

读者对《实务单元测试》的评价

对于任何.NET 程序员，无论他使用.NET 的何种语言进行编码，能得到这本书将会是一件很荣幸的事情。得到之后，他们不应该让这本书束之高阁，而应该是坐在显示器前慢慢翻阅这本书。对于任何程序员，单元测试都是一项必不可少的技能；而 Andy 和 Dave 已经为他们编写了一本针对该主题的经典书籍。

Justin Gehtland, Relevance LLC 的创建者

实务程序员系列又为我们带来了一本非常有用的指导书籍。本书直接面向 C#程序员，使用了最流行的单元测试包 NUnit，不仅仅介绍了单元测试的一些基础知识，还展现了应该测试哪些方面，如何进行测试。向所有的.NET 开发者推荐这本书。

Mike Gunderloy, ADT 杂志的核心编辑

作为 Mono 项目的一部分，我们针对我们的类库，有章法地创建和维护了大量的单元测试。对于任何希望编写可靠代码的程序员，这将是一本极其难得的书籍。

Minguel de Icaza, Mono 项目组, Novell 公司

Andy 和 Dava 为单元测试撰写了一本优秀的、实用的和具有实务指导意义的书籍，书中基于最新的 NUnit 版本，展现了许多有用的例子。

Charlie Poole, NUnit 框架的开发者

只要使用 Dava 和 Andy 给出的方法，你就可以大大减少代码臭虫的数量，从而给你带来更快的开发速度和更好的程序。尝试这些技术吧——他们肯定会适合你的。

Ron Jeffries, www.XProgramming.com

译序

随着敏捷开发方法的风行，单元测试的思想也在遍地开花。越来越多的开发人员和公司开始学习和接受这种思想，并且在日常的开发工作中进行一些尝试。但是在尝试中，人们却遇到了不少问题和障碍。比如，如何将单元测试的思想传播给小组中的每一位成员，并让他们心悦诚服地接受；如何坚持这种边写边测的做法并最终养成一种习惯；如何进行高效的单元测试，而不是把时间无意义地浪费在对 `getter`、`setter` 的测试之上。

在 UT 中，Dave 和 Andy 为我们揭开了这层层面纱，游刃有余地为我们展现单元测试的世界。尤其难能可贵的是，作者并没有采用列举教条的方式来说教，而是以一种与朋友对话谈心的方式来说服你，进而从良师益友的角度来教导你，最后再很有导师风范地给你指明一些设计层次上的问题；最终让你充满自信，做到对代码胸有丘壑。我相信，本书将不仅仅是学习单元测试的 `starter kit`，还将会成为你工作中常伴身边的良师益友。

对于每个在单元测试上探索和工作的读者，本书希望能够帮助你们少跌几次跤，少碰几次壁，尽快踏上单元测试的康庄大道。这也是作者和我们的愿望。

本书由我和陶文两人合译。陶文的文笔技术都很不错，与他合作是一段美妙的经历。本书编辑方舟、陈元玉付出了辛勤的工作，为中文版增色不少。力求体现原书的思想精髓，使作者的睿智之言不因语言转化而略失光彩，这是我们最大的心愿。同时，也希望读者阅读过程中能够提出宝贵意见，帮助我们不断提高和改进译本。

在此，我要感谢周筠老师、陈英老师、方舟、陈元玉和我的爱人！

关于程序员修炼三部曲

在本系列的第一本书《The Pragmatic Programmer (程序员修炼之道：从小工到专家)》中，对现代软件开发的许多实质性话题作了概要性的介绍，并且获得了广泛的好评。自从该书 1999 年第一次印刷出版之后，就有许多读者向我们询问是否能再出一些后续书籍，或者该书的续篇。我们将会考虑此事；但首先，我们觉得有必要先提供几本基础方面的系列书籍。

在《程序员修炼之道：从小工到专家》出版之后的这些年来，我们发现：那些刚开始从事软件开发的读者，非常希望能够在软件开发的基本细节方面得到适当的指引，这样有助于他们早点养成良好的开发习惯；另一方面，那些经验丰富的读者，虽然已经能够完全理解书中的大多数内容，但是他们仍然希望在说服和指导开发小组中其他组员时，能得到一些帮助。现在，我们可以很高兴地告诉这些读者，我们已经有了几本能够给他们带来真正帮助的书籍了。

《Pragmatic Starter Kit (程序员修炼三部曲)》是一套由三本小册子组成的丛书，覆盖了对于现代软件开发非常重要的基础性知识。就内容而言，这三本小册子展现了实际操作、工具使用和开发哲学，而所有这些都是帮助开发小组正常开发、不断进步、带来高开发效率的利器。了解并掌握了这些知识之后，你和你的小组成员将能够很容易地养成好的开发习惯，并且对于你们所开发的项目而言，等于在外面织上了一层安全的网，它能让你感到安全和舒适。

第一本小册子《Pragmatic Version Control (版本控制之道)》讲述如何使用版本控制给整个项目打基础。打个比方来说，一个没有使用版本控制的项目，就像一个没有 UNDO 按钮的文字处理器：你输入的字符越多，错误所造成的伤害也就越大，因为没有撤销选择的机制。《Pragmatic Version Control (版本控制之道)》这本书将会告诉你：如何有效地使用版本控制系统，从中获取最大的好处和安全性，而不必拘泥于极端死板或者冗长可怕的过程。

《Pragmatic Unit Testing (单元测试之道)》是这个系列的第二本。单元测试是一项很重要的技术,它能够在程序员编写代码的同时,提供及时真实的反馈。遗憾的是,许多开发者对单元测试都不十分理解,没有意识到它可以令开发者的工作变得更加轻松。这本书有两种语言的版本可供选择:针对Java的JUnit版和针对C#的NUnit版。

第三本书《Pragmatic Automation (项目自动化之道)》¹囊括了在代码构建、测试和发布过程的自动化方面需要的一些非常重要的实践和技术。很少有项目会因为时间过多而失败,往往都是由于时间不足所造成;因此,《Pragmatic Automation (项目自动化之道)》主要是告诉你如何让计算机来完成更多的重复性工作,从而程序员能更加专注于比较有趣也比较困难的工作。

就书的风格而言,这几本书和我们的第一本书一样都是很通俗的。主要是帮助解决和满足程序员在日常工作中遇到的一些具体问题和需要。然而,这几本书并不是那种只给出一般问题、泛泛而谈的肤浅之作,它会让你充分理解这些知识,这样即使面对新出现的问题(本书可能没有明确提到的问题),你也能够根据这些知识找出自己的解决方法。

需要本书和其他书籍的更新信息,以及一些针对开发者和项目经理的相关资源,请访问我们的网站:

<http://www.pragmaticprogrammer.com>

感谢你的阅读,请记住要让阅读充满乐趣!

¹ 2004年7月出版(编者注:中译本《项目自动化之道》预定2005年初出版)。

前言

欢迎来到单元测试的世界。我们希望这本书能成为你和你的团队非常有价值的资源。你可以告诉我们它给你带来的帮助,或者我们需要做哪些改进,也可以通过访问我们网站¹中为《Pragmatic Unit Testing (单元测试之道)》专门开设的网页,并点击“FeedBack”来给我们提供反馈。

反馈能使这些书变得更加优秀,从而使开发者本身甚至整个项目都变得更加优秀。注重实效的 (Pragmatic) 程序员几乎都是使用这种来自现实的反馈 (如单元测试) 来优化代码,调整设计。

是什么让我们对单元测试如此重视呢?我们都明白,对一个程序员而言,单元测试是非常重要的,因为它提供了你所需要的反馈。试想,如果没有单元测试的话,你可能就像在一个 yellow legal pad²上写程序,只能做最好的祈祷来盼望它能够正常运行。

这当然不是个可行的做法。

对此,本书可以给你提供帮助,它主要针对的读者是那些具有一定编码和设计经验,但是对单元测试并不是很有经验的 C#程序员。

虽然我们是使用 C#来编写书中的代码,使用的是 NUnit 框架;但是就单元测试中的这些概念而言,无论你用的是什么语言,比如 C++、Fortran、Ruby、Smalltalk 或者 VisualBasic 都是一样的;而且迄今为止,有超过 60 种不同的语言具有自己的、类似于 NUnit 的测试框架,这些框架都可以从网上免费下载³。

¹ <http://www.pragmaticprogrammer.com/sk/ut/>。

² 译注: A pad of ruled, usually yellow writing paper that measures 8¹/₂ by 14 inches。

³ <http://www.xprogramming.com/software.htm>。

对于已经用过单元测试的一些高级程序员，我们希望本书同样可以给你带来一些惊喜。你们可以跳过介绍 NUnit 用法的那些基础性章节，把精力放到后面的一些章节，内容包括如何来考虑怎样测试，测试是如何影响设计的，以及你可能会遇到的某些能影响整个开发小组的问题。

而且，要记住本书只是一个开始。它可能是你读过的第一本讲述单元测试的书，但是我们不希望它是最后一本。

■ 何处获取源码

在本书的许多地方，你都会发现 C#代码的例子；其中某些是完整的程序，而另外一些则是一些程序的片断。如果你希望运行这些例子代码，或者查看完整的源码（而不是书中所给出的代码片断），你可以留意一下源码所在页的边缘：我们将会给出包含该源码片断的完整源码所在的文件名。

某些源码片断将会随着讨论的深入不断增加，因此在某些主目录和子目录中，你会发现一些源码文件的名称是相同的，只是子目录所在的版本有更新而已（如 rev1，rev2 等等）。

本书中所有代码可以在本书专题网站上获取，地址为：

<http://www.pragmaticprogrammer.com/starter kit/ut/index.html>

■ 排版标记的规范

黑体

表明这里的名词是正要被定义的名词，或者来自于其他语言的名词。

定宽字体

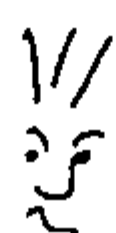
表明这里是方法名称，文件名称，类的名称，或者其他各种常量字符串。

xxx xx xx;

表明这里是源码中不重要的部分，在书中的代码片断中，它们被省略了。



曲线箭头标记表明这些内容是比较高级的，如果你第一次没有看懂的话，可以跳过。



“开发者 Joe”，他是我们的卡通朋友，在此他会提出一个相关的问题，你或许会发现这个问题非常有用。



Stop 按钮表明：读到这个地方的时候，你应该停下来，开始思考所提出的问题，或者在计算机上亲手做一些实验；得到结果之后，才继续阅读。

■ 基于语言的版本

《实务单元测试》为两种编程语言提供了不同的版本：

- 面向 Java 的 JUnit 版本。
- 面向 C# 的 NUnit 版本。

■ 致谢

我们要特别感谢本书下面的许多参与者：Mitch Amiano, Nascif Abousalh-Neto, Andrew C. Oliver, Jared Richardson 以及 Bobby Woolf，他们帮助我们输入，给了我们许多宝贵的建议和开发过程中发生的小故事。

我们还要感谢我们的审阅者。在指正我们的错误、疏忽和偶尔不太规范的语言习惯方面，他们花了很多时间和精力；这其中包括：Gareth Hayter, Dominique Plante, Charlie Poole, Maik Schmidt 以及 David Starnes。

非常感谢所有对本书做出贡献的人们，感谢你们的工作和支持。

Andy Hunt and Dave Thomas

March, 2004

pragprog@pragmaticprogrammer.com

目 录

关于程序员修炼三部曲.....	xi
前言.....	xiii
第 1 章 序言.....	1
1.1 自信地编码	2
1.2 什么是单元测试.....	3
1.3 为什么要使用单元测试	4
1.4 我需要做什么呢.....	5
1.5 如何进行单元测试.....	7
1.6 不写测试的借口	7
1.7 本书概要	12
第 2 章 你的首个单元测试.....	13
2.1 计划你的测试	14
2.2 测试一个简单的方法	15
2.3 使用 NUnit 来运行测试	16
2.4 运行例子	22
2.5 更多的测试.....	26
第 3 章 使用 NUnit 编写测试	27
3.1 构建单元测试.....	27
3.2 NUnit 的各种断言	29
3.3 NUnit 框架.....	31
3.4 NUnit 测试的组成.....	33
3.5 自定义 NUnit 断言	40
3.6 NUnit 和异常	41
3.7 临时忽略一些测试	42

第 4 章 测试哪些内容: Right-BICEP.....	45
4.1 结果是否正确	46
4.2 边界条件	49
4.3 检查反向关联	50
4.4 使用其他手段来实现交叉检查	50
4.5 强制产生错误条件	51
4.6 性能特性	52
第 5 章 CORRECT 边界条件.....	55
5.1 一致性	56
5.2 有序性	57
5.3 区间性	59
5.4 引用/耦合性	62
5.5 存在性	63
5.6 基数性	64
5.7 时间性	66
5.8 自己动手尝试	68
第 6 章 使用 Mock 对象.....	73
6.1 简单的替换	74
6.2 Mock 对象.....	75
6.3 正规化 Mock Objects.....	79
6.4 什么时候不应使用 Mock.....	93
第 7 章 好的测试所具有的品质.....	95
7.1 自动化	96
7.2 彻底的	97
7.3 可重复	99
7.4 独立的	99
7.5 专业的	100
7.6 对测试进行测试	102
第 8 章 在项目中进行测试.....	105
8.1 把测试代码放到哪儿	105
8.2 测试的礼貌	108
8.3 测试的频率	109

8.4 测试与遗留代码 110

8.5 测试与评审 113

第 9 章 设计话题..... 117

9.1 面向测试的设计 117

9.2 为测试而重构 119

9.3 测试类的不变性 130

9.4 测试驱动的设计 132

9.5 测试无效的参数 134

附录 A Gotchas..... 137

A.1 只要代码能工作就可以 137

A.2 “冒烟”测试 137

A.3 “请让我的机器来运行” 138

A.4 浮点数问题 138

A.5 测试耗费的时间太多了 139

A.6 测试总是失败 139

A.7 在某些机器上测试失败 140

附录 B 资源 141

B.1 网络资源 141

B.2 参考书目 143

附录 C 注重实效的单元测试：总结..... 145

附录 D 习题答案..... 147

索引..... 155

第 1 章

序言

Introduction

在一个软件项目中，我们可以做许多各式各样的测试，而且这些测试也是必须的。其中的某些测试需要用户的大量参与；而某些则需要有专门的质量保证小组来进行；或者需要其他的一些昂贵资源。

然而，这些测试并不是我们所要谈论的主题。

我们谈的是**单元测试**：它是项目成功、个人成功的一个不可或缺的部分，但对它，人们却又存在各种各样的误解。单元测试其实是相对廉价而简单的技术，但它能让你更高效地写出质量更好的代码。

说到测试，大凡组织和个人都会满怀雄心壮志，但是往往只是在项目快要结束的时候才想起测试。而那时的进度压力一定非常紧迫，所以结果往往只是浅尝辄止或者干脆就不测了。

许多程序员觉得测试只是一件麻烦事：一种自找的烦恼，唯一的“效果”就是让他们没法专注于手上的正经事——cutting code（堆砌代码）。

每个人都同意，是的，该做更多的测试。这种人人同意的事情还多着呢，是的，该多吃蔬菜，该戒烟，该多休息，该多锻炼……这并不意味着我们中的所有人都会这么去做，不是吗？

但是单元测试却远远不仅仅是上面这些——也许你会认为单元测试是花菜那一类的，而我要说它更像一勺美味的调料，它让每份菜肴品尝起来都

更加可口。单元测试的设计目的并不是为了获得一些更好的整体质量。也就是说，它并不是一个针对最终用户、项目经理和开发组长的工具；而是由程序员自己来完成，并且最终受益的也是程序员自己。我们是为了自身的利益去使用单元测试的，从而让我们的工作变得更加轻松。

简而言之，在有些时候，使用单元测试本身就能决定你的项目的成败。接下来，让我们来看看下面这个小故事。

1.1 自信地编码

有一次——或许就是上个礼拜二——有两个开发者：**Pat** 和 **Dale**。他们面临着相同的最后期限，而这一天也越来越近了。**Pat** 每天都在着急地编写代码，写完一个类又写一个类，写完一个函数又接着写另一个函数，还经常不得不停下来做一些调整，使得代码能够通过编译。

Pat 一直保持着这种工作方式，直到最后期限的前一天。而这时已经是演示所有代码的时候了。**Pat** 运行了最上层的程序，但是一点输出也没有，什么都没有。这时只好用调试器来单步跟踪了。“Hmm，决不可能是这样的”，**Pat** 想，“此时这个变量绝对不是 0 啊”。于是，**Pat** 只能回过头来看代码，尝试着跟踪一下这个难以琢磨的程序的调用流程。

时间已经越来越晚了，**Pat** 找到并且纠正了这个 bug；但在这个过程中，**Pat** 又找到了其他好几个 bug；如此几次过后，bug 还是存在。而程序输出那边，仍然没有结果。这时，**Pat** 已经筋疲力尽了，完全搞不清楚为什么会这样，认为这种（没有输出的）行为是毫无道理的。

而于此同时，**Dale** 并没像 **Pat** 那么快地写代码。**Dale** 在写一个函数的时候，会附带写一个简短的测试程序来测试这个函数。这里没有什么特殊的地方，只是添加了一个简单的测试，来判断函数的功能是否和程序员期望的一致。显然，考虑如何写，然后把测试写出来，是需要占用一定时间的；但是 **Dale** 在未对刚写的函数做出确认之前，是不会接着写新代码的。也就是说，只有等到已知函数都得到确认之后，**Dale** 才会继续编写下一个函数，然后调用前面的函数等等。

在整个过程中，Dale 几乎不使用调试器；而且对 Pat 的模样也有些困惑不解：只见他头埋在两手之间，嘀咕着各种难听的话语，咒骂着计算机，充血的眼球同时盯着好几个调试窗口。

最后期限终于到了，Pat 未能完成任务。而 Dale 的代码被集成到整个系统中，并且能够很好地运行。之后，在 Dale 的模块中，出现了一个小问题；但是 Dale 很快就发现了问题所在，在几分钟之内就解决了问题。

现在，是该总结一下上面这个小故事的时候了：Dale 和 Pat 的年纪相当，编码能力相当，智力也差不多。唯一的区别就是 Dale 非常相信单元测试；对于每个新写的函数，在其他代码使用这个函数并对它形成依赖之前，都要先做单元测试。

而 Pat 则没有这么做，他总是“知道”代码的行为应该和所期望的完全一样，并且等到所有代码都差不多写完的时候，才想起来运行一下代码。然而到了这个时候，要想定位 bug，或者，甚至是确定哪些代码的行为是正确的，哪些代码的行为是错误的，都为时已晚了。

1.2 什么是单元测试

单元测试是开发者编写的一小段代码，用于检验被测代码的一个很小的、很明确的功能是否正确。通常而言，一个单元测试是用于判断某个特定条件（或者场景）下某个特定函数的行为。例如，你可能把一个很大的值放入一个有序 list 中去，然后确认该值出现在 list 的尾部。或者，你可能会从字符串中删除匹配某种模式的字符，然后确认字符串确实不再包含这些字符了。

执行单元测试，是为了证明某段代码的行为确实和开发者所期望的一致。

对于客户或最终使用者而言，这种测试必要吗，它与验收测试有关吗？这个问题仍然很难回答。事实上，我们在此并不关心整个产品的确认、验证和正确性等等；甚至此时，我们都不去关心性能方面的问题。我们所要做的一切就是要证明代码的行为和我们的期望一致。因此，我们所要测试的是规模很小的、非常独立的功能片断。通过对所有单独部分的行为建立起信心，确信它们都和我们的期望一致；然后，我们才能开始组装和测试整个系统。

毕竟，要是我们对手上正在写的代码的行为是否和我们的期望一致都没把握，那么其他形式的测试也都只能是浪费时间而已。在单元测试之后，你还需要其他形式的测试，有可能是更正规的测试，那一切就都要看环境的需要来决定。总之，做测试如同做善事，总是要从家¹开始。

1.3 为什么要使用单元测试

单元测试不但会使你的工作完成得更轻松，而且会令你的设计变得更好，甚至大大减少你花在调试上面的时间。

在我们上面的小故事里面，Pat 因为假设底层的代码是正确无误的而卷入麻烦之中，先是高层代码中使用了底层代码；然后这些高层代码又被更高层的代码所使用，如此往复。在对这些代码的行为没有任何信心的前提下，Pat 等于是在假设上面用竖立卡片堆砌了一间房子——只要将下面卡片轻轻移动，整间房子就会轰然倒塌。

当基本的底层代码不再可靠时，那么必需的改动就无法只局限在底层。虽然你可以修正底层的问题，但是这些对底层代码的修改必然会影响到高层代码，于是高层代码也连带地需要修改；以此递推，就很可能动到更高层的代码。于是，一个对底层代码的修正，可能会导致对几乎所有代码的一连串改动，从而使修改越来越多，也越来越复杂。于是，整间由卡片堆成的房子就由此倒塌，从而使整个项目也以失败告终。

Pat 总是说：“这怎么可能呢？”或者“我实在想不明白为什么会这样”。如果你发现自己有时候也会有这种想法，那么这通常是你对自己的代码还缺乏足够信心的表现——你并不能确认哪些是工作正常的而哪些不是。

为了获得 Dale 所具有的那种对代码的信心，你需要“询问”代码究竟做了什么，并检查所产生的结果是否确实和你所期望的一致。

这个简单的想法描述了单元测试的核心内涵：这个简单有效的技术就是为了令代码变得更加完美。

¹ 译注：begins at home，这里的家指的是代码最基本的正确性。

1.4 我需要做什么呢

引入单元测试是很简单的，因为它本身就充满了乐趣。然而在项目交付的时候，我们给客户和最终用户的仍然是产品代码，而不包含单元测试的代码；因此，我们必须对单元测试的目的有个充分的认识。首先也是最重要的，使用单元测试是为了使你的工作——以及你队友的工作——完成得更加轻松。

■ 它的行为和我的期望一致吗

最根本的，你需要回答下面这个问题：“这段代码达到我的目的了吗？”也许就需求而言，代码所做的是错误的事情，但那是另外一个问题了。你要的是代码向你证明它所做的就是**你所期望的**。

■ 它的行为一直和我的期望一致吗

许多开发者说他们只编写一个测试。也就是让所有代码从头到尾跑一次，只测试代码的一条正确执行路径，只要这样走一遍下来没有问题，测试也就算是完成了。

但是，现实生活当然不会这么事事顺心，事情也不总是那么美好：代码会抛出异常，硬盘会没有剩余空间，网络会掉线，缓冲区会溢出等——而我们写的代码也会出现 bug。这就是软件开发的“工程”部分。就“工程”而言，土木工程师在设计一座桥梁的时候，必须考虑桥梁的负载、强风的影响、地震、洪水等等。电子工程师要考虑频率漂移、电压尖峰、噪音，甚至这些同时出现时所带来的问题。

你不能这样来测试一座桥梁：在风和日丽的某一天，仅让一辆车顺利地开过这座桥。显然，这种测试对于桥梁测试来说是远远不够的。类似地，在测试某段代码的行为是否和你的期望一致时，你需要确认：在任何情况下，这段代码是否都和你的期望一致；譬如在风很大、参数很可疑、硬盘没有剩余空间、网络掉线的时候。

■ 我可以依赖单元测试吗

不能依赖的代码是没有多大用处的。但更糟糕的是，那些你自认为可以信赖的代码（但是结果证明这些代码是有 bug 的）有时候也会让你花很多时间在跟踪和调试上面。显然，几乎没有项目可以允许你在这上面浪费太多的时间，因此无论如何，你都要避免这种“前进一步，后退两步”的开发方法。也就是说，要让开发过程保持稳定的步伐前进。

没人能够写出完美无缺的代码；但是这并没有关系——只要你知道问题的所在就足够了。许多大型软件项目的失败，诸如只能把坏了的太空船搁浅在遥远的行星，或者在飞行的途中就爆炸了，都能通过认知软件的限制来避免。例如，Arianne 5 号火箭软件重用了来自于之前一个火箭项目的一个程序库，而这个程序库并不能处理新火箭的飞行高度（比原来火箭要高）²，从而在起飞 40 秒之后就发生了爆炸，导致 5 亿美元的损失。

显然，我们希望能够依赖于所编写的代码，并且清楚地知道这些代码的功能和约束。

例如，假设你写了一个反转数值序列的方法。在测试的过程中，你也许会传一个空序列给这个程序——但导致了程序崩溃。实际上，程序并没有要求该程序必须能够接收一个空序列，因此你可以只在方法的注释中说明这个约束：如果传递一个空序列给这个方法，那么这个方法将会抛出一个异常。现在你马上就知道了该代码的约束，从而也就不需要用其他很麻烦的方法来解决这个问题（因为在某些地点要解决这个问题并不方便，比如在高空大气层中）。

■ 单元测试说明我的意图了吗

对于单元测试而言，一个最让人高兴的意外收获就是它能够帮助你充分理解代码的用法。从效果上而言，单元测试就像是能执行的文档，说明了在你用各种条件调用代码时，你所能期望这段代码完成的功能。

² 致航空迷：数值溢出的原因是由于新轨道增加了火箭的垂直加速度，从而出现了一个更大的水平倾斜。

项目成员能够通过查看单元测试来找到如何使用你所写代码的例子。如果他偶然发现了一个你没有考虑到的测试用例，那么他也可以很快地知道这个事实：你的代码可能并不支持这个用例。

显然，在正确性方面，可执行的文档有它的优势。与普通的文档不同的是，单元测试不会出现与代码不一致的情况（当然，除非你选择不运行这些测试）。

1.5 如何进行单元测试

单元测试本来就是一项简单易学的技术；但是如果能够遵循一些指导性原则(guideline)和基本步骤，那么学习将会变得更加容易和有效。

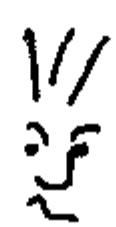
首先要考虑的是在编写这些测试方法之前，如何测试那些可疑的方法。有了这样一个大概的想法之后，你将可以在编写实现代码的时候，或者之前，编写测试代码本身。

下一步，你需要运行测试本身，或者同时运行系统模块的所有其他测试，甚至运行整个系统的测试，前提是这些测试运行起来相对较快。在此，我们要确保**所有的测试都能够通过**，而不只是新写的测试能够通过；这一点是非常重要的。也就是说，在保证不引入直接 bug 的同时，你也要保证不会给其他的测试带来破坏。

在这个测试过程中，我们须要确认每个测试究竟是通过了还是失败了——但这并不意味着你或者其他倒霉的人须要查看每个输出，然后才决定这些代码是正确的还是错误的。在此，你慢慢地就会养成一个习惯：只要用眼睛瞄一下测试结果，就可以马上知道所有代码是否都是正确的，或者哪些代码是有问题的。关于这个问题，我们将留在讨论如何使用单元测试框架时来具体讨论。

1.6 不写测试的借口

在听过了我们合情合理、热情洋溢的阐述之后，某些开发者会点头并且同意单元测试的必要性，但是也许仍然会告诉我们由于某些原因，不能编写单元测试。下面就是我们所听到的一些借口，当然其中也包含了我们的辩解。



Joe 提问:

什么是间接损害?

间接损害是: 在整个系统中, 当某一部分加入了新特性, 或者修复了一个 bug 之后, 给系统的其他 (与前面可能是互不相关的) 部分引入了一个新的 bug (或者损害)。如果无视这种损害并且继续开发的话, 那么将可能带来一个很危险的问题, 最后可能会导致整个系统崩溃, 并且没人能够修复。

我们有时候把这种损害称为 “Whac-a-Mole” 效应。在狂欢节中有一种游戏叫做 “Whac-a-Mole”, 在此游戏中, 当小孔中的机械鼠探出头来的时候, 参与者的任务就是敲这个机械鼠的头部, 然而正在这个时候, 机械鼠的头马上又缩了回去, 不让你打个正着; 而位于另一个孔的另一个机械鼠此时又探出头来。就这样, 机械鼠的头以非常快的速度探出又缩回, 使你总是打不到他, 从而也就得不了分。最后, 参与者通常只能无助地乱敲; 然而机械鼠的动作却总是出乎你的意料。

实际上, 对于代码而言, 大范围的间接损害就具有这样类似的效果。

编写单元测试太花時間了

对于开始做单元测试的初学者而言, 这是出现次数最多的借口。当然, 事实并不是这样的。首先, 我们需要关注一下: 在编写代码的时候, 你在哪些地方花费了更多的时间。

大多数人都把测试看成某种只有在项目结束时才做的工作。但是, 如果你到了那个时候才做单元测试的话, 那么肯定会花费更多的时间。事实上, 要想只在项目快要结束的时候才做单元测试, 那简直是扯淡。

至少, 我们可以这样来看待单元测试: 假设你想用一台割草机来清除两英亩的草地。如果你在草很稀疏的时候就开始行动, 那么工作将会很简单。但是如果你等到这块草地长上粗壮大树、缠绕灌木的时候才准备行动, 那么工作将会变得非常困难。

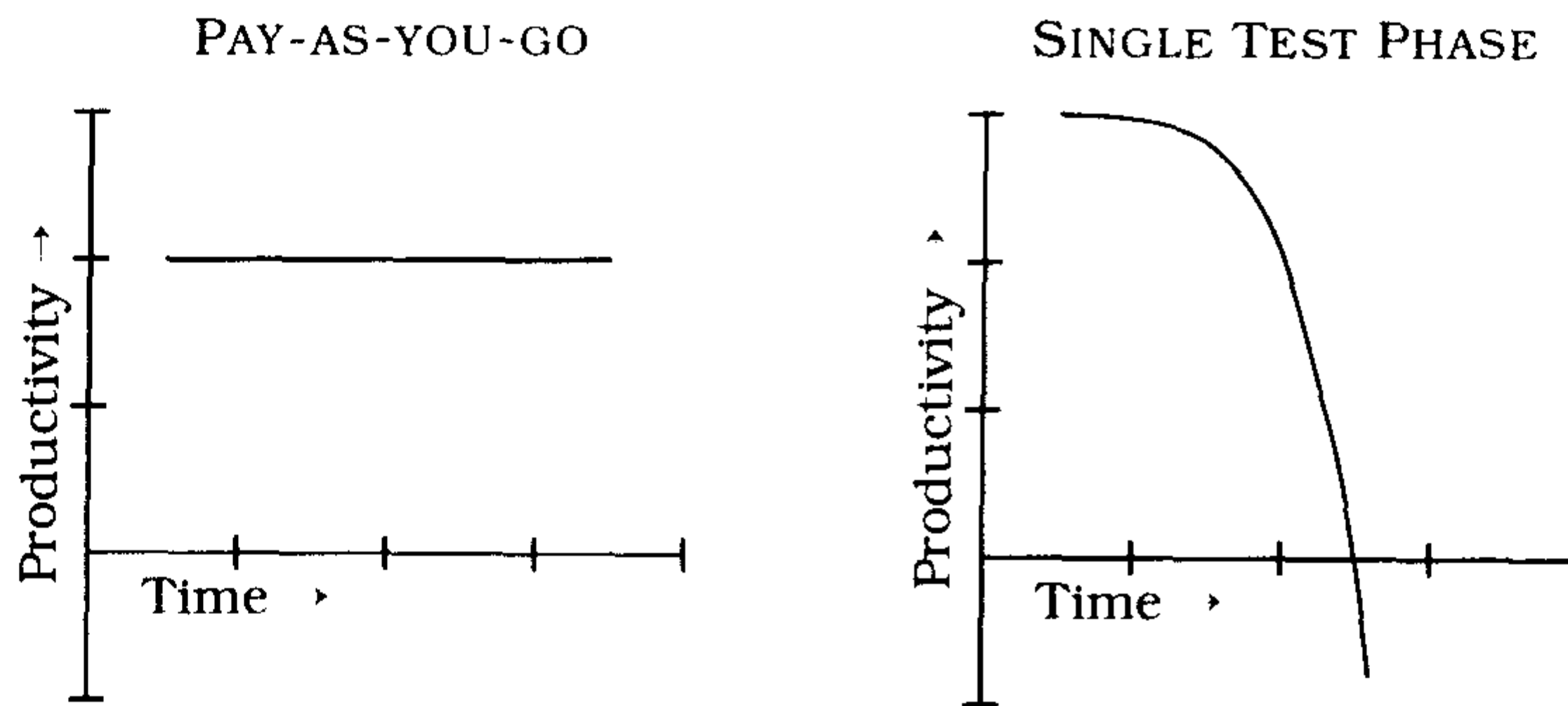


图1.1 “立即测试”和“单一测试阶段”的比较

从长远看来，使用“立即测试模型”的代价比“延后测试模型”的代价要低。在你编写实现代码的时候，同时编写独立的测试代码，在项目最后就可以避免出现做了无用功的问题；代码中的 bug 也会更少，因为你所依赖的都是已经通过测试的代码。于是，通过在开发过程中多花一点时间在编写单元测试上面，你就可以最小化在项目后期花费大量时间的风险。

从图 1.1 中你可以看到，“立即测试”和“延后测试”之间并没有权衡可言；而是直线效率和指数效率之间的对比，而且对于后者而言，复杂度会不断增加，并且在项目后期，很多工作需要从头再来。所有这些额外的工作都会影响你的工作效率，如图 1.1 所示。

显然，单元测试也并非免费的午餐。在立即测试模型中，单元测试是有开销的（在时间和金钱上面）。但是如果你查看右边曲线的方向，你会发现它花费了更多的开销——效率曲线急剧下降；而且生产率甚至会变成负值；这些生产率损耗可以很容易导致一个项目失败。

因此，如果你仍然认为在编写产品代码的时候，还是没有时间编写测试代码，那么请先考虑下面这些问题：

1. 对于所编写的代码，你在调试上面花了多少时间？
2. 对于以前你自认为正确的代码，而实际上这些代码却存在重大的 bug，你花了多少时间在重新确认这些代码上面？
3. 对于一个别人报告的 bug，你花了多少时间才找出导致这个 bug 的源码位置？

对于那些没有使用单元测试的程序员而言，上面这些问题所耗费的时间的递增速度是很快的，而且随着项目的深入，递增速度会变得更快；而另一方面，适当的单元测试却可以很大程度地减少这些时间，从而能够为你腾出足够的时间来编写所有的单元测试——甚至可能还有剩余的空闲时间。

运行测试的时间太长了

合适的测试是不会让这种情况发生的。实际上，大多数测试的执行都是非常快的，因此你在几秒之内就可以运行成千上万个测试。但是有时某些测试会花费很长的时间，因此我们不能每次都运行这些测试，有时你就要暂时停止运行这些测试。

在这种情况下，需要把这些耗时的测试和其他测试分开。通常可以每天运行这种测试一次，或者几天一次；而对于运行很快的测试，则可以经常运行。

测试代码并不是我的工作

这是一个很有趣的借口。请问，到底什么是你的工作呢？假设你的工作只是为了编写产品代码。如果对于那些没有把握的代码，你就随便地扔给测试组，那么你实际上并没有完成你的工作。实际上，期望别人来清理我们的代码是很不好的做法；甚至在某种情况下，如果别人指出了一大摞有错误的代码，那么也意味着你的职位也就到此为止了。

另一方面，如果测试员或者 QA（Quality Assurance）组发现很难在你的代码中挑出错误，那么你的名声将会一路高升——同时提升的还有你的职位。

我并不清楚代码的行为，所以也就无从测试

如果你实在不清楚代码的行为，那么估计现在并不是编码的时候。或许你应该先建立一个原型，这样才有助于你认清需求。

如果你并不知道代码的行为，那么你又如何知道你编写的代码是正确的呢？

但是这些代码都能够编译通过

OK，还没有人把这个当成一个借口，至少没有大声地说出来。但是估计这很容易会成为一个借口，因为有些人总是认为：一个成功的编译就是成功的标记；通过了编译，就像通过了天堂的大门一样。

但是，编译器的默许只是对代码一种非常粗略的肯定。实际上，任何编译器和解释器都只能验证你的语法是否正确，并不能保证代码的行为也正确。例如，C# 的编译器可以很容易地检测到下面这行代码是错误的：

```
static void Main() {
```

上面有个明显的打印错误，第 2 个单词应该是 `static`，而不是 `static`。这是很容易看出来的。但是现在假设你编写了下面这些代码：

```
public void Addit(Object anObject) {
    List myList = new List();
    myList.Add(anObject);
    myList.Add(anObject);
    // more code...
}
```

Main.cs

你是否想加同一个对象到相同 list 两次呢？或许是，或许不是。就这一点来说，编译器并不能告诉你，只有你知道你期望代码完成什么样的功能³。

公司请我来是为了写代码，而不是写测试

如果使用同样的逻辑，那么我们可以说：公司付给你薪水，并不是让你整天都在调试代码。显然，公司付给你薪水是为了让你编写产品代码，而单元测试大体上是一个工具，是一个和编辑器、开发环境、编译器等处于同一位置的工具。

³ 对于哪些基于你现有代码生成测试代码的自动化测试工具，同样会陷入这个陷阱——它们只会使用你编写的代码，而不是你期望的代码。

如果我让测试员或者 QA (Quality Assurance) 人员没有工作，那么我会觉得很内疚

你并不需要担心这些。请记住，我们在此只是谈论单元测试，而它只是一种针对源码的、低层次的，为程序员而设计的测试。在整个项目中，还有其他的很多测试需要这些人来完成，如：功能测试、验收测试、性能测试、环境测试、有效性测试、正确性测试、正规分析等等。

