软件工程实践丛书

PEARSON Prentice Hall

敏捷软件开发

原则、模式与实践

Agile Software Development

Principles, Patterns, and Practices

- 第13届软件开发震憾大奖获奖作品
- 国际软件工程和开发大师最新力作
- 众多名家一致推荐的敏捷开发指南
- 软件工程发展史上的里程碑性巨著

(美) Robert C. Martin 著

邓辉 译 孟岩 审

清华大学出版社

敏捷软件开发
原则、模式与实践

内容简介

在本书中，享誉全球的软件开发专家和软件工程大师Robert C. Martin将向您展示如何解决软件开发人员、项目经理及软件项目领导们所面临的最棘手的问题。这本综合性、实用性的敏捷开发和极限编程方面的指南，是由敏捷开发的创始人之一所撰写的。

- 讲述在预算和时间要求下，软件开发人员和项目经理如何使用敏捷开发完成项目。
- 使用真实案例讲解如何用极限编程来设计、测试、重构和结对编程。
- 包含了极具价值的可多次使用的C++和Java源代码。
- 重点讲述了如何使用UML和设计模式解决面向客户系统的问题。

关于作者

Robert C. Martin

Robert C. Martin是Object Mentor公司的总裁。Martin和他的软件咨询队伍使用面向对象设计、模式、UML、敏捷方法学和极限编程，在世界各地都有他们的客户。他还是好几本畅销书的作者，包括《Designing Object-Orient C++ Applications Using the Booch Method》(Prentice Hall, 1995)。Martin博士还是《Pattern Languages of Program Design 3》(Addison-Wesley, 1997)一书的主编、《More C++ Gems》(Cambridge, 1995)一书的编辑，以及《XP in Practice》(Addison-Wesley, 2001)的合作作者。他还是1996-1999年《C++ Report》杂志的总编，并多次在国际会议和展览中发表富有特色的演讲。

关于JOLT大奖

从1990年开始，美国权威的“软件开发”杂志每年从图书、设计工具等6个类别近千个候选产品中评选出每个类别的1个年度震憾大奖（Jolt Award）和2-3个效率大奖（Productivity Award），以表彰这些产品在推动软件开发方面的杰出贡献。回顾历史获奖名单，入选的均为对业界产生深远影响的里程碑式作品。

名家推荐

“这是第一本将敏捷方法、模式和当代软件开发基础揉合在一起的书。当Bob Martin发言的时候，我们最好洗耳恭听。”
—— John Vlissides，《Design Patterns》与《Pattern Hatching》的作者

“我等这本书很久了，Bob有太多可以左右我们技术的实践经验可以向我们传授。”
—— Martin Fowler，《UML Distilled》与《Refactoring》的作者

“在本书中，Bob Martin以一个开发大师和教育家的身份向我们献上了一份大礼。他的实用的技术和流畅的文笔给我们以启迪。”
—— Craig Larman，《Applying UML and Patterns》的作者

“本书充满了对于软件开发的实用建议。不管你是想成为一个敏捷开发人员，还是只想提高技能，本书都同样有用。我一直在期盼着本书的出版，它没有令我失望。”
—— Erich Gamma，《Design Patterns》的作者

“我对Uncle Bob的第一印象：‘优秀的对象思想’。你手中持有的是能让你受益终生的‘优秀的对象思想’。”
—— Kent Beck，《Extreme Programming Explained》的作者

关注Robert C. Martin的下一本书…… 详情请参阅本书末的“读者反馈卡”

ISBN 7-302-07197-7
9 787302 071976 >
定价：59.00元

PEARSON Prentice Hall

文稿编辑：尤晓东
封面设计：付剑飞
读者信箱：Book@21bj.com
信息网站：www.AgileSoftwareDevelopment.cn
www.ePress.cn
www.34.cn
http://www.pearsoned.com

软件工程实践丛书

敏捷软件开发

原则、模式与实践

（美）Robert C. Martin 著
邓 辉 译
孟 岩 审

清华大学出版社
北 京

内 容 简 介

享誉全球的软件开发专家和软件工程大师 Robert C. Martin 向您介绍如何解决软件开发人员、项目经理及软件项目领导们所面临的最棘手的问题。这本综合性、实用性的敏捷开发和极限编程方面的指南，讲述了在预算和时间要求下软件开发人员和项目经理如何使用敏捷开发完成项目；使用真实案例讲解如何用极限编程来设计、测试、重构和结对编程；包含了极具价值的可重用的 C++和 Java 源代码；还重点讲述了如何使用 UML 和设计模式解决面向客户系统的问题。

本书于 2003 年荣获第 13 届软件开发图书震撼大奖，适于用作高校计算机专业本科生、研究生和软件学院的软件工程和软件开发相关课程的教材或参考书，也适于软件开发和管理人员提高自身水平学习之用。

图书在版编目（CIP）数据

敏捷软件开发：原则、模式与实践/（美）马丁著；邓辉译，孟岩审. —北京：清华大学出版社，2003

（软件工程实践丛书）

书名原文：Agile Software Development: Principles, Patterns, and Practices

ISBN 7-302-07197-7

Ⅰ.敏… Ⅱ.①马… ②邓… ③孟… Ⅲ.软件开发 Ⅳ.TP311.52

中国版本图书馆 CIP 数据核字（2003）第 078273 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦

http://www.tup.com.cn　　**邮　　编**：100084

社 总 机：010-62770175　　**客户服务**：010-62776969

文稿编辑：尤晓东

封面设计：付剑飞

印 刷 者：北京国马印刷厂

发 行 者：新华书店总店北京发行所

开　　本：185×260　**印张**：31.25　**插页**：2　**字数**：942 千字

版　　次：2003 年 9 月第 1 版　2003 年 9 月第 1 次印刷

书　　号：ISBN 7-302-07197-7/TP・5239

印　　数：1～5000

定　　价：59.00 元

敏捷软件开发宣言

我们正在通过亲身实践以及帮助他人实践，揭示更好的软件开发方法。通过这项工作，我们认为：

个体和交互	胜过	过程和工具
可以工作的软件	胜过	面面俱到的文档
客户合作	胜过	合同谈判
响应变化	胜过	遵循计划

虽然右项也具有价值，

但我们认为左项具有更大的价值。

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

敏捷宣言遵循的原则

我们遵循以下原则：

- 我们最优先要做的是通过尽早的、持续的交付有价值的软件来使客户满意。
- 即使到了开发的后期，也欢迎改变需求。敏捷过程利用变化来为客户创造竞争优势。
- 经常性地交付可以工作的软件，交付的间隔可以从几个星期到几个月，交付的时间间隔越短越好。
- 在整个项目开发期间，业务人员和开发人员必须天天都在一起工作。
- 围绕被激励起来的个体来构建项目。给他们提供所需的环境和支持，并且信任他们能够完成工作。
- 在团队内部，最具有效果并且富有效率的传递信息的方法，就是面对面的交谈。
- 工作的软件是首要的进度度量标准。
- 敏捷过程提倡可持续的开发速度。责任人、开发者和用户应该能够保持一个长期的、恒定的开发速度。
- 不断地关注优秀的技能和好的设计会增强敏捷能力。
- 简单——使未完成的工作最大化的艺术——是根本的。
- 最好的构架、需求和设计出自于自组织的团队。
- 每隔一定时间，团队会在如何才能更有效地工作方面进行反省，然后相应地对自己的行为进行调整。

面向对象设计的原则

SRP　单一职责原则
就一个类而言，应该仅有一个引起它变化的原因。

OCP　开放—封闭原则
软件实体（类、模块、函数等）应该是可以扩展的，但是不可修改。

LSP　Liskov 替换原则
子类型必须能够替换掉它们的基类型。

DIP　依赖倒置原则
抽象不应该依赖于细节。细节应该依赖于抽象。

ISP　接口隔离原则
不应该强迫客户依赖于它们不用的方法。接口属于客户，不属于它所在的类层次结构。

REP　重用发布等价原则
重用的粒度就是发布的粒度。

CCP　共同封闭原则
包中的所有类对于同一类性质的变化应该是共同封闭的。一个变化若对一个包产生影响，则将对该包中的所有类产生影响，而对于其他的包不造成任何影响。

CRP　共同重用原则
一个包中的所有类应该是共同重用的。如果重用了包中的一个类，那么就要重用包中的所有类。

ADP　无环依赖原则
在包的依赖关系图中不允许存在环。

SDP　稳定依赖原则
朝着稳定的方向进行依赖。

SAP　稳定抽象原则
包的抽象程度应该和其稳定程度一致。

极限编程实践

完整团队

XP 项目的所有参与者（开发人员、业务分析师、测试人员等等）一起工作在一个开放的场所中，他们是同一个团队的成员。这个场所的墙壁上随意悬挂着大幅的、显著的图表以及其他一些显示他们进度的东西。

计划游戏

计划是持续的、循序渐进的。每 2 周，开发人员就为下 2 周估算候选特性的成本，而客户则根据成本和商务价值来选择要实现的特性。

客户测试

作为选择每个所期望的特性的一部分，客户定义出自动验收测试来表明该特性可以工作。

简单设计

团队保持设计恰好和当前的系统功能相匹配。它通过了所有的测试，不包含任何重复，表达出了编写者想表达的所有东西，并且包含尽可能少的代码。

结对编程

所有的产品软件都是由两个程序员、并排坐在一起在同一台机器上构建的。

测试驱动开发

程序员以非常短的循环周期工作，他们先增加一个失败的测试，然后使之通过。

改进设计

随时改进糟糕的代码。保持代码尽可能的干净、具有表达力。

持续集成

团队总是使系统完整地被集成。

集体代码所有权

任何结对的程序员都可以在任何时候改进任何代码。

编码标准

系统中所有的代码看起来就好像是被单独一个——非常值得胜任的——人编写的。

隐喻

团队提出一个程序工作原理的公共景像。

可持续的速度

团队只有持久才有获胜的希望。他们以能够长期维持的速度努力工作。他们保存精力，他们把项目看作是马拉松长跑，而不是全速短跑。

中文版序：软件之美

除了我的家庭，软件是我的挚爱。通过它，我可以创造出美的东西。软件之美在于它的功能，在于它的内部结构，还在于团队创建它的过程。对用户来说，通过直观、简单的界面呈现出恰当特性的程序就是美的。对软件设计者来说，被简单、直观地分割，并具有最小内部耦合的内部结构就是美的。对开发人员和管理者来说，每周都会取得重大进展，并且生产出无缺陷代码的具有活力的团队就是美的。美存在于所有这些层次之中，它们都是本书内容的一部分。

软件开发人员如何学到创造美的知识呢？在本书中，我讲授了一些原则、模式以及实践，它们可以帮助软件开发人员在追求美的程序、设计以及团队的道路上迈出第一步。其中，我们探索了基本的设计原则，软件设计结构的通用模式以及有助于团队融为一个有机整体的一系列实践。由于本书是关于软件开发的，所以包含了许多代码。仔细研究这些代码是学习本书所教授的原则、模式以及实践的最有效方法。

人们需要软件——需要许多的软件。50 年前，软件还只是运行在少量大型、昂贵的机器之上。30 年前，软件可以运行在大多数公司和工业环境之中。现在，移动电话、手表、电器、汽车、玩具以及工具中都运行有软件，并且对更新、更好软件的需求永远不会停止。随着人类文明的发展和壮大，随着发展中国家不断构建它们的基础设施，随着发达国家努力追求更高的效率，就需要越来越多的软件。如果在所有这些软件之中，都没有美存在，这将会是一个很大的遗憾。

我们知道软件可能会是丑陋的。我们知道软件可能会难以使用、不可靠并且是粗制滥造的；我们知道有一些软件系统，其混乱、粗糙的内部结构使得对它们的更改既昂贵又困难；我们还见过那些通过笨拙、难以使用的界面展现其特性的软件系统；我们同样也见过那些易崩溃且行为不当的软件系统。这些都是丑陋的系统。糟糕的是，作为一种职业，软件开发人员所创建出来的美的东西却往往少于丑的东西。如果你正在阅读这本书，那么你也许就是那个想去创造美而不是丑的人。

最好的软件开发人员都知道一个秘密：美的东西比丑的东西创建起来更廉价，也更快捷。构建、维护一个美的软件系统所花费的时间、金钱都要少于丑的系统。软件开发新手往往不理解这一点。他们认为做每件事情都必须要快，他们认为美是不实用的。错！由于事情做得过快，他们造成的混乱致使软件僵化，难以理解。美的系统是灵活、易于理解的，构建、维护它们就是一种快乐。丑陋的系统才是不实用的。丑陋会降低你的开发速度，使你的软件昂贵而又脆弱。构建、维护美的系统所花费的代价最少，交付起来也最快。

我希望你能喜爱这本书。我希望你能像我一样学着以创建美的软件而骄傲，并享受其中的快乐。如果你从本书中略微看到了这种快乐，如果本书使你开始感受到了这种骄傲，如果本书点燃了你内心欣赏这种美的火花，那么就远超过我的目标了。

Robert C. Martin
2003 年 8 月 21 日

The Beauty of Software

Robert C. Martin's foreword for Agile Software Development Chinese Version

Next to my family, software is my passion. It is a medium in which I try to create beauty. The beauty of software is in it's function, in it's internal structure, and in the way in which it is created by a team. To a user, a program with just the right features presented through an intuitive and simple interface, is beautiful. To a software designer, an internal structure that is partitioned in a simple and intuitive manner, and that minimizes internal coupling, is beautiful. To developers and managers, a motivated team of developers making significant progress every week, and producing defect-free code, is beautiful. There is beauty on all these levels, and they are all part of the topic of this book.

How do software developers learn how to create this beauty? In this book I teach the principles, patterns, and practices that can help software developers take the first steps towards beautiful programs, designs, and teams. In these pages we explore basic design principle, common patterns in the structure of a software design, and a set of practices that can help a team knit itself into a functioning whole. Since this book is about software development, it contains a lot of code. The principles, patterns, and practices that this book teaches are learned most effectively by carefully studying that code.

Our world needs software -- lots of software. Fifty years ago software was something that ran in a few big and expensive machines. Thirty years ago it was something that ran in most companies and industrial settings. Now there is software running in our cell phones, watches, appliances, automobiles,toys, and tools. And need for new and better software never stops. As our civilization grows and expands, as developing nations build their infrastructures, as developed nations strive to achieve ever greater efficiencies, the need for more and more software continues to increase. It would be a great shame if, in all that software, there was no beauty.

We know that software can be ugly. We know that it can be hard to use,unreliable, and carelessly structured. We know that there are software systems whose tangled and careless internal structures make them expensive and difficult to change. We know that there are software systems that present their features through an awkward and cumbersome interface. We know that there are software systems that crash and misbehave. These are ugly systems. Unfortunately, as a profession, software developers tend to create more ugly systems than beautiful ones. If you are reading this book, then you are someone who wants to create beauty and not ugliness.

There is a secret that the best software developers know. Beauty is cheaper than ugliness. Beauty is faster than ugliness. A beautiful software system can be built and maintained in less time, and for less money, than an ugly one. Novice software developers don't understand this. They think that they have to do everything fast and quick. They think that beauty is impractical. No! By doing things fast and quick, they make messes that make the software stiff, and hard to understand. Beautiful systems are flexible and easy to understand. Building them and maintaining them is a joy. It is ugliness that is impractical. Ugliness will slow you down and make your software expensive and brittle. Beautiful systems cost the least to build and maintain, and are delivered soonest.

I hope you enjoy this book. I hope you learn to take as much pride and joy in the creation of beautiful software that I do. If this book can show you just an inkling of that joy, motivate you to feel just the beginnings of that pride, and provide just a spark of appreciation for that beauty, then it has more than accomplished my goal for it.

七年等待换来的经典(代序)

孟 岩

Robert C. Martin 的经典著作《敏捷软件开发》（*Agile Software Development*）中文版面世，这是计算机技术出版领域的一件大喜事。即使在技术图书非常繁荣的今天，本书的问世也仍然是值得广大开发者格外留意和关注的事件。这不仅是因为它刚刚荣获 2002 年度 Jolt 震撼大奖，更因为这本书本身的价值和独特魅力所在。

Robert Cecil Martin 是软件开发领域里响当当的名字。20 世纪 70 年代，当他还是个年轻小伙子的时候就已经是一位有名的 UNIX 黑客。经过长期的开发实践后，他成了软件开发领域中的知名专家。20 世纪 90 年代初，Rational 软件公司首席科学家 Grady Booch 邀请他加盟，希望借助他丰富的实践经验，结合 Booch 自己的软件设计理念，开发一套创新性的软件产品。这就是大名鼎鼎的 Rational ROSE 的早期版本！在 Rational 公司期间，Martin 丰富的实践经验与 Booch 深厚的理论功底形成了完美的组合，把面向对象设计的理论与实践推向了高峰。1994 年，Grady Booch 的经典著作 *Object-Oriented Analysis and Design with Applications* 问世，几乎同时，Martin 的第一本著作 *Designing Object-Oriented C++ Application Using the Booch Method* 也由 Prentice Hall 出版。这两本书相互辉映，当时引起了很大的反响。Martin 的著作结合了当时最流行的面向对象语言 C++和最出色的面向对象设计建模方法 Booch Method，以大量实例讲解技术概念和应用方法，分析透彻、讲解务实、技术精妙，在读者中声誉极佳。该书不仅为 Martin 确立了软件设计领域顶尖专家的地位，而且奠定了他独特的写作风格。Martin 本人对该书非常有感情，多年后，在我给他写信谈到这本书时，他还很得意地宣称：这是他的代表作，其中的大部分内容即使在今天也不过时。没有十年磨一剑的功力，又怎能创造出长盛十年而不衰的经典？

大约 2001 年 10 月，我有机会复印了 Martin 那本 1995 年的名著，阅读之后，大为震撼，并决定不惜重金从国外购买原版，同时尽力向其他朋友推荐。不少朋友阅读之后亦非常兴奋，甚至有人说："读此书方知什么是面向对象设计"，可见对该书的评价之高。惟一可惜的是，该书年代久远，不少内容有些陈旧了。很自然地，我和我的朋友们对于该书的第 2 版都是翘首以待。

大约在 2002 年 4 月，我从 Amazon 网站上看到了该书第 2 版的预订信息。当时的名字叫做 *Designing Object-Oriented C++ Application Using UML*。通过因特网上的调查，我了解到整个面向对象设计开发社群都在高度关注这本书的写作进展情况。已经被冠以"鲍勃大叔"昵称的 R. C. Martin 曾经允诺在 1998 年推出此书，但结果一拖再拖，搞得整个开发社群的人们"群情激愤"。正如本书前言开头有一幅漫画插图所示，一位叫做 Claudia Frers 的女士在 1999 年的 UML World 大会上愤怒地声讨鲍勃大叔："可是你说过，去年就可以完成这本书啊！"。面对读者爱极而怒的反应，我想鲍勃大叔虽然表面上不免尴尬，内心却一定是暗喜的。而包括他在内的所有人，当时可能都想不到，这本书居然还要拖上 3 年，才会最终以全新的面貌出现在读者面前。

看到该书的预订页面后，我立刻请求一位师长帮我从 Amazon 网站上预订。之后，我就开始了漫长的等待过程。这是多么奇异的一段等待啊！我先是发现这本书的名字变了，变成了 *Designing Object-Oriented C++*，没有"UML"了。书名变短了，不过看来内容是扩充了，因为预告篇幅增大到了 700 多页，这倒也不错。可是到了大约 2002 年 8 月份，我突然发现，这本书不见了！在原来的网页上出现了一本名叫"*Agile Software Development*"（《敏捷软件开发》，简称 ASD）的书，虽然作

者也是 Robert C. Martin，但只有 500 多页了。书名变短了，内容也缩水了！我当然很激愤，想不到 Amazon 也会来偷梁换柱这一招！忍耐了一段时间之后，我在 comp.object 上见人就问："您知道 Bob 大叔的那本 C++书第 2 版到哪儿去了吗？那本 *Agile Software Development* 又是怎么回事？我的 money 还能要得回来吗？……"其实我很清楚，Bob 大叔本人就经常在这里活动，我这招"敲山震虎"迟早会引得 Bob 大叔本人出来解释的。果然，两天后我就收到署名"Uncle Bob"的来信，信中客气地说，这本 *Agile Software Development* 就是原来那本书的第 2 版。不过内容、风格已经完全变化了，应该算是一本全新的书。虽然书中大多数代码是用 Java 写的，但是 Bob 大叔向我保证，这本书的内容绝对对得起我付出的每一分银子。

OK，我还有什么可说的呢？Bob 大叔亲自出来解释，我当然满意了。2002 年 10 月底，一个大大的包裹放到了我的桌子上——ASD 到了。其时不单国内很少有人拿到这本书，就是在美国，该书签名首发的活动也还在筹备中。因此，我大概可以算是全国最早阅读此书的前 5 人之一了吧。

从 1995 年第 1 版面世算起，整个面向对象设计社群经过了 7 年的漫长等待，终于等到了这本书。7 年中，斗转星移，物是人非，多少英雄升起又落下，多少神话创生又破灭！当这本硬壳封面的精美图书沉甸甸地放在手上，怎能不让人感慨！

阅读本书的过程，是一个充满了发现、领悟和兴奋的过程。我只读了其中几个章节，就已经知道本书大大超过了事前的预期，是当代软件开发领域最杰出的著作之一。此时，著名的美国 *Software Development* 杂志每年一度的 Jolt 评奖工作正式开始，我毫不犹豫地给本书投了一票。后来我愉快地得知，本书果然如我所愿夺得了 2002 年度技术图书类最高奖项——第 13 届震憾大奖（Jolt 大奖）。我在 Amazon 网站上对本书的评论中，我的结论写道：本书不但肯定是 2002 年度最好的设计类技术图书，而且在今后的几年中，也很难出现超过它的作品。那篇短评也算是为本书的宣传做了一点点事，为此我感到非常高兴。

听说在台湾省，3 位资深工程师联名上书某出版公司要求引进此书，并且自告奋勇担任翻译工作。这些资深工程师收入丰厚，根本不是为了那点微薄的翻译酬劳而来，完全是希望能够把这本极有价值的好书介绍给更多的业内人士，造福大众。一本书能够引起这么大的热诚，可见它的魅力了。

那么本书究竟好在哪里呢？真正关心这个问题的读者，应该先阅读原著。Bob 大叔是位实践大师，读者必须通过阅读实践，才能够真正体会到本书的精妙之处。我在此只谈谈本书给我印象最深的三个特点。

首先，该书延续了 Bob 大叔著作的一贯特色，以实例为本，不尚空谈，因此格外真实，摄人心魄。Martin 从多年的写作中提炼出一种独特的风格，就是"引领式传授"。与别的作者不同，面对问题，Martin 并不是把最后那个完美的答案一下子放在你面前，让你拜倒在他脚下，被他的睿智折服。他很清楚，当读者在大师面前拜倒的同时，除了挫折感外，同时也会丧失自信。而自信是一个设计者由成功走向成功的最关键因素。因此，Martin 还原了一个真实的设计过程——带领着读者一起设计。读者在书中能够看到，作者也会犯错误，也要面临痛苦的选择，也会做一些愚蠢的决定，也会被一些"激动人心"的想法诱惑，但是，最后他能够跨越重重障碍，看透重重迷雾，得到优秀的设计。他跟我们一样，是一个有血有肉的设计者，而不是什么天赋异禀的天才。正因为如此，他能够达到的，我们也能够达到。这样的风格，使我们从阅读中能够逐渐体会到软件设计最精髓的东西：张力与平衡。如何选择，如何思考，如何面对困境，这些才是最宝贵的财富。这些财富，就在我们自己的大脑里，而 Martin 的书，就是打开这个财富宝库的钥匙。此外，读者在阅读本书的时候会发现，Martin 在教授具体技术的时候，始终以实例教学为主。比如关于 UML 的附录，就是通过两个

实例来教授 UML 的基础知识。诸位不妨从这两个附录开始阅读此书，体会一下作者的写作风格，看一看在 UML 学习方面，究竟是传统的罗列式教材给你的印象更深，还是以实例为依托的 Martin 式风格更有效率。

其次，详略得当。Martin 很清楚哪些东西应该讲得细致一些，哪些东西可以讲得抽象一些。这一点也是大师风范。软件设计这个领域，有很多东西是应该讲清楚，却很少有人能够讲得清楚的，比如设计中的权衡，实例的综合运用等等。还有很多东西，是没有必要像有些书那样长篇大论的，比如说一些基本的设计原则或模式，一些开发过程思想，一些技术理念等等。这些东西讲得简明扼要能够给人以智慧的启迪，讲得冗长拖沓，定下几十上百条规则方法，则必然脱离实际，堵塞和束缚读者的思想。Martin 的书可说决无此弊，这也应当归功于他多年的实践经验。从本书中我们可以看到，他对于思想和原则一般是言简意赅，意到为止。而对于有助于提高读者实际设计能力的实例，则不惜笔墨，详加阐述，可谓煞费苦心。

最后一点，也是相当重要的一点，本书选材匠心独到，精彩绝伦，特别是还有大量独创性和创新性的技术，来自 Martin 多年的研发实践，十分珍贵。比如本书的 Acyclic Visitor、Extension Object、Monostate、Taskmaster 等模式，并不属于经典的 23 模式，很多都是来自 Martin 自己的创造，配合实例解析，有助于大幅提高读者的设计实践能力。

总之，本书是近年来难得的佳作，希望大家都能好好阅读和体会这本书，它将为你的设计能力带来一个质的提升，同时，也有助我们对“怎样算是一本好的技术图书”有更进一步的认识。

译 者 序

关于软件开发方面的书真是不少，有过程方法的、有最佳实践的、还有设计原则的。但是当你真正进行软件开发实践时，却会发现这些书中告诉你的知识在实际运用的时候总是和期望的效果有一定的差距。我在这方面有深刻的体会。经过一段时间的反思，我隐约觉得，应该还有某个存在于过程方法、最佳实践以及设计原则之外的东西来有机地把它们结合起来，才能真正地发挥它们的最大效用。这种东西不是可以形式化的条条框框，而是活跃于人的大脑中的某种思维方法。看完了Robert C. Martin的《敏捷软件开发：原则、模式与实践》之后，我有一种豁然开朗的感觉。本书把这种思维方法阐述得再清晰不过了。

