一些规则，用于实例化一个实现 `IItemTypeService` 接口的类)。这些模块中的代码很简单，与示例中的其他部分没有关联，所以这里不再对其进行详细解释。可以自由查看从 www.wrox.com 下载的示例代码。

首先，使用 `StandardKernel` 的实例获取一个实现 `IItemTypeService` 接口的对象实例。然后调用 `GetItemTypes` 方法，这一方法属于被调用的 `IItemTypeService` 接口的实现，它会返回一个对象的实例，实现 `ICollection<ItemType>` 接口(`ItemType` 对象的列表)。

`ItemType` 类有两个属性：`Id` 和 `Name`。`GetItemTypes` 方法返回系统中使用的 `ItemType`(应当是从数据存储中提取)。由于只需要名称，因此可以使用 `Select LINQ` 命令，仅从 `GetItemTypes` 方法调用返回的 `ItemTypes` 列表中，选择每件物品的名称。然后利用 `ToArray` 扩展方法将 `Select` 命令的结果转换为字符串数组，该数组是由服务返回的，如图 10-2 所示。

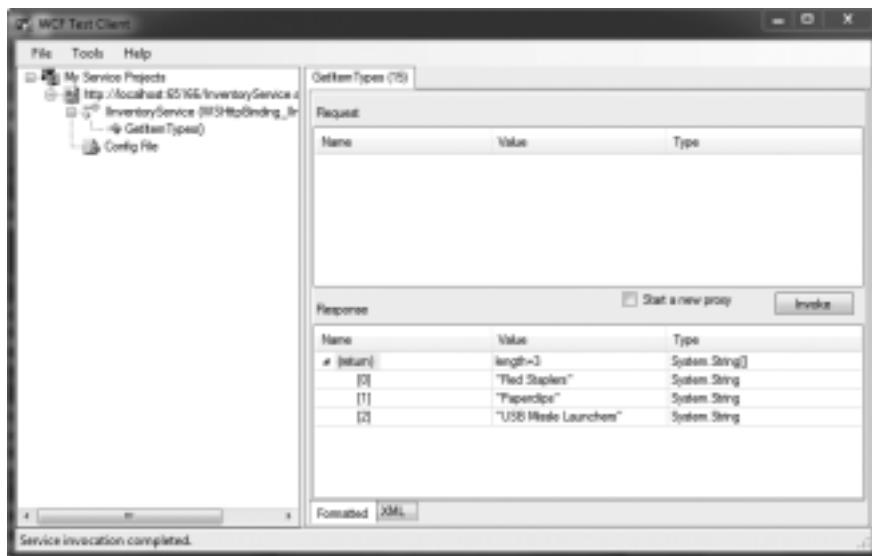


图 10-2

10.2.2 向服务引入依赖项注入

尽管 InventoryService 中的代码非常简单,仍然需要测试。测试 InventoryService 的一个基本障碍是它依赖于 IItemService 接口的一个实现,而现在无法注入一个实现 IItemService 接口的对象实例。这就意味着这个测试依赖于由静态绑定 Ninject 模块提供的 IItemService 接口的实现。

WCF 服务应用程序与 ASP.NET 应用程序和 Windows 窗体/WPF 应用程序有一个类似的共同特性:WCF 服务的实例化、ASP.NET 页面和 Windows 窗体是由 .NET 运行时执行的。如果不深入钻研 .NET 运行时,在这些对象的实例化方面就会受到限制。由于越来越多的开发人员希望能针对这些对象使用依赖项注入,因此微软创建了一些可以连接到 .NET 运行时的挂接(钩),使我们也能针对这些类型的对象使用依赖项注入。

Ninject 有一个大型扩展库,用于在各种不同情况下通过 Ninject 支持依赖项注入。库中包括 Ninject.Extensions.Wcf 扩展,它提供了一组类,用以通过 Ninject 为 WCF 服务使用依赖项注入。添加这些扩展只需要几个步骤,然后就允许 WCF 服务无缝使用依赖项注入了。

Ninject WCF 扩展包含这个示例中使用的两个程序集:Ninject.Extensions.Wcf 和 Ninject.Extensions.Wcf.CommonServiceLocator。将这两个程序集以引用方式添加到 OSIM.ExternalServices 项目中,如图 10-3 所示。

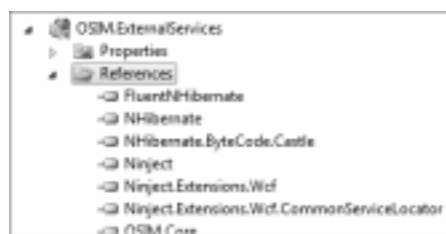


图 10-3

这两个程序集包含一些 Ninject 类用于将 WCF 服务应用程序(OSIM.ExternalServices)转换为 Ninject 应用程序。

第一个需要实现的修改就是使该 OSIM.ExternalServices 应用程序变为 Ninject WCF 应用程序的一个实例。OSIM.ExternalServices 应用程序是从一个类继承而来,修改该类就可以完成上述修改。这一信息位于 Global.asax.cs 文件。默认情况下,该文件不会在创建 WCF 服务项目时添加到项目中。这里的约定为:只在需要修改该文件提供的某一行为时(如应用程序开始某些活动时)才会创建该文件。如果该文件未被包含在里面,.NET 运行时将使用该类的标准定义来创建应用程序。

在该例中,需要改变 Global 类的基类,所以将该文件添加到 OSIM.External-Services 项目。为此,右击 OSIM.ExternalServices 项目,并选择 Add | New Item 命令。将显示如图 10-4 所示的 Add New Item 对话框。

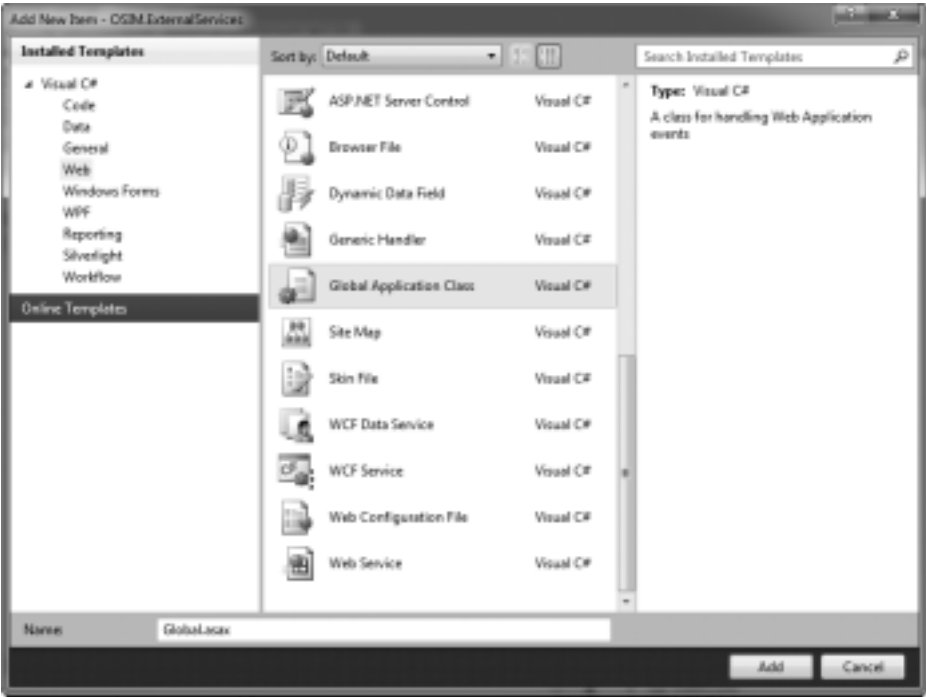



图 10-4

如图 10-4 所示，选择 Global Application Class 文件类型。保持默认名称 Global.asax 不变，然后单击 Add 按钮。Global.asax 文件被添加到 OSIM.ExternalServices 项目中，如图 10-5 所示。

查看 Global.asax 文件的内容，发现 Global 类现在是从 System.Web.HttpApplication 继承而来：



图 10-5



可从
wrox.com
下载源代码

```
namespace OSIM.ExternalServices
{
    public class Global : System.Web.HttpApplication
    {
        ...
    }
}
```

Global.asax.cs

要在 OSIM.ExternalServices 项目中使用 Ninject WCF 扩展，需要将 Global 类的声明修改为继承自 NinjectWcfApplication 基类。NinjectWcfApplication 类位于 Ninject.Extensions.Wcf 名称空间。由于还需要访问 Ninject 根名称空间中的类，所以在 Global.asax.cs 文件中添加 using 语句，以包含这些名称空间。



```
using Ninject;
using Ninject.Extensions.Wcf;
```

接下来修改 `Global` 类的声明，使它继承 `NinjectWcfApplication` 而不是 `System.Web.HttpApplication`：



```
namespace OSIM.ExternalServices
{
    public class Global : NinjectWcfApplication
    {
        ...
    }
}
```

Global.asax.cs

`NinjectWcfApplication` 基类定义了一个名为 `CreateKernel` 的抽象方法，它必须由 `Global` 类实现：



```
protected override IKernel CreateKernel()
{
    throw new NotImplementedException ();
}
```

Global.asax.cs

`CreateKernel` 方法的目的在于为开发人员提供一个引用 `Ninject` 模块的位置，这些模块中包含了一些规则，用于创建供给当前应用程序中的 WCF 服务使用的各种类。在完成 `OSIM.ExternalServices` 应用程序的改进之后返回该方法，并为 `InventoryService` WCF 服务创建一个 `Ninject` 模块。

在 `OSIM.ExternalServices` 项目中是一个名为 `InventoryService.svc` 的文件。该文件类似于 `.aspx` 文件，它提供了有关 `InventoryService` 服务端点的运行时元数据，还提供了一个链接，指向 `InventoryService.svc.cs` 代码隐藏文件中 `InventoryService` 的实际实现。



```
< %% ServiceHost Language="C#" Debug="true"
    Service="OSIM.ExternalServices.InventoryService"
    CodeBehind="InventoryService.svc.cs" % >
```

InventoryService.cs

`ServiceHost` 指令的特性之一是 `Factory`。`Factory` 特性指定应当使用哪个类来实例化 WCF 服务。如果没有提供该特性，.NET 运行时将使用标准的 WCF 服务工厂来创建服务实现的实例。为了在 `OSIM.ExternalServices` 项目中支持依赖项注入而进行重新设计时，将用于实例化 `InventoryService` 的工厂改为 `NinjectServiceHostFactory`：



可从
wrox.com
下载源代码

```
< %@ ServiceHost Language="C#" Debug="true"
Service="OSIM.ExternalServices.InventoryService"
CodeBehind="InventoryService.svc.cs"
Factory="Ninject.Extensions.Wcf.NinjectServiceHostFactory"% >
```

InventoryService.cs

要将 OSIM.InventoryService 和 InventoryService.svc 文件的内部机制转为使用 Ninject，这是需要执行的最后一个步骤。接下来的步骤都是一些标准步骤，利用 Ninject 在类中实现依赖项注入。

首先，需要创建一个 Ninject 模块，用于存储创建服务实现类的规则。调用该模块 ExternalServicesModule，将它放到 OSIM.ExternalServices 项目中一个名为 Modules 的新文件夹中，如图 10-6 所示。

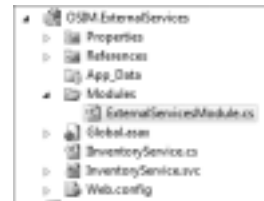


图 10-6

在 ExternalServicesModule.cs 文件中，按照前面各章创建其他 Ninject 模块的方式创建 ExternalServicesModule 类：



可从
wrox.com
下载源代码

```
using Ninject.Modules;
namespace OSIM.ExternalServices.Modules
{
    public class ExternalServicesModule : NinjectModule
    {
        public override void Load()
        {
            Bind < IInventoryService > ().To<InventoryService>();
        }
    }
}
```

ExternalServicesModule.cs

OSIM.ExternalServices 项目只有一个服务 InventoryService，所以只需要为这一个“接口/类”对提供绑定信息。我们利用适当项目中的其他模块为 ItemTypeService 提供规则，以及它需要的全部依赖项。

现在将注意力转回 Global.asax.cs 文件中 Global 类的 CreateKernerl 方法。现在已经有了为 OSIM.ExternalServices 类准备的模块，因此可以提供该方法的实现，它将创建一个内核，供 OSIM.ExternalServices WCF Services 应用程序使用。首先，需要为 OSIM.ExternalServices 项目中所需要的各种 Ninject 模块添加 using 语句：



可从
wrox.com
下载源代码

```
using OSIM.Core.Services;
using OSIM.ExternalServices.Modules;
using OSIM.Persistence;
```

ExternalServicesModule.cs

现在添加 CreateKernel 方法的实现，它基于 PersistenceModule、CoreServicesModule 和 ExternalServicesModule 提供 Ninject StandardKernel 的一个实例：



```
protected override IKernel CreateKernel()
{
    return new StandardKernel(new PersistenceModule(),
        new CoreServicesModule(),
        new ExternalServicesModule());
}
```

ExternalServicesModule.cs

现在返回实际的 InventoryService 实现。现在，它看起来如下：

```
public class InventoryService : IInventoryService
{
    public string[] GetItemTypes()
    {
        var kernel =
            new StandardKernel(new PersistenceModule(), new
CoreServicesModule());
        var itemTypeService = kernel.Get<IItemTypeService>();

        var itemTypeList = itemTypeService.GetItemTypes()
            .Select(x => x.Name)
            .ToArray();

        return itemTypeList;
    }
}
```

InventoryService.svc.cs

要将 InventoryService 从依赖于静态绑定的服务转换到注入服务，需要向 InventoryService 类添加一个构造函数，以接受一个实现 IItemTypeService 接口的对象实例，并将它存储为成员变量：



```
private IItemTypeService _itemTypeService;

public InventoryService(IItemTypeService itemTypeService)
{
    _itemTypeService = itemTypeService;
}
```

ItemTypeService.svc.cs

最后一步是从 GetItemTypes 方法中删除用于以下目的的代码：创建 Ninject 内核，并

使用该内核获取实现 `IItemTypeService` 类的实例。还需要将任何调用 `IItemTypeService` 本地实例的代码修改为使用成员变量 `_itemTypeService`：



```
public string[] GetItemTypes()
{
    var itemTypeList = _itemTypeService.GetItemTypes()
        .Select(x => x.Name)
        .ToArray();

    return itemTypeList;
}
```

ItemTypeService.svc.cs

马上就可以注意到：`GetItemTypes` 方法中的代码变得整洁多了，更容易理解。编译应用程序，并再次从 WCF Test 客户端调用 `GetItemTypes` 方法，以验证该服务仍能正常工作，如图 10-7 所示。

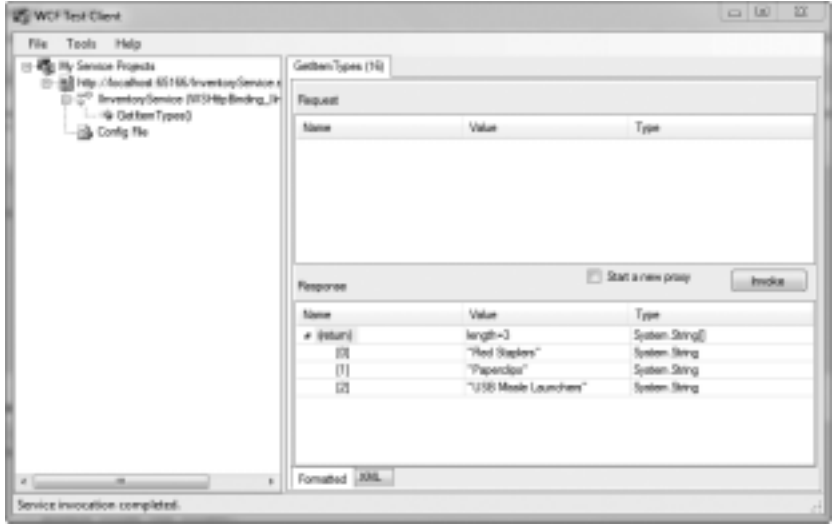


图 10-7

10.2.3 编写测试

现在可以注入实现 `IItemType` 接口的类的模拟实例，所以可以为 `InventoryService` WCF 服务编写单元测试。首先创建一个 `cs` 文件，用来保存单元测试。我已经把该文件放在 `OSIM.UnitTests` 项目的 `OSIM.ExternalServices` 文件夹中，称之为 `InventoryServiceTests.cs`，如图 10-8 所示。

要测试 `InventoryService`，需要向 `OSIM.UnitTests` 项目添加对 `OSIM.ExternalServices` 项目的引用，如图 10-9 所示。

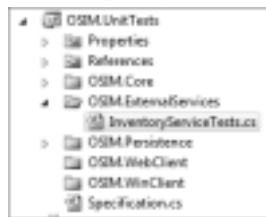


图 10-8

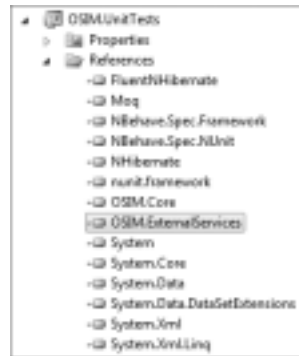


图 10-9

接下来，使用前面使用过的 BDD 命名风格，创建测试类及其基类：



可从
wrox.com
下载源代码

```
namespace OSIM.UnitTests.OSIM.ExternalServices
{
    public class when_using_the_external_inventory_service : Specification
    {
    }

    public class and_getting_a_list_of_item_types :
    when_using_the_external_inventory_service
    {
    }
}
```

InventoryServiceTests.cs

要使用 NUnit 框架中的特性，需要添加一个 using 语句，以引入 NUnit.Framework 名称空间。还需要创建 InventoryService 类的一个实例，所以也需要 OSIM.ExternalServices 名称空间。因为将创建 IItemTypeService 核心域服务的一个模拟实例，所以需要为 Moq 名称空间及 OSIM.Core.Services 名称空间添加 using 语句。最后，和前面的单元测试一样，使用 NBehave 提供的扩展方法来评估结果，所以需要包含 NBehave.Spec.NUnit 名称空间。



可从
wrox.com
下载源代码

```
using Moq;
using NBehave.Spec.NUnit;
using NUnit.Framework;
using OSIM.Core.Services;
using OSIM.ExternalServices;
```

InventoryServiceTests.cs

该测试的 Because_of 方法执行 InventoryService 的 GetItemTypes 方法，并将所得到的字符串数组存储到成员变量结果中：



可从
wrox.com
下载源代码

```
protected override void Because_of()
{
    _result = _inventoryService.GetItemTypes();
}
```

InventoryServiceTests.cs

当然，这就需要声明 `_result` 和 `_inventoryService` 成员变量：



可从
wrox.com
下载源代码

```
private IInventoryService _inventoryService;
private string[] _result;
```

InventoryServiceTests.cs

接下来，为该测试类创建实际测试方法。为了确保 `InventoryService` 正确执行，需要验证 `_result` 数组中的元素数目正确，并验证让 `IItemTpeService` 的模拟实现返回的 3 项就是 `_result` 数组中表示的各项。该方法称为 `then_a_list_of_item_types_should_be_returned`：