我的公司并不会让我在真实系统中运行单元测试

Whoa! 我们所讨论的只是针对开发者的单元测试。也就是说，如果你可以在其他的环境下（例如在正式的产品系统中）运行这些测试的话，那么它们就不再是单元测试，而是其他类型的测试了。实际上，你可以在你的本机运行单元测试，使用你自己的数据库，或者使用 mock 对象（见第 6 章）。

如果 QA 部门或者其他测试人员希望在产品或者其他阶段运行这些测试的话，你可以调整一些技术细节，从而使他们也可以运行一些测试；但是同时要让他们清楚，这些调整后的测试已经不再是单元测试了。

1.7 本书概要

第 2 章“你的首个单元测试”将会对单元测试做一个整体的概括；同时我们还要介绍一些编写单元测试的规则，而主要的规则我们将留在第 3 章“使用 NUnit 编写测试”中介绍。接下来，我们将会用几章的篇幅来介绍如何发现哪些部分需要测试，以及如何对他们进行测试。

在第 7 章，我们将会介绍优秀测试所具有的一些品质。在第 8 章，我们会针对你自己的项目，介绍高效使用单元测试的一些前提条件，这一章同时还讨论了如何处理具有遗留代码的项目。第 9 章“设计话题”将会介绍测试如何影响整个项目的设计（从好的方面而言）。

附录包含了很多额外的有用信息：一些常见的与单元测试相关的问题，关于安装 NUnit 的方法和一些有用的资源，同时包含了参考书目。我们最后还给出了一些总结性的句子，介绍了本书的一些要点和建议。

现在，让我们坐下来，放松一下，开始进入编写更好代码的世界。

第 2 章

你的首个单元测试

Your First Unit Tests

如我们在介绍中所说过的，一个单元测试是程序员编写的一段代码，用于执行另一段代码并确定那段代码的行为是否和程序员的期望一致。

然而在实际中，你应该怎么做呢？

为了验证代码的行为是否和你的期望一致，你需要使用一些 *assertion*（断言），它是一个简单的方法调用，用于判断某个语句是否为真。例如，方法 `IsTrue` 将会检查给定的二元条件是否为真；如果条件非真，则该断言将会失败。具体的实现如下面所示：

```
public void IsTrue(bool condition) {  
    if (!condition) {  
        abort();  
    }  
}
```

AssertTrue.cs

实际上，你可以使用断言来检查所有的这类事物，譬如以下为检查两个数字是否相等：

```
int a = 2;  
// XXXX ...  
// ...  
IsTrue(a == 2);  
// XXXX ...
```

如果由于某种原因，当调用 `IsTrue()` 的时候，`a` 并不等于 2，那么上面的程序将会中止。

既然我们经常需要检查是否相等，那么如果能够有一个专门针对数字的

断言将会更加方便。例如，为了检查两个整数是否相等，我们可以编写下面的方法，它接收两个整型参数：

```
public void AreEqual(int a, int b) {  
    IsTrue(a == b);  
}
```

有了这两个断言之后，我们就可以编写一些测试了。在接下来的内容中，我们将会介绍更多的断言；而有关在单元测试中如何使用断言的内容，将留到下一章介绍。现在，让我们先考虑：在编写任何代码之前，需要做哪些测试呢？

2.1 计划你的测试

我们将从下面这个简单的例子开始：它是一个独立静态方法，用于查找 list 中的最大值：

```
static int Largest(int[] list);
```

比如，给定一个数组[7,8,9]，这个方法应该返回 9。这就构成了一个很合理的测试。然而，你还想出其他的一些测试吗？就这个方法，在你继续阅读之前，请思考一分钟，并且把你想到的所有测试都记下来。



在继续阅读之前，先思考一小段时间……

你想到了多少个测试呢？

显然，对一个给定 list 而言，元素的位置对该方法的结果并不会产生影响；于是，你马上会想到下面这些测试（在下面中，我们将把输入输出写为：“输入 → 你所期望的输出”这样的格式）。

- [7, 8, 9] → 9
- [8, 9, 7] → 9
- [9, 7, 8] → 9

如果 list 中有两个相等的最大值，将会出现什么情况呢？

- [7, 9, 8, 9] → 9

由于这些元素都是 `int` 类型，而不是对象，那么你可能就不会在意哪个 9 会作为返回结果，只要是其中的一个返回就可以了。

如果 `list` 就只有一个元素，结果会是怎么样的呢：

- `[1] → 1`

并且如果 `list` 所包含的是负数，结果又会是什么样：

- `[-9, -8, -7] → -7`

返回结果看起来有些奇怪，但是 `-7` 确实要比 `-9` 大，这也正是我们期望的结果。很高兴的是，我们现在就可以获得结果；而不需要在调试器或者产品代码中才可以看到该函数的结果；实际上，在产品代码中，要观察函数的结果并非如此显而易见的。

为了使上面的概念更加具体，让我们来编写一个“寻找最大数”的函数，并且对它进行测试。下面是我们的首份实现代码：

```
Line 1  public class Cmp {
-
-      ///
-      /// <summary>
5      /// Return the largest element in a list.
-      /// </summary>
-      /// <param name="list"> A list of integers </param>
-      /// <returns>
-      /// The largest number in the given list
10     /// </returns>
-     ///
-     public static int Largest(int[] list) {
-         int index, max=Int32.MaxValue;
-         for (index = 0; index < list.Length-1; index++) {
15             if (list[index] > max) {
-                 max = list[index];
-             }
-         }
-         return max;
20     }
-
- }
```

Largest.cs

现在，我们对测试就已经有了一些概念了。接下来，我们将介绍在 C# 中如何使用 NUnit 框架来编写测试。

2.2 测试一个简单的方法

通常而言，你需要让你的首个测试非常简单，因为除了代码本身以外，还有其他的许多因素需要被测试：凌乱的类名，程序库的位置，还要确认代

码能够通过编译。在你的首个测试中，你就需要确保这些因素都是正确的。而在这之后，你就无须再担心这些因素了；因此在你调试一个复杂测试的同时，就不再须要调试这些复杂的集成问题。

首先，让我们先传递一个没有重复元素的简单数组给这个函数，并对这种情况进行测试。下面是该测试的完整代码。对于测试类，我们将留在下一章说明；在此，我们只要专注于断言语句就可以了：

```
using NUnit.Framework;
[TestFixture]
public class TestLargest {
    [Test]
    public void LargestOf3() {
        Assert.AreEqual(9, Cmp.Largest(new int[] {8,9,7}));
    }
}
```

C# note: 上面这种创建匿名数组的语法看起来有些奇怪，这完全是作者个人的习惯，在这里这样写只是为了使代码更加简短。如果你不喜欢上面这种写法，那么完全可以用下面的代码来代替它（尽管前面的语法也是一种常用的语法）：

```
[Test]
public void LargestOf3Alt() {
    int[] arr = new int[3];
    arr[0] = 8;
    arr[1] = 9;
    arr[2] = 7;
    Assert.AreEqual(9, Cmp.Largest(arr));
}
```

这就是你的首个测试。

请运行这个测试，并且确认它能够通过；为了完成这一步，我们需要先熟悉一下如何使用 NUnit 来运行测试。

2.3 使用 NUnit 来运行测试

NUnit 是一个可以免费获得的¹开源的产品，它提供了一套测试框架和一个测试运行程序（test runner）。一种分发的形式是 C# 源码，你可以自己编译这份代码，并且在你的计算机上安装 NUnit。另一种方式是使用 Microsoft Installer（MSI）文件。

¹ <http://www.nunit.org>

//



Joe 提问:

什么是 Open Source(开源)?

确切地说, 什么是开源? 开源指的是一些可以免费提供源码的软件。这通常意味着你可以免费地获得该软件产品, 并且能够修改它, 给它增加功能, 或者与你的朋友分享等等。

使用安全吗? 就大多数情况而言, 与对应的商业软件相比, 开源产品用起来更加安全, 因为它们是开放的, 所以已经有成千上万个有兴趣的开发者对它们进行了测试。而且, 在开源社团中, 恶意程序、监视程序、病毒等相关问题都很少, 甚至没有。

它合法吗? 完全合法。就像你能够自由制作一首歌曲, 编写一本书, 然后把它分享给别人(或者售卖给别人)一样, 你也可以自由地编写一些代码, 然后分享给其他人(或者售卖给其他人)。有多种开源的版权声明, 阐明与软件相关的自由行为。在你发布任何包含开源组件的软件之前, 你都应该仔细检查相关的特定版权约定。

我可以参与吗? 我们当然非常希望你能参与。开源的威力来自于许多天各一方的朋友: 这些人和你一样, 他们知道如何编写程序, 并且同时需要拥有某些新的特性。你能够给 NUnit 添加一个属性吗? 当然可以! 你可以编辑程序库或者任何一个 test runner 中的源码, 修改这些源码, 然后使用自己修改过的代码。你还可以通过 email 的形式把你的修改告知该产品的维护者, 从而他们就有机会把你的修改集成到下一版本中。

安装 NUnit 的最简单方法就是运行 MSI 文件,然后它会启动一个熟悉的 Windows 安装向导。注意,MSI 支持只内建于 Windows2000 和 XP 中,因此对于 Windows95、98、ME 和 NT4.0 的用户,你需要先从 Microsoft 的网站下载你的操作系统版本的安装服务(在 www.microsoft.com 中搜索“Windows Installer Redistributable”)。Linux 爱好者可能需要看一下 Mono, .Net 针对 Linux 环境的一个开源的移植版本。它通常包含于对应的 NUnit 版本中。

下一步,你需要编译我们前面给出的代码。如果你使用 Visual Studio,那么可以为我们的例子代码创建一个类型为 *Class Library* 的新项目。把我们的“产品”代码输入到一个名为 *Largest.cs* 的文件中,同时把测试代码输入到名为 *TestLargest.cs* 的文件中。如果你不想从头输入这些程序——我们已经在我们的网站提供了本书的源码,这可能会令你非常高兴²。

注意,测试代码使用了 `NUnit.Framework`,因此你需要增加一个指向 `nunit.framework.dll` 的引用才能编译这份代码。在 Visual Studio 中的主菜单中选择“Project”,然后选择“Add Reference...”,然后会弹出一个对话框,之后在 .Net 选项标签下选择“nunit.framework”,然后点击“SELECT”按钮把这个 dll 添加到组件列表中,如图 2.1 所示;最后点击 OK。现在你的项目就能够使用 NUnit 框架的功能了。

之后就可以依照通常的方法构建项目了(在 Visual Studio 中,可以使用 CTRL-SHIFT-B 快捷键)。现在,你就得到了一个程序集(assembly),但这只是一个库而已,我们如何运行它呢?

Test Runners 会解决这个问题。Test runner 知道如何寻找具有 [TestFixture] 属性的类和类中的 [Test] 方法。然后运行这些测试,并且统计哪些测试通过了,哪些失败了;最后返回给你一份结果报告。

有 3 种使用 test runner 的方法:

1. NUnit GUI。
2. NUnit 的命令行。
3. Visual Studio 的插件。

² <http://www.pragmaticprogrammer.com/sk/ut/>

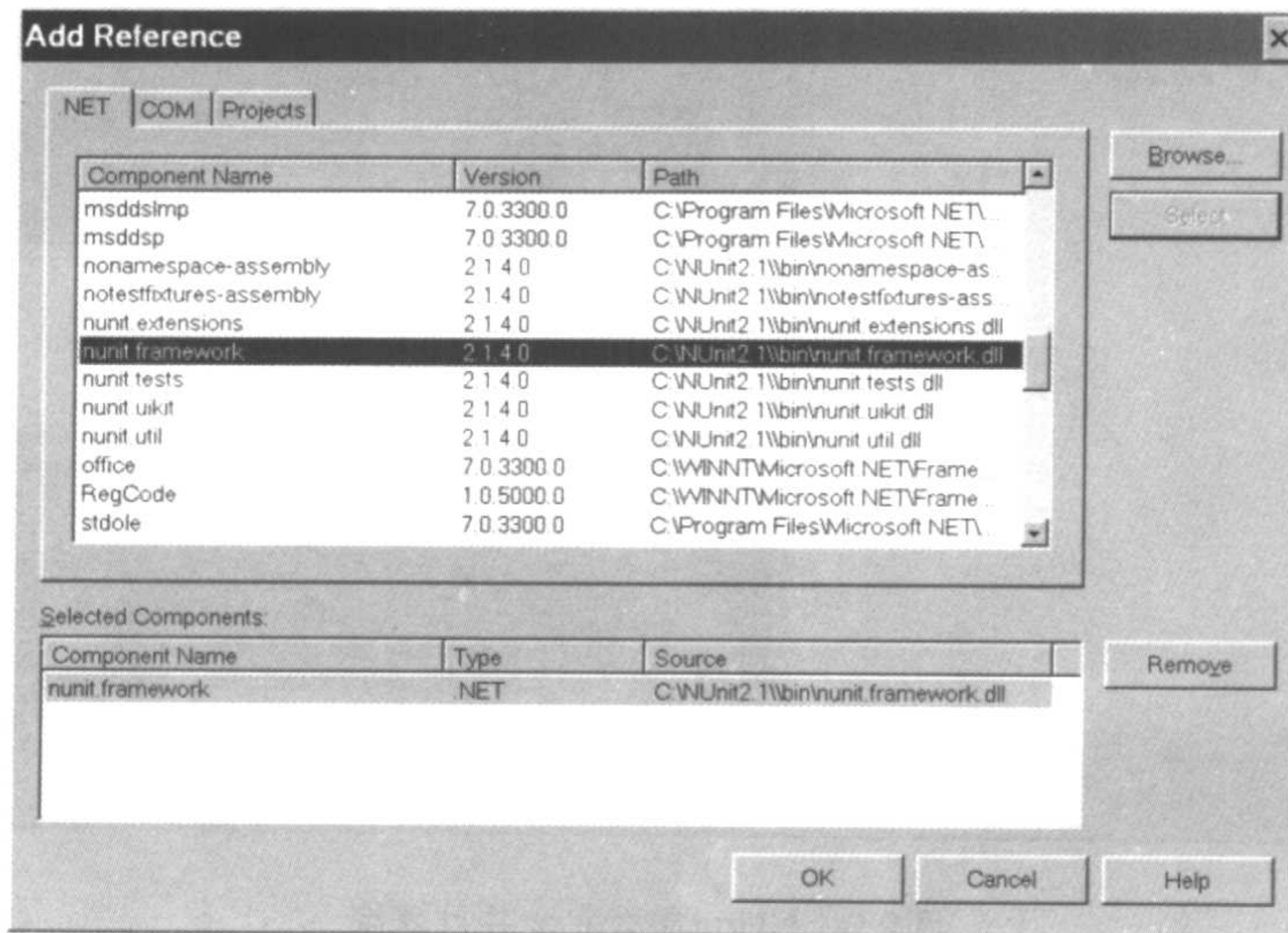


图2.1 添加NUnit的程序集引用

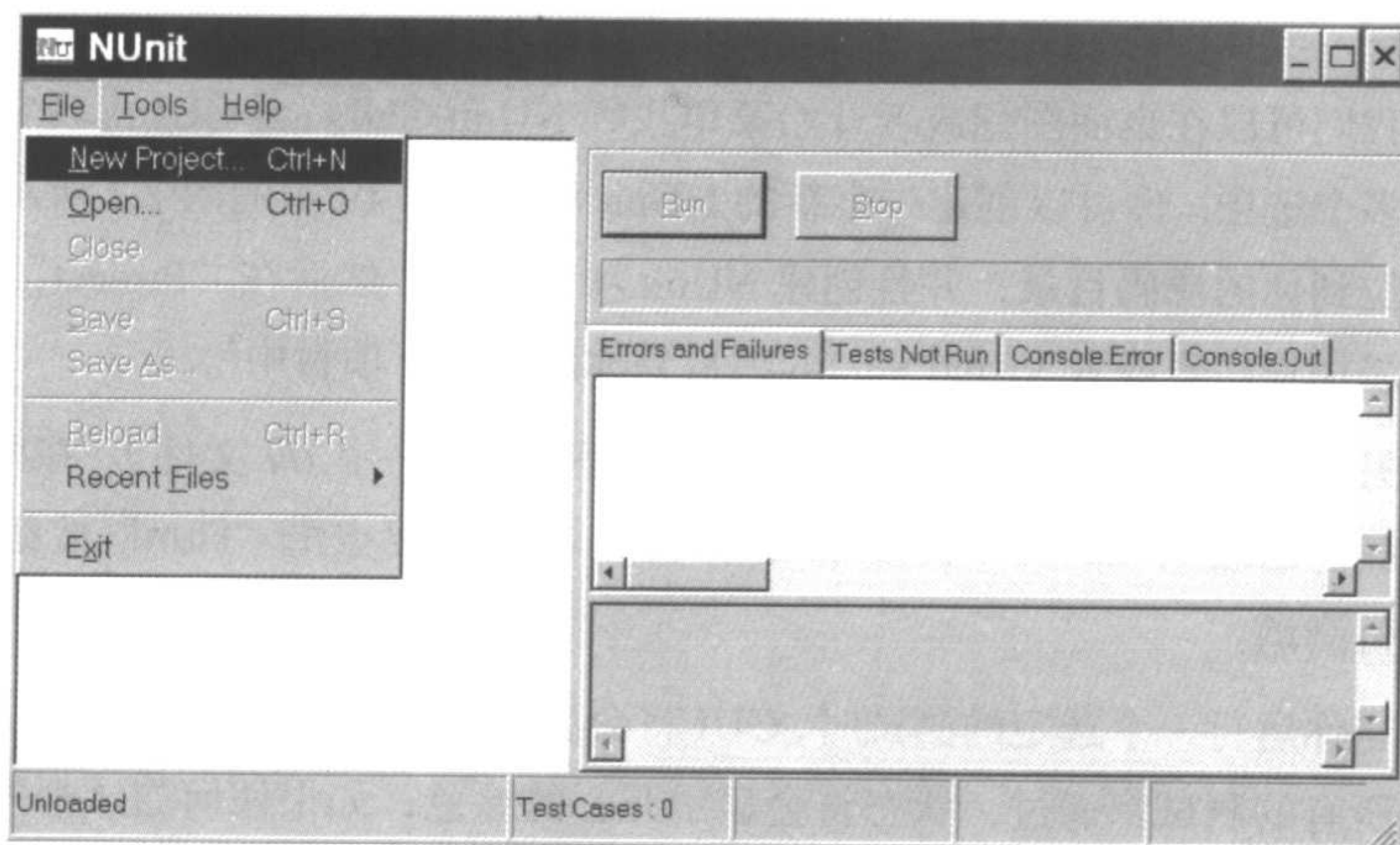


图2.2 创建一个新的NUnit项目

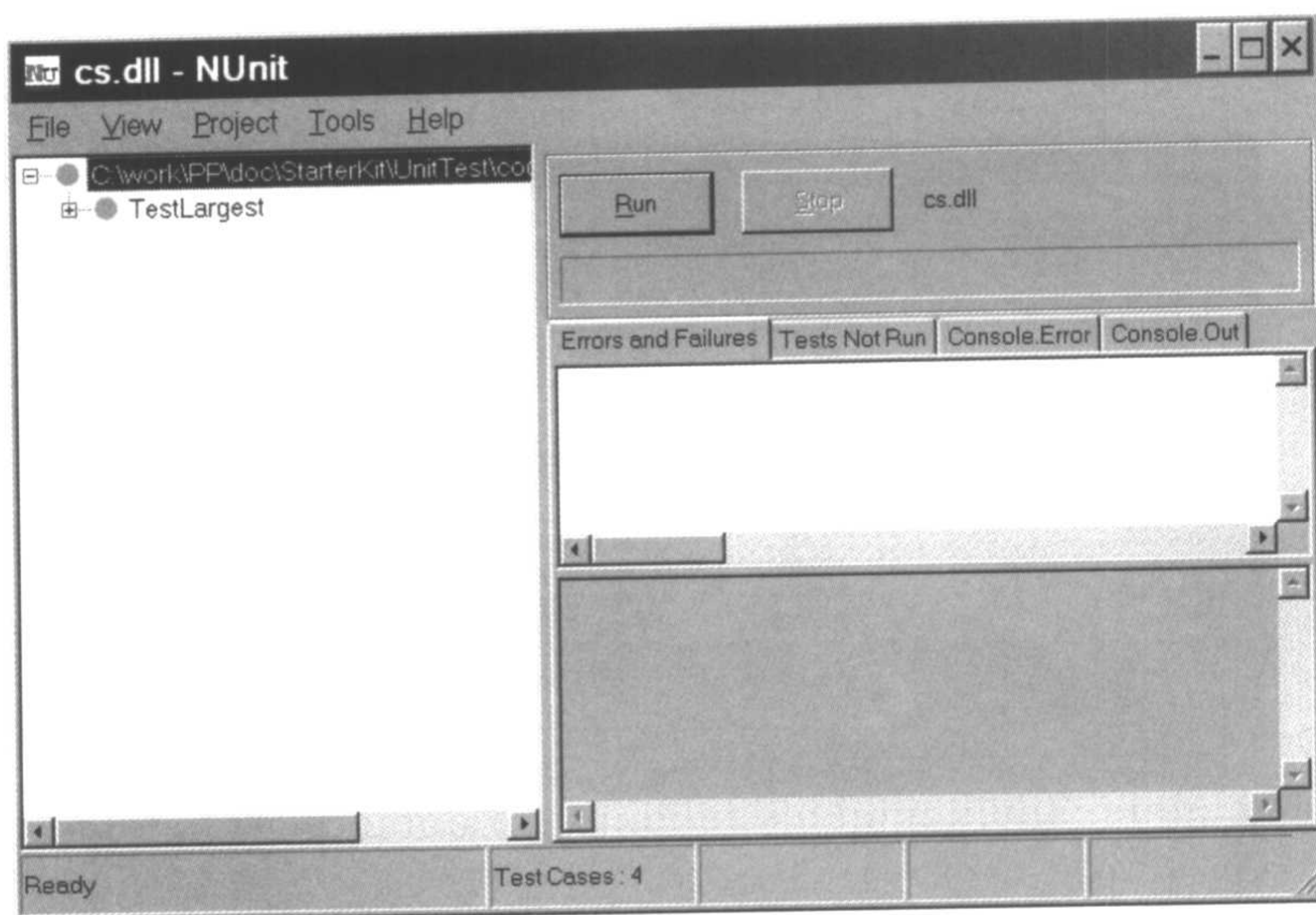


图2.3 NUnit加载完全并且就绪

■ NUnit GUI

我们可以在桌面图标或者开始菜单找到 NUnit GUI。当 GUI 出现之后，你有多个选项。你可以创建一个新的 NUnit 项目，如 19 页的图 2.2 所示；然后定位到你的源码目录，并且创建 NUnit 项目文件。然后在“Project”菜单下，添加程序集或者 Visual Studio 项目到你的 NUnit 项目中³。

另一种方法，你可以直接 Open 一个程序集（一个 *.dll* 文件）。在图 2.3 中，我们已经从 *dll* 文件直接加载测试了。现在只要点击“Run”按钮就可以进行测试。

当你运行一个选定的测试时，GUI 将会显示一个大的、有颜色的状态栏。如果所有的测试都通过，状态将会显示开心的绿色；如果任何测试都失败，状态栏将会变成生气的红色。如果状态栏呈现警戒黄色的话，那意味着一些测试被跳过了（具体将在后面说明）。

³ 可以使用菜单 Tool/Options 中的一个个性化设置来打开 Visual Studio 的支持。

■ NUnit 命令行

也可以从命令行来运行 NUnit；当把项目构建和测试自动化时，这将带来极大的便利。你只需要把 NUnit *bin* 目录（也就是说，你安装 NUnit 的路径加上 “\bin”）添加到你的 path 环境变量即可。

就目前的 shell 而言，你可以在命令行设置 path 变量，如下所示：

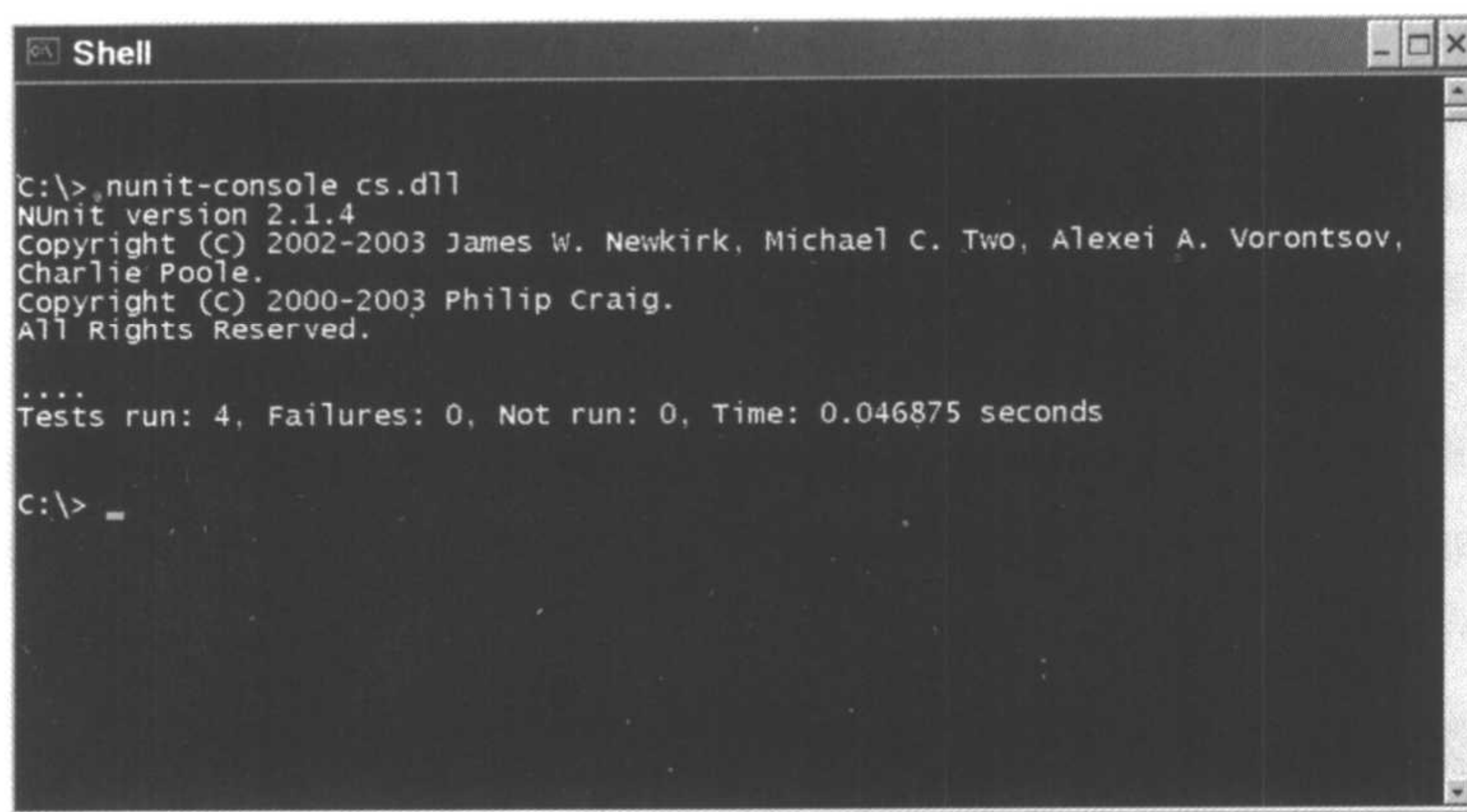
```
C:\> set "PATH=%PATH%;C:\Program Files\Nunit V2.2\bin"
```

更长久的做法可以通过访问下面路径：*Control Panel/System/Advanced/Environment Variable*，然后添加 NUnit 的 *bin* 目录到 Path 变量中（见图 2.4）。



图2.4 添加到Windows系统路径中

为了能够从命令行运行，需要输入命令 `nunitconsole`，然后是 NUnit 的项目文件或者 `dll`。你将会看到类似于图 2.5 的输出信息。



```
C:\>nunit-console cs.dll
NUnit version 2.1.4
Copyright (C) 2002-2003 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov,
Charlie Poole.
Copyright (C) 2000-2003 Philip Craig.
All Rights Reserved.

....
Tests run: 4, Failures: 0, Not run: 0, Time: 0.046875 seconds

C:\> _
```

图2.5 NUnit命令行用法

■ Visual Studio 的 NUnit 插件

最后，还有几种能把 NUnit 集成到 Visual Studio 的插件⁴。`nunit-addin` 添加了这种只要右击源码并且选择“Run Test(s)”就能运行任何测试的能力；而测试的输出会显示在 Visual Studio 的 Output 面板中。其他类似的项目插件添加了可视化的测试报告，以及其他的一些特性。

2.4 运行例子

你现在应该可以运行首个测试了。



在继续阅读之前，请先运行这个单元测试……

⁴ 诸如：<http://www.mutantdesign.co.uk/nunit-addin/>。

运行上面的代码之后，你可能会看到类似下面的错误信息：

```
Failures:
1) TestLargest.LargestOf3 :
    expected:<9>
    but was:<2147483647>
    at TestLargest.LargestOf3() in c:\testlargest.cs:line 6
```

Whoops! 这和我的期望并不一致。它为什么会返回一个那么大的数字，而不是9呢？这个大数又是从哪里来的呢？它看起来有点像 `int` 类型的最大值...oh,我在第13行的地方敲错了一点代码：`max=Int32.MaxValue`；该行代码应该是：`max=0`。我们的想法是初始化 `max` 为一个这样的值，任何一个其他的数字和 `max` 比较大小之后，马上就可以代替 `max`。让我们修改前面的代码，重新编译，并测试它，最后确认它能够通过测试。

接下来我们将考虑：最大数如果位于 `list` 的不同位置——第1个，最后一个，或者中间，那么将会出现什么情况。显然，bug 经常会出现于边界值。在这个例子中，边界值是指：`list` 的首个位置和最后一个位置。在下面的代码中，我们将把这3个断言都写在一个测试中，但我们每次只添加一个断言。对于最大值位于中间的情况，我们已经有了一个用例了：

```
using NUnit.Framework;
[TestFixture]
public class TestLargest {
    [Test]
    public void LargestOf3() {
        Assert.AreEqual(9, Cmp.Largest(new int[] {8,9,7}));
    }
}
```

TestLargest.cs

请输入上面的这些测试代码，并且运行它。Hey，它通过了！现在让9位于中间（我们接下来需要在上面的 `LargestOf3()` 里再添加一个断言）：

```
[Test]
public void LargestOf3() {
    Assert.AreEqual(9, Cmp.Largest(new int[] {9,8,7}));
    Assert.AreEqual(9, Cmp.Largest(new int[] {8,9,7}));
}
```

TestLargest.cs

它又一次通过了。为了考虑测试的完整性，让我们再编写一个有意思的断言：



```
[Test]
public void LargestOf3() {
    Assert.AreEqual(9, Cmp.Largest(new int[] {9,8,7}));
    Assert.AreEqual(9, Cmp.Largest(new int[] {8,9,7}));
    Assert.AreEqual(9, Cmp.Largest(new int[] {7,8,9}));
}
```

在继续阅读之前，请先运行上面的代码……

Failures:

```
1) TestLargest.LargestOf3 :
    expected:<9>
    but was:<8>
    at TestLargest.LargestOf3() in c:\testlargest.cs:line 5
```

在最后一个测试中，为什么函数返回的最大值是 8 呢？好像代码根本就没有考虑 list 中的最后一个元素似的。有了这个想法之后，我们又可以很确信地察觉到代码中存在另一个拼写错误：for 循环结束得太早了。这正是“off-by-one”错误的一个例子。在我们的代码中：

```
for (index = 0; index < list.Length-1; index++) {
```

但是实际上正确的代码应该是：

```
for (index = 0; index <= list.Length-1; index++) {
for (index = 0; index < list.Length; index++) {
```

在继承自 C 的语言中（例如 Java 和 C#），第 2 个表达式是比较常用的，但是你同时看到，它更加容易导致 off-by-one 错误。现在，让我们修改前面的代码，然后运行测试。但是仔细考虑这类 bug 之后，同时也在告诉你一个经验教训：在这里使用迭代器（使用 C# 的 foreach 语句）将会更好。那样的话，你就可以避免这种 off-by-one 错误了。

现在，你还可以检查重复的最大值：也就是编写并且运行下面的代码（从现在开始，我们只是给出新加的方法，而不会显示前面的代码）：

```
[Test]
public void TestDups() {
    Assert.AreEqual(9, Cmp.Largest(new int[] {9,7,9,8}));
}
```

到现在为止，一切都和我们的期望一致。接下来，我们需要只有一个元素的 list：

```
[Test]
public void TestOne() {
    Assert.AreEqual(1, Cmp.Largest(new int[] {1}));
}
```

Hey, 它通过了! 你正在朝正确的方向前进, 并且确信我们在这个例子中所发现的 bug 都已经被除去了。但是, 等一下, 我们还没有检查都是负值的 list:

```
[Test]
public void TestNegative() {
    int [] negList = new int[] {-9, -8, -7};
    Assert.AreEqual(-7, Cmp.Largest(negList));
}
```

TestLargest.cs

在继续阅读之前, 请先运行上面的代码……

Failures:

```
1) TestLargest.TestNegative :
    expected:<-7>
    but was:<0>
    at TestLargest.TestNegative() in c:\testlargest.cs:line 4
```

Whoops! 这个 0 是从哪里来的呢?