本书具有两大特色。第一，很多讲述软件开发的书籍，要么是仅仅涉及过程方法方面的内容，要么是仅仅涉及设计原则方面的内容。这些做法相对于整个软件开发活动来说都是片面的。其实，过程方法、设计原则以及最佳实践是一个不可分割的整体。孤立地去使用任何一部分都无法获得最佳的效果。最有效的方法应该是根据自己开发团队的实际情况，找出一种能够有效地把这三者结合起来并使它们相互支持的方法。比如，大家都知道每周（或每日）构建是一种得到广泛认可的最佳实践方法，但并不是只要你每周/日都去构建了就能得到好的效果，有时，结果可能会更糟。要想使这种方法有效，还需要其他方面的支援。每周/日构建的前提是软件必须是易于每周/日构建的。也就是说，你需要对软件中的依赖关系进行管理，使之具有每周/日构建的基础。而这种依赖关系的管理是需要设计原则来指导和度量的。这只是其中一个例子，本书中到处都体现着作者的这种主导思想和实践。如果读者能够在这个方面好好体会的话，肯定会对软件开发有一个更为全面、深入的理解，从而可以更加有效地去使用这些过程方法、设计原则以及最佳实践。

第二，本书的核心是软件设计，但是它对软件设计的理解以及讲解方式非常的特别。许多有关软件设计的书籍中，要么先讲述一些设计原则、模式，然后再给出几个简单的在理想情况下的应用；要么是拿一个最终的设计结果来剖析，然后告诉你它们是多么的优美。当时，你可能真会那么认为，但是当你试图在自己的实际开发中应用时，总会发现情况是完全不一样的。此时，你要么束手无策，要么会误用设计原则、模式。究其原因，主要是因为，在此类书中所讲述的不是真正的设计，只是设计的部分内容，而忽略了设计中最为重要的方面。设计是人的思维的一种动态活动，是设计者针对自己的问题的思索、权衡、折衷、选择的过程。其中会出现很多在理想情况下不会出现的问题，对这些问题的处理水平才是真正的设计水平。同样，本书中到处都是这样的思考过程。针对每个案例，作者都会和你一起思索、一起探讨、一起权衡、一起验证。本书中所展示的是一个个完整的设计活动过程。通过这些案例的学习，相信读者肯定会对设计有一个更深刻的理解。此外，本书中也讲述了很多的设计模式，但是和很多其他讲述模式的书不同的是，它更多的是在告诉你什么时候不要去使用模式，去抵制模式的诱惑，以免带来不必要的复杂性。在对模式狂热吹捧的今天，本书无疑是一剂纠偏良药，可以让你更加合理、有效地使用模式。

其实，这些内容正是软件开发活动中最本质，同时也是最难以琢磨的内容。要把这些内容通过文字表达出来更是非常困难的，这也是这方面的书籍凤毛麟角的原因。然而在本书中，Robert C. Martin先生能把这些内容编写得如此清晰、如此易于理解，充分展示了作者深厚的技术功底和卓越的表达能力。因此，本书能从众多优秀书籍中脱颖而出获得第13届Jolt大奖，就是意料之中的了。

本书主要包含4部分内容，这些内容对于今天的软件工程师都非常的重要，它们是：

- 敏捷方法：主要讲述了如何去使用敏捷方法，其中有很大一部分内容是告诉你为什么要这样做。
- 面向对象设计原则：本书包含了 12 个面向对象设计原则，涵盖了包的设计和类的设计。这是我所见过的对这方面内容讲解得最清晰、最彻底、最深刻的惟一的一本书。
- 设计模式：本书中讲述了 23 个设计模式，并都有具体的实例。讲解的重点如何在实际的应用中去使用模式，如何根据当前问题的上下文以及约束力去选择最适合的模式，以及何时避免使用模式。
- UML：本书不是关于 UML 的，但是为了让读者更好地理解书中的内容，作者使用了一些 UML 图来展示设计思路。同时，本书中也对如何有效地使用 UML 做了深入的阐述。本书中有两个附录专门对 UML 进行了简介。

总之，本书是写给一线软件工程师的。如果你想学习 UML，如果你想学习如何去设计软件，如果你想学习设计模式，如果你想学习最好的软件开发实践，那么请阅读本书。

感谢 Robert C. Martin 先生为我们写了一本如此优秀的著作，此书必将成为经典。我从本书中学到了很多。我相信本书也不会令您失望。

邓 辉，2003.8.7 于上海

又及：

- 感谢本书作者 Robert C. Martin 先生总是非常及时、耐心地回答我的询问，他的回答使本译本更加贴近作者的本意。
- 感谢本译本的责编尤晓东和审校孟岩先生，他们的努力使本译本增色不少。
- 最后要感谢的是在我的翻译过程中一直陪伴在我身边，给了我很大帮助的人。她就是我的妻子孙鸣。我把最诚挚的感谢献给她。任何珍宝都没有她的陪伴重要。

译者简介

邓辉，软件工程师，对软件设计、面向对象、泛型以及模式有深入的研究和实践，尤其喜欢敏捷软件开发。在 IBM developerWorks 上发表过多篇关于设计、模式以及测试方面的文章。联系信箱：dhui@263.net。

原英文版序

刚刚交付了 Eclipse 开放源码项目的一个主要的版本之后，我就立即开始写这篇序言。我仍然还在恢复之中，思维还有些模糊。但是有一件事情我却比以往更加清楚，那就是：交付产品的关键因素是人，而不是过程。我们成功的诀窍很简单：和那些沉醉于交付软件的人一起工作，使用适合于自己团队的轻量过程进行开发，并且不断地去适应。

如果去了解我们团队中的开发人员，就会发现他们都是一些认为编程是开发活动的中心的人。他们不仅编写代码，而且还经常对代码进行融会贯通，以保持对系统的理解。使用代码验证设计所提供的反馈，对于获得对设计的信心来说是至关重要的。我们的开发人员知道模式、重构、测试、增量交付、频繁构建以及其他一些改变了我们看待当今方法学方式的 XP（极限编程）最佳实践的重要性。

对于那些具有高技术风险以及变化的需求的项目来说，熟练地掌握这种开发风格是它们取得成功的先决条件。虽然敏捷开发对于形式和文档保持低调，但是当涉及重要的开发实践时，它却表现出极度的关注。让这些实践“活”起来正是本书的中心内容。

Robert 是面向对象社团中一位长期的积极参与者，对于 C++实践、设计模式以及面向对象设计的一般原则都有贡献。他是一位 XP 和敏捷方法的早期的和直率的提倡者。本书就构建于这些贡献之上，覆盖了敏捷开发实践的全部内容。这是一项了不起的成就。不仅如此，Robert 在说明每一件事情时，都使用了研究案例和大量的代码，完全和敏捷实践相符。他阐明编程和设计的方式就是实际去编程。

这本书中充满了对于软件开发的明智的建议。不管你是想成为一个敏捷（Agile）开发人员，还是想提高已有的技能，它都是同样有用的。我一直在期盼着这本书，它没有令我失望。

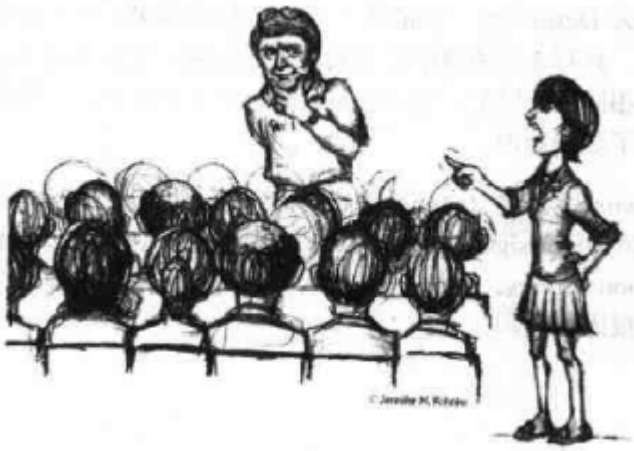
Erich Gamma
Object Technology International

献给 Ann Marie，Angela，Micah，Gina，Justin，Angelique，Matt 和 Alexis……

没有任何珍宝比我的家庭的陪伴更重要，

没有任何宝藏都比她们爱的安慰更丰富。

前　言



Bob，你说过会在去年写完这本书的。

——Claudia Frers, *UML World*, 1999

敏捷开发（Agile Development）是一种面临迅速变化的需求快速开发软件的能力。为了获取这种敏捷性，我们需要使用一些可以提供必要的纪律和反馈的实践。我们需要使用一些可以保持我们的软件灵活、可维护的设计原则，并且我们需要知道一些已经被证明针对特定的问题可以平衡这些原则的设计模式。本书试图把所有这3个概念编织在一起，使它们成为一个有机的整体。

本书首先描述了这些原则、模式以及实践，然后通过学习许多的案例来演示是如何应用它们的。更重要的是，研究案例介绍的并不是最终已完成的结果，而是设计的历程。你会看到设计者犯错误；你会看到他们是如何识别出错误并最终改正错误；你会看到他们对于难题的苦苦思索以及对于一些权衡和含糊问题的苦恼；你会看到设计的行为。

隐藏在细节之中

本书包含有许多的Java和C++代码。我希望你能够仔细地学习这些代码，因为在很大程度上，代码正是本书的要旨。代码就是本书要讲的内容的实现。

本书采用了一种重复的讲解方式。它由一系列不同规模的案例研究组成。有一些非常的小，有一些却需要用几章来描述。每个案例研究之前都有一些针对该案例研究的预备内容。例如，在薪水支付案例研究之前，就有一些描述在该案例研究中用到的面向对象设计原则和模式的章节。

本书首先对开发实践和过程进行了讨论，其中穿插了许多小的案例研究以及示例。从这些穿插点处，本书转移到设计和设计原则的主题上，接着是一些设计模式、更多的管理包的设计原则以及更多的模式。所有这些主题都附有案例研究。

因此，请准备好学习一些代码并钻研一些UML（统一建模语言）图。你将要学习的书籍是非常技术性的，并且其中要教授的知识和恶魔一样，也隐藏在细节之中。

一段小史

大约 6 年前，我写了一本名为：*Designing Object-Oriented C++ Application using the Booch Method* 的书。它是我的一部主要作品，并且它的效果和销量都令我非常满意。

本书开始时是被作为 *Designing* 一书的第 2 版的，但是结果却并非如此。在本书中所保留的原书中的内容是非常少的。只有 3 章的被维持下来，并且对这些章节进行了重大的修改。书的意图、精神以及许多的知识是相同的。但是，自 *Desinging* 出版 6 年以来，在软件设计方面我又学到了非常多的知识。本书表现了这些知识。

几年过去了！*Designing* 刚好在 Internet 爆炸式流行之前出版。从那时起，我们要面临的缩略词的数量已经翻了一倍，诸如：Design Patterns、Java、EJB、RMI、J2EE、XML、XSLT、HTML、ASP、JSP、Servlets、Application Servers、ZOPE、SOAP、C#、.NET 等等。我要告诉你，使本书的内容跟得上最新的技术知识是很困难的。

与 Booch 的合作

1997 年，Booch 和我联系，让我帮他撰写他的非常成功 *Object-Oriented Analysis and Design with Applications* 一书的第 3 版。以前，我和 Grady 在一些项目中有过合作，并且我是他的许多作品（包括 UML）的热心读者和参与者。因此，我高兴地接受了。我邀请了我的好朋友 Jim Newkirk 来帮助完成这项工作。

在接下来的 2 年中，我和 Jim 为 Booch 的书撰写了许多的章节。当然，这些成果意味着我不可能在这本书中按照我本来想的那样投入同样多的努力，但是我觉得 Booch 的书值得我这样做。另外，当时本书完全只是 *Designing* 的第 2 版，并且我的心思也不在其上。如果我要说一些东西的话，我想说一些新的并且不同的东西。

糟糕的是，Booch 的这个版本的书并没有完成。在正常的时间里很难抽出空来撰写一本书。在浮躁的“.com”泡沫期间，这几乎是不可能的。Grady 也更加忙于 Rational 以及一些像 Capapulse 这样的新风险投资企业的事务。因此这项工作就停止了。最后，我问 Grady 和 Addison-Wesley 是否可以把我和 Jim 撰写的那些章节包含在这本书中。他们很有风度地同意了。于是，一些案例研究和 UML 的章节就从此而来。

极限编程的影响

1998 年晚期，XP 崭露头角，并向我们所珍爱的关于软件开发的观念进行挑战。我们是应该在编写任何代码前先创建许多 UML 图呢？还是应该避开任何种类的 UML 图而仅仅编写大量代码呢？我们是应该编写大量的描述我们设计的叙述性文档呢？还是应该努力使代码具有自释义能力以及表达力，这样辅助性的文档就不再必要了呢？我们应该结对编程吗？我们应该在编写产品代码前先编写测试吗？我们应该做什么呢？

我是凑巧接触到这次革命的。在 20 世纪 90 年代的中后期，Object Mentor 公司在面向对象（OO）设计以及项目管理问题上帮助了许多公司。我们帮助这些公司完成它们的项目。作为帮助的一部分，

我们慢慢地向团队中灌输我们自己的一些看法和实践。糟糕的是，这些看法和实践没有被记录下来。它们只是一个从我们传递给我们的客户的口头规约。

到 1998 年，我认识到我们需要把我们的过程和实践写下来，这样我们就可以更好地把它们表达给我们的客户。于是，我在 *C++ Report*[①] 上撰写了许多关于过程的论文。这些文章都没有达到目的。它们提供了丰富的信息并且在某些情况下也挺引人入胜，但是它们不是对我们在项目中实际应用的实践和看法的整理，而是对影响我数十年的价值观念的一种不经意的放弃。Kent Beck 向我指出了这一点。

与 Kent Beck 的合作

1998 年晚期，当我正为整理 Object-Mentor 的过程烦恼时，我偶然接触到了 Kent 在极限编程（eXtreme Programming，简称 XP）方面的一些文字。这些文字散布在整个 Ward Cunningham 的 wiki[②] 中并且和其他一些人的文字混合在一起。尽管如此，通过一些努力和勤奋，我还是抓住了 Kent 所谈论的要点。这激起了我极大的兴趣，但是仍有一些疑虑。XP 中的某些东西和我的开发过程观念完全吻合，但是其他一些东西，比如：缺乏明确的设计阶段，却令我迷惑不解。

我和 Kent 来自完全不同的软件环境。他是一个公认的 Smalltalk 顾问，而我却是一个公认的 C++ 顾问。这两个领域之间很难相互交流。在它们之间几乎有一个孔恩式的（Kuhnian）[③]思维范式（paradigm）隔阂。

在其他情况下，我绝不会邀请 Kent 为 *C++ Report* 撰写论文。但是我们关于过程认识上的一致填补了语言上的隔阂。1999 年 2 月，我在慕尼黑的 OOP 会议上遇到了 Kent。他在进行关于 XP 的讲演，而我在进行面向对象设计原则的讲演，我们讲演的场所正好面对面。由于无法听到他的讲演，我就在午餐时找到了 Kent。我们谈论了 XP，并且我邀请他为 *C++ Report* 撰写一篇论文。这是一篇很棒的论文，其中描述了 Kent 和一位同事在 1 小时左右的现场系统开发中所进行的彻底的设计改变。

在接下来的几个月中，我经历了一段缓慢的对自己关于 XP 的担心进行分类的过程。我最大的担心在于所采用的过程中没有一个显式的预先设计阶段。我在这一点上有些犹豫。难道我没有义务去教导我的客户以及整个行业设计很值得在上面花费时间吗？

最后，我认识到实际上我自己也没有经历这样一个阶段。甚至在我所撰写的所有的关于设计、Booch 图和 UML 图的论文以及书籍中，我总是把代码作为验证这些图是否有意义的一种方式。在我所有的客户咨询中，我会先花费 1 到 2 个小时帮助他们绘制一些图，然后会使用代码来指导他们去考察这些图。我开始明白虽然 XP 关于设计的措词有点陌生（在 Kuhnian[④]的意义上），但是这些措词背后的实践对我来说却很熟悉。

我另外一个关于 XP 的担心非常容易处理。我私底下实际上一直是一个结对程序员。XP 使我可以光明正大地和同伴沉醉于一起编程的快乐之中。重构、持续集成以及现场客户对我来说都非常易

① 这些论文可以在 http://www.object.mentor.com 的“publications”部分找到。共有 4 篇。头 3 篇名为：“Iterative and Incremental Development（I, II, III）”。最后一篇名为：“C.O.D.E Culled Object Development procEss”。

② http://c2.com/cgi/wiki。这个网站中包含有数量众多的涉及各种各样主题的论文。它的作者的数目成百上千。据说，只有 Ward Cunningham 才能使用几行 Perl 煽动一场社会革命。

③ 写于 1995 至 2001 年之间的任何可信的学术作品中肯定使用了术语“Kuhnian”。它指的是 *The Structure of Scientific Revolutions* 一书，作者为 Thomax S. Kuhn，由芝加哥大学出版社出版于 1962 年。

④ 如果你在一篇文章中提到两次 Kuhn，就会得到额外的信任。

于接受。它们都非常接近于我先前建议我的客户的工作方式。

有一个 XP 实践对我来说是一个新发现。当你第一次听到测试优先设计时会觉得它似乎很平常。它指示要在编写任何产品代码前先编写测试用例。编写的所有产品代码都是为了让失败的测试用例通过。对于以这种方式编写代码所带来的意义深远的后果，我始料未及。这个实践完全改变了我编写软件的方法，并把它变得更好了。在本书中，你可以看到这个改变。本书中有些代码编写于 1999 年之前。这些代码都没有测试用例。但是，所有编写于 1999 年之后的代码都带有测试用例，并且测试用例一般都首先出现。我确信你会注意到它们之间的差别。

于是，到 1999 年秋天，我确信 Object Mentor 应该采用 XP 作为它的过程选择，并且我应该放弃编写自己的过程的愿望。Kent 在表达 XP 的实践和过程方面已经做了一项卓越的工作，相比起来我自己那不充分的尝试就显得苍白无力了。

本书组织

本书被组织成 6 大部分，其后跟有一些附录。

- 第 I 部分：敏捷开发。本部分描述了敏捷开发的概念。它先介绍了敏捷联盟宣言，然后提供了对极限编程（XP）的概述，接着讨论了许多阐明个别极限编程实践的小案例——特别是那些影响我们设计和编写代码方式的实践。
- 第 II 部分：敏捷设计。本部分中的章节谈论了面向对象软件设计。头一章中提出了问题：什么是设计？它讨论了管理复杂性的问题以及技术。最后，以面向对象类设计的一些原则作为本部分的结束。
- 第III部分：薪水支付案例研究。这是本书中最大的并且也是最完整的案例研究。它描述了一个简单的批量处理薪水支付系统的面向对象设计和 C++实现。本部分的头几章描述了该案例研究会用到的一些设计模式。最后两章包含了完整的案例研究。
- 第IV部分：打包薪水支付系统。本部分以描述面向对象包设计的一些原则作为开始。接着，它通过增量地打包上一部分中的类来继续阐明这些原则。
- 第 V 部分：气象站案例研究。本部分中包含了一个起初打算用于 Booch 的书的案例。气象站案例研究描述了一个公司做出了一项重要的商务决策，并且阐明了 Java 开发团队对此是如何做出反应的。同样，本部分开始时描述了一些会用到的设计模式，接着以对设计和实现的描述作为结束。
- 第VI部分：ETS 案例研究。本部分描述了作者参与的一个实际项目。这个项目自 1999 年起就已经产品化。它是一个自动考试系统，用来对美国注册建筑师委员会的注册考试进行答题和评分。
- UML 表示法附录。头两个附录包含了几个用来描述 UML 表示法的简单案例研究。
- 其他附录

如何使用本书

如果你是一个开发人员……

请从头至尾地阅读本书。本书主要是写给开发人员的，它包含以敏捷的方式开发软件所需要的信息。从头至尾阅读可以首先学习实践，接着是原则，然后是模式，最后是把它们全部联系起来的案例研究。把所有这些知识整合起来会帮助你完成项目。

如果你是一个管理人员或者业务分析师……

请阅读第 I 部分“敏捷开发”。这一部分中的章节提供了对敏捷原则和实践的深入讨论。内容涉及需求、计划、测试、重构以及编程。它会给你一些有关如何构建团队以及管理项目的指导，帮助你完成项目。

如果你想学习 UML……

请首先阅读附录 A“UML 表示法 I：CGI 示例”。接着阅读附录 B“UML 表示法 II：统计多路复用器”。然后，阅读第III部分“薪水支付案例研究”的所有章节。这种阅读方法在 UML 语法和使用方面会给你提供一个好的基础。它同时也会帮助你在 UML 和像 Java 或者 C++这样的编程语言之间进行转换。

如果你想学习设计模式……

要想找到一个特定的模式，可以使用“设计模式列表”找到你感兴趣的模式。

要想在总体上学习模式，请阅读第 II 部分“敏捷设计”学习设计原则，然后阅读第III部分“薪水支付案例研究”、第IV部分“打包薪水支付系统”、第 V 部分“气象站案例研究”以及第VI部分“ETS 案例研究”。这些部分定义了所有的模式，并且展示了如何在典型的情形中使用它们。

如果你想学习面向对象设计原则……

请阅读第 II 部分“敏捷设计”、第III部分“薪水支付案例研究”以及第IV部分“打包薪水支付系统”。这些章节将会描述面向对象设计的原则，并且向你展示如何去使用这些原则。

如果你想学习敏捷开发方法……

请阅读第 I 部分“敏捷开发”。这一部分描述了敏捷开发，内容涉及需求、计划、测试、重构以及编程。

如果你只想笑一笑……

请阅读附录 C“两个公司的讽刺小品”。

致　　谢

衷心感谢以下人士：

Lowell Lindstrom, Brian Button, Erik Meade, Mike Hill, Michael Feather, Jim Newkirk, Micah Martin, Angelique Thouvenin Martin, Susan Rosso, Talisha Jefferson, Ron Jeffries, Kent Beck, Jeff Langr, David Farber, Bob Koss, James Grenning, Lance Welter, Pascal Roy, Martin Fowler, John Goodsen, Alan Apt, Paul Hodgetts, Phil Markgraf, Pete McBreen, H.S.Lahman, Dave Harris, James Kanze, Mark Webster, Chris Biegay, Alan Francis, Fran Daniele, Patrick Lindner, Jake Warde, Amy Todd, Laura Steele, William Pietr, Camille, Trentacoste, Vince O'Brien, Gregory Dulles, Lynda Castillo, Craig Larman, Tim Ottinger, Chris Lopez, Phil Goodwin, Charles Toland, Robert Evans, John Roth, Debbie Utley, John Brewer, Russ Ruter, David Vydra Ian Smith, Eric Evans, 硅谷 Patterns Group 中的每一个人, Pete Brittingham, Graham Perkins, Philp, 以及 Richard MacDonald。

本书的审阅者为：

Pete McBreen/McBreen Consulting
Setphen J.Mellor/Projtech.com
Brian Button/Object Mentor Inc.
Bjarne Stroustrup/AT&T Research
MicahMartin/Objecct Mentor Inc.
James Grenning/Object Mentor Inc.