可从
wrox.com
下载源代码

```
[Test]
public void then_a_list_of_item_types_should_be_returned()
{
    _result.Count().ShouldEqual(_expectedNumberOfItems);
    _result.OfType < string > ().Select(x => x == _itemOneName)
        .FirstOrDefault()
        .ShouldNotBeNull();
    _result.OfType < string > ().Select(x => x == _itemTwoName)
        .FirstOrDefault()
        .ShouldNotBeNull();
    _result.OfType < string > ().Select(x => x == _itemThreeName)
        .FirstOrDefault()
        .ShouldNotBeNull();
}
```

InventoryServiceTests.cs

该示例使用 `Select LINQ` 语句来验证在测试设计中给出名字的各项存在于 `_result` 字符串数组中。现在需要添加一个 `using` 语句，将 `LINQ` 名称空间引入该类：



可从
wrox.com
下载源代码

```
using System.Linq;
```

InventoryServiceTests.cs

在 `then_a_list_of_item_types_should_be_returned` 测试方法中使用了几个成员变量，用于表示这个测试的期望值。需要在测试类中声明这些成员变量：



可从
wrox.com
下载源代码

```
private int _expectedNumberOfItems;
private string _itemOneName;
private string _itemTwoName;
private string _itemThreeName;
```

该测试编译通过，所以现在可以运行它并观察其结果了。这时，我预料测试会失败，事实也的确如此(如图 10-10 所示)。



图 10-10

10.2.4 实现依赖项的存根

InventoryServiceTests.cs 文件的第 41 行调用了 `_inventoryService` 成员变量的 `GetItemTypes` 方法。在前面各章中曾经强调，除非是因为测试失败而需要编写代码，否则不要编写任何代码。在编写测试时，这条规则同样适用。因为 `InventoryService` 拥有已经存在的应用程序代码，所以不需要另外为它编写任何代码。但我们希望确认正在创建的测试对于验证现有逻辑来说并不是过于复杂。从失败的测试中可以看到，需要在 `_inventoryService` 成员变量中提供实现了 `IInventoryService` 接口的类的实例。向 `and_getting_a_list_of_item_types` 测试类中添加 `Establish_context` 方法，以便能够提供一个实现 `IInventoryService` 接口的类的实例：



```
protected override void Establish_context()
{
    base.Establish_context();

    _itemTypeService = new Mock < IItemTypeService > ();
    _inventoryService = new InventoryService(_itemTypeService.Object);
}
```

`InventoryService` 需要一个实现 `IItemTypeService` 接口的类的实例。因为这是一个单元测试，所以想根据 `IItemTypeService` 接口提供一个模拟对象，以代替该依赖项。这就意味

着需要向类定义中添加_itemTypeService 成员变量的声明：



```
private Mock < IItemTypeService > _itemTypeService;
```

InventoryServiceTests.cs

这时，已经修正了导致该测试失败的缺陷。现在可以再次运行测试，并查看所做的工作是否足以使测试通过，如图 10-11 所示。

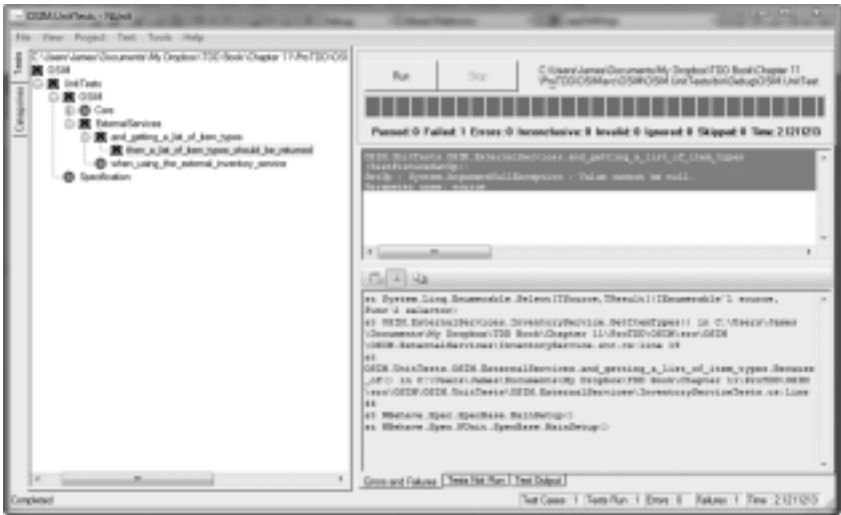


图 10-11

新的错误显示了一件很有趣的事情：InventoryService.svc.cs 的第 19 行引发了 Argument-NullException 异常：



```
var itemTypeList = _itemTypeService.GetItemTypes()  
    .Select(x => x.Name)  
    .ToArray();
```

InventoryServiceTests.cs

只要 ItemTypeService 的 GetItemTypes 方法的结果没有返回 Null，这行代码就能正常工作。现在的情况是：尝试编写一个测试，用来验证一项可能正常的功能时，发现了一个潜在的缺陷。如果没有这个测试，该缺陷可能会逃过检查。通过进行全面的单元测试，即使是 InventoryService 类中 GetItemTypes 方法这样简单的内容也不放过，使得发现了系统中的潜在错误。这个错误可能会通过 QA 流程，直到生产时才会被发现。这应当是一条教训：不要低估测试的威力。无论代码看起来多简单，随处都可能存在缺陷。

第 12 章介绍了对这一缺陷的测试和纠正过程。就现在来说，该测试的解决方案是为 IItemTypeService 的模拟提供一个存根，向 InventoryService 返回一个已填充的 ItemType 数组：



```

_itemOneName = "USB drives";
_itemTwoName = "Nerf darts";
_itemThreeName = "Flying Monkeys";
var itemTypeOne = new ItemType {Id = 1, Name = _itemOneName};
var itemTypeTwo = new ItemType {Id = 2, Name = _itemTwoName};
var itemTypeThree = new ItemType {Id = 3, Name = _itemThreeName};
var itemTypeList = new List < ItemType >
{
    itemTypeOne,
    itemTypeTwo,
    itemTypeThree
};
_expectedNumberOfItems = itemTypeList.Count;
_itemTypeService.Setup(x => x.GetItemTypes())
.Returns(itemTypeList);

```

InventoryServiceTests.cs

ItemType 存在于 OSIM.Core.Entities 名称空间中，所以需要添加一个 using 语句，将该名称空间放到这个测试类中。与此类似，还会用到 System.Collections.Generic 名称空间中的 List 类，这也需要一个 using 语句：



```

using System.Collections.Generic;
using OSIM.Core.Entities;

```

InventoryServiceTests.cs

该测试在 InventoryServiceTests.cs 文件中的完整实现现在应当类似如下：



```

using System.Collections.Generic;
using OSIM.Core.Entities;
using System.Linq;
using Moq;
using NBehave.Spec.NUnit;
using NUnit.Framework;
using OSIM.Core.Services;
using OSIM.ExternalServices;

namespace OSIM.UnitTests.OSIM.ExternalServices
{
    public class when_using_the_external_inventory_service : Specification
    {
        public class and_getting_a_list_of_item_types :
            when_using_the_external_inventory_service
        {
            private IInventoryService _inventoryService;
            private string[] _result;
            private Mock < IItemTypeService > _itemTypeService;

```

```
private int _expectedNumberOfItems;
private string _itemOneName;
private string _itemTwoName;
private string _itemThreeName;

protected override void Establish_context()
{
    base.Establish_context();

    _itemTypeService = new Mock < IItemTypeService > ();
    _inventoryService = new InventoryService(_itemTypeService.Object);

    _itemOneName = "USB drives";
    _itemTwoName = "Nerf darts";
    _itemThreeName = "Flying Monkeys";
    var itemTypeOne = new ItemType {Id = 1, Name = _itemOneName};
    var itemTypeTwo = new ItemType {Id = 2, Name = _itemTwoName};
    var itemTypeThree = new ItemType {Id = 3, Name = _itemThreeName};
    var itemTypeList = new List < ItemType >
    {
        itemTypeOne,
        itemTypeTwo,
        itemTypeThree
    };
    _expectedNumberOfItems = itemTypeList.Count;
    _itemTypeService.Setup(x => x.GetItemTypes())
        .Returns(itemTypeList);
}
protected override void Because_of()
{
    _result = _inventoryService.GetItemTypes();
}

[Test]
public void then_a_list_of_item_types_should_be_returned()
{
    _result.Count().ShouldEqual(_expectedNumberOfItems);
    _result.OfType < string > ().Select(x => x == _itemOneName)
        .FirstOrDefault()
        .ShouldNotBeNull();
    _result.OfType < string > ().Select(x => x == _itemTwoName)
        .FirstOrDefault()
        .ShouldNotBeNull();
    _result.OfType < string > ().Select(x => x == _itemThreeName)
        .FirstOrDefault()
        .ShouldNotBeNull();
}
}
```

InventoryServiceTests.cs

10.2.5 验证结果

下一步是再次运行该测试，如图 10-12 所示。

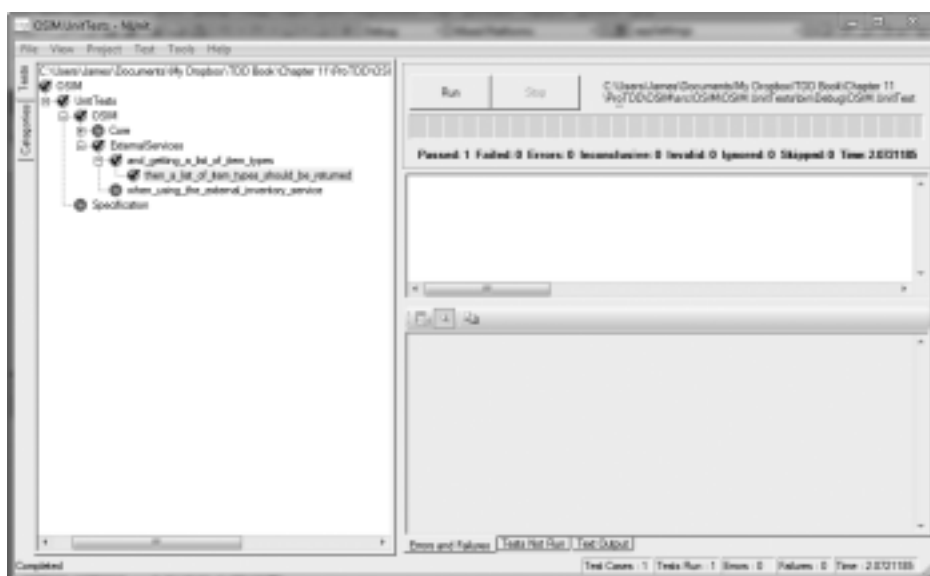


图 10-12

这个得以通过的测试表明：已经验证了 InventoryService 的功能。在开发过程逐步推进过程中，将使用该测试及其他测试来验证 InventoryService 的功能未被破坏。

10.2.6 要留意的问题多发区域

WCF 服务不同于应用程序中使用的其他类。从设计上来看，它们通过一个通信通道与其他应用程序和过程通信。该通信通道利用一些传输协议向 WCF 服务传送信息或从中接收信息。

最常用的传输协议是 HTTP 和 TCP。WCF 还通过命名管道提供机器之间的通信，通过微软消息队列(Microsoft Message Queuing, MSMQ)提供队列传送。除了这些主要的传输方法之外，WCF 还提供了一种开放的体系结构，利用它可以创建自定义通道，以使用任何可用传输方法。

大多数情况下，一个服务所选择的传输方法并不会对测试造成影响。但在某些情况下，在设计 WCF 服务时可能利用了一种具体传输类型的特定属性。如果能够异步调用 WCF 服务，也可能会使 WCF 服务的测试变得复杂。

对于这些情况应当稍微多考虑些。应当考虑如何调用这一服务实现的被测方法，使其更接近准备采用的传输的属性。这些情况都是服务开发的边缘情况，但重要的是要知道这些情况的存在。出现这些情况时，我经常求助于互联网。有些开发人员面临的挑战与你类似，他们经常访问“堆栈溢出”(stackoverflow.com)和“MSDN 开发人员论坛”(social.

msdn.microsoft.com/Forums/en-US/categories)等网站。在有疑问时，可以在这些网站上寻求别人的建议。这样可以让你少受很多挫折，并有助于保持 WCF 服务的可测试性。

10.3 本章小结

WCF 服务在 .NET 应用程序中日益普遍。由于用户需要改变，再加上其他一些因素，如移动设备变得更加流行，所以对服务的需要将只会上升。在应用程序开发实践中，拥有针对 WCF 服务的测试策略是非常重要的。

设计出色的 WCF 服务中是不包含业务逻辑的。它们仅依赖于应用程序域层中的业务逻辑，WCF 服务就是使用利用这个域层。WCF 服务本身应当仅关注自己特有的处理内容，如验证和类型转换。这样，WCF 服务就非常简短。由于这些服务仍然是代码，所以仍然需要测试。

利用 Ninject 等依赖项注入框架，可以从 WCF 服务中清除静态绑定的依赖项。注入依赖项的功能打开了可测 WCF 服务的大门。一旦 WCF 服务应用程序可以支持依赖项注入，那对 WCF 服务的测试就变得与其他类一样简单了：为模拟依赖项提供测试实现，并验证其功能。

第 11 章

测试 WPF 和 Silverlight 应用程序

作者：Michael Eaton

本章内容

- 为何测试 WPF 和 Silverlight 应用程序非常困难
- WVM 模式基础知识

Windows Presentation Foundation(Windows 表示基础 ,WPF)和 Silverlight 都是功能强大的框架，可以让我们以可视方式创建出非常出色的应用程序。这两种框架都使用可扩展应用标记语言(Extensible Application Markup Language ,XAML)作为其标记语言。WPF 用于创建桌面应用程序，而 Silverlight 则用于开发大多数浏览器应用程序和 Windows Phone 7 应用程序。尽管这些技术的功能都非常强大，但要测试使用这些框架开发的应用程序却可能非常困难。沿着前面介绍的 ASP.NET 和 WinForms 等技术的足迹，WPF 和 Silverlight 都默认采用代码隐藏文件。代码隐藏文件对于快速获取应用程序并使其能够运行是非常出色的，但也正因为这些文件，使得为应用程序编写自动化测试变得极为困难，甚至成为不可能的事情。基于代码隐藏的应用程序也可能很难维护，特别是在添加越来越多的功能时。在 ASP.NET 的世界里，MVC 模式已经成为创建高度可测试应用程序的标准，并能避免代码隐藏的痛点。

到本章结束时，就能理解对用户界面的测试为什么很困难了。我们将会学习一些模式，用来减少在测试基于 WPF 和 Silverlight 的应用程序时所遇到的困难，即使不能减少大多数

此类困难 ,至少也能减少其中的一部分 ,还将看到如何使用 TDD 创建一个 WPF 应用程序。



提示 :

本章假定你拥有基本的 WPF 和 Silverlight 知识和技巧。如果不熟悉这些技术 ,有许多有关这些主题的书籍和网站可供阅读。

11.1 测试用户界面时的问题

尽管 XAML 在用户界面和逻辑之间进行了非常清晰的区分 ,但代码隐藏文件的使用又将两者紧密耦合在一起。这种耦合意味着不能在没有用户界面的情况下实例化用户隐藏。在基于代码隐藏的应用程序中 ,用于处理用户交互、验证甚至是对数据库或其他服务进行调用的所有用户代码最终都可以采用代码隐藏方式。在自动化测试中 ,绝对不希望为了测试用户界面背后的逻辑而被迫创建该界面。当然有可能从一个单元测试创建该用户界面 ,但这样会使测试变得更为困难 ,更容易产生错误。当隐藏代码与其他服务交互时 ,这一过程会变得更为困难。

要为 WPF 和 Silverlight 应用程序编写自动化测试 ,需要采用一种方式将逻辑和用户界面完全隔离 ,以便不需要用户界面就能实例化逻辑 ,反之亦然。有很多方法可以做到这一点 ,但最近出现了一种模式 ,已经成为 WPF 开发的事实标准 ,而且在 Silverlight 开发中也正变得越来越流行 ,下文将对此进行讨论。

11.1.1 MVVM 模式

尽管 “ 模型-视图-视图模型 ” (MVVM ,其发音可以为 “ moovem ”)模式不是万能钥匙 ,但它的确使测试变得轻松 ,具体做法就是将屏幕级别的代码放在一个完全与 XAML 隔离的类中。MVVM 不仅有助于提高可测试性 ,还有助于提高可重用性。如果与用户体验(UX)设计师一同合作 ,那就可以专注于这些标记 ,而不用担心编程或业务逻辑。Microsoft Expression Blend 是一项以设计人员为中心的出色工具 ,它可以完美地与 MVVM 模型一同协作。图 11-1 是有关 MVVM 工作方式的高阶综述。

一言以蔽之 ,模型包含数据 ,视图呈现数据 ,视图模型(View Model)将两者结合在一起 ,根据需要对数据进行排序和筛选。视图模型还为视图提供了方法和命令。每个视图(XAML)有一个相应的“视图模型”。视图模型对视图一无所知 ,但视图通常拥有关于视图模型的详细信息 ,至少知道它要被绑定到的内容。还有可能是多个视图使用同一视图模



图 11-1

型。每个视图模型都根据自己的需要进入必要的模型。这些模型本身就是原来的旧代码对象，它们只是数据的容器而已，尽管它们可能是域模型，也可能不是。

视图为了使用视图模型，它必须绑定到视图模型公开的公共属性。这一操作通常是通过以下方式完成的：将视图的 DataContext 设置为视图模型的实例，然后将各个控件绑定到视图模型的各个属性。因此，几乎可以避免代码隐藏中的所有代码，至少可以避免那些与填充控件有关的代码。如果拥有 WinForms 或经典 Visual Basic(第 3 版到第 6 版)的背景知识，数据绑定听起来总是不错的，但当超出基本情景之外使用时，它就崩溃了。Silverlight 和 WPF 中的数据绑定相对于过去的数据库绑定是一个非常重大的改进，使 MVVM 应用变得很容易。下面的代码给出一个模型(ViewModel)和一个视图的基本示例：

```
public class MyModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class MyViewModel : INotifyPropertyChanged
{
    Private MyModel _instanceOfMyModel;
    public MyViewModel(MyModel instanceOfMyModel)
    {
        _instanceOfMyModel = instanceOfMyModel;
    }

    private string _firstName;
    public string FirstName
    {
        get { return _firstName; }
        set
        {
            _firstName = value;
            OnPropertyChanged("FirstName");
        }
    }

    private string _lastName;
    public string LastName
    {
        get { return _lastName; }
        set
        {
            _lastName = value;
            OnPropertyChanged("LastName");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void OnPropertyChanged(string propertyName)
    {

```

```

        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

< Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525" >
    < Grid >
        < TextBox Text="{Binding FirstName}" / >
        < TextBox Text="{Binding LastName}" / >
    < /Grid >
< /Window >

```

要将它们结合在一起工作，所使用的代码非常简单：

```

var vm = new MyViewModel(new MyModel { FirstName = "John", LastName = "Doe" });
var view = new MainWindow { DataContext = vm };
view.Show();

```

我们注意到，该示例中的构造函数接收了一个实参：

```

public MyViewModel(MyModel instanceOfMyModel)
{
    _instanceOfMyModel = instanceOfMyModel;
}