看来用 0 来初始化 max 是个错误的做法; 因为我们实际上需要的初始 max 值应该是 MIN_VALUE, 从而才能比所有的负数都小:

```
max = Int32.MinValue
```

做了这个改正之后, 让我们再次运行这个测试——前面的所有测试都应该能够通过, 而且该测试现在也能通过了。

遗憾的是, 针对方法 “largest” 的初始规范是不完整的, 因为它并没有说明: 在数组为空的时候, 会出现什么情况。我们认为这是一个错误; 并且在函数的上面添加了一些代码; 也就是说, 这些代码会对这种前提条件进行判断, 在 list 的长度为 0 的时候, 会抛出一个运行期异常:

```
public static int Largest(int[] list) {
    int index, max=Int32.MinValue;
    if (list.Length == 0) {
        throw new ArgumentException( "largest: Empty list");
    }
    // ...
```

Largest.cs

在完成了上面的这些测试以及对测试进行了深入思考之后, 我们意识到需要一个设计上的改动。实际上, 这已经不是什么稀奇的事情, 而是我们要特别强调的一点。因此, 对于最后一个测试, 在传递一个空数组给程序时, 我们需要检查是否会抛出一个异常。而针对异常进行测试的话题, 我们将留在第 41 页讨论; 但是现在, 你可以学着下面的代码照葫芦画瓢:



```
[Test, ExpectedException(typeof(ArgumentException))]  
public void TestEmpty() {  
    Cmp.Largest(new int[] {});  
}
```

最后，还有一点要提醒的是：所有的代码——测试的和产品的——都应该是简洁和清晰的。尤其是测试代码，更应该具有很好的可读性，哪怕性能差一些或者形式上啰嗦一些。

2.5 更多的测试

在这一章中，我们从一个非常简单的方法开始，然后再加上一些有趣的而且确实查出了 bug 的测试。注意我们并没有胡乱地编写测试，或者盲目地尝试每个可能的数字组合；而是选择一些可能暴露问题的有趣例子。但另一方面，这些测试就是你能想到的所有测试了吗？

是否还存在其他合适的测试呢？

由于在接下来的章节中我们将一直关注测试，所以现在或许需要一种辨析代码的好方法，来帮助我们能够更可靠、更有规律地想出好的测试项目。我们将把这个问题留到第4章讨论，在第3章中，我们先深入介绍如何使用 NUnit。

第 3 章

使用 NUnit 编写测试

Writing Tests in NUnit

在上一章中，我们已经简单介绍了如何编写测试；现在是深入介绍测试代码和产品代码之间的差异的时候了，看看各种形式的 NUnit 断言（assertions），以及 NUnit 测试的结构和组成等等。

3.1 构建单元测试

如果你有一个名为 `CreateAccount` 的被测试函数，那么你的第一个测试函数的名称也许就是 `CreateSimpleAccount` 什么的。其中，方法 `CreateSimpleAccount` 以恰当的参数调用 `CreateAccount` 并验证 `CreateAccount` 的行为是否和它宣称的一样。当然你也可以有许多测试方法来执行 `CreateAccount`（毕竟，不是所有的 `account` 都是简单的）。

图 3.1 展示了这两块代码之间的关系。

测试代码仅限于我们内部使用。客户或者最终用户永远都不会看到，更不会使用这些代码。因此，产品代码——也就是最后要发布给客户或者放入产品中的代码——对测试代码是一无所知的。产品代码最后将撇下测试代码独自闯入一个寒冷的世界（这通常意味着测试代码在另一个项目里，在它自己的程序集中）。

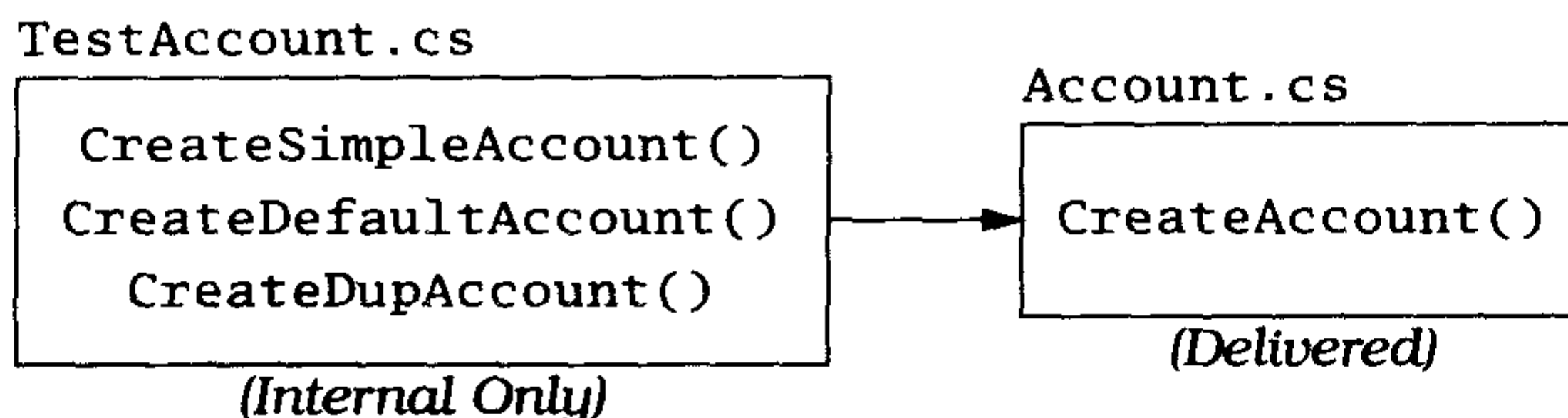


图3.1 测试代码和产品代码

测试代码必须要做以下这几件事情：

- 准备测试所需要的各种条件（创建所有必须的对象，分配必要的资源等等）。
- 调用要测试的方法。
- 验证被测试方法的行为和期望是否一致。
- 完成后清理各种资源。

对于测试代码，你也是用一般的方式编写和编译，这和项目中普通源码是一样的。测试代码可能偶尔会用到某些额外的程序库，但是除此之外，测试代码并没有任何特别之处——它们也只是普通代码而已。

当执行测试代码的时候，请记住你从来不直接运行产品代码；至少，并非像一个普通用户那样。而是借助于测试代码，让它根据小心控制的条件来执行产品代码。

现在，尽管你“能够”从头编写所有的测试，但是这种做法是极端低效的。因此，在本书的剩余内容中，我们假定你在使用 NUnit 框架。更具体地说，我们会在例子中使用 C# 语言给出针对 NUnit2.2 的特定的方法调用和类。不同版本之间可能会有些细微的差异，但基本概念在所有的版本中都是一样的，甚至针对其他语言或者环境的测试框架也是一样的。

3.2 NUnit 的各种断言

如我们前面所看到的一样，NUnit 提供了一些辅助函数用于帮助你确定某个被测试函数是否工作正常。通常而言，我们把所有这些函数统称为断言。它让你可以确定：某条件是否为真；两个数据是否相等，或者不等，或者其他的一些情况。在接下来的内容中，我们将逐个介绍 NUnit 提供的每一个断言（assert）方法。

下面每个方法都会记录是否失败了（断言为假）或者有错误了（遇到一个意料外的异常）的情况，并且通过 NUnit 的一些类来报告这些结果。对于命令行版本的 NUnit 而言，这意味着将会在命令行控制台上显示一些错误消息。对于 GUI 版本的 NUnit 而言，如果出现失败或者错误，将会显示一个红色的条和一些用于对失败进行详细说明的辅助消息。

当一个失败或者错误出现的时候，当前测试方法的执行流程将会被中止，但是（位于同一个 test fixture 中的）其他测试将会继续运行。

断言是单元测试最基本的组成部分。因此，NUnit 程序库以 Assert 类的静态方法的形式提供了不同形式的多种断言。

■ AreEquals

```
Assert.AreEqual(expected, actual [, string message])
```

这是使用得最多的断言形式。在上面的参数中，*expected* 是你的期望值（通常都是硬编码的），*actual* 是被测试代码实际产生的值，*message* 是一个可选的消息，如果提供的话，将会在发生错误的时候报告这个消息。当然，你完全可以不提供这个 *message* 参数，而只提供 *expected* 和 *value* 这两个值。

任何对象都可以拿来作相等性测试：适当的相等性判断方法会被用来做这样的比较。譬如，我们可能会使用这个方法比较两个字符串的内容是否相等。此外，对于原生类型（int, decimal 等）和 Object 类型也提供了不同的函数签名。

计算机并不能精确地表示所有的浮点数，通常都会有一些偏差。因此，如果你想用断言来比较浮点数（在 C# 中，是类型为 float 或者 double 的数），则需要指定一个额外的误差参数。它表明你需要多接近才能认为两数“相

等”。

```
Assert.AreEqual(expected,
                actual,
                tolerance [, string message])
```

对于商业程序而言，只要精确到小数点的后 4 位或者后 5 位就足够了。对于进行科学计算的程序而言，则可能需要更高的精度。

例如，下面的断言将会检查实际的计算结果是否等于 3.33，但是该检查只精确到小数点的后两位：

```
Assert.AreEqual(3.33, 10.0/3.0, 0.01, "Wanted 3 1/3");
```

■ IsNull

```
Assert.IsNull(object [, string message])
Assert.IsNotNull(object [, string message])
```

验证一个给定的对象是否为 null（或者为非 null），如果答案为否，则将会失败。message 参数是可选的。

■ AreSame

```
Assert.AreSame(expected, actual [, string message])
```

验证 *expected* 参数和 *actual* 参数所引用的是否为同一个对象，如果不是的话，将会失败。message 参数是可选的。

■ IsTrue

```
Assert.IsTrue(bool condition [, string message])
```

验证给定的二元条件是否为真，如果为假的话，将会失败。message 参数是可选的。

如果你发现测试代码像下面这样，宛如废话一般：

```
Assert.IsTrue(true);
```

那么你就该好好想想这些代码了。对于这种写法，除非是被用于确认某个分支，或者异常逻辑才有可能是正确的选择；否则的话，很可能就是一个糟糕的主意。显然，你怎么都不会愿意在一页代码中看到只在该页的末尾出现许多 `Assert.IsTrue(true)` 语句（也就是说，只是为了确认代码能够运行到末尾，没有中途死掉，并以为它就必然工作正常了）。这哪里是测试，这简直就是一厢情愿的幻想。

除了测试条件为真之外，你也可以测试条件是否为假：

```
Assert.IsFalse(bool condition [, string message])
```

上面代码用于验证给定的二元条件是否为假；如果不是的话（为真），该测试将会失败，message 参数是可选的。

■ Fail

```
Assert.Fail([string message])
```

上面的断言将会使测试立即失败，其中 message 参数是可选的。这种断言通常被用于标记某个不应该被到达的分支，但它在实际中并不常用。

■ 使用断言

一般而言，一个测试方法会包含有多个断言，因为你需要验证该方法的多个方面以及内在的多种联系。当一个断言失败的时候，该测试方法将会被中止——从而导致该方法中剩余的断言这次就无法执行了。此时你不能有别的想法，而只能是在继续测试之前先修复这个失败的测试。依此类推，你不断地修复一个又一个的测试，沿着这条路径慢慢前进。

你应当期望所有的测试在任何时候都能够通过。在实践中，这意味着当你引入一个 bug 的时候，只有一到两个测试会失败。在这种情况下，把问题分离出来将会相当容易。

当有测试失败的时候，无论如何都不能给原有代码再添加新的特性！此时你应该尽快地修复这个错误，直到让所有的测试都能顺利通过。

为了遵行上面的这种原则，你需要一种能够运行所有测试——或者一组测试、某个特殊子系统等等——的辅助方法。

3.3 NUnit 框架

到目前为止，我们只是介绍了断言方法本身。显然，你不能只是简单地把断言方法写到源文件里面，然后就希望它这就能运行起来。你需要一个框架，那就要比这多做一些工作了。幸运的是，也不会多做太多。

下面是一段简单的测试代码，它展示了开始使用该框架的最小要求：

```

Line 1  using NUnit.Framework;
-       [TestFixture]
-       public class TestSimple {
-           [Test]
5         public void LargestOf3() {
-             Assert.AreEqual(9, Cmp.Largest(new int[] {8,9,7}));
-         }
-     }

```

尽管上面的代码非常清楚，但还是让我们来看看这段代码的每一部分。

首先，第 1 行的 using 声明引入了必需的 NUnit 类。NUnit framework 提供了我们所需要的单元测试的功能，包括所有我们在前面描述的断言方法。（记住，为了让上面的 using 语句能够工作，你需要在你的项目添加一个指向 NUnit DLL 的引用）

接下来，我们在第 3 行定义了一个类：每个包含测试的类都必须如上用 [TestFixture] attribute 标记。而且这个类必须声明为 public（只有这样运行测试的程序才可以找到它），而且它必须有一个 public 的、没有参数的构造函数（默认构造函数就行了）。

最后，测试类包含了用 [Test] attribute 标记的方法。在这个例子中，我们在第 5 行写了一个名为 LargestOf3 的方法。而所有用 [Test] attribute 标记的 public 方法都会被 NUnit 自动运行。

在上面的例子中我们展示了一个测试，它只有一个测试方法，而这个测试方法中又仅有一个断言。当然，在测试方法中，我们是可以写多个断言的，像这样：

```

using NUnit.Framework;
[TestFixture]
public class TestSimple {
    [Test]
    public void LargestOf3() {
        Assert.AreEqual(9, Cmp.Largest(new int[] {8,9,7}));
        Assert.AreEqual(100, Cmp.Largest(new int[] {100,4,25}));
        Assert.AreEqual(64, Cmp.Largest(new int[] {1,64,38}));
    }
}

```

TestSimple.cs

TestSimple.cs

在此，我们在一个测试方法里面使用了 3 个 `AreEqual` 断言。

3.4 NUnit 测试的组成

如我们之前所看到的一样，一个 `test fixture`（也就是一个用 `[TestFixture]` attribute 标记的类）包含一个或多个测试方法：每个方法包含一个或多个断言。一个程序集(assembly)可以包含多个 `test fixture`。

我们通常可以通过给 `test runner`（运行测试的程序）指定程序集来运行它所包含的所有测试。你还可选择使用 NUnit 的命令行或者 GUI 来指定运行程序集中的哪个 `test fixture`。

在 GUI 中，只要通过点击，你就可以单选中一个测试，或者单选中一个 `test fixture`，或者选中整个程序集；然后这些测试会被执行。

在命令行中，你可以如下指定一个程序集或者一个特定的 `test fixture`：

```
c:\> nunit-console assemblyname.dll /fixture: ClassName
```

有了这种灵活性之后，你可能需要思考另一个问题：如何把测试方法组织成一个独立的程序集和 fixtures，从而让测试更加容易。

例如，你也许会想一次运行所有跟数据库相关的测试，或者所有 Fred 编写的测试（上一个项目时 Fred 才刚来，目前仍然是处于试用期，所以你想对他看紧点）。

但是假设你在一个 `fixture` 中已经有了与数据库相关的测试，而 Fred 的测试位于另一个 `fixture`，那么你可能希望一起运行这两个 `fixture`。如果每次都要手工选择两个 `fixture`，显然是很麻烦的（不管是使用命令行还是 GUI），而且你也不希望 Fred 把所有的测试都写到你的数据库 `fixture` 中。

幸运的是，你可以把现存的多个 `test fixture` 组合进一个 `test suite` 中，一个 `test suite` 是一些 `TestFixture` 类的集合。任何测试类都可以包含一个用 `[Suite]` attribute 标记的静态方法。这个方法将返回一个 `TestSuite`，它就是一个 `TestFixture` 类集合。

```

using NUnit.Framework;
using NUnit.Core; // Requires nunit.core.dll
[TestFixture]
public class TestClassSuite {
    [Suite]
    public static TestSuite Suite {
        get {
            TestSuite suite = new TestSuite( "Name of Suite");
            suite.Add(new DatabaseTests());
            suite.Add(new FredsTests());
            // add others...
            return suite;
        }
    }
}

```

你添加到 suite 的类 (DatabaseTests, FredsTest 等) 是一些标记有 [TestFixture] 的类。在每个类中, 仍然是标记有 [Test] 特性的方法; 而框架将会自动地运行这些方法。suite 功能并不会妨碍你在需要的时候运行单独的 fixture。他们让你能给这些 fixture 分组: 通过运行 Suite(), 你就可以一次运行一堆 text fixture 了。

Suite 是层次化组织测试的一个很有用的机制。特别是对于无人照料的构建, 用它来调整测试集合是非常方便的。

但是对于日常的测试, 它却不那么有用了。你可能会希望能够把所有的测试分开成好几部分, 然后基于一个你以前编写过的大测试集合中, 选择并执行各种测试。

幸运的是, NUnit 提供了另一种机制, 你可以用它来对每个单独的测试方法和 test fixture 分门别类。

■ 分类 (Categories)

NUnit 用 category 的概念提供了标记和运行一个个单独的测试和 fixture 的简单方法。一个 category 是指你定义的一个名字。你可以把不同的测试方法关联到一个或多个 category, 然后在运行测试的时候, 选择你想要运行的 category。

假定你有一个测试, 它使用了一个用于解决旅行商最短路径问题的高效算法——关于该算法, 是指旅行商 Bob 要访问 n 个城市, 如何才能使花费的时间最少。有趣的是: 在城市不多的情况下, 该旅行商算法能够工作得很好; 但是该算法是指数复杂度的——也就是说, 如果增加到几百个城市, 那么使

用该算法的程序可能需要运行 20 000 年才能计算出答案；即使是 50 个城市可能也需要好几个小时的运行时间。因此，在默认情况下，你可能并不想包含这些测试。

你可以使用 NUnit 的 `category` 来把运行时间较短的测试和运行时间较长的测试分开。显然，前一种测试将经常被运行，后一种则可能只是在夜晚构建的时候才运行一次。

`category` 是以 `attribute` 的形式来指定的。当声明方法的时候，你可以提供一个字符串来指定它的 `category`。然后，在你运行测试的时候，可以指定你要运行哪些 `category`（你可以一次指定运行多个 `category`）。

例如，假设你具有许多只需要几秒钟就能完成的方法，但是另有一个方法却要花费很长的时间才能运行完成。你可以通过使用分类名“Short”和“Long”来标记它们（如果你希望对 Fred 盯紧点，你甚至可以创建一个名为 Fred 的 `category`）。

```

Line 1  using NUnit.Framework;
-       using NUnit.Core;
-
-       [TestFixture]
5       public class TestShortestPath {
-
-           [Test]
-           [Category( "Short")]
-           public void Use5Cities() {
10          TSP tsp = new TSP(); // Load with default cities
-           Assert.AreEqual(140, tsp.ShortestPath(5));
-       }
-
-           [Test, Category( "Short")] // Can specify either way
15          public void Use10Cities() {
-           TSP tsp = new TSP(); // Load with default cities
-           Assert.AreEqual(586, tsp.ShortestPath(10));
-       }
-
20          // This one takes a while...
-           [Test]
-           [Category( "Long")]
-           public void Use50Cities() {
-           TSP tsp = new TSP(); // Load with default cities
25          Assert.AreEqual(2300, tsp.ShortestPath(50));
-       }
-   }

```

TestShortestPath.cs

你可以为分开的 2 行指定多个特性(这个例子是指: `Test` 和 `Category`)，如第 8 行和第 9 行；或者把它们组合到 1 行，如第 14 行。

现在，如果你选择只运行“Short”方法，那么方法 `Use2Cities` 和 `Use10Cities` 将会被运行。而如果你选择“Long”方法，那么只有 `Use50Cities` 会被选择运行。你还可以同时选择这两个 categories 来运行所有三个方法。

在 GUI 中，你在如图 3.2 中的标签页中选择你要运行的 Categories。只选择那些感兴趣的，然后点击 **ADD** 按钮即可。

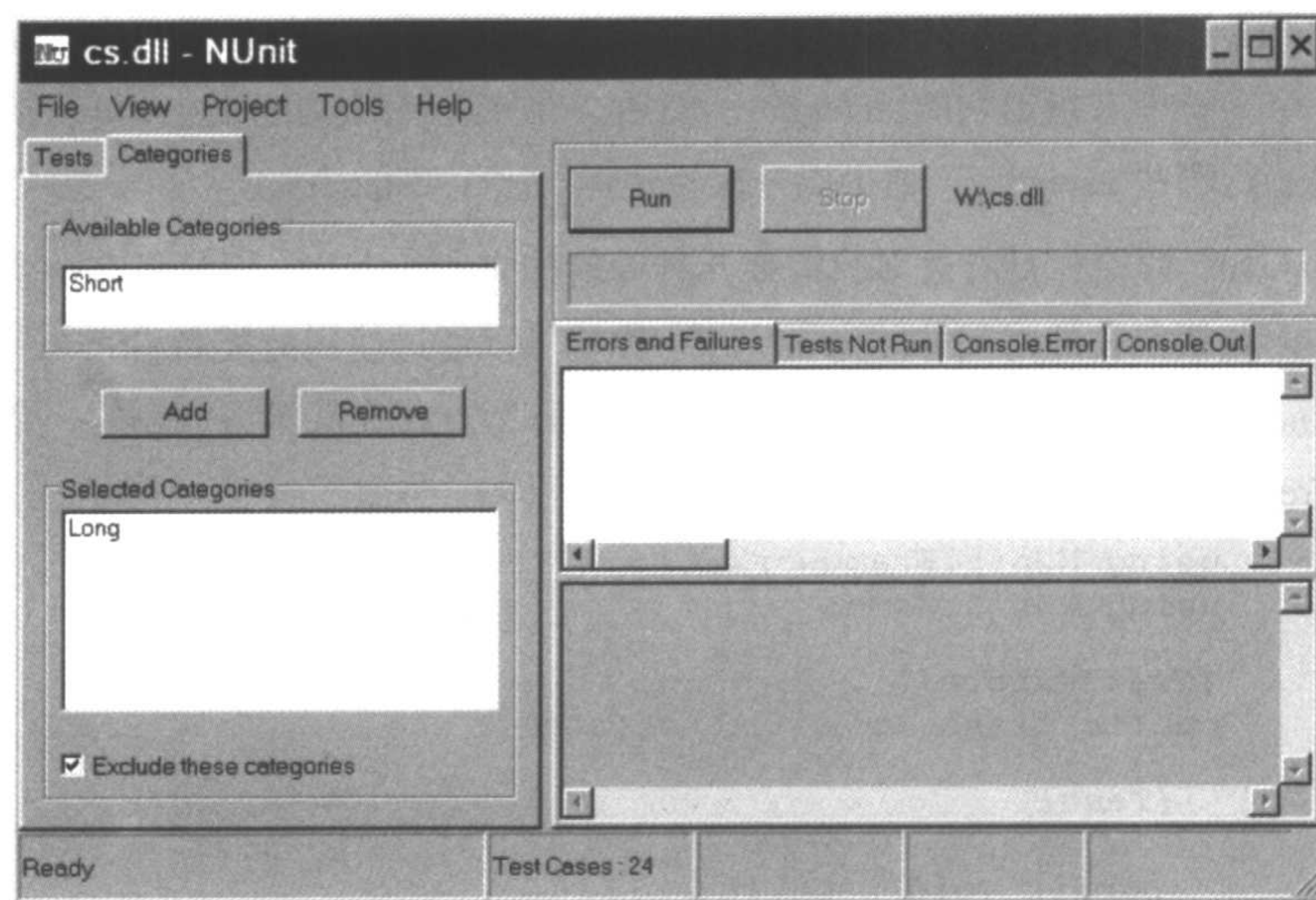


图3.2 NUnit Category的选择

在命令行中，你同样可以单独指定要包含的 categories。只需要添加如下的参数：

```
/includes=string;string,...
```

注意，多个 category 名之间是用分号（“;”）隔开的。

你还可以选择排除办法，排除一些 category，从而其他的 category 都会得到运行。在 GUI 中，有一个选项框来做这项工作；而命令行选项则是

比较奇怪，使用 `/exclude` 参数。

但我们还可能会有其他的需求：当没有选择任何 `category` 的时候，我们希望某些 `category` 的测试能够被运行；而当显式选择了一些 `category` 的时候，则只有被选择的 `category` 才会被运行。

为了支持这一点，你可以指定 `[Category] attribute` 的 `Explicit` 属性为真：

```
[Category("SpecialEquipmentNeeded", Explicit=true)]
```

在没有指定任何 `category` 的情况下，这种语法将会排除该 `category` 被运行。默认情况下，当没有指定 `category` 的时候，你的运行将会包含所有没有显式指定 `Explicit`（即属性值为 `false`）的 `category`。然而，如果 GUI 或者命令行已经指定了一个 `category`，那么只有这个指定的 `category` 才会被运行。

除了标记一个测试方法属于一个 `category` 之外，你还可以标记整个 `fixture` 属于一个 `category`。例如，如果你想把整个 `test fixture` 标记为 `long-running`（这样也就没有必要重复标记每个测试方法了），你可以编码如下：

```
Line 1  using NUnit.Framework;
        using NUnit.Core;
        -
        - [TestFixture, Category("Long")]
5       public class TestShortestPath2 {
        -
        -     [Test]
        -     public void Use50Cities() {
        -         TSP tsp = new TSP(); // Load with default cities
10        Assert.AreEqual(2300, tsp.ShortestPath(50));
        -     }
        -
        -     [Test]
        -     public void Use100Cities() {
15        TSP tsp = new TSP(); // Load with default cities
        -         Assert.AreEqual(4675, tsp.ShortestPath(100));
        -     }
        -
        -     [Test]
20        public void Use150Cities() {
        -         TSP tsp = new TSP(); // Load with default cities
        -         Assert.AreEqual(5357, tsp.ShortestPath(150));
        -     }
        -
25    }
```

TestShortestPath2.cs

现在你可以通过使用一个 `category` 名快速地排除整个 `fixture` 了。

■ Per-method 的 Setup 和 Teardown

每个测试的运行都应该是互相独立的；从而你就可以在任何时候，以任意的顺序运行每个单独的测试。

为了获得这样的好处，在每个测试开始之前，你都需要重新设置某些测试环境，或者在测试完成之后，你需要释放一些资源。借助于一些 Attribute，NUnit 让你指定两种方法，分别用于环境的建立和清理：

```
[SetUp]
public void MySetup() {
    ...
}

[TearDown]
public void MyTeardown() {
    ...
}
```

在上面例子中，在调用每个[Test]方法之前，调用方法 MySetup()；并且在每个测试方法完成之后，调用方法 MyTeardown()。

例如，假设对于每个测试，你都需要某种数据库连接。这时，你就不需要在每个测试方法中重复建立连接和释放连接了，而只须在 SetUp 和 TearDown 方法中分别建立和释放连接：

```
[TestFixture]
public class TestDB {
    private Connection dbConn;

    [SetUp]
    public void MySetup() {
        dbConn = new Connection( "oracle", 1521, user, pw);
        dbConn.Connect();
    }

    [TearDown]
    public void MyTeardown() {
        dbConn.Disconnect();
        dbConn = null;
    }

    [Test]
    public void TestAccountAccess() {
        // Uses dbConn
        XXX XXX XXXXXX XXX XXXXXXXXXXXX;
        XX XXX XXX XXXX X XX XXXX;
    }

    [Test]
    public void TestEmployeeAccess() {
        // Uses dbConn
        XXX XXX XXXXXX XXX XXXXXXXXXXXX;
        XXXX X X XX XXX XX XXXX;
    }
}
```

TestDB.cs

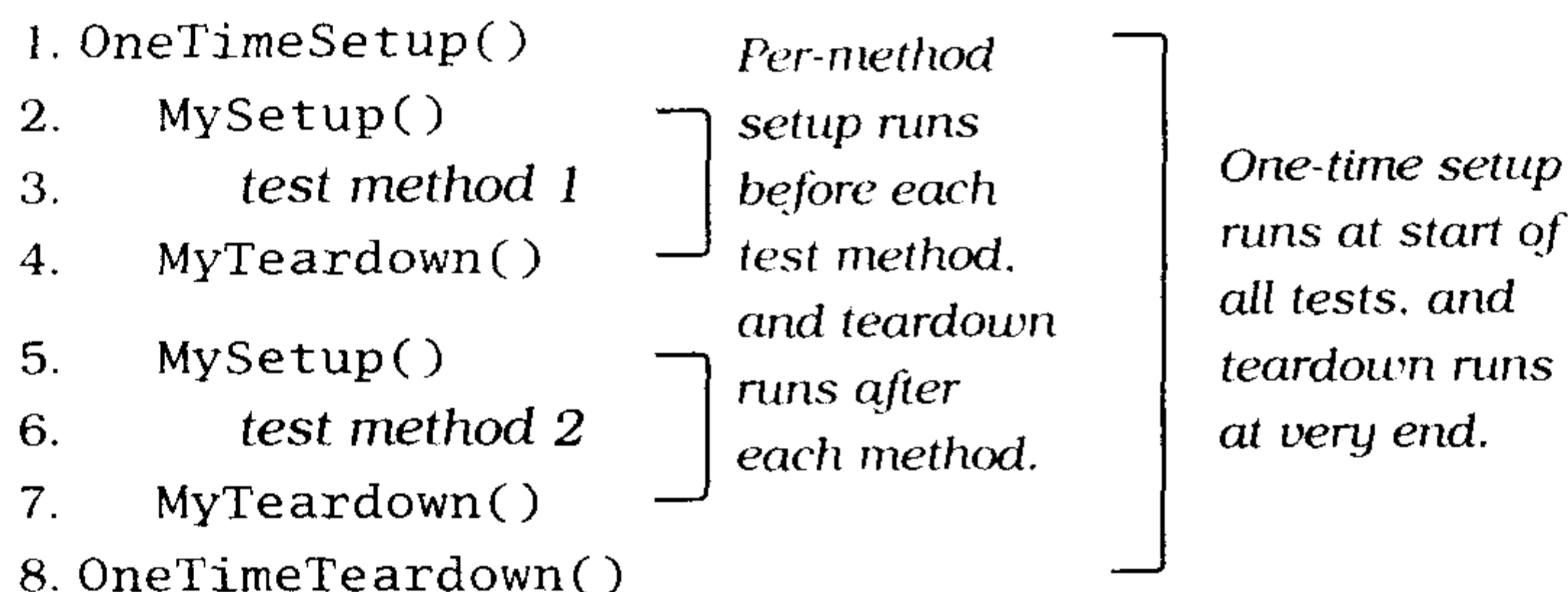


图3.3 代码的执行顺序

在上面的例子中，调用 `TestAccountAccess()` 之前，将会先调用 `MySetup()`；然后在 `TestAccountAccess()` 完成之后，会接着调用 `MyTeardown()`。在第 2 个测试函数 `TestEmployeeAccess()` 中，也是按顺序先调用 `MySetup()`，再调用该函数，最后调用 `MyTeardown()`。

■ Per-class Setup 和 Per-class Teardown

一般而言，你只须针对每个方法设置运行环境；但是在某些情况下，你须要对整个 test class 设置一些环境，以及在类中的所有方法都执行完成后做一些清理工作。要达到这种效果，你需要 per-class setup 和 per-class teardown（就执行顺序而言，per-test 和 per-class 之间的区别可以参见图 3.3）。你只须要在声明 setup 方法的时候标记如下的 attribute：

```
[TestFixtureSetup]
public void OneTimeSetup() {
    ...
}

[TestFixtureTearDown]
public void OneTimeTeardown() {
    ...
}
```

现在你就可以在同一个类中同时使用 per-class 和 per-test 的方法了。

3.5 自定义 NUnit 断言

通常而言，NUnit 所提供的标准断言对大多数测试已经足够了。然而，在某些环境下，譬如你要处理一个特殊的数据类型，或者处理对多个测试都共享的一系列操作，那么如果有自定义的断言，将会更加方便。

最糟糕的做法就是将一段测试代码一次又一次地复制过来使用。

在你的测试代码中，拷贝和粘贴公有代码同样会是一个毁灭性的灾难。测试代码的质量应该和产品代码一样。也就是说，你在编写测试代码的时候，也应该维持好的编码原则，诸如 DRY 原则¹、解耦合原则、正交性原则等等。因此，你需要把一些共同的测试代码抽取到方法中，并且在你的测试用例中使用这些方法。

不要害怕编写具有自定义风格的断言方法。例如，假设你要测试一个财务应用程序，并且假设所有的测试都会使用一种数据类型 `Money`：

```
using System;
using NUnit.Framework;

public class MoneyAssert {
    /**
     * Assert that the amount of money is an even
     * number of dollars (no cents)
     */
    public static void AssertNoCents(Money amount,
                                     String message) {
        Assert.AreEqual(
            amount.AsDecimal(),
            Decimal.Truncate(amount.AsDecimal()),
            message);
    }
    /**
     * Assert that the amount of money is an even
     * number of dollars (no cents)
     */
    public static void AssertNoCents(Money amount) {
        AssertNoCents(amount, "");
    }
}
```

MoneyAssert.cs

¹ DRY 是“Don't Repeat Yourself”的缩写。他的基本技术要求：系统中的每个知识都必须有一个独立的、明确的、可信的表示。

在此，我们提供了两种形式的断言：一种接收一个 `string` 参数，另外一种则没有。注意，我们并没有拷贝和粘贴代码，而只是把第二个调用委托给了第一个。

现在，对于项目中需要测试 `Money` 类的其他测试类，都可以使用我们自定义的断言方法了：

```
using NUnit.Framework;
[TestFixture]
public class TestSomething {
    [Test]
    public void CountDeMonet() {
        Money m = new Money(42.00);
        m.Add(2);
        MoneyAssert.AssertNoCents(m);
    }
}
```

TestSomething.cs

3.6 NUnit 和异常

对于测试而言，下面两种异常是我们可能会感兴趣的：

1. 从测试代码抛出的可预测异常。
2. 由于某个模块（或代码）发生严重错误，而抛出的不可预测异常。

也许和你想象的正好相反，异常在这里是彻头彻尾的好东西——它们能够告诉我们什么东西出错了。有时，在一个测试中，我们需要被测试方法抛出一个异常。例如，有一个名为 `ImportList()` 的方法。如果传入参数是一个 `null list`，那么我们将希望该方法抛出一个 `ArgumentException` 异常。在这种情况下，我们就需要显式地测试这一点。

为了能够测试预期的异常，NUnit 提供了 `[ExpectedException]` attribute。

```
[TestFixture]
public class TestException {
    [Test, ExpectedException(typeof(ArgumentException))]
    public void TestForException() {
        WhitePages.ImportList(null);
        // Shouldn't get to here
    }
}
```

TestException.cs

测试方法现在被期望为抛出一个异常（在 `ImportList()` 的调用中）。于是，如果没有抛出异常，测试将会失败。如果异常如预料那样抛出了，那么测试将会通过。注意，一旦期望的异常抛出了，测试方法中剩余的代码就会被跳过。

通常而言,对于方法中每个被期望的异常,你都应该写一个专门的测试,来确认该方法在应该抛出异常的时候,确实会抛出异常。然而,这样虽然能够确认期望的异常,但是对于出乎意料的异常,应该怎么办呢?

NUnit 将会为你处理这件麻烦的事情。例如,假设你正在从文件中读取数据。此时,你并不需要自己捕获可能的 I/O 异常,而只需要让 NUnit 把这些异常交给测试框架来处理就可以了:

```
[Test]
public void TestData1() {
    StreamReader sr = new StreamReader("data.txt");
    XXX XXX XXXXXXX XXXXX XXXX;
}
```

实际上, NUnit 框架可以捕获任何异常,并且把它报告为一个错误,这些都不需要你的参与。更好的是, NUnit 不只是让一个断言失败,而是能够跟踪整个堆栈,并且报告 bug 的堆栈调用顺序;当你需要查找一个失败测试的原因时,这将非常有用。

3.7 临时忽略一些测试

通常而言,你都希望所有的测试在任何时候都能够顺利通过。但假设你之前想到一些测试,并且编写了这些测试;现在你正在编写能够通过测试的实现代码。那么这些还不具备实现代码的新测试未能通过,又该怎么办?

虽然你可以继续编写这些测试,但你现在却不能让测试框架运行这些测试。为了解决这个问题, NUnit 提供了一个 [Ignore] 属性:

```
[Test, Ignore( "Not ready to test this yet")]
public void TestSomething() {
    XXX XXX XXXXXXX XXXXX XXXX;
}
```

NUnit 将会告知测试框架:这个测试被跳过了(并且在 GUI 版本中给出黄色的状态栏,以防在后面遗忘)。

在其他的测试框架和语言中,你需要重命名该方法,或者把该方法进行注释。例如,当在 Java 中使用 JUnit 的时候,用 test 开头的方法(例如

`testSomething`) 将会作为测试运行, 你需要把该方法重命名为非 `test` 开头的方法; 然后直到你能够运行该测试方法的时候, 才把方法名称重新改回来。然而, 在任何语言中, 进行编译的代码都应该是很干净的; 如果你的代码还没有达到这个条件, 那么你就需要把不需要的部分注释掉。

无论如何, 你要避免养成忽略“失败的测试结果”的习惯。

现在, 你已经知道了如何编写测试。在接下来的内容中, 应该是我们进一步介绍找出哪些内容需要测试的时候了。

第 4 章

测试哪些内容：Right-BICEP

What to Test: The Right-BICEP

乍看一个方法或类，就要找出所有可能出问题的地方，也就是所有隐藏在深处的 bug，将会是非常困难的。对于经验丰富的人，他们通常都能够洞察出系统哪些方面最可能出现问题的地方，然后集中地对这些方面进行测试；但是如果不具备那么多经验的话，要想发现这些可能会出问题的 bug，将会是相当困难且令人沮丧的。但是另一方面，最终的用户通常都能够发现我们的 bug，让我们非常难堪，这同时给职业升迁带来极大的不利。因此，在 bug 挖掘方面，我们迫切需要一些指导，一些能够告诉我们哪些方面需要测试的指导。

让我们先介绍 6 个值得测试的具体部位，它们能够提高你的测试技巧。我们把这 6 个方面统称为：Right-BICEP：

- **Right**——结果是否正确？
- **B**——是否所有的边界条件都是正确的？
- **I**——能查一下反向关联吗？
- **C**——能用其他手段交叉检查一下结果吗？
- **E**——你是否可以强制错误条件发生？
- **P**——是否满足性能要求？

4.1 结果是否正确

Right BICEP

对于测试而言，首要的也是最明显的任务就是查看所期望的结果是否正确——验证结果。

我们在第2章已经介绍了一些简单的数据验证：判断一个方法的返回值是否为序列中的最大值。

通常这些都是简单的测试，甚至这类测试的一部分在需求说明中都已经指定了。如果文档中没有的话，那么你或许需要问问其他人员。总之，你必须能够最终回答这个关键问题。

如果代码能够运行正确，我要怎么才知道它是正确的呢

如果你不能很好地回答这个问题，那么编写代码——或者测试——完全就是在浪费时间。“且慢一下”你可能会这么说“上面这句话听起来并不那么‘敏捷’！试想，如果文档比较晦涩或者不完整的话，该怎么办呢？这是否意味着我们都不能编写代码，而必须等到文档都已经齐备且清楚时才能继续编写代码呢”？

不是，完全不是。如果文档真的还不明了，或者不完整的话，至少你总是可以自己发明出一些需求来。虽然从用户的角度来看，这些功能或许是不准确的，但是你现在就可以知道你编写的代码要做什么，从而你就能够回答上面的问题了。

当然，你必须安排一些来自用户的反馈以调整你的假设。在代码的整个生命期中，“正确”的定义可能会不断在变；但是无论如何，你至少需要确认代码所做的和你的期望是一致的。

■ 使用数据文件

对于许多有大量测试数据的测试，你可能会考虑用一个独立的数据文件来存储这些测试数据，然后让单元测试读取该文件。这并不困难——甚至并不需要使用XML¹文件。图4.1是TestLargest的一个版本，它从一个测试文件中读取所有的测试数据。

¹ 显然，这是一个笑话。在今天的项目中，几乎都会使用XML，难道不是吗？

```

using System;
using System.IO;
using System.Collections;
using NUnit.Framework;

[TestFixture]
public class TestLargestDataFile {

    /* 运行testdata.txt中的所有测试数据（并没有测试
     * 异常的情况）。如果有任何文件IO发生错误，
     * 我们将会获得一个错误。
     */

    [Test]
    public void TestFromFile() {
        String line;
        StreamReader rdr =
            new StreamReader( "../testdata.txt");
        while ((line = rdr.ReadLine()) != null) {
            if (line.StartsWith( "#")) { // 忽略注释
                continue;
            }
            String[] tokens = line.Split(null);
            // 读取期望值
            String val = tokens[0];
            int expected = Int32.Parse(val);
            // 添加Largest函数的参数
            ArrayList argument_list = new ArrayList();
            for (int i=1; i < tokens.Length; i++) {
                argument_list.Add(Int32.Parse(tokens[i]));
            }
            // 转化为原生数组
            int[] args = (int[])argument_list.ToArray(
                typeof(int));
            // 运行断言
            Assert.AreEqual(expected,
                Cmp.Largest(args));
        }
    }
}

```

图4.1 TestLARGESTDATAFILE：从一个文件中读取测试数据

数据文件的格式很简单：每行一些数字，其中第1个数字是期望的答案，剩余的数字就是要用来测试的参数。另外，我们使用井号（#）来表示所在行是注释，因此你可以在测试文件中添加一些有意义的注释或者描述。

测试文件的具体形式大概如下：

```
#
# Simple tests:
#
9 7 8 9
9 9 8 7
9 9 8 9
#
# Negative number tests:
#
-7 -7 -8 -9
-7 -8 -7 -8
-7 -9 -7 -8
#
# Mixture:
#
7 -9 -7 -8 7 6 4
9 -1 0 9 -7 4
#
# Boundary conditions:
#
1 1
0 0
2147483647 2147483647
-2147483648 -2147483648
```

testdata.txt

如果和上面的例子一样只有很少的东西要测。也许就不值得费这么大力了。但是假如我们面对的是一个很复杂的应用程序，而表格中有几百甚至几千个测试数据，那么测试文件就是一个很有吸引力的选择。

多注意一下测试数据。不管是文件中的还是代码中的测试数据，都很有可能是错误的。实际上，经验告诉我们，测试数据比代码更有可能是错的，特别是人工计算的，或者来自原有系统计算结果的测试数据（系统添加的新特性，可能故意导致了不同结果）。因此，当测试数据显示有错误发生的时候，你应该在怀疑代码前先对测试数据检查两三遍。

另外，还有一些值得考虑的：代码本身是否并没有测试任何异常的情况。要实现这个功能，需要怎么做呢？

一个原则是：对于验证被测方法是正确的这件事情，如果某些做法能够使它变得更加容易，那么就采纳它吧。

4.2 边界条件

在前面“求最大值”的例子中，我们发现了几个边界条件：最大值位于数组末尾，数组包含负数，或者数组为空等等。 Right **B** ICEP

找边界条件是做单元测试中最有价值的工作之一，因为 bug 一般就出现在边界上。一些需要你考虑的条件有：

- 完全伪造或者不一致的输入数据，例如一个名为“!*W:X\&Gi/w~>g/h#WQ@”的文件。
- 格式错误的数字，例如没有顶层域名的电子邮件地址，就像 fred@foobar 这样的。
- 空值或者不完整的值（如 0，0.0，“”和 null）。
- 一些与意料中的合理值相去甚远的数值。例如一个人的岁数为 10 000 岁。
- 如果要求的是一个不允许出现重复数值的 list，但是传入的是一个存在重复数值的 list。
- 如果要求的是一个有序 list，但是传入的是一个无序的 list；或者反之。例如，给一个要求排好序的算法传入一个未排序的 list——甚至一个反序的 list。
- 事情到达的次序是错误的，或者碰巧和期望的次序不一致。例如，在未登录系统之前，就尝试打印文档。

一个想到可能的边界条件的简单办法就是记住助记短语 CORRECT。对于其中的每一条，都应该想想它是否与存在于被测方法中的某个条件非常类似，而当这些条件被违反时，出现的又是什么情形：

- **Conformance**（一致性）——值是否和预期的一致。
- **Ordering**（顺序性）——值是否如应该的那样，是有序或者无序的。
- **Range**（区间性）——值是否位于合理的最小值和最大值之内。

- **Reference** (依赖性) ——代码是否引用了一些不在代码本身控制范围之内的外部资源。
- **Existence** (存在性) ——值是否存在 (例如, 是否是非 null, 非 0, 在一个集合中等等)。
- **Cardinality** (基数性) ——是否恰好有足够的值?
- **Time** (相对或者绝对的时间性) ——所有事情的发生是否是有序的? 是否是在正确的时刻? 是否恰好及时?