非常感谢 Grady Booch 和 Paul Becker 允许我在本书中包含原本用于 Grady 的 *Object Oriented Analysis and Design with Applications* 第 3 版中的那些章节。

特别感谢 Jack Reeves，他很有风度地允许我再版他的论文“什么是设计”。还要特别感谢 Erich Gamma 为本书做序。Erich 希望这次的字体会好一些！

每章开头处那美妙、偶尔还有些炫目的插图是 Jennifer Knohnke 绘制的。散布在章节中间的装饰插图是 Angela Dawn Marin Brooks 的可爱作品，她是我的女儿，也是我生活中的快乐之一。

资　　源

本书中的所有源代码都可以从 www.objectmentor.com/PPP 下载。

关于作者和参与者

Robert C.Martin

Robert C.Martin（Bob 大叔）自 1970 年起就是一个软件专家，并且自 1999 年起成为了国际性的软件顾问。他是 Object Mentor 公司的创始人和总裁，该公司拥有一个富有经验的顾问团队，在 C++、Java、.NET、面向对象、模式、UML、敏捷（Agile）方法以及极限编程（XP）领域为世界范围内的客户提供指导。1995 年，Robert 撰写了畅销书：*Designing Object Oriented C++ Applications using the Booch Method*，该书由 Prentice Hall 出版。在 1996－1999 年，他担任 *C++ Report* 的总编。1997 年，他担任由 Addison-Wesley 出版的 *Pattern Language of Program Design 3* 一书的主编。1999 年，他担任由 Cambridge 出版社出版的 *More C++ Gems* 一书的编辑。2001 年，他和 James Newkirk 合作撰写了由 Addison-Wesley 出版的 *XP in Practice* 一书。2002 年，他撰写了期待已久的《敏捷软件开发：原则、模式与实践》（*Agile Software Development: Principles, Patterns, and Practices*）一书，该书由 Prentice Hall 出版，并荣获 2002 年度（第 13 届）美国软件开发震憾（Jolt）大奖。他在各种行业杂志上发表了许多论文，并经常在国际性会议以及展览会上演讲。他是一个非常快乐的人。

James W. Newkirk

James W. Newkirk 是一个软件开发经理和构架师。他有 18 年的开发经验，涉及领域从实时微控制器编程到 web 服务。他是 *Extreme Programming in Practice* 一书的作者之一，该书由 Addison- Wesley 于 2001 年出版。自 2000 年 8 月以来，他一直使用.NET 框架进行工作，他在.NET 的一个单元测试工具 NUNIT 的开发方面做了不少的工作。

Robert S. Koss

Robert S.Koss 博士编写软件已有 29 年了。他把面向对象设计的原则应用到了许多项目中，在这些项目中，他担当过从程序员到资深构架师等许多角色。Koss 博士为全世界数千名学生教授过数百门面向对象设计和编程语言的课程。目前，他在 Object Mentor 公司担当资深顾问。

目　录

第Ⅰ部分　敏捷开发

第Ⅱ部分　敏捷设计

第III部分 薪水支付案例研究

第IV部分 打包薪水支付系统

第Ⅴ部分　气象站案例研究

第Ⅵ部分　ETS 案例研究

附　录

设计模式列表

第I部分 敏捷开发

人与人之间的交互是复杂的、并且其效果从来都难以预期，但却是工作中最为重要的方面。

——Tom DeMacro 和 Timothy Lister
《人件》，第 5 页

原则（principle）、模式（pattern）和实践（practice）都是重要的，但是使它们发挥作用的是人。正如 Alistair Cockburn 所说的[①]，“过程和方法对于项目的结果只有次要的影响。首要的影响是人。”

如果把程序员团队看做是由过程驱动的组件（component）所组成的系统，那么就无法对它们进行管理。人不是“插入即兼容的编程装置。”[②]如果想要项目取得成功，就必须构建起具有合作精神的、自组织（self-organizing）的团队。

那些鼓励构建这种团队的公司比那些认为软件团队不过是由无关紧要的、雷同的一群人堆砌的公司具有大得多的竞争优势。有凝聚力的团队将具有最强大的软件开发力量。

① 私人交谈。
② 此语出自 Kent Beck。

第1章 敏捷实践

教堂尖顶上的风标，即使由钢铁制成，如果不懂得顺应风势的艺术，一样会被暴风立即摧毁。

——海因里希·海涅（1797－1856，德国诗人）

许多人都经历过由于没有实践的指导而导致的项目噩梦。缺乏有效的实践会导致不可预测性、重复的错误以及努力的白白浪费。延期的进度、增加的预算和低劣的质量致使客户对我们丧失信心。更长时间的工作却生产出更加低劣的软件产品，也使得开发人员感到沮丧。

一旦经历了这样的惨败，就会害怕重蹈覆辙。这种恐惧激发我们创建一个过程来约束我们的活动、要求有某些人为制品（artifacts）输出。我们根据过去的经验来规定这些约束和输出，挑选那些在以前的项目中看起来好像工作得不错的方法。我们希望这些方法这次还会有效，从而消除我们的恐惧。

然而，项目并没有简单到使用一些约束和人为制品就能够可靠地防止错误的地步。当连续地犯错误时，我们会对错误进行诊断，并在过程中增加更多的约束和人为制品来防止以后重犯这样的错误。经过多次这样的增加以后，我们就会不堪巨大、笨重的过程的重负，极大地削弱我们完成工作的能力。

一个大而笨重的过程会产生它本来企图去解决的问题。它降低了团队的开发效率，使得进度延期，预算超支。它降低了团队的响应能力，使得团队经常创建错误的产品。遗憾的是，许多团队认为，这种结果是因为他们没有采用更多的过程方法引起的。因此，在这种失控的过程膨胀中，过程会变得越来越庞大。

用失控的过程膨胀来描述公元2000年前后的许多软件公司中的情形是很合适的。虽然有许多团队在工作中并没有使用过程方法，但是采用庞大、重型的过程方法的趋势却在快速地增长，在大公司中尤其如此（参见附录C）。

1.1　敏捷联盟

2001 年初，由于看到许多公司的软件团队陷入了不断增长的过程的泥潭，一批业界专家聚集在一起概括出了一些可以让软件开发团队具有快速工作、响应变化能力的价值观（value）和原则。他们称自己为敏捷（Agile）联盟。[1]在随后的几个月中，他们创建出了一份价值观声明。也就是敏捷联盟宣言（The Manifesto of the Agile Alliance）。

敏捷联盟宣言

敏捷软件开发宣言

我们正在通过亲身实践以及帮助他人实践，揭示更好的软件开发方法。通过这项工作，我们认为：

- 个体和交互　胜过　过程和工具
- 可以工作的软件　胜过　面面俱到的文档
- 客户合作　胜过　合同谈判
- 响应变化　胜过　遵循计划

虽然右项也有价值，但是我们认为左项具有更大的价值。

Kent Beck	*James Grenning*	*Robert C. Martin*
Mike Beedle	*Jim Highsmith*	*Steve Mellor*
Arie van Bennekum	*Andrew Hunt*	*Ken Schwaber*
Alistair Cockburn	*Ron Jeffries*	*Jeff Sutherland*
Ward Cunningham	*Jon Kern*	*Dave Thomas*
Martin Fowler	*Brian Marick*	

1. 个体和交互胜过过程和工具

人是获得成功的最为重要的因素。如果团队中没有优秀的成员，那么就是使用好的过程也不能从失败中挽救项目，但是，不好的过程却可以使最优秀的团队成员失去效用。如果不能作为一个团队进行工作，那么即使拥有一批优秀的成员也一样会惨败。

一个优秀的团队成员未必就是一个一流的程序员。一个优秀的团队成员可能是一个平均水平的程序员，但是却能够很好地和他人合作。合作、沟通以及交互能力要比单纯的编程能力更为重要。一个由平均水平程序员组成的团队，如果具有良好的沟通能力，将要比那些虽然拥有一批高水平程序员，但是成员之间却不能进行交流的团队更有可能获得成功。

合适的工具对于成功来说是非常重要的。像编译器、IDE、源代码控制系统等，对于团队的开发者正确地完成他们的工作是至关重要的。然而，工具的作用可能会被过分地夸大。使用过多的庞大、笨重的工具就像缺少工具一样，都是不好的。

我们的建议是从使用小工具开始，尝试一个工具，直到发现它无法适用时才去更换它。不是急

---

① agilealliance.org。

着去购买那些先进的、价格昂贵的源代码控制系统，相反先使用一个免费的系统直到能够证明该系统已经不再适用。在决定为团队购买最好的CASE工具许可证（license）前，先使用白板和方格纸，直到有足够的理由表明需要更多的功能。在决定使用庞大的、高性能的数据库系统前，先使用平面文件（flat file）。不要认为更大的、更好的工具可以自动地帮你做得更好。通常，它们造成的障碍要大于带来的帮助。

记住，团队的构建要比环境的构建重要得多。许多团队和管理者就犯了先构建环境，然后期望团队自动凝聚在一起的错误。相反，应该首先致力于构建团队，然后再让团队基于需要来配置环境。

2．可以工作的软件胜过面面俱到的文档

没有文档的软件是一种灾难。代码不是传达系统原理和结构的理想媒介。团队更需要编制易于阅读的文档，来对系统及其设计决策的依据进行描述。

然而，过多的文档比过少的文档更糟。编制众多的文档需要花费大量的时间，并且要使这些文档和代码保持同步，就要花费更多的时间。如果文档和代码之间失去同步，那么文档就会变成庞大的、复杂的谎言，会造成重大的误导。

对于团队来说，编写并维护一份系统原理和结构方面的文档将总是一个好主意，但是那份文档应该是短小的（short）并且主题突出的（salient）。“短小的”意思是说，最多有一二十页。“主题突出的”意思是说，应该仅论述系统的高层结构和概括的设计原理。

如果全部拥有的仅仅是一份简短的系统原理和结构方面的文档，那么如何来培训新的团队成员，使他们能够从事与系统相关的工作呢？我们会非常密切地和他们在一起工作。我们紧挨着他们坐下来帮助他们，把我们的知识传授给他们。我们通过近距离的培训和交互使他们成为团队的一部分。

在给新的团队成员传授知识方面，最好的两份文档是代码和团队。代码真实地表达了它所做的事情。虽然从代码中提取系统的原理和结构信息可能是困难的，但是代码是惟一没有二义性的信息源。在团队成员的头脑中，保存着时常变化的系统的脉络图（road map）。人和人之间的交互是把这份脉络图传授给他人的最快、最有效的方式。

许多团队因为注重文档而非软件，导致进度拖延。这常常是一个致命的缺陷。有一个叫做“Martin文档第一定律（Martin's first law of document）”的简单规则可以预防该缺陷的发生：

直到迫切需要并且意义重大时，才来编制文档。

3．客户合作胜过合同谈判

不能像订购日用品一样来订购软件。你不能够仅仅写下一份关于你想要的软件的描述，然后就让人在固定的时间内以固定的价格去开发它。所有用这种方式来对待软件项目的尝试都以失败而告终。有时，失败是惨重的。

告诉开发团队想要的东西，然后期望开发团队消失一段时间后就能够交付一个满足需要的系统来，这对于公司的管理者来说是具有诱惑力的。然而，这种操作模式将导致低劣的质量和失败。

成功的项目需要有序、频繁的客户反馈。不是依赖于合同或者关于工作的陈述，而是让软件的客户和开发团队密切地在一起工作，并尽量经常地提供反馈。

一个指明了需求、进度以及项目成本的合同存在根本上的缺陷。在大多数的情况下，合同中指

明的条款远在项目完成之前就变得没有意义。[1]那些为开发团队和客户的协同工作方式提供指导的合同才是最好的合同。

我在1994年为一个大型的、需要多年完成的、有50万行代码的项目达成的合同，可以作为一个成功合同的样例。作为开发团队的我们，每个月的报酬相对是比较低的。大部分的报酬要在我们交付了某些大的功能块后才支付。那些功能块没有在合同中详细地指明。合同中仅仅声称在一个功能块通过了客户的验收测试时才支付该功能块的报酬。那些验收测试的细节也没有在合同中指明。

在这个项目开发期间，我们和客户紧密地在一起工作。几乎每个周五，我们都会把软件提交给客户。到下一周的周一或者周二，客户会给我们一份关于软件的变更列表。我们会把这些变更放在一起排定优先级，然后把它们安排在随后几周的工作中。客户和我们如此紧密地在一起工作，以至于验收测试根本就不是问题。因为他们周复一周地观察着每个功能块的演进，所以他们知道何时这个功能块能够满足他们的需要。

这个项目的需求基本处于一个持续变化的状态。大的变更是很平常的。在这期间，也会出现整个功能块被减掉，而加进来另外一些功能块。然而，合同和项目都经受住了这些变更，并获得成功。成功的关键在于和客户之间真诚的协作，并且合同指导了这种协作，而不是试图去规定项目范围的细节和固定成本下的进度。

4．响应变化胜过遵循计划

响应变化的能力常常决定着一个软件项目的成败。当我们构建计划时，应该确保计划是灵活的并且易于适应商务和技术方面的变化。

计划不能考虑得过远。首先，商务环境很可能会变化，这会引起需求的变动。其次，一旦客户看到系统开始运作，他们很可能会改变需求。最后，即使我们熟悉需求，并且确信它们不会改变，我们仍然不能很好地估算出开发它们需要的时间。

对于一个缺乏经验的管理者来说，创建一张优美的PERT或者Gantt图并把他们贴到墙上是很有诱惑力的。他们也许觉得这张图赋予了他们控制整个项目的权力。他们能够跟踪单个人的任务，并在任务完成时将任务从图上去除。他们可以对实际完成的日期和计划完成的日期进行比较，并对出现的任何偏差做出反应。

实际上发生的是这张图的组织结构不再适用。当团队增加了对于系统的认识，当客户增加了对于需求的认识，图中的某些任务会变得可有可无。另外一些任务会被发现并增加到图中。简而言之，计划将会遭受形态（shape）上的改变，而不仅仅是日期上的改变。

较好的做计划的策略是：为下两周做详细的计划，为下三个月做粗略的计划，再以后就做极为粗糙的计划。我们应该清楚地知道下两周要完成的任务，粗略地了解一下以后三个月要实现的需求。至于系统一年后将要做什么，有一个模糊的想法就行了。

计划中这种逐渐降低的细致度，意味着我们仅仅对于迫切的任务才花费时间进行详细的计划。一旦制定了这个详细的计划，就很难进行改变，因为团队会根据这个计划启动工作并有了相应的投入。然而，由于计划仅仅支配了几周的时间，计划的其余部分仍然保持着灵活性。

---

① 有时是远在合同签署之前就变得没有意义。

1.2　原　　则

从上述的价值观中引出了下面的 12 条原则，它们是敏捷实践区别于重型过程的特征所在。

1．我们最优先要做的是通过尽早的、持续的交付有价值的软件来使客户满意。

MIT Sloan 管理评论杂志刊登过一篇论文，分析了对于公司构建高质量产品方面有帮助的软件开发实践。[①]该论文发现了很多对于最终系统质量有重要影响的实践。其中一个实践表明，尽早地交付具有部分功能的系统和系统质量之间具有很强的相关性。该论文指出，初期交付的系统中所包含的功能越少，最终交付的系统的质量就越高。

该论文的另一项发现是，以逐渐增加功能的方式经常性地交付系统和最终质量之间有非常强的相关性。交付得越频繁，最终产品的质量就越高。

敏捷实践会尽早地、经常地进行交付。我们努力在项目刚开始的几周内就交付一个具有基本功能的系统。然后，我们努力坚持每两周就交付一个功能渐增的系统。

如果客户认为目前的功能已经足够了，客户可以选择把这些系统加入到产品中。

或者，他们可以简单地选择再检查一遍已有的功能，并指出他们想要做的改变。

2．即使到了开发的后期，也欢迎改变需求。敏捷过程利用变化来为客户创造竞争优势。

这是一个关于态度的声明。敏捷过程的参与者不惧怕变化。他们认为改变需求是好的事情，因为那些改变意味着团队已经学到了很多如何满足市场需要的知识。

敏捷团队会非常努力地保持软件结构的灵活性，这样当需求变化时，对于系统造成的影响是最小的。 在本书的后面部分，我们会学习一些面向对象设计的原则和模式，这些内容会帮助我们维持这种灵活性。

3．经常性地交付可以工作的软件，交付的间隔可以从几周到几个月，交付的时间间隔越短越好。

我们交付可以工作的软件（working software），并且尽早地（项目刚开始很少的几周后）、经常性地（此后每隔很少的几周）交付它。我们不赞成交付大量的文档或者计划。我们认为那些不是真正要交付的东西。我们关注的目标是交付满足客户需要的软件。

4．在整个项目开发期间，业务人员和开发人员必须天天都在一起工作。

为了能够以敏捷的方式进行项目的开发，客户、开发人员以及涉众之间就必须要进行有意义的、频繁的交互。软件项目不像发射出去就能自动导航的武器，必须要对软件项目进行持续不断地引导。

5．围绕被激励起来的个人来构建项目。给他们提供所需要的环境和支持，并且信任他们能够完成工作。

在敏捷项目中，人被认为是项目取得成功的最重要的因素。所有其他的因素——过程、环境、管理等等——都被认为是次要的，并且当它们对于人有负面的影响时，就要对它们进行改变。

例如，如果办公环境对团队的工作造成阻碍，就必须对办公环境进行改变。如果某些过程步骤

---

① *Product-Development Practices That Work: How Internet Companies Build Software*，MIT Sloan Management Review，Winter 2001，Reprint number 4226。

对团队的工作造成阻碍，就必须对那些过程步骤进行改变。

6. 在团队内部，最具有效果并且富有效率的传递信息的方法，就是面对面的交谈。

在敏捷项目中，人们之间相互进行交谈。首要的沟通方式就是交谈。也许会编写文档，但是不会企图在文档中包含所有的项目信息。敏捷团队不需要书面的规范、书面的计划或者书面的设计。团队成员可以去编写文档，如果对于这些文档的需求是迫切并且意义重大的，但是文档不是默认的沟通方式。默认的沟通方式是交谈。

7. 工作的软件是首要的进度度量标准。

敏捷项目通过度量当前软件满足客户需求的数量来度量开发进度。它们不是根据所处的开发阶段、已经编写的文档的多少或者已经创建的基础结构（infrastructure）代码的数量来度量开发进度的。只有当30%的必须功能可以工作时，才可以确定进度完成了30%。

8. 敏捷过程提倡可持续的开发速度。责任人、开发者和用户应该能够保持一个长期的、恒定的开发速度。

敏捷项目不是50米短跑；而是马拉松长跑。团队不是以全速启动并试图在项目开发期间维持那个速度；相反，他们以快速但是可持续的速度行进。

跑得过快会导致团队精力耗尽、出现短期行为以致于崩溃。敏捷团队会测量他们自己的速度。他们不允许自己过于疲惫。他们不会借用明天的精力来在今天多完成一点工作。他们工作在一个可以使在整个项目开发期间保持最高质量标准的速度上。

9. 不断地关注优秀的技能和好的设计会增强敏捷能力。

高的产品质量是获取高的开发速度的关键。保持软件尽可能的简洁、健壮是快速开发软件的途径。因而，所有的敏捷团队成员都致力于只编写他们能够编写的最高质量的代码。他们不会制造混乱然后告诉自己等有更多的时间时再来清理它们。如果他们在今天制造了混乱，他们会在今天把混乱清理干净。

10. 简单——使未完成的工作最大化的艺术——是根本的。

敏捷团队不会试图去构建那些华而不实的系统，他们总是更愿意采用和目标一致的最简单的方法。他们并不看重对于明天会出现的问题的预测，也不会在今天就对那些问题进行防卫。相反，他们在今天以最高的质量完成最简单的工作，深信如果在明天发生了问题，也会很容易进行处理。

11. 最好的构架、需求和设计出自于自组织的团队。

敏捷团队是自组织的团队。任务不是从外部分配给单个团队成员，而是分配给整个团队，然后再由团队来确定完成任务的最好方法。

敏捷团队的成员共同来解决项目中所有方面的问题。每一个成员都具有项目中所有方面的参与权力。不存在单一的团队成员对系统构架、需求或者测试负责的情况。整个团队共同承担那些责任，每一个团队成员都能够影响它们。

12．每隔一定时间，团队会在如何才能更有效地工作方面进行反省，然后相应地对自己的行为进行调整。

敏捷团队会不断地对团队的组织方式、规则、规范、关系等进行调整。敏捷团队知道团队所处的环境在不断地变化，并且知道为了保持团队的敏捷性，就必须要随环境一起变化。

1.3　结　　论

每一位软件开发人员、每一个开发团队的职业目标，都是给他们的雇主和客户交付最大可能的价值。可是，我们的项目以令人沮丧的速度失败，或者未能交付任何价值。虽然在项目中采用过程方法是出于好意的，但是膨胀的过程方法对于我们的失败至少是应该负一些责任的。敏捷软件开发的原则和价值观构成了一个可以帮助团队打破过程膨胀循环的方法，这个方法关注的是可以达到团队目标的一些简单的技术。

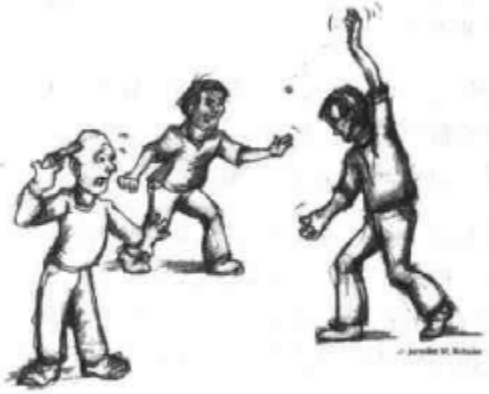
在撰写本书的时候，已经有许多的敏捷过程可供选择。包括：SCRUM①，Crystal②，特征驱动软件开发（Feature Driven Development，简称 FDD）③，自适应软件开发（Adaptive Software Development，简称 ADP）④，以及最重要的极限编程（eXtreme Programming，简称 XP）。⑤

参考文献

1. Beck, Kent. *Extreme Programming Explained: Embracing Change*. Reading, MA: Addison-Wesley, 1999.
2. Newkirk, James, and Robert C. Martin. *Extreme Programming in Practice*. Upper Saddle River, NJ: Addsion-Wesley, 2001.
3. Highsmith, James A. *Adapting Software Development: A Collaborative Approach to Managing Complex Systems*. New York, NY: Dorset House, 2000.

① www.controlchaos.com。
② crystalmethodologies.org。
③ *Java Modeling In Color With UML: Enterprise Components and Process*, Peter Coad, Eric Lefebvre, and Jeff De Luca, Prentice Hall, 1999。
④ [Highsmith2000]。
⑤ [Beck 1999], [Newkirk2001]。

第 2 章　极限编程概述



作为开发人员，我们应该记住，XP 并非惟一选择。

——Pete MaBreen

在上一章中，我们简要介绍了有关敏捷软件开发方法方面的内容，但它没有确切地告诉我们去做些什么；其中给出了一些泛泛的陈述和目标，却没有给出实际的指导方法。本章要改变这种状况。

2.1　极限编程实践

极限编程（eXtreme Programming，简称 XP）是敏捷方法中最著名的一个。它由一系列简单却互相依赖的实践组成。这些实践结合在一起形成了一个胜于部分结合的整体。本章我们将简要地探讨一下这个整体，在后续的章节中，会对一些单独的实践进行研究。

2.1.1　客户作为团队成员

我们希望客户和开发人员在一起紧密地工作，以便于彼此知晓对方所面临的问题，并共同去解决这些问题。

谁是客户？XP 团队中的客户是指定义产品的特性并排列这些特性优先级的人或者团体。有时，客户是和开发人员同属一家公司的一组业务分析师或者市场专家。有时，客户是用户团体委派的用户代表。有时，客户事实上是支付开发费用的人。但是在 XP 项目中，无论谁是客户，他们都是能够和团队一起工作的团队成员。

最好的情况是客户和开发人员在同一个房间中工作，次一点的情况是客户和开发人员之间的工作距离在 100 米以内。距离越大，客户就越难成为真正的团队成员。如果客户工作在另外一幢建筑或另外一个州，那么他将会很难融合到团队中来。

如果确实无法和客户在一起工作，该怎么办呢？我的建议是去寻找能够在一起工作、愿意并能够代替真正客户的人。

2.1.2 用户素材

为了进行项目计划，必须要知道和项目需求有关的内容，但是却无需知道得太多。对于做计划而言，了解需求只需要做到能够估算它的程度就足够了。你可能认为，为了对需求进行估算，就必须要了解该需求的所有细节，其实并非如此。你必须要知道存在很多细节，也必须要知道细节的大致分类，但是你不必知道特定的细节。

需求的特定细节很可能会随时间而改变，一旦客户开始看到集成到一起的系统，就更会如此。看到新系统的问世是关注需求的最好时刻。因此，在离真正实现需求还很早时就去捕获该需求的特定细节，很可能会导致做无用功以及对需求不成熟的关注

在 XP 中，我们和客户反复讨论，以获取对于需求细节的理解，但是不去捕获那些细节。我们更愿意客户在索引卡片上写下一些我们认可的词语（a few words），这些片言只语可以提醒我们记起这次交谈。基本上在和客户进行书写的同一时刻，开发人员在该卡片上写下对应于卡片上需求的估算。估算是基于和客户进行交谈期间所得到的对于细节的理解进行的。

用户素材（user stories）就是正在进行的关于需求谈话的助记符。它是一个计划工具，客户可以使用它并根据它的优先级和估算代价来安排实现该需求的时间。

2.1.3 短交付周期

XP 项目每两周交付一次可以工作的软件。每两周的迭代（iteration，也可称为重复周期或循环周期）都实现了涉众的一些需求。在每次迭代结束时，会给涉众演示迭代生成的系统，以得到他们的反馈。

1. 迭代计划

每次迭代通常耗时两周。这是一次较小的交付，可能会被加入到产品中，也可能不会。它由客户根据开发人员确定的预算而选择的一些用户素材组成。

开发人员通过度量在以前的迭代中所完成的工作量来为本次迭代设定预算。只要估算成本的总量不超过预算，客户就可以为本次迭代选择任意数量的用户素材。

一旦迭代开始，客户就同意不再修改当次迭代中用户素材的定义和优先级别。迭代期间，开发人员可以自由地将用户素材分解成任务（task），并依据最具技术和商务意义的顺序来开发这些任务。

2. 发布计划

XP 团队通常会创建一个计划来规划随后大约 6 次迭代的内容，这就是所谓的发布计划。一次发布通常需要 3 个月的工作。它表示了一次较大的交付，通常此次交付会被加入到产品中。发布计划是由一组客户根据开发人员给出的预算所选择的、排好优先级别的用户素材组成。

开发人员通过度量在以前的发布中所完成的工作量来为本次发布设定预算。只要估算成本的总量不超过预算，客户就可以为本次发布选择任意数目的用户素材。客户同样可以决定在本次发布中用户素材的实现顺序。如果开发人员强烈要求的话，客户可以通过指明哪些用户素材应该在哪次迭代中完成的方式，制订出发布中最初几次迭代的内容。

发布计划不是一成不变的，客户可以随时改变计划的内容。他可以取消用户素材，编写新的用户素材，或者改变用户素材的优先级别。

2.1.4　验收测试

可以以客户指定的验收测试（Acceptance Tests）的形式来捕获有关用户素材的细节。用户素材的验收测试是在就要实现该用户素材之前或实现该用户素材的同时进行编写的。

验收测试使用能够让它们自动并且反复运行的某种脚本语言编写，这些测试共同来验证系统按照客户指定的行为运转。

编写验收测试所使用的语言随着系统的增长、演化而增长、演化。客户可以召集开发人员开发一个简单的脚本系统，或者他们拥有一个可以开发脚本系统的独立的质量保证（QA）部门。许多客户借助于 QA 来开发验收测试工具，并自己编写验收测试。

一旦通过一项验收测试，就将该测试加入到已经通过的验收测试集合中，并决不允许该测试再次失败。这个不断增长的验收测试集合每天会被多次运行，每当系统被创建时，都要运行这个验收测试集。如果一项验收测试失败了，那么系统创建就宣告失败。因而，一项需求一旦被实现，就再不会遭到破坏。系统从一种工作状态变迁到另一种工作状态，期间，系统的不能工作状态时间决不允许超过几个小时。

2.1.5　结对编程

所有的产品（production）代码都是由结对的程序员使用同一台电脑共同完成的。结对人员中的一位控制键盘并输入代码，另一位观察输入的代码并寻找着代码中的错误和可以改进的地方。[①]两个人强烈地（intensely）进行着交互，他们都全身心地投入到软件的编写中。

两人频繁互换角色。控制键盘的可能累了或者遇到了困难，他的同伴会取得键盘的控制权。在一个小时内，键盘可能在他们之间来回传递好几次。最终生成的代码是由他们两人共同设计、共同编写的，两人功劳均等。

结对的关系每天至少要改变一次，以便于每个程序员在一天中可以在两个不同的结对中工作。在一次迭代期间，每个团队成员应该和所有其他的团队成员在一起工作过，并且他们应该参与了本次迭代中所涉及的每项工作。