```

本例中，该实参是一个模型的实例，其中保存了希望视图显示的数据。尽管看起来以这种方式传递数据显得有些奇怪，但这样就可以让我们在使用 Ninject 等依赖项注入框架了。

INotifyPropertyChanged 界面是数据绑定的一个关键组件。当一个属性变化时，使用它来通知绑定客户端。例如，如果一个文本框被绑定到 ViewModel 的属性，而 ViewModel 实现 INotifyPropertyChanged，只要在这一属性发生变化时引发 NotifyPropertyChanged 事件，该视图就会得到通知并进行更新。

尽管有可能在没有框架的情况下使用 MVVM 模型，但强烈建议避免自己编写所有 MVVM 管道，要找一个自己习惯使用的框架。使用框架可以让我们更多地关注业务，而不是模式的实现细节。一些更流行的 MVVM 框架包括 :Prism(<http://compositewpf.codeplex.com>)，由微软模式与实践(Microsoft Patterns and Practices)团队创建和维护；MVVM Light(<http://mvvmlight.codeplex.com/>)，由 Laurent Bugnion 编写；MVVM Foundation(<http://mvvmfoundation.codeplex.com/>)，由 Josh Smith 编写；Cinch(<http://cinch.codeplex.com/>)，由 Sacha Barber 编写；Caliburn. Micro(<http://caliburnmicro.codeplex.com/>)，由 Rob Eisenberg 编写。

每种框架都有自己的强项和弱项，所以应当对所有这些框架进行考察，然后根据自己的需要做出明智的决策。我比较喜欢的框架是 Caliburn. Micro，因此除非另行指明，本章的所有示例都使用这一框架。它为表格带来的一大好处就是能够在不编写任何绑定语句的

情况下实现绑定。它只需要知道如何为类、属性和方法命名就能完成绑定。听起来似乎有些让人迷惑，但 Caliburn.Micro 实际上在后台创建了必要的绑定语句——它只不过为我们清除了这一特定步骤。Caliburn.Micro 极为灵活，允许调整它的大多数行为。由于它是开源的，因此，如果它所做的与你所想的不完全一样，你可以做一点偶然要做的事情，并修改代码。我很少做那些 Caliburn.Micro 能做的事情，所以一定要查看前面给出的网站和 Eisenberg 的博客。如果公司对开源有抵触情绪，那 Prism 则是一个非常出色的框架。

11.1.2 MVVM 如何使 WPF/Silverlight 应用程序可测试

MVVM 允许将用户界面和管理它的代码完全隔离。有了这种隔离，就可以实例化管理代码，而不用担心用户界面的显示。在一个自动化测试套件中，最不愿意做的事情就是担心出现一个窗口，需要单击按钮或输入一些文本。

使用 MVVM 模式的典型 WPF 应用程序拥有不同形式的模型(Model)、视图(View)和视图模型(ViewModel)，还有各种其他文件夹和类。这里将应用程序的结构形式设置为如图 11-2 所示。

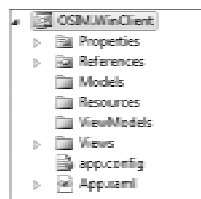


图 11-2

很多时候，这些模型位于一个完全与视图和视图模型相隔离的项目中。

要测试使用 MVVM 模式的应用程序时，将所有测试精力投注到视图模型。事实上，在考虑视图模型时，比较好的方式应当是：直接与视图模型进行的交互应当与使用视图完全类似，至少从终端用户的角度来说是这样的。考虑以下用户情景：

一个用户应当能够搜索物品类型。

如果将这一情景分解为更小的片段，则需要以下对象：

- 一个文本框，用于输入要搜索的物品
- 一个 Search 按钮，用于启动搜索过程
- 一个列表框，用于显示搜索结果
- 一个 Clear 按钮，供用户在希望重置搜索条件时使用
- 一个 Cancel 按钮，用于关闭 Search 屏幕

我坚信，在满足所有需求之前不要允许用户执行操作。在该例中，必须在登录过程中处理以下需求：

- 在输入搜索条件之前，用户不能单击 Search 按钮来启动搜索过程。
- 只有输入某些要搜索的内容之后，用户才能单击 Clear 按钮。
- 如果用户搜索成功，则在列表框中显示结果。
- 用户可以在任意时间单击 Cancel 按钮。

如果使用标准的隐藏代码来处理按钮单击事件和数据输入验证，会很难实现测试自动化，最终只能手动启动应用程序，并输入希望关注哪些内容，以查看结果是否与预期相同。

更好的做法是开始针对待编写视图模型编写测试。



提示：

Caliburn.Micro 的原则之一是关于配置约定的思想。因此，我为我的视图模型以及它们所包含的方法和属性采用了一个特定的命名约定。如果我的约定看起来有些奇怪，或者与你目前使用的约定不匹配，那也不用太着急，因为 Caliburn.Micro 允许定制所使用的约定。

利用 BDD 命名风格，首先为最初的测试集合编写基类：



可从
wrox.com
下载源代码

```
public class when_attempting_to_search : Specification
{
}
```

SearchTests.cs

有了基类之后，就可以转而编写测试，处理搜索条件属性为空的情况了。记住，如果搜索条件属性为空，意味着用户没有输入任何内容，那应当不允许用户进行搜索。该测试的 Because_of 方法将 Criteria 属性设置为 string.Empty，然后将 SearchViewModel 的 CanSearch 属性的布尔结果放在 _result 变量中：



可从
wrox.com
下载源代码

```
Using System;
Using NBehave.Spec.NUnit;
Using NUnit.Framework;
Namespace OSIM.UnitTests.OSIM.WinClient
{
    Public class and_the_search_criteria_is_blank : when_attempting_to_search
    {
        Protected override void Because_of()
        {
            _searchViewModel.Criteria = string.Empty;
            _result = _searchViewModel.CanSearch;
        }
        . . .
    }
}
```

SearchTests.cs

紧跟在 Because_of 方法之后，声明所需要的变量，然后在 Establish_context 方法中创建一个实际的 SearchViewModel 实例：



可从
wrox.com
下载源代码

```
Private bool _result;
Private ISearchViewModel _searchViewModel;
Protected override void Establish_context()
{
```

```

        base.Establish_context();
        _searchModel = new SearchViewModel();
    }

```

SearchTests.cs

then_cansearch_should_be_false 测试只是检查_result 变量(其中保存了 CanSearch 属性的值), 以确保它为假:



```

[Test]
Public void then_cansearch_should_be_false()
{
    _result.ShouldEqual(false);
}

```

SearchTests.cs

同样, 由于首先编写了测试, 还缺少 ISearchViewModel, 因此它不会成功建立。首先介绍该界面的基本内容, 然后仅编写足以使该测试通过的代码:



```

public interface ISearchViewModel
{
    string Criteria { get; set; }
    bool CanSearch { get; }
}

```

SearchViewModel.cs

接下来是 ISearchViewModel 的基本要求具体实现:



```

public class SearchViewModel : PropertyChangedBase, ISearchViewModel
{
    public string Criteria { get; set; }
    public bool CanSearch { get { return true; } }
}

```

SearchViewModel.cs

PropertyChangedBase 是 Caliburn.Micro 框架的一部分。它是 INotifyPropertyChanged 的一个实现, 其中包含一个很方便的方法, 使得不再需要在视图模型中到处散布魔力字符串(magic string)。INotifyPropertyChanged 背后的整体思路是: 当视图模型中的一个属性被绑定到视图中的控件后, 对这个属性所做的任何修改都会自动反映到视图中。尽管实现 INotifyPropertyChanged 并不是这些测试的需求, 但如果知道应用程序将会需要它, 就像本例中的情况, 那就可以在这里包含它, 尽管现在对它的需求还不是非常明显。在使绑定变得真正卓越的因素中, INotifyPropertyChanged 是其中之一, 但它又的确为编写视图模型的过程添加了一些阻力。使用 Caliburn.Micro 等框架会减少一点此类阻力。

在将 PropertyChangedBase 添加到 SearchViewModel 之后,需要修改属性设置方法,以生成该通知。Caliburn.Micro 为此提供了两种方法。一种方法是以属性的名称为字符串,就像已经自己实现了 INotifyPropertyChanged:

```
NotifyOfPropertyChange("Criteria");
```

另一种方法是采用 lambda 表达式:

```
NotifyOfPropertyChange(() => Criteria);
```

我个人偏爱 lambda 表达式,因此对于该示例来说,需要扩展 SearchViewModel 中的 Criteria 属性,以使用第二版本:



```
Private string _criteria;
Public string Criteria
{
    Get { return _criteria; }
    set
    {
        _criteria = value;
        NotifyOfPropertyChange(() => Criteria);
    }
}
```

SearchViewModel.cs

运行此测试将会产生错误和熟悉的红色条。还记得这行代码吗?



```
Public bool CanSearch { get { return true; } }
```

SearchViewModel.cs

在 CanSearch 中,true 值是硬编码的,因为希望确保该测试会失败,以便能够遵循“红灯-绿灯-重构”工作流程。因为该测试希望确保 Criteria 属性中必须有些内容,而且因为希望现在通过测试,所以对该属性进行以下修改:



```
Public bool CanSearch
{
    Get { return !string.IsNullOrEmpty(Criteria); }
}
```

SearchViewModel.cs

重新运行该测试应会通过,并得到一个绿色条。现在重构该代码,使其做一点返回硬编码值之外的事情了:

```
Public bool CanSearch
```

再次运行测试时将会通过，并得到一个绿色条。

下面的测试确保在 Criteria 属性中有取值时用户能够进行搜索：



```
Public class and_the_search_criteria_is_not_blank : when_attempting_to_search
{
    Private bool _result;
    Private ISearchViewModel _searchViewModel;

    Protected override void Establish_context()
    {
        base.Establish_context();
        _searchViewModel = new SearchViewModel();
    }

    Protected override void Because_of()
    {
        _searchViewModel.Criteria = "foo";
        _result = _searchViewModel.CanSearch;
    }

    [Test]
    Public void then_cansearch_should_be_true()
    {
        _result.ShouldEqual(true);
    }
}
```

SearchTests.cs

因为有了在 and_the_search_criteria_is_blank 中编写的代码，所以该测试现在首次通过了。

要满足用户情景的需求，仍然必须编写测试，以帮助驱动视图模型的其余部分。前面编写的第一个测试只是为了确保在搜索条件属性不为空时，用户能够启动搜索过程。

如果用户希望单击 Clear 按钮，将 Search 条件属性重置为空时，那会怎么样呢？只有用户输入了要搜索的内容之后，才能启用 Clear 按钮。针对这一要求编写两个测试。第一个测试 Criteria 为空时的情况：



```
Public class and_wanting_to_clear_when_criteria_is_empty :
when_attempting_to_search
{
    Private bool _result;
    private ISearchViewModel _searchViewModel;
    protected override void Establish_context()
    {
        base.Establish_context();
        _searchViewModel = new SearchViewModel();
    }

    Protected override void Because_of()
```



```

    {
        _result = _searchModel.CanClear;
    }

    [Test]
    Public void then_canclear_should_be_false()
    {
        _result.ShouldEqual(true);
    }
}

```

SearchTests.cs

当然，这一代码不会编译通过，因为 SearchViewModel 没有 CanClear 属性。要添加这个属性是相当简单的。首先，需要更新 ISearchViewModel 接口：



```
bool CanClear { get; }
```

SearchViewModel.cs

SearchViewModel 中 CanClear 的实现也是非常简单的：



```

public bool CanClear
{
    get { return false; }
}

```

SearchViewModel.cs

绿色条显示该测试失败，所以现在该使其通过了。这也非常容易，只要改变 CanClear 属性即可：



```

public bool CanClear
{
    get { return !string.IsNullOrEmpty(Criteria); }
}

```

SearchViewModel.cs

在进行这一修改之后，应当显示绿色条。现在该编写相对的测试了：



```

public class and_wanting_to_clear_when_criteria_is_not_empty :
    when_attempting_to_search
{
    private bool _result;
    private ISearchViewModel _searchModel;
    protected override void Establish_context()
    {
        base.Establish_context();
        _searchModel = new SearchViewModel();
    }
}

```

```

        _searchModel.Criteria = "foo";
    }
    protected override void Because_of()
    {
        _result = _searchModel.CanClear;
    }

    [Test]
    public void then_cancel_clear_should_be_false()
    {
        _result.ShouldEqual(true);
    }
}

```

SearchTests.cs

现在已经编写了一些测试，以确保用户先输入搜索内容之后才能实际开始搜索。还编写了一个测试，用以确保用户总是可以清除搜索内容，从头再来。如果用户输入了搜索条件，并实际搜索或清除，会发生什么呢？开始一个清除测试。用户单击 Clear 按钮时，应当发生两件事。第一，搜索条件应当被重置为空。第二，如果存在任何搜索结果，这些结果应当被清除。

因为正在使用 MVVM 模型，所以不是在 SearchButton_Click 或 ClearButton_Click 事件中编写代码，而是需要编写一些能够绑定到按钮的代码。使用 Caliburn.Micro，要做的全部工作就是在视图模型中创建一些方法，它们的名称与视图中的控件名称相同。因此，如果在视图中有一个 x.Name="Clear" 的按钮，那所需要的全部内容就是 SearchViewModel 中一个同样名为"Clear"的方法。这是使用 Caliburn.Micro 等框架的另一个原因。



可从
wrox.com
下载源代码

```

public class and_executing_clear : when_attempting_to_search
{
    private bool _canClear;
    private string _result;
    private ISearchViewModel _searchModel;
    protected override void Establish_context()
    {
        base.Establish_context();
        _searchModel = new SearchViewModel();
        _searchModel.Criteria = "foo";
    }

    protected override void Because_of()
    {
        _canClear = _searchModel.CanClear;
        _searchModel.Clear();
        _result = _searchModel.Criteria;
    }

    [Test]
    public void then_criteria_should_be_blank()

```

```

    {
        _result.ShouldEqual(string.Empty);
    }
}

```

SearchTests.cs

运行这一测试将会失败，这是因为 SearchViewModel 中缺少 Clear 方法而导致建立过程失败。在更新 ISearchViewModel 后，正确编写仅足以使此测试恰好失败的代码。下面是更新后的 ISearchViewModel：



可从
wrox.com
下载源代码

```

public interface ISearchViewModel
{
    string Criteria { get; set; }
    bool CanSearch { get; }
    bool CanClear { get; }
    void Clear();
}

```

SearchViewModel.cs

下面是 SearchViewModel 中 Clear 方法的最简单实现：



可从
wrox.com
下载源代码

```

public void Clear()
{
}

```

SearchViewModel.cs

将出现红色条，显示以下消息：



可从
wrox.com
下载源代码

```

Test 'OSIM.UnitTests.OSIM.WinClient.and_executing_clear
.then_criteria_should_be_blank' failed:
    Expected string length 0 but was 3. Strings differ at index 0.
    Expected: < string.Empty >
    But was: "foo"
    -----^

```

SearchViewModel.cs

为使此测试通过，需要修改 Clear 方法：



可从
wrox.com
下载源代码

```

public void Clear()
{
    Criteria = string.Empty;
}

```

SearchViewModel.cs

现在将会看到一切皆绿了，但仍然缺点东西。当用户单击 Clear 按钮时，不仅搜索条

件应当被重置为空，所显示的所有结果也应当被清除。

现在向 `and_executing_clear` 类中添加另一个测试：



```
[Test]
public void then_results_should_be_cleared()
{
    _searchResults.ShouldBeNull();
}
```

SearchTests.cs

由于不能建立，因此需要更新 `ISearchViewModel` 和 `SearchViewModel`，以支持搜索结果。其结果是 `List<ItemType>`：



```
public interface ISearchViewModel
{
    // previous code not shown
    List<ItemType> Results { get; set; }
```

SearchViewModel.cs

下面是最简单的实现：



```
Private List<ItemType> _results;
public List<ItemType> Results
{
    get { return _results; }
    set
    {
        _results = value;
        OnPropertyChanged(() => Results);
    }
}
```

SearchViewModel.cs

这次没有出现预期的红色条，该测试通过了！在检查刚刚编写的代码时，这一现象总是个危险信号，因为在预期出现错误但却看到测试通过时，从来都不是好事。

如果研究一下该测试，就可以看到发生了什么：



```
[Test]
public void then_results_should_be_cleared()
{
    _searchResults.ShouldBeNull();
}
```

SearchTests.cs

`Results` 的这一最简单实现总是返回 `Null`，这是因为我们还没有真正建立这些结果。将

在 Search 测试中处理这一问题,但现在可以走个捷径,修改 `and_executing_clear` 测试的 `Establish_context()` 方法,将 `Results` 设置为 `new List < ItemType > ()` :



可从
wrox.com
下载源代码

```
protected override void Establish_context()
{
    base.Establish_context();
    _searchViewModel = new SearchViewModel();
    _searchViewModel.Criteria = "foo";
    _searchViewModel.Results = new List < ItemType > ();
}
```

SearchTests.cs

再次运行该测试,将会看到熟悉的红色条和预料之中的失败。要使这一测试通过,需要确认在执行 `Clear` 时,`Results` 设置为 `Null`。`Clear` 方法现在应当类似如下:



可从
wrox.com
下载源代码

```
public void Clear()
{
    Criteria = string.Empty;
    Results = null;
}
```

SearchViewModel.cs

绿色条说明,一切都与预期中一样正常,所以现在可以编写更多测试以推动 `SignInViewModel` 类的其他部分了。前面的测试都是 `when_attempting_to_search` 规范的组成部分,都与用户能够单击 `Search` 按钮之前发生的行为有关。现在绝对要进入一个新的领域了:用户已经输入了要搜索的物品,并单击了 `Search` 按钮。尽管还没有创建视图,但在视图模型中还有大量工作要做。

首先创建名为 `when_searching` 的基类:



可从
wrox.com
下载源代码

```
Public class when_searching : Specification { }
```

SearchTests.cs

第一个编写的测试是检查用户正在搜索的物品是否存在:



可从
wrox.com
下载源代码

```
Public class and_searching_for_an_item_that_does_not_exist : when_searching
{
    Private List < ItemType > _results;
    Private bool _canSearch;
    Private ISearchViewModel _searchViewModel;
    Protected override void Establish_context()
    {
        base.Establish_context();
        _searchViewModel = new SearchViewModel();
    }
}
```

```

        _searchModel.Criteria = "foo";
    }

```

SearchTests.cs

注意 `_searchModel` 的 `Criteria` 属性如何被设置为 “foo”？这种方法用来模拟用户输入了一个并不存在的 `ItemType` 的名字。记住，在最终绑定 `SearchViewModel` 的 `Criteria` 属性中的文本框时，输入到文本框中的所有内容都反映在视图模型本身。

`Because_of()` 方法执行 `SearchViewModel` 的 `Search` 方法。完成搜索时，在 `_result` 变量中存储一个表示是否返回结果的值。注意如何存储 `CanSearch` 属性的一个副本：



```

Protected override void Because_of()
{
    _canSearch = _searchModel.CanSearch;
    _searchModel.Search();
    _results = _searchModel.Results;
}

```

SearchTests.cs

测试 `then_results_should_be_null` 完成几项工作。第一，它确保用户可以登录。这有助于捕获一些愚蠢的错误，比如没有在 `Establish_context` 中正确设置测试。然后，该测试查看是否正确地设置了 `ErrorMessage` 属性，因为在调用 `SignIn` 方法时密码不正确：



```

[Test]
Public void then_results_should_be_null()
{
    _canSearch.ShouldBeTrue();
    _results.ShouldBeNull();
}