在下一章中, 我们将进一步介绍所有这些边界条件。

4.3 检查反向关联

Right B **I** CEP

对于一些方法, 我们可以使用反向的逻辑关系来验证它们。例如, 你可以用对结果进行平方的方式来检查一个计算平方根的函数, 然后测试结果是否和原数据很接近:

```
[Test]
public void SquareRootUsingInverse() {
    double x = MyMath.SquareRoot(4.0);
    Assert.AreEqual(4.0, x*x, 0.0001);
}
```

TestRoots.cs

类似地, 为了检查某条记录是否成功地插入了数据库, 你也可以通过查询这条记录来验证, 等等。

要注意的是, 当你同时编写了原方法和它的反向测试时, 一些 bug 可能会被在两个函数中都出现的错误所掩盖。在可能的情况下, 应该使用不同的原理来编写反向测试。在上面平方根的例子中, 我们用的只是普通的乘法来验证我们的方法。而在数据库查找的例子中, 我们大概可以使用厂商提供的查找方法来测试我们的插入。

4.4 使用其他手段来实现交叉检查

Right BI **C** EP

你同样可以使用其他的手段来交叉检查函数的结果。

通常而言, 计算一个量会有一种以上的算法。我们可能会基于运行效率或者其他的特性来选择算法。那是我们要在产品中使用的; 但是在测试用的系统中, 可以使用剩下算法中的一个来交叉测试结果。当确实存在一种经过

验证并能完成任务的算法，只是由于速度太慢或者太不灵活而没有在产品代码中使用时，这种交叉检查的技术将非常有效。

我们可以充分利用一些比较弱的版本来检查我们新写的超级漂亮的版本，看它们是否产生了相同的结果。²

```
[Test]
public void SquareRootUsingStd() {
    double number = 3880900.0;
    double root1 = MyMath.SquareRoot(number);
    double root2 = Math.Sqrt(number);
    Assert.AreEqual(root2, root1, 0.0001);
}
```

TestRoots.cs

另一种办法就是：使用类本身不同组成部分的数据，并且确信它们能“合起来”。例如，假设我们正在做一个图书馆的数据系统（就是那个用砖瓦建起来的能借我们真正书籍的图书馆）。在这个系统中，对于每一本具体的书，它的数量永远是平衡的。也就是说，借出数加上躺在架子上的库存数应当永远等于总共所藏的书籍数量。这些就是数据的两个分开的不同组成部分（借出数和库存数）。它们甚至可以由不同类的对象来汇报它们，但是它们仍然必须遵循上面的约束（即平衡，总数恒定）。因而，我们可以在它们之间进行交叉检查，即用一种数量检查另一种数量。

4.5 强制产生错误条件

在真实世界中，错误总是会发生：磁盘会满，网络连线会断开，电子邮件会多得像掉进了黑洞，而程序会崩溃。你应当能够通过强制引发错误，来测试你的代码是如何处理所有这些真实世界中的问题的。

Right BLC E P

搞一个无效参数之类的做法是小菜一碟，但是不拔网线要模拟网络错误就需要一些特殊的技术了。我们将在第 6 章的 73 页中讨论一种使用 Mock

² 某些电子数据表引擎（如 Microsoft 的 Excel）应用了类似的技术从下面两种情况进行检查：所选择的模型和方法是否是解决特定问题的适当方法，从不同方法得出来的答案是否一致。



对象来解决问题的办法。

但是在那之前，先考虑还有哪些环境方面的约束，可以用来测试你的函数？在往下读之前先列一个小清单出来。

先好好想想，然后再往下读……

下面是一些我们能想到的环境方面的因素。

- 内存耗光。
- 磁盘用满。
- 时钟出问题。
- 网络不可用或者有问题。
- 系统过载。
- 受限的调色板。
- 显示分辨率过高或者过低。

4.6 性能特性

Right BICE **P**

一个检查起来会很有益处的部分是性能特性，而不是性能本身。然而性能特性却有一种类似于“随着输入尺寸慢慢变大，问题慢慢变复杂”的趋势。

我们想要的是一个性能特性的快速回归测试。很多时候，也许我们发布的第一个版本工作正常，但是第二个版本不知道为何变得很慢。我们不知道为什么，也不知道改变了什么，或者什么时候谁干的，几乎一切都是未知的。而最终用户绝对不会轻易放过你。

为了避免这种尴尬场景发生，你可以考虑实现一些粗糙测试来确保性能曲线能够保持稳定。例如，假设我们已经编写了一个过滤器，它能够鉴别希望阻止的 Web 站点（毕竟使用我们新的产品来查看淫秽图片也许会让我们惹上法律上的麻烦。）

那段代码在十几个样板站点上都工作正常,但要是 10 000 个呢? 100 000 个呢? 让我们写点单元测试来看看吧。

```
[Test]
public void FilterRanges() {
    Timer timer = new Timer();
    String naughty_url = "http://www.foooooo.com";
    // First, check a bad URL against a small list
    URLFilter filter = new URLFilter(small_list);
    timer.Start();
    filter.Check(naughty_url);
    timer.End();
    Assert.IsTrue(timer.ElapsedTime < 1.0);
    // Next, check a bad URL against a big list
    filter = new URLFilter(big_list);
    timer.Start();
    filter.Check(naughty_url);
    timer.End();
    Assert.IsTrue(timer.ElapsedTime < 2.0);
    // Finally, check a bad URL against a huge list
    filter = new URLFilter(huge_list);
    timer.Start();
    filter.Check(naughty_url);
    timer.End();
    Assert.IsTrue(timer.ElapsedTime < 3.0);
}
```

TestFilter.cs

这给了我们一些保证,保证了我们仍然满足了性能方面的要求。但是运行这一个测试就花去了 6~7 秒钟,所以我们可能不想每次都运行它。因此,只要每晚或者每隔几天运行它一次,就能快速地定位到我们可能引入的任何问题,而此时仍然有时间来修正它们。

第 5 章

CORRECT 边界条件

CORRECT Boundary Conditions

代码中的许多 Bugs 都出现在“边界条件”附近，也就是说，在那些条件下，代码的行为可能不同于平常的、每天都能运行到的程序路径。

例如，假设你有个接收两个整数的函数：

```
public int Calculate(int a, int b) {  
    return a / (a+b);  
}
```

大多数时候，代码会返回你所期望的值。但是如果 a 和 b 的总和正好为零，你将会得到 `DivideByZeroException` 异常，而不是期望的返回值。这就是边界条件——在这种条件下代码可能会突然出错，至少不像你所期望的那样运行。

我们用首字母缩略词 CORRECT 来帮助你考虑要如何测试哪些边界条件。

- 一致性 (Conformance)——值是否符合预期的格式？
- 有序性 (Ordering)——一组值是该有序的，还是该无序的？

TestRoots.cs

- 区间性 (Range) —— 值是否在一个合理的最大值和最小值的范围之内?
- 引用, 耦合性 (Reference) —— 代码是否引用了一些不受代码本身直接控制的外部因素?
- 存在性 (Existence) —— 值是否存在 (例如, 非 null, 非零, 包含于某个集合等)?
- 基数性¹ (Cardinality) —— 是否恰好有足够的值?
- 时间性, 绝对的或者相对的 (Time) —— 所有事情是否都是按顺序发生的? 是否在正确的时间? 是否及时?

下面让我们按顺序逐一来查看这些边界条件。请记住, 在考虑上面的每一个方面时, 你都需要同时考虑传入方法作为实参的数据和由方法或者类所维护的内部数据。

你需要好好回答的根本问题就是:

还有什么会出错

一旦你想到有什么可能会出错, 就写一个针对它的测试。当测试通过之后, 再问自己“还有什么会出错?” 然后再写一个测试, 如此往复。

5.1 一致性

CORRECT

很多时候, 你所期望的或产生的数据必须符合某种特定的格式。比如一个 e-mail 地址并不仅仅是一个简单的字符串。你期望 e-mail 地址是这样的格式:

```
name@somewhere.com
```

可能还会有一些附加的用点 (.) 隔开的部分:

```
firstname.lastname@subdomain.somewhere.com
```

甚至还有如下更为少见的格式:

```
firstname.lastname%somewhere@subdomain.somewhere.com
```

假设你正在写一个程序, 这个程序可以把用户的名字从 e-mail 地址中抽取出来。你会期望用户的名字就是符号 @ 前面的那部分。但是, 如果没有 @ 这个标志, 你的代码将会怎么处理呢? 代码还能工作吗? 抛出一个异常吗? 这

¹ 也称为集合的势, 指集合包含的元素的个数。

是否也是一个你需要考虑的边界条件呢？²

验证诸如“e-mail 地址、电话号码、账号和文件名”等格式化数据通常都是简单明了的。但是对于结构相对复杂的数据又该怎么处理呢？假设你正在阅读某一类报告数据，其中包括一个头部记录，这个头部记录链接到了一些数据记录，最后是尾部记录。我们需要测试多少种情况呢？

- 如果没有头部记录，只有数据记录和尾部记录，要怎样处理？
- 如果没有数据记录，只有头部记录和尾部记录，要怎样处理？
- 如果没有尾部记录，只有头部记录和数据记录，要怎样处理？
- 如果只有一个尾部记录，要怎样处理？
- 如果只有一个头部记录，要怎样处理？
- 如果只有一个数据记录，要怎样处理？

就像那个简单的 e-mail 地址的例子一样，必须考虑到：如果数据不能像你期望的那样与结构相一致，将会出现什么情况。

当然，如果你正在创建一个像 e-mail 地址这样的数据（可能根据不同的信息来源来创建）或者具有如上结构的数据，你就需要测试你的结果并确保其一致性。

5.2 有序性

另一个需要考虑的方面是数据的顺序，或者是在一个很大的数据集中 C^oRECT 某一数据的位置。比如在上一章中所列举的 Largest () 的例子中，一个 bug 是否会暴露出来，取决于你所查找的最大值是否是位于 list 的最前或者最后。

这是有序性的一个方面。任何一种搜索程序都应该针对搜索目标位于最前或者最后的条件做测试，因为许多常见的 bug 在这种情况下都能找出来。

² E-mail 地址实际上是很复杂的。仔细阅读注解 RFC822，你将会惊讶地发现这点。

有序性还包含另一个方面。假设你正在写一个函数，它接收一个包含餐馆菜单的集合。你期望开胃菜排在菜单的第一位，接着是沙拉（以及所有重要的调味品），然后是主菜，最后是甜点和巧克力。

如果最前是甜点，主菜在最后，则你的代码怎么处理？

如果这种情况可能发生，并且在发生的时候，你的程序需要处理它，那么你就需要测试这种情况，并且把这个问题说清楚。现在，也许你的方法并不需要担心这种情况。或许应该在用户的输入层就解决这个问题（参见后面的“测试无效参数”）。

如果你正在写一个排序函数，在集合已经有序的情况下，可能会出现什么情况呢？或者更糟糕的，原集合已经反向排序好了，会出现什么情况呢？询问你自己这是否会导致问题——这些问题是否也值得编写一个测试。

如果你的方法是让某些元素维持有序，就应该检查一下它们是否真的有序。假设你有一个方法，它属于 GUI 的一个部分，负责把点餐菜单发送到厨房，那么你就应该做一个测试来保证菜单上的菜目顺序符合正确的上菜顺序：

```
[Test]
Public void kitchenOrder(){
    Order order = new Order();
    FoodItem dessert = new Dessert ("chocolate Decadence")
    FoodItem entree = new Entree ("Beef Oscar")
    FoodItem salad = new Salad ("Parmesan Peppercorn")

    //加菜单
    order.AddFoodItem (dessert);
    order.AddFoodItem (entree);
    order.AddFoodItem (salad);

    //但还是应该按照上菜的顺序送上来
    IEnumerator itr = order.GetEnumerator();
    Assert.AreEqual (salad, itr.Current);
    itr.MoveNext ();
    Assert.AreEqual (entree, itr.Current);
    itr.MoveNext ();
    Assert.AreEqual (dessert, itr.Current);
    Itr.MoveNext ();

    //菜都上完了
    Assert.IsFalse (itr.MoveNext());
}
```

TestKitchen.cs

当然，从客户的角度来看，你需要把代码修改得足够灵活，这样可以让他们在别人都吃沙拉的时候按自己的意愿先吃冰淇淋。这种情况下，你需要加一个测试来证实你侄子的冰淇淋是和其他人的沙拉是一起上的，而祖母的冰淇淋则是和你的咖啡一起上的。

5.3 区间性

Range(区间)是一个针对所有类似下列情景的单词：对于一个变量，它所 CO^RRECT 属类型的取值范围可能比你需要或想要的更加宽广。例如，我们通常使用整型来表示一个人的岁数，但是显然没有人能够活到 200 000 岁，尽管 200 000 是一个合法的整型值。类似地，虽然我们是用整型来存储角度，但是在一个圆中，角度值不会大于 360°。

在一个好的面向对象设计中，你通常都不会使用一个原生类型（如 `int`, `Int32`）来存储一个具有边界的值，诸如岁数、罗盘头指向的角度等：

```
using System;

//
// Compass bearing
//

public class Bearing {
    protected int bearing; // 0..359
    //
    // Initialize a bearing to a value from 0..359
    //

    public Bearing(int num_degrees) {
        if (num_degrees < 0 || num_degrees > 359) {
            throw new ArgumentException( "Bad bearing");
        }
        bearing = num_degrees;
    }
    //
    // Return the angle between our bearing and another.
    // May be negative.
    //

    public int AngleBetween(Bearing anOther) {
        return bearing - anOther.bearing;
    }
}
```

Bearing.cs

注意，返回的角度值只是一个 `int` 值，因为我们并没有给返回结果强加任何区间约束(例如，它可能会是一个负值)。

在上面的代码中，通过把 bearing 这个概念封装到一个类中，你现在就可以在系统中的某处过滤掉那些不合理的值。另外，你也不能创建出一个位于区间之外的 Bearing 对象。因此，对于系统中使用 Bearing 对象的其他部分，现在就可以确认它们包含的肯定是合理的角度值。

其他的区间可能就不是这么显而易见了。例如，假设你有一个维护了两对 x,y 坐标的类。坐标值都属于整型，并且可以具有任意值；但是，区间具有如下的约束：这两个点(两个坐标代表的点)必须能够描述一个所有边都不大于 100 个 unit 的矩形。也就是说，这两对 x,y 值的允许区间是互相依赖的。因此，对于可能影响其中一个坐标的任何方法，你都需要进行区间测试，从而确保结果区间中的两对 x,y 值是合法的。针对这个主题的更多信息，可以参考第 117 页“设计话题”中的“不变性”。

既然在多个测试中你都可能需要调用这个区间验证，那么我们应该实现一个新的 assert 方法：

```
public const int MAX_DIST = 100;
static public void AssertPairInRange(Point one,
                                     Point two,
                                     String message) {
    Assert.IsTrue(Math.Abs(one.X - two.X) <= MAX_DIST,
                  message);
    Assert.IsTrue(Math.Abs(one.Y - two.Y) <= MAX_DIST,
                  message);
}
```

但是你想要测试的最常见区间很可能是取决于物理数据结构的问题，而不是应用领域的约束。举个简单的例子，一个针对 String 的 stack (栈) 类需要使用一个数组：

```
public class MyStack {
    public MyStack() {
        stack = new String[100];
        nextIndex = 0;
    }
    public String Pop() {
        return stack[--nextIndex];
    }
    // Delete n items from the stack en-masse
    public void Delete(int n) {
        nextIndex -= n;
    }
    public void Push(String aString) {
        stack[nextIndex++] = aString;
    }
}
```

TestPair.cs

```

    public String Top() {
        return stack[nextIndex-1];
    }

    private int nextIndex;
    private String[] stack;
}

```

MyStack.cs

这里隐藏着一些潜在的 bugs, 因为对空栈或者栈溢出的情况都没有做任何检测。然而当我们操作索引变量 `nextIndex` 的时候, 有一点是肯定为恒真的: `(next_index >= 0 && next_index < stack.Length)`。因此, 我们需要检查这一点来确保上面这个表达式是正确的。

现在, `next_index` 和 `stack` 都是私有变量; 你不会希望只是因为测试而将它们都暴露出来吧。有几个办法可以解决这个问题; 目前而言, 我们只是在 `Stack` 中写了一个叫做 `CheckInvariant()` 的特殊方法:

```

public void CheckInvariant() {
    if (!(nextIndex >= 0 &&
        nextIndex < stack.Length)) {
        throw new InvariantException(
            "nextIndex out of range: " + nextIndex +
            " for stack length " + stack.Length);
    }
}

```

MyStack.cs

现在测试程序就可以调用 `CheckInvariant()`, 从而在不直接访问这些变量的前提下, 确保 `stack` class 的内部没有出现越界行为。

```

using NUnit.Framework;

[TestFixture]
public class TestMyStack {

    [Test]
    public void Empty() {
        MyStack stack = new MyStack();
        stack.CheckInvariant();
        stack.Push( "sample");
        stack.CheckInvariant();
        // Popping last element ok
        Assert.AreEqual( "sample", stack.Pop());
        stack.CheckInvariant();
        // Delete from empty stack
        stack.Delete(1);
        stack.CheckInvariant();
    }
}

```

TestMyStack.cs

当你运行这个测试时，你很快就会发现我们确实需要加入一些区间检查！

```
TestCase 'TestMyStack.Empty' failed: InvariantException
nextIndex out of range: -1 for stack length 100
    mystack.cs(34,0): at MyStack.CheckInvariant()
    testmystack.cs(20,0): at TestMyStack.Empty()
```

在一个简单的测试环境中，发现并解决这一类错误与等到该错误都隐藏到应用程序中时才寻找错误相比，这种做法要简单得多。

几乎所有的索引概念（无论是否为整型索引）都应该被大量的测试。这里有一些建议供你开始测试时参考：

- 开始索引和结束索引有相同的值。
- 第一个索引值大于最后一个索引值。
- 索引值是负的。
- 索引值大于允许值。
- Count 不能匹配确切索引的个数。
- ...

5.4 引用/耦合性

CORRECT

你的程序引用了哪些位于程序之外的事物？或者任何外部依赖？类应该处于什么样的状态？程序运行还需要存在哪些其他的条件？

例如，网站上一个用来显示用户账号记录的方法，可能要求用户先登录。栈的 `Pop()` 方法要求一个非空的栈。给汽车换挡，使其从运行挡转换到停车挡，要求汽车最后是停着的。

如果对于类的状态、其他对象的状态，或者全局应用程序的状态，你需要做一些假设，那么你就需要对代码进行测试，保证其在假设未满足的情况下运行良好。例如，汽车微处理控制传动器的代码可能有一组用来检查以下特殊情况的测试：传动器的状态（是否切换到停车挡）取决于汽车的状态（处于移动状态还是停车状态）。

```

[Test]
public void JamItIntoPark() {
    transmission.Shift(DRIVE);
    car.AccelerateTo(35);
    Assert.AreEqual(DRIVE, transmission.CurrentGear);
    // should silently ignore
    transmission.Shift(PARK);
    Assert.AreEqual(DRIVE, transmission.CurrentGear);
    car.AccelerateTo(0); // i.e., stop
    car.BrakeToStop();
    // should work now
    transmission.Shift(PARK);
    Assert.AreEqual(PARK, transmission.CurrentGear);
}

```

一个已知方法的前条件 (*preconditions*) 是指：系统必须处于什么状态下，该方法才能运行。在上面的例子中，使传动器停止的前条件就是汽车的发动机（应用程序系统中的一个分离部件）必须停止。这是该方法必须满足的文档化要求，因此我们需要确认：当前条件不能满足的时候，程序的行为仍然是正确的（在这一特殊情况下，传动器将忽略其请求）。

在方法的最后，后条件 (*postconditions*) 是指：你的方法将会保证哪些状态发生。程序直接返回的结果显然是要检查的，但如果程序具有任何副作用，那你也需要对它们进行检查。在上面的例子中，刹车就具有令车停下来的副作用。

某些语言甚至自身就支持前条件和后条件：有兴趣的读者可以从《Object-Oriented Software Construction》[Mey97]一书中了解有关 Eiffel 语言的内容。

5.5 存在性

通过询问下面的问题，你也许就能够发现许多 bugs：“给定的事物存在吗？” CORRECT

对于你传入或者维护的值，先询问自己如果值不存在——如果它为 null，或者空值，或者等于 0，方法的行为将会怎么样？

面对这些不存在的数据，C#库的许多方法会抛出某种异常。然而要调

试一个隐藏在库深处的运行期异常通常并不容易，这也正是问题所在。然而，报告诸如“Age isn't set”的异常通常都能够让我们很容易地捕获到问题所在。

在期望值不存在的时候，大多数方法都会失败；但这可能并不是你所期望的结果。因此，你可以对这些情况进行测试——查看当由于某些查询失败，你获得了一个 null 而不是 CustomerRecord 的时候，发生了哪些事情；或者当网络不通、文件不存在的时候，又会发生什么事情。

Ah, yes: 环境中的任何事物并不能保证其肯定存在——网络、文件的 URL、许可码、用户、打印机——等等其他你能列举的事物。当你期望它们存在的时候，它们可能并不存在；因此，你要确认你已经做了足够的测试：null、0、空字符串和其他与存在性相关的陷阱。

确认你的方法处理了“不存在”的情况。

5.6 基数性

CORRECT

这里的基数与高高在上的神明，或者与林间的小麻雀毫无关联，它指的是计数（counting）。

计算机程序员对于计数并不擅长，特别是在不能用十个指头帮着数的时候。比方说，快速地回答以下的问题，就用你的大脑，别用指头、白纸、或者 UML 等来帮忙：

假如你有 12 英尺的院子需要用桩子围起来，而相邻桩子的间隔为 3 英尺，你需要多少个桩子呢？

如果你和我们中的大多数一样，你可能想都没想就答 4 了。不幸的是，这是错误的答案——你需要 5 个篱笆桩，如 66 页中图 5.1。这个例子，以及类似的常见错误，因为出现得太多了，所以人们就给它取了一个优雅的名字“界桩错误(fencepost error)。”

事实上，上面只是“off by one”错误的一种形式而已，而且这种偶然性的致命错误时常会折磨许多程序员。因此，你需要想出办法来测试你的函数是否能够正确计数，并且检查最后的计数值。

这是一个和存在性相关的问题，但是你现在需要确信：你计算得到的数目和你所需要的数目是一致的。在大部分情况下，你只须考虑下面 3 种值：

1. 零
2. 一
3. 多于一

这被称为“0-1- n 原则”，这里有个前提是你能够处理多于一个的情况。对于你而言，也许处理 10、20 或者 1000 都是小菜一碟。在大多数情况下也确实如此；因此，我们针对基数性的测试主要是考虑：我们是否具有两个或者多个测试对象。当然，在某些情况下，具体数目的不同也可能会引入一些差异——对你而言，10 或者 250 可能会是一个很重要的数值。

假设你正在维护一个薄烤饼店中被订购食物的一个前十名列表。每次一个订单下来，你必须调整前十个条目。同时，你还需要以实时数据的形式，把当前的前十名条目发送到薄饼店老板的 PDA 上。此时，你需要测试哪些内容呢？

- 当列表条目还不足十个的时候，能出报表吗？
- 当列表空无一物时，能出报表吗？
- 当列表只有一个条目时，还能出报表吗？
- 当列表还没有十个条目时，能添加一个新的条目吗？
- 当列表什么都没有时，能添加一个新的条目吗？
- 当列表条目数为一时，能添加一个新的条目吗？
- 要是菜单本身就没十个条目，怎么办？
- 那要是菜单中没有任何条目，又怎么办？

看了你给出的列表之后，老板现在改变主意了。他想要一个前二十位的列表。这时你要做哪些修改呢？

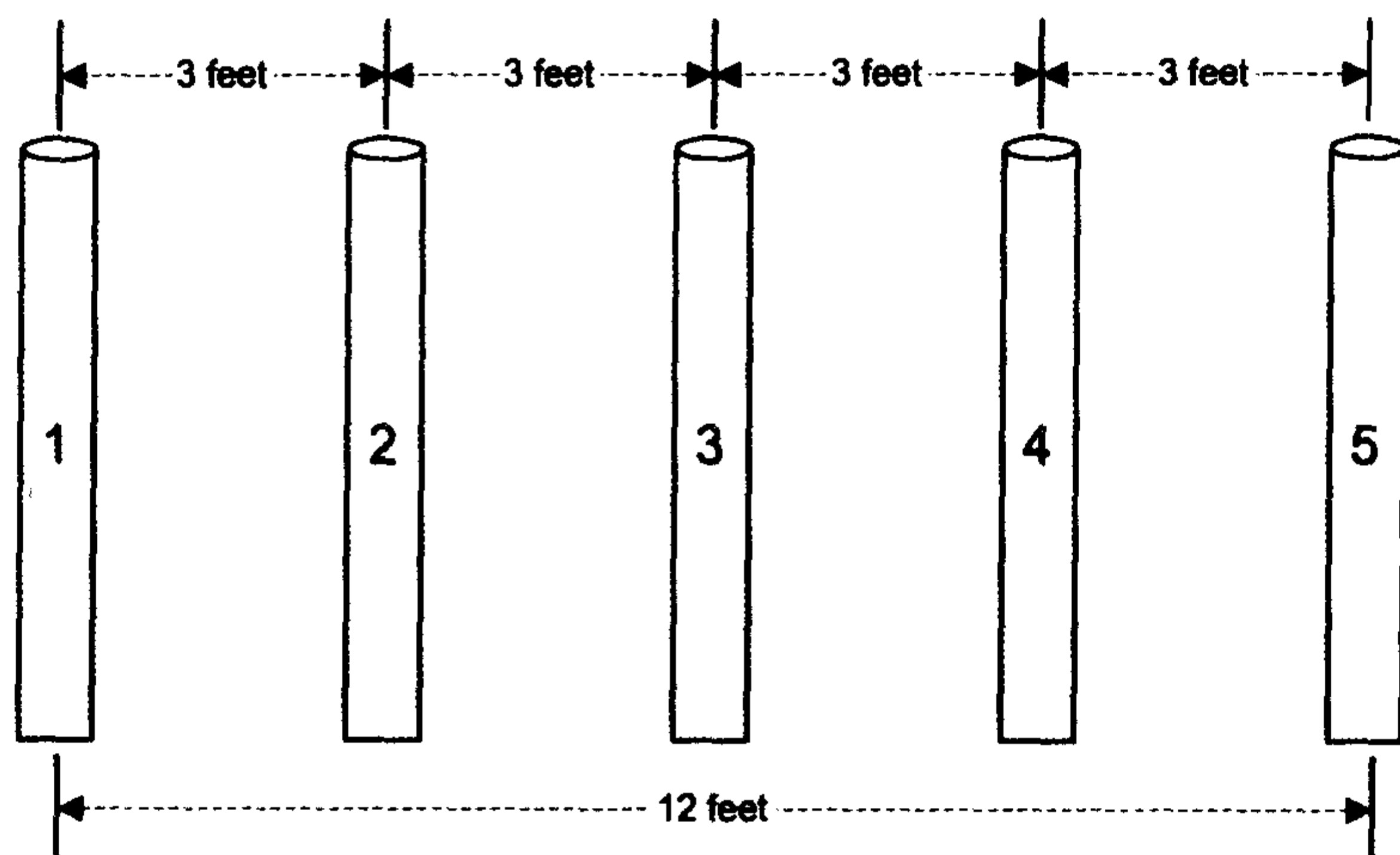


图5.1 一组篱笆桩

正确的答案是“一行代码”，类似下面的代码：

```
public MaxEntries {
    get { return 20; }
}
```

现在，老板又不喜欢长长的列表，要你把它改成只列出前五位的列表（毕竟他的 PDA 是相当小的）；你可以回到上面那一行代码，改变那个数字。而测试应该能够自动更正，因为它使用的是同一个属性，而不是具体的值。

因此，测试最后将专注于 0、1 和 n 这样的边界条件上，其中 n 的数值也很可能会随着业务需求而变化。

5.7 时间性

CORRECT **T**

CORRECT 缩写中的最后一个边界条件是 Time。你需要始终记得以下这些与时间相关的方面：

- 相对时间（时间上的顺序）。
- 绝对时间（消耗的时间和钟表上的时间）。
- 并发问题。

一些接口会拥有自己的状态；因此，为了维持这些内部状态，你期望 `Login()` 会在 `Logout()` 之前被调用，`PrepareStatement()` 会在 `ExecuteStatement()` 之前被调用，`Connect()` 在 `Read()` 之前被调用，而 `Read()` 又在 `Close()` 之前被调用，等等。

要是这些函数的被调用顺序出错，会如何？也许你应当尝试使用非预期的顺序来调用这些方法。譬如，试着跳过第一个、最后一个或者序列中间的某一个过程。就如前面例子中数据的顺序可能会给你带来麻烦一样（描述于 57 页的“有序性”一节），现在的顺序指的是函数被调用的顺序。

相对时间也许还包括代码中的超时问题：为了获得一个存活期较短的资源，你的函数会等待多长时间。如我们后面简短讨论的，你需要在你的代码中验证可能的出错条件，包括超时出错等。可能你已经遇到过未受超时保护的情形——你能想出代码由于等待了一些永远不能得到的资源而阻塞的情形吗？

这把我们引到了消耗时间的话题：要是你等待的事情消耗了“太多”的时间，会怎样？要是你的函数花了好长一段时间才返回给调用者，那又会怎么样？

接下来要考虑的是钟表上的实际时间。在大多数情况下，这并不会对代码产生影响。但是用一种精确的角度来看，一天中的时间可能也会对结果有所影响。这儿有一个快速问答题，你需要回答是正确还是错误：一年中的每一天都是 24 小时？

答案是“要看情况”。在 UTC（Universal Coordinated Time, Greenwich Mean Time 即 GMT 的现代版本）中，答案是“正确”。在不遵守 Daylight Savings Time（DST）的地方，答案是“正确”。然而在美国的大部分地区（遵守 DST），答案是“错误”。在四月，你会遇到只有 23 小时的一天（spring forward[c1]），在十月你会遇到有 25 小时的一天（fall back）。这意味着如下的算术或许不能如你预期般正常工作了：比如 1:45 AM 加上 30 分钟可能等于 1:15。

但是对于所有时间敏感的代码，你已经在边界上做过测试了，对吧？包括那些采用 DST 的地方和不采用 DST 的地方，你都已经测试过了吗？

Oh, 不要假设任何底层库会替你正确处理这些问题。不幸的是, 一遇上时间问题, 底层库的很多代码都会出现问题。

最后, 时间带来的最棘手问题是并发访问和同步访问问题。对于这个话题, 可能需要一整本书来讲解如何设计、实现、调试多线程的并发程序, 因此我们在此就不详细讨论这个话题, 但是要指出的是: 你今天用大多数语言编写的大多数代码, 以后将会运行于多线程的环境中。

因此应该询问自己, 要是有多多个线程同时使用同一个对象, 会出什么问题? 存在全局或者实例级别的, 并且需要同步的数据或函数? 外部是如何访问文件或者硬件的? 请确保给必要的数据和函数添加了 lock 关键字, 并且尽量在日常测试中把多线程作为测试的一部分。

5.8 自己动手尝试

现在我们已经讲过了 RIGHT-BICEP 和 CORRECT 的测试办法了, 该是你尝试使用它们的时候了。

对于下面的每个场景和例子, 写出你能想出的尽可能多的单元测试。

■ 练习

答案见 148 页

1. 一个简单的栈类。根据通常意义的栈的语义, 把字符串对象压入到栈内, 并将其弹出。这个类提供了以下函数:

```
using System;
public interface StackExercise {
    /// <summary>
    /// Return and remove the most recent item from
    /// the top of the stack.
    /// </summary>
    /// <exception cref="StackEmptyException">
    /// Throws exception if the stack is empty.
    /// </exception>
    String Pop();
    /// <summary>
    /// Add an item to the top of the stack.
    /// </summary>
```

```

    /// <param name="item">A String to push
    /// on the stack</param>
    void Push(String item);
    /// <summary>
    /// Return but do not remove the most recent
    /// item from the top of the stack.
    /// </summary>
    /// <exception cref="StackEmptyException">
    /// Throws exception if the stack is empty.
    /// </exception>
    String Top();
    /// <summary>
    /// Returns true if the stack is empty.
    /// </summary>
    bool IsEmpty();
}

```

StackExercise.cs

这儿有一些让你入手的提示：什么是最可能出问题的？当栈首度初始化时的行为应该是怎么样的？在它被用过一段时间后呢？它确实如它应该的那样吗？

2. 一个购物车。这个类让你添加、删除，以及计算购物车中物品的数量。 答案见 149 页

会出现的边界条件是什么？对于你能删除的商品，有没有什么暗含的约束？如果车是空的，有没有什么有意思的问题？

```

public interface ShoppingCart {
    /// <summary>
    /// Add this many of this item to the
    /// shopping cart.
    /// </summary>
    /// <exception cref="NegativeCountException">
    /// </exception>
    void AddItems(Item anItem, int quantity);
    /// <summary>
    /// Delete this many of this item from the
    /// shopping cart
    /// </summary>
    /// <exception cref="NegativeCountException">
    /// </exception>
    /// <exception cref="NoSuchItemException">
    /// </exception>
    void DeleteItems(Item anItem, int quantity);
    /// <summary>
    /// Count of all items in the cart
    /// (that is, all items x qty each)
    /// </summary>
    int ItemCount { get; }
    /// Return iterator of all items
    IEnumerable GetEnumerator();
}