这将极大地促进知识在团队中的传播。仍然会需要一些专业知识，并且那些需要一定专业知识的任务通常需要合适的专家去完成，但是那些专家几乎会和团队中的所有其他人结过对。这将加快专业知识在团队中的传播。这样，在紧要关头，其他团队成员就能够代替所需要的专家。

Laurie Williams[②]和 Nosek[③]的研究表明，结对非但不会降低开发团队的效率，而且会大大减少缺陷率。

---

① 我曾经见过这样的结对编程的情景，其中一位成员控制键盘，另一位成员控制鼠标。
② [Williams2000], [Cockburn2001]。
③ [Nosek]。

2.1.6　测试驱动的开发方法

本书第 4 章是关于测试方面的内容，其中详细地论述了测试驱动的开发方法。在下面的段落中，仅对此进行快速的浏览。

编写所有产品代码的目的都是为了使失败的单元测试能够通过。首先编写一个单元测试，由于它要测试的功能还不存在，所以它会运行失败。然后，编写代码使测试通过。

编写测试用例和代码之间的更迭速度是很快的，基本上几分钟左右。测试用例和代码共同演化，其中测试用例循序渐进地对代码的编写进行指导。（参见第 6 章的例子。）

作为结果，一个非常完整的测试用例集就和代码一起发展起来。程序员可以使用这些测试来检查程序是否正确工作。如果结对的程序员对代码进行了小的更改，那么他们可以运行测试，以确保更改没有对程序造成任何的破坏。这会非常有利于重构（在后面的章节中进行论述）。

当为了使测试用例通过而编写代码时，这样的代码就被定义为可测试的代码。这样做会强烈地激发你去解除各个模块间的耦合，这样能够独立地对它们进行测试。因而，以这种方式编写的代码的设计往往耦合性较弱。面向对象设计的原则在进行这种解除耦合方面具有巨大的帮助作用。[①]

2.1.7　集体所有权

结对编程中的每一对都具有拆出（check out）任何模块并对它进行改进的权力。没有程序员对任何一个特定的模块或技术单独负责。每个人都参与 GUI 方面的工作；[②]每个人都参与中间件方面的工作；每个人都参与数据库方面的工作。没有人比其他人在一个模块或者技术上具有更多的权威。

这并不意味着 XP 不需要专业知识。如果你的专业领域是有关 GUI 的，那么你最有可能去从事 GUI 方面的任务，但是你将会被邀请去和别人结对从事有关中间件和数据库方面的任务。如果你决定去学习另一门专业知识，那么你可以承担相关的任务，并和能够传授你这方面知识的专家一起工作。你不会被限制在自己的专业领域。

2.1.8　持续集成

程序员每天会多次拆入（check in）他们的代码并进行集成，规则很简单。第一个拆入的只要完成拆入就可以了，所有其他的人负责代码的合并（merge）工作。

XP 团队使用非阻塞的（nonblocking）源代码控制工具。这意味着程序员可以在任何时候拆出任何模块，而不管是否有其他人已经拆出这个模块。当程序员完成对模块的修改并把该模块拆入回去时，他必须要把他所做的改动和在他前面拆入该模块的程序员所做的任何改动进行合并。为了避免合并的时间过长，团队的成员会非常频繁地拆入他们的模块。

结对人员会在一项任务上工作 1～2 个小时。他们创建测试用例和产品代码。在某个适当的间歇点，也许远远在这项任务完成之前，他们决定把代码拆入回去。最重要的是要确保所有的测试都能够通过。他们把新的代码集成进代码库中。如果需要，他们会对代码进行合并。如果有必要，他们

① 参阅本书第 II 部分。
② 这里我不是在提倡 3 层构架。我只是选择了 3 种常见的软件技术。

会和先于他们拆入的程序员协商。一旦集成进了他们的更改，他们就构建新的系统。他们运行系统中的每一个测试，包括当前所有运行着的验收测试。如果他们破坏了原先可以工作的部分，他们会进行修正。一旦所有的测试都通过了，他们就算完成了此次拆入工作。

因而，XP 团队每天会进行多次系统构建，他们会重新创建整个系统。[①]如果系统的最终结果是一张 CD，他们就录制该 CD。如果系统的最终结果是一个可以访问的 Web 站点，他们就安装该 Web 站点，或许会把它安装在一个测试服务器上。

2.1.9　可持续的开发速度

软件项目不是全速的短跑，它是马拉松长跑。那些一跃过起跑线就开始尽力狂奔的团队在远离终点前就会筋疲力尽。为了快速地完成开发，团队必须要以一种可持续的速度前进。团队必须保持旺盛的精力和敏锐的警觉。团队必须要有意识地保持稳定、适中的速度。

XP 的规则是不允许团队加班工作。在版本发布前的一个星期是该规则的惟一例外。如果发布目标就在眼前并且能够一蹴而就，则允许加班。

2.1.10　开放的工作空间

团队在一个开放的房间中一起工作，房间中有一些桌子，每张桌子上摆放了两到三台工作站，每台工作站前有给结对编程的人员预备的两把椅子，墙壁上挂满了状态图表、任务明细表、UML 图等等。

房间里充满了交谈的嗡嗡声，结对编程的两人坐在互相能够听得到的距离内，每个人都可以得知另一人何时遇到了麻烦，每个人都了解对方的工作状态，程序员们都处在适合于激烈地进行讨论的位置上。

可能有人认为这种环境会分散人的注意力，很容易会让人担心由于持续的噪音和干扰而一事无成。事实上并非如此。而且，密歇根大学的一项研究表明，在“充满积极讨论的屋子（war room）”里工作，生产率非但不会降低，反而会成倍地提高。[②]

2.1.11　计划游戏

在下一章“计划”中，会详细地介绍 XP 的计划游戏方面的内容。在这里，先简要描述一下。

计划游戏（planning game）的本质是划分业务人员和开发人员之间的职责。业务人员（也就是客户）决定特性（feature）的重要性，开发人员决定实现一个特性所花费的代价。

在每次发布和每次迭代的开始，开发人员基于在最近一次迭代或者最近一次发布中他们所完成的工作量，为客户提供一个预算。客户选择那些所需的成本合计起来不超过该预算的用户素材。

依据这些简单的规则，采用短周期迭代和频繁的发布，很快客户和开发人员就会适应项目的开

① Ron Jeffries 讲到，“End to end is father than you think.”。
② http://www.sciencedaily.com/releases/2000/12/001206144705.htm。

发节奏。客户会了解开发人员的开发速度。基于这种了解，客户能够确定项目会持续多长时间，以及会花费多少成本。

2.1.12　简单的设计

XP 团队使他们的设计尽可能地简单、具有表现力（expressive）。此外，他们仅仅关注于计划在本次迭代中要完成的用户素材。他们不会考虑那些未来的用户素材。相反，在一次次的迭代中，他们不断变迁系统设计，使之对正在实现的用户素材而言始终保持在最优状态。

这意味着 XP 团队的工作可能不会从基础结构开始，他们可能并不先去选择使用数据库或者中间件。团队最开始的工作是以尽可能最简单的方式实现第一批用户素材。只有当出现一个用户素材迫切需要基础结构时，他们才会引入该基础结构。

下面三条 XP 指导原则（mantras）可以对开发人员进行指导。

1．考虑能够工作的最简单的事情

XP 团队总是尽可能寻找能实现当前用户素材的最简单的设计。在实现当前的用户素材时，如果能够使用平面文件，就不去使用数据库或者 EJB（企业级 Java Bean）；如果能够使用简单的 socket 连接，就不去使用 ORB（对象请求代理）或者 RMI（远程方法调用）；如果能够不使用多线程，就别去用它。我们尽量考虑用最简单的方法来实现当前的用户素材。然后，选择一种我们能够实际得到的和该简单方法最接近的解决方案。

2．你将不需要它

是的，但是我们知道总有一天会需要数据库，会需要 ORB，也总有一天得去支持多用户。所以，我们现在就需要为那些东西做好准备，不是吗？

如果在确实需要基础结构前拒绝引入它，那么会发生什么呢？XP 团队会对此进行认真的考虑。他们开始时假设将不需要那些基础结构。只有在有证据，或者至少有十分明显的迹像表明现在引入这些基础结构比继续等待更加合算时，团队才会引入这些基础结构。

3．一次，并且只有一次

极限编程者不能容忍重复的代码。无论在哪里发现重复的代码，他们都会消除它们。

导致代码重复的因素有许多，最明显的是用鼠标选中一段代码后四处粘贴。当发现那些重复的代码时，我们会通过定义一个函数或基类的方法来消除它们。有时两个或多个算法非常相似，但是它们之间又存在着微妙的差别，我们会把它们变成函数，或者使用 TEMPLATE METHOD 模式。[①]无论是哪一种代码重复之源，一旦发现，就必须被消除。

消除重复最好的方法就是抽象。毕竟，如果两种事物相似的话，必定存在某种抽象能够统一它们。这样，消除重复的行为会迫使团队提炼出许多的抽象，并进一步减少了代码间的耦合。

① 参见第 14 章，“Template Method 模式和 Strategy 模式：继承与委托”。

2.1.13　重构[1]

第 5 章会对重构（refactoring）进行详细的讨论，下面只是做简单的介绍。

代码往往会腐化。随着我们添加一个又一个的特性，处理一个又一个的错误，代码的结构会逐渐退化。如果对此置之不理的话，这种退化最终会导致纠结不清，难于维护的混乱代码。

XP 团队通过经常性的代码重构来扭转这种退化。重构就是在不改变代码行为的前提下，对其进行一系列小的改造（transformation），旨在改进系统结构的实践活动。每个改造都是微不足道的，几乎不值得去做。但是所有的这些改造叠加在一起，就形成了对系统设计和构架显著的改进。

在每次细微改造之后，我们运行单元测试以确保改造没有造成任何破坏，然后再去做下一次改造。如此往复，周而复始，每次改造之后都要运行测试。通过这种方式，我们可以在改造系统设计的同时，保持系统可以工作。

重构是持续进行的，而不是在项目结束时、发布版本时、迭代结束时、甚至每天快下班时才进行的。重构是我们每隔一个小时或者半个小时就要去做的事情。通过重构，我们可以持续地保持尽可能干净、简单并且具有表现力的代码。

2.1.14　隐喻

隐喻（metaphore）是所有 XP 实践中最难理解的一个。极限编程者在本质上都是务实主义者，隐喻这个缺乏具体定义的概念使我们觉得很不舒服。的确，一些 XP 的支持者经常讨论把隐喻从 XP 的实践中去除。然而，在某种意义上，隐喻却是 XP 所有实践中最重要的实践之一。

想像一下智力拼图玩具。你怎样知道如何把各个小块拼在一起？显然，每一块都与其他块相邻，并且它的形状必须与相邻的块完美地吻合。如果你无法看到但是具有很好的触觉，那么通过锲而不舍地筛选每个小块，不断地尝试它们的位置，也能够拼出整个图形。

但是，相对于各个小块的形状而言，还有一种更为强大的力量把这些复杂的小块拼装在一起。这就是整张拼图的图案。图案是真正的向导。它的力量是如此之大，以致于如果图案中相邻的两块不具有互相吻合的形状，你就能断定拼图玩具的制作者把玩具做错了。

这就是隐喻，它是将整个系统联系在一起的全局视图；它是系统的未来景像，是它使得所有单独模块的位置和外观（shape）变得明显直观。如果模块的外观与整个系统的隐喻不符，那么你就知道该模块是错误的。

隐喻通常可以归结为一个名字系统。这些名字提供了一个系统组成元素的词汇表，并且有助于定义它们之间关系。

例如，我曾经开发过一个以每秒 60 个字符的速度将文本输出到屏幕的系统。以这样的速度，字符充满整个屏幕需要一段时间。所以我们让产生文本的程序把产生的文本放到一个缓冲区中。当缓冲区满了的时候，我们把该程序交换到磁盘上。当缓冲区快要变空时，我们把该程序交换回来并让它继续运行。

---

[1] [Fowler99]。

我们用装卸卡车拖运垃圾来比喻整个系统。缓冲区是小卡车，屏幕是垃圾场，程序是垃圾制造者。所有的名字相互吻合，这有助于我们从整体上去考虑系统。

另举一例，我曾经开发过一个分析网络流量的系统。每 30 分钟，系统会轮询许多的网络适配器，并从中获取监控数据。每个网络适配器为我们提供一小块由几个单独变量组成的数据，我们称这些数据块为“面包切片”。这些面包切片是待分析的原始数据。分析程序“烤制”这些切片，因而被称为“烤面包机”。我们把数据块中的单个变量称为“面包屑”。总之，它是一个有用并且有趣的隐喻。

2.2 结　论

极限编程是一组简单、具体的实践，这些实践结合在一起形成了一个敏捷开发过程。该过程已经被许多团队使用过，并且取得了好的效果。极限编程是一种优良的、通用的软件开发方法。项目团队可以拿来直接采用，也可以增加一些实践，或者对其中的一些实践进行修改后再采用。

参考文献

1. Dahl, Dijkstra. *Structured Programming*. New York: Hoare, Academic Press, 1972.
2. Conner, Daryl R. *Leading at the Edge of Chaos*. Wiley, 1998.
3. Cockburn, Alistair. *The Methodology Space*. Humans and Technology technical report HaT TR.97.03（dated 97.10.03）, http://members.aol.com/acockburn/papers/methyspace/ methyspace.htm.
4. Beck, Kent. *Extreme Programming Explained: Embracing Change*. Reading, MA: Addison-Wesley, 1999.
5. Newkirk, James, and Robert C. Martin. *Extreme Programming in Practice*. Upper Saddle River, NJ: Addison-Wesley, 2000.
6. Williams, Laurie, Robert R. Kessler, Ward Cunningham, Ron Jeffries. *Strengthening the Case for Pair Programming*. IEEE Software, July—Aug. 2000.
7. Cockburn, Alistair, and Laurie Williams. *The Costs and Benefits of Pair Programming*. XP2000 Conference in Sardinia. Reproduced in *Extreme Programming Examined*, Giancarlo Succi, Michele Marchesi. Addison-Wesley, 2001.
8. Nosek, J.T. *The Case for Collaborative Programming*. Communications of the ACM（1998）:105-108.
9. Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

第3章 计 划

当你能够度量你所说的，并且能够用数字去表达它时，就表示你了解了它；若你不能度量它，不能用数字去表达它，那么说明你的知识就是匮乏的、不能令人满意的。

——凯尔文勋爵（英国物理学家），1883

下面的内容是对极限编程（XP）[①]中计划游戏（planning game）部分的描述。它和在其他敏捷[②]方法，如 SCRUM[③]、Crystal[④]、特征驱动开发方法（Feature-Driven Development，简称 FDD）[⑤]以及自适应软件开发（Adaptive Software Development，简称 ADP）[⑥]中做计划的方式相似。不过，那些过程方法都没有极限编程对此描述得详细、精确。

3.1 初始探索

在项目开始时，开发人员和客户会尽量确定出所有真正重要的用户素材。然而，他们不会试图去确定所有的用户素材。随着项目的进展，客户会不断编写新的用户素材。素材的编写会一直持续到项目完成。

开发人员共同对这些素材进行估算。估算是相对的，不是绝对的。我们在记录素材的卡片上写上一些“点数”来表示实现这个素材所需要的相对时间。我们可能不能确定一个“点”代表多少时间，但是我们知道实现 8 个点的素材所需要的时间是实现 4 个点的两倍。

探究、分解和速度

过大或者过小的素材都是难以估算的。开发人员往往会低估那些大的素材而高估那些小的素

① [Beck99], [Newkirk2001]。
② www.AgileAlliance.org。
③ www.controlchaos.com。
④ crystalmethodologes.org。
⑤ *Java Modeling In Color With UML: Enterprise Components and Process* by Peter Coad, Eric Lefebvre, and Jeff De Luca, Prentice Hall, 1999。
⑥ [Highsmith2000]。

材。任何过大的素材都应该被分解成小一点的部分，任何过小素材都应该和其他小的素材合并。

例如，考虑下面这个用户素材："用户能够安全地进行存款、取款、转账活动。"这是个大的素材。对它进行估算将会很困难，有可能还不准确。然而，我们可以把它分解成以下几个更容易估算的素材。

- 用户可以登录
- 用户可以退出
- 用户可以向其账户存款
- 用户可以从其账户取款
- 用户可以从其账户向其他账户转账

当分割或合并一个素材时，应该对其重新进行估算。简单地加上或者减去估算值是不明智的。对一个用户素材进行分解或者合并的主要原因，是为了使其大小适于被准确地估算。当一个估算为5点的素材被分解为几个点数总和达到10点的素材时，没什么令人惊讶的！10点是更精确的估算。

相对的估算没有给出户素材的绝对大小，因此无法帮助我们决定何时进行分解或者合并。为了知道用户素材的绝对大小，需要一个称为速度（velocity）的因子。如果知道了准确的速度，就可以将用户素材的估算点数乘以速度得到实现该用户素材的实际时间。例如，如果速度是"2天实现一个素材点"，那么相对估算是4点的用户素材，应该需要8天的实现时间。

随着项目的进展，由于可以度量每次迭代中已经完成的用户素材点数，所以对于速度的度量会越来越准确。然而，在开始时，开发人员可能对他们的速度没有很好的认识。开发人员必须要创建一个初始的猜测值，在创建这个猜测值时，可以采用他们感觉会带来最好结果的任何方式进行。此时对于准确性的需要不是非常急迫，所以他们无需在这上面花费过多的时间。通常，花费几天时间去原型化一到两个用户素材来了解团队的速度就足够了。这样的一个原型化过程称为探究（spike）。

3.2 发布计划

如果知道了开发速度，客户就能够对每个素材的成本有所了解。他们也知道每个素材的商业价值和优先级别。据此，他们就可以选择那些想要最先完成的素材。这种选择不是单纯依据优先级别进行的。一些重要的但是实现起来代价高昂的素材可能会被推迟实现，而会先去实现一些不那么重要的但是代价要低廉得多的素材。此类选择属于商务（business）决策范畴。让业务人员来选定那些会给他们带来最大利益的素材。

开发人员和客户对项目的首次发布时间达成一致，通常也就是2～4个月后的事情。客户挑选在该发布中他们想要实现的素材，并大致确定这些素材的实现顺序。客户不能选择与当前开发速度不符的更多的素材。由于开发速度开始时并不准确，所以选择也是粗略的。但是此时选择的准确性不是非常重要。当开发速度变得更准确一点时，可以再对发布计划进行调整。

3.3 迭代计划

开发人员和客户决定迭代规模，一般需两周。同样地，客户选择他们想要在首次迭代中实现的素材。他们不能选择与当前开发速度不符的更多的素材。

迭代期间用户素材的实现顺序属于技术决策范畴，开发人员采用最具技术意义的顺序来实现这些素材。他们可以串行地实现，完成了一个再完成下一个；或者他们分摊这些素材，然后一起并行地

开发。这完全取决于他们。

一旦迭代开始，客户就不能再改变该迭代期内需要实现的素材。除了开发人员正在实现的素材外，客户可以任意改变或重新安排项目中的其他任何素材。

即使没有完成所有的用户素材，迭代也要在先前指定的日期结束。他们会合计所有已经完成的素材的估算值，然后计算出本次迭代的开发速度。这个速度会被用于计划下一次的迭代。规则很简单：为每次迭代做计划时采用的开发速度就是前一次迭代中测算出来的开发速度。如果团队在最近一次迭代中完成了31个素材点，那么他们应该计划在下一次迭代中也完成31个点。他们的开发速度是每次迭代31个点。

这样的速度反馈有助于保持计划与团队实际状况相同步。如果团队在专业知识和工作技能方面有所提高，那么开发速度也会得到相应的提高。如果有人离开了团队，开发速度就会降低。如果系统构架朝有利于开发的方向演化，那么开发速度就会提高。

3.4 任务计划

在新的迭代开始时，开发人员和客户共同制定计划。开发人员把素材分解成开发任务，一个任务就是一个开发人员能够在4～16小时之内实现的一些功能。开发人员在客户的帮助下对这些素材进行分析，并尽可能完全地列举出所有的任务。

可以在活动挂图、白板和其他方便的媒介上列出这些任务。接着，开发人员逐个签订他们想要实现的任务。在开发人员签订一项任务的时候，会以随意的任务点数对那项任务进行估算。[①]

开发可以签订任意类型的任务。精通数据库的人员并非必须要签订数据库相关的任务。如果愿意，精通GUI的人员也可以签订数据库相关的任务。看起来好像无法人尽其能，但正如你将看到的，会有对这种状况进行管理的一种机制。这样做的好处是显而易见的。开发人员对整个项目了解得越多，那么团队就会越健康、越有知识。我们希望项目的知识能够传播给每一个团队成员，即便这种知识是和他们的专业无关的。

每个开发人员都知道在最近一次的迭代中所完成的任务点数，这个数字可以作为下一次迭代中的个人预算。没有人会签订超出他们预算的任务点数。

任务的选择一直到所有的任务都被分配出去，或者所有的开发人员都已经用完了他们的预算时为止。如果还有任务没有分配出去，那么开发人员会进行相互协商，基于各自的专长交换相应的任务。如果这样做都不能分配完所有任务，那么开发人员就要求客户从本次迭代中去掉一些任务或者素材。如果所有的任务都已经被分配，并且开发人员仍然具有预算空间去完成更多的任务，那么他们会向客户要求更多的素材。

迭代的中点

在迭代进行到一半的时候，团队会召开一次会议。在这个时间点上，本次迭代中所安排的半数素材应该被完成。如果没有完成，那么团队会设法重新分配没有完成的任务和职责，以保证在迭代结束时能够完成所有的素材。如果开发人员

① 许多开发人员发现使用“理想编程时间”作为他们的任务点数是有用的。

不能实现这样的重新分派，则需要告知客户。客户可以决定从迭代中去掉一个任务或素材。至少，客户可以指出那些最低优先级别的任务和素材，这样开发人员可以避免在其上花费时间。

例如，假设在本次迭代中客户选择了 8 个素材，总共 24 个素材点，同样假设这些素材被分解成 42 个任务。在迭代的中点，我们希望应该完成 21 个任务即 12 个素材点。这 12 个素材点代表的必须是全部被完成的素材。我们的目标是要完成素材，而不仅仅是任务。如果在迭代结束的时候，90% 的任务已被完成，但没有一个素材是被完全完成的，这将是恶梦一般的情景。在迭代的中点，我们希望看到拥有一半素材点数的完整的素材被完成。

3.5 迭　　代

每两周，本次迭代结束，下次迭代开始。在每次迭代结束时，会给客户演示当前可运行的程序。要求客户对项目程序的外观、感觉和性能进行评价。客户会以新的用户素材的方式提供反馈。

客户可以经常看到项目的进展，他们可以度量开发速度。他们可以预测团队工作的快慢，并且他们可以在早期安排实现高优先级别的素材。简而言之，他们拥有他们需要的所有数据和控制权，可以按照他们的意愿去管理项目。

3.6 结　　论

通过一次次的迭代和发布，项目进入了一种可以预测的、舒适的开发节奏。每个人都知道将要做什么，以及何时去做。涉众经常地、实实在在地看到项目的进展。他们看到的不是画满了图、写满了计划的记事本，而是可以接触到、感觉到的可以工作的软件，并且他们还可以对这个软件提供自己的反馈。

开发人员看到的是基于他们自己的估算并且由他们自己度量的开发速度控制的合理的计划。他们选择他们感觉舒适的任务，并保持高的工作质量。

管理人员从每次迭代中获取数据，他们使用这些数据来控制和管理项目。他们不必采用强制、威胁或者恳求开发人员忠心的方式去达到一个武断的、不切实际的目标。

这听起来好像是美好轻松的，其实不是这样。涉众对过程产生的数据并不总是满意的，特别是在刚刚开始时。使用敏捷方法并不意味着涉众就可以得到他们想要的。它只不过意味着他们将能够控制着团队以最小的代价获得最大的商业价值。

参考文献

1. Beck, Kent. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison- Wesley, 1999.
2. Newkirk, Jame, and Robert C. Martin. *Extreme Programming in Practice*. Upper Saddle River, NJ: Addison-Wesley, 2001.
3. Highsmith, Jmes A. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House, 2000.