```

SearchTests.cs

同样，由于 `SearchViewModel` 没有包含 `Search` 方法，所以建立过程失败。在向 `ISearchViewModel` 接口添加以下代码后，就可以仅在 `SearchViewModel` 中编写足够的实际代码，使这一测试失败：



```

Void Search();

```

SearchViewModel.cs

要完成该过程，编写用来实现 `SearchViewModel` 中 `Search` 方法的最基本代码：



```

Public void Search()
{
    Results = new List < ItemType > ();
}

```

SearchViewModel.cs

运行这一测试应当会失败，并产生一个红色条。现在应当重构 `SignInViewModel`，使该特定测试得以通过了：



```
public void Search()
{
    if(Criteria == "foo")
        Results = null;
    else
        Results = new List < ItemType > ();
}
```

SearchViewModel.cs

一旦该测试通过，就可以编写 `and_searching_for_an_itemtype_that_does_exist` 了。除以下代码之外，其设置相同：



```
protected override void Establish_context()
{
    base.Establish_context();
    _searchViewModel = new SearchViewModel();
    _searchViewModel.Criteria = "USB";
}

[Test]
public void then_results_should_be_null()
{
    _canSearch.ShouldEqual(true);
    _results.ShouldNotBeEmpty();
}
```

SearchTests.cs

红色条出现，说明该测试失败。回想 `Search` 的实现：



```
public void Search()
{
    if(Criteria == "foo")
        Results = null;
    else
        Results = new List < ItemType > ();
}
```

SearchViewModel.cs

要使该特定测试得以通过，最简单的方法是修改 `Search`，使其返回某些硬编码值：



```
public void Search()
{
    if (Criteria == "foo")
        Results = null;
    else
    {
        Results = new List < ItemType > ();
        if (Criteria == "USB")
        {
            Results.Add(new ItemType { Id = 1, Name = "USB Key - 2GB" });
            Results.Add(new ItemType { Id = 2, Name = "USB Key - 4GB" });
            Results.Add(new ItemType { Id = 3, Name = "USB Key - 8GB" });
            Results.Add(new ItemType { Id = 4, Name = "USB Key - 16GB" });
        }
    }
}
```

SearchViewModel.cs

运行这一测试将得到一个熟悉的绿色条。从大局来看，不是返回硬编码数据，而是调用服务或存储库类中的方法，向其传递搜索条件，并把结果设置为这些方法的输出。

11.1.3 将所有内容结合在一起

最后，该实现 SearchView 了！谢天谢地，这是一个很简单过程。我们会注意到，由于所使用的命名约定以及 Caliburn.Micro 带来的好处，根本就不需要绑定语句。Criteria 文本框被命名为 Criteria。Caliburn.Micro 将查看视图模型，以查看一个具有匹配名字的属性。一旦找到了匹配内容，就会创建实际绑定：



```
< UserControl x:Class="OSIM.WinClient.SearchView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Background="LightGray"
    Height="200" Width="325" >
    < Grid >
        < DockPanel LastChildFill="True" >
            < StackPanel
                Orientation="Horizontal"
                DockPanel.Dock="Top" >

                < TextBlock Text="Criteria:" Margin="0,0,5,0" / >
                < TextBox x:Name="Criteria" Width="100" Margin="0,0,5,0" / >
                < Button x:Name="Search" Content="Search" Margin="5,0,5,0" / >
                < Button x:Name="Clear" Content="Clear" Margin="5,0,5,0" / >

            < /StackPanel >
        < /DockPanel >
    < /Grid >
< /UserControl >
```



```

        < ListBox x:Name="Results" DisplayMemberPath="Name" / >
    < /DockPanel >
    < /Grid >
< /UserControl >

```

SearchView.xaml

该项目现在如图 11-3 所示。

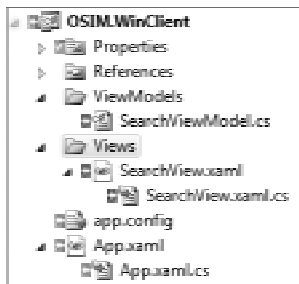


图 11-3

要利用 Caliburn.Micro，需要在 WPF 项目中做一点设置工作。首先修改 App.xaml，以清除 StartupUri 标记。应当对 App.xaml.cs 进行整理，使其基本上只包含以下内容：



可从
wrox.com
下载源代码

```

public partial class App : Application
{
    public App()
    {
    }
}

```

App.xaml.cs

下一步是向项目中添加一个“引导程序”，用来配置框架。在 WinClient 项目的根目录下，创建一个名为 WinClientBootstrapper 的类，类似如下：



可从
wrox.com
下载源代码

```

public class WinClientBootstrapper : Bootstrapper< SearchViewModel > { }

```

App.xaml.cs

这就是要做的全部工作。记住仅编写能使项目工作的最简单引导程序。Caliburn.Micro 网站(<http://caliburnmicro.codeplex.com/>)上包含了很多非常出色的内容，描述了可以用引导程序所做的全部事情，所以一定要去看看这些内容。现在向 app.xaml 中添加一点 XAML：



可从
wrox.com
下载源代码

```

< Application x:Class="OSIM.WinClient.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:OSIM.WinClient"
>
< Application.Resources >

```

```

    < ResourceDictionary >
      < ResourceDictionary.MergedDictionaries >
        < ResourceDictionary >
          < local:WinClientBootstrapper x:Key="bootstrapper" / >
        < /ResourceDictionary >
      < /ResourceDictionary.MergedDictionaries >
    < /ResourceDictionary >
  < /Application.Resources >
< /Application >

```

App.xaml.cs

如果是在编写 Silverlight 应用程序，那 App.xaml 看起来会有所不同：



```

< Application x:Class="OSIM.WinClient.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
  presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:OSIM.WinClient"
  >
  < Application.Resources >
    < local:WinClientBootstrapper x:Key="bootstrapper" / >
  < /Application.Resources >
< /Application >

```

App.xaml.cs

就是它了。这就是使用 Caliburn.Micro 所需要的全部内容。根据所使用的约定，以及对 App.xaml 和 WinClientBootstrapper 类的修改，在运行 WinClient 应用程序时，将会出现基本的搜索屏幕。注意，在 Criteria 字段中输入内容之前，Search 和 Clear 按钮是一直被禁用的。如果在 Criteria 文本框中输入了内容，然后单击了 Clear 按钮，这些内容将会被清除，Search 和 Clear 按钮再次被禁用。输入某一条件并单击 Search 按钮，将显示一个匹配 ItemTypes 的列表。图 11-4 显示了非常基本的搜索屏幕。



图 11-4

该项目的创建步骤是首先编写视图模型，然后编写测试以支持所需的功能。也可以先

从视图入手，但无论采用哪种方法，所创建的应用程序都经过了彻底的单元测试，而且满足在本章开头给出的用户情景的需求。

11.2 本章小结

WPF 和 Silverlight 都是功能非常强大的框架，可以使用 XAML 创建丰富的应用程序。用这些框架编写单元测试应用程序可能非常困难，但这不是应当避免的工作。与 WCF 和 ASP.NET 一样，重要的是对不同客户端拥有可靠的测试策略。

MVVM 不是克服代码隐藏范例的局限性，也不是试图实例化紧密耦合的类，而是促进了用户界面与其控制代码之间的真正分离。

利用 MVVM，不仅可以对应用程序进行单元测试，还可以使应用程序拥有更高的可维护性。它还有助于促进开发人员/设计人员之间的关系：让设计人员仅关注 XAML，而不用为代码操心。几种优秀的 MVVM 框架移除了实现这一模型的负担，让我们能够专注于编写优秀软件的过程。

第 部分

需求和工具

- 第 12 章 应对缺陷和新的需求
- 第 13 章 有关优秀工具的争论
- 第 14 章 结论



第 12 章

应对缺陷和新的需求

本章内容

- 理解在现代业务计算环境中应用程序为什么要灵活
- 使用 TDD 处理应用程序中的修改
- 通过 TDD 建立更灵活的应用程序
- 在应用程序中引入新功能或标识缺陷的情况下练习 TDD

应用程序开发工作不是在真空中进行的。敏捷项目方法的日益流行就是这一事实的一个证明。即使是在敏捷环境中，从列出第一条需求到部署应用程序生产，大多数开发工作也要耗时数月(甚至数年)。这样就留出了很多时间来修改需求。

即使在应用程序到达生产阶段之后，修改也不会结束。无论开发团队和 QA 部门的工作多么出色，大多数应用程序都会有一些缺陷，要等到生产阶段才会被发现。即使一个应用程序能够最终付诸生产，没有发现缺陷，来自各种外部因素(客户、政府、市场因素)的操作和影响也会提出修改应用程序的要求。

在 TDD 中，缺陷和新需求的出现都意味着可能需要创建新测试。用“测试先行”的策略来应对缺陷和新需求，就能更轻松地提交高质量应用程序，并且不会破坏已有功能。阅读本书已经帮助你获得了处理这些情况时需要的技巧和知识。现在只需要以稍微不同的方式来运用这些技巧和知识。

12.1 处理修改

修改是不可避免的。在历史上，大多数业务应用程序都未能很好地处理修改。这是因为尽管开发人员和体系结构设计师倍加小心，但大多数应用程序仍然变得脆弱。没有单元测试和集成测试套件的安全保证，大多数应用程序都会变成像“叠叠木(Jenga tower)”一样的代码。不久之后，应用程序就会变得非常不稳定，使大多数开发人员害怕触及它，唯恐整体崩溃。

这些应用程序缺乏自动化单元测试和集成测试。没有一种方法能够对修改后的应用程序自动进行递归测试，没有人能够保证应用程序在修改之后仍能正常工作。在应用程序开发过程中采用 TDD 方法，可以提供一整套此类测试，用以确保在进行修改后，应用程序本身仍然保持稳定。

TDD 的使用还有另外一个副作用：使应用程序在面对修改时能够更加灵活。采用 TDD 的开发人员似乎要比未采用 TDD 的开发人员更注意遵守 SOLID 原则，这既可能是出于设计原因，也可能是编写可测试代码时的副作用。遵守 SOLID 原则会使软件更易于维护和扩展，在无法避免修改的环境中这显然是一项好处。

12.1.1 修改的发生

一个应用程序在部署生产之后，可能会因为多种原因而需要修改。可能是公司采用了一种新的业务策略，或者确定要进入一个新市场。许多行业如金融业和医学部门都受到地方政府和中央政府的严格管控。随时都可能通过新的法律法规。某个应用程序的客户可能希望添加新的功能，或者必须对现有功能进行修改。而且没有哪个应用程序是毫无缺陷的。

无论是什么来源、什么原因，只要对应用程序进行了修改，开发人员就必须应对新的需求和缺陷。在一个从开始就一直使用 TDD 的应用程序中，方法可能比较明显。但即使应用程序不是使用 TDD 开发的，在开发新功能和修改缺陷时，仍然可以采用(也应当)这一实践方法。测试保证了应用程序具有一定的质量水平，这一质量承诺在部署后的开发工作中也应当遵守。

1. 添加新功能

在任何时候都可能向应用程序中引入新功能。在使用敏捷方法的正常开发周期内引入新功能时，没有什么值得注意的。只需要像任何其他功能一样，将它们添加到待办事项列表中并排入日程即可。在应用程序部署生产之后，如果开发团队的主体已经转向另一个项目了，再添加功能就有些不同了。

有一点非常重要：在应用程序的主开发周期结束之后添加的功能，应当像应用程序主开发周期之内所知道和建立的功能一样对待。不要掉入这样一种思考陷阱：由于应用程序

已经“完工”，所以就可以削去边边角角，以速度为借口而使质量打折扣。总是会有新功能的。也总是存在缺陷的。作为一位开发人员，允许自己以质量为代价而换取“捷径”，最终会让自己掉入灾难性的滑坡之下，最终导致一个没有完整测试套件的、不可维护的脆弱应用程序。

在向应用程序中添加一项新功能时，需要经历的过程应当与应用程序中其他功能所经历的过程相同。需要再次部署在建立应用程序其他功能时的发现和设计过程，即使新功能很小也应当如此。没有“小功能”。一位智者曾经说过：“你是根据我的大小来评价我的，是吗？”一个设计、实现不当的“小功能”会像一个处置不当的“大”功能一样，带来同样多的灾难，导致同样大的破坏、同样快速地激怒用户群。

只要新功能的设置正确，再加上来自项目业务部门和技术部门提供的信息，就可以开始开发了。和在主应用程序开发周期中开发的功能一样，新功能的开发也从测试开始。和之前应用程序开发一样，该测试也应当失败。针对新功能的测试失败，表明该测试很可能测试了正确的(不存在)功能，所关注的功能还没有在应用程序中实现。如果该测试在没有另行编写任何代码的情况下通过了，就要研究研究了。该测试是真的测试这个新功能吗？它是否在正确的位置进行测试？所讨论的功能是否已经存在于应用程序中了？是否被应用程序中的其他东西遮挡了？在继续进行后续工作之前，必须回答这些问题。

在编写了正确的测试之后，目的就与应用程序的主开发周期中一样了：应当尽量仅编写足以使测试得以通过的最少量代码。同时，必须确保现有测试都不会失败。如果这些测试开始失败，那就得判断其原因，并努力让这些测试及新测试都能通过。在所有测试都通过之后，新的功能完成，可以进行部署了。

2. 处理缺陷

在我的开发团队中，缺陷定义为代码中的一些功能，它们未能反映用户情景及/或特性中指定的规范，这就意味着只有应用程序功能不同于开发团队手中的规范才会存在缺陷。当一个业务用户告诉开发人员：“应用程序的规范指出在这种情况下应当应用 7.5% 的税率，但应用程序应用了 6% 的税率”，那他就是在描述一个缺陷。规范表明，对于一些给定情景应当应用 7.5% 的税率，但事实却并非如此。当一个业务用户告诉开发人员：“我知道规范中指出，在星期二订购 5 件以上商品的客户将会获得 15% 的折扣，但我真正的意思是当一位用户在星期五订购 100 美元或更多商品时，会为他免除运费”，那他就是在描述一项新功能。从技术上来说，如果这位业务用户真的希望删除星期二的折扣功能，那他就是在描述两项功能。

即使方法是相同的，这一区分也是非常重要的。在请求加入新功能时，需要根据开发团队的工作余量和公司指定给这项新功能的优先级来安排开发工作的进度。如果公司希望立即实现这一功能，那就必须推迟进度安排中的其他某项功能。而在接到缺陷报告时，开发团队会根据缺陷的严重程度，重新调整其工作的优先级，以尽快修正缺陷。由于质量对于开发团队来说是很重要的，所以缺陷总是拥有较高的优先级。团队的投入就是为了交付

高质量的应用程序，他们将努力修复缺陷，在迭代周期的最后，交付双方一致认可的功能。

在为缺陷开发补丁时，其方法与功能的开发方法相同。第一步是与公司讨论这一缺陷，以确认开发团队理解了应用程序应当如何运转，这一行为与应用程序目前的功能有何不同。在重新验证了业务规则之后，开始编写测试。和新功能一样，该测试应当失败。如果测试没有失败，可能意味着没有测试到正确的内容。该测试应当暴露这一缺陷。采用这种方法，不仅可以保证目前修复了缺陷，还能保证它不会重现。

12.1.2 从测试开始

第 11 章中，在开发 InventoryService WCF 服务时发现了一个潜在缺陷，如图 12-1 所示。

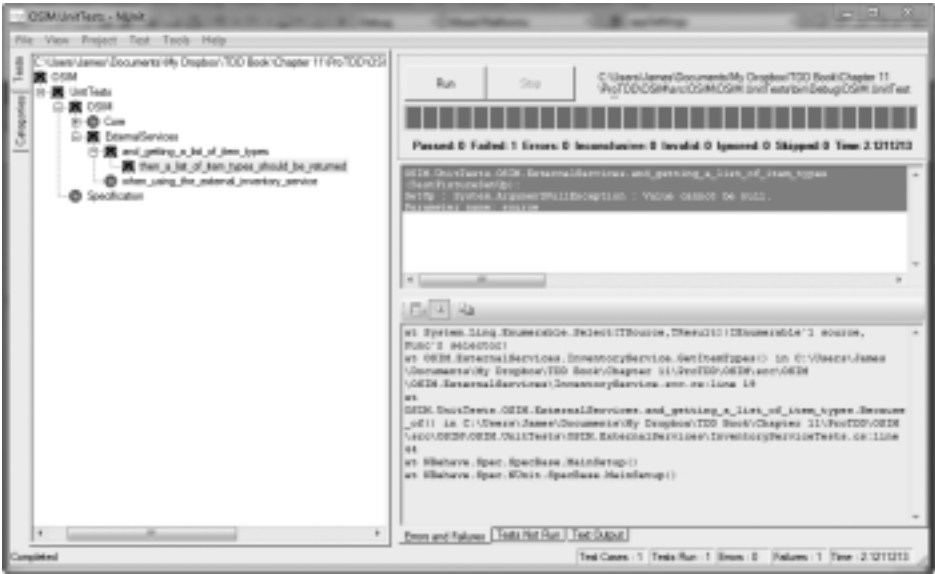


图 12-1

下面列出生成这一错误的代码：



```
public string[] GetItemTypes()
{
    var itemTypeList = _itemTypeService.GetItemTypes()
        .Select(x => x.Name)
        .ToArray();

    return itemTypeList;
}
```

InventoryService.svc.cs

当 ItemTypeService 的 GetItemTypes 方法返回 Null(空)时，就会发生这一错误。对于该方法来说，这是不希望出现的行为。当然，ItemTypeService 的 GetItemTypes 方法应当没有

理由返回 Null。即使数据存储中没有 ItemTypes，Fluent NHibernate 仍然会返回一个列表的空实例。ItemTypeService 的 GetItemTypes 方法只是将这个空列表向上传送给 InventoryService 的 GetItemTypes 方法。

那为什么要为它操心呢？在我的团队中，对于生产中的所有应用程序有一项要求：用户永远都不应当看到未处理的异常。除了潜在的安全泄露之外，大多数用户不知道如何从这种情况中恢复。如果一个应用程序停止响应，必须重新启动，使用户可能丢失工作，那会使大多数用户感到不安。需要把用户与未受处理的 .NET 异常隔离开来，使其受到保护。

处理这一缺陷时，首先创建一个测试。具体来说，我希望看到一个测试，该测试中 ItemTypeService 的 GetItemTypes 方法的存根返回一个 Null：



```
public class and_getting_a_list_of_item_types_when_the_returned_list_is_null :
    when_using_the_external_inventory_service
{
    private IInventoryService _inventoryService;
    private string[] _result;
    private Mock < IItemTypeService > _itemTypeService;

    protected override void Establish_context()
    {
        base.Establish_context();

        _itemTypeService = new Mock < IItemTypeService > ();
        _inventoryService = new InventoryService(_itemTypeService.Object);

        List < ItemType > itemTypeList = null;
        _itemTypeService.Setup(x => x.GetItemTypes()).Returns(itemTypeList);
    }

    protected override void Because_of()
    {
        _result = _inventoryService.GetItemTypes();
    }

    [Test]
    public void then_an_empty_list_of_item_types_should_be_returned()
    {
        _result.ShouldNotBeNull();
        _result.Count().ShouldEqual(0);
    }
}
```