```

ShoppingCart.cs

答案见 150 页

3. 一个传真调度程序。这段代码将从一个指定的文件名发传真到一个美国的电话号码。其中有校验的需求；带区号的美国电话必须是这样的格式 $xnn-nnn-nnnn$ ，其中 x 必须是在 2 到 9 之间，而 n 可以是 0 到 9 的任何数字。以下的区域是预留的，现在还不是有效的区号： $x11$ 、 $x9n$ 、 $37n$ 、 $96n$ 。

函数的原型是：

```
///
/// Send the named file as a fax to the
/// given phone number.
/// <exception cref="MissingOrBadFileException">
/// </exception>
/// <exception cref="PhoneFormatException">
/// </exception>
/// <exception cref="PhoneAreaCodeException">
/// </exception>
public bool SendFax(String phone, String filename)
```

给定了这些需求，你可以想出哪些边界条件？

答案见 151 页

4. 一个刺绣用的自动缝纫机。这个控制它的类有几个基本的命令。 $coordinates(0,0)$ 代表机器的左下角。随着你向右上角移动， x 和 y 不断增加，最终坐标为 $x = TableSize.Width - 1$ 而 $y = TableSize.Height - 1$ 。

坐标以厘米的分数形式来表示：

```
public void MoveTo(double x, double y);
public void SewTo(double x, double y);
public void SetWorkpieceSize(double width,
                             double height);
public Size WorkpieceSize { get; }
public Size TableSize { get; }
```

这儿有一些真实世界的约束可能会比较有意思；你当然不能缝纫稀薄的空气，而且你不能缝纫比机器更大的工件。

给定了这些需求，你可以想出哪些边界条件？

答案见 152 页

5. 一个音频/视频的位移编辑器。一个提供控制 VCR 或者卡带的函数的类。其中有一个“当前位置”的记号，位于带子的开头处（BOT）至带子的结尾处（EOT）之间的某个地方。

你可以询问当前的位置，并从该位置移动到其他给定的位置。快进从当前

位置向 EOT 移动一段距离。倒带从当前位置向 BOT 移动一段距离。

当带子第一次被载入时，它自动位于 BOT。

```
using System;
public interface AVTransport {
    /// Move the current position ahead by this many
    /// seconds. Fast-forwarding past end-of-tape
    /// leaves the position at end-of-tape
    void FastForward(double seconds);
    /// Move the current position backwards by this
    /// many seconds. Rewinding past zero leaves
    /// the position at zero
    void Rewind(double seconds);
    /// Return current time position in seconds
    double CurrentTimePosition();
    /// Mark the current time position with label
    void MarkTimePosition(String name);
    /// Change the current position to the one
    /// associated with the marked name
    void GotoMark(String name);
}
```

AVTransport.cs

6. 音频/视频位移编辑器，2.0 版。除了前面说过的，现在你还能用秒、分钟或者帧（在本例中，一秒就是固定的 30 帧）来定位，而且还能相对开头和结尾进行移动。 答案见 153 页

第 6 章

使用 Mock 对象

Using Mock Objects

单元测试的目标是一次只验证一个方法，但是倘若遇到这样的情况：某个方法依赖于其他一些难以操控的东西，诸如网络、数据库，甚至是微软最新的股票价格，将会发生什么情况呢？

要是你的测试代码依赖于系统的其他部分，甚至系统的多个其他部分呢？在这种情况下，倘若不小心，最终可能会发现自己几乎初始化了系统的每个组件，而这只是为了给某一个测试创造足够的运行环境，让它可以运行起来。这样不仅消耗了时间，还给测试过程引入了大量的耦合因素：例如，某人兴冲冲地改变了一个接口或者数据库中的一张表，突然地，你那卑微的单元测试的准备代码神秘地挂掉了。在这种情况下发生了几次之后，即使是最有耐心的开发者也会泄气，甚至最终会放弃所有的测试。但其实还是有一些我们能够用来改善这种境况的技术的。

在电影和电视制作中，工作人员通常会为真正的演员提供一些替身（standins 或 doubles）。具体来说，当摄影师在调整灯光或者摄像机角度的时候，他们会使用灯光替身：即高度和体形都与真正的演员相像，但地位并不那么重要的人，而那些身价不凡的真正的演员此时可能正在他们的拖车休息室里懒洋洋地躺着呢。

摄影师接下来将借助这些灯光替身来测试他们搭建的摄影环境，量量摄影机到演员的鼻子的距离，细细调整灯光的亮度和角度，直到不需要的阴影

完全消失为止，等等。而这些顺从的替身只是立在那儿，一点也不会抱怨他们扮演的角色“缺少动感”。

在单元测试中我们也要使用类似于电影中灯光替身一类的替代品：一种和真品非常接近的赝品，至少表面上看起来是这样；但是就我们测试的目的而言它更加好用。

6.1 简单的替换

我们需要做的是撇开真实世界中的实际对象，而以功能一样的盟友——我们自制的“灯光替身”——来代替它们。例如，我们可能并不想在真正的数据库上做测试，或者也不想使用当前墙上钟表显示的真实时间；让我们来看一个简单的例子吧。

假设在代码中调用 `Now` 属性来返回系统当前的日期和时间，其中 `Now` 的定义大概是这个样子：

```
public DateTime Now {  
    get {  
        return DateTime.Now;  
    }  
}
```

（一般而言，我们通常建议对应用程序范围外的功能调用进行包装，从而能够更好地封装它们——上面就是一个很好的例子。）在上面的代码中，我们把当前时间的概念包装在自己写的代码里面，因此调试就容易了一些：

```
public DateTime Now {  
    get {  
        if (DEBUG)  
            return currentTime;  
        else  
            return DateTime.Now;  
    }  
}
```

你还可以使用其他的调试路径来操纵“当前时间”这个系统概念，从而让那些不这样做就需要等很长时间才能发生的事件马上发生。

SystemEnvironment.cs

Examples.cs

这是替换真正功能的手段之一，但有些凌乱。首先，只有在代码一致调用你封装的 `Now`、完全没有直接调用 `DateTime.Now` 的时候，这种方法才有效。但是我们需要的是更加干净——并且更加面向对象化——同时可以实现相同功能的方法。

6.2 Mock 对象

幸运的是，有一种测试模式可以帮助我们：**mock 对象**。Mock 对象也就是真实对象在调试期的替代品。在许多情况下，mock 对象都可以给我们带来帮助。Tim Mackinnon [MFC01]给出了下面这个列表：

- 真实对象具有不可确定的行为（产生不可预测的结果，如股票行情）。
- 真实对象很难被创建。
- 真实对象的某些行为很难触发（如网络错误）。
- 真实对象令程序的运行速度很慢。
- 真实对象有（或者是）用户界面。
- 测试需要询问真实对象它是如何被调用的（例如，测试可能需要验证某个回调函数是否被调用了）。
- 真实对象实际上并不存在（当需要和其他开发小组，或者新的硬件系统打交道的时候，这是一个普遍问题）。

借助于 mock 对象，我们就可以解决上面提到的所有问题。在使用 mock 对象进行测试的时候，总共有 3 个关键步骤，分别是：

1. 使用一个接口来描述这个对象。
2. 为产品代码实现这个接口。
3. 以测试为目的，在 mock 对象中实现这个接口。

因为被测试代码只会通过接口来引用对象，所以它完全可以不知道它引

用的究竟是真实对象还是 mock 对象。让我们再次回顾一下上述的系统当前时间那个例子。我们将创建针对真实环境因素的几个对象，其中一个因素就是当前时间：

```
public interface Environmental{
    DateTime Now {
        get;
    }
    //其他的方法都省略了...
}
```

接下来，我们将编写真实的实现代码：

```
public class SystemEnvironment : Environmental {
    public DateTime Now {
        get {
            return DateTime.Now;
        }
    }
}
```

最后是 mock 的实现：

```
using System;
public class MockSystemEnvironment : Environmental {
    private DateTime currentTime;
    public MockSystemEnvironment(DateTime when) {
        currentTime = when;
    }
    public DateTime Now {
        get {
            return currentTime;
        }
    }
    public void IncrementMinutes(int minutes) {
        currentTime = currentTime.AddMinutes(minutes);
    }
}
```

注意在 mock 实现里面，我们给构造函数传入了一个 DateTime 对象，用作初始的当前时间值，并且构造函数用一个实例变量把它保存了下来。该变量就是当我们请求当前时间时所返回的值。同时，我们提供了一个 IncrementMinutes 方法，它会给这个时间加上给定的分钟数，这让你可以控制 mock 对象所返回的日期和时间。

现在假设我们已经编写了一个新的应用程序，它有一个名为 Reminder() 的方法，而且如果是在下午 5 点之后调用该方法，那么它将会响起“go home”的口哨声。（Flintstones 迷们将会想起 Fred 正是在这个时候从它的马车上跳下来，然后爬到恐龙的后背上的）。另外，该方法依赖于 Now 属性。在下面

的代码中，一些细节被略去了，不过我们感兴趣的部分都在下面：

```

Line 1  using System;
-
-  public class Checker {
-      Environmental env;
5
-      public Checker(Environmental env) {
-          this.env = env;
-      }
-
10     public void Reminder() {
-         DateTime now = env.Now;
-
-         if (now.Hour >= 17) {
-             env.PlayWavFile( "quit_whistle.wav");
15         }
-     }
- }

```

Checker.cs

在产品环境（卖给客户的真正的代码）中，当初始化这个类的对象时，传入的是一个真实的 `SystemEnvironment`；而另一方面，测试代码传入的则是 `MockSystemEnvironment`。

使用 `env.Now` 的被测试代码并不知道测试环境和真实环境之间的区别，因为它们都实现了相同的接口。现在，你可以借助 mock 对象，通过把时间设置为已知值，并检查行为是否如预期那样来编写测试了。

除了我们已经展示的 `Now` 属性，`Environmental` 接口还有一个 `PlayWavFile()` 函数（在前面 *Checker.cs* 的第 14 行用到了的）。我们通过给 mock 对象添加一些额外的支持代码，从而能够在不倾听计算机喇叭的情况下，添加测试来观察它是否被调用了。

```

private bool soundWasPlayed = false;
public void PlayWavFile(string fileName) {
    soundWasPlayed = true;
}
// For convenience, check the sound played
// flag and reset it in one method call
public bool CheckAndResetSound() {
    bool value = soundWasPlayed;
    soundWasPlayed = false;
    return value;
}

```

MockSystemEnvironment.cs

把所有这些放到一起，一个使用了这些设置的测试大概是这样的：

```

Line 1  using System;
-       using NUnit.Framework;
-
-       [TestFixture]
5       public class TestChecker {
-
-       [Test]
-       public void QuittingTime() {
-
10          DateTime when = new DateTime(2004,10,1,16,55,0);
-          MockSystemEnvironment env;
-          env = new MockSystemEnvironment(when);
-          Checker checker = new Checker(env);
-
15          // No alarm sounds at 16:55
-          checker.Reminder();
-          Assert.IsFalse(env.CheckAndResetSound(), "16:55");
-
-          // Now try 17:00
20          env.IncrementMinutes(5);
-          checker.Reminder();
-          Assert.IsTrue(env.CheckAndResetSound(), "17:00");
-
-          // And finally 19:00
25          env.IncrementMinutes(120);
-          checker.Reminder();
-          Assert.IsTrue(env.CheckAndResetSound(), "19:00");
-      }
-  }

```

TestChecker.cs

代码创建了一个 `DateTime` 对象，它包含了将在第一个测试中使用的时间，并且把这个对象传递给了一个我们将用来运行这些测试的新的 mock 系统环境。

在第 16 行，我们调用了 `Reminder()`，这将（不知情地）使用 mock 环境。下一行的调用 `Assert.IsFalse()` 检查 `.wav` 文件是不是还没播放，因为在 mock 对象的环境中，此时还不是 quitting time。但是我们将很快调整时间；第 20 行把 mock 时间调整到刚好是 quitting time（顺便提一下，这是一个很好的边界条件）。我们然后再一次调用 `Reminder()` 函数。这次，声音已经被播放过了，因为我们在第 22 行中调用 `Assert.IsTrue` 时确认了 `.wav` 文件这次已经播放过了。

最终，我们将再次测试 `Reminder()` 函数，这次把 mock 时钟向前调两个小时。看看吧，在 mock 环境中改动和检查边界是多么容易啊！你不再需要弯下腰来听 PC 喇叭的声音了，也不需要重设时钟、拉线或别的什么麻烦事。

因为我们已经有了一个提供所有系统功能的现成接口，所以在更多的时候，人们可能（也许吧）会使用它而不是直接调用诸如 `DateTime.Now()` 这样的东西，而是躲在接口背后拥有了控制一切行为的能力。

这就是 mock 对象的全部：伪装出真实世界的某些部分，使你可以集中精力测试好自己编写的代码。

6.3 正规化 Mock Objects

在过去，拥有调用子程序的能力就是一个极大的进步。然后，代码库变得流行起来了；什么东西都要做成一个库。现在，库都不够棒了。你不得不认真地选择并采用一款框架。

在 .NET 中，有多款 mock 对象的框架供选择（可以在 <http://www.mockobjects.org> 找到一个很好的列表）。这节我们将看看其中的一款，DotNetMock¹。

DotNetMock 框架实际上是一个三合一：

1. 它是一个框架（当然是框架啦），让你可以有组织地创建 mock 对象。
2. 它包含一小套预先定义的 mock 对象，你可以拿来测试你的程序。
3. 最后，它还有一项技术，动态 mock，让你能够在不需编写乱七八糟的代码的前提下，就可以创建 mock 对象。

让我们逐个来看看这个三合一。但是在我们开始之前，需要清楚我们是在 .NET 的 CLR 环境中，所以这一个框架可以面向任何一种 .NET 语言来创建 mock 对象。

■ DotNetMock 框架

当你对它有些了解之后，会发现创建一个 mock 对象真的很简单：只不过是一些代码实现一个特殊的接口，返回你需要它返回的值，而这能够

¹ <http://sourceforge.net/projects/dotnetmock>

帮助你检查它的调用方式是否和期望一致。于是，创建 mock 对象的基本框架也相应地变得简单。

为了演示这点，让我们看看如何使用 DotNetMock 框架来帮助我们测试一个访问控制（access control）库。我们将从一个很简单的类 `AccessController` 开始。每个 `AccessController` 对象负责控制一种资源。我们将把资源的名称传给构造函数。为了决定一个用户是否能够使用对象的资源，我们调用对象的 `CanAccess()` 函数传入用户的名字和密码。这个类的代码大概是这样的：

```
using System;
public class AccessController {
    private ILogger logger;
    private String resource;
    public AccessController(String resource,
                             ILogger logger) {
        this.logger = logger;
        this.resource = resource;
        logger.SetName( "AccessControl");
    }
    public bool CanAccess(String user, String password) {
        logger.Log( "Checking access for " + user +
                    " to " + resource);
        if (password == null || password.Length == 0) {
            logger.Log( "Missing password. Access denied");
            return false;
        }
        // more checks...
        logger.Log( "Access granted");
        return true;
    }
}
```

AccessController.cs

我们将要测试这段访问控制代码，但是注意它调用了一些外部的 logger 对象。这并不是一个大问题：因为 logger 是通过接口调用的。

```
using System;
public interface ILogger {
    void SetName(String name);
    void Log(String msg);
}
```

ILogger.cs

我们可以很容易地创建该接口的 mock 对象：仅仅需要实现两个函数，一个用于设置正在进行 logging 的内容的名字，而另外一个用于 log 实际的消息。

```

using System;
public class MockLogger1 : ILogger {
    public void SetName(String name) {
    }
    public void Log(String msg) {
    }
}

```

MockLogger1.cs

有了这些，我们现在可以编写基本的单元测试了。在第一个测试中，将检测 access controller 是否在我们没有传入密码的情况下正确地拒绝了访问。

```

using System;
using NUnit.Framework;
[TestFixture]
public class TestAccessController {
    [Test]
    public void MissingPassword1() {
        MockLogger1 logger = new MockLogger1();
        AccessController access =
            new AccessController( "secrets", logger);
        Assert.IsFalse(access.CanAccess( "dave", null));
        Assert.IsFalse(access.CanAccess( "dave", ""));
    }
}

```

TestAccessController.cs

然而，这个测试并没有验证我们的 access controller 是否 logging 了正确的消息。这就是 mock 对象框架能够帮上忙的地方——它的一大特色就是使得管理预期行为(expectations)变得特别容易²；框架的使用使得记录我们预期发生的事情更容易，然后就能验证它们是否确实发生了。

我们先扩展 mock 对象以验证 access controller 是否正确给 logger 设置了它的名字。我们通过给 MockLogger 类添加 expectations 来完成这个任务。

```

using System;
using DotNetMock;
public class MockLogger2 : MockObject, ILogger {
    private ExpectationValue _name =
        new ExpectationValue( "name");
    // Mock control interface
    public String ExpectedName {
        set { _name.Expected = value; }
    }
}

```

² 我们显然需要更多的这类框架来帮助应付那些有苛刻要求的终端用户。

MockLogger2.cs

```
// Implement ILogger interface
public void SetName(String name) {
    _name.Actual = name;
}
public void Log(String msg) {
}
}
```

一大堆代码都发生了变化：首先，我们现在使用了 DotNetMock 名字空间，并让 MockLogger 扩展自 MockObject 类。为了让它能够编译，给我们的项目添加了对 DotNetMock.dll 的引用。这个 DLL 存在于前面给出的 SourceForge 的 URL 中的 DotNetMock 包中。

接下来给我们的类添加了 ExpectationValue 成员变量。一个 expectation 基本上就是一个保存了两个值的变量：它的预期值和它的实际当前值。在测试的最后，我们能够调用一个 Verify 函数，而 mock 对象框架会检查所有 expectation 对象中的预期和实际值是否都相等。如果它们确实如此，我们的代码一定是如预期那样工作的。如果不是，Verify 会抛出一个 assertion 错误，从而单元测试会告知测试失败了。

第三，我们需要提供一种设置预期值和实际值的办法。实际值是通过标准的 ILogger 的 SetName() 函数设置的：当一个调用者调用 SetName() 时，我们需要改变与这个名字关联的值。

然而，为了设置预期值，我们需要一个新的函数——它并不是 ILogger 接口的一部分。我们提供一个名为 ExpectedName 的 accessor 来让我们能够对它进行设置。

这些都完成了之后，我们现在可以测试 access controller 是否正确地把它的名字赋给了 logger 对象。

```
[Test]
public void MissingPassword2() {
    MockLogger2 logger = new MockLogger2();
    logger.ExpectedName = "AccessControl";
    AccessController access =
        new AccessController( "secrets", logger);
    Assert.IsFalse(access.CanAccess( "dave", null));
    logger.Verify();
}
```

注意观察：这个测试是如何在创建 access controller 之前把 logger 名字的预期值设置给 mock logger 对象的。在测试运行之后，代码调用 mock 对象

TestAccessController.cs

的 `Verify()` 函数来确认实际值等同于它的预期值。最后，我们看到的是绿色的标记：说明一切正常。

然而，我们仍然没有测试 `access controller` 是否给 `logger` 写了正确的消息：我们希望它会说 “Checking access for dave to secrets,” 以及 “Missing password. Access denied.”。在此我们遇到了一个有两个值的 `expectation`。幸运的是，框架给我们提供了 `expectation` 集合类来处理这种情况。我们将更新我们的 `mock logger` 来使用它们。

```

Line 1  using System;
        using DotNetMock;
        -
        - public class MockLogger3 : MockObject, ILogger {
5
        -     private ExpectationValue _name =
        -         new ExpectationValue( "name");
        -
        -     private ExpectationArrayList _msgs =
10         new ExpectationArrayList( "msgs");
        -
        -     // Mock control interface
        -
        -     public String ExpectedName {
15         set { _name.Expected = value; }
        -     }
        -
        -     public void AddExpectedMsg(String msg) {
        -         _msgs.AddExpected(msg);
20     }
        -
        -     // Implement ILogger interface
        -     public void SetName(String name) {
        -         _name.Actual = name;
25     }
        -
        -     public void Log(String msg) {
        -         _msgs.AddActual(msg);
        -     }
30 }

```

MockLogger3.cs

在第 9 行，我们创建了一个名为 `msgs` 的新 `expectation` 集合。如 `ExpectationValue` 一样，它同时保存有多对实际值和预期值；这和前面非常类似，唯一的区别只是它让我们能够同时保存好几对值。第 18 行的 `AddExpectedMsg` 函数让我们把预期值添加到列表中，而且我们修改了实际的 `Log` 调用，用于把实际值添加到列表之中。

现在我们可以修改测试来验证产生的是正确的 `log` 消息了。

```

[Test]
public void MissingPassword3() {
    MockLogger3 logger = new MockLogger3();
    logger.ExpectedName = "AccessControl";
    logger.AddExpectedMsg(
        "Checking access for dave to secrets");
    logger.AddExpectedMsg(
        "Missing password. Access denied");
    AccessController access =
        new AccessController( "secrets", logger);
    Assert.IsFalse(access.CanAccess( "dave", null));
    logger.Verify();
}

```

注意, 我们已经添加了两个对 `AddExpectedMsg()` 的调用。当 `Verify()` 在测试最后被调用时, mock logger 将会检查 access controller 实际 log 的消息与这些消息是否完全相同。

有时 mock 对象需要处理更多的调用。例如, 你可能想要验证一个销售报告函数反复运行于美国的 50 个州的情况, 而又不希望实际地给 `ExpectationArrayList` 一个个添加每个州的名字。幸运的是, 框架实现了一个 `ExpectationCounter` 类。你可以设置一个预期的计数值, 然后调用 `Inc()` 函数来步增实际的计数。

■ 预提供的 Mock Objects

使用标准的框架来做测试的一个好处是你可以从创建一个标准的 mock 对象库开始, 然后在多个项目中重用它们。事实上, 在开源世界中, 你甚至可能会发现: 许多其他人已经创建好了的 mock 对象正是你所需要的, 而且它们是免费的。实际上, `DotNetMock` 包就带了一些 (不多) 这种拿来即用的 mock 对象包, 位于 `DotNetMock.Framework` 之中。这儿, 我们将看看其中的一个——`Data`, 它实现了 .NET 的 `System.Data` 中的许多接口。

让我们接着来实现 access controller 吧。在验证了密码确实已经提供了之后, 我们现在需要访问一个数据库表, 并验证确有一行给出了这个用户, 并且密码一样和有权访问资源。

NUnit's 内建的 Mocks

NUnit 2.2 包含内建的、轻量级的、动态 mock 对象的功能。参见 NUnit 中的文档以了解所有的细节，但是内建的系统至少具有以下特性：

- 动态创建任何接口或者 MarshalByReference 类的默认 mock 实现。
- 允许任意的调用，或者对于任何非显式期望的函数，将拒绝调用。
- 期待是以给定的顺序来调用函数。
- 期待一个函数不会被调用。
- 给用特定参数或者任意参数调用的函数指定返回值。
- 给用特定参数或者任意参数调用的函数指定要抛出的异常。
- 改写 DynamicMock 以给任意函数提供自定义的实现。

如果你想不用下载，并也不想学习一个功能完整且庞大的产品，但是却想尝试 mock 对象，这可能就是你最佳的选择了。

```

using System;
using System.Data;
using System.Data.SqlClient;

public class AccessController1 {
    private ILogger      logger;
    private String       resource;
    private IDbConnection conn;

    public static readonly String CHECK_SQL =
        "select count(*) from access where " +
        "user=@user and password=@password " +
        "and resource=@resource";

    public AccessController1(String resource,
                              ILogger logger,
                              IDbConnection conn) {

        this.logger    = logger;
        this.resource   = resource;
        this.conn       = conn;
        logger.SetName( "AccessControl");
    }

    public bool CanAccess(String user, String password) {
        logger.Log( "Checking access for " + user +
                    " to " + resource);

        if (password == null || password.Length == 0) {
            logger.Log( "Missing password. Access denied");
            return false;
        }

        IDbCommand cmd = conn.CreateCommand();
        cmd.CommandText = CHECK_SQL;
        cmd.Parameters.Add(
            new SqlParameter( "@user",      user));
        cmd.Parameters.Add(
            new SqlParameter( "@password", password));
        cmd.Parameters.Add(
            new SqlParameter( "@resource", resource));
        IDataReader rdr = cmd.ExecuteReader();
        int rows = 0;
        if (rdr.Read())
            rows = rdr.GetInt32(0);
        cmd.Dispose();
        if (rows == 1) {
            logger.Log( "Access granted");
            return true;
        }
        else {
            logger.Log( "Access denied");
            return false;
        }
    }
}

```

AccessController1.cs

这份测试代码可能比前面那个要复杂得多，主要是因为我们想把各种用于访问数据库的对象（connection，command，各种参数以及返回结果的

reader) 编织到一起。我们还要设置一套合理的 expectations 来确保底层的代码正确地调用了数据库层。

```

Line 1  using System;
        using NUnit.Framework;
        using DotNetMock.Framework.Data;
        -
        -
        5  [TestFixture]
        -  public class TestAccessController1 {
        -      [Test]
        -      public void TestValidUser() {
        -          MockLogger3 logger = new MockLogger3();
        10      logger.ExpectedName = "AccessControl";
        -      logger.AddExpectedMsg(
        -          "Checking access for dave to secrets");
        -      logger.AddExpectedMsg( "Access granted");
        -
        15      // set up the mock database
        -      MockDbConnection conn = new MockDbConnection();
        -      MockCommand cmd = new MockCommand();
        -      MockDataReader rdr = new MockDataReader();
        -
        20      conn.SetExpectedCommand(cmd);
        -      cmd.SetExpectedCommandText(
        -          AccessController1.CHECK_SQL);
        -      cmd.SetExpectedExecuteCalls(1);
        -      cmd.SetExpectedParameter(
        25          new MockDataParameter( "@user",      "dave"));
        -      cmd.SetExpectedParameter(
        -          new MockDataParameter( "@password", "shhh"));
        -      cmd.SetExpectedParameter(
        -          new MockDataParameter( "@resource", "secrets"));
        30
        -      cmd.SetExpectedReader(rdr);
        -      object [,] rows = new object[1,1];
        -      rows[0, 0] = 1;
        -      rdr.SetRows(rows);
        35
        -      AccessController1 access =
        -          new AccessController1( "secrets", logger, conn);
        -      Assert.IsTrue(access.CanAccess( "dave", "shhh"));
        -      logger.Verify();
        40      conn.Verify();
        -      cmd.Verify();
        -      }
        -  }
    
```

TestAccessController1.cs

在第 3 行, 我们引入了 DotNetMock 框架的 Data 组件。在测试函数内, 我们和以前一样由创建和设置 mock logger 入手。第 16 行, 我们创建了三个 mock 数据库对象: connection, command (用来发出对数据库的 SQL 请求), reader (用来返回查询的结果)。

我们现在需要把这三个对象关联起来。第 20 行告诉 connection 对象: 当它需要产生一个 Command 对象的时候, 它应当使用我们的 mock command 对象: cmd。我们然后设置 command 对象的 expectations: 它应当接收到的 SQL, 它应当被执行的次数, 以及它应当接收到的参数。

并非一切都是尽善尽美的

细心的读者可能奇怪，为什么我们的新的 `AccessController` 类要自讨苦吃地使用 `Reader` 对象获得执行查询的次数。为什么我们不干脆直接用 `command` 对象的 `ExecuteScalar` 函数返回计数呢？

不幸的是，`IDbCommand` 的 `mock` 对象实现并不完整（至少在写本书的时候是如此）。虽然 `ExecuteScalar` 是实现了的，但是它返回的永远是 `null`。这意味着我们不能在测试中使用它。

从 31 行开始，是设置 `reader` 对象的部分。它首先与 `command` 关联（因而当 `mock command` 被执行时，它将返回这个 `reader` 对象）。我们然后设置它的结果集，一个二维的对象数组，包含由查询返回的所有行以及每行中的所有列。在此，结果集仅仅包含一行一列，也就是那个计数结果，但是我们仍然需要把它包装到二维数组中。

最终，在 36 行，我们创建了自己的 `access controller`，并且检查“dave”是否可以用密码“shhh”访问资源“secrets”。因为这些值与我们给查询设置的值互相对应，所以 `access controller` 应该能够使用我们的 `mock` 数据库对象，它将返回一个计数“1”，同时说明访问被接受了。测试的最后，通过调用 `Verify()` 方法，我们验证 `logger`，`connection`，以及 `command mock` 对象在被测试函数中是否使用正确。

■ 动态 Mock 对象

有时会遇到这样的情况——当你要测试一些使用了原先写好了的接口的内容时，发现并没有预先写好的 mock 对象。通常，你可以很容易地编写一个新的 mock 对象出来。但是假如你要创建的 mock 对象的接口很庞大，有几十个函数和 accessor，怎么办呢？那可能意味着编写一个实现这个接口的 mock 对象会耗费大量的时间。如果你仅仅需要接口中的一两个函数来运行你的测试的话，那就尤其烦人了。

这就是动态 mock 对象的用武之地了。它们使你可以创建一个用起来就像实现了完整接口的对象，但实际上它又是完全通用的。你仅仅需要告诉这个对象，当你的代码调用了其中的函数时如何做出反应。这可以节省可观的时间，也使得你需要维护的代码更少。

为了能够更直观，假设我们有一个与客户相关的程序。在某个地方，一个开发者创建了一个巨型的 `ICustomer` 接口，其中定义了一大堆 accessor 来获得客户的信息。它大概是这个样子：

```
using System;

public interface ICustomer {
    String Title           { get; }
    String FirstName       { get; }
    String MiddleInitial   { get; }
    String LastName        { get; }
    String NameSuffix       { get; }
    String SSN             { get; }
    Address HomeAddress     { get; }
    Address WorkAddress     { get; }
    Date FirstContacted    { get; }
    Date LastContacted     { get; }
    int ContactCount       { get; }
    // and so on, and so on, for 30 more accessors
}
```

iCustomer.cs

我们当前的任务就是实现一个类，它能生成给客户的信件。作为该项任务的一部分，我们需要能够同时生成两种问候格式：长的和短的（因而我们的信可以在信封上使用“Mr Joe Smith, III”，在信内使用“Dear Mr Smith”）。我们的代码大概是这样的。

```

using System;
using System.Text;
public class Mailer {
    private ICustomer customer;
    public Mailer(ICustomer customer) {
        this.customer = customer;
    }
    public String ShortGreeting() {
        return customer.Title + " " + customer.LastName;
    }
    public String FullGreeting() {
        StringBuilder result = new StringBuilder();
        Append(result, customer.Title);
        Append(result, customer.FirstName);
        Append(result, customer.MiddleInitial);
        Append(result, customer.LastName);
        if (customer.NameSuffix.Length > 0)
            result.Append( " " + customer.NameSuffix);
        return result.ToString();
    }
    private void Append(StringBuilder result,
                        String field) {
        if (field != null && field.Length > 0)
            if (result.Length > 0)
                result.Append( " ");
            result.Append(field);
    }
}

```

为了测试这些代码，我们需要创建某种客户对象并把它传给我们的函数。因为接口相当大，这可能需要很大的工作量。这时就需要动态 mock 对象出来帮忙了。

动态 mock 包³通过创建代理对象来完成任务。这些代理对象被设计成它们真实世界中对应物的替身。在使用动态 mock 对象时，我们可以使用一个代理来取代测试中的真实对象。然而，我们仍然需要控制这个产生的代理对象，因为我们需要告诉它如何响应请求。这时该要 controller 登场了。

controller 负责控制动态 mock 对象。你使用 controller 来创建一个 mock 的实例，并告诉 mock 如何去做。

³ 直到我们编写本书为止，总共有两种动态 mock 包。一种是 DotNetMock 附带的，另一种是 NMock(<http://nmock.truemesh.com/>)。尽管它们之间的 API 有细微的差异，但是底层代码都是一样的。

在接下来的代码中，`controller` 是一个创建于 13 行的、类型为 `IMock` 的对象。在其下一行，我们告诉 `controller` 创建实际的 `mock` 对象——那个实现了 `ICustomer` 接口的对象。

```

Line 1  using System;
        using DotNetMock.Dynamic;
        using NUnit.Framework;
        -
        5
        - [TestFixture]
        - public class TestCalculator {
        -
        -     [Test]
    10     public void AgeCalculation() {
        -
        -         IMock mock =
        -             new DynamicMock(typeof(ICustomer));
        -         ICustomer customer = (ICustomer)mock.Object;
    15     mock.SetValue( "Title",          "Mr.");
        -     mock.SetValue( "FirstName",      "Fred");
        -     mock.SetValue( "MiddleInitial",   "E");
        -     mock.SetValue( "LastName",        "Bloggs");
        -     mock.SetValue( "NameSuffix",      "III");
    20
        -     Mailer mailer = new Mailer(customer);
        -     Assert.AreEqual( "Mr. Bloggs",
        -                     mailer.ShortGreeting());
        -     Assert.AreEqual( "Mr. Fred E Bloggs, III",
    25                     mailer.FullGreeting());
        -     }
        - }

```

TestMailer.cs

一旦拥有了 `controller`，我们还能编程控制 `mock` 对象行为。就我们的测试目的而言，这是相当容易的：仅仅需要它对各种包含客户名字的组件返回给定的值。从 15 行开始，`controller` 的 `SetValue` 调用就设置了一些值。然后我们把 `mock-customer` 传递给 `Mailer`，并测试两种问候格式。

值得注意的一件关键事情是，我们不再需要实现 `ICustomer` 接口中的所有函数了。我们仅仅需要设置我们要用的那些。这使得工作简单了许多，同时也是用来做测试的动态 `mock` 的关键好处之一。

除了设置函数调用的返回值，你还可以用动态 `mock` 对象做更多的事情。在 `mock` 对象中，你可以使用各种 `Expect...` 调用来把一系列行为和反应都编码到 `Mock` 对象中。接下来的代码虽然有点人为的做作，但它至少展示了 `Expect` 的一些用法。

```

Line 1  using System;
        using NUnit.Framework;
        using DotNetMock.Dynamic;
        -
        5  interface ITaxCalculator {
            decimal CalculateTax(decimal amount, String state);
        }
        -
        public class ExpectExamples {
10
        public ExpectExamples() {
            -
            IMock mock = new DynamicMock(typeof(ITaxCalculator));
            -
15      mock.ExpectAndReturn( "CalculateTax",7.25,100, "TX");
            mock.ExpectAndReturn( "CalculateTax",7.00,100, "NC");
            mock.ExpectAndThrow( "CalculateTax",
            -
            new ArgumentOutOfRangeException(), 100, "XX");
            -
20      ITaxCalculator calc = (ITaxCalculator)mock.Object;
            -
            // The following asserts simply illustrate
            // how the mock object would be used.
            // This is not a typical test.
25      Assert.AreEqual(7.25,
                        calc.CalculateTax(100, "TX"));
            Assert.AreEqual(7.0,
                        calc.CalculateTax(100, "NC"));
            -
30      try {
            calc.CalculateTax(100, "XX");
            Assert.Fail( "Should have thrown an exception");
            }
            catch (ArgumentOutOfRangeException) { ; }
35
            mock.Verify();
        }
    }

```

在 13 行,代码针对一个(有一点傻傻的)营业税计算器创建了一个 mock 对象。然后,从 15 行开始的后续三行,将编码这个 mock 对象,让它预期三个针对 CalculateTax() 的调用。在第一次调用中,函数预期传入参数为 100 和“TX”,并且应当返回值 7.25。在下一次调用中,它应当在传入参数 100 和“NC”之后返回 7.0。在最后一次调用中,它应当抛出 ArgumentOutOfRangeException 异常,因为它会被传入一个无效的状态缩写(“XX”)。

通常,我们之后会把这个 mock 对象传递给其他一些被测对象。然而,为了保持例子的简短,这儿就直接在它之上运行一些断言。如我们使用对象一样,它验证 CalculateTax() 函数确实被调用了三次,而且每次传入的都是预期的参数。同时还使 mock 对象在每次调用中返回预期值(或者抛出所需的异常)。

6.4 什么时候不应使用 Mock

Mock 对象是一项相当诱人的技术，但由于它们需要编写代码，所以毫无疑问会给你的项目带来负担。每次当你发现自己在盘算着编写一个 mock 对象来帮助测试的时候，停下来用几分钟时间好好想想是否有其他的解决办法。具体来说，问问你自己这个简单的问题：“我真的需要 mock 对象吗？”有时你可以通过简单的重构来消除对 mock 对象的依赖。

作为一个（尽管有些做作）例子：假设我们正在编写“用相对较慢的线路下载文件到手持设备”的代码。因为一些硬件方面的约束，在发送了一块数据之后，并且在再次尝试与设备交互之前，我们必须等一小会。这段被迫等待的时间的长短取决于发送数据的数量——做硬件的那些家伙给我们提供了一个针对等待时间的数据表格。

我们先编写一个子程序，它的等待时间依赖于发送数据的数量：

```
public void WaitForData(int dataSize) {  
    int timeToWait;  
    if (dataSize < 100)  
        timeToWait = 50;  
    else if (dataSize < 250)  
        timeToWait = 100;  
    else if (dataSize < 600)  
        timeToWait = 150;  
    else  
        timeToWait = 200;  
    Thread.Sleep(timeToWait);  
}
```