第4章 测 试

烈火验真金，逆境磨意志。

——卢修斯•塞尼加（公元前4～公元65）

编写单元测试是一种验证行为，更是一种设计行为。同样，它更是一种编写文档的行为。编写单元测试避免了相当数量的反馈循环，尤其是功能验证方面的反馈循环。

4.1 测试驱动的开发方法

如果我们能够在设计程序前先设计测试方案，情况会怎么样？如果我们能够做到：除非缺乏某个功能将导致测试失败，否则就拒绝在程序中实现该功能，情况会怎么样？如果我们能做到：除非由于缺少某行代码将导致测试失败，否则就拒绝在程序中增加哪怕一行代码，情况又会怎样？如果首先编写失败的测试表明需要一项功能，然后再逐渐地增加那项功能使测试通过，情况又会怎么样？这对于我们正在编写的软件的设计有什么影响呢？如果存在这样一组包罗万象的测试，我们能够从中得到什么好处呢？

第一个也是最明显的一个影响，是程序中的每一项功能都有测试来验证它的操作的正确性。这个测试套件可以给以后的开发提供支援。无论何时我们因疏忽破坏了某些已有的功能，它就会告诉我们。我们可以向程序中增加功能，或者更改程序结构，而不用担心在这个过程中会破坏重要的东西。测试告诉我们程序仍然具有正确的行为。这样，我们就可以更自由地对程序进行改进。

还有一个更重要但是不那么明显的影响，是首先编写测试可以迫使我们使用不同的观察点。我们必须从程序调用者的有利视角去观察我们将要编写的程序。这样，我们就会在关注程序的功能的同时，直接关注它的接口。通过首先编写测试，我们就可以设计出便于调用的软件。

此外，通过首先编写测试，我们就迫使自己把程序设计为可测试的。把程序设计为易于调用和可测试的，是非常重要的。为了成为易于调用和可测试的，程序必须和它的周边环境解耦。这样，首先编写测试迫使我们解除软件中的耦合（forces us to decouple the software）。

首先编写测试的另一个重要效果，是测试可以作为一种无价的文档形式。如果想知道如何调用

一个函数或者创建一个对象，会有一个测试展示给你看。测试就像一套范例，它帮助其他程序员了解如何使用代码。这份文档是可编译、可运行的。它保持最新。它不会撒谎。

4.1.1 一个测试优先设计的示例

最近，我编写了一个名为“Hunt the Wumpus”的程序，仅仅是为了好玩。这是一个简单的冒险类游戏，玩家在洞穴中移动，设法在被 Wumpus 吃掉前杀掉 Wumpus。洞穴是由一系列通过过道互相连接的房间组成。每一个房间可以具有通向东、南、西、北方向的通道。玩家通过告诉计算机要行走的方向而四处移动。

在我为这个程序首先编写的测试中，有一个是程序 4.1 中的 testMove。这个函数创建了一个新的 WumpusGame 对象，通过一个东面的通道把房间 4 连接到房间 5，把玩家放置在房间 4 中，发出了向东移动的命令，接着断言玩家应该在房间 5 中。

程序 4.1

```
public void testMove()
{
    WumpusGame g = new WumpusGame();
    g.connect(4,5,"E");
    g.setPlayerRoom(4);
    g.east();
    assertEquals(5, g.getPlayerRoom());
}
```

这段测试代码是在编写 WumpusGame 程序前完成的。我采用了 Ward Cunningham 的建议，按照便于我们自己阅读的方式编写了这个测试。我相信只要按照测试所暗示的结构去编写 WumpusGame 程序，就能够通过测试。这种方法称为有意图的编程（intentional programming）。在实现之前，先在测试中陈述你的意图，使你的意图尽可能地简单、易读。你相信这种简单和清楚会给程序指出一个好的结构。

揭示意图编程（programming by intent）立即引导我产生了一个有趣的决定。测试代码中没有使用 Room 类。把一个房间连接到另一个房间的动作传达了我的意图。看起来，我不需要一个 Room 类来使表达更加容易。相反，我可以仅仅使用整数来表示房间。

这看起来好像不够直观。毕竟，这个游戏在你看来都是关于房间的，在房间之间移动，发现房间中包含的东西，等等。由于缺乏了一个 Room 类，我的意图所暗示的设计就有缺陷了吗？

我可以说在 Wumpus 游戏中，连接（connection）这个概念要比房间的概念重要得多。我可以说最初的测试指出了一个好的解决问题的方法。的确，我认为事情是这样的，但是那并不是我试图想让大家特别注意的。我想让大家特别注意的是测试在非常早的阶段就为我们阐明了一个重要的设计问题。首先编写测试的行为就是在各种设计决策中进行辨别的行为。

注意，测试告诉了我们程序是如何工作的。我们中的大多数都可以非常容易地根据这个简单的规格说明实现 WumpusGame 的 4 个已经命名了的方法。同样，命名并实现其他 3 个方向的命令也没有什么困难的。如果以后我们想知道如何把两个房间连接起来，或者如何朝一个特定的方向移动，这个测试会直截了当地展示给我们该如何去做。

4.1.2　测试促使模块之间隔离

在编写产品代码之前，先编写测试常常会暴露程序中应该被解耦合的区域。例如，图 4.1 展示了一个薪水支付应用（payroll application）的简单的 UML 图。[①]类 Payroll 使用 EmployeeDatabase 获得 Employee 对象，它要求 Employee 去计算自己的薪水。接着，它把计算结果传递给 CheckWriter 对象产生出一张支票。最后，它在 Empolyee 对象中记录下支付信息，并把 Empolyee 对象写回到数据库中。

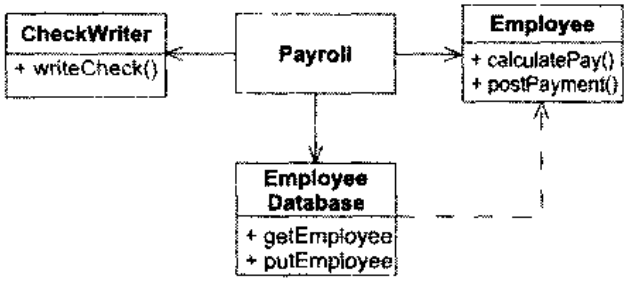


图 4.1　耦合在一起的薪水支付应用模型

假定还没有编写任何代码。这个图也是在经历了快速的设计探讨后，刚刚才画在白板上的。[②]现在，需要编写规定 Payroll 对象行为的测试。有许多和编写这些测试相关的问题。首先，要使用什么数据库呢？Payroll 对象需要从若干种类的数据库中读取数据。我们必须要在能够对 Payroll 类进行测试前，编写一个功能完善的数据库吗？我们要把什么数据加载到数据库中呢？其次，我们如何来验证打印出来的支票的正确性？我们无法编写出一个能够观察打印机打印出来的支票并验证上面的数额正确性的自动测试程序来！

使用 MOCK OBJECT 模式[③]可以解决这些问题。我们可以在 Payroll 类以及它的所有协作者之间插入接口，创建实现这些接口的测试桩（test stub）。

图 4.2 展示了这个结构。现在，Payroll 类使用接口和 EmployeeDatabase、CheckWriter 以及 Emplyee 进行通信，创建了 3 个实现这些接口的 MOCK OBJECTS。PayrollTest 对象对这些 MOCK OBJECTS 进行查询，来检验 Payroll 对象是否正确地对它们进行了管理。

程序 4.2 展示了测试的意图。测试中创建了合适的 MOCK OBJECTS，把它们传递给 Payroll 对象，告诉 Payroll 对象为所有雇员支付薪水，接着要求 MOCK OBJECTS 去验证所有已开支票的正确性以及所有已记录支付信息的正确性。

当然，这个测试所检查的都是 Payroll 应该使用正确的数据调用正确的函数。它既没有真正地去检查支票的打印，也没有真正地去检查一个真实数据库的正确刷新。相反，它检查了 Payroll 类应该具有与它在孤立情况下同样的行为。

你也许想知道为何需要 MockEmployee 类。看起来好像可以直接使用真实的 Employee 类。如果真是那样，我会毫不在乎的使用它。在本例中，我认为对于检查 Payroll 类的功能来说，Employee 类显得复杂了点。

---

① 如果对 UML 不了解的话，可以参见附录 A 和附录 B，其中对此进行了详细的描述。
② 参见[Jeffries2001]。
③ [Mackinnon2000]。

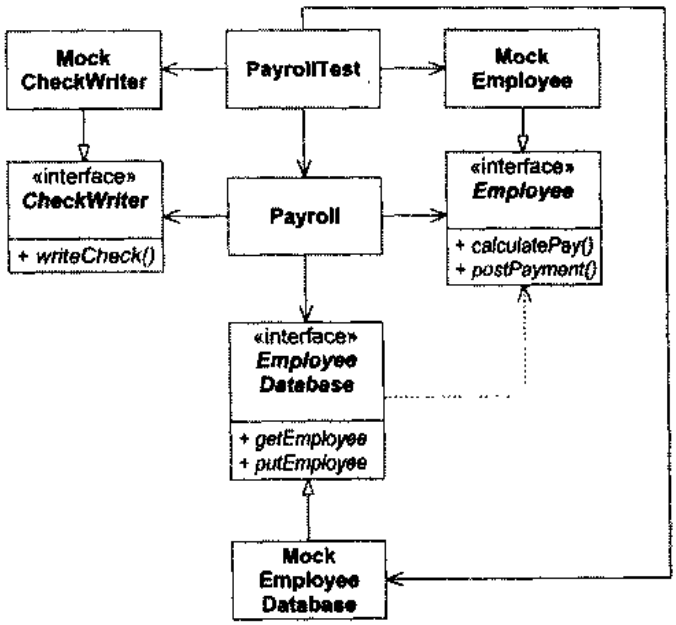


图 4.2　使用 Mock Objects 测试方法，解除了耦合的薪水支付应用模型

程序 4.2

```
public void testPayroll()
{
    MockEmployeeDatabase db = new MockEmployeeDatabase();
    MockCheckWriter w = new MockCheckWriter();
    Payroll p = new Payroll(db, w);
    p.payEmployees();
    assert(w.checksWereWrittenCorrectly());
    assert(db.paymentsWerePostedCorrectly());
}
```

4.1.3　意外获得的解耦合

对于 Payroll 类的解耦合是件好事。这允许我们可以互换使用不同种类的数据库和支票打印机，这种互换能力既是为了测试，也是为了应用的扩展性。我觉得为了进行测试而进行解耦合是有趣的。显然，为了测试而对模块进行隔离的需要，迫使我们以对整个程序结构都有益的方式对程序进行解耦合。在编写代码前先编写测试改善了设计。

本书中的大部分内容是依赖性管理方面的设计原则。这些原则在解耦合类和包方面提供了一些指导和技术。如果把这些原则作为单元测试策略的一部分来实践它们，就会发现这些原则是非常有用的。正是单元测试在解耦合方面提供了很多的推动和指导。

4.2　验收测试

作为验证工具来说，单元测试是必要的，但是不够充分。单元测试用来验证系统的小的组成单元应该按照所期望的方式工作，但是它们没有验证系统作为一个整体时工作的正确性。单元测试是用来验证系统中

个别机制的白盒测试（white-box tests）。[①]验收测试是用来验证系统满足客户需求的黑盒测试（black-box tests）。[②]

验收测试由不了解系统内部机制的人编写。客户可以直接或者和一些技术人员（可能是QA人员）一起来编写验收测试。验收测试是程序，因此是可以运行的。然而，通常使用专为应用程序的客户创建的脚本语言来编写验收测试。

验收测试是关于一项特性（feature）的最终的文档。一旦客户编写完成了验证一项特性的验收测试，程序员就可以阅读那些验收测试来真正地理解这项特性。所以，正如单元测试作为可编译、运行的有关系统内部结构的文档那样，验收测试是有关系统特性的可编译、执行的文档。

此外，首先编写验收测试的行为对于系统的构架方面具有深远的影响。为了使系统具有可测试性，就必须要在很高的系统构架层面对系统进行解耦合。例如，为了使验收测试无需通过用户界面（UI）就能够获得对于业务规则的访问，就必须要以满足这个目的的方式来解除用户界面和业务规则之间的耦合。

在项目迭代的初期，会受到用手工的方式进行验收测试的诱惑。但是，这样做使得在迭代的初期就丧失了由自动化验收测试的需要带来的对系统进行解耦合的促进力，所以是不明智的。当在最早开始迭代时，如果非常清楚地知道必须要自动化验收测试，就会做出非常不同的系统构架方面的权衡。并且，正如单元测试可以促使你在小的方面做出优良的设计决策一样，验收测试可以促使你在大的方面做出优良的系统构架决策。

创建一个验收测试框架（framework）看起来是件困难的任务。然而，如果仅仅创建框架中对单个迭代包含的特性进行验收测试所需要的那部分，就会发现并不困难。你还会发现所花费的努力是值得的。

4.2.1 验收测试示例

再次考虑一下薪水支付应用程序。在首次迭代中，必须能够向数据库中增加和删除雇员。必须能够为当前存在数据库中的雇员创建支付薪水的支票。还好，此次只需处理带薪雇员（salaried employees）。其他种类的雇员可以放在后面的迭代中处理。

我们还没有编写任何代码，也没有进行任何设计。这是开始考虑验收测试的最好时机，揭示意图编程再一次成为有用的工具。我们应该以我们认为验收测试应该的样子去编写它们，然后可以构造脚本语言，并根据脚本语言的结构来构造薪水支付系统。

我想使验收测试便于编写并且易于改变。我想把它们放置在一个配置管理工具中，并且把它们保存起来以便于随时可以运行它们。因此，采用简单的文本文件来编写验收测试应该是比较合理的。

下面是一段验收测试脚本的例子：

```
AddEmp 1429 "Robert Martin" 3215.88
Payday
Verify Paycheck EmpId 1429 Grosspay 3215.88
```

① 了解并依赖于被测模块内部结构的测试。
② 不了解并且不依赖于被测模块内部结构的测试。

在这个例子中，我们把雇员号为 1429 的雇员存入数据库。他的名字叫“Robert Martin”，他每月的薪水是 3215.88 美元。接着，我们告诉系统发薪日到了，该给所有的雇员发放薪水了。最后，我们核实产生了一张雇员号为 1429，Grosspay 域的值为 3215.88 的支票。

很明显，客户可以非常容易地编写这种脚本。同样，也非常容易在这种脚本中增加新的功能。然而，我们要考虑一下它所暗示的系统结构。

脚本的头两行针对的是薪水支付应用的功能。我们可以称这些行为薪水支付操作，这是薪水支付应用程序的使用者期望的功能。然而，Verify 的那一行并不是薪水支付应用程序的使用者期望的操作。这一行是专门针对验收测试的指令。

这样，验收测试框架必须要解析这个文本文件，分离开薪水支付操作和验收测试指令。它必须把薪水支付操作发送给薪水支付应用程序，接着为了对结果数据进行验证，它使用验收测试指令来对薪水支付应用程序进行查询。

这已经把重点放到了薪水支付程序的构架上面。薪水支付程序将必须接受直接来自使用者的输入，并且也要接受来自验收测试框架的输入。我们想把这两条输入的途径尽早地合并。因此，看起来薪水支付程序好像应该需要一个操作处理器，来处理从多个输入源到来的形如 AddEmp 和 Payday 的操作。我们需要为那些操作的描述找到共同的形式，以使得专用的代码数量保持最少。

一种解决方案是使用 XML 来表示输入给薪水支付应用的操作。验收测试框架当然可以产生 XML 格式的输出，并且薪水支付系统的 UI 好像也可以产生 XML 格式的输出。这样，我们可以看到如下所示的操作表示：

```
<AddEmp PayType=Salaried>
  <EmpId>1429</EmpId>
  <Name>Robert Martin</Name>
  <Salary>3215.88</Salary>
</AddEmp>
```

这些操作可以通过子程序调用、套接字、甚至批处理输入文件的方式进入薪水支付应用程序。在开发的过程中，从一种方式改变到另一种方式是一项简单的工作。因此，在初期迭代期间，我们可以采用从文件读入操作的方法，以后再把它改为 API 方式或者套接字方式。

验收测试框架如何调用 Verify 指令呢？很明显，它必须使用某些方法来访问由薪水支付应用程序所产生的数据。同样，我们想让验收测试框架不必从已经打印出来的支票上读取数据，我们有更好的方法。

我们可以让薪水支付应用程序以 XML 的形式产生它的支付支票。验收测试框架可以获取这个 XML 描述，并向它询问适当的数据。最后一步是要把以 XML 形式描述的支票打印出来，这是微不足道的一项工作，足以手工地进行验收。

这样，薪水支付应用程序可以创建包含所有支付支票信息的 XML 文档。看上去可能像这样：

```
<Paycheck>
  <EmpId>1429</EmpId>
  <Name>Robert Martin</Name>
  <Grosspay>3215.88</Grosspay>
</Paycheck>
```

很明显，当给验收测试框架提供这个 XML 文档时，它就能够执行 Verify 指令了。

同样地，可以通过套接字、API 的方式传递这个 XML 文档，或者把它存放到文件中。对于最初的迭代来说，文件是最简单的方式。因此，我们以最简单的方式开始薪水支付应用程序的开发，它从一个文件读入 XML 形式描述的操作，并且以 XML 的形式把支付支票输出到一个文件中。验收测试框架会读取文本形式的操作，把它们转换为 XML 的形式并写入一个文件。接着它会调用薪水支付应用程序。最后，它读取薪水支付应用程序输出的 XML 数据，并调用 Verify 指令。

4.2.2 意外获得的构架

注意验收测试对于薪水支付系统构架的影响。关于首先编写测试的一个绝对的事实是很快就导致我们有了使用 XML 来描述输入和输出的想法。这个构架把操作的来源和薪水支付应用本身解耦。同时，它也把支付支票打印机制和薪水支付应用本身解耦。这些是好的构架决策。

4.3 结 论

测试套件运行起来越简单，就会越频繁地运行它们。测试运行得越多，就会越快地发现和那些测试的任何背离。如果能够一天多次地运行所有的测试，那么系统失效的时间就决不会超过几分钟。这是一个合理的目标。我们决不允许系统倒退。一旦它工作在一个确定的级别上，就决不能让它倒退到一个稍低的级别。

然而，验证仅仅是编写测试的好处之一。单元测试和验收测试都是一种文档形式，那样的文档是可以编译和执行的；因此，它是准确和可靠的。此外，编写测试所使用的语言是明确的，并且它们的观看者使这些语言非常易读。程序员能够阅读单元测试，因为单元测试是使用程序员编程的语言编写的。客户能够阅读验收测试，因为验收测试是使用客户自己设计的语言编写的。

也许，测试最重要的好处就是它对于构架和设计的影响。为了使一个模块或者应用程序具有可测试性，必须要对它进行解耦合。越是具有可测试性，耦合关系就越弱。全面地考虑验收测试和单元测试的行为对于软件的结构具有深远的正面影响。

参考文献

1. Mackinnon, Tim, Steve Freeman, and Philip Craig. Endo-Testing: Unit Testing With Mock Objects. *Extreme Programming Examined.* Addison-Wesley, 2001.
2. Jeffries, Ron, et al. *Extreme Programming Installed.* Upper Saddle River, NJ: Addsion- Wesley, 2001.

第5章 重 构

大千世界中，惟一缺乏的就是人类的注意力。

——凯文·凯利（“新经济”的首席预言家之一），连线杂志

本章讲述的是关于人的注意力的。阐述人们应该专注于手边的工作并且确信自己正在尽全力，说明了使事物能够工作和使事物正确之间的区别，介绍了我们放入代码结构中的价值。

在 Martin Fowler 的名著《重构》一书中，他把重构（Refactoring）定义为：“……在不改变代码外在行为的前提下对代码做出修改，以改进代码的内部结构的过程。”[①]可是我们为什么要改进已经能够工作的代码的结构呢？不是还有句古老的谚语，“如果它没有坏，就不要去修理它！”吗？

每一个软件模块都具有三项职责。第一个职责是它运行起来所完成的功能。这也是该模块得以存在的原因。第二个职责是它要应对变化。几乎所有的模块在它们的生命周期中都要变化，开发者有责任保证这种改变应该尽可能地简单。一个难以改变的模块是拙劣的，即使能够工作，也需要对它进行修正。第三个职责是要和阅读它的人进行沟通。对该模块不熟悉的开发人员应该能够比较容易地阅读并理解它。一个无法进行沟通的模块也是拙劣的，同样需要对它进行修正。

怎样才能让软件模块易于阅读、易于修改呢？本书的主要内容都是关于一些原则和模式的，使用这些原则和模式可以帮助你创建更加灵活和具有适应性的软件模块。然而，要使软件模块易于阅读和修改，所需要的不仅仅是一些原则和模式。还需要你的注意力，需要纪律约束，需要创造美的激情。

① [Fowler99], p.xvi。

5.1 素数产生程序：一个简单的重构示例[①]

观察程序 5.1 中所示的代码，这个程序会产生素数。它是一个大函数，其中有辅助阅读的注释和很多单字母变量。

程序 5.1 GeneratePrimes.java，版本 1

```
/**
 * This class Generates prime numbers up to a user specified maximum.
 * the algorithm used is the Sieve of Eratosthenes.
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
 * d. c. 194, Alexandria. The first man to calculate the circumference
 * of the Earth. Also known for working on calendars with leap
 * years and ran the library at Alexandria.
 * <p>
 * The algorithm is quite simple. Given an array of integers starting
 * at 2. Cross out all multiples of 2. Find the next uncrossed
 * integer, and cross out all of its multiples. Repeat until
 * you have passed the square root of the maximum value.
 *
 * @author Robert C. Martin
 * @version 9 Dec 1999 rcm
 */
import java.util.*;

public class GeneratePrimes
{
  /**
   * @param maxValue is the generation limit.
   */
  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue >= 2) // the only valid case
    {
      // declarations
      int s = maxValue + 1; // size of array
      boolean[] f = new boolean[s];
      int i;

      // initialize array to true.
      for (i = 0; i < s; i++)
        f[i] = true;

      // get rid of known non-primes
      f[0] = f[1] = false;

      // sieve
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++)
      {
        for (j = 2 * i; j < s; j += i)
          f[j] = false; // multiple is not prime
      }
```

① 这个程序最初是在一个 XP Immersion 中编写的，使用的是由 Jim Newkirk 编写的测试。Kent Beck 和 Jim Newkirk 在学员面前对它进行重构。在这里，我尽量再现那个重构。

```
    // how many primes are there?
    int count = 0;
    for (i = 0; i < s; i++)
    {
      if (f[i])
        count++; // bump count.
    }

    int[] primes = new int[count];

    // move the primes into the result
    for (i = 0, j = 0; i < s; i++)
    {
      if (f[i])            // if prime
        primes[j++] = i;
    }

    return primes;  // return the primes
  }
  else // maxValue < 2
    return new int[0]; // return null array if bad input.
 }
}
```

为 GeneratePrimes 编写的单元测试可以参见程序 5.2。它采用了一种统计学的方法，主要检查产生器能否产生 0、2、3 以及 100 以内的素数。在第一种情况下，应该没有素数；在第二种情况下，应该有一个素数，并且该素数应该是 2；在第三种情况下，应该有两个素数，它们应该是 2 和 3。在最后一种情况下，应该有 25 个素数，其中最后一个是 97。如果所有这些测试都通过了，那么就认为产生器是可以工作的。我怀疑这种做法的可靠性，但是我不能想像出一个合理的情况，在这个情况下这些测试都将通过但是函数却是错误的。

程序 5.2 TestGeneratePrimes.java

```
import junit.framework.*;

public class TestGeneratePrimes extends TestCase
{
  public static void main(String args[])
  {
      junit.swingui.TestRunner.main(
        new String[] {"TestGeneratePrimes"});
  }
  public TestGeneratePrimes(String name)
  {
    super(name);
  }

  public void testPrimes()
  {
    int[] nullArray = PrimeGenerator.generatePrimes(0);
    assertEquals(nullArray.length, 0);

    int[] minArray = PrimeGenerator.generatePrimes(2);
    assertEquals(minArray.length, 1);
    assertEquals(minArray[0], 2);

    int[] threeArray = PrimeGenerator.generatePrimes(3);
```

```
    assertEquals(threeArray.length, 2);
    assertEquals(threeArray[0], 2);
    assertEquals(threeArray[1], 3);

    int[] centArray = PrimeGenerator.generatePrimes(100);
    assertEquals(centArray.length, 25);
    assertEquals(centArray[24], 97);
  }
```

在重构这个程序时，我使用了 IntelliJ Idea（一个重构浏览器）。使用该工具可以非常容易地进行提取方法以及重新命名变量和类等常用重构手法。

把全部功能变成 3 个分离的功能似乎是非常清楚的了。第一个功能是对所有的变量进行初始化，并做好过滤所需的准备工作；第二个功能执行真正的过滤工作；第三个功能把过滤后的结果存放到一个整型数组中。为了在程序 5.3 中更清晰地展现这个结构，我把那些功能提取出来放在 3 个分离的方法中。另外，去掉了一些不必要的注释，并且把类名更改为 PrimeGenerator。更改后的代码仍然通过所有测试。

在提取这些功能时，迫使我把一些函数级的局部变量提升为类级的静态域。我认为这澄清了局部变量和作用范围更大一点的变量的区别。

程序 5.3 PrimeGenerator.java，版本 2

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum.  The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
 * multiples.  Repeat until there are no more multiples
 * in the array.
 */

import java.util.*;

public class PrimeGenerator
{
    private static int a;
    private static boolean[] f;
    private static int[] primes;

    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            initializeSieve(maxValue);
            sieve();
            loadPrimes();
            return primes;      // return the primes
        }
    }

    private static void loadPrimes()
    {
        int i;
        int j;
```

```
        // how many primes are there?
        int count = 0;
        for (i = 0; i < s; i++)
        {
            if (f[i])
                count++;        // bump count.
        }

        primes = new int[count];

        // move the primes into the result
        for (i=0, j=0; i < s; i++)
        {
            if (f[i])           // if prime
                prime[j++] = i;
        }
    }

    private static void sieve()
    {
        int i;
        int j;
        for (i=2; i < Math.sqrt(s) + 1; i++)
        {
            if (f[i])       // if i is uncrossed, cross out its multiples.
            {
                for (j=2*i; j < s; j += i);
                    f[j] = false;   // multiple is not prime
            }
        }
    }

    private static void initializeSieve(int maxValue)
    {
        // declarations;
        s = maxValue + 1;   // size of array
        f = new boolean[s];
        int i;

        // initialize array to true.
        for (i = 0; i < s; i++)
            f[i] = true;

        // get rid of known non-primes
        f[0] = f[1] = false;

    }
}
```

initializeSieve 函数有一些凌乱，所以在程序 5.4 中对它进行了相当大的整理。首先把所有使用变量 s 的地方替换为 f.length。然后，更改了 3 个函数的名字，使它们更具表达力。最后，重新安排了 initializeArrayOfIntegers（也就是原先的 initializeSieve）的内部结构，使它更易于阅读。更改后的代码仍然通过所有测试。

程序 5.4 PrimerGenerator.java，版本 3（部分）

```
public class PrimeGenerator
{
```

```
    private static boolean[] f;
    private static int[] result;

    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            initializeArrayOfIntegers(maxValue);
            crossOutMultiples();
            putUncrossedIntegerIntoResult();
            return result;
        }
    }

    private static void initializeArrayOfIntegers(int maxValue)
    {
        f = new boolean[maxValue + 1];
        f[0] = f[1] = false;
        for (i = 2; i < f.length; i++)
            f[i] = true;
    }
}
```

下一步，来看看 crossOutMultiples，这个函数和其他一些函数中有许多形如 if(f[i] == true)的语句。这条语句的意图是检查 i 是否没有被筛选过，所以把 f 改名为 unCrossed。但是改名后产生了像 unCrossed[i] = false 这样难看的语句。我发现双重否定是令人迷惑的。所以把数组名更改为 isCrossed，并且更改了所有布尔值的含意。更改后的代码仍然通过所有测试。

我去掉了设置 isCrossed[0]和 isCrossed[1]为 true 的初始化语句，并确认函数中的所有部分都不会通过小于 2 的索引使用 isCrossed 数组。我提取出了 crossOutMultiples 的内部循环部分，并把它命名为 crossOutMultipleOf。同样，我觉得 if(isCrossed[i] == false)是令人迷惑的，所以创建了一个名为 notCrossed 的函数，把原来的 if 语句更改为 if(notCrossed(i))。更改后的代码仍然通过所有测试。

我用了一小段的时间来写注释，这个注释试图解释为何只需遍历至数组长度的平方根。这引导我把计算部分提取出来放到一个独立的函数中，其中可以放置说明性的注释。在书写注释时，我认识到这个平方根是数组中任意整数的最大素因子。所以，我就按照这个含意给变量和处理它的函数起了名字。所有这些重构的结果参见程序 5.5。更改后的代码仍然通过所有测试。

程序 5.5 PrimeGenerator.java，版本 4（部分）

```
public class PrimeGenerator
{
    private static boolean[] isCrossed;
    private static int[] result;

    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            initializeArrayOfIntegers(maxValue);
            crossOutMultiples();
            putUncrossedIntegerIntoResult();
```

```
            return result;
        }
    }

    private static void initializeArrayOfIntegers(int maxValue)
    {
        isCrossed = new boolean[maxValue + 1];
        for (i = 2; i < isCrossed.length; i++)
            isCrossed[i] = false;
    }

    private static void crossOutMultiples()
    {
        int maxPrimeFactor = calcMaxPrimeFactor();
        for (int i = 2; i <= maxPrimeFactor; i++)
            if (notCrossed(i))
                crossOutMultiplesOf(i);
    }

    private static int calcMaxPrimeFactor()
    {
        // We cross out all multiples of p; where p is prime.
        // Thus, all crossed out multiples have p and q for
        // factors.  If p > sqrt of the size of the array, then
        // q will never be greater than 1.  Thus p is the
        // largest prime factor in the array, and is also
        // the iteration limit.