InventoryServiceTests.cs

该测试看起来应当类似于第 10 章为验证 InventoryService WCF 服务的功能所编写的测试。事实上，它们几乎是完全相同的，只有两处例外。第一个要指出的变化是，ItemTypeService 模拟的 GetItemTypes 存根没有返回 ItemTypes 的列表，而是返回 Null。这样就重复了 ItemTypeService 类的 GetItemType 方法返回 Null 的情景。另一处变化是测试方法 then_a_

list_of_item_types_should_be_returned 没有在 List 中查找项目。它只是验证_result 成员变量不是 Null，而是空的。

运行该测试，在图 12-2 中可以看到这个测试失败了，表示在 InventoryService 中导致缺陷的情景出现了。



图 12-2

12.1.3 修改代码

现在有一个测试，能够暴露 InventoryService WCF 服务中的缺陷。在开始编写代码之前，应当运行所有这些测试，并验证针对该缺陷的测试事实上只是当前唯一失败的测试。该测试如图 12-3 所示。

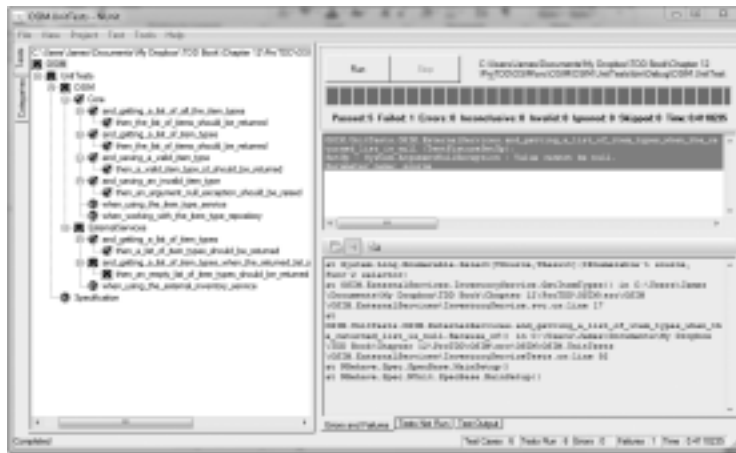


图 12-3

下一步是查看 InventoryService 的 GetItemTypes 方法中的代码，找到该缺陷发生在哪里。整个 GetItemTypes 方法如下：



```
public string[] GetItemTypes()
{
    var itemTypeList = _itemTypeService.GetItemTypes()
        .Select(x => x.Name)
        .ToArray();
    return itemTypeList;
}
```

InventoryServiceTests.cs

InventoryService.svc.cs 文件中的第 17 行在 InventoryService 的 GetItemTypes 方法中调用了 _itemTypeService.GetItemTypes。要修复这一代码很简单。在调用 Select 和 ToArray 扩展方法之前，需要有一种方法来捕获 _itemTypeService.GetItemTypes 方法的输出，并检查结果是否为 Null。如果 _itemTypeService.GetItemTypes 方法的结果为 Null，则需要返回一个空的字符串数组：



```
public string[] GetItemTypes()
{
    var itemTypeList = _itemTypeService.GetItemTypes();
    if (itemTypeList == null)
    {
        return new string[0];
    }
    var itemTypeArray = itemTypeList.Select(x => x.Name).ToArray();
    return itemTypeArray;
}
```

InventoryService.svc.cs

重新在 and_getting_a_list_of_item_types_when_the_returned_list_is_null 中运行这个测试，将会看到新代码纠正了这一缺陷，如图 12-4 所示。

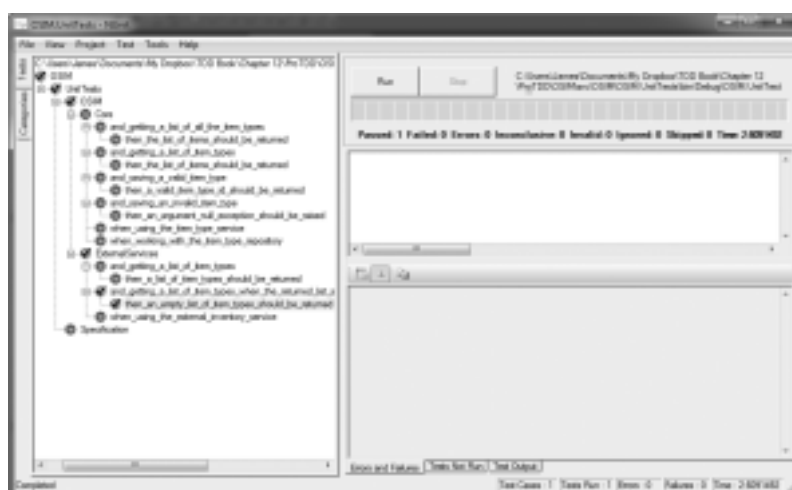


图 12-4

InventoryService 中 InventoryService 的新实现能够正常工作。但代码开始变得有点复杂和冗长了。而且也不再非常认真的遵守 SOLID 原则——特别是单一职责原则(SRP)。

应当对该代码稍微进行重构,使其更整洁一些。对 GetItemTypes 方法的主要修改是将检查_itemTypeService.GetItemTypes 方法结果是否为 Null 的功能析取出来。如果返回的 ItemTypes 的 List 不是 Null,它就析取出 List 中的信息,将它转换为一个字符串数组,由 InventoryService 返回。因此,可以通过修改代码来完成这一任务:



可从
wrox.com
下载源代码

```
public string[] GetItemTypes()
{
    var itemTypeList = _itemTypeService.GetItemTypes();
    var itemTypeArray = ReturnValidItemNameListArray(itemTypeList);

    return itemTypeArray;
}

private static string[] ReturnValidItemNameListArray
    (IEnumerable < ItemType > itemTypeList)
{
    return itemTypeList == null ?
        new string[0] :
        itemTypeList.Select(x => x.Name).ToArray();
}
```

InventoryService.svc.cs

这段代码添加了一个名为 ReturnValidItemNameListArray 的方法,它有一个形参,就是通过调用_itemTypeService.GetItemTypes 方法返回的 List。ReturnValidItemNameListArray 查看输入形参 itemTypeList,以判断它是否为 Null。如果是 Null,则 ReturnValidItemNameListArray 返回一个空字符串数组。如果 itemTypeList 为 Null,则 ReturnValidItemNameListArray 从 itemTypeList 选择名称,然后将该列表传输到一个从 ReturnValidItemNameListArray 方法返回的数组。GetItemTypes 方法只需要返回 ReturnValidItemNameListArray 方法的结果。

从 InventoryService 的 GetItemTypes 方法中析取逻辑马上就可以提高该方法的可读性。与此类似,由于 ReturnValidItemNameListArray 只做一件事情,所以其算法的可读性也更好了。在 GetItemTypes 方法中,可以仅返回 ReturnValidItemNameListArray 方法的结果,而不将它存储在 itemTypeArray 中。但在该例中,将 ReturnValidItemNameListArray 方法的结果存储在 itemTypeArray 变量中可以提高可读性,所以选择保存该变量。

在执行这一重构操作之后运行该测试,发现重构没有向代码中引入任何缺陷,如图 12-5 所示。

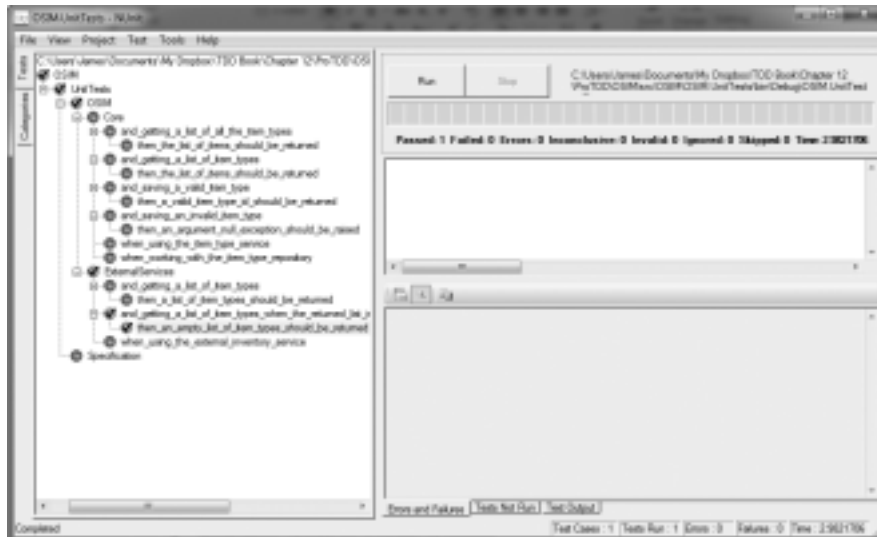


图 12-5

该测试通过了。根据定义，该重构过程并不是修改这一缺陷所需要的全部内容。现在应当验证这些代码修改没有引入更多的缺陷。

12.1.4 使测试保持通过状态

除了修复缺陷或添加功能之外，还有一点非常重要：要确保没有破坏任何已有的功能。我有一套单元测试来验证该应用程序中的所有其他功能是正常的。通过运行这一整套测试，可以验证已经修复了该缺陷，而没有向应用程序中引入更多缺陷，如图 12-6 所示。

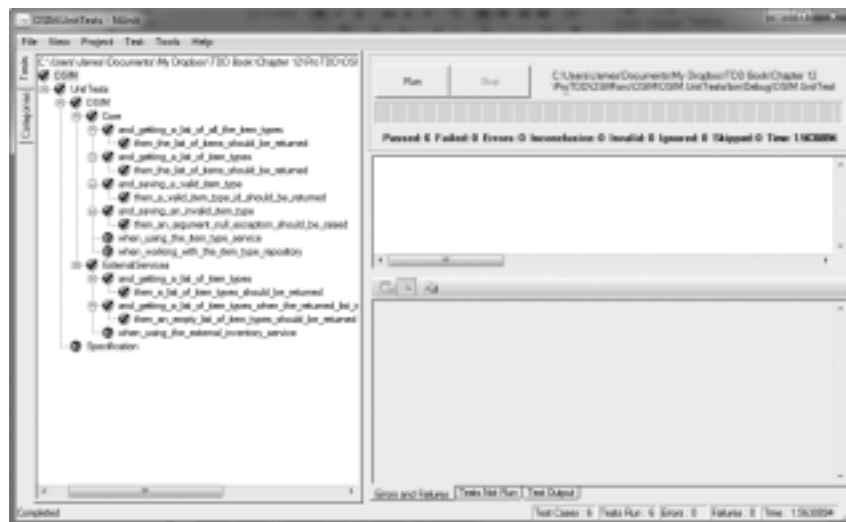


图 12-6

所有测试，包括为了修复 InventoryService WCF 服务中的缺陷而创建的新测试，都通

过了。这意味着缺陷已经被修复，而且应用程序中的其他功能都没有被破坏。现在可以将这个应用程序发送给 QA 进行检验，准备生产了。

12.2 本章小结

业务中发生变化是不可避免的。法律、市场和公司策略的改变都是很正常的。要像抗拒这些改变是没有用的。相反，应用程序开发人员需要学习如何包容改变。正确、认真地实施 TDD 可以为应用程序提供一项好处：使应用程序变得更灵活，更能适应变化。

在处理新功能时，所采用的业务分析、设计和评估协议都应当与开发应用程序时所使用的相同。即使应用程序已经交付或部署生产也应当这样。不要想着“嗯，这只是一个很小的修改”，从而掉入抄捷径的陷阱中。如果实现不当，即使是很小的修改也可能会严重损坏应用程序，失去客户的信任。

无论是处理新功能还是缺陷，开发过程的第一步都是相同的：编写测试。测试在开始时应当是失败的。如果测试通过，那就需要做一点研究。该测试是否测试了正确的功能或缺陷？是否正在测试应用程序的正确部分？新请求的功能是否已经作为某一其他功能的副产品而存在？记住，仅仅因为立即通过了一个测试，并不意味着已经修复了应用程序中的缺陷，或者新功能已经存在于其中了。

在编写了新测试并看到它失败之后，下面的步骤与应用程序的主开发期间是相同的。仅编写足以使新测试通过的最少代码。新测试通过时，要么是修复了缺陷，要么是实现了新功能。在必要时进行重构，但不要添加任何不必要的代码。使用全套测试来验证应用程序的其他部分没有因为所做的修改而受到负面影响。只要所有新旧测试都通过了，应用程序就可以发送给 QA 进行验证，并部署生产了。

第 13 章

有关优秀工具的争论

作者：Jeff McWherter

本章内容

- 单元测试框架
- 模拟框架
- 依赖项注入框架
- 其他有用工具
- 如何向团队介绍 TDD

和软件开发行为中的许多其他过程一样，TDD(测试驱动开发)及其所使用的工具也会在软件开发人员之间引发许多争论。这些争论经常像宗教战争一样，争论哪种是最佳工具。本章讨论了在 TDD 过程中非常有用的各种工具，并提供了有关这些工具的一些看法。本章将帮助您判断哪些工具最适合自己的。

13.1 测试运行程序

拥有一个自己使用顺手的测试运行程序对于执行 TDD 是非常关键的。许多测试框架(如前面各章大多数示例中使用的 NUnit)都带有一个用于运行测试的 GUI。我们已经了解到，单元测试需要快速运行，但如果有关于测试的反馈，并能采用熟悉的方式来运行这些

测试，则可以确保自己会运行这些测试。一般情况下，这些随单元测试框架提供的 GUI 的替代品可以让生活更轻松一些。

13.1.1 TestDriven.NET

TestDriven.Net 是一种很流行的测试运行程序，它能够运行在 NUnit、MbUnit 和 MSTest 等框架中创建的测试。TestDriven.Net 个人版可以供学生和开源开发人员免费使用。公司开发人员必须支付适当费用，但由于这样就不必再为这些测试框架打开 GUI 界面，所以也是值得的。

Visual Studio 中集成了 TestDriven.Net。右击测试，并选择 Run Test 命令，可以运行测试，如图 13-1 所示。

强烈建议创建一个键盘快捷方式，这样在希望运行测试时，就不需要每次都右击了。要在 Visual Studio 中设置键盘快捷方式，可以选择 Option | Keyboard | Settings 命令，打开如图 13-2 所示的对话框。

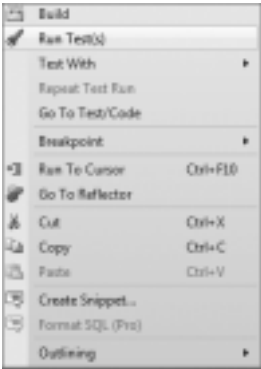


图 13-1

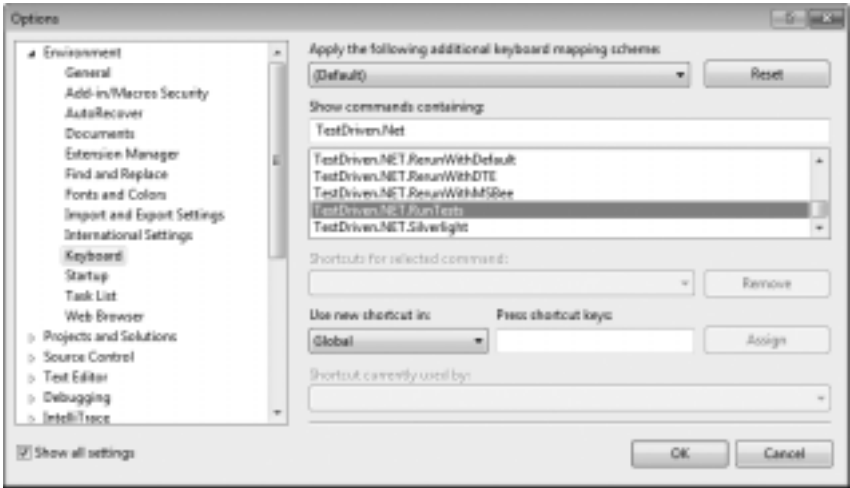


图 13-2

许多测试运行程序如 TestDriven.Net，还允许调试测试。如果试图在一个复杂测试中找出正在执行哪些内容，单步执行测试是很有用的。TestDriven.Net 的商业版本和个人版本可以从 <http://www.testdriven.net/download.aspx> 下载。

13.1.2 Developer Express 测试运行程序

Code Rush 来自 Developer Express，是一个功能强大的编码辅助工具，它最近的版本

中包含了一个很出色的测试运行程序。它与其他测试运行程序类似，但能在 Code Rush 中使用，为用户提供高度可扩展的工具。另外，它是开箱即用的，而且效果很棒。Code Rush 已经成为非常有名的 Visual Studio 加载项，可以帮助开发人员快速编写代码，并以一种非常优美的方式向开发人员提供建议。和其他测试运行程序一样，可以通过以下方式来执行测试：右击测试/测试组件，或者设置一个键盘快捷方式。Developer Express 测试运行程序不仅非常快速，而且还可以很容易地看出哪些测试通过、哪些测试失败，测试代码如图 13-3 所示。

```

1  @public class HTMLHelpersTest
2  {
3      [Test]
4      @public void Should_Truncate_Text_Over_Five_Characters_Long()
5      {
6          string textToTruncate = "This is my text";
7          string expected = "This ..";
8          string actual = HTMLHelper.Truncate(textToTruncate, 5);
9          Assert.AreEqual(expected, actual);
10     }
11
12     [Test]
13     @public void Should_Not_Truncate_Text_When_No_Length_Is_Passed_In()
14     {
15         string textToTruncate = "This is my text";
16         string expected = "This is my text";
17         string actual = HTMLHelper.Truncate(textToTruncate, 0);
18         Assert.AreEqual(expected, actual);
19     }
20
21     [Test]
22     @public void Should_Not_Append_Ellipses_when_Truncating_Text_That_Is_Shorter_Than_Length()
23     {
24         string textToTruncate = "This is my text";
25         string expected = "This is my text";
26         string actual = HTMLHelper.Truncate(textToTruncate, 50);
27         Assert.AreEqual(expected, actual);
28     }
29 }

```

图 13-3

从 http://www.devexpress.com/Products/Visual_Studio_Add-in/index.xml 可以找到有关 Code Rush 的更多信息。

13.1.3 Gallio

Gallio 是一种高度可扩展的自动化平台，它提供了一个公用的对象模型，供测试运行程序和运行时服务等工具进行互操作。简单来说，Gallio 是一种 GUI 测试运行程序，它运行使用 MbUnit、MbUnitCpp、MSTest、NBehave、NUnit、xUnit、csUnit 和 RSpec 创建的测试。Gallio 还为 MSBuild、NAnt、Pex、CCNet、Powershell、Resharper、TestDriven.Net、dotCover、TypeMock 和 Visual Studio 等提供了不同级别的支持与集成。在 <http://www.gallio.org> 上可以找到有关 Gallio 的更多信息。

图 13-4 显示了 Gallio Icarus GUI，它同一项目中运行用 MbUnit 和 NUnit 创建的测试。我曾经与一位开发人员合作，他有一个坏习惯：在一个项目中途切换测试框架。Gallio 对该项目是有好处的。