Example.cs

现在，让我们来测试这个函数，但是有一个问题：查看它是否工作正常的唯一办法是检查它是否对 dataSize 参数的各种值休眠了正确长度的时间。这不是一个好写的测试，因为我们必须设置一个 fudge factor(误差系数)，而且我们度量的等待时间不可能是精确的；我们甚至必须设置某种监视线程来确保睡眠不会进行太久。这儿还有值得考虑的耗时问题：如果运行我们的测试导致 Thread.Sleep 被调用好几次，那么我们的测试就需要相当长的时间才能完成——这样是不受欢迎的。

在读完本章之后，你可能会首先尝试用 mock 对象来解决这些问题。如果我们用某种 mock 对象代替 Thread，我们可以验证它的 Sleep() 函数是否以预期的值进行调用。但是类 Thread 不是一个接口，即便是，它也有一大堆属性和成员。

现在正是发现问题的时候：我们可以重新设计我们的代码以使得它稍微好测试一些？当然，我们可以！

```
public int HowLongToWait(int dataSize) {
    int timeToWait;
    if (dataSize < 100)
        timeToWait = 50;
    else if (dataSize < 250)
        timeToWait = 100;
    else if (dataSize < 600)
        timeToWait = 150;
    else
        timeToWait = 200;
    return timeToWait;
}

public void WaitForData(int dataSize) {
    Thread.Sleep(HowLongToWait(dataSize));
}
```

Example.cs

在这份代码中，我们把等待部分分割成两个函数。一个根据数据的大小来计算要等待的毫秒数量，另一个调用上面这个函数，并把返回值作为参数传递给 Thread.Sleep()。如果我们假定框架的 Sleep() 函数工作正常，那么也许就不再需要测试第二个函数了：我们能够用眼睛盯着它并且看到它的确如它应该的那样。这样就只须测试“计算等待的时间的函数”，而这个任务是非常简单的。

```
[Test]
void WaitTimes() {
    Waiter w = new Waiter();
    Assert.AreEqual(50, w.HowLongToWait(0));
    Assert.AreEqual(50, w.HowLongToWait(99));
    Assert.AreEqual(100, w.HowLongToWait(100));
    Assert.AreEqual(100, w.HowLongToWait(249));
    Assert.AreEqual(150, w.HowLongToWait(250));
    Assert.AreEqual(150, w.HowLongToWait(599));
    Assert.AreEqual(200, w.HowLongToWait(600));
}
```

Example.cs

一个简单的重构让我们获得了一个更佳的设计，并且消除了与编写测试相关的一堆痛苦。

第 7 章

好的测试所具有的品质

Properties of Good Tests

单元测试是非常有威力的魔法，但是如果使用不当也会浪费你大量的时间，从而对项目造成巨大的影响。另一方面，如果没有恰当地编写和实现单元测试，在维护和调试这些测试上面，很容易就会浪费很多的时间，从而影响产品代码和整个项目。

我们不能让这种情况出现；记住，你做单元测试的首要原因就是让你的工作更轻松！幸运的是，这儿有一些你应该遵循的简单原则，它们能够让这些麻烦不会出现在你的项目之中。

好的测试应该具有以下品质，合称为 A-TRIP：

- 自动化 (Automatic)。
- 彻底的 (Thorough)。
- 可重复 (Repeatable)。
- 独立的 (Independent)。
- 专业的 (Professional)。

让我们来看看每个词的含义。

7.1 自动化

A-TRIP

单元测试需要能够被自动地运行。我们说“自动化”至少包括两方面的含义：调用测试自动化和检查结果自动化。

调用一个或多个单元测试对你而言必须是非常容易的，因为你将会成天、甚至日复一日地重复这项工作。因此，要运行你想要的测试，真的不能比在 IDE 中按下一个按钮或者在提示符下键入一行命令更复杂。一些 IDE 甚至能够设置为在后台持续地运行单元测试。

维护这个环境最重要的在于：不要引入一个由于需要手动步骤而打破这个自动化模型的测试。对于测试需要的任何条件（数据库、网络连接，等等），都应该让它们成为测试自身的一个自动化组成部分。如第 6 章中讲述的 Mock objects，就可以帮助你使得测试不受真实环境改变的影响。

但是测试并非只由你一人来运行。别处的机器也会针对所有的签入（checked-in）代码持续地运行所有的单元测试。这个自动化的、无人员参与的检查像一台“检查装置（back stop）”；一种能够用来确保任何代码签入都不会损坏位于任何机器的任何测试的机制。显然，在理想世界中这不是必须的，因为你可以依靠每个开发者运行自身必需的所有测试来获得保证。

但这不是理想世界。可能对位于项目的一个遥远角落里的某些必要的测试，开发者并没有运行它们。也可能开发者有一些只存在于自己机器上的代码使得所有测试都能通过，但是他们并没有签入这些代码，因而即便测试在他们自己的机器上运行是正常的，相同的测试也可能会在其他地方失败。

你可能需要了解管理持续构建和测试的一些方法，这方面可以借鉴 Cruise Control¹ 系统和其他一些开源产品。

另一方面，“自动化”是指测试必须能够自己决定它是通过了还是失败了。因为如果雇佣一个人（你或者其他不幸的牺牲者）来通读测试的输出，

¹ <http://ccnet.thoughtworks.com>

然后验证代码能否正常工作，是一种能导致项目失败的做法。而且一致性回归（consistent regression）的一个重要特征就是让测试能够检查自身。另外，人类对于这些重复性的工作并不非常擅长，况且除此之外我们还有更加重要的事情要做——还记得你的项目吗？

让测试自己来运行和检查的想法是至关重要的，因为这意味着你不需要对它有任何思考——它只碰巧是项目的一部分而已。只有这样，测试才能够完全胜任自己的职责，作为我们项目的安全网的主要部分。（版本控制和自动化是“安全网”的另外两个主要部分。）它能够在你坠下时及时抓住你，但是它们不会挡住你前进的道路。因此，当自动化被破坏的时候，就需要你百分之百的注意力。

7.2 彻底的

好的单元测试是彻底的；它们测试了所有可能会出问题的情况。但是那要多彻底呢？一个极端是，对于每行代码、代码可能到达的每个分支、每个可能抛出的异常等等，都可以作为测试的对象。另一个极端是，你仅仅测试最可能的情况——边界条件、残缺和畸形的数据等等。事实上，这是一个基于你的项目需求的决策问题。

A- **T** RIP

如果希望获得更完整的覆盖，你或许可以从代码覆盖工具获得一些帮助（例如“NCover”，<http://workspaces.gotdotnet.com/ncover>）。

这些工具能够帮助你知道被测试的代码到底有多少是被实际执行了的。

我们还需要认识到一个很重要的事实：bug 并不是均匀分布于源代码之中的；相反，它们更倾向扎堆于一块问题区域之中（关于这一点有一个很有意思的故事，参见下页方框中的内容）。

这种现象引出了著名的战斗口号“不要修修补补，完全重写！”经常，抛弃一块有很多 bug 的代码并且重新编写会令开销更小、痛苦更少。当然，现在重新来编写代码就更加安全了：你有了一套单元测试来保证新的代码获得应有的正确行为。

报告了的 Bug vs. 单元测试覆盖率

我最近有一个客户不大相信单元测试的威力。团队的一小部分成员擅长并且习惯于给自己的模块编写单元测试，大部分成员对待测试则比较随意，而另外一些甚至完全拒绝引入任何单元测试。

作为每小时构建过程的一部分，我们编写了一个简单的 Ruby 脚本来执行一个针对测试覆盖率的 quick-and-dirty 分析：它计算出每个模块的测试代码中的 assert 个数与产品代码的函数个数的比率。测试良好的每个函数应该对应 3、4 甚至更多个 assert；没有测试的函数的代码将一个也没有。这个分析在每次构建的时候都会运行并且产生一个柱状图，然后把测试最多的模块排在最前，没有测试的模块排在最后。

在收集了几个星期的图表之后，我们把柱状图不加解释地呈给项目经理看。他非常惊讶地看到所有有问题的模块都扎堆式地聚集在最后——而他原先还以为我们是根据 QA 部门和客户支持的 bug 报告生成这个图表的，后来才知道我们是根据测试覆盖率的柱状图生成该表的。他产生这种误解也是正常的，因为对位于图最前的那些模块（测试良好的），他基本上是知之甚少，说明 bug 也几乎没有；因为如果真有问题，是肯定会报告给他并且让他知道的。但是尾部的那些模块（没有单元测试的那些），他和支持部门的经理就非常了解了，而他们甚至还基于这些模块，让当地的药店储存更多的胃药供应。

结果是近似线性的：单元测试越多，问题越少。

7.3 可重复

每个测试应该独立于所有其他的测试，而且还必须独立于周围的环境。A-T R IP
 目标只有一个，就是测试应该能够以任意的顺序一次又一次地运行，并且产生相同的结果。这意味着测试不能依赖于不受你直接控制的任何外部因素。

在必要的时候，应该使用 mock object 来隔离测试的外界因素，并且使测试能够独立于环境。如果你不得使用来自真实世界的一些因素（比方说，数据库），那么请确保你不会受到其他开发者的影响。也就是说，每个开发者都需要使用自己的“sandbox²”，这可以是针对 Oracle 的数据库实例，也可以是位于某个非标准端口的 web 服务器。

如果测试结果不具备重复性，那么对于一些糟糕的情况，你可能会对测试结果感到万分惊讶。更糟糕的是，这种类型的惊讶通常是假的——它不是真正的 bug，而只是测试本身的问题。显然，你可不能把时间浪费在追逐这些幻影般的问题上面。

每个测试应当每次产生相同的结果。如果没有，那么它应当告诉你代码中存在有真正的 bug。

7.4 独立的

测试应该是简洁而且精炼的，这意味着每个测试都应该有很强的针对性，并且独立于环境和其他的测试（记住，其他的开发者可能在同一时刻运行着这些相同的测试）。A-TR I P

在编写测试时，确保你一次只测试了一样东西。

但是这并不意味着在一个测试中你只能使用一个 assert，而是说明了：一个测试函数应该专注于产品代码中的一个函数，或者组合起来并共同提供某个特性的一组函数。

² 可以认为是包含了自己很多混杂内容的代码库。

有时，一个测试函数可能仅仅测试了产品代码中的一个复杂函数的一小部分，此时你可能需要多个测试函数来完全彻底地验证该函数。

理想情况下，你能够在潜在的 bugs 和测试代码之间有可跟踪的对应关系。换句话说，当一个测试失败了，应该立刻就可以知道代码中潜在 bug 的位置。

“独立的”还意味着没有测试依赖于任何其他测试；你应当可以在任何时候以任意顺序运行任何单个测试。你也不希望某个测试必须依赖于任何前面运行的其他测试、才能够开始运行。

我们已经展示了帮助你做到这点的机制：每个测试都有的 `setup` 和 `teardown` 函数以及每个类的 `setup` 和 `teardown` 函数。使用这些函数来确保每个测试获得一个全新的开始——而且不会影响任何可能后续运行的测试。

记住，你不能假定 NUnit 的测试会以某个特定的顺序运行，而且随着你开始不断合并测试和 suites，从而使测试数目不断增长之后，你真的无法依赖测试运行的顺序。

John Donne³ 可能对于人来说是对的，但是对单元测试并非正确：每个测试都应该是一座孤岛。

7.5 专业的

A-TRI **P**

你为单元测试编写的代码是真实的；一些人可能辩称它甚至比你卖给客户的代码还要真实。这意味着必须使用和产品代码相同的专业水准来编写和维护测试代码。在测试代码中，针对好设计的所有普遍规则——维护封装，采用 DRY 原则、降低耦合，等等——都必须如在产品代码中那样得到遵循。

掉入编写非常冗余的测试代码的陷阱是很容易的；也就是说，代码总是一遍又一遍地做相同的事情，其中甚至连一个函数或者对象也没有。这是

³ 译注：一个处于 16 世纪和 17 世纪的英国诗人，说过一句名言：*No man is an Island!* 所以作者后面说 *every test should be an island*。

糟糕的事情。测试代码必须以同产品代码相同的风格来编写。这意味着你需要抽取出共同且重复的代码，并把它们的功能放入一个函数之中，从而可以在多处调用该函数。

你可能发现你累积了几个相关的测试方法，此时你应该把它们封装到一个类中。此时没有必要犹豫，大胆地去创建一个新类来实现封装，即使它只用于测试。这种做法不仅仅是可行的，还应该得到鼓励：测试代码是真实的代码。在一些情况下，你甚至需要创建一个更大的框架，或者创建一个数据驱动的测试设施（还记得 47 页中的那个针对 `TestLargest` 的简单的 `file reader` 吗？）。

不要浪费时间测试那些对你没有帮助的方面。记住，不要为了测试而测试。测试必须是完整的，因为它必须能够测试到一个函数中可能会出现 bug 的所有情况。而对于不大可能包含 bug 的情况，就不要去测试了。这意味着你通常不应该浪费时间来测试诸如 `property accessors` 的简单内容，如：

```
public Money Balance {
    get { return balance; }
}
```

坦白地说，这儿不存在编译器不能捕获的错误。针对上面函数的测试只是浪费时间而已。然而，如果 `accessor` 在其中还做了一些其他工作，那么它顿时变得有意思起来了——而我们想要测试它。

```
public Money Balance {
    get {
        return posted.GetBalance() -
            unposted.GetDebits() +
            unposted.GetCredits();
    }
}
```

这可能就是值得测试的代码。

最终，意料中的结果是编写的测试代码至少和产品代码一样多。是的，你没看错，那是对的。如果你在你的产品中有 20 000 行代码，那么期待 20 000 行或者更多的单元测试代码来验证它是合情合理的。测试代码确实很多，这也是为什么需要让它保持简洁且精炼、设计良好且重构充分，如产品代码一样专业的部分原因。

7.6 对测试进行测试

到目前为止，我们的计划有一个重大的概念上的弱点。对代码进行测试从而确信它工作正常是一个伟大的想法，但是你必须编写代码来执行这些测试。但如果在我们所编写的测试代码中有 bug，会出现什么情况呢？这是否意味着：对于测试产品代码的测试代码，你是否需要编写新的测试代码来对它们进行测试呢？？？而如果这样延续下去，何时才能终止呢？

幸运的是，你不需要做得那么极端。有两件事情可以帮助你确保测试代码是正确的：

- 在修正 bug 的同时改进测试。
- 通过引入 bug 来证明测试。

■ 如何修正 Bug

在修正 bugs 的时候，你采用的步骤对于单元测试来说是非常重要的。许多时候，一个既有的测试会暴露代码中的一个 bug，此时你就可以简单地修正代码然后观察这个警醒的测试是否能通过。

当一个 bug “在荒漠中”被发现并被报告回来时，这意味着在网中有一个漏洞——一个被遗忘的测试。这正是你修补这个漏洞的机会，然后确保这个 bug 绝对不会再漏网了。而所需要做的就是以下四个简单的步骤：

1. 验明 bug。
2. 编写一个将失败的测试来证明 bug 的存在。
3. 修正代码，让测试通过。
4. 验证所有的测试仍然可以通过（也就是，你没有在修补的时候损坏其他的测试）。

这种应用真实世界的反馈来帮助改进测试的简单机制是十分有效的。随着时间的推移，你可以期待你的测试覆盖率能够稳定地增长，而从已经存在的代码中逃脱到荒漠的 bug 的数目会同时下降。

当然，随着你编写新的代码，你毫无疑问会引入新的 bug，其中也包括现有测试未能捕捉到的新类型的 bug。但是在修补任何 bug 的时候，问问你自己这个关键问题：

```

[Test]
public void Add() {
    // Create a new account object
    Account acct = new Account();
    // Populate with our test person
    acct.SetPerson(TEST_PERSON_1);
    // Add it to the database
    DatabaseHandler.Add(acct);
    // Should find it
    Assert.IsTrue(DatabaseHandler.Search(TEST_PERSON_1));
}

```

图7.1 针对把一个Person添加入数据库的测试

相同的问题还会在其他地方发生吗

然后，无论你修补的是一个老特性的 bug 还是一个新特性的 bug，都把已经学到的教训应用到整个项目中。如果能把你的新发现的知识应用到适当的单元测试之中，那么你所做的已经不仅仅是修复一个 bug，而是捕捉到了一类 bugs。

■ Spring the Trap (自设圈套)

如果你不能确信测试是否正确，最简单的事情是“spring the trap”：让产品代码产生你想要检测的那个 bug，然后验证测试是否如预期一般地失败。

例如，假定你已经有了一个测试函数，它会添加一个客户的账号到数据库中，然后如图 7.1 中的代码那样尝试查找该账户。可能你对于“查找”那部分测试是否正常工作不大确定——即它也许在记录没有被成功添加的情况下也会报告成功。

因此你可能要进入到 DatabaseHandler 的 Add() 函数并且把它“短路”：仅返回而不把记录添加到数据库之中。现在你应当看到断言失败了，因为记录没有被添加。

但是等等，你可能会有疑问，前一个运行的测试留下来的记录呢？它不会在数据库之中吗？不，它不会，因为以下几个原因：

- 你可能并不是真正的对实际的数据库进行测试。被上面的 test case 执行的代码可能处于所示的 add 函数和实际的底层数据库调用之间。这些数据库调用可能被一个 mock object 处理了，它的数据在两次运行之间实际上并没有被持久化保存。

- 测试是独立的。所有的测试可以以任何顺序运行，而且互相之间没有依赖，因而即使真实的数据库是这个测试的一部分，setup 和 tear-down 也必须确保你获得了一个“干净的 sandbox”来工作。前面的 spring the trap 的尝试将能够帮助证明这一点。

此时那些信奉极限编程的家伙会宣称：他们有序的测试优先的开发实践能避免这种糟糕测试的问题——有些测试在该失败的时候而不失败。在测试优先的开发之中，你编写代码仅仅是为了让原先失败的测试能够通过。一旦测试通过了，你就知道你刚添加的代码符合测试的要求。这就让你永远都能绝对确信你引入的代码修正了原先那个失败的测试，而另一方面正是这个测试才驱动你编写这些产品代码。

但是即使杯子和嘴唇之间也会发生许多事故。尽管测试优先的开发确实很大程度上改善了前面的一些问题，但仍然会有因巧合导致误导的可能。对于这些情形，你可以借助于故意的“springing the trap”来消除任何迟疑，从而确信一切如你所预期。

最后，记住要编写的是 A-TRIP（自动化、彻底的，可重复、独立的，专业的）的测试；在新 bug 和一类 bug 被发现的同时，不断添加你的单元测试；并且检验你的测试确实能发现它们所瞄准的 bugs。

然后向后一坐，静观你项目中的问题如魔法般消失。

第 8 章

在项目中进行测试

Testing on a Project

到目前为止我们所谈论的测试都是单独的、孤立的实践。当然在现实生活中，你可能会有要一同工作的队友。你们要一起来执行单元测试，而这往往会带来一堆的问题。

8.1 把测试代码放到哪儿

在一个小型的单人项目中，测试代码的位置和产品代码的封装可能并不是很重要，但在大一点的项目中，它就成为至关重要的问题。在以下的内容中，我们将看到多种组织产品代码和测试代码的方法。

一般而言，你不会由于测试的缘故去破坏任何的封装（或者如妈妈常说的，“don't expose your privates!”（不要泄漏你的隐私！））。在多数情况下，你应当能够通过执行它的公有函数来测试一个类。然而，如果在私有成员和保护成员中内藏了重要的功能，这就应该引起警觉，说明其内部还有一个类正在挣扎欲出。在这种情况下，打开封装获得测试通过并工作正常的代码，可能要好于安守封装得严严实实的、但没有经过测试且可能不能工作的代码。

■ 同一目录

假设你在编写一个类，名字为：

```
com.pragprog.wibble.Account
```

相应的测试位于：

```
com.pragprog.wibble.TestAccount
```

组织测试代码的最优先也即最简单的方法是把测试简单地包含在同一项目、同一程序集之中，紧挨着产品代码。

其好处是 `TestAccount` 能够访问 `Account` 的 `internal` 和 `protected internal` 成员变量和函数；但缺点是测试代码到处都是，堆积在产品代码的目录中。这可能是也可能不是一个问题，取决于你如何创建最终卖给客户的发布版本。

大部分时间，这个问题足以使得我们更偏爱其他的解决方案。但是对于小型项目而言，该问题也许就显得无关紧要。

■ 单独的程序集

下一个可能的方案是在一个独立于产品代码的程序集中创建你的测试。

其好处是能清楚地分离你的产品代码和测试代码。

缺点是现在测试代码位于不同的程序集之中；你不再能访问 `internal` 或者 `protected internal` 成员了，除非你的测试代码使用了产品代码的一个子类，并且该子类暴露了必要的成员。例如，假设你想要测试的类是这样的：

```
namespace FacilitiesManagment {
    public class Pool {
        protected Date lastCleaned;

        public void xxxx xx {
            xxx xxx xxxx;
        }
        ...
    }
}
```

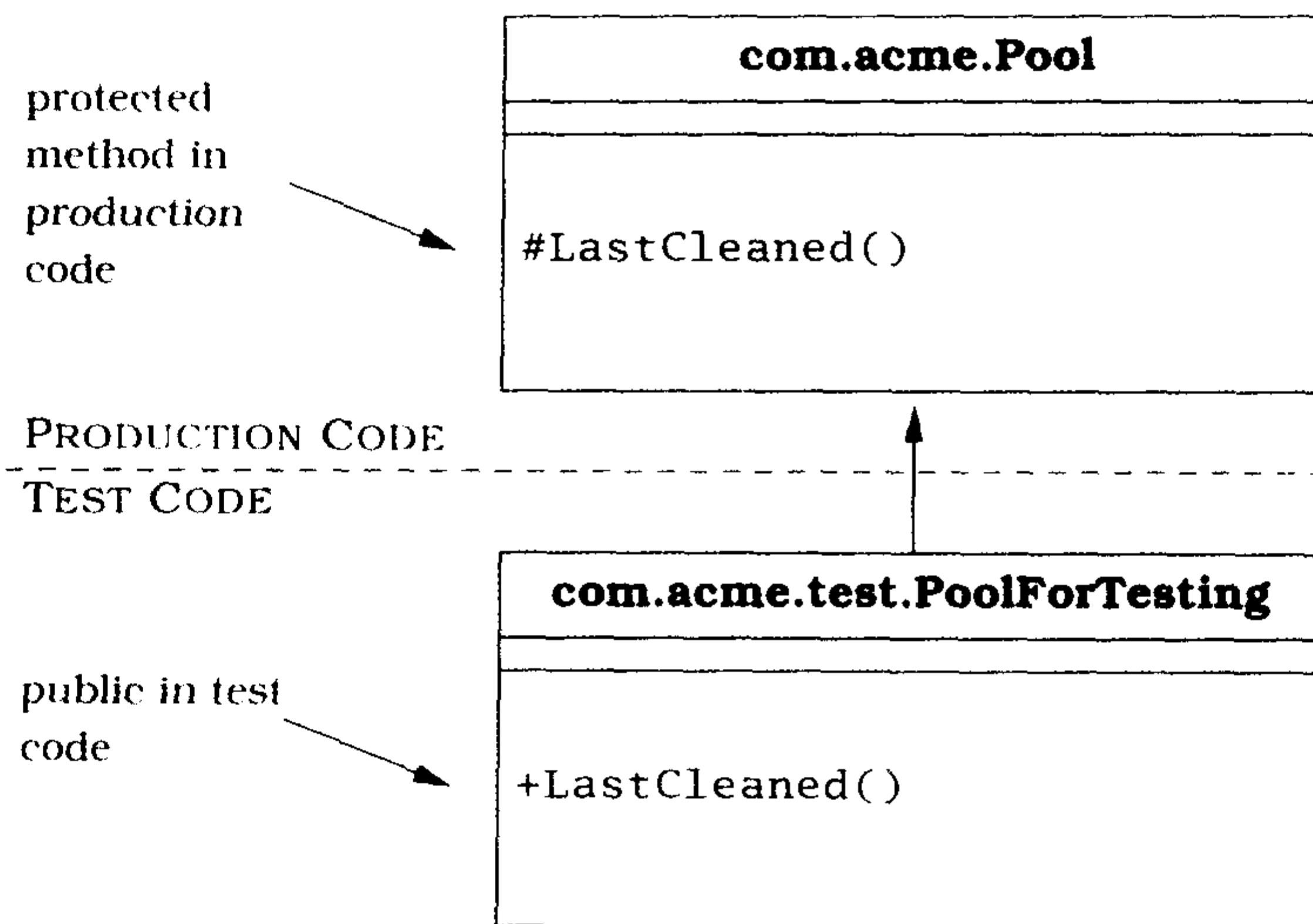


图8.1 子类暴露了用于测试的方法

为了测试，你需要获得 non-public 的数据，这些数据告诉你 pool 最后是什么时候清空的，但是并没有针对该数据的 accessor。（如果有的话，pool 联合会可能会以此控告我们，因为他们并不希望把这些信息变成公有的。）所以为了达到测试的目的，你要写一个子类来暴露它们。

```

using FacilitiesManagment;

namespace FacilitiesManagmentTesting {
    public class PoolForTesting : Pool {
        public Date LastCleaned {
            get { return lastCleaned; }
        }
    }
}
  
```

然后在测试代码中使用 `PoolForTesting`，而不直接使用 `Pool`（参见图 8.1）。事实上，你可以让这个类声明为私有的，而只让测试代码访问它（从而确保我们不会受到控告）。

无论团队决定采用何种形式，都要确保它能得到一致的保持。当你以某种方式建立系统部分测试的时候，你就不能用其他方式建立别的测试，而导致系统保持有两种不同组织风格的测试。正确的做法应该是：选中一种看上去适合你所处环境的风格，并且在系统中的所有单元测试中坚持这种风格。

8.2 测试的礼貌

独立测试和与其他人一起测试的最大区别在于使工作中的测试和代码同步。

当与小组成员一同工作时，你将使用某种版本控制系统，诸如 Visual SourceSafe 或者 CVS。（如果你对版本控制不熟悉，或者想获得有关如何搭建 CVS 并让它正常工作方面的帮助，参见[TH03]。）

在小组协同的环境中（甚至即便是在单人环境中），你应当在你签入代码的时候（或者说是让它能够被其他人访问时），确信这些代码已经有了一套完整的单元测试，而且全部测试都能通过。实际上，在加入了你的新代码之后，整个系统的每个测试都应该仍然可以通过。

规则很简单：一旦有任何人访问你的代码，则所有地方的所有的测试都需要能通得过。因为你一般都需要与小组的其他成员以及版本控制系统相当紧密地同步工作，所以这浓缩起来就是“所有的测试应该在任何时刻都能通过”。

许多小组确立了一些制度来“提醒”开发者，让他们知道破坏构建或者破坏测试的后果。这些制度一般是通过列举一些不好的代码形式。譬如，也许你就曾经签入（或者说是让代码能够被其他开发者访问）过这样的代码：

- 不完整的代码（比如，仅签入了一个类文件，而忘记签入它所依赖的其他文件）。
- 不能编译的代码。
- 代码能编译，但是会破坏已经存在的代码，比如使得已经存在的代码编译失败。
- 没有相应单元测试的代码。
- 不能通过单元测试的代码。
- 通过了自己的测试，但是导致系统其他地方的其他测试失败的代码。

如果发现你触犯了这些可憎错误中的任何一条，你将会受到惩罚，罚你给整个小组提供明天早上的油炸圈、啤酒、苏打，或者是冰镇的玛格丽塔酒，还可能会让你给构建用的机器当保姆，要不就是其他象征性的服务。这种气

氛轻松、略带强迫的制度通常足以让大家对不小心的事故具有足够的警惕。但如果你不得不对代码进行一个不兼容的修改，又会怎样呢，或者假如你做出了一个改动，而这个改动确实导致了系统其他地方的测试失败，又会怎么样呢？

真正确切的答案取决于你在项目中所使用的方法学和过程。但是不管怎么样，你必须以某种方式协调你的改动以及由其他人负责的代码——你必须这么做！意思是说一次把所有必要的改动都做完，因而小组的其他人看到的是一致的（真正能工作的）景象，而不是一个破碎的、不带来任何好处的，诸如“工作进行中”这样的提示。（要获得关于如何使用版本控制来设置尝试性的开发者分支方面的信息，参见[TH03]。）

在某些时候，真实世界中的事情并不总是尽如人意的。要完成所有不兼容的部分也许需要几小时甚至几天才行，而在这期间正常的构建都被破坏了。如果只能这样的话，那么要确保项目组人员之间有着良好的沟通，让每个人都知道构建在一段时间内无法进行，从而每个人都能根据需要为此做出自己的打算。如果与你无关，这可能正是你开车出去加加油，或者到沙滩去玩一两天的好机会。如果你要参与其中，那么尽快把事情搞定，让其他的人能够快速地从海滩回来投入到工作当中！

8.3 测试的频率

你应该多频繁地运行单元测试呢？这取决于你在干什么和你的个人习惯，但除此之外还有一些我们认为能带来帮助的一般性原则。你执行足够的测试是希望确信你捕捉了所有你需要捕捉的错误，但是也不能太过了，以至于影响了产品代码的正常编写。

■ 编写新的函数

编译并运行本地的单元测试。

■ 修正 bug

运行测试来让 bug 现形；修正并再次运行单元测试。

■ 每次成功编译之后

运行本地的单元测试。

■ 每次对版本控制的签入

运行所有的模块或者系统的单元测试。

■ 持续不断地

应当有一台专门的机器来运行完整的构建和测试。每次都应该从头开始，并且整天自动运行（要么是周期性的，要么是每当有版本控制的签入行为的时候）。

注意在规模大一点的项目中，你可能无法在几个钟头内完成一次对整个系统的编译和测试。你可能只能在夜间进行完整的构建和测试。对于再大一些的项目，也许就不得不用数日来完成这项工作了一——这是一个窘境，因为自动构建的间隔周期越长，一个问题从产生到发现的“反馈间隔”也就越长。

多多少少做一些持续构建的原因是它能够快速地发现任何问题。与等待其他开发者偶然发现一个构建问题并报告回来相比，你显然更加期望能够自己发现问题。通常的做法是让一台构建用的机器来充当一个持续的开发者，因为这增加了发现问题的几率，而且不需要一个真正的开发者来做这件事情。

当构建机器确实发现了一个问题时，整个团队都能得到一个警告，表明此时得到的任何新加入代码都是不安全的，并让他们以现有代码继续工作。这比起把你困在得到了不能运行的新代码的两难境地要好多了。

要获得有关如何设置自动构建和测试系统，在夜间持续地构建，以及总体上的自动化方面的信息，参见[Cla04]。

8.4 测试与遗留代码

到目前为止，我们谈论的都是在编写新代码的情况下如何做单元测试。但是如果你的项目已经有了大量的代码，而这些代码并没有进行过单元测试，我们还没有谈到应该如何来进行这类代码的测试。

这完全取决于那些代码所处的状况。如果它是合理的、重构充分的以及模块化的，那么你可以获得所有你需要的独立部分，然后很容易地添加单元测试；另一方面，如果那是一团糟，所有的东西纠缠在一起，那么不重写一些代码可能就没法做测试了。大部分既有项目都没有经过什么重构，但通常已经足够模块化了，因此你可以直接添加单元测试。

对于新写的代码，你显然应该同时编写单元测试。这也许意味着你需要暴露或者打开现存系统的某些部分，或者创建 `mock objects` 来测试你的新功能。

对于已经存在的代码，你可以选择为可测试的一切代码系统地添加大量的单元测试。但这不是非常有实效的（`Pragmatic`）。最好是只给最有可能出问题的代码添加测试，这样才能通过有限的投入获得更大的回报。

在这种环境中，单元测试的一个最重要的作用就是防止倒退（`back-sliding`）：避免维护性的更正和修缮导致现存特性产生 `bug`。在日常性的新代码开发中，我们通常会用 `NUnit` 的单元测试来做回归测试（为的是确信新的代码没有损坏任何已经工作正常的代码），但是在处理遗留代码时，回归测试甚至更为重要。

而且它不需要覆盖所有的遗留代码，只需要关注那些最有可能出问题的部分。考虑下面这个来自于一个注重实务的（`pragmatic`）开发者的真实故事（文中的团队在项目中用的是 `Java` 和 `JUnit`，但是他们同样可以是 `C#`、`Cobol`、`C++`、`Ruby` 或者任何其他的编程语言）。

■ 回归测试节省了时间

“Tibbert Enterprises¹ 出售好几种应用程序，但它们都基于一个公共的底层库，这个库是用来访问对象数据库的。

一天，我偶然听到一些应用程序组的开发者谈论一个他们遇到的持久化方面的问题。在产品的底层接口中，你可以使用对象的名字来查找对象，

¹ 并非他们的真实名字。

名字中包括对象的路径。因为应用程序在它和底层的代码之间有好几层，而底层的代码又要经过另外好几层才能到达对象数据库。这样，当应用程序出现问题时，需要很长时间才能把问题隔离出来。

而此时应用程序出问题了。在应用程序开发组的一半人员花了整整一天跟踪 bug 之后，他们发现了那个 bug 是出在底层的访问数据库的代码之中——如果你在名称之中有一个空格，程序会很糟糕地突然死掉。在隔离了与数据库访问相关的底层代码之后，他们把 bug 告诉了代码的所有者，并且提供了一个修正这个 bug 的补丁。代码的所有者感谢了他们，接受了他们的修正补丁，并把修正过的代码提交到代码仓库。

但是第二天，程序又死了。应用程序的开发小组再一次跟踪。这次只花了半天的时间（因为他们已经轻车熟路了），bug 还是出在同一个地方。这次的情况是如果在对象的路径（而不是名字本身）有一个空格，程序也会死掉。显然，在整合补丁的时候，开发者同时引入了一个新的 bug。再一次地，他们跟踪到了它，并提供了一个补丁。但是在第三天，程序又死了！显然，开发者再一次引入了原先的 bug。

应用程序组的经理和我坐下来讨论这个问题，并告诉我他的组在这个星期中花费了几乎两人月在这个问题上（而且很可能影响到了整个公司的其他小组）。然后，我们开发了一些 JUnit 测试，测试的对象是应用程序产品要用到的底层 API 调用，并且特别对对象名和路径中使用空格的两种情况，都添加了访问数据库的测试。我们把产品置于持续构建（用的是 Cruise-Control）的控制之下，因而单元测试会在每次代码提交回代码仓库时自动运行。

不用说都知道，接下来的一周中，测试在原先的开发者手上接连失败了两天。后来，当获得针对测试失败的自动化通知时，经理来到了我的办公室握手致谢。

你看到了，没有 JUnit 测试，不正确的代码在夜间构建时会告知整个公司。但是有了我们的持续构建测试，代码所有者（以及他的经理和测试人员）立刻就看到了问题，而且他能够在公司的其他人使用这些代码之前立刻修正它。事实上，自从那以后这个测试有一半时间是失败的。但是它被捕捉到了，因而就不再是什么大不了的问题了。因为这些测试，那个产品现在相当稳定。

我们现在有了一个制度，每个发生了两次的问题必须在周末之前得到 JUnit 的测试。

在这个故事之中，Tibbert Enterprises 没有怎么使用单元测试来证明一切工作正常，而是用它们来防止问题的再次发生。随着问题一个个被捕捉到，他们最终会把单元测试扩张到覆盖整个产品，而不仅仅是出了问题的地方。

当你面对的是一个没有任何自动化测试的项目时，那么上面的方法或许就是最好的办法。记住，一口吃不成一个胖子；任何事情都需要一步一个脚印地做。

8.5 测试与评审

能成功的团队一般都有代码评审的习惯。这可能是非正式的：资深一些的员工稍微看看代码；或者两个人使用极限编程的结对编程方法一起来编码。也可能是非常正式的：有 check list 和小型的评审团。

无论你是否进行代码评审（我们建议你去做），都要让测试代码成为评审过程的一个组成部分。既然测试代码的标准与产品代码一样高，它们当然也应该被评审。

事实上，有时候扩展一下“测试先行于设计”的思想是有帮助的：在编写产品代码之前，同时编写和评审测试。也就是说，编码和评审以这样的顺序进行：

1. 编写 test case 和/或 测试代码。
2. 评审 test case 和/或 测试代码。
3. 经评审修改 test case 和/或 测试代码。
4. 编写能通过所有测试的产品代码。
5. 评审产品代码和测试代码。
6. 在每次评审后，修改测试代码和产品代码。

对测试代码的评审是非常有用的。因为它不仅仅增加了第一时间发现 bug 的机会，而且通过让所有人都来参与评审，还改善了团队的沟通。团队成员可以借助这个机会，看到其他人是怎么做测试的，看到团队的习惯是怎样的，并且有助于让所有人都保持坦诚。

在评审中，你可以使用本书 145 页的 checklists 帮助鉴别可能的 test case。但是不要过分地去测试不大可能出问题的代码，或者仅仅由于测试的原因，对相似的代码一次又一次地重复测试。