        Double maxPrimeFactor = Math.sqrt(isCrossed.length) + 1;
        Return (int) maxPrimeFactor;
    }

    private static void crossOutMultiplesOf(int i)
    {
    for (int multiple = 2*i;
            multiple < isCrossed.length;
            multiple += i)
        isCrossed[multiple] = true;
    }

    private static boolean notCrossed(int i)
    {
    return isCrossed[i] == false;
    }
```

最后一个要重构的函数是 putUncrossedIntegersIntoResult。这个函数具有两部分功能。第一部分计算了数组中没有被过滤掉的整数的数目，并创建了一个同样大小的数组来存放这些结果；第二部分把那些没有被过滤掉的整数搬移到结果数组中。我把第一部分功能提取出来，放到它自己的函数中，并做了其他一些清理工作。更改后的代码仍然通过所有测试。

程序 5.6 PrimeGenerator.java，版本 5（部分）

```
private static void putUncrossedIntegersIntoResult()
{
    result = new int [numberOfUncrossedIntegers()];
    for (int j=0; i=2; i < isCrossed.length; i++)
        if (notCrossed(i))
            result[j++] = i;
}
```

```
private static int numberOfUncrossedIntegers()
{
    int count = 0;
    for (int i=2; i<isCrossed.length; i++)
        if (notCrossed(i))
            count++;

    return count;
}
```

最后再读一遍

接着，我对整个程序做了最后的审视，从头至尾地阅读了一遍，几乎像阅读一个几何证明。这是非常关键的一个步骤。直到现在，我们重构的都是代码片断。现在，我想看看把这些片断结合在一起是否是一个具有可读性的整体。

首先，我并不喜欢 initializeArrayOfIntegers 这个名字。实际上，初始化的并不是一个整数数组，而是一个布尔值数组。然而，更名为 initializeArrayOfBooleans 并不会有什么改善。在这个方法中，真正要做的是保留所有的相关整数，以便于接下来能够过滤掉它们的倍数。我同样也不喜欢 isCrossed 作为布尔值数组的名字。因此把它更改为 crossOut。更改后的代码仍然通过所有测试。

有人也许会觉得更改名字的工作比较琐碎，但是借助于一个重构浏览器，将足以对付这些调整——花费的代价微乎其微。即使在没有重构浏览器的情况下，使用简单的搜索和替换操作也可以轻而易举地完成这项工作。并且，测试可以极大程度地减少我们不知不觉中破坏一些功能的机会。

我不记得在编写有关 maxPrimeFactor 的代码时抽的是什么烟。呀！数组长度的平方根未必就是素数；那个方法没有计算出最大的素因子；说明性的注释是错误的。所以，我重写了该注释，使它可以更好地解释平方根背后的原理，并且适当地给所有的变量重新命了名。[①]更改后的代码仍然通过所有测试。

+1 在那里究竟起了什么作用？肯定是有点偏执了。我担心具有小数位的平方根会转换为小一点的整数，以至于不能充当遍历的上限。但是这种做法是不必要的。真正的遍历上限是小于或者等于数组长度平方根的最大素数。我去掉了+1。

测试都通过了，但是最后的更改使我相当紧张。我理解平方根背后的原理，但是我总觉得会有一些边角的地方没有测试到。所以，我另外编写了测试，用来检查在 2～500 之间所产生的素数列表中没有倍数存在。（参见程序 5.8 中的 testExhaustive 函数。）新的测试通过了，减轻了我的恐惧。

代码的其他部分读起来相当的优美。所以我觉得我们已经完成了重构。最后的版本如程序 5.7 和程序 5.8 所示。

程序 5.7 PrimeGenerator.java（最终版）

```
/**
```

① 在 Kent Beck 和 Jim Newkirk 重构该程序时，他们根本就没有使用平方根。Kent 认为平方根很难理解，并且如果从头至尾遍历数组的话，那么没有测试会失败。但是我不能使自己放弃效率方面的考虑。这一点显现出了我的汇编语言根源。

```
 * This class Generates prime numbers up to a user specified
 * maximum.  The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
 * multiples.  Repeat until there are no more multiples
 * in the array.
 */

public class PrimeGenerator
{
  private static boolean[] crossedOut;
  private static int[] result;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      uncrossIntegersUpTo(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void uncrossIntegersUpTo(int maxValue)
  {
    crossedOut = new boolean[maxValue + 1];
    for (int i = 2; i < crossedOut.length; i++)
      crossedOut[i] = false;
  }

  private static void crossOutMultiples()
  {
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
      if (notCrossed(i))
        crossOutMultiplesOf(i);
  }

  private static int determineIterationLimit()
  {
    // Every multiple in the array has a prime factor that
    // is less than or equal to the root of the array size,
    // so we don't have to cross of multiples of numbers
    // larger than that root.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
  }

  private static void crossOutMultiplesOf(int i)
  {
    for (int multiple = 2*i;
         multiple < crossedOut.length;
         multiple += i)
      crossedOut[multiple] = true;
  }

  private static boolean notCrossed(int i)
```

```
  {
    return crossedOut[i] == false;
  }

  private static void putUncrossedIntegersIntoResult()
  {
    result = new int[numberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        result[j++] = i;
  }

  private static int numberOfUncrossedIntegers()
  {
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        count++;

    return count;
  }
}
```

程序5.8 TestGeneratePrimes.java（最终版）

```
import junit.framework.*;

public class TestGeneratePrimes extends TestCase
{
  public static void main(String args[])
  {
     junit.swingui.TestRunner.main(
       new String[] {"TestGeneratePrimes"});
  }
  public TestGeneratePrimes(String name)
  {
    super(name);
  }

  public void testPrimes()
  {
    int[] nullArray = PrimeGenerator.generatePrimes(0);
    assertEquals(nullArray.length, 0);

    int[] minArray = PrimeGenerator.generatePrimes(2);
    assertEquals(minArray.length, 1);
    assertEquals(minArray[0], 2);

    int[] threeArray = PrimeGenerator.generatePrimes(3);
    assertEquals(threeArray.length, 2);
    assertEquals(threeArray[0], 2);
    assertEquals(threeArray[1], 3);

    int[] centArray = PrimeGenerator.generatePrimes(100);
    assertEquals(centArray.length, 25);
    assertEquals(centArray[24], 97);
  }

  public void testExhaustive()
  {
```

```
    for (int i = 2; i<500; i++)
      verifyPrimeList(PrimeGenerator.generatePrimes(i));
  }

  private void verifyPrimeList(int[] list)
  {
    for (int i=0; i<list.length; i++)
      verifyPrime(list[i]);
  }

  private void verifyPrime(int n)
  {
    for (int factor=2; factor<n; factor++)
      assert(n%factor != 0);
  }
}
```

5.2 结　论

重构后的程序读起来比一开始要好得多，程序工作得也更好一些。我对这个结果非常满意。程序变得更易理解，因此也更易更改，且程序结构的各部分之间互相隔离，这也使它更容易更改。

你也许担心提取出仅仅调用一次的函数会对性能造成负面的影响。我认为在大多数情况下，提取出函数所增加的可读性是值得花费额外的一些微小开销的。然而，也许那些少许的开销存在于深深的内部循环中，这将会造成较大的性能损失。我的建议是假设这种损失是可以忽略的，等待以后再去证明这种假设是错误的。

这值得我们花费时间吗？毕竟，程序已经可以完成所需的功能。我强烈推荐你应该经常对你所编写和维护的每一个模块进行这种重构实践。投入的时间和随后为自己和他人节省的努力相比起来是非常少的。

重构就好比用餐后对厨房的清理工作。第一次你没有清理它，你用餐是会快一点。但是由于没有对盘碟和用餐环境进行清洁，第二天做准备工作的时间就要更长一点。这会再一次促使你放弃清洁工作。的确，如果跳过清洁工作，你今天总是能够很快用完餐，但是脏乱在一天天的积累。最终，你得花费大量的时间去寻找合适的烹饪器具，凿去盘碟上已经干硬的食物残余，并把它们洗擦干净以使它们适合于烹饪。饭是天天要吃的。忽略掉清洁工作并不能真正加快做饭速度。

重构的目的，正像在本章中描述的，是为了每天清洁你的代码。我们不想让脏乱累积，我们不想“凿去并洗擦掉”随着时间累积的“干硬的”比特，我们想通过最小的努力就能够对我们的系统进行扩展和修改。要想具有这种能力，最重要的就是要保持代码的清洁。

关于这一点，我怎么强调都不过分。本书中所有的原则和模式对于脏乱的代码来说将没有任何价值。在学习原则和模式前，首先学习编写清洁的代码。

参考文献

1. Fowler, Martin. *Refactoring: Improve the Design of Existing Code.* Reading, MA: Addison- Wesley, 1999.

第 6 章　一次编程实践

设计和编程都是人的活动。忘记了这一点，将会失去一切。

——Bjarne Stroustrup，1991

为了演示一下 XP 的编程实践，Bob Koss（RSK）和 Bob Martin（RCM）要在一个小型的应用程序中使用结对编程（pair programming）的方法，你可以在一边进行观看。在创建该应用程序的过程中，会使用测试驱动的开发方法以及大量的重构。接下来的一幕是这两个 Bob 于 2000 年末在一家旅馆中实际编程情景的真实再现。

在创建这个程序的过程中，我们犯了很多的错误。这些错误包括代码方面的、逻辑方面的、设计方面的以及需求方面的。在学习本章时，会看到我们围绕这几个方面所进行的活动：识别出错误和误解，然后处理它们。过程是混乱的——过程中只要有人参与都是这样。结果……唔，令人吃惊，竟然能够从这样一个混乱的过程中出现秩序。

这个程序是计算保龄球比赛得分的，所以如果知道保龄球比赛的规则，会有助于理解本章内容。如果对保龄球比赛的规则不了解的话，可以察看章末的补充内容。

6.1　保龄球比赛

RCM：可以帮忙编写一个保龄球记分小程序吗？

RSK：（自言自语，“XP 中结对编程的实践规定当有人请求帮助时，不能够说“不”。若请求的人是你的老板，就更不能拒绝了。”）当然可以，Bob，非常高兴帮助你。

RCM：太好了，我想编写一个应用程序来记录一届保龄球联赛。需要记录下所有的比赛、确定团队的等级、确定每次周赛的优胜者和失败者，并且准确地记录每场比赛的成绩。

RSK：棒极了。我曾经是个很好的保龄球选手。这件事情很有趣。你已经列出了一些用户素材，想先做哪一个呢？

RCM：先来实现记录一场比赛成绩的功能吧！

RSK：好。它指的是什么呢？该素材的输入和输出是什么呢？

RCM：在我看来，输入只是一个投掷（throw）的序列。一次投掷仅仅是一个整数，表明了此次投球所击倒的木瓶数目。输出就是每一轮（frame）的得分。

RSK：如果你在这个练习中担任客户的角色，会希望什么形式的输入和输出呢？

RCM：好，我担任客户。我们需要一个函数，调用它可以添加投掷，还需要另外的函数用来获取得分。有几分像下面的样子：

```
ThrowBall(6);
ThrowBall(3);
assertEquals(9, getScore( ) );
```

RSK：好，我们需要一些测试数据。我来画一张记分卡的小草图（参见图 6.1）。

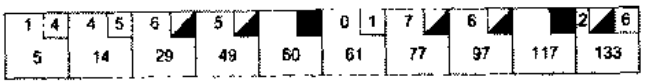
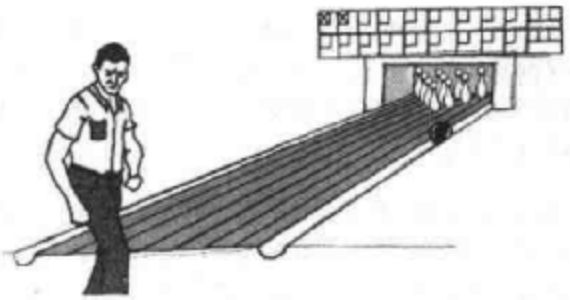


图 6.1　典型的保龄球比赛记分卡

RCM：这名选手发挥的很不稳定。

RSK：或许喝醉了，但是可以作为一个相当好的验收测试用例。



RCM：我们还需要其他的验收测试用例，稍后再考虑吧。该如何开始呢？要做一个系统设计吗？

RSK：我不介意用 UML 图来说明从记分卡中得到的一些问题领域概念。从中会发现一些候选对象，可以在随后的编码时使用。

RCM：（戴上他那顶强大的对象设计者的帽子）好，显然，Game 对象由一系列共 10 个 Frame 对象组成，每个 Frame 对象可以包含 1 个、2 个或者 3 个 Throw 对象。

RSK：好主意。这也正是我所想的。我立刻把它画出来（参见图 6.2）。

RSK：好，来选取一个要测试的类。从依赖关系链的尾部开始，依次往后如何？这样测试会容易些。

图 6.2　保龄球记分卡的 UML 图

RCM：当然可以。我们来创建 Throw 类的测试用例。

RSK：（开始键入代码）

```
//TestThrow.java---------------------------------
import junit.framework.*;

public class TestThrow extends TestCase
{
    public TestThrow(String name)
    {
```

```
        super(name);
    }

    //public void test????
}
```

RSK：Throw 对象应该具有什么行为呢？

RCM：它保存着比赛者所击倒的木瓶数。

RSK：好，你只用了寥寥数语，可见它确实没做什么事情。也许我们应该重新审视一下，来关注具有实际行为的对象，而不是仅仅存储数据的对象。

RCM：嗯，你的意思是说实际上 Throw 这个类也许不必存在？

RSK：是的，如果不具有任何行为，能有多重要呢？我还不知道它是否应该存在。我只是觉得如果我们去关注那些不仅仅只有 setter 和 getter 方法的对象的话，会更有效率。但是如果你想控制的话……（将键盘推到 RCM 面前。）

RCM：好吧，我们上移至依赖链上的 Frame 类，看看是否能在编写该类的测试用例时，完成 Throw 类。（把键盘推回给 RSK。）

RSK：（想知道 RCM 是在通过这种方式将我引入一个死胡同来教育我呢，还是他确实同意我的观点）好，新的文件，新的测试用例。

```
//TestFrame.java--------------------------------
import junit.framework.*;

public class TestFrame extends TestCase
{
    public TestFrame(String name)
    {
        super(name);
    }

    //public void test????
}
```

RCM：嗯，这是第二次键入这样的代码了。现在，你想到一些针对 Frame 类的有趣的测试用例吗？

RSK：Frame 类可以给出它的分数，每次投掷击倒的木瓶数，以及是否为全中或者补中……

RCM：好，用代码来说明问题。

RSK：（键入）

```
//TestFrame.java--------------------------------
import junit.framework.*;

public class TestFrame extends TestCase
{
    public TestFrame(String name)
    {
        super(name);
    }

    public void testScoreNoThrows()
    {
```

```
        Frame f = new Frame();
        assertEquals(0, f.getScore() );
    }
}

//Frame.java-----------------------------------
public class Frame
{
    public int getScore()
    {
        return 0;
    }
}
```

RCM：好，测试用例通过了，但是 getScore 实际上是一个愚蠢的方法。如果向 Frame 中加入一次投掷的话，它就会失败。所以我们来编写这样的测试用例，它会加入一些投掷，然后检查得分。

```
//TestFrame.java

public void testAddOneThrow()
{
    Frame f = new Frame();
    f.add(5);
    assertEquals(5, f.getScore() );
}
```

RCM：编译通不过。Frame 类中没有 add 方法。

RSK：我打赌如果你定义这个方法，就会通过编译 ;-)

RCM：

```
//Frame.java-----------------------------------
public class Frame
{
    public int getScore()
    {
        return 0;
    }

    public void add(Throw t)
    {
    }
}
```

RCM：（自言自语）这不可能编译通过的，因为还没有编写 Throw 类。

RSK：和我说说，Bob。在测试中传给 add 方法的是一个整数，而该方法期望一个 Throw 对象。add 方法不能具有两种形式。在我们再次关注 Throw 类前，你能描绘一下 Throw 类的行为吗？

RCM：哦，我甚至都没有注意到我写的是 f.add(5)。我应该写 f.add(new Throw(5))，但那太不优雅了。我真正想写的就是 f.add(5)。

RSK：先不管是否优雅，我们暂时把美学的考虑放到一边。你能描绘一下 Throw 对象的行为吗？是二元表示吗，Bob？

RCM：10110101101010100101。我不知道 Throw 是否具有一些行为。我现在觉得 Throw 就是 int。不过，我们不必再考虑它了，因为我们可以让 Frame.add 接受一个 int。

RSK：我觉得这样做的根本原因就是简单。当出现问题时，可以再使用一些复杂的方法。

RCM：同意。

```
//Frame.java-----------------------------------
public class Frame
{
    public int getScore()
    {
        return 0;
    }

    public void add(int pins)
    {
    }
}
```

RCM：好，编译通过而测试失败了。现在，我们来通过测试。

```
//Frame.java-----------------------------------
public class Frame
{
    public int getScore()
    {
        return itsScore;
    }

    public void add(int pins)
    {
       itsScore += pins;
    }
    private int itsScore = 0;
}
```

RCM：编译和测试都通过了，这明显太简单了。下一个测试用例是什么？

RSK：先休息一会儿好吗？

```
------------------------- Break ------------------------
```

RCM：不错。但 Frame.add 是一个脆弱的方法。如果用 11 作为参数去调用它会怎样呢？

RSK：如果发生这种情况，可以抛出异常。但是谁会去调用它呢？这个程序会成为被数千人使用的应用框架以至于我们必须要对这种情况进行防护吗？还是仅仅被你一人使用呢？如果是后者，只要调用它时不传入 11 就没问题了。（暗笑。）

RCM：好主意，系统的其他测试会捕获无效的参数。如果我们遇到麻烦，再把这个检查加进来也不迟。目前，add 函数还不能处理全中和补中的情况。我们编写一个测试用例来表现这种情况。

RSK：嗯……如果用调用 add(10)来表示一个全中，那么 getScore 应该返回什么值呢？我不知道该如何写这个断言，也许我们提出的问题是错误的，或者我们选择提问的对象是错误的。

RCM：如果调用了 add(10)，或者调用了 add(3)后又调用了 add(7)，那么随后调用 Frame 的 getScore 方法是没有意义的。此时当前 Frame 对象必须要根据随后几个 Frame 实例的得分才能计算自己的得分。如果后面的 Frame 实例还不存在，那么它会返回一些令人讨厌的值，像－1。我不希望返回－1。

RSK：是的，我也不喜欢返回－1 这个想法。你刚刚引入了一个概念，就是 Frame 之间要互相知晓。谁会持有这些不同的 Frame 对象呢？

RCM：Game 对象。

RSK：那么 Game 依赖于 Frame，而 Frame 反过来又依赖于 Game。我不喜欢这样。

RCM：Frame 不必依赖于 Game，可以把它们放置在一个链表中。每个 Frame 持有指向它前面以及后面 Frame 的指针。要获取一个 Frame 的得分，该 Frame 会获取前一个 Frame 的得分；如果该 Frame 中有补中或者全中的情况，它会从后面 Frame 中获取所需的得分。

RSK：好的，不过不太形象，我感觉有些不清楚。写一些代码看看吧。

RCM：好。我们首先要编写一个测试用例。

RSK：是针对 Game 呢，还是另一个针对 Frame 的呢？

RCM：我认为应该针对 Game，因为是 Game 构建了 Frame 并把它们互相连接起来。

RSK：你是想停下我们正在做的有关 Frame 的工作，而跳转到 Game 上去呢？还是只想要一个 MockGame 对象来完成 Frame 正常运转所需要的工作呢？

RCM：我们停止在 Frame 上的工作，转到 Game 上来吧。Game 的测试用例应当可以证明我们需要 Frame 链表。

RSK：我不知道它们是怎样证明的。我需要代码。

RCM：（键入代码）

```
//TestFrame.java--------------------------------
import junit.framework.*;

public class TestFrame extends TestCase
{
    public TestFrame(String name)
    {
        super(name);
    }

    public void testoneThrows()
    {
       Game g = new Game();
       g.add(5);
        assertEquals(5, g.score() );
    }
}
```

RCM：看上去合理吗？

RSK：当然合理，但是我仍然在寻找需要 Frame 链表的证据？

RCM：我也是，我们继续这些测试用例，看看会有什么结果。

```
//Game.java-----------------------------------------
public class Game
{
```

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

    public void add(int pins)
    {
    }
}
```

RCM：好，编译通过，而测试失败了。现在我们来让测试通过。

```
//Game.java-----------------------------------------
public class Game
{
    public int score()
    {
        return itsScore;
    }

    public void add(int pins)
    {
       itsScore += pins;
    }
    private int itsScore = 0;
}
```

RCM：测试通过了，很好。

RSK：不错，但我仍在寻找需要 Frame 对象链表的重要证据。最初就是它致使我们认为需要 Game。

RCM：是的，这也是我正在寻找的。我肯定一旦加进了有关补中和全中的测试用例，就必须得构建 Frame，并把它们用链表链接在一起。但是在代码迫使这样做前，我不想这样做。

RSK：好主意。我们来继续逐步完成 Game。编写另外一个关于有两次投掷但没有补中的情况的测试怎么样？

RCM：好，这应该会立刻通过测试。我们来试试。

```
//TestGame.java-------------------------------------------

public void testTwoThrowsNoMark()
{
    Game g = new Game();
    g.add(5);
    g.add(4);
    assertEquals(9, g.score());
}
```

RCM：是的，这个测试通过了。现在我们来试一下有 4 次投掷但没有补中和全中的情况。

RSK：嗯，这个测试也能通过。但这不是我所期望的，我们可以一直增加投掷数，甚至根本不需要一个 Frame。但是我们还不曾考虑补中或者全中的情况。也许到那时我们就会需要一个 Frame。

RCM：这也是我正在考虑的。不管怎样，考虑一下这个测试用例：

```
//TestGame.java-------------------------------------------
public void testFourThrowsNoMark()
{
```

```
        Game g = new Game();
        g.add(5);
        g.add(4);
        g.add(7);
        g.add(2);
        assertEquals(18, g.score());
        assertEquals(9, g.scoreForFrame(1));
        assertEquals(18, g.scoreForFrame(2));
    }
```

RCM：这看上去合理吗？

RSK：当然合理。我忘了必须要能显示每轮的得分。啊，我把咱们画的记分卡草图当可乐杯垫来用了。这是我忘记的原因。

RCM：（叹气）好，首先我们给 Game 加入 scoreForFrame 方法使测试失败。

```
//Game.java-----------------------------------------

    public int scoreForFrame(int frame)
    {
       return 0;
    }
```

RCM：好极了，编译通过，测试失败了，现在，怎样通过测试呢？

RSK：我们可以定义 Frame 对象了，但这是通过测试的最简单方法吗？

RCM：不是，事实上，我们只需要在 Game 中创建一个整数数组。每次对 add 的调用都会在这个数组里添加一个新的整数。每次对 scoreForFrame 的调用只需要前向遍历这个数组并计算出得分。

```
//Game.java-----------------------------------------
public class Game
{
    public int score()
    {
        return itsScore;
    }

    public void add(int pins)
    {
       itsThrows[itsCurrentThrow++] = pins;
       itsScore += pins;
    }

    public int scoreForFrame(int frame)
    {
       int score = 0;
       for (int ball = 0;
            frame > 0 && (ball < itsCurrentThrow);
            ball += 2, frame--)
       {
            score += itsThrows[ball] + itsThrows[ball+1];
       }
       return score;
    }

    private int itsScore = 0;
```

```
    private int[] itsThrows = new int[21];
    private int itsTurrentThrow = 0;
}
```

RCM：（对自己很满意）看，可以工作了。

RSK：为什么要用 21 这个魔数（magic number）呢？

RCM：它表示一场比赛中最大可能的投掷数。

RSK：讨厌。让我猜猜，你年轻时，是一个 Unix hacker，并自豪于把整个应用程序编写在没人理解的一条语句中。需要重构 scoreForFrame，这样可以更好的理解它。但是在考虑重构前，我来问另外一个问题。Game 是放置这个方法的最好地方吗？我认为 Game 违反了单一职责原则（Single Responsibility Principle，简称 SRP）。[1]它接收投掷并且知道如何计算每轮的得分。你觉得增加一个 Scorer 对象如何？

RCM：（粗鲁地摆了一下手）目前我还不知道这个函数该放在哪里，现在我感兴趣的只是让记分程序工作起来。完成所需的功能后，我们再来讨论 SRP 的价值。不过，我明白你所说的 Unix hacker 指的是什么。我们来简化这个循环。

```
public int scoreForFrame(int theFrame)
{
  int ball = 0;
  int score = 0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
  {
      score += itsThrows[ball++] + itsThrows[ball++];
  }

  return score;
}
```