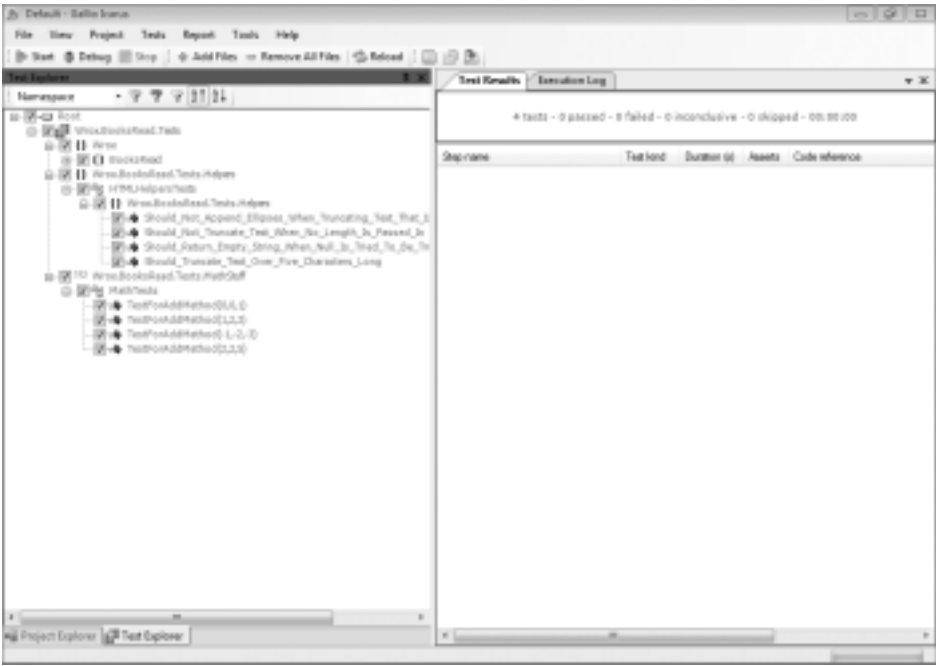


图 13-4

13.2 单元测试框架

单元测试框架就像是一双很好的工作靴。刚开始时你会讨厌它们，因为它们让你感到很不舒服，但越穿就越会爱上它们。等你有了一双新靴子时，或者像这里所说的新测试框架时，就会再次因为其不舒服而开始抱怨。单元测试框架更容易在不同的测试工具之间引发最具宗教意味的战争。

13.2.1 MSTest

MSTest 是由微软创建的测试框架，从 2005 年开始，它就已经包含在 Visual Studio 的指定版本中。MSTest 在开发人员之间激起了许多热烈的讨论。毕竟，MSTest 与所有其他单元测试框架都是类似的，只是语法稍有不同。我的体验是这些测试运行的速度要稍慢一些，但你应当自己尝试一下。

MSTest 的语法与 NUnit 非常相似，只是 Test Fixture 有一个 TestClass 特性，实际测试有一个 TestMethod 特性：



```
[TestClass]
public class HTMLHelpersTests
{
    [TestMethod]
    public void Should_Truncate_Text_Over_Five_Characters_Long()
    {
    }
```

```

string textToTruncate = "This is my text";
string expected = "This ..";
    string actual = HTMLHelper.Truncate(textToTruncate, 5);
    Assert.AreEqual(expected, actual);
}

[TestMethod]
public void Should_Not_Truncate_Text_When_No_Length_Is_Passed_In()
{
string textToTruncate = "This is my text";
string expected = "This is my text";
    string actual = HTMLHelper.Truncate(textToTruncate, 0);

    Assert.AreEqual(expected, actual);
}
}

```

MSTest.cs

如图 13-5 所示 ,MSTest 的运行方式与使用 Visual Studio 中的测试运行程序的方式类似。

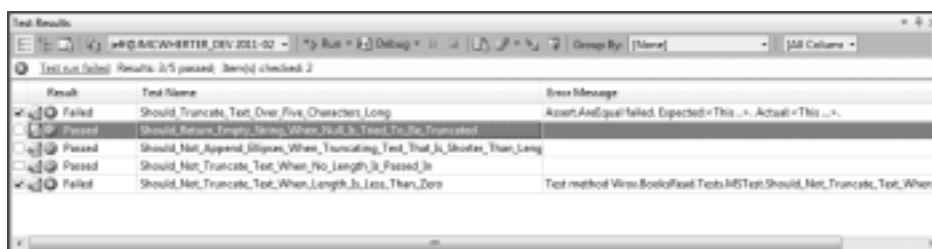


图 13-5

作为单元测试框架 ,MSTest 在我的推荐列表中排名很靠后 ,这主要是因为它的速度问题。但是没理由放弃所包含的其他出色测试工具 ,如数据库驱动测试、性能测试工具和有序测试。

13.2.2 MbUnit

2004 年 ,Jonathan“ Peli ”de Halleux 根据 Marc Clifton 的一系列论文创建了 MbUnit (当时称为 gUnit)。MbUnit 本身就支持一种被称为“行测试”的概念。可以创建一个测试和多个数据集 ,并根据方法属性中的定义执行测试。每一组数据都是独立执行的 ,被看为运行程序中的唯一测试。

下面的代码是一个针对 Add 方法的测试 ,Add 是一个静态方法 ,对两个整数求和。下面的测试有 3 个输入 : a 值、b 值和 expected 结果。利用 Row 属性可以为该测试创建输入。



```
[TestFixture]
public class MathStuffTests
{
    [RowTest]
    [Row(5, 3, 8)]
    [Row(0, 0, 0)]
    [Row(-1, -2, -3)]
    public void AddTests(int a, int b, int expected)
    {
        int test = MathStuff.Add(a, b);
        Assert.AreEqual(test, expected);
    }
}
```

MbUnit.cs

如图 13-6 所示，在报告测试结果时，MbUnit 测试运行程序将它看作 3 个独立测试。

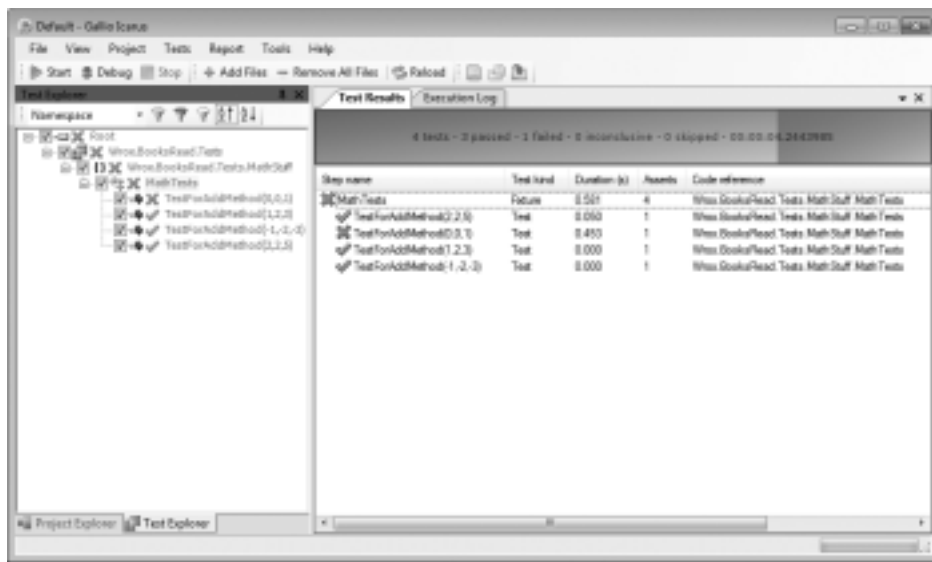


图 13-6

MbUnit 可以从 <http://www.mbunit.com/> 下载。

13.2.3 xUnit

xUnit 是一个开源测试框架，由微软的 Brad Wilson 在 Jim Newkirk 的协作下开发，后者是 NUnit 的最早作者之一。xUnit 采用一种最简单方法，通过方法属性来提供框架功能。这种最简单方法有助于使测试保持简单和整洁。xUnit 与前面讨论的其他测试框架之间的主要区别之一就是：没有 Setup 和 Teardown 方法特性。为了保持 xUnit 主题的简单性，可以在需要安装或分解功能时，使用测试类的构造函数和 dispose 方法。

根据在前面各章已经掌握的 NUnit 知识，下面的代码示例看起来应当很熟悉了。xUnit 测试有一个 Fact 特性。当需要确保一个方法在指定时间内返回时，[Fact(Timeout=20)]特性是很有用的。



```
[Fact]
public void AddTest()
{
    int expected = 10;
    int actual = MathStuff.Add(5, 5);
    Assert.Equal(expected, actual);
}

[Fact(Timeout = 20)]
public void TimeOutTest()
{
    System.Threading.Thread.Sleep(40);
    Assert.Equal(1, 1);
}
```

如需有关 xUnit 最简语法的更多信息，参阅 Code Plex 上 xUnit 项目页中 NUnit、MbUnit、MSTest 和 xUnit 的语法对照表，网址为：<http://xunit.codeplex.com/wikipage?title=Comparisons & ProjectName=xunit>。

13.3 模拟框架

第 2 章讨论了模拟对象的基础知识，在本书中，由于 Moq 框架的语法非常简单，所以一直在使用它。Moq 正在变成一种很流行的模拟框架，但并不是唯一可用的框架。其他框架有一些可用于模拟不同对象的功能。

Moq 使用声明性语法来模拟对象，这不同于许多模拟框架所支持的记录/回放模拟方法。许多此类框架正在放弃记录/回放方法，而转向一种易于使用的准备、执行、断言(AAA)方法。

13.3.1 Rhino Mocks

Rhino Mocks 是一种免费的模拟框架，它的创建者是 Orin Eini，他因在 nHibernate 和 Castle 项目中的工作而闻名。Rhino Mocks 因为简单易用、拥有功能强大的特性而流行。尽管它的语法不像 Moq 的语法那样整洁，但也是一种值得研究的工具。Rhino Mocks 支持两种不同风格的模拟对象：记录/回放方法和“准备、执行和断言”语法。“记录/回放”现在被看作模拟对象的旧式方法，但了解它们之间的区别还是有意义的。

下面的示例使用 AAA 风格的模拟对象，它类似于使用 Moq 进行的模拟，如第 2 章中所了解的内容。基本的 Rhino Mocks 语法类似于如下：



可从
wrox.com
下载源代码

```
MockRepository mocks = new MockRepository();
IDependency dependency = mocks.CreateMock < IDependency > ();

// create our expectations
Expect.Call(dependency.GetSomething("parameter")).Return("result");
dependency.DoSomething();
mocks.ReplayAll();

// test the middle layer
Thing thing = new Thing(dependency);
thing.DoWork();

// verify the expectations
mocks.VerifyAll();
```

RhinoMocks.cs

该例为 IDependency 接口创建了一个模拟对象，该接口最终将被传递给 Thing 类。对于该模拟对象设置了两点预期：调用 GetSomething 方法，它有一个字符串形参和一个返回值；取消对 DoSomething 方法的依赖关系。利用该模拟对象的 VerifyAll 方法来验证这两点预期。

可以对 Rhino Mocks 研究得再深入一点，创建一个对于项目有实际用处的测试：



可从
wrox.com
下载源代码

```
[Test]
public void Mocking_With_Rhino_Mocks()
{
    MockRepository mocks = new Rhino.Mocks.MockRepository();

    // create the repository object; the real object would make calls to the DB
    IItemTypeRepository repository = mocks.CreateMock < IItemTypeRepository > ();

    // mock the call to get an item with the ID of 2 and return null,
    // to mock not finding an item
    Rhino.Mocks.Expect.Call(repository.GetById(2)).Return(null);

    // get the mocking ready
    mocks.ReplayAll();

    // inject our mock into our service layer
    ItemPresenter presenter = new ItemPresenter(repository);

    // service.GetItem will call the mocked repository, which will call
    // repository.GetById, which will return null
    ItemType item = presenter.GetItem(2);

    // should be null
    Assert.IsNull(item);
}
```

RhinoMocks.cs

在某些情况下，可能希望在模拟对象中抛出异常，而不是返回一个值。创建模拟错误

的测试是一个好主意。下面的示例模拟了 `MethodThatThrowsError` 方法中的一个 `DivideByZeroException` 错误：



可从
wrox.com
下载源代码

```
itemRepository.Expect(m => m.MethodThatThrowsError("")).Throw(new
    DivideByZeroException("Error"));
```

RhinoMocks.cs

Rhino Mocks 中事件的用法也非常简单。首先，需要能够访问“事件引发程序”(Event Raiser) 然后模拟一个被引发的事件，如下面的示例所示。它模拟了 `MyEvent` 事件，为"result" 返回一个字符串值：



可从
wrox.com
下载源代码

```
itemRepository.GetEventRaiser(v => v.MyEvent += null).Raise("result");
```

RhinoMocks.cs

还可以使用 Rhino Mocks，以记录/回放语法来模拟对象。记录/回放过程分为两个阶段。在记录阶段，定义希望如何与模拟对象交互。在第二个阶段，即回放阶段，输入测试代码，并与 Mock 对象交互：



可从
wrox.com
下载源代码

```
MockRepository mocks = new MockRepository();
IItemTypeRepository mock = mocks.CreateMock<IItemTypeRepository>();

using (mocks.Record())
{
    Expect.Call(mock.GetById(3)).Return(null);
}

using (mocks.Playback())
{
    Assert.AreEqual(null, mock.GetById(3));
}
```

RhinoMocks.cs

Rhino Mocks 有一个庞大的追随群，也是一个值得考虑的出色模拟工具。要开始使用 Rhino Mocks，访问 <http://www.ayende.com/projects/rhino-mocks.aspx>。

13.3.2 Type Mock

Type Mock 不同于 Rhino Mocks 和 Moq，它使用中间语言(IL)在运行时以模拟实现来代替实际实现。Type Mock 可以在任意时间模拟系统中的任意对象，所以不需要担心如何将模型注入系统中。还有一点也很重要，就是要注意 Type Mock 是商业产品，因此不是免费的。

Type Mock 的另一个主要好处是：模拟对象不需要从接口继承。Type Mock 可以模拟对象的具体实现。利用 Type Mock，可以模拟不受自己控制的对象，如第三方库，甚至

是.NET 框架的组成部分。

下面的代码非常类似于 Rhino Mocks 语法：



```
// Arrange
IItemTypeRepository repository = Isolate.Fake.Instance < IItemTypeRepository > ();
Isolate.WhenCalled(() => repository.GetById(2)).WillReturn(null);

// Act
ItemPresenter presenter = new ItemPresenter(repository);
IItemType item = presenter.GetItem(2);

// Assert
Assert.IsNull(item);
```

TypeMock.cs

当没有接口而需要模拟 `DateTime.Now` 等东西时,TypeMock 的真正威力就发挥出来了。考虑以下代码。它有一个名为 `IsExpired` 的方法,用于查找一个常数值 11/21/1981,并将它与 `DateTime.Now` 对比,看后者是否过期：



```
public class Item
{
    private DateTime EXPIRATION_DATE = new DateTime(1981, 11, 21);

    public bool IsExpired()
    {
        bool tmpRtn = false;

        if (DateTime.Now > EXPIRATION_DATE)
            tmpRtn = true;

        return tmpRtn;
    }
}
```

TypeMock.cs

可以编写逻辑,将对 `DateTime.Now` 的调用从该方法中抽象出去,然后向 `IsExpired` 方法中传递一个日期。但如果使用 TypeMock,就可以模拟 `DateTime.Now` 并进行测试,以确保 `IsExpired` 方法正在检查过期日期。

在下面的 Isolated 测试中,当调用 `DateTime.Now` 时,返回一个新的日期。在第一个测试中,它是一个过期日期,而第二个测试返回一个未过期的日期：



```
[Test, Isolated]
public void Item_Should_Be_Expired()
{
    Isolate.WhenCalled(() => DateTime.Now).WillReturn(new DateTime(1981,
        11, 22));
    Item item = new Item();
    Assert.True(item.IsExpired());
}
```



```

    }

    [Test, Isolated]
    public void Item_Should_Not_Be_Expired()
    {
        Isolate.WhenCalled(() => DateTime.Now).WillReturn(new DateTime(1981,
            11, 21));
        Item item = new Item();
        Assert.IsFalse(item.IsExpired());
    }

```

TypeMock.cs

由于能够模拟任何对象，所以 Type Mock 非常适用于旧应用程序，对于旧应用程序，测试都是一些事后补救措施。有关 Type Mock 隔离框架的信息可以在 <http://www.typemock.com/> 找到。

13.4 依赖项注入框架

如果把自己的应用程序看作拼图，依赖项注入(DI)框架就是把这些拼图块整合在一起的工具。有人甚至可能会说，依赖项注入框架就是超级工厂设计模式的实现。第 5 章讨论了依赖项注入的基础知识。这些示例使用 Ninject，但 Ninject 并不是市场上唯一的依赖项注入框架。

过去，大多数依赖项注入框架是使用 XML 文件配置的，而 XML 文件很快就会变得难以维护。大多数现代依赖项注入框架仍然支持这种配置方法，但人们一般都不太赞同该方法，开始向更干净利落的方法转换。这一部分研究了一些替代的依赖项注入框架，使用第 5 章的示例向业务应用程序中注入依赖项。

在选择依赖项注入框架时，应当遵循两条规则：

- 使框架保持一定距离。
- 重点关注“控制反转”(Inversion of Control)模式。

13.4.1 Structure Map

Structure Map 是一种开源容器框架，它有一个使用非常便利的 API，使代码易于阅读和维护。Structure Map 的强大功能之一就是能够自动模拟容器。

利用 Structure Map 自动模拟功能，可以在测试需要时自动创建存根。尽管仍然需要对模拟对象设置预期，但仍然可以节省在自己测试中创建模拟的时间。

下面的示例重新创建了第 5 章的示例，它当时使用 Ninject 向一个业务应用程序类中注入依赖项。首先，需要将接口映射到其各自的具体实现。在 Structure Map 和大多数 DI 框架中，这一步骤只需要执行一次，也就是在应用程序启动时执行。这一过程的成本可能非

常昂贵,具体取决于所映射的对象数目。在 Web 应用程序中,这一任务通常是在 Global.asax 文件中完成的,但对于这些简短的示例来说,将创建一个启动类,在设置映射时将会调用它:



```
public static class StructureMap_IocBootStrapper
{
    public static void SetupForIoC()
    {
        // the setup. If we were working with ASP.net this would occur in
        // Global.asax
        ObjectFactory.Initialize(x =>
        {
            x.ForRequestedType <ILoggingDataSink> ()
                .TheDefaultIsConcreteType <LoggingDataSink> ();

            x.ForRequestedType <ILoggingComponent> ()
                .TheDefaultIsConcreteType <LoggingComponent> ();

            x.ForRequestedType <IDataAccessComponent> ()
                .TheDefaultIsConcreteType <DataAccessComponent> ();

            x.ForRequestedType <IWebServiceProxy> ()
                .TheDefaultIsConcreteType <WebServiceProxyComponentProvider> ();
            x.ForRequestedType <IPersonRepository> ()
                .TheDefaultIsConcreteType <PersonRepository> ();

            x.ForRequestedType <IPersonService> ()
                .TheDefaultIsConcreteType <PersonService> ();
        });
    }
}
```