最后，你可能想要跟踪重复出现的常见问题。这可能是需要投注更多关注的地方，或者与这些相关的其他条款应该添加到你的标准的评审 checklist 中去。

例如，在多年前的客户那儿，我发现许多开发者误用了异常处理。在核心代码中遍布了类似下面这样的片断：

```
try {  
    DatabaseConnection dbc = new DatabaseConnection();  
    InsertNewRecord(dbc, record);  
    dbc.Close();  
} catch (Exception) {}
```

也就是说，他们简单地忽略了可能出现的任何异常。这不仅仅导致了随机地忽略了异常，而且系统也可能泄漏数据库连接资源——出现的任何错误都会导致 `Close` 被跳过。

我们把这个问题添加到了已知典型问题的列表当中，等待到评审时进行检查。在评审代码时，这些声名狼藉的 `catch` 语句都会先被发现，然后添加针对出错条件（**RIGHT-BICEP** 中的“E”）的各种单元测试。最后的代码应该是要么向上层传播异常，要么处理异常；而不再像上面那样把异常“吃了”。这个简单过程的结果使系统的稳定性得到了极大的加强。

第 9 章

设计话题

Design Issues

到目前为止，我们已经讨论了如何写单元测试，怎样用单元测试来帮助你理解和验证代码的功能以及操作方面的特性。但其实单元测试还能提供许多改善代码的设计和架构的机会。

在本章中，我们将关注下面这几个设计层面的话题：

- 通过使用面向测试的设计方法，更好地分离关注点。
- 通过定义类不变性更好地阐明设计意图。
- 通过使用测试驱动的设计方法，改善接口的设计。
- 确立和局部化验证的责任。

9.1 面向测试的设计

在软件设计与实现中，“关注点的分离”也许是唯一一个称得上最重要的概念。这个词牵涉很广，包括：封装性、正交性、耦合性和所有其他一切有助于编写“害羞代码（shy code）¹”的计算机科学概念[HT00]。

通过有意地设计出方便测试的代码，你可以让代码具有更好的结构（例如“shy”）和可维护性。例如，假设你在编写一个休眠一个小时的方法。你应先进行一堆计算，然后调用 `sleep()` 函数：

¹ 在作者的文章中，对“shy code”的解释是：code shouldn't reveal too much of itself and shouldn't be too nosy into other affairs。

```

public void SleepUntilNextHour() {
    int howlong;
    XX XXXX X XXXX XX XX XXX;
    // Calculate how long to wait...
    X X XX XXX XXX X X XX;
    XX XXXX X XXXX XX XX XXX;
    Thread.Sleep(howlong);
    return;
}

```

你要如何测试上面这段代码呢？等一个小时吗？还是设置一个计时器、调用这个函数、等这个函数返回、再来检查时间、处理一下函数在该被调用而其实并没有被调用时的情况——这所有一切都会让事情变得更加糟糕。我们在第6章就已经看到了一个类似的例子，而且那个例子也非常好，值得我们在下面再次用它来说明问题。接下来，我们将再次重构该例子中的方法，让它更加容易测试。

下面，我们将不再让计算休眠时间的功能和 Sleep() 函数本身合并在一起，而是把它们分离开来：

```

public void SleepUntilNextHour() {
    int howlong = MilliSecondsToNextHour(DateTime.Now);
    Thread.Sleep(howlong);
    return;
}

```

哪些是需要从函数中分离的呢？是对系统的 Sleep() 函数的调用还是计算要等待多少时间的代码？我们或许并不能确定 C# 的 Thread.Sleep() 函数的行为是否能如声明那般一切正常（但即使并非如此，我们的原则也只是先怀疑自己编写的代码）。因此，就目前而言，你只需要测试要休眠的时间的秒数是否计算得正确，而针对计时器的测试和各种逻辑（这里并没有涉及到一个小时的等待）可以简单地表示如下：

```
Assert.AreEqual(10000, MilliSecondsToNextHour(DATE_1));
```

如果能够确认 milliSecondsToNextHour() 的行为是正确的，就有理由说 sleepUntilNextHour() 是可信的——即使不可信的话，我们也知道问题肯定是和 Sleep() 本身相关，而和数值计算没有关系。另外，你在其他情况下还可以重用 milliSecondsToNextHour() 函数。

这就是前面内容的寓意所在：通过令代码更加容易测试来改善代码的设计。通过改写代码，你克服了前面遇到的一些问题，使得代码可以测试，

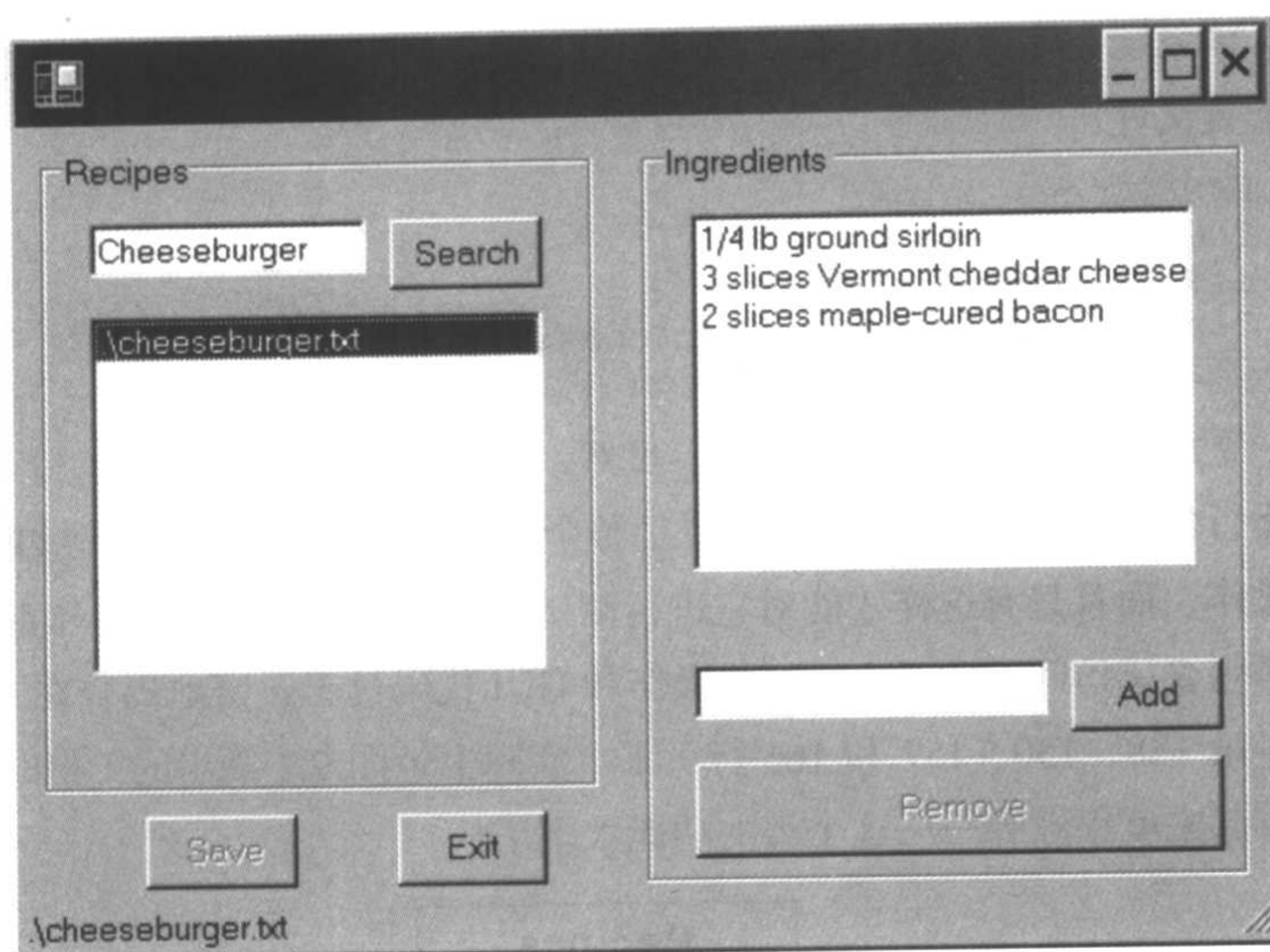


图9.1 Recipes 图形界面截图

并且同时得到一个更加简洁的设计，也让代码更容易扩展和维护，当然还有测试。

我不想用过多的例子和技术细节让你觉得厌烦，实际上你只需要在编写代码的时候，记住下面这个根本性的问题就可以了：

我要如何对代码进行测试呢

如果答案并不那么显而易见，或者测试代码看起来写得非常丑陋，甚至难以编写的话，那么你就应该把这种情况看成一种征兆。它暗示着你的设计可能需要修改，改变一些设计，直到让代码易于测试为止。最后通过这些努力，你的设计一定会好很多。

9.2 为测试而重构

让我们来看一个现实生活中的例子。一个新手设计了一个食谱管理系统，下面是该系统的一些片断，图 9.1 中的 GUI 是很清楚的：只有一个类，并且把 GUI 行为和文件 IO 操作混合在一起。

cheeseburger.txt

该系统会使用逐行的格式读写每份食谱；类似于 INI 文件或者其他形式的配置文件：

```
NAME=Cheeseburger
INGREDIENTS=3
1/4 lb ground sirloin
3 slices Vermont cheddar cheese
2 slices maple-cured bacon
```

下面就是该系统的全部代码。显然，这些代码是很难测试的。因为无论是运行代码的哪个部分，你都需要把整个程序跑起来，并且只能在 GUI 中进行操作，而且只有这样才能对它进行测试。所有的文件 IO 操作和查询服务都必须直接访问 GUI 窗口，这些操作和 GUI 代码过于紧密地耦合在一起（例如，第 138、150、157 和 166 行）了。实际上，图 9.2 给出的 UML 类图，它看起来就很臃肿——这个类的功能太多了！

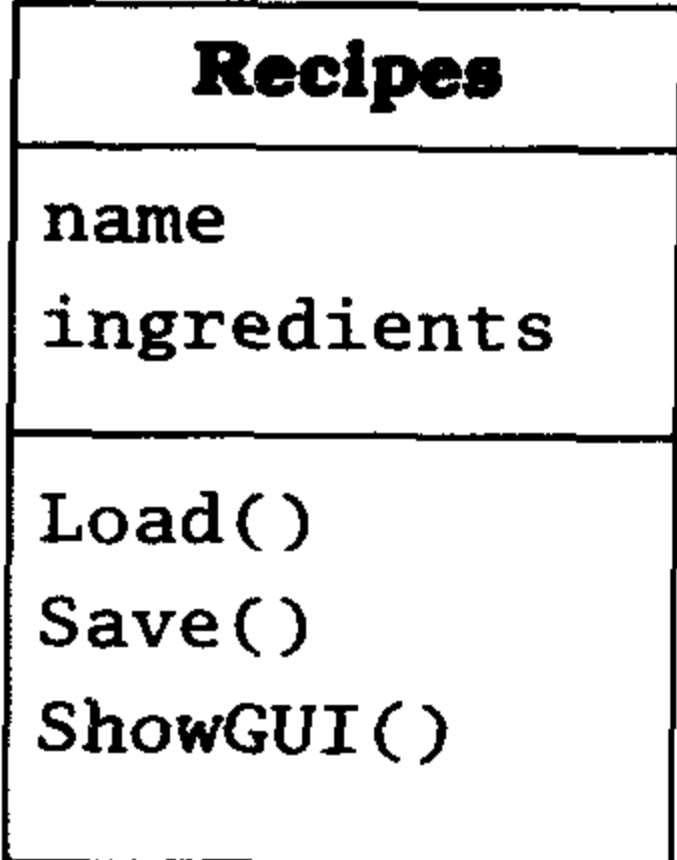


图9.2 原始的Recipes静态类图

```
Line 1  using System;
-       using System.Drawing;
-       using System.Collections;
-       using System.ComponentModel;
5       using System.Windows.Forms;
-       using System.Data;
-       using System.IO;
-
-       public class Recipes : Form {
10      private Button exitButton = new Button();
-       private StatusBar statusBar = new StatusBar();
-       private GroupBox groupBox1 = new GroupBox();
-       private TextBox titleText = new TextBox();
-       private Button searchButton = new Button();
15      private ListBox searchList = new ListBox();
-       private GroupBox groupBox2 = new GroupBox();
-       private ListBox ingredientsList = new ListBox();
-       private Button removeButton = new Button();
-       private TextBox ingredientsText = new TextBox();
20      private Button saveButton = new Button();
-       private Button addButton = new Button();
-
```

```

-   public Recipes() {
-       InitializeComponent();
25  }
-
-   private void InitializeComponent() {
-       exitButton.Location =
-           new System.Drawing.Point(120, 232);
30  exitButton.Size = new System.Drawing.Size(48, 24);
-       exitButton.Text = "Exit";
-       exitButton.Click +=
-           new System.EventHandler(exitButton_Click);
-
35  statusBar.Location = new System.Drawing.Point(0, 261);
-       statusBar.Size = new System.Drawing.Size(400, 16);
-
-       groupBox1.Controls.Add(searchList);
-       groupBox1.Controls.Add(searchButton);
40  groupBox1.Controls.Add(titleText);
-       groupBox1.Location = new System.Drawing.Point(8, 8);
-       groupBox1.Size = new System.Drawing.Size(176, 216);
-       groupBox1.TabStop = false;
-       groupBox1.Text = "Recipes";
45
-       searchList.Location = new System.Drawing.Point(16, 56);
-       searchList.Size = new System.Drawing.Size(144, 147);
-       searchList.SelectedIndexChanged +=
-           new System.EventHandler(
50  searchList_SelectedIndexChanged);
-
-       searchButton.Location = new System.Drawing.Point(112, 24);
-       searchButton.Size = new System.Drawing.Size(48, 24);
-       searchButton.Text = "Search";
55  searchButton.Click +=
-           new System.EventHandler(searchButton_Click);
-
-       titleText.Location = new System.Drawing.Point(16, 24);
-       titleText.Size = new System.Drawing.Size(88, 20);
60
-       groupBox2.Controls.Add(addButton);
-       groupBox2.Controls.Add(ingredientsText);
-       groupBox2.Controls.Add(removeButton);
-       groupBox2.Controls.Add(ingredientsList);
65  groupBox2.Location = new System.Drawing.Point(200, 8);
-       groupBox2.Size = new System.Drawing.Size(192, 248);
-       groupBox2.TabStop = false;
-       groupBox2.Text = "Ingredients";
-
70  addButton.Location = new System.Drawing.Point(136, 176);
-       addButton.Size = new System.Drawing.Size(48, 23);
-       addButton.Text = "Add";
-       addButton.Click +=
-           new System.EventHandler(addButton_Click);
75
-       ingredientsText.Location = new System.Drawing.Point(16, 176);
-       ingredientsText.Size = new System.Drawing.Size(112, 20);
-
-       removeButton.Enabled = false;
80  removeButton.Location = new System.Drawing.Point(16, 208);
-       removeButton.Size = new System.Drawing.Size(168, 32);
-       removeButton.Text = "Remove";
-       removeButton.Click +=
-           new System.EventHandler(removeButton_Click);
85

```

```

- ingredientsList.Location = new System.Drawing.Point(16, 24);
- ingredientsList.Size = new System.Drawing.Size(160, 134);
- ingredientsList.SelectedIndexChanged +=
-     new System.EventHandler(
90     ingredientsList_SelectedIndexChanged);
-
- saveButton.Enabled = false;
- saveButton.Location = new System.Drawing.Point(40, 232);
- saveButton.Size = new System.Drawing.Size(48, 24);
95 saveButton.Text = "Save";
- saveButton.Click +=
-     new System.EventHandler(saveButton_Click);
-
- AutoScaleBaseSize = new System.Drawing.Size(5, 13);
100 ClientSize = new System.Drawing.Size(400, 277);
- Controls.Add(saveButton);
- Controls.Add(groupBox2);
- Controls.Add(groupBox1);
- Controls.Add(statusBar);
105 Controls.Add(exitButton);
- groupBox1.ResumeLayout(false);
- groupBox2.ResumeLayout(false);
- ResumeLayout(false);
- }
110 [STAThread]
- static void Main() {
-     Directory.SetCurrentDirectory(@"..\..\recipes\");
-     Application.Run(new Recipes());
115 }
-
- private void exitButton_Click(object sender,
-                               System.EventArgs e) {
-     Application.Exit();
120 }
-
- private void searchButton_Click(object sender,
-                                 System.EventArgs e) {
-     String toMatch = "*" + titleText.Text + "*";
125
-     try {
-         string[] matchingFiles = Directory.GetFiles(@"..", toMatch);
-         searchList.DataSource = matchingFiles;
-     }
130     catch (Exception error) {
-         statusBar.Text = error.Message;
-     }
- }
-
135 private void
searchList_SelectedIndexChanged(object sender,
                               System.EventArgs e) {
-     string file = (string)searchList.SelectedItem;
-     string line;
140     char[] delim = new char[] { '=' };
-
-     statusBar.Text = file;
-
-     using (StreamReader reader =
145         new StreamReader(file)) {

```

```

-         while ((line = reader.ReadLine()) != null) {
-             string [] parts = line.Split(delim, 2);
-             switch (parts[0]) {
-                 case "NAME":
150                 titleText.Text = parts[1];
-                 break;
-                 case "INGREDIENTS":
-                     try {
-                         int count = Int32.Parse(parts[1]);
155                         ingredientsList.Items.Clear();
-                         for (int i = 0; i < count; i++)
-                             ingredientsList.Items.Add(reader.ReadLine());
-                     }
-                     catch (Exception error) {
160                         statusBar.Text = "Bad ingredient count: " +
-                             error.Message;
-                         return;
-                     }
-                     break;
165                 default:
-                     statusBar.Text = "Invalid recipe line: " + line;
-                     return;
-             }
-         }
170     }
-     saveButton.Enabled = false;
- }

- private void removeButton_Click(object sender,
175     System.EventArgs e) {
-     int index = ingredientsList.SelectedIndex;
-     if (index >= 0) {
-         statusBar.Text = "Removed " +
-             ingredientsList.SelectedItem;
180         ingredientsList.Items.RemoveAt(index);
-         saveButton.Enabled = true;
-     }
- }

- private void addButton_Click(object sender,
185     System.EventArgs e) {
-     string newIngredient = ingredientsText.Text;
-     if (newIngredient.Length > 0) {
-         ingredientsList.Items.Add(newIngredient);
190         saveButton.Enabled = true;
-     }
- }

- private void
195 ingredientsList_SelectedIndexChanged(object sender,
-     System.EventArgs e) {
-     int index = ingredientsList.SelectedIndex;
-     if (index < 0)
-         removeButton.Enabled = false;
-     else {
200         removeButton.Text = "Remove " +
-             ingredientsList.SelectedItem;
-         removeButton.Enabled = true;
-     }
- }

```

Recipes.cs

```

205     }
    ~
    ~     private void saveButton_Click(object sender,
    ~                                     System.EventArgs e) {
    ~         string fileName = titleText.Text + ".txt";
    ~         ICollection items = ingredientsList.Items;
210         using (StreamWriter file =
    ~             new StreamWriter(fileName, false)) {
    ~             file.WriteLine( "NAME={ 0} ", titleText.Text);
    ~             file.WriteLine( "INGREDIENTS={ 0} ", items.Count);
215             foreach (string line in items) {
    ~                 file.WriteLine(line);
    ~             }
    ~         }
    ~         statusBar.Text = "Saved " + fileName;
220     }
    ~ }

```

显然，我们需要改进这些代码。让我们从创建一个表示食谱的独立的类开始吧。有了这个类之后，我们就可以很容易地构造测试食谱的代码，譬如，在屏幕上、文件中、网络上传输食谱、添加食谱等等。该类只是一个简单的数据表示，而且只含有提供成员变量的访问属性。

```

Line 1  using System;
    ~
    ~ using System.Collections;
    ~
    ~ public class Recipe : IEnumerable {
    ~     5     protected String name;
    ~     protected ArrayList ingredients;
    ~
    ~     public Recipe() {
    ~         name = "";
    ~         ingredients = new ArrayList();
    ~     }
    ~
    ~     public Recipe(Recipe another) {
    ~         name = another.name;
    ~         ingredients = new ArrayList(another.ingredients);
    ~     }
    ~
    ~     public String Name {
    ~         get { return name; }
    ~         set { name = value; }
    ~     }
    ~
    ~     public void AddIngredient(String aThing) {
    ~         ingredients.Add(aThing);
    ~     }
    ~
    ~     public IEnumerator GetEnumerator() {
    ~         return ingredients.GetEnumerator();
    ~     }
    ~
    ~     30     public int NumIngredients {
    ~         get { return ingredients.Count; }
    ~     }
    ~ }

```

Recipe.cs

下一步,我们需要把原来 Recipes 类中保存文件到磁盘以及从磁盘加载文件的代码分离出来。

为了区分文件 IO 和其他类型的 IO, 我们可以在一个辅助类中实现文件 IO, 然后在该类中操作 Recipe 对象。我们是希望把所有对 GUI 控件的引用都从原来代码中抽取出来, 用实例的成员变量替换它们。

```

Line 1  public class RecipeFile {
-
-     public Recipe Load(String fileName) {
-         Recipe result = new Recipe();
-         string line;
5         char [] delim = new char[] { '=' };
-
-         using (StreamReader reader = new StreamReader(fileName)) {
-             while ((line = reader.ReadLine()) != null) {
10                string [] parts = line.Split(delim, 2);
-                switch (parts[0]) {
-                    case "TITLE":
-                        result.Name = parts[1];
-                        break;
15                case "INGREDIENTS":
-                    try {
-                        int count = Int32.Parse(parts[1]);
-                        for (int i = 0; i < count; i++)
-                            result.AddIngredient(reader.ReadLine());
20                } catch (Exception error) {
-                    throw new RecipeFormatException(
-                        "Bad ingredient count: " + error.Message);
-                }
-                break;
25            }
-        }
-    }
-    return result;
- }
30
-     public void Save(String fileName, Recipe recipe) {
-         using (StreamWriter file =
-             new StreamWriter(fileName, false)) {
-             file.WriteLine("NAME={ 0} ", recipe.Name);
35             file.WriteLine("INGREDIENTS={ 0} ",
-                 recipe.NumIngredients);
-             foreach (String line in recipe) {
-                 file.WriteLine(line);
-             }
40         }
-     }
- }

```

RecipeFile.cs

现在我们就可以编写一个真正的单元测试: 测试对硬盘的文件读写操作, 而且并不需要使用到任何 GUI 代码。

```

Line 1  using System;
-       using System.Collections;
-       using System.IO;
-       using NUnit.Framework;
5
-       [TestFixture]
-       public class TestRecipe {
-
-           [Test]
10          public void SaveandRestore() {
-
-              const String test_name =
-                  "Cheeseburger";
-              const String test_ing1 =
15                  "1/4 lb ground sirloin";
-              const String test_ing2 =
-                  "3 slices Vermont cheddar cheese";
-              const String test_ing3 =
-                  "2 slices maple-cured bacon";
20
-              // Save one out
-              Recipe rec = new Recipe();
-              rec.Name = test_name;
-              rec.AddIngredient(test_ing1);
25              rec.AddIngredient(test_ing2);
-              rec.AddIngredient(test_ing3);
-
-              RecipeFile filer = new RecipeFile();
-              filer.Save( "test.recipe", rec);
30
-              try {
-                  // Now get it back
-                  Recipe rec2 = new Recipe();
-                  filer = new RecipeFile();
35                  rec2 = filer.Load( "test.recipe");
-
-                  Assert.AreEqual(test_name, rec2.Name);
-                  IEnumerator itr = rec2.GetEnumerator();
-
40                  Assert.AreEqual(test_ing1, itr.Current);
-                  itr.MoveNext();
-                  Assert.AreEqual(test_ing2, itr.Current);
-                  itr.MoveNext();
-                  Assert.AreEqual(test_ing3, itr.Current);
45                  itr.MoveNext();
-                  Assert.IsFalse(itr.MoveNext());
-              } finally {
-                  File.Delete( "test.recipe");
-              }
50          }
-      }

```

TestRecipe.cs

在第 13 行中，我们声明了一些用于测试的常量字符串。接下来新建了一个空的 Recipe 对象，并且在第 23 行中用这个字符串常量给它的成员赋值。显然，我们可以直接用文字字符串（literal string）给成员赋值，但是由于在后面需要用这个字符串来检查测试结果，所以把这个字符串声明为一个常量，从而能在两处对它进行引用。

给 Recipe 对象的成员全部赋值之后，我们在第 29 行调用 Save() 方法，把这份食谱存储到文件中。然后，我们创建了一个全新的 Recipe 对象，并且在第 35 行调用辅助类的 Load 方法把文件加载到新对象中。

加载完对象之后，我们现在就可以开始运行一系列完整的 assert 语句，检验在 rec 中加入的测试数据是否和 rec2 加载的数据完全一致。

最后，第 48 行我们使用了一个 File 类，删除了测试所使用的临时文件。在此，我们使用了一个 finally 分支，从而保证无论断言是否成功，该临时文件都能够被删除。

现在，我们就可以用一般的方法运行单元测试，验证读写文件的代码是否正确。

在继续阅读之前请先运行上面的例子……



Failures:

```
1) TestRecipe.SaveandRestore :
   String lengths differ. Expected length=12, but was length=0.
   Strings differ at index 0.

   expected:<"Cheeseburger">
   but was:<" ">
   -----^
   at TestRecipe.SaveandRestore() in testrecipe.cs:line 37
```

Whoops! 看起来代码的运行结果和我们的期望并不一致——我们在加载的食谱对象中并没有获得对应的名称。当我们在 *RecipeFile.cs* 的代码中保存文件的时候，使用的是“NAME”作为字段名；而当我们读取该文件的时候 (load() 函数的 12 行)，却使用字段名“TITLE”来进行读取；这当然不能通过测试。我们可以把读取的域名改为“NAME”，从而和文件保存时使用的域名相匹配；但别急，稍等一会儿，你应该问问自己下面这个重要的问题：

其他的代码还会存在同样的问题吗

使用常量字符串来作为关键字是一个很好的想法，但是同时也会由于拼写错误或者名称不匹配等，给错误的引入打开了大门，这也正如我们前面的例子所示的那样。因此，也许上面这个失败的测试要告诉我们一些更多的信息——或许你应该对代码进行重构，把这些文字字符串都表示成一个字符串

常量。重构之后的类大概是这样：

```

Line 1  public class RecipeFile {
-
-      const String NAME_TOK = "NAME";
-      const String INGREDIENTS_TOK = "INGREDIENTS";
5
-      public Recipe Load(String fileName) {
-          Recipe result = new Recipe();
-          string line;
-          char [] delim = new char[] { '=' };
10
-          using (StreamReader reader = new StreamReader(fileName)) {
-              while ((line = reader.ReadLine()) != null) {
-                  string [] parts = line.Split(delim, 2);
-                  switch (parts[0]) {
15
-                      case NAME_TOK:
-                          result.Name = parts[1];
-                          break;
-                      case INGREDIENTS_TOK:
-                          try {
20
-                              int count = Int32.Parse(parts[1]);
-                              for (int i = 0; i < count; i++)
-                                  result.AddIngredient(reader.ReadLine());
-                              } catch (Exception error) {
-                                  throw new RecipeFormatException(
25
-                                      "Bad ingredient count: " + error.Message);
-                              }
-                          break;
-                      }
-                  }
30
-              }
-          return result;
-      }
-
-      public void Save(String fileName, Recipe recipe) {
35
-          using (StreamWriter file =
-              new StreamWriter(fileName, false)) {
-              file.WriteLine( "{ 0} = { 1} ",
-                  NAME_TOK, recipe.Name);
-              file.WriteLine( "{ 0} = { 1} ",
40
-                  INGREDIENTS_TOK, recipe.NumIngredients);
-              foreach (String line in recipe) {
-                  file.WriteLine(line);
-              }
-          }
45
-      }
-  }

```

RecipeFile.cs

借助于这些简单的修修改改，我们已经对原来的程序作了很大的改进。
为了测试文件 IO，我们：

- 创建了一个单独的 `Recipe` 类。
- 把文件 IO 读写操作代码从 GUI 中分离出来，放入类 `RecipeFile` 中。
- 为了避免拼写错误等 bug，把文本字符串换成字符串常量。

现在，我们已经为 `Recipe` 提供了测试基本功能的单元测试；最后，我们需要把新的 `Recipe` 类再集成到 GUI 中，并且使用新的文件 IO 类 `RecipeFile`，结果类似于图 9.3。

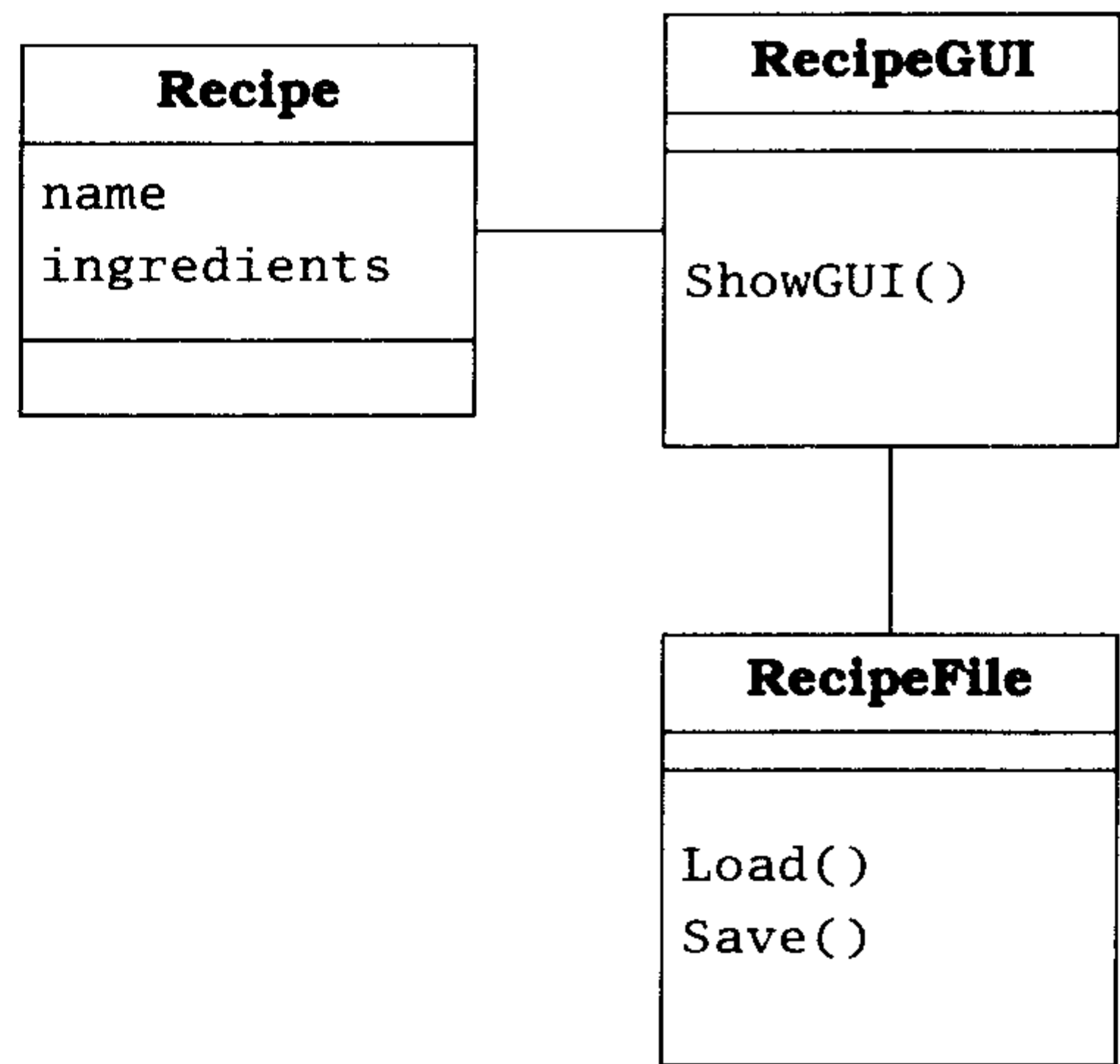


图9.3 重构后的Recipes静态类图

现在，`RecipeGUI` 包含了一个类型为 `Recipe` 的对象，并且使用辅助类 `RecipeFile` 来读取和写入该对象。当用户点击 **save** 按钮的时候，GUI 将会根据窗口中的值给 `Recipe` 对象赋值，并且调用 `RecipeFile.Save()` 来保存该对象的值。当加载一份新食谱的时候，GUI 将会使用 `RecipeFile.Load()` 的返回值来加载 `Recipe` 对象，并从该对象获得相应的值。

对 GUI 的测试是很困难的，而且如果花费很大精力在 GUI 测试上面，往往也得不偿失。因此，通过把纯粹的 GUI 代码和应用程序的核心功能分开，你就可以在不涉及 GUI 的情况下，很容易地增加和测试业务逻辑。

主要的 GUI 类 `RecipeGUI`（在最开始的代码中被看成是 `Recipes` 类）应该只包含面向 GUI 的代码：`widgets`、`callbacks`（回调）等等。因此，所有的业务逻辑和文件 IO 操作都应该是非 GUI 的，而且是完全可测试的类。

在这个例子中，我们做测试的额外回报是获得了一个简洁的设计。



9.3 测试类的不变性

另一个改善类设计的方法是：定义和验证“类的不变性”。

类的不变性是指一个或者几个针对类对象的断言。如果一个类对象有效的话，这些断言都必须为真。这一点是不可改变的。²

例如，对一个实现了 `sorted list` 的类，它的不变性就是：它的内容必须是有序的。这就意味着：不管外面发生了任何事情，或者调用了该类的任何方法，这个 `list` 都必须是有序的——至少从对象的外面看来是有序的。当然，在一个方法里面，当类需要对自身进行调整的时候，不变性可能会短暂地被破坏。但是当方法返回的时候，或者说对象可以被外界使用的时候（如在一个多线程环境里），该不变性必须永远为真，否则就说明代码中存在一个 `bug`。

这就意味着：在该类的每一个单元测试中，你都可以测试它的不变性。

通常而言，类的不变性是内部实现的产物：如内部计数器、某些成员变量是 `populated` 的事实等等；另外，对于验证用户输入等类似操作而言，并不是不变性的职责所在。当编写单元测试的时候，我们一次只希望测试一种情况。但是在每次测试的时候，你都需要确信类的整个状态都是一致的——你需要确信没有造成任何间接的破坏。

² 针对前条件，后条件和不变性的更多信息，可以参考[Mey97]。

下面就是一些适合应用类的不变性的领域。

■ 结构化

最普遍的不变性是结构化属性。也就是说，这里的不变性指的是数据的结构化属性。例如，在一个以订单为中心的系统中，你可以有下列不变性：

- 每个条目行必须属于一份订单。
- 每份订单必须要包含一个或多个条目。

当用数组来存储数据的时候，你通常需要使用一个成员变量来作为数组的索引。下面就是针对该索引的不变性：

- 索引必须大于或者等于 0。
- 索引必须小于数组的长度。

你需要检查这些不变性，来验证这些条件是否不成立。假设你对数组的索引执行某种计算；你可能期望在整个单元测试中都检查上面的不变性，从而确保类永远也不会处于不一致的状态。例如，在针对 `stack` 类的例子中，我们在第 60 页给出了这种验证。

结构化错误通常都会导致程序抛出一个异常，或者突然中止。如果发生这种问题，那就意味着不变性检查失败了；有了不变性，我们获得的好处是：当不变性被违反的时候，你或许能够立刻——在出现异常情况的时候——知道具体原因所在，甚至还可能知道哪些条件被违反了。如果没有不变性，那么错误的出现位置可能会和 `bug` 所在相去甚远；而返回去跟踪的原因通常也需要花费许多时间，可能几分钟，但也可能几天。

更加重要的是，不变性检查能够让你确信另一个事实：你并不是基于某种巧合，才令测试通过的。这是因为，对于一个实际存在的 `bug`，即使显式的测试并没有捕捉到它，不变性或许也可以帮助你提前捕捉到这个 `bug`。

■ 数学不变性

其他的一些约束通常都与数学相关。在此，并不是为了验证数据结构的物理特性，而是需要考虑它的逻辑模型。例如：

- 银行账号的借贷必须平衡。
- 不同单位计算出来的总数必须一致（一个尤其流行的例子就是太空船）。

这听起来已经类似于我们前面所讨论的边界条件了，而且在某种条件下也确实如此。其中的不同之处是：在类的所有可视状态中，不变性都必须为真。也就是说，不变性并非一个短暂的条件，它永远都为真的。

■ 数据一致性

在很多时候，一个对象可以以多种不同的方式来表示同一份数据——购物车的商品序列、商品的销售总数和购物车中的商品总数等，这些都是互相关联的。根据一个详细的商品条目表，你就可以做出其他的两个图。于是，这里就存在一个不变性：这些图都必须是一致的。否则的话，说明代码存在一个 bug。

9.4 测试驱动的设计

如果你总是在编写实现代码之前，就先编写它们的测试代码[Bec00]，那么你就是在使用测试驱动的开发这么一种很有用的技术。这种编码方式的一个优点是，你可以享受到“测试驱动的设计”的好处，并且大大改善接口的设计。