RCM：好了一点，但是 score+= 这个表达式具有副作用。不过，这个表达式中两个加数表达式的求值顺序无关紧要，所以这里不会造成副作用。（是这样吗？两个增量操作会不会在任意一个数组运算前完成呢？）

RSK：我认为可以做个实验来证明这里不会有任何副作用，但是这个函数还不能处理补中和全中的情况。我们是应该继续使它更易读些呢？还是应该给它添加更多的功能呢？

RCM：实验只对特定的编译器有意义。其他的编译器可能采用不同的求值顺序。我不知道这是不是一个问题，但是我们还是先来去除这种可能的顺序依赖，然后再编写更多的测试用例来添加功能。

```
public int scoreForFrame(int theFrame)
{
  int ball = 0;
  int score = 0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
```

[1] 参见第 8 章“单一职责原则（SRP）”。

```
            currentFrame++)
        {
            int firstThrow = itsThrows[ball++];
            int secondThrow = itsThrows[ball++];
            score = firstThrow + secondThrow;
        }

        return score;
    }
```

RSK：好，下个测试用例。我们来试试补中的情况。

```
    public void testSimpleSpare()
    {
            Game g = new Game();
    }
```

RCM：我已经厌倦总是写这个了，我们来重构一下测试，把 Game 对象的创建放到 setUp 函数中吧。

```
    //TestGame.java--------------------------------
    import junit.framework.*;

    public class TestGame extends TestCase
    {
        public TestGame(String name)
        {
          super(name);
        }

        private Game g;

        public void setUp()
        {
           g = new Game();
        }

        public void testOneThrows()
        {
           g.add(5);
          assertEquals(5, g.score() );
        }

        public void testTwoThrowsNoMark()
        {
          g.add(5);
          g.add(4);
          assertEquals(9, g.score());
        }

        public void testFourThrowsNoMark()
        {
          g.add(5);
          g.add(4);
          g.add(7);
          g.add(2);
          assertEquals(18, g.score());
          assertEquals(9, g.scoreForFrame(1));
          assertEquals(18, g.scoreForFrame(2));
        }
```

```
    public void testSimpleSpare()
    {
    }
}
```

RCM：好多了，现在来编写关于补中的测试用例。

```
public void testSimpleSpare()
{
   g.add(3);
   g.add(7);
   g.add(3);
   assertEquals(13, g.scoreForFrame(1));
}
```

RCM：好，测试失败了，现在我们要让它通过。

RSK：我来写吧。

```
public int scoreForFrame(int theFrame)
{
   int ball = 0;
  int score = 0;
   for (int currentFrame = 0;
        currentFrame < theFrame;
        currentFrame++)
   {
       int firstThrow = itsThrows[ball++];
       int secondThrow = itsThrows[ball++];

       int frameScore = firstThrow + secondThrow;
       // spare needs next frames first throw
       if (frameScore ==10 )
           score += frameScore + itsThrows[ball++];
       else
           score += frameScore;
   }

   return score;
}
```

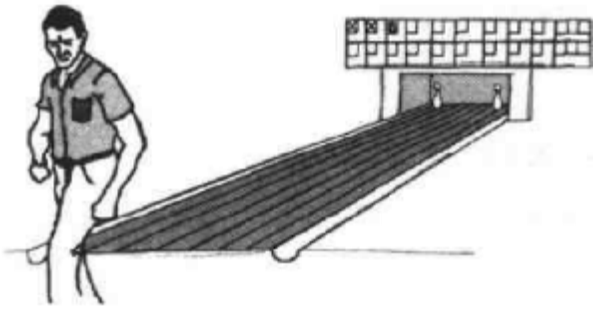
RSK：啊哈，可以工作了。

RCM：（抢过键盘）不错，但是我认为在 frameScore==10 时不应该对变量 ball 递增。有个测试用例可以证明我的观点。

```
Public void testSimpleFrameAfterSpare()
{
   g.add(3);
   g.add(7);
   g.add(3);
   g.add(2);
   assertEquals(13, g.scoreForFrame(1));
   assertEquals(18, g.score());
}
```

RCM：哈，看，测试失败了。现在，如果把这个讨厌的额外递增操作去掉……

```
if ( frameScore == 10)
   score += frameScore + itsThrows[ball];
```



RCM：嗯……测试还是失败了……是 score 方法错了吗？把测试用例改为调用 scoreForFrame(2)试试。

```
Public void testSimpleFrameAfterSpare()
{
   g.add(3);
   g.add(7);
   g.add(3);
   g.add(2);
   assertEquals(13, g.scoreForFrame(1));
   assertEquals(18, g.scoreForFrame(2));
}
```

RCM：唔……测试通过了。肯定是 score 方法出问题了。我们来看一下。

```
public int score()
{
   return itsScore;
}

public void add(int pins)
{
   itsThrows[itsCurrentThrow++] = pins;
   itsScore += pins;
}
```

RCM：是的，是错了。score 方法只是返回木瓶数的和，而不是正确的得分。我们要让 score 做的是用当前轮作为参数去调用 scoreForFame()。

RSK：我们不知道当前是哪轮。我们来把这个信息加到现有的每个测试中，当然，每次一个。

RCM：好的。

```
//TestGame.java--------------------------------
  public void testOneThrows()
  {
    g.add(5);
    assertEquals(5, g.score() );
    assertEquals(1, g.getCurrentFrame());
  }

//Game.java-----------------------------------
  public int getCurrentFrame()
  {
   return 1;
  }
```

RCM：不错，可以工作了。但是没什么意义。我们完成下一个测试用例。

```
public void testTwoThrowsNoMark()
{
  g.add(5);
  g.add(4);
  assertEquals(9, g.score());
  assertEuqals(1, g.getCurrentFrame());
}
```

RCM：同样无趣，再来下一个。

```
public void testFourThrowsNoMark()
{
  g.add(5);
  g.add(4);
  g.add(7);
  g.add(2);
  assertEquals(18, g.score());
  assertEquals(9, g.scoreForFrame(1));
  assertEquals(18, g.scoreForFrame(2));
  assertEquals(2, g.getCurrentFrame());
}
```

RCM：这次失败了。现在我们来让它通过。

RSK：我认为算法很简单，因为每轮有两次投掷，所以只要将投掷数除以 2 就可以了。除非有全中的情况……但是我们至今还没有考虑过全中的情况，此时也忽略了吧。

RCM：（在+1 和-1 之间胡乱调整，直到可以正常工作。）[①]

```
public int getCurrentFrame()
{
 return 1 + (itsCurrentThrow - 1)/2;
}
```

RCM：这不太令人满意。

RSK：如果不是每次都去计算它，会怎么样呢？如果每次投掷后去调整一个 currentFrame 成员变量，会怎么样呢？

RCM：不错，我们来试试。

```
//Game.java--------------------------------
  public int getCurrentFrame()
  {
   return itsCurrentFrame;
  }

  public void add(int pins)
  {
   itsThrows[itsCurrentThrow++] = pins;
   itsScore += pins;
   if (firstThrow == true)
   {
       firstThrow = false;
```

① Dave Thomas 和 Andy Hunt 称之为“基于巧合编程（programming by coincidence）”。

```
        itsCurrentFrame++;
    }
    else
    {
        firstThrow = true;
    }
  }

  private int itsCurrentFrame = 0;
  private Boolean firstThrow = true;
```

RCM：好，可以工作了。但是这也意味着当前轮指的是最近一次投掷所在轮，而不是下一次投掷所在轮。只要我们记住这一点，就没有问题。

RCM：我没有这么好的记忆力，我们来把程序修改得更易读些。但是在调整前，我们先把代码从 add() 中提取出来，放到一个称为 adjustCurrentFrame()或者其他名字的私有成员函数中。

RCM：好，听起来不错。

```
public void add(int pins)
{
  itsThrows[itsCurrentThrow++] = pins;
  itsScore += pins;
  adjustCurrentFrame();
}

private void adjustCurrentFrame()
{
  if (firstThrow == true)
  {
      firstThrow = false;
      itsCurrentFrame++;
  }
  else
  {
      firstThrow = true;
  }
}
```

RCM：现在，我们把变量和函数的名字改得更清晰些。我们该如何称呼 itsCurrentFrame 呢？

RSK：我挺喜欢这个名字。但我认为对它递增的位置不对。在我看来，当前轮是正在进行的投掷所在轮。所以应该在该轮最后一次投掷完毕后，才对它递增。

RCM：我同意。我们来修改测试用例以体现这一点，然后再去修正 adjustCurrentFrame。

```
//TestGame.java--------------------------------
public void testTwoThrowsNoMark()
{
  g.add(5);
  g.add(4);
  assertEquals(9, g.score());
  assertEuqals(2, g.getCurrentFrame());
}

public void testFourThrowsNoMark()
{
  g.add(5);
```

```
    g.add(4);
    g.add(7);
    g.add(2);
    assertEquals(18, g.score());
    assertEquals(9, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
    assertEquals(3, g.getCurrentFrame());
}

//Game.java---------------------------------
private void adjustCurrentFrame()
{
   if (firstThrow == true)
   {
       firstThrow = false;
   }
   else
   {
       firstThrow = true;
       itsCurrentFrame++;
   }

  private int itsCurrentFrame = 1;
}
```

RCM：不错，可以工作了。现在我们来为 getCurrentFrame 编写两个具有补中情况的测试用例。

```
public void testSimpleSpare()
{
   g.add(3);
   g.add(7);
   g.add(3);
   assertEquals(13, g.scoreForFrame(1));
   assertEquals(2, g.getCurrentFrame());
}


public void testSimpleFrameAfterSpare()
{
   g.add(3);
   g.add(7);
   g.add(3);
   g.add(2);
   assertEquals(13, g.scoreForFrame(1));
   assertEquals(18, g.scoreForFrame(2));
   assertEquals(3, g.getCurrentFrame());
}
```

RCM：通过了。现在，回到原先的问题上。我们要让 score 能够工作。现在可以让 score 去调用 scoreForFrame(getCurrentFrame()-1)。

```
public void testSimpleFrameAfterSpare()
{
   g.add(3);
   g.add(7);
   g.add(3);
   g.add(2);
   assertEquals(13, g.scoreForFrame(1));
   assertEquals(18, g.scoreForFrame(2));
```

```
    assertEquals(18, g.score());
    assertEquals(3, g.getCurrentFrame());
}

//Game.java-----------------------------------------------
public int score()
{
    return scoreForFrame(getCurrentFrame()-1);
}
```

ECM：TestOneThrow 测试用例失败了，我们来看看。

```
public void testOneThrows()
{
  g.add(5);
  assertEquals(5, g.score() );
  assertEquals(1, g.getCurrentFrame());
}
```

RCM：只有一次投掷，第 1 轮是不完整的。score 方法调用了 scoreForFrame(0)。这真讨厌。

RSK：也许是， 也许不是。这个程序是写给谁的？谁会去调用 score()呢？假定不会针对不完整轮调用该方法合理吗？

RCM：是的，但是它让我觉得不舒服。为了解决这个问题，我们要从 testOneThrow 测试用例中去掉 score？这是我们要做的吗？

RSK：可以这样做，甚至可以去掉整个 testOneThrow 测试用例。它曾把我们引到所关心的测试用例上。但现还有实际用处吗？在所有其他测试用例中依然具有对于该问题的覆盖。

RCM：是的，我明白你的意思。好，去掉它。（编辑代码，运行测试，出现绿色的指示条。）啊，很好。现在最好来关注关于全中的测试用例。毕竟，我们想看到所有这些 Frame 对象被构建成一个链表，不是吗？（窃笑。）

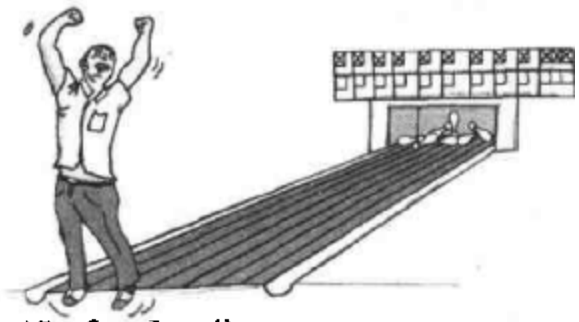
```
public void test()
{
  g.add(10);
  g.add(3);
  g.add(6);
  assertEquals(19, g.scoreForFrame(1) );
  assertEquals(28, g.score() );
  assertEquals(3, g.getCurrentFrame());
}
```

ECM：好，像预期一样，编译通过，测试失败了。现在要通过测试。

```
//Game.java-----------------------------------------------
public class Game
{
  public void add(int pins)
  {
   itsThrows[itsCurrentThrow++] = pins;
   itsScore += pins;
   adjustCurrentFrame(pins);
  }

  private void adjustCurrentFrame(int pins)
  {
   if (firstThrow == true)
   {
       if (pins == 10)     //strike
           itsCurrentFrame++;
       else
           firstThrow = false;
   }
   else
   {
       firstThrow = true;
       itsCurrentFrame++;
   }
  }

  public int scoreForFrame(int theFrame)
  {
    int ball = 0;
    int score = 0;
   for (int currentFrame = 0;
        currentFrame < theFrame;
        currentFrame++)
   {
       int firstThrow = itsThrows[ball++];
       if (firstThrow == 10)
       {
           score += 10 + itsThrows[ball] + itsThrows[ball+1];
       }
       else
       {
           int secondThrow = itsThrows[ball++];

           int frameScore = firstThrow + secondThrow;
           // spare needs next frames first throw
           if (frameScore ==10 )
               score += frameScore + itsThrows[ball];
           else
               score += frameScore;
       }
   }

   return score;
  }
  private int itsScore = 0;
  private int[] itsThrows = new int[21];
  private int itsCurrentThrow = 0;
  private int itsCurrentFrame = 1;
```

```
    private Boolean firstThrow = true;
  }
```

RCM：不错，不是特别难，我们来看看能否为一次完美的比赛记分。



```
public void testPerfectGame()
{
   for (int i=0; i<12; i++)
   {
       g.add(10);
   }
   assertEquals(300, g.score());
   assertEquals(10, g.getCurrentFrame());
}
```

RCM：奇怪，它说得分是 330。怎么会是这样？

RSK：因为当前轮一直被累加到了 12。

RCM：唉，要把它限定到 10。

```
private void adjustCurrentFrame(int pins)
{
   if (firstThrow == true)
   {
       if (pins -= 10)     //strike
           itsCurrentFrame++;
       else
           firstThrow = false;
   }
   else
   {
       firstThrow = true;
       itsCurrentFrame++;
   }
   itsCurrentFrame = Math.min(10, itsCurrentFrame);
}
```

RCM：该死，这次它说得分是 270。出什么问题了？

RSK：Bob，score 函数把 getCurrentFrame 减了 1，所以它给出的是第 9 轮的得分，而不是第 10 轮的。

RCM：什么，你是说，应该把当前 frame 限定到 11 而不是 10？我再试试。

```
itsCurrentFrame = Math.min(11, itsCurrentFrame);
```

RCM：好，现在得到了正确的得分，但是却因为当前轮是 11，不是 10 而失败了。烦人！当前轮真是个难办的事情。我们希望当前轮指的是比赛者正在进行的投掷所在轮，但是在比赛结束时，

这意味着什么呢？

RSK：也许我们应当回到原先的观点，认为当前轮指的是最后一次掷球所在轮。

RCM：或者，我们也许要提出最近的完整轮这样一个概念？毕竟，在任何时间点上比赛的得分都是最近的完整轮的得分。

RSK：一个完整轮指的是可以为之计算得分的轮，对吗？

RCM：是的，如果一轮中有补中的情况，那么要在下一个球投掷后该轮才算完整。如果一轮中有全中的情况，那么要在下两个球投掷后该轮才算完整。如果一轮中没有上述两种情况出现，那么该轮中第二球投掷完毕后就算完整了。

等一会……我们正要使 score()方法可以工作，对吗？我们所需要做的就是在比赛结束时让 score()调用 scoreForFrame(10)。

RSK：你怎么知道比赛结束了呢？

RCM：如果 adjustCurrentFrame 对 itsCurrentFrame 的增加超过 10，那么比赛就是结束了。

RSK：等等。你的意思是说如果 getCurrentFrame 返回了 11，比赛就算结束了。可程序现在就是这样做的呀！

RCM：嗯。你的意思是我们应该修改测试用例，使之和程序一致？

```
public void testPerfectGame()
{
   for (int i=0; i<12; i++)
   {
       g.add(10);
   }
   assertEquals(300, g.score());
   assertEquals(11, g.getCurrentFrame());
}
```

RCM：不错，通过了。这和让 getMonth 在 1 月份时返回 0 性质是一样的。可是我还是觉得不太舒服。

RSK：也许后面后有好的方法。现在，我发现了一个 bug。我可以？（抢过键盘。）

```
public void testEndOfArray()
{
   for (int i=0; i<9; i++)
   {
       g.add(0);
       g.add(0);
   }
   g.add(2);
   g.add(8);       // 10th frame spare
   g.add(10);      // Strike in last position of array.
   assertEquals(20, g.score());
}
```

RSK：嗯。没有失败。我以为既然数组的第 21 个元素是一个全中，计分程序会试图把数组的第 22 个和第 23 个元素的加进去。但是我想它可能没有这么做。

RCM：嗯，你还在想着记分对象，不是吗？无论如何，我都明白你的意思，但是由于 score 决不会

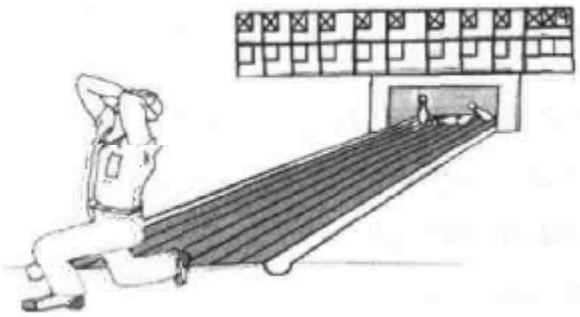
用大于 10 的参数去调用 scoreForFrame，所以这最后一次全中实际上没有被作为全中处理。只是为了最后一轮中补中的完整性才把它作为 10 分计算的。我们决不会越过数组的边界。

RSK：好的，我们来把原先记分卡上的数据输入到程序中。

```
Public void testSampleGame()
{
   g.add(1);
   g.add(4);
   g.add(4);
   g.add(5);
   g.add(6);
   g.add(5);
   g.add(6);
   g.add(6);
   g.add(10);
   g.add(0);
   g.add(1);
   g.add(7);
   g.add(3);
   g.add(6);
   g.add(4);
   g.add(10);
   g.add(2);
   g.add(8);
   g.add(6);
   assertEquals(133, g.score());
}
```

RSK：不错，测试通过了。你还能想到其他的一些测试用例吗？

RCM：是的，我们来多测试一些边界情况。这个如何？一个可怜的家伙投掷出了 11 次全中，而最后一次仅击中了 9 个。



```
public void testHeartBreak()
{
   for (int i=0; i<11; i++)
       g.add(10);
   g.add(9);
   assertEquals(299, g.score());
}
```

RCM：通过了。好的，再来测试一下第 10 轮是补中的情况如何？

```
public void testTenthFrameSpare()
{
   for (int i=0; i<9; i++)
       g.add(10);
```

```
        g.add(9);
        g.add(1);
        g.add(1);
        assertEquals(270, g.score());
    }
```

RCM：（高兴地盯着绿色的指示条）也通过了。我再也想不出更多的测试用例了，你呢？

RSK：我也想不出了，我认为已经覆盖了所有的情况。此外，我实在想重构这个混乱的程序。我还是认为应该有 scorer 对象。

RCM：是的，scoreForFrame 函数确实很乱，我们来考虑一下。

```
public int scoreForFrame(int theFrame)
{
  int ball = 0;
  int score = 0;
  for (int currentFrame = 0;
       currentFrame < theFrame;
       currentFrame++)
  {
      int firstThrow = itsThrows[ball++];
      if (firstThrow == 10)
      {
          score += 10 + itsThrows[ball] + itsThrows[ball+1];
      }
      else
      {
          int secondThrow = itsThrows[ball++];

          int frameScore = firstThrow + secondThrow;
          // spare needs next frames first throw
          if (frameScore ==10 )
              score += frameScore + itsThrows[ball];
          else
              score += frameScore;
      }
  }

  return score;
}
```

RCM：我很想把 else 子句的实现体提取出来作为一个名为 handleSecondThrow 的单独函数，但是因为它使用了 ball、firstThrow 以及 secondThrow 这些局部变量，所以不行。

RSK：我们可以把这些局部变量变为成员变量。

RCM：是的，这对你认为的把记分部分剥离出来放到它自己的 scorer 对象中去的看法又多了几分支持。好，我们来试试。

RSK：（抢过键盘。）

```
private void adjustCurrentFrame(int pins)
{
   if (firstThrowInFrame == true)
   {
       if (pins == 10)     //strike
           itsCurrentFrame++;
```

```
            else
                firstThrowInFrame = false;
        }
        else
        {
            firstThrowInFrame = true;
            itsCurrentFrame++;
        }
        itsCurrentFrame = Math.min(10, itsCurrentFrame);
    }

    public int scoreForFrame(int theFrame)
    {
      ball = 0;
      int score = 0;
       for (int currentFrame = 0;
            currentFrame < theFrame;
            currentFrame++)
       {
            firstThrow = itsThrows[ball++];
            if (firstThrow == 10)
            {
                score += 10 + itsThrows[ball] + itsThrows[ball+1];
            }
            else
            {
                secondThrow = itsThrows[ball++];

                int frameScore = firstThrow + secondThrow;
                // spare needs next frames first throw
                if (frameScore ==10 )
                    score += frameScore + itsThrows[ball];
                else
                    score += frameScore;
            }
       }

       return score;
    }
    private int ball;
    private int firstThrow;
    private int secondThrow;

    private int itsScore = 0;
    private int[] itsThrows = new int[21];
    private int itsCurrentThrow = 0;
    private int itsCurrentFrame = 1;
    private boolean firstThrowInFrame = true;
```

RSK：没想到会有名字冲突。我们已经有了一个名为 firstThrow 的实例变量。不过，把它命名为 firstThrowInFrame 会更好一些。无论如何，现在可以工作了。这样就可以把 else 子句剥离到它自己的函数中去。

```
    public int scoreForFrame(int theFrame)
    {
      ball = 0;
      int score = 0;
       for (int currentFrame = 0;
            currentFrame < theFrame;
```

```
            currentFrame++)
        {
            firstThrow = itsThrows[ball++];
            if (firstThrow == 10)
            {
                score += 10 + itsThrows[ball] + itsThrows[ball+1];
            }
            else
            {
                score += handleSecondThrow();
            }
        }

        return score;
    }

    private int handleSecondThrow()
    {
        int score = 0;
        secondThrow = itsThrows[ball++];

        int frameScore = firstThrow + secondThrow;
        // spare needs next frames first throw
        if ( frameScore == 10)
            score += frameScore + itsThrows[ball];
        else
            score += frameScore;
        return score;
    }
```

RCM：看一下 scoreForFrame 方法的结构！用伪代码来描述，看起来像这样：

```
if strike
    score += 10 + nextTwoBalls();
else
    handleSecondThrow;
```

RCM：如果把它改成下面的形式会怎样？

```
if strike
    score += 10 + nextTwoBalls();
else if spare
    score += 10 + nextBall();
else
    score += twoBallInFrame()
```

RSK：好极了！这正是保龄球的记分规则，不是吗？好的，我们来看看能否在实际的函数中实现这个结构。首先，我们来改变一下增加 ball 变量的方式，使得在上面三种情况中可以独立地操纵它。

```
public int scoreForFrame(int theFrame)
{
    ball = 0;
    int score = 0;
    for (int currentFrame = 0;
         currentFrame < theFrame;
         currentFrame++)
    {
        firstThrow = itsThrows[ball];
        if (firstThrow == 10)
        {
```

```
            ball++;
            score += 10 + itsThrows[ball] + itsThrows[ball+1];
        }
        else
        {
            score += handleSecondThrow();
        }
    }

    return score;
}

private int handleSecondThrow()
{
    int score = 0;
    secondThrow = itsThrows[ball+1];

   int frameScore = firstThrow + secondThrow;
   // spare needs next frames first throw
   if ( frameScore == 10)
   {
    ball += 2;
    score += frameScore + itsThrows[ball];
   }
   else
   {
    ball += 2;
    score += frameScore;
   }
   return score;
}
```

RCM：（抢过键盘）好，现在我们来去掉 firstThrow 和 secondThrow 变量，并用适当的函数来替代它们。

```
public int scoreForFrame(int theFrame)
{
  ball = 0;
  int score = 0;
   for (int currentFrame = 0;
        currentFrame < theFrame;
        currentFrame++)
   {
       firstThrow = itsThrows[ball];
       if (strike())
       {
           ball++;
           score += 10 + nextTwoBalls();
       }
       else
       {
           score += handleSecondThrow();
       }
   }

   return score;
}

private boolean strike()
{
   return itsThrows[ball] == 10;
```

```
}

private int nextTwoBalls()
{
   return intThrows[ball] + itsThrows[ball+1];
}
```

RCM：这一步成功了。继续。

```
private int handleSecondThrow()
{
   int score = 0;
   secondThrow = itsThrows[ball+1];

  int frameScore = firstThrow + secondThrow;
  // spare needs next frames first throw
  if  (spare())
  {
   ball += 2;
   score += 10 + nextBall();
  }
  else
  {
   ball += 2;
   score += frameScore;
  }
  return score;
}

private boolean spare()
{
   return (itsThrows[ball] + itsThrows[ball+1]) == 10;
}

private int nextBall()
{
   return itsThrows[ball];
}
```

RCM：好，也成功了。现在来处理 frameScore。

```
private int handleSecondThrow()
{
   int score = 0;
   secondThrow = itsThrows[ball+1];

  int frameScore = firstThrow + secondThrow;
  // spare needs next frames first throw
  if  (spare())
  {
   ball += 2;
   score += 10 + nextBall();
  }
  else
  {
   score += twoBallsInFrame();
   ball += 2;
  }
  return score;
 }
```

```
private int twoBallsInFrame()
{
  return itsThrows[ball] + itsThrows[ball+1];
}
```