StructureMap.cs

在该示例中做的第一件事情就是创建对象工厂的一个实例,并调用 initialize。initialize 方法获取一个表达式,用来配置 Structure Map。添加这一逻辑后,就可以向业务应用程序中注入依赖项了。



```
public class StructureMap_ThingThatImplementsABusinessService
{
    public StructureMap_ThingThatImplementsABusinessService()
    {
        StructureMap_IocBootStraper.SetupForIoC();
        // getting our objects to work with. This usually occurs in the
        // constructor of the class you need the objects for
        var logger = ObjectFactory.GetInstance <ILoggingComponent> ();
        var personRepository = ObjectFactory.GetInstance <IPersonRepository> ();
        // start working with our objects that have been loaded
        var person = personRepository.GetPerson(3);
    }
}
```

```
    }
}
```

StructureMap.cs

上面的示例重新创建了第 5 章使用 Ninject 的示例。为简单起见,它没有像第 5 章的示例那样映射所有依赖项。对于大型项目来说,映射所有依赖项可能会非常麻烦。Structure Map 的另一个出色功能是自动注册或自动扫描功能。Structure Map 查看接口,并尝试根据默认约定将它们匹配到具体实现。例如,如果我们有一个 IfooBar 接口,它很可能就是与 FooBar 类相匹配。如果默认映射不正确,就可以创建一个配置文件来设置正确的映射。



```
public static void SetupForIoC_Scan()
{
    // the setup. If we were working with ASP.net this would occur in
    // Global.asax
    ObjectFactory.Initialize(x =>
    {
        x.Scan(s =>
        {
            s.TheCallingAssembly();
            s.WithDefaultConventions();
        });
    });
}
```

StructureMap.cs

这个开源 Structure Map 项目可以在 <http://structuremap.net/structuremap/> 找到。

13.4.2 Unity

来自微软“模式与实践”(Patterns and Practices)群组的 Unity 是本章讨论的最新 DI 框架。Unity 不支持其他 DI 框架所支持的许多高级功能,但它支持的功能足以完成工作。

为了实现这个现在应当已经非常熟悉的示例,需要添加对 Microsoft.Practices.Unity 程序集和 Microsoft.Practices.ObjectBuilder2 程序集的引用。之后,在引导类中创建一个 UnityContainer 对象,然后启动映射,代码如下:



```
public static class Unity_IoCBootStraper
{
    public static UnityContainer BaseContainer = new UnityContainer();

    public static void SetupForIoC()
    {
        // the setup. If we were working with ASP.net this would occur in
        // Global.asax

        BaseContainer.RegisterType<ILoggingDataSink, LoggingDataSink>();
    }
}
```

```

BaseContainer.RegisterType <ILoggingComponent, LoggingComponent> ();
BaseContainer.RegisterType <IDataAccessComponent, DataAccessComponent
> ();

BaseContainer.RegisterType <IWebServiceProxy,
WebServiceProxyComponentProvider> ();
BaseContainer.RegisterType <IPersonRepository, PersonRepository> ();
BaseContainer.RegisterType <IPersonService, PersonService> ();
}
}

```

Unity.cs

要在业务对象中获得依赖项，只需要调用 UnityContainer 对象的 resolve 方法：



```

public class Unity_BusinessApplication
{
    public Unity_BusinessApplication()
    {
        Unity_IoCBootStraper.SetupForIoC();

        // getting our objects to work with. This usually occurs in the
        // constructor of the class you need the objects for
        var logger = Unity_IoCBootStraper.BaseContainer
            .Resolve <ILoggingComponent> ();
        var personRepository = Unity_IoCBootStraper.BaseContainer
            .Resolve <IPersonRepository> ();
        // start working with our objects that have been loaded
        var person = personRepository.GetPerson(3);
    }
}

```

Unity.cs

默认情况下，在解析 UnityContainer 对象中的一个对象时，将根据默认映射获得该对象的一个新实例。Unity 的出色功能之一是可以改变这一功能，并在需要时返回该对象的一个单态。RegisterType 功能有一个取得 LifeTimeManager 对象的重载。Unity 具有一个“容器控制生命周期管理器” (Container Controller Lifetime Manager)，它实际上是一个单态：

```

BaseContainer.RegisterType < ILoggingComponent, LoggingComponent > (new
ContainerControlledLifetimeManager());

```

Unity 是开源的，可以在 CodePlex 找到，地址为：<http://unity.codeplex.com/>。由于 Unity 拥有非常丰富的文档和网络信息，所以对于新接触 Inversion of Control(控制反转)模式的人来说，Unity 是个不错的起点。

13.4.3 Windsor

Windsor 由 Castle 项目维护，是最早出现的针对 .NET 的开源依赖项注入框架之一。在

这里讨论的所有依赖项注入框架中，Windsor 拥有最多的拥护者，为依赖项注入提供了最成熟、最强大的实现。由于这个项目拥有如此之多的功能，因此为了使用某些最高级的功能，其学习曲线可能会比较高。以下代码实现了一个示例。可以看出，它与其他依赖项注入框架非常相似：



```
public static class Windsor_IoCBootStraper
{
    public static WindsorContainer BaseContainer = new WindsorContainer();

    public static void SetupForIoC()
    {
        // the setup. If we were working with ASP.net this would occur in
        // Global.asax.
        BaseContainer.AddComponent <ILoggingDataSink, LoggingDataSink> ();
        BaseContainer.AddComponent <ILoggingComponent, LoggingComponent> ();
        BaseContainer.AddComponent <IDataAccessComponent, DataAccessComponent> ();
        BaseContainer.AddComponent <IWebServiceProxy,
            WebServiceProxyComponentProvider> ();
        BaseContainer.AddComponent <IPersonRepository, PersonRepository> ();
        BaseContainer.AddComponent <IPersonService, PersonService> ();
    }
}

public class Windsor_BusinessApplication
{
    public Windsor_BusinessApplication()
    {
        Windsor_IoCBootStraper.SetupForIoC();

        // getting our objects to work with. This usually occurs in the
        // constructor of the class you need the objects for
        var logger = Unity_IoCBootStraper.BaseContainer
            .Resolve <ILoggingComponent> ();

        var personRepository = Unity_IoCBootStraper.BaseContainer
            .Resolve <IPersonRepository> ();

        // start working with our objects that have been loaded
        var person = personRepository.GetPerson(3);
    }
}
```

Windsor.cs

Castle Windsor 可以从 <http://stw.castleproject.org/Windsor.MainPage.ashx> 下载。

13.4.4 Autofac

有些依赖项注入框架包含一个界面，允许在不采用 XML 文件的情况下进行配置，Autofac 是最早的此类框架之一。因此，许多开发人员开始使用这种依赖项注入框架。在使

用 Autofac 框架时,可以使用反射、lambda 表达式或预先制作的实例来创建组件。Container-Builder 对象提供了 Register 功能支持对象注册全部所需的方法,代码如下:



```
public static class Autofac_IocBootStraper
{
    public static IContainer BaseContainer { get; private set; }

    public static void SetupForIoC()
    {
        var builder = new ContainerBuilder();
        builder.RegisterType < IPersonRepository > ().As <PersonRepository> ();

        BaseContainer = builder.Build();
    }

    public static TService Resolve <TService> ()
    {
        return BaseContainer.Resolve <TService> ();
    }
}
```

AutoFac.cs

要将依赖项注入到业务应用程序中,只需要调用引导类中的 Resolve 方法:



```
public class Autofac_BusinessApplication
{
    public Autofac_BusinessApplication()
    {
        Autofac_IocBootStraper.SetupForIoC();

        // getting our objects to work with. This usually occurs in the
        // constructor of the class you need the objects for
        var logger = Autofac_IocBootStraper.Resolve <ILoggingComponent> ();
        var personRepository = Autofac_IocBootStraper
            .Resolve <IPersonRepository> ();
        // start working with our objects that have been loaded
        var person = personRepository.GetPerson(3);
    }
}
```

在大多数情况下,依赖项注入框架之间的性能差别不会影响到对框架的选择,但要注意,AutoFac 是这里所列框架中速度最快的。Autofac 框架可以在 Google 代码中找到,地址为: <http://code.google.com/p/autofac/>。

13.5 其他有用工具

一些工具应当属于“其他”类别。在执行 TDD 时并不一定需要它们,但对于应用程序的测试十分有用。

13.5.1 nCover

nCover 是一种代码覆盖率测试工具。它对测试和代码进行分析,并报告这些测试所覆盖代码的百分比。nCover 有两个版本。一个是开源版本,另一个是包含更多功能的商业产品。

图 13-7 显示了针对 Wrox.BooksRead.Web 项目的测试覆盖率。可以看到 HTMLHelper 类的测试覆盖率为 100%,而其他项目的测试覆盖率为 0%。在该示例中,所有测试都是为 HTMLHelper 创建的,所以这个结果是有意义的。

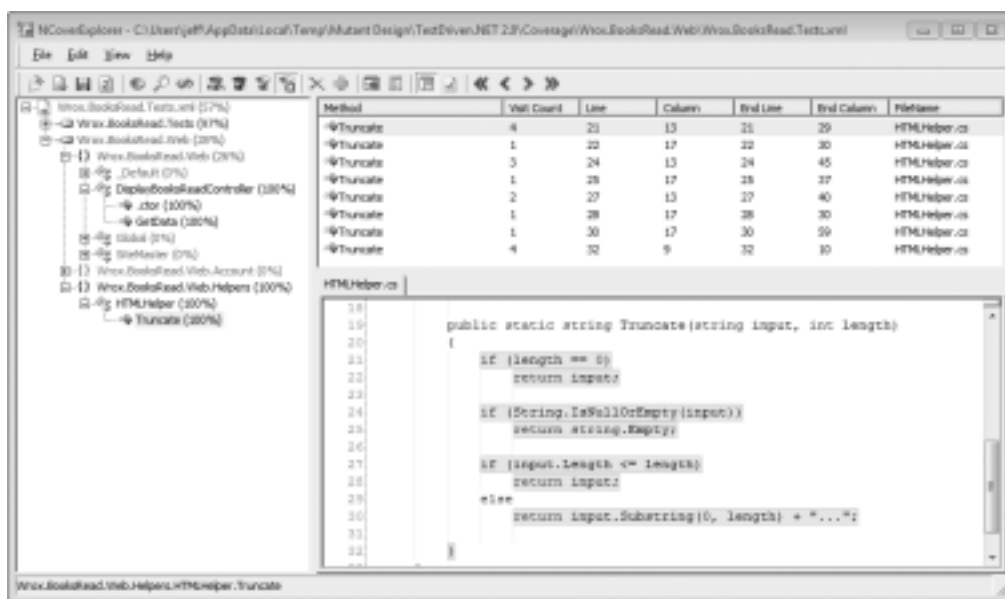


图 13-7

nCover 可以从 <http://www.ncover.com/> 下载。

13.5.2 PEX

PEX 是 Program Exploration 的简称,它是来自微软研究院(Microsoft Research)的一个项目,能够从现有代码生成单元测试。PEX 是 Visual Studio 加载项,只有很少量的配置,可以生成一套参数化单元测试,而且代码覆盖率很高。

PEX 不仅生成测试,还会针对如何应对失败测试提供建议。PEX 对于运行遗留代码或者查找那些未被正确处理的边缘情况是非常出色的。PEX 不能包治百病,在使用时也要多加小心。TDD 不仅涉及测试,还与设计有关。而 PEX 则只能查找错误。图 13-8 显示,PEX 为 Truncate HTML 函数创建了 5 个单元测试,其中的一个失败了。

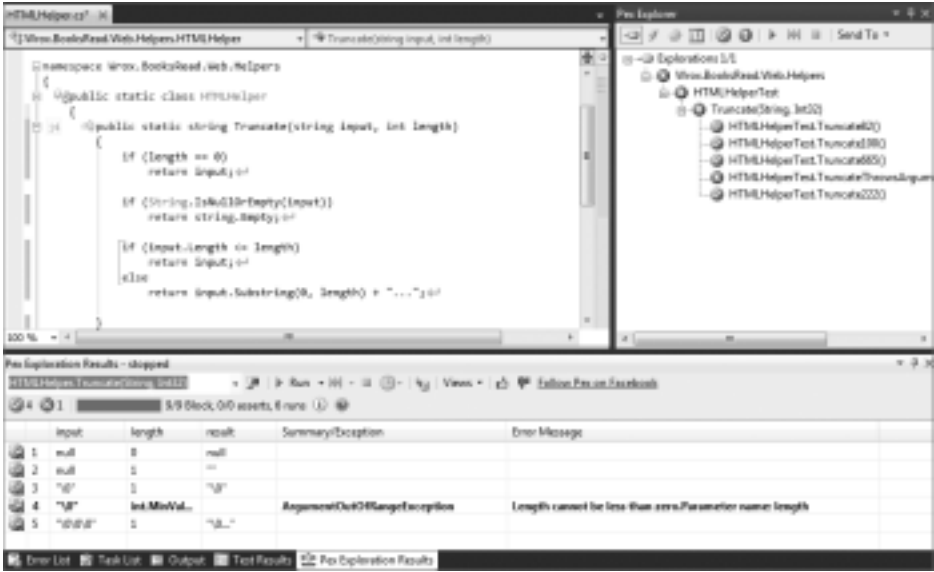


图 13-8

测试 4 之所以失败，是因为向它传递了一个负数。这是 PEX 的好处：它捕获了原始单元测试中漏掉的边缘情况。我们应当返回去，在 HTMLHelpersTests 类中添加一个测试，并添加以下测试来测试这种边缘情况：

```
[Test]
public void Should_Not_Truncate_Text_When_Length_Is_Less_Than_Zero()
{
    string textToTruncate = "This is my text";
    string expected = "This is my text";
    string actual = HTMLHelper.Truncate(textToTruncate, int.MinValue);
    Assert.AreEqual(expected, actual);
}
```

可以从以下地址了解有关 PEX 的更多内容：<http://research.microsoft.com/en-us/projects/pex/>。

13.6 如何向团队介绍 TDD

你可能感到奇怪：向团队介绍 TDD 的章节与工具有什么关系，但能够改变他人核心思考方式的能力是一件极为有用的工具。我们经常会被问到这样一个问题，而且无数次为之争论，所以干脆综述如下：如果你自己希望通过学习新过程来提升自己，那应当只与这类人合作；反之亦然。如果你的工作单位对质量与人员的重视高于过程，介绍 TDD 就很容易了。

13.6.1 在拒绝改变的环境中工作

当你结束一天的工作要离开时，签入版本控制的代码就是你在项目上留下的标记。如果这些代码不能正常工作，你要对其负责。TDD 有一个初始学习曲线，但一旦掌握了它，就能像过去一样快速创建代码。即使一个团队不乐意为遵循 TDD 实务而进行改变，你自己是可以改变的。与某个采用 TDD 方法的人员搭档，自己来学习这种方法。熟练之后，就可以慢慢地开始在项目实现所学到的 TDD 知识。建议不要回过头去，重构已有数据库中的所有内容；只需要慢慢开始，并记录下量度数字。管理者都喜欢图表，当你向他们展示：最近一个版本的缺陷数比上一个版本减少了 20%，你就可以告诉他们，在你负责的项目部分实施了 TDD。

这时，管理者要么对你的所作所为大动肝火，要么因为系统减少了 20% 的缺陷而感到兴奋。如果管理者感到不安，你也许应当换一个能够体现自己思想价值的位置了。在 Andrew Hunt 和 David Thomas 所著 *The Pragmatic Programmer* 一书(Addison-Wesley, 1999, ISBN: 9780201616224)第 1 章中，讲述了这种向团队介绍新概念的方法。它与两个关于“石头汤”和“煮熟了的青蛙”的故事有关。在后面一个例子中，如果把一只青蛙扔到一壶沸水里，它会马上跳出来。但是如果把青蛙放到冷水中，然后慢慢地把水加热到沸腾，青蛙就会慢慢地适应不断升高的温度，一直待在水里，直到死去。

13.6.2 在接受改变的环境中工作

有些公司允许员工成长和学习新东西。如果在这样一个地方工作，那向团队讲授 TDD 的最佳方法就是介绍一位导师。搭档编程策略是加快开发人员速度的最佳方式。本书的作者就是通过与其他人员的搭档来学习 TDD 的。这是学习这一过程的最佳方式。如果能有一位对 TDD 有深刻理解的导师，就可以帮助团队中的后进成员加快速度，超过那些通过书本来学习 TDD 的人员。

如果您的公司不希望其他人进入本公司，则“代码精研(code retreat)”这样的活动也是与其他人讨论的好机会。有关“代码精研”的更多信息，参见 <http://coderetreat.ning.com/>。

13.7 本章小结

本章介绍了许多对 TDD 过程有帮助的工具。研究新工具时，建立拿出一个简单示例，用各种工具来重现它，然后得出自己的结论。

NUnit 出现已经有许多年了，拥有很多拥护者，但这并不必然使它成为完成任务的最佳工具。xUnit 是目前的新测试框架，在许多方面都与 NUnit 都有明显不同。那些有着明显不同的工具正是应当尝试的工具，因为可能正是它们能让你的生活变得更轻松。在思考时跳出条条框框，尝试新的事物，会在许多方面都有所裨益。

软件开发人员在自己的职业生涯中不断前行。应当能够回顾一下过去编写的代码，并弄清楚：“我当时是怎么想的？”通过这种方式，就能更好地做好自己的工作。研究新工具只是提高自己的一种方式。

在选择使用哪些工具时，一定要选择自己最熟悉的工具。有些业务情景中是不允许使用开源工具的。很多时候，不允许使用这些开源工具是因为出于各种原因而对开源产生的忧虑。例如，许可问题或者“我不能通过拨打电话向某人寻求支持”等借口都是很常见的。要学习如何找出充分的理由来说明这些工具如何提高生产效率。当最终归结为选择一种工具时，从来没有人因为使用微软工具而被解雇。

第 14 章

结 论

本章内容

- 复习本书介绍的概念和技术
- 理解使用 TDD 的最佳实践
- 复习用 TDD 进行开发的好处
- 如何向团队介绍 TDD

本书已经提供了很多信息，这些信息为成为 TDD 开发人员奠定了坚实的基础。前面已经学习了实际运行 TDD 所需要的原则和技术。可以将其中许多技术运用于日常开发任务，以确保提交精心编写的高质量软件。

除了 TDD 的原则和技术之外，还学习了许多用于支撑应用程序 TDD(测试驱动开发)的技术、思想和原则。我们已经学习了 SOLID 原则，为开发出精心编写的可维护应用程序提供了一套规则。还看到了敏捷式开发方法如何为开发团队提供必要的时间、空间和信息，以便使用 TDD 方法成功地开发应用程序。

本书包含大量信息。但即使你把本书阅读 100 次，但让这些原则和技术变为现实的唯一途径还是在实际工作中运用。TDD 就像是学习一门语言：如果不能每天都练习和使用它，就无法把自己的技能发挥至极限。如果准备把自己对 TDD 的实践推向深入，要记住这里给出的临别建议。

14.1 已经学到的内容

除了给出一些框架和模式(如 Fluent NHibernate 和库)的相关指南外,本书主要是为你奠定 TDD 原则中的坚实基础。我们已经学习了一些成功实践 TDD 所需要的模式和技术,如依赖项注入。还看到如何遵守 SOLID 原则,敏捷开发如何帮助 TDD 开发人员完成自己的任务。本章将给出最后一些有助于应用 TDD 的指南。