因为总是自己亲自来调用这些接口，所以通常都能获得更好的接口设计。正如一句谚语所说——你总是能够利用这些反馈来改善接口的设计。

也就是说，通过先编写测试，你现在需要把自己当成代码的用户，而不是代码的实现者。从这个角度来看，你能够在接口如何被使用方面获得更多的感知，从而有更多的机会来改善设计。

例如,假设你在编写一个小函数,用于格式化某些特殊的打印页。在此,需要指定一些打印尺寸,因此你首个版本的代码也许是这样的:

```
AddCropMarks(PSStream str, double paper_width,
               double paper_height,
               double body_width,
               double body_height);
```

接下来你开始编写测试(基于现实的数据)。但是此时从测试代码中你发现了一种模式:

```
public Process() {
    XXX XX XXXXXX XXX 4X X XX XXX XXX XXXXX XX XX;
    X XX X XXX XXXX XX XXX XX XXXXXX XXXX;
    AddCropMarks(str, 8.5, 11.0, 6.0, 8.5);
    XX XXX X XXX XXX XX X XXX XXX XXX XX XX XX;
    X XXX XXX XXXX X XXX XXX XXX XXXX XX XXX XX;
    AddCropMarks(str, 8.5, 11.0, 6.0, 8.5);
    XX XX XXXXX XX XX XXX XXX XXX XXXX XX XX XX XX;
    X XX XX X XXXX XXX X XXXX XX XX XX XXX XXX XX;
    AddCropMarks(str, 8.5, 11.0, 6.0, 8.5);
    XXX XX XXXXXXXX XXX XXX XXXXX X XXX XXXX XX XXXXXX;
    XX X XXX XXXX XXXX XXX XXXX XXXX XX X X XX XX;
    AddCropMarks(str, 5.0, 7.0, 4.0, 5.5);
    XX XXX XXX XX X XXX XXX XXX XXXX XX XX XX XXX XX;
    XXX XX XXXXX XX XX XXX X XXX XXXX XX XX XX XXX;
    AddCropMarks(str, 5.0, 7.0, 4.0, 5.5);
    XX XX XXXXX XX X XX XXX XXX XXXX XX XX;
    X XXX X XXX XXXX XX XX XXX XXXX XX;
}
```

如上所示,实际使用的纸的大小通常只有固定的几种,但你仍然可以使用任意大小的纸。因此,要做的首要事情——让测试更加容易——当然是把纸的大小规范分离到一个单独的类中。

```
PaperSpec standardPaper1 = new PaperSpec(8.5, 11.0,
                                           6.0, 8.5);
PaperSpec standardPaper2 = new PaperSpec(5.0, 7.0,
                                           4.0, 5.5);

XXX XX XXXXXXXX XXX XXX XXXXX X XXX XXXX XX XXXXXX;
XX X XXX XXXX XXXX XXX XXXX XXXX XX X X XX XX;
AddCropMarks(str, standardPaper1);
AddCropMarks(str, standardPaper1);
XX XXX XXX XX X XXX XXX XXX XXXX XX XX XX XXX XX;
XXX XX XXXXX XXX XX XXX X XXX XXXX XX XX XX XXX;
AddCropMarks(str, standardPaper2);
```

现在,测试代码变得更加简洁和简单。相应地,应用程序代码也变得更加简洁。

另一方面,既然标准的纸张大小是固定的,那么可以创建一个工厂类来封装所有标准大小纸张的创建。

```

public class StandardPaperFactory {
    public static PaperSpec LetterInstance;
    public static PaperSpec A4Instance;
    public static PaperSpec LegalInstance;
    XXXXXX XXXXXX XXXXXXXXXXXX XXXXXXXXXXXXXX;
    XXXXXX XXXXXX XXXXXXXXXXXX XXXXXXXXXXXXXX;
}

```

通过追求让测试代码的编写变得更加简洁和容易的效果，你同时也令产品代码的编写变得更加简洁和容易。

■ Try it

练习

答案见 153 页

7. 设计一个利息计数器，它将基于两个工作日之间的工作日天数来计算利息。它使用测试先行的设计，并且按步骤地进行设计，一次只设计一个方面。

9.5 测试无效的参数

当第一次执行测试的时候，有些人会问一个这样的问题：“我需要测试我的类是否对它的参数进行了验证？”。如果用最有咨询师风范的口吻来回答这个问题，那么答案是“这并不一定……”。

你的类应该验证它的参数吗？如果是的话，那么上面的答案就为 yes，你将需要测试这个功能是否为正确的。但是这里却引出了一个更大的问题：谁负责验证输入的数据呢？

对于大多数系统，这个答案是模棱两可的，最多也是偶然的。你并不能信任系统的任何部分已经检查了输入数据，因此你只能亲自检查这些数据——或者至少针对你特别关注的情况来，检查数据的某些方面。事实上，对于这些数据，最后的情况可能会是要么所有人都对它们做了检查，要么是根本就没有人检查它们。另外，重复检查这些数据也违反了 DRY 原则[HT00]，同时还浪费了大量的时间和精力——而我们通常并没有多少额外的时间和精力可以浪费在这上面。

在一个设计良好的系统中，你首先就应该建立那些负责验证工作的系统部件，并且把这些部件都局部化到一个小的、外界知道的系统部分中。

因此，对于一个系统，你需要提问的第一个问题就是：谁负责检查输入数据的有效性？

通常，我们发现最简单的办法就是“别让野蛮人进门”。在系统入口的位置就检查输入，因此在系统的内部也就不需要再次检查。也就是说，如果数据能够通过入口的检查，并且深入到系统的内部，那么系统内部的组件也就可以信任这些输入数据了，因为它们并非“野蛮人”。

这就像医院的手术室或者工厂的无尘室。在你进入它们之前，要先对你——或者任何工具和材料——都进行相当严格的杀毒或清理环节；但是一旦你进入之后，那么就应确信整个环境已经处于无菌的状态。如果此时该环境仍然带有污染，那么这将是一个很大的灾难。你将不得不重新对整个环境进行重新消毒。

在一个软件系统中，任何面向外部的部分（一个 UI，或者另一个系统的接口）都需要是健壮的，而且不能允许让任何不正确的、或者无效的数据顺利通过。而“正确性”或者“有效性”的定义应该是文档的一部分，你需要对它们进行测试。

那么这些和单元测试又有什么关系呢？

它能告诉你哪些是需要进行测试的。如前面所说过的，对于输入的数据，如果你的代码并不负责对它们进行检查，那么就不必在检查它们上面浪费时间。而如果是你的责任，那么就需要加倍地警惕——因为现在系统中的其余部分完全依赖于你的检查结果，并且只依赖于你的检查结果。

但那也别紧张，你还有单元测试呢。

附录 A

Gotchas¹

这里介绍的是一些比较普遍的 Gotchas，也就是一些麻烦、问题或者误解。在我们日常的开发中，这些问题总是一次又一次地出现，并且会迷惑我们。

A.1 只要代码能工作就可以

某些人可能会认为：只要代码能够运行就可以了，即使单元测试不能通过也无所谓。实际上，没有单元测试的代码——或者说没有通过单元测试的代码——是不完整的代码。对于这种代码，你只是还不知道在何处，或者何时，会出问题而已。在这种情况下，你得到的实际上是两方面的最糟糕的结果：一方面，你所有编写单元测试的努力都白费了；另一方面，你仍然不能确保你的代码能够完成你所期望的事情。

如果测试不能通过的话，则应该以对待错误代码的态度来对待这些测试。

A.2 “冒烟”测试

某些开发者认为：相较单元测试而言，“冒烟”测试就已经足够了。也就是说，如果一个方法能够从头到尾完整运行，那么该方法也就通过了。

很容易看到下面这种测试：在整个测试中，除了在最后出现的一个 `Assert.IsTrue(true)` 的断言，测试自身没有任何其他断言。更加复杂一些的或许会有多个 `Assert.IsTrue(true)`，但也就仅此而已。这些测试代码的目的仅限于：“代码能够运行到这里吗？”

¹ 编者注：Gotcha 是 I've got you 的省略语，为“难倒你了”、“问倒你了”的意思。

而这是远远不够的。如果没有验证任何数据或者行为，那么你给自己营造的只不过是一种虚无的安全感——你或许会认为代码已经经过测试了，但实际上并没有。

请格外注意如上面的测试。如果发现了这种测试，请马上更正它们。真正的测试是要检查结果的。没有检查结果的测试只是在浪费时间而已。

A.3 “请让我的机器来运行”

在某些项目中，出现的另一个病态的问题是这么一个老掉牙的借口：“它并没有失败啊，它在我的机器上是能够通过的啊”。这就说明了其中存在着和环境相关的 bug。当出现了这种情形时，问问自己这些问题：

- 是否所有的一切都受到了版本控制工具的管理？
- 本机的开发环境和那台机器的开发环境是否一样呢？
- 这里是否存在一个真正的 bug，它只不过是在其他的机器里偶然现身了（可能其他的机器速度更快、内存更多或者更少等等）。

特别是最终的用户，尤其不喜欢听到这样的借口——这些代码只能在你的机器运行，而在他们的机器则不能运行。

所有的测试必须都能够在所有的机器通过，否则的话，代码就可能存在着问题。

A.4 浮点数问题

很多开发者可能会怀念当年在教室中讨论浮点数的情节。在现实的计算机中，因为能够使用的位数是有限的，硬件只能近似地表示浮点数，因此我们必须多给一些东西。

这就意味着 $1.333 + 1.333$ 并不完全等于 2.666 。它们会非常接近，但不会完全相等。这也是 NUnit 的浮点数断言要求你指定一个精度，用来近似地比较两个值（见第 29 页的讨论）的原因。

但是你同时应该知道：“足够接近”有时候也是带有欺骗性的。例如，就精度而言，你的测试相比真实的要求来说可能还不够。或者你会对下面的错误消息感到疑惑：

```
Failures:
1) TestXyz.TestMe :
    expected:<1.000000000>
    but was:<1.000000000>
at TestXyz.TestMe() in TestXyz.cs:line 10
```

“哎呀，它们看起来确实是相等的呀！”，但实际上它们并不相等——这两个数之间必定有点区别，只是这种区别不能被 `print` 方法显示出来而已。

另一方面，当你使用日期或者时间类型的时候，可能也会遇到类似的问题。两个日期看起来可能是相等的——但是实际上它们可能在毫秒级别上是不等的。

A.5 测试耗费的时间太多了

单元测试需要能够运行得相当快，毕竟，你需要经常地运行它们。但有时候你可能会突然发现运行测试代码的时间太长了，已经妨碍了你日常编写测试和代码的速度。

这意味着：在这种时候，你需要重新检查你的测试了，把那些运行时间比平均运行时间长的测试单独分离出来，并且把这些单独测试重新组合起来。

之后，你可以有选择地运行这一组测试，例如一天一次 `build` 的时候，或者在每次签入代码的时候，而不是每次修改代码都需要运行这一组测试。

另一方面，同时别让这些测试跑得太远，甚至很少有得到运行的机会。

A.6 测试总是失败

某些组可能会发现测试总是一次又一次地失败。一些小的代码改动立刻就会令很多测试失败，而且需要花费很多精力才能让所有的测试再次通过。

这实际上是一种过度耦合的迹象。如果测试代码和外部的数据或者系统的其他部分等发生过度耦合，通常就会出现这种情况。

只要你认为这是一个问题，你就需要马上修复它。先把系统中必要的部分分离出来，从而让测试更加强壮。具体的做法有：借助于产品代码中最小化耦合度的相同技术；见[HT00]中对于正交性和耦合度的详细描述，或者[FBB+99]中有关重构和设计臭味的详细信息，并且请记住使用 Mock Objects（第6章）帮助把你从现实世界中解耦出来。

A.7 在某些机器上测试失败

这里有一个梦魇般的场景：所有的测试在大多数机器上都能够通过。但是对于某些机器，却全部失败了，或者只是间断地失败。

这究竟是因为什么呢？这些机器之间究竟有什么区别呢？

比较明显的答案可能是下面这些资源的差异：操作系统版本号、运行库、C#运行引擎、数据库驱动等等。不同版本的软件可能会有不同的 bug，解决方法甚至特性，因此对于配置不同的机器，很有可能会出现不同的行为。

但是如果机器的配置都完全相同呢，为什么还是得到不同的结果？

其他的原因还可能包括：一台机器的运行速度稍微比别的机器快一些，时间片分配的差异导致资源竞争问题，或者并发性问题，单处理器或多处理器的差异等等。

实际上，这是一个 bug，只是碰巧在以前没有出现而已。在测试失败的机器，应该使用通常的调试方法，跟踪到 bug 所在的位置。尽量找到这个 bug 针对这台机器是在什么位置出现的，然后修复这个 bug，之后确认所有的测试在所有的机器上都能够通过。

附录 B

资源

Resources

B.1 网络资源

Cruise Control .NET

<http://ccnet.thoughtworks.com>

CruiseControl.NET 是用于微软.NET 平台的自动化的持续集成服务器。它集成了 Nant, NUnit 和大部分主要的开源的和专有的版本控制系统。

DotNetMock

<http://sourceforge.net/projects/dotnetmock>

包含.NET 环境下 Mock Object 信息的资料库,以及关于测试的一般性资料。

NCover

<http://workspaces.gotdotnet.com/ncover>

简单的代码覆盖工具。它从命令行开启运行并输出包含代码覆盖率统计信息的 XML 文件。对于 monitored assemblies, 需要 pdb 文件, 并产生逐行的访问计数。还包含一个简单的 XSLT 转换来使得能够在浏览器上阅读输出。

NMock

<http://nmock.truemesh.com>

NMock 是用于.NET 的动态 mock-object 库。

NUnit

<http://nunit.org>

这个用于微软.NET 环境的基于 xUnit 的单元测试框架完全是用 C#编写

的，而且已经被完全的重新设计过了，采用了许多.NET 独有的语言特性，包括属性和其他反射相关的功能。NUnit 让.NET 的所有语言都能使用 xUnit。

NUnit-Addin

<http://www.mutantdesign.co.uk/nunit-addin>

针对 NUnit 的 Visual-Studio 集成。

Pragmatic Programming

<http://www.pragmaticprogrammer.com>

Pragmatic Programming 和原作者的主页。这儿你能找到本书中的所有源代码例子、附加的资料、更新过的 URL、勘误表、本系列的新卷出版的消息以及其他资源。

xUnit

<http://www.xprogramming.com/software.htm>

许多种语言和环境的单元测试框架。

B.2 参考书目

- [Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 2000.
- [Cla04] Mike Clark. *Pragmatic Automation*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, (planned for) 2004.
- [FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Reading, MA, 1999.
- [HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.
- [Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1997.
- [MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In Giancarlo Succi and Michele Marchesi, editors, *Extreme Programming Examined*, chapter 17, pages 287.302. Addison Wesley Longman, Reading, MA, 2001.
- [TH03] Dave Thomas and Andy Hunt. *Pragmatic Version Control*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.

Pragmatic Unit Testing: Summary

General Principles:

- ☐ Test anything that might break
- ☐ Test everything that does break
- ☐ New code is guilty until proven innocent
- ☐ Write at least as much test code as production code
- ☐ Run local tests with each compile
- ☐ Run all tests before check-in to repository

Questions to Ask:

- ☐ If the code ran correctly, how would I know?
- ☐ How am I going to test this?
- ☐ What *else* can go wrong?
- ☐ Could this same kind of problem happen anywhere else?

What to Test: Use Your RIGHT-BICEP

- ☐ Are the results **right**?
- ☐ Are all the **boundary** conditions CORRECT?
- ☐ Can you check **inverse** relationships?
- ☐ Can you **cross-check** results using other means?
- ☐ Can you force **error conditions** to happen?
- ☐ Are **performance** characteristics within bounds?

Good tests are A TRIP

- ☐ **A**utomatic
- ☐ **T**horough
- ☐ **R**epeatable
- ☐ **I**ndependent
- ☐ **P**rofessional

CORRECT Boundary Conditions

- ☐ **C**onformance — Does the value conform to an expected format?
- ☐ **O**rdering — Is the set of values ordered or unordered as appropriate?
- ☐ **R**ange — Is the value within reasonable minimum and maximum values?
- ☐ **R**eference — Does the code reference anything external that isn't under direct control of the code itself?
- ☐ **E**xistence — Does the value exist? (e.g., is non-null, non-zero, present in a set, etc.)
- ☐ **C**ardinality — Are there exactly enough values?
- ☐ **T**ime (absolute and relative) — Is everything happening in order? At the right time? In time?

<http://www.pragmaticprogrammer.com/sk/ut>

附录 C：注重实效的单元测试：总结

一般原则：

测试任何可能失败的地方。
测试任何已经失败的地方。
对于新加的代码，在被证明正确之前，
都可能是有问题的。
至少编写和产品代码一样多的测试代码。
针对每次编译都做局部测试。
签入代码之前做全局测试。

要回答的问题：

我如何知道代码运行是否正确呢？
我要如何对它进行测试？
还有哪些方面可能会发生错误？
这个问题是否会在其他的地方出现呢？

测试哪些方面：使用你的RIGHT-BICEP

结果是否正确（Right）？
边界（boundary）条件是否正确？
是否可以检查反向（inverse）关联？
是否可以使用其他方法来跨检查（cross-check）结果？
错误条件(error condition)是否可以重现？
性能方面是否满足条件？

好的测试是一个TRIP

Automatic（自动的）。
Thorough（全面的）。
Repeatable（可重复的）。
Independent（独立的）。
Professional（专业的）。

CORRECT 边界条件

一致性（Conformance）——值是否符合预期的格式？
有序性（Ordering）——一组值是该有序的，还是该无序的？
区间性（Range）——值是否在一个合理的最大值和最小值的范围之内？
引用，耦合性（Reference）——代码是否引用了一些不受代码本身直接控制的外部因素？
存在性（Existence）——值是否存在（例如，非 null，非零，包含于某个集合等）？
基数性（Cardinality）——是否恰好有足够的值？
时间性，绝对的或者相对的（Time）——所有事情是否都是按顺序发生的？是否在正确的时间？是否及时？

附录 D

习题答案

Answers to Exercises

■ 习题 1：位于 68 页

一个简单的栈类。根据通常意义的栈的语义，把字符串对象压入到栈内，以及将其弹出。这个类提供了以下函数：

```
using System;

public interface StackExercise {

    /// <summary>
    /// Return and remove the most recent item from
    /// the top of the stack.
    /// </summary>
    /// <exception cref="StackEmptyException">
    /// Throws exception if the stack is empty.
    /// </exception>
    String Pop();

    /// <summary>
    /// Add an item to the top of the stack.
    /// </summary>
    /// <param name="item">A String to push
    /// on the stack</param>
    void Push(String item);

    /// <summary>
    /// Return but do not remove the most recent
    /// item from the top of the stack.
    /// </summary>
    /// <exception cref="StackEmptyException">
    /// Throws exception if the stack is empty.
    /// </exception>
    String Top();

    /// <summary>
    /// Returns true if the stack is empty.
    /// </summary>
    bool IsEmpty();
}
```

StackExercise.cs

这儿有一些让你入手的提示：什么是最可能出问题的？当栈首度初始化时的行为应该是怎么样的？在它被用过一段时间后呢？确实如它应该的那样吗？

答案 1:

- 对于全新的栈，IsEmpty() 应当为真，Top() 和 Pop() 应当抛出异常。
- 从空的栈开始，调用 Push() 来连续压入一个测试字符串到栈中。验证 Top() 返回那个字符串，而 IsEmpty() 返回 false。
- 调用 Pop() 来移除那个测试字符串，并且验证那是同一个字符串¹。现在 IsEmpty() 应当为真。再次调用 Pop() 来验证一个异常被抛出了。
- 现在，再做一次相同的测试，但是这次添加多个条目到栈中。确信你取回的条目和顺序都是正确的（最新进的条目应当是被返回的那个）。
- 压入一个 null 到栈中然后弹出它；确认你取回的是 null。
- 确信在栈抛出了异常之后，你还能使用它。

■ 习题 2：位于 69 页

一个购物车。这个类让你添加，删除，以及计算购物车中物品的数量。会出现的边界条件是什么？对于你能删除的商品，有没有什么暗含的约束？如果车是空的，有没有什么有意思的问题？

```
public interface ShoppingCart {
    /// <summary>
    /// Add this many of this item to the
    /// shopping cart.
    /// </summary>
    /// <exception cref="NegativeCountException">
    /// </exception>
    void AddItems(Item anItem, int quantity);

    /// <summary>
    /// Delete this many of this item from the
    /// shopping cart
    /// </summary>
    /// <exception cref="NegativeCountException">
    /// </exception>
    /// <exception cref="NoSuchItemException">
```

¹ 在这种情况下，使用 Assert.AreEqual() 并不好，你应该使用 Assert.AreSame() 来确认它们是相同的对象。

```

    /// </exception>
    void DeleteItems(Item anItem, int quantity);

    /// <summary>
    /// Count of all items in the cart
    /// (that is, all items x qty each)
    /// </summary>
    int ItemCount { get; }

    /// Return iterator of all items
    IEnumerable GetEnumerator();
}

```

答案 2:

- 用数量 0 来调用 AddItems, 而 ItemCount 应当没有变化。
- 用数量 0 来调用 DeleteItem, 而 ItemCount 应当没有变化。
- 用负数来调用 AddItems, 而这应当引发异常。
- 用负数来调用 DeleteItem, 而这应当引发异常。
- 调用 AddItems 而条目的数量应当增加, 无论那个条目是否存在。
- 调用 DeleteItem 删除并不存在的条目, 而这应当引发异常。
- 当车内没有东西的时候调用 DeleteItem, 而 ItemCount 应当仍为 0。
- 调用 DeleteItem 删除的数量大于车内已经有的物品的数量, 而这应当引发异常。
- 在车内空无一物的时候调用 GetEnumerator, 而这应当返回一个空的 iterator (也就是说, 它是一个真正的 IEnumerable 对象 (非空) 但是没有包含任何条目)。
- 对几个条目反复调用 AddItem, 并验证车内的物品增加了 (如 GetEnumerator() 和 ItemCount() 报告的那样)。

提示: 你可以把这里的好几个 assert 合并为一个测试。例如, 你可以从空的车子入手, 加入 3 个条目, 然后一次删除一个。

■ 习题 3：位于 70 页

一个传真调度程序。这段代码将从一个指定的文件名发传真到一个美国的电话号码。其中有校验的需求；带区号的美国电话必须是这样的格式 *xnn-nnn-nnnn*，其中 *x* 必须是在 2 到 9 之间，而 *n* 可以是 0 到 9 的任何数字。以下的区域是预留的，现在还不是有效的区号：*x11*、*x9n*、*37n*、*96n*。

函数的原型是：

```
///
/// Send the named file as a fax to the
/// given phone number.
/// <exception cref="MissingOrBadFileException">
/// </exception>
/// <exception cref="PhoneFormatException">
/// </exception>
/// <exception cref="PhoneAreaCodeException">
/// </exception>

public bool SendFax(String phone, String filename)
```

给定了这些需求，你可以想出哪些边界条件？

答案 3：

- 区号从 111、211 到 911、290、291 以及从 999、370 到 379 或者 960 到 969 的电话号码，这些都应当抛出 `PhoneAreaCodeException` 异常。
- 有过多数字的电话号码（三部分中的一个、区号、前缀号、电话号）应当抛出 `PhoneFormatException` 异常。
- 没有足够数字的电话号码（三部分中的一个）应当抛出 `PhoneFormatException` 异常。
- 带有非法字符（空格、字母，等等）的电话号码应当抛出 `PhoneFormatException` 异常。
- 缺少横杠的电话号码应当抛出 `PhoneFormatException` 异常。
- 有多个横杠的电话号码应当抛出 `PhoneFormatException` 异常。
- 一个空的（`null`）电话号码应当抛出 `PhoneFormatException` 异常。
- 一个不存在的文件应当抛出 `MissingOrBadFile` 异常。
- 一个空的（`null`）文件名也应当抛出那个异常。
- 一个没有内容的（`empty`）文件应当抛出 `MissingOrBadFile` 异常。
- 一个格式不正确的文件应当抛出 `MissingOrBadFile` 异常。

■ 习题 4：位于 70 页

一个刺绣用的自动缝纫机。这个控制它的类有几个基本的命令。
coordinates (0,0) 代表机器的左下角。随着你向右上角移动，x 和 y 不断增加，最终坐标为 $x = \text{TableSize.Width} - 1$ 而 $y = \text{TableSize.Height} - 1$ 。

坐标以厘米的分数形式来表示：

```
public void MoveTo(double x, double y);
public void SewTo(double x, double y);
public void SetWorkpieceSize(double width,
                             double height);
public Size WorkpieceSize { get; }
public Size TableSize { get; }
```

这儿有一些真实世界的约束可能会比较有意思；你当然不能缝纫稀薄的空气，而且你不能缝纫比机器更大的工件。

给定了这些需求，你可以想出哪些边界条件？

答案 4：

- 一个或者两个方向上的巨大数值坐标
- 巨形尺寸的工件
- 一个或者两个方向上的零或负值坐标
- 零或负值的工件尺寸
- 超出了工件的坐标
- 比台板更大的工件

■ 习题 5：位于 70 页

一个音频/视频的位移编辑器。一个提供控制 VCR 或者卡带的函数的类。其中有一个“当前位置”的记号，位于带子的开头处 (BOT) 到带子的结尾处 (EOT) 之间的某个地方。

你可以询问当前的位置，并从那移动到其他给定的位置。快进从当前位置向 EOT 移动一段距离。倒带从当前位置向 BOT 移动一段距离。

当带子第一次被载入时，它自动位于 BOT。

```
using System;

public interface AVTransport {

    /// Move the current position ahead by this many
    /// seconds. Fast-forwarding past end-of-tape
    /// leaves the position at end-of-tape
    void FastForward(double seconds);

    /// Move the current position backwards by this
    /// many seconds. Rewinding past zero leaves
    /// the position at zero
    void Rewind(double seconds);

    /// Return current time position in seconds
    double CurrentTimePosition();

    /// Mark the current time position with label
    void MarkTimePosition(String name);

    /// Change the current position to the one
    /// associated with the marked name
    void GotoMark(String name);

}
```

答案 5:

- 验证起始位置是 BOT。
- 以允许的（没有越过带子的结尾）距离快进一段，然后以同样的距离倒回。应当还是位于起始的位置。
- 以允许的（没有越过带子的开头）距离倒退一段，然后以同样的距离快进。应当还是位于起始的位置。
- 快进越过带子的结尾，然后用同样的距离倒回。应当比起始位置要靠前一段合适的距离以反映你不能越过带子的结尾前进。
- 用另一方向（倒带越过带子的开头）尝试同样的事情。
- 标记数个位置并且在把当前位置四处乱移之后回到那些位置。
- 标记一个位置并且在没有移动到别处前返回那儿。

■ 习题 6：位于 71 页

音频/视频位移编辑器，2.0 版。除了前面说过的，现在你还能用秒、分钟或者帧（在本例中，一秒就是固定的 30 帧）来定位，而且还能相对开头和结尾进行移动。

答案 6:

使用不同的单位交叉检查结果：用一种单元移动并用另外一种单位验证你的位置；用一种单位前进，用另外一种单位移回来，等等。

■ 习题 7：位于 134 页

设计一个利息计算器，它能基于两个日期之间的工作日的数目计算利息。使用测试优先的设计，一步步慢慢来。

答案 7:

1. 这是一个可能的场景，是你可能采取的步骤。真正正确的答案是没有的；这个练习仅仅是让你对测试优先的设计方法有所思考。

首先从简单地计算任意两个日期之间的天数开始。这些测试可能包括：

- 开始日期和结束日期用同一个值。
- 尝试普通的开始日期小于结束日期的情况。
- 尝试错误的开始日期大于结束日期的情况。
- 尝试越过了年关的日期（比如从 2003 年 10 月 1 日到 2004 年 3 月 1 日）。
- 尝试长度大于一年的日期（从 2003 年 10 月 1 日到 2006 年 12 月 1 日）。

2. 然后，从计算中除去周末，使用同样类型的测试。

3. 现在除去法定节假日和/或者公司节假日。这导致了可能的计算利息的疑问：你如何定义节假日？你必须在编写测试的时候面对这些问题；你是否认为这么做改进了你的接口？

4. 最终，执行利息计算本身。你可能从诸如这样的测试开始：

- 利息的数量永远都不应该是负数（不变量）。
- 开始日期和结束日期相等时利息应当是 0.0。

索引

Index

Symbols

0-1-*n* rule, 65

A

A-TRIP, 95

A/V transport exercise, *see*

Exercises, A/V transport

AccessController class, 80

Accessors, 101

Actual, 29

AddCropMarks(), 133

ADO (mock objects), 84

Agile, 46

Amount of test code, 101

Anonymous array, 16

Arianne 5 rocket, 6

Assert

 custom, 60

 definition, 29

Assert class, 29

AreEqual(), 29

Assert.AreEqual(), 13

Assert.AreSame(), 30

Assert.Fail(), 31

Assert.IsFalse(), 31

Assert.IsNull(), 30

Assert.IsTrue(), 13, 30, 137

Assumptions, 62

Automatic, 96

Automation, xii

B

Bad magic, 95

Bearing.cs, 59

Big ball of mud, 111

Blank, 63

Boolean conditions, 13, 30

Boundary conditions, 49, 55

Breaking the build/tests, 108

Broccoli, 1

Bugs

 clumping, 97, 98

 elusive, 140

 fixing, 102

 identifying likely, 45

 in sort routines, 58

 isolating, 10, 31

 list position, 24, 57

 memory, 138

 phantom, 99

 rewriting due to, 10

 traceable to unit tests, 100

Build machine, 96

Business logic, 66, 130

C

C#

 library versions, 140

C# Exception, *see* Exception

Cardinality, 64

Career limiting move, 10

[Category], 35

Checker.cs, 77

checkInvariant(), 61

Class Invariant, *see* Invariant

Clean room, 135

Code examples

 finding the source to, xiv

Collateral damage, 130

 definition, 8

Concurrency, 66, 68, 140

Confidence, 3

Conformance, 56

Continuous build/integration,
96, 110
Copy and paste, 40, 101
CORRECT, 55
Costs, 10
Coupling, 73, 140
Cross-checking, 50
CruiseControl, 96, 141
CVS, 108

D

Data structures, 60
DateTime, 74
Daylight savings time, 67
Debugging, 2, 10
Dependencies, 62, 140
Developer sandbox, 99, 104
Donne, John, 100
DotNetMock, 79–94, 141
DRY principle, 100, 134
definition, 40n
Dynamic mock objects, 89

E

E-mail address format, 56
Elephant
how to eat, 113
Encapsulation, 59, 74, 105
Engineering, 5
Environment, *see* Test code,
environment
Environmental.cs, 76
Environmental constraints, 52
Equality, 13
deceptive, 139
Error conditions, 51
Examples, *see* Code examples
Exception, 25, 29, 41, 42, 63,
114
Excuses, 7
Exercises
A/V transport, 70, 151
fax machine, 70, 150
interest calculator, 134
sewing machine, 70, 151
shopping cart, 69, 148
stack, 68, 147
Existence, 63
Expectation (mock objects), 82
Expected, 29
[ExpectedException], 41, 42

External dependencies, 62, 140
Extreme Programming, 104, 113

F

Factory class, 133
Failing tests, *see* Test code,
broken
Fax machine exercise, *see*
Exercises, fax machine
Feedback, xiii, 102, 110, 132
Fence post errors, 64
FilterRanges(), 53
Finally, 127
Floating-point numbers, 29, 138
Formal testing, 4

G

GMT, 67
Good neighbor, 127

H

House of cards, 4

I

IDE, 11, 96
Improving tests, 102
Independent, 38, 99
Indexing concepts, 62
Input data validation, 134
Interest calculator exercise, *see*
Exercises, interest
calculator
Invariant, 60, 61, 130
on an index, 131
Inverse relationships, 50

J

JamItIntoPark(), 62

K

KitchenOrder(), 58

L

Largest(), 14–26
Largest.cs, 15
Legacy code, 110, 111
Lighting doubles, 73
Long-running tests, 139

M

Message, 29
 Mock objects, 12, 73–94, 96, 140
 definition, 75
 dynamic, 89
 steps to using, 75
 MockObject class, 82
 MockSystemEnvironment.cs, 76
 MockSystemEnvironment, 77
 MoneyAssert.cs, 40
 MP3 player exercise, *see*
 Exercises, MP3 player
 MyStack.cs, 60–62

N

NCover, 97, 141
 NMock, 90n, 141
 Now, 74
 Null, 30, 63
 Numeric overflow, 6n
 NUnit, 141
 attributes, 32, 33, 35, 38, 41, 42
 custom asserts, 40
 and exceptions, 41
 minimum framework, 32
 order of tests, 100
 selecting tests, 33
 NUnit-Addin, 142

O

Object identity, 30
 Off-by-one errors, 24, 64
 Open Source
 definition, 17
 Ordering, 57

P

Pair programming, 113
 Pay-as-you go model, 9
 Performance, 52
 Phantom bugs, 99
 Postconditions
 definition, 63
 Pragmatic Automation, xlii
 Pragmatic Programmers
 email address, xv
 website, xliin
 Pragmatic Programming, 142

Pragmatic Starter Kit, xi
 Pragmatic Version Control, xi, 108
 Preconditions
 definition, 63
 Private access, 105
 Production code, 5, 100
 definition, 27
 Production system, 12
 Professional, 100, 114
 Properties file, 120
 Property accessors, *see* Accessors
 Protected access, 105
 Prototype, 11
 Public access, 105

R

Range, 59
 Recipe.cs, 124
 RecipeFile.cs, 125
 Recipes.cs, 120–124
 Refactoring, 119, 140
 Reference, 62
 Regression, 52
 Repeatable, 99
 Requirements, 6, 25, 46, 66, 139
 Restaurant order, 57
 Results
 analyzing, 7, 96, 138
 Retrospectives, 113
 Return on investment, 111
 Reviews, 113
 Right, 46
 RIGHT-BICEP, 45

S

Sandbox, 99, 104
 Scientific applications, 30
 SendFax(), 70, 150
 Separation of concerns, 117
 [SetUp], 38
 Setup code
 execution order, 39
 Sewing machine exercise, *see*
 Exercises, sewing machine
 Shopping cart exercise, *see*
 Exercises, shopping cart
 “Shy” code, 117
 Side-effects, 63
 Single testing phase, 9

sleep, 118
 SleepUntilNextHour(), 117
 Smoke test, 137
 Software engineering, 5
 Sort routines, 58
 SQL (mock objects), 84
 Stack exercise, *see* Exercises,
 stack machine
 Stand-ins, 73
 StandardPaperFactory, 133
 String constants, 128
 Stubs, 74
 [Suite], 33
 Synchronized, 68
 Syntax vs. semantics, 11
 System.Data, 84
 SystemEnvironment.cs, 76

T

Team communication, 114
 Team environment, 108
 [TearDown], 38
 Teardown code
 execution order, 39
 [Test], 32
 Test code
 and property accessors, 101
 broken, 31, 137, 139
 cleanup, 127
 compiling, 28
 correlate to bugs, 100
 and data files, 46
 environment, 138
 first test, 15
 invoking, 96
 linear, 100
 locating, 105
 long running, 139
 ordering, 100
 vs. *production code*, 28, 101
 required actions, 28
 results, 7
 reviewing, 114
 selecting, 33
 testing, 102
 Test coverage analysis tools, 97,
 98
 Test data, 48
 Test setup
 per-class, 39
 per-test, 38

Test suites
 definition, 33
 Test-driven design, 114, 132
 TestAdd(), 103
 [TestFixture], 32
 Testing
 acceptance, 3, 12
 and design, architecture,
 25, 117
 courtesy, 108
 environment, 138
 excuses, 7
 formal, 4
 frequency, 109
 functional, 12
 GUI, 129
 metrics, 98
 performance, 3, 12
 regression, 52, 111
 responsibility, 135
 TestLargest.cs, 16
 TestLargest, 26
 TestLargestDataFile, 47
 TestMyStack(), 61
 TestRecipe.cs, 125
 TestSimple.cs, 32
 Thorough, 97
 Time, 8, 10, 66, 139
 Timeouts, 67
 Tolerance, 138
 Traveling salesman algorithm, 35

U

Unit testing
 definition, 3
 intentional sabotage, 103
 potential dangers, 95
 using, 32
 UTC, 67

V

Validation, 46
 and verification, 3, 12
 formatted data, 57
 input data, 134
 user input, 134
 Version control, xl, 108

W

Wall-clock time, 67
 Whac-a-Mole, 8

X

XML, 46
xUnit, 142

Z

Zero, 63

[G e n e r a l I n f o r m a t i o n]

书名=单元测试之道C#版：使用NUnit_ [美] 托马斯 D a v i d T h o m a s 著 陈伟柱
陶文译_11340772

作者= B E X P

S S 号=

加密地址=

页数= 1 5 9

下载位置= h t t p : / / b o o k 2 . 5 r e a d . c o m / 3 0 0 - 1 1 / d i s k n r j s / n r j
s 1 4 1 / 0 3 / ! 0 0 0 0 1 . p d g

前言
目录
正文