RSK：Bob，你没有用一致的方式去增加变量 ball。在补中和全中的情况中，你是在记分前去增加它的。而在调用 twoBallsInFrame 的情况中，你却在记分后增加它。这样代码就依赖于这个顺序了！怎么回事？

RCM：对不起，我本应当解释一下的。我打算把这个增量操作移到全中、补中和 twoBallInFrame 中去。这样的话，它们就会从 scoreForFrame 函数中消失，并且该函数看上去就很像伪码形式了。

RSK：好，再信你一次。不过要记住，我可看着呢。

RCM：好的，现在不会再使用 firstThrow、secondThrow 和 frameScore 了，可以把它们去掉了。

```
public int scoreForFrame(int theFrame)
{
  ball = 0;
  int score = 0;
   for (int currentFrame = 0;
        currentFrame < theFrame;
        currentFrame++)
   {
       if (strike())
       {
           ball++;
           score += 10 + nextTwoBalls();
       }
       else
       {
           score += handleSecondThrow();
       }
   }

   return score;
}

private int handleSecondThrow()
{
   int score = 0;
  // spare needs next frames first throw
  if  (spare())
  {
   ball += 2;
   score += 10 + nextBall();
  }
  else
  {
   score += twoBallsInFrame();
   ball += 2;
  }
  return score;
}
```

RCM：（从他的眼神可以看出出现了绿色的指示条）现在，因为惟一耦合这 3 种情况的变量是 ball，而 ball 在每种情况下都是独立处理的，所以可以把这 3 种情况合并在一起。

```
public int scoreForFrame(int theFrame)
{
  ball = 0;
  int score = 0;
   for (int currentFrame = 0;
        currentFrame < theFrame;
        currentFrame++)
   {
       if (strike())
       {
           ball++;
           score += 10 + nextTwoBalls();
       }
       else if (spare())
       {
           ball += 2;
           score += 10 + nextBall();
       }
       else
       {
           score += twoBallsInFrame();
       }
   }
   return score;
}
```

RSK：好，现在可以使 ball 增加的方式一致，并为该函数起一个更清楚一些的命名。（抢过键盘。）

```
public int scoreForFrame(int theFrame)
{
  ball = 0;
  int score = 0;
   for (int currentFrame = 0;
        currentFrame < theFrame;
        currentFrame++)
   {
       if (strike())
       {
           score += 10 + nextTwoBallsForStrike();
           ball++;
       }
       else if (spare())
       {
           score += 10 + nextBallForSpare();
           ball += 2;
       }
       else
       {
           score += twoBallsInFrame();
           ball += 2;
       }
   }
   return score;
}

private int nextTwoBallForStrike()
{
   return itsThrows[ball+1] + itsThrows[ball+2];
}
```

```
private int nextBallForSpare()
{
  return itsThrows[ball+2];
}
```

RCM：看一下 scoreForFrame 方法！这正是保龄球记分规则的最简洁描述。

RSK：但是，Bob，Frame 对象的链表呢？（窃笑，窃笑。）

RCM：（叹气）我们被过度图示设计的恶魔迷惑了。我的天，3 个画在餐巾纸背面的小方框，Game、Frame，还有 Throw，看来还是太复杂了，并且是完全错误的。

RSK：以 Throw 类开始就是错误的。应该先从 Game 类开始！

RCM：确实是这样！所以，下次我们试着从最高层开始往下进行。

RSK：（喘气）自上而下设计！？？！？！？

RCM：更正一下，是自上而下，测试优先设计。坦白地说，我不知道这是不是一个好的规则。只是这次，它帮了我们。所以下次，我会再次尝试它看看会发生什么。

RSK：是的，无论如何，我们仍然还要做些重构。ball 变量只是 scoreForFrame 和它的附属方法的一个私有的迭代器（iterator）。它们都应当被移到另外一个对象中去。

RCM：哦，是的，就是你所说的 scorer 对象。终究还是你对了。我们来完成这项工作。

RSK：（抢过键盘，进行了几个小规模的更改，期间进行了一些测试……）

```
//Game.java ------------------------------------------------
public class Game
{
  public int score()
  {
      return scoreForFrame(getCurrentFrame() - 1);
  }

  public int getCurrentFrame()
  {
      return itsCurrentFrame;
  }

  public void add(int pins)
  {
      itsScorer.addThrow(pins);
      itsScore += pins;
      adjustCurrentFrame(pins);
  }

  private void adjustCurrentFrame(int pins)
  {
      if (firstThrowInFrame == ture)
      {
          if (pins == 10)     // strike
              itsCurrentFrame++;
          else
              firstThrowInFrame = false;
      }
      else
      {
          firstThrowInFrame = true;
          itsCurrentFrame++;
```

```
        }
        itsCurrentFrame = Math.min(11, itsCurrentFrame);
    }

    public int scoreForFrame(int theFrame)
    {
        return itsScorer.scoreForFrame(theFrame);
    }

    private int itsScore = 0;
    private int itsCurrentFrame = 0;
    private boolean firstThrowInFrame = true;
    private Scorer itsScorer = new Scorer();
}

//Scorer.java ------------------------------------------------
public class Scorer
{
    public void addThrow(int pins)
    {
        itsThrows[itsCurrentThrow++] = pins;
    }

    public int scoreForFrame(int theFrame)
    {
        ball = 0;
        int score=0;
        for (int currentFrame = 0;
             currentFrame < theFrame;
               currentFrame++)
        {
            if (strike())
            {
                score += 10 + nextTwoBallsForStrike();
              ball++;
            }
            else if ( spare() )
            {
                score += 10 + nextBallForSpare();
                ball+=2;
            }
            else
            {
                score += twoBallsInFrame();
                ball+=2;
            }
        }

        return score;
    }

    private boolean strike()
    {
        return itsThrows[ball] == 10;
    }

    private boolean spare()
    {
        return (itsThrows[ball] + itsThrows[ball+1]) == 10;
    }
```

```
    private int nextTwoBallsForStrike()
    {
        return itsThrows[ball+1] + itsThrows[ball+2];
    }

    private int nextBallForSpare()
    {
        return itsThrows[ball+2];
    }

    private int twoBallsInFrame()
    {
        return itsThrows[ball] + itsThrows[ball+1];
    }

    private int ball;
    private int[] itsThrows = new int[21];
    private int itsCurrentThrow = 0;
}
```

RSK：好多了。现在 Game 只知晓 Frame，而 Scorer 对象只计算得分。完全符合单一职责原则！

RCM：不管怎样，确实好多了。你注意到 itsScore 变量已经不再使用了吗？

RSK：哈，你说的对。去掉它。（极为高兴地开始进行删除。）

```
Public void add(int pins)
{
   itsScorer.addThrow(pins);
   adjustCurrentFrame(pins);
}
```

RSK：不错。现在，我们可以整理 adjustCurrentFrame 了吗？

RCM：可以。我们来看看它。

```
private void adjustCurrentFrame(int pins)
{
      if (firstThrowInFrame == ture)
      {
          if (pins == 10)     // strike
              itsCurrentFrame++;
          else
              firstThrowInFrame = false;
      }
      else
      {
          firstThrowInFrame = true;
          itsCurrentFrame++;
      }
      itsCurrentFrame = Math.min(11, itsCurrentFrame);
}
```

RCM：好，首先把增量操作移到一个单独的方法中，并在该方法中把轮限定到 11。（呵，我还是不喜欢那个 11。）

RSK：Bob，11 意味着游戏的结束。

RCM：是的。呵。（抓过键盘，做了些变动，其间也进行了一些测试。）

```
private void adjustCurrentFrame(int pins)
```

```
{
        if (firstThrowInFrame == ture)
        {
            if (pins == 10)     // strike
                advanceFrame();
            else
                firstThrowInFrame = false;
        }
        else
        {
            firstThrowInFrame = true;
            advanceFrame();
        }
}

private void advanceFrame()
{
        itsCurrentFrame = Math.min(11, itsCurrentFrame + 1);
}
```

RCM：好了一点。现在我们来把关于全中情况的判断取出来作为一个独立的方法。（做了几步改进，每次都运行测试。）

```
private void adjustCurrentFrame(int pins)
{
        if (firstThrowInFrame == ture)
        {
            if (adjustFrameForStrike(pins) == false)
                firstThrowInFrame = false;
        }
        else
        {
            firstThrowInFrame = true;
            advanceFrame();
        }
}

private boolean adjustFrameForStrike(int pins)
{
   if (pins == 10)
   {
       advanceFrame();
       return ture;
   }
   return false;
}
```

RCM：真是不错，现在，来看看那个 11。

RSK：你确实不喜欢它，是吗？

RCM：是的，看一下 score()函数。

```
public int score()
{
   return scoreForFrame(getCurrentFrame() - 1);
}
```

RCM：这个-1 怪怪的。我们只在该方法中使用了 getCurrentFrame，可是我们还得调整它的返回值。

RSK：该死，你是对的。我们在这上面反复多少次了？

RCM：太多了。但是现在好了。代码希望 itsCurrentFrame 表示的是最后一次投掷所在轮，而不是将要进行的投掷所在轮。

RSK：唉，这样会破坏很多测试用例。

RCM：事实上，我觉得可以把 getCurrentFrame 从所有的测试用例中去掉，并把 getCurrentFrame 函数本身也去掉。其实它是不会被用到的。

RSK：好的，我明白你的意思。我来完成它。这就像解救一匹跛马的痛苦一样。（抓过键盘。）

```
//Game.java-----------------------------------------
 public int score()
 {
  return scoreForFrame(itsCurrentFrame);
 }

 private void advanceFrame()
 {
  itsCurrentFrame = Math.min(10, itsCurrentFrame + 1);
 }
```

RCM：哦。你是想说我们一直为之困扰，而要做的就是把限制从 11 改到 10，并且移走-1。天哪！

RSK：是的，Bob 大叔，我们实在是不值得为之苦恼。

RCM：adjustFrameForStrike()有些表意不清。我想修正一下。这样如何？

```
Private void adjustCurrentFrame(int pins)
{
   if ((firstThrowInFrame && pins == 10) ||
         (!firstThrowInFrame))
      advanceFrame();
   else
      firstThrowInFrame = false;
}
```

RSK：好主意，也通过了测试，但是我不喜欢那个长长的 if 语句。这样如何？

```
Private void adjustCurrentFrame(int pins)
{
   if (strike(pins) || !firstThrowInFrame)
      advanceFrame();
   else
      firstThrowInFrame = false;
}

private boolean strike(int pins)
{
   return (firstThrowInFrame && pins == 10);
}
```

RCM：是的，很好。甚至可以更进一步。

```
Private void adjustCurrentFrame(int pins)
{
   if (lastBallInFrame(pins))
      advanceFrame();
   else
      firstThrowInFrame = false;
}
```

```
private boolean lastBallInFrame(int pins)
{
   return strike(pins) || !firstThrowInFrame;
}
```

RSK：好极了！

RCM：好，看起来好像已经完成了。我们来浏览一下整个程序，看它是否尽可能地简单、尽可能易于交流。

```
//Game.java ------------------------------------------------
public class Game
{
   public int score()
   {
       return scoreForFrame(itsCurrentFrame);
   }

   public void add(int pins)
   {
       itsScorer.addThrow(pins);
       adjustCurrentFrame(pins);
   }

   private void adjustCurrentFrame(int pins)
   {
    if (lastBallInFrame(pins))
      advanceFrame();
    else
      firstThrowInFrame = false;
    }

  private boolean lastBallInFrame(int pins)
  {
    return strike(pins) || !firstThrowInFrame;
  }

  private boolean strike(int pins)
  {
    return (firstThrowInFrame && pins == 10);
  }

  private void advanceFrame()
   {
       itsCurrentFrame = Math.min(10, itsCurrentFrame + 1);
   }

   public int scoreForFrame(int theFrame)
   {
       return itsScorer.scoreForFrame(theFrame);
   }

   private int itsCurrentFrame = 0;
   private boolean firstThrowInFrame = true;
   private Scorer itsScorer = new Scorer();
}

//Scorer.java ----------------------------------------------
public class Scorer
{
```

```
public void addThrow(int pins)
{
    itsThrows[itsCurrentThrow++] = pins;
}

public int scoreForFrame(int theFrame)
{
    ball = 0;
    int score=0;
    for (int currentFrame = 0;
         currentFrame < theFrame;
           currentFrame++)
    {
        if (strike())
        {
            score += 10 + nextTwoBallsForStrike();
    ball++;
        }
        else if ( spare() )
        {
            score += 10 + nextBallForSpare();
            ball+=2;
        }
        else
        {
            score += twoBallsInFrame();
            ball+=2;
        }
    }

    return score;
}

private boolean strike()
{
    return itsThrows[ball] == 10;
}

private boolean spare()
{
    return (itsThrows[ball] + itsThrows[ball+1]) == 10;
}

private int nextTwoBallsForStrike()
{
    return itsThrows[ball+1] + itsThrows[ball+2];
}

private int nextBallForSpare()
{
    return itsThrows[ball+2];
}

private int twoBallsInFrame()
{
    return itsThrows[ball] + itsThrows[ball+1];
}

private int ball;
private int[] itsThrows = new int[21];
```

```
    private int itsCurrentThrow = 0;
}
```

RCM：行，看起来确实不错。我想不出来还有什么需要做的。

RSK：是的，确实不错。为了保险起见，我们来查看一下测试代码。

```
//TestGame.java ------------------------------------------------------
import junit.framework.*;

public class TestGame extends TestCase
{
   public TestGame(String name)
   {
       super(name);
   }

   private Game g;

   public void setUp()
   {
       g = new Game();
   }

   public void testTwoThrowsNoMark()
   {
       g.add(5);
       g.add(4);
       assertEquals(9, g.score());
   }

   public void testFourThrowsNoMark()
   {
       g.add(5);
       g.add(4);
       g.add(7);
       g.add(2);
       assertEquals(18, g.score());
       assertEquals(9,  g.scoreForFrame(1));
       assertEquals(18, g.scoreForFrame(2));
   }

   public void testSimpleSpare()
   {
       g.add(3);
       g.add(7);
       g.add(3);
       assertEquals(13, g.scoreForFrame(1));
   }

   public void testSimpleFrameAfterSpare()
   {
       g.add(3);
       g.add(7);
       g.add(3);
       g.add(2);
       assertEquals(13, g.scoreForFrame(1));
       assertEquals(18, g.scoreForFrame(2));
       assertEquals(18, g.score());
   }
```

```
public void testSimpleStrike()
{
    g.add(10);
    g.add(3);
    g.add(6);
    assertEquals(19, g.scoreForFrame(1));
    assertEquals(28, g.score());
}

public void testPerfectGame()
{
    for (int i=0; i<12; i++)
    {
        g.add(10);
    }
    assertEquals(300, g.score());
}

public void testEndOfArray()
{
    for (int i=0; i<9; i++)
    {
        g.add(0);
        g.add(0);
    }
    g.add(2);
    g.add(8); // 10th frame spare
    g.add(10); // Strike in last position of array.
    assertEquals(20, g.score());
}

public void testSampleGame()
{
    g.add(1);
    g.add(4);
    g.add(4);
    g.add(5);
    g.add(6);
    g.add(4);
    g.add(5);
    g.add(5);
    g.add(10);
    g.add(0);
    g.add(1);
    g.add(7);
    g.add(3);
    g.add(6);
    g.add(4);
    g.add(10);
    g.add(2);
    g.add(8);
    g.add(6);
    assertEquals(133, g.score());
}

public void testHeartBreak()
{
    for (int i=0; i<11; i++)
        g.add(10);
    g.add(9);
```

```
        assertEquals(299, g.score());
    }

    public void testTenthFrameSpare()
    {
        for (int i=0; i<9; i++)
            g.add(10);
        g.add(9);
        g.add(1);
        g.add(1);
        assertEquals(270, g.score());
    }
}
```

RSK：几乎覆盖了所有的情况。你还能想出其他有意义的测试用例吗？

RCM：想不出来了，我认为这是一套完整的测试用例集。从中去掉任何一个都不好。

RSK：那我们就完成了。

RCM：我也这么认为。非常感谢你的帮助。

RSK：别客气，这很有趣。

6.2 结　　论

完成本章后，我把它发布在 Object Mentor 的 Web 站点上。[①]许多人阅读后给出了自己的意见。有些人认为这篇文章不好，因为其中几乎没有涉及面向对象设计方面的任何内容。我认为这种回应很有趣。必须在每一个应用、每一个程序中都要进行面向对象的设计吗？本例就是一个不太需要面向对象设计的情形。事实上，仅有 Scorer 类稍微有一点面向对象的味道，不过那也只是一个简单的分割（partitioning），而不是真正的 OOD（面向对象的设计）。

另有一些人认为确实应该有 Frame 类。有人竟然创建了一个包含 Frame 类的程序版本，该程序比上面所看到的要大得多，也复杂得多。

一些人觉得我们对 UML 有失公正。毕竟，在开始前我们没有做一个完整的设计。餐巾纸背面的有趣的小 UML 图（见图 6.2）不是一个完整的设计。其中没有包括序列图（sequence diagram）。我认为这种看法更加奇怪。就我而言，即使在图 6.2 中加入序列图，也不会促使我们放弃 Throw 类和 Frame 类。事实上，那样做反而会使我们觉得这些类是必需的。

图示是不需要的吗？当然不是。嗯，实际上，对于某些我所碰到的情形是不需要的。就本章中的程序而言，图示就没有任何帮助。它们甚至分散了我们的注意力。如果遵循这些图示，所得到的程序就会具有很多不必要的复杂性。你也许会说同样也会得到一个非常易于维护的程序，但是我不同意这种说法。我们刚刚浏览的程序是因为易于理解所以才易于维护的，其中没有会导致该程序僵化（rigid）或者脆弱（fragile）的不当依赖关系。

所以，是的，图示有时是不需要的。何时不需要呢？在创建了它们而没有验证它们的代码就打算去遵循它们时，图示就是无益的。画一幅图来探究一个想法是没有错的。然而，画一幅图后，不应该假定该图就是相关任务的最好设计。你会发现最好的设计是在你首先编写测试，一小步一小步前进时逐渐形成的。

① http://www.objectmentor.com。

保龄球规则概述

保龄球是一种比赛，比赛者把一个哈蜜瓜大小的球顺着一条窄窄的球道朝 10 个木瓶投掷。目标是要在每次投掷中击倒尽可能多的木瓶。

一局比赛由 10 轮组成。在每轮的开始，10 个木瓶都是竖立摆放的。比赛者可以投掷两次来尝试击倒所有的木瓶。

如果比赛者在第一次投掷中就击倒了所有的木瓶，则称之为“全中”，并且本轮结束。

如果比赛者在第一次投掷中没有击倒所有的木瓶，但在第二次投掷中成功的击倒了所有剩余的木瓶，则称之为“补中”。

一轮中第二次投掷后，即使还有没有被击倒的木瓶，本轮也宣告结束。

全中轮的记分规则为：10，加上接下来的两次投掷击倒的木瓶数，再加上前一轮的得分。

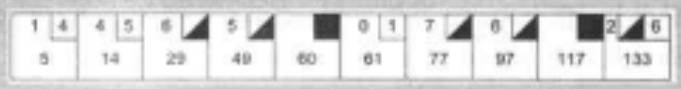
补中轮的记分规则为：10，加上接下来的一次投掷击倒的木瓶数，再加上前一轮的得分。

其他轮的记分规则为：本轮中两次投掷所击倒的木瓶数，加上前一轮的得分。

如果第 10 轮为全中，那么比赛者可以再多投掷两次，以完成对全中的记分。

同样，如果第 10 轮为补中，那么比赛者可以再多投掷一次，以完成对补中的记分。

因此，第 10 轮可以包含 3 次掷球而不是 2 次。



上面的记分卡展示了一场虽然不太精彩，但具有代表性的比赛的得分情况。

第 1 轮中，比赛者第一次投掷击倒了 1 个木瓶，第二次投掷又击倒了 4 个。于是第一轮的得分是 5。

第 2 轮中，比赛者第一次投掷击倒了 4 个木瓶，第二次投掷又击倒了 5 个。本轮中共击倒了 9 个木瓶，再加上前一轮的得分，本轮的得分是 14。

第 3 轮中，比赛者第一次投掷击倒了 6 个木瓶，第二次投掷又击倒了剩余的所有木瓶，因而是一个补中。只有到下一次掷球后才能计算本轮的得分。

第 4 轮中，比赛者第一次投掷击倒了 5 个木瓶。此时可以完成第 3 轮的记分。第 3 轮的得分为 10，加上第 2 轮的得分（14），再加上第 4 轮中第一次投掷击倒的木瓶数（5），结果是 29。第 4 轮的最后一次掷球为补中。

第 5 轮为全中。此时计算第 4 轮的得分为：29+10+10 = 49。

第 6 轮的成绩很不理想。第一个球滚入了球道旁的槽中，没有击倒任何木瓶。第二个球仅击倒了一个木瓶。第 5 轮中全中的得分为：49+10+0+1 = 60。

其余的部分可以自行计算。

第II部分 敏捷设计

如果敏捷性（Agility）是指以微小增量的方式构建软件，那么究竟如何去设计软件呢？又如何去确保软件具有灵活性、可维护性以及可重用性的良好结构呢？如果以微小增量的方式构建软件，难道不是打着重构的旗号，而实际上却导致了许多无用的代码碎片和返工吗？难道不会忽视全局视图吗？

在敏捷团队中，全局视图和软件一起演化。在每次迭代中，团队改进系统设计，使设计尽可能适合于当前系统。团队不会花费许多时间去预测未来的需求和需要，也不会试图在今天就构建一些基础结构去支撑那些他们认为明天才会需要的特性。他们更愿意关注当前的系统结构，并使它尽可能地好。

拙劣设计的症状

我们如何知道软件设计的优劣呢？第 7 章中列举并描述了拙劣设计的症状，演示了那些症状如何在一个软件项目中累积，并描述了如何去避免它们。

这些症状定义如下：

- 僵化性（Rigidity）：设计难以改变。
- 脆弱性（Fragility）：设计易于遭到破坏。
- 牢固性（Immobility）：设计难以重用。
- 粘滞性（Viscosity）：难以做正确的事情。
- 不必要的复杂性（Needless Complexity）：过分设计。
- 不必要的重复（Needless Repetition）：滥用鼠标[①]。
- 晦涩性（Opacity）：混乱的表达。

这些症状在本质上和代码的“臭味”（smell）相似[②]，但是它们所处的层次稍高一些。它们是遍及整个软件结构的臭味，而不仅仅是一小段代码。

原则

在本部分的其余章节中，描述了一些面向对象设计的原则，这些原则有助于开发人员消除设计

① 指滥用鼠标进行拷贝、粘贴——译者注。
② [Fowler99]。

中的臭味，并为当前的特性集构建出最好的设计。

这些原则如下：面向对象原则

- 单一职责原则（The Single Responsibility Principle，简称 SRP）
- 开放－封闭原则（The Open-Close Principle，简称 OCP）
- Liskov 替换原则（The Liskov Substitution Principle，简称 LSP）
- 依赖倒置原则（The Dependency Inversion Principle，简称 DIP）
- 接口隔离原则（The Interface Segregation Interface，简称 ISP）

这些原则是数十年软件工程经验来之不易的成果。它们不是某一个人的成果，而是许许多多软件开发人员和研究人员思想和著作的结晶。虽然在此把它们表述为面向对象设计的原则，但是事实上它们只是软件工程中一直存在的原则的特例而已。

臭味和原则

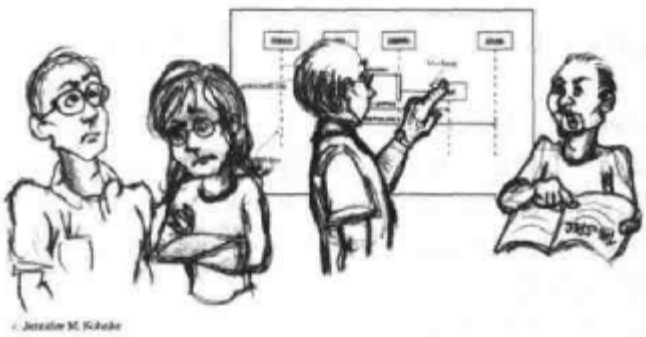
设计中的臭味是一种症状，是可以主观（如果不能客观的话）进行度量的。这些臭味常常是由于违反了这些原则中的一个或者多个而导致的。例如，僵化性的臭味常常是由于对开放－封闭原则（OCP）不够关注的结果。

敏捷团队应用这些原则来除去臭味。当没有臭味时，他们不会应用这些原则。仅仅因为是一个原则就无条件的去遵循它的做法是错误的。这些原则不是可以随意在系统中到处喷洒的香水。过分遵循这些原则会导致不必要的复杂性（Needless Complexity）的设计臭味。

参考文献

1. Martin, Fowler. *Refactoring*. Addison-Wesley. 1999.

第 7 章　什么是敏捷设计



“在按照我的理解方式审查了软件开发的生命周期后，我得出一个结论：实际上满足工程设计标准的惟一软件文档，就是源代码清单。”

——Jack Reeves

1992 年，Jack Reeves 在 *C++ Journal* 杂志上撰写了一篇题为“什么是软件设计？”的开创性的论文。[①]在这篇文章中，Reeves 认为软件系统的源代码是它的主要设计文档。用来描绘（represent）源代码的图示只是设计的附属物而不是设计本身。结果表明，Jack 的论文是敏捷开发的先驱。

在随后的内容中，我们会经常谈到“设计”。你不应该认为设计就是一组和代码分离的 UML 图。一组 UML 图也许描绘了设计的一些部分，但是它不是设计。软件项目的设计是一个抽象的概念。它和程序的概括形状（shape）、结构以及每一个模块、类和方法的详细形状和结构有关。可以使用许多不同的媒介（media）去描绘它，但是它最终体现为源代码。最后，源代码就是设计。

7.1　软件出了什么错

如果幸运，你会在项目开始时就有了想得到的系统的清晰图像。系统的设计是存在于你头脑中的一副至关重要的图像。如果更幸运一点，在首次发布（release）时，设计依然保持清楚。

接着，事情开始变糟。软件像一片坏面包一样开始腐化。随着时间的流失，腐化蔓延、增长。丑陋腐烂的