14.1.1 你是自己代码的客户

作为开发人员,在构造自己的软件时就会使用服务和组件。这些部分结合起来构成一个整体,这个整体要大于其组成部分相加之和。通常,这些服务和组件已由你或你开发团队中的一位成员编写完毕。这意味着你所开发的这些服务和组件主要使用者是你和你的开发团队。这些服务和组件应当易于理解、便于使用。开发团队的新成员应当能够快速、轻松地根据这些服务的组件名称和方法签名判断其功能和用法。

在 TDD 中,对于自己创建的任何服务或组件,你都是其第一位客户。在为自己创建的服务或组件编写测试时,你就成为第一个使用此服务或组件的人。该服务或组件是否简单易用呢?命名和方法签名能否清楚地表明服务是如何工作的?界面是否整洁,有没有容易混淆的名称或重复功能?通过使用 TDD 来开发这些服务和组件,使自己成为它们的第一位用户。你是否正在创建一个自己认为很容易使用的服务或组件?通过在 TDD 实践中生成服务和组件接口,可以创建出一些精致、直观且简单易用的接口和类。

14.1.2 逐步找出解决方案

开发软件应用程序是一项大任务。将任何应用程序的开发看作一个大步骤,会徒劳地将你和你的团队引向失败。在软件开发中有一种广为人知的说法:不要尝试把整个海洋煮沸。它的意思是说,一项大型任务看起来可能非常复杂,不可能完成,但如果将它分解为几个小任务,这项工作可能马上会变得没有那么困难了。软件可能非常复杂。不要试图一次性解决软件应用程序开发中所包含的全部问题,而是将它作为几个小问题来处理。采用一种可管理方式,每次解决这些小问题中的一个。不要把一项功能或用户情景看作一个需要 40 小时完成的任务,把它看作 10 个需要 4 小时完成的任务可能会更容易。甚至还可以更好一点,把它分解为 20 个需要 2 小时完成的任务。

处理小型任务有许多好处。小型任务更容易理解。不要处理一个包含许多变化部分的庞大问题,而是一次只处理一个特定的关注领域。小型任务更容易实现。不需要等到一周之后才能知道自己能否解决手边的问题,而是在几小时内就能知道。这样就使开发人员能够拥有完成任务的动力和信心。

14.1.3 用调试器作为手术器械

没有使用 TDD 的开发人员倾向于把调试器作为主要开发工具。对于大多数没有采用 TDD 的开发人员，验证代码的第一步通常是在调试器中执行代码，查看它做些什么。对于这些开发人员来说，调试器是一件钝器：一个必须用来将自己的代码锤打成形的棍子。这种方法是很浪费的。

在调试器中验证代码的过程非常缓慢。开发人员通常必须在要查看的代码之前数行或数个方法处设置断点，然后逐步执行所关心的代码。由于开发人员需要单步执行代码，并人工查看和计算数据，所以这一过程很慢。对于一位开发人员来说，要在调试器中单步执行一个功能单位，并验证功能是否正确，其花费的时间完全可以运行所有功能的测试，并马上知道它们是否能够正常工作。使用调试器还会得出不一致的结果。大多数情况下，应用程序需要某种形式的用户交互来获得需要验证的功能，这就为错误的出现留出了空间。我们非常可能输入错误值或者按下错误按钮。

而使用 TDD 的开发人员倾向于比较保守地使用调试器。测试可以验证代码能否正确工作；不需要在调试器看着代码逐行执行。如果精心编写了测试，而且代码非常简单，就不会需要太频繁地使用调试器。

在使用 TDD 时，调试器使用的针对性更强、应用面更窄。因为测试隔离了每个功能单元，所以知道在代码中查找缺陷的确切位置。我们可以在紧临缺陷的位置设置断点，直接进入代码中的问题区域。到达问题区域时，我们知道自己正在寻找什么，因为已经有一个失败测试描述了当前错误。在查找到问题的原因之后，就可以编写一个测试，用于确保这一问题不再发生。这种测试是可重复的，速度也很快，也就是说可以继续验证这一代码能够正确工作，而不需要再次在调试器中运行代码。

14.2 TDD 最佳实践

最后，TDD 实践应当反映我们的需要以及开发团队和应用程序的需要。在使用 TDD 时，有很大的空间可用来定制自己团队的工作途径与方法。大多数成功的开发团队都遵循类似的策略。在日后的 TDD 实践中，应当记住这些指导规则。

14.2.1 使用有意义的名字

名字非常重要。好的名字可以很轻松地标识类、方法或变量的目的和功能。相反，如果名字不合适，可能无法标识这些对象的意图。在为测试命名时，一定要明确指出将类和测试方法命名为什么。类和方法的名称应当清楚地指出测试的前提条件或假设、正在测试的操作或功能，以及希望在成功测试中看到什么结果。保持名称的描述性是保持代码可读性和可维护性的关键要素。

14.2.2 为一个功能单元至少编写一个测试

TDD 中的第一个 D 表示“驱动”。也就是说在编写代码向应用程序中添加功能之前，应当编写一个测试。作为一种必要条件，这意味着每个功能单元至少要有一个测试。对于许多类型的功能来说，一个测试是不够的。那些仅测试代码中“幸福路径”的代码是很容易编写的。但遗憾的是，这些测试通常不能描述完整的故事。

重要的是：不仅要针对输入形参中的可接受范围来测试代码，还要考虑那些输入值超出可接受范围的情况。第一步是从公司找出所关注方法的可接受输入范围。显然，应当测试绝对属于这一范围的取值。还应当测试这一范围的边界情况。如果传递一个输入形参的最小值或最大值，该方法的工作状况是否与输入介于最大值和最小值之间的形参值是相同呢？应当有一个测试来验证这一情况。如果形参值落到定义的边界之外，又会怎么样呢？记住，所有输入都是魔鬼。应当测试那些可接受范围之外的值，以确保代码能够恰当地处理这些情况。

14.2.3 保持模拟的简单性

一个严格的模拟就是一组规则或预期，主要涉及对其方法的调用顺序和调用次数(以及用什么样的实参来调用)。在本书中没有花费太多时间来讨论它们。这是因为通常我不太喜欢使用严格的模拟。在一个单元测试中如果需要严格的模拟，说明被模拟的服务或组件可能有不必要的复杂接口。使用和模拟该服务或组件所需要的知识，会封装服务或组件的内部功能以使它与其他类隔离的界限变得模糊。之所以要抽象所讨论的功能，其原因就是为了使其使用变得更容易。如果为了模拟一个要使用的服务或组件而必须遵守一长串规则，就会使该服务或组件的模拟变得很困难，还会使测试变得非常脆弱，因为这些规则可能会变化，从而需要在测试中做出相应改变。

在某些情况下，严格模拟是必要的。.NET 框架或其他工具包中的特定服务和组件可能会拥有极为复杂的接口。在这些情况下，框架或工具包 API 不愿发生巨大变化(特别在永远不会对其进行升级的情况下)，所以这些测试就不会像严格模拟自己的组件或服务时那样脆弱。最后，严格模拟总比根本没模拟要好。

一般来说，应当使模拟保持简单。不要进行不必要的模拟。一个接口可能有许多方法，但我们只关注其中的一个。只模拟关心的方法，让所有其他方法都保持未模拟状态。这样就能使测试更容易理解，确保不会编写多余的代码，只需要保证测试通过就够了。

14.3 TDD 的好处

TDD 的主要好处就是拥有一套可以随时调用的测试，用来验证应用程序的功能是正确的。这本身就是一个巨大的好处。它确保了应用程序总能像描述的那样正常工作。只要能够通过这组测试，就能确保在开发过程中不会破坏已有功能。TDD 的其他间接好处也值得

注意：

- **设计更佳**——要真正发挥 TDD 的优势，应用程序应当广泛采用 DI 模式。DI 的使用有助于放松应用程序中的耦合。没有服务和组件被静态绑定到另一服务或组件。这样就极大地提高了应用程序的灵活性，使其能够开放地面对修改和扩展。
TDD 的大多数使用者都采用 SOLID 原则。无论是从设计上要采用这些原则，还是因为要编写可测试代码而产生的副作用，其结果都是一样的。所编写的代码更容易理解、更便于维护，而且极为灵活。这种代码能够向客户提供更高的品质。
- **缺陷更少**——通过使用 TDD，可以编写出能够正常工作的业务代码，而不是缺陷。因为有一套测试能够精确地反映业务需求，而且这些测试能够通过，所以就会相信代码中不会或很少包含因为未能正确实现业务需求而导致的缺陷。但仍然可能会因为需求不完整或本身存在问题而导致缺陷。要处理这些缺陷，首先编写一个测试来标识它们，然后再进行修复。这个测试的存在可以确保这个缺陷不会再次出现。
代码的质量要视测试而定。如果这些测试没有准确地反映应用程序的业务需求，就不能证明代码具有很高的品质。要确保对代码的质量要求和对测试的质量要求同等看待，这一点很重要。
- **宽松的团队**——大多数开发人员不喜欢使用不稳定或难以理解的代码基。使用 TDD 开发应用程序时，可以在每一步骤都使用一套测试来验证代码基的正确功能。通过 TDD 创建的应用程序，其设计通常要更好一些，使开发人员能够快速地提高代码基的速度和效率。例如，用 TDD 开发的应用程序倾向于采用针对性更强、更容易使用的小型类。所有这些都会减少缺陷，这也是开发人员所喜欢的。而这又会使整个团队更开心、更宽松，生产效率也更高。

14.4 如何向团队介绍 TDD

一些开发人员可能会对 TDD 感到有些畏惧。特别是对那些没有花费必要的时间来离职学习新技术的开发人员，这一点尤为突出。TDD 方法与传统的开发软件确实有很大不同。一些开发人员可能难以理解和应对这种比较剧烈的变化。一些策略可以帮助开发团队的成员来理解和采用 TDD。向他们介绍本书就是一个很好的开始。此外，还可以使用一些策略让队友们对 TDD 发生兴趣。

在全新项目中引入 TDD 是最容易的。在计划一个项目或开发工作刚刚开始时，围绕依赖项注入(DI)来设计应用程序比较容易。如果刚刚开始开发一个新项目，就从向团队介绍依赖项注入开始吧。采用依赖项注入是一大步，对于不熟悉 TDD 的团队来说，依赖项注入可以很好地向团队介绍他们将在后续 TDD 实践中学习的原则。只要团队理解并习惯了使用依赖项注入，就可以介绍编写自动化单元测试的理念了。在团队拥有编写单元测试所需要的技能之后，再介绍测试的概念。步子不要太大，不要让团队一下子接受过多内

容。在每个步骤都要等一下，使团队有时间习惯新的实践、模式或技术。这个过程没有设定的时间表；你会知道他们在什么时候可以开始下一步骤。

大多数开发人员面对的是已经开发过一段时间的项目或处于维护模式的项目。这些项目是最难向其中引入 TDD 的。这些应用程序大多没有采用依赖项注入，从而难以进行真正的单元测试。不提倡为了引入依赖项注入而将一个系统的所有功能开发或维护暂停很长时间。这种做法的成本收益率很低。更好的做法是在继续编写或维护应用程序时开始引入 TDD 的概念。在能够添加且不会破坏应用程序和危及其稳定性的地方添加依赖项注入。编写新功能或修正缺陷时，围绕新工作编写测试。最初，编写集成测试要比编写单元测试更轻松一些。这样也可以，集成测试总比根本没有测试好。Michael C. Feathers 撰写的 *Working Effectively with Legacy Code* (Prentice Hall, 2004, ISBN: 9780131177055) 提供了许多策略和技术，用于在那些创建时未考虑可测试性的应用程序中进行测试。

为 TDD 采用“管理层换购”通常是一项很困难的任务。TDD 意味着一个很长的学习曲线，尤其是对那些正在学习它的开发人员或开发团队，可能使最初的开发工作变得非常缓慢。管理层的任务是保证按时交付软件，并且不能超出预算。我们必须学会用他们的语言向他们介绍 TDD。好的策略是向他们证明：尽管前面的开发过程花费的时间要长一些，但在应用程序到达 QA 时会发现缺陷数目大幅减少，在部署应用程序之后，会得到更高的客户满意度，这些优点都足以弥补在刚开始时的缓慢进程。管理层喜欢图表和数字。将这一信息进行量化，并展示实际节省的数字。互联网上提供的许多研究内容都可以支持我们的观点。要善于利用这些研究内容。向管理层证实 TDD 提供的价值要高于未采用 TDD 的开发，必然可以引起他们的注意。

14.5 本章小结

本书已经讲解了 TDD 的原则和技术。但是，任何一本书中都无法包含有关 TDD 的全部内容。TDD 是旅程，而不是目的地。本书可以帮你做一些准备工作，踏上成为 TDD 开发人员的旅程，但最终还是靠自己才能让它成为现实。

你是自己代码的主要客户。TDD 方法可以帮助创建和定义服务和组件的接口，使开发工作更容易。在编写测试时，要考虑如何使用被测方法或类。它是否有意义？它是否直观？要正确使用它，是否需要大量培训或解释？如果自己都不喜欢这些问题的答案，就应当考虑重新设计接口，使它变得更好。

不要尝试把海洋煮沸。拥有大量可变部分的任务可能难以完成。将这些大型任务分解为较小的简单任务。完成这些小型任务会使工作更容易。能够快速完成几个小型任务有助于增加动力，树立完成任务的信心。

实施 TDD 可以收获许多好处。因为代码能够反映业务需求，所以应用程序中的缺陷会更少。为了保持代码的可测性，需要遵守一些原则，而采用这些原则的一个附带作用就是得到一套设计更佳的类、接口和方法。你的代码能够代表业务需求，提高用户的满意度。

最后，开发团队将能够交付一个精心设计的应用程序，拥有高品质的代码基。

在向团队介绍 TDD 时，要慢慢开始。TDD 与他们过去习惯的做法有显著不同。首先从 TDD 的基础原则依赖项注入着手。一旦团队理解并习惯了依赖项注入，就可以向他们介绍编写测试的理念。当团队熟悉并精通每项原则、实务或技术之后，就可以介绍下一项了。不要过快介绍变化，要掌握团队的节奏。当他们为下一步做好准备之后，你会知道的。

附录 A

TDD Katas

本附录内容

- 练习本书所学内容的重要性
- 为什么必须与他人共享代码
- 练习错误作法是如何浪费时间的
- OSIM Kata

Kata 是日语中表示“练习”的词汇。我最早是在 8 岁开始学习跆拳道时听到这个词的。Kata 就是一系列动作设计，用来帮助学员强化技巧，牢记一些特定的攻击和防守模式。如果曾经看到两位经过训练的高水平武术家对战，肯定会注意到他们难以置信的移动和攻防速度。这就是长时间练习 kata 的结果。

熟能生巧。天赋当然是有帮助的。但大多数运动员、音乐家和任何在自己领域取得极大成功的人都会告诉你：正是通过练习将天赋的潜能转换为成功的现实。在 Malcolm Gladwell 的 *Outliers* 一书中(Little, Brown and Company, 2008, ISBN: 9780316017923)，定义了“10 000 小时准则”。简而言之，这条准则是说：成功的关键主要是将一项特定任务练习大约 10 000 小时。他给出了一个很有说服力的论据，引用了甲壳虫乐队、比尔·盖茨和泰格·伍兹的成功。

运用 TDD Katas

要想成为一位高效率的 TDD 开发人员，“实践”对于培养其所需技术非常重要。必须掌握新的技能，如依赖项注入、MVC/MVVM、单元测试框架和支持框架(如 NBehave)。在使用 TDD 时，还必须进行许多精神方面的调整，如采用 SOLID 原则、习惯“测试优先”开发的理念。

为了创建一个 TDD kata，本附录为 OSIM 应用程序提供了一系列用户情景。该应用程序在设计上是不够完整的。我希望你能够把这些用户情景拿来，以 kata 的形式完成 OSIM。做一次，然后再做一次。然后再做一次。一直不停地做，直到对自己的 TDD 技能感到满意为止。如果目前没有在有支持 TDD 的环境中工作，这一点尤为必要。您的技能就像是一把锯子，它可能是由能工巧匠使用最上等的材料制作的，但如果不保养它，它就不会长久保持锋利。

共享作品

几年以前，我开始玩冰球。冰球的一大部分技能(我觉得应当有 90%)是滑冰的能力。我多少知道一点如何滑冰，但还没有真正达到能够参与竞赛的水平。我非常急切地开始练习；我抓住每一点可供利用的冰上时间，花费几个小时进行练习。我已经做得相当不错了，但还是有一件事困扰着我，那就是转身。专业冰球手可以在不减速的情况下急转弯。我希望也能够做到这一点，所以我花费了无数的时间进行练习。问题是我从来都没有能做好这种动作。我有些不能理解。我认为我所做的动作与我看到其他人做的动作完全一样，那为什么我的水平没有提高呢？这有点令人灰心。

有一天，我参加一次开放训练，与一位做过教练的朋友说起这件事。我解释了我的问题，并问他能否给点帮助。他让我滑出去，做几次转弯。他立即知道我的问题所在：我的双脚分得太开了。我根据他的指导对自己的技术进行了调整，不久之后，我就能按照自己希望的方式转弯了。如果不是有人看了我的动作，并为我提供了另外一个视角，我可能永远都不知道要这样改变。

软件开发也是一样——特别是学习 TDD 等新技术时。要与别人分享自己的代码，这一点很重要。别人会看到你看不到的问题，还会有你没有考虑过的视角。别人将会在你认为自己认真检查过的地方看到一些懈怠的方法。别人能够看出你可能因为感情用事而犯错的地方。不要低估与别人共享工作并从他人获得反馈的威力。我花费了许多时间来练习错误的转弯方法。这是对时间和精力巨大浪费。不要犯我曾经犯过的错误；尽早开始向别人展示你的代码。

OSIM 用户情景

下面是 OSIM 应用程序的一个用户情景清单。注意，它们只描述了系统的期望功能，

而不是如何实现它们。具体如何实现要由你来决定。复制 OSIM 应用程序的一个副本，然后开始实现这些用户情景。如果感到自己需要练习某一具体领域，可以添加一些自己的用户情景。在完成任务并向别人展示了代码之后，可以抛弃它，重新来过。

- 用户必须能够登录应用程序并经过验证。
- 用户必须能够登出应用程序。
- 用户必须能够查看可用物品类型列表。
- 用户必须能够向系统添加物品类型。
- 用户必须能够从系统中删除物品类型。
- 用户必须能够登录新库，并记录接收量。
- 用户必须能够在分发时登出库。
- 系统必须跟踪哪些供应品(类型和数量)被分发给哪个部门。
- 系统必须跟踪每个物品类型的内部价格(供应部门向其他部门收取的供应价格)。
- 系统必须保存当前库存的汇总。
- 系统必须保存每个部门月供应账单的汇总。
- 系统必须允许用户为每种物品类型指定再订购级别。
- 当某物品类型的当前库存低于该再订购级别之后，系统必须提醒用户。
- 系统必须按月为每个部门准备一份账单。
- 系统必须为每个物品类型跟踪各项目销售商价格(外部销售商对每个项目收取的价格)。
- 系统必须生成月度使用报表(每个项目的分发数量)。
- 系统必须生成月度成本报表(向外部销售商支付多少，为哪种项目支付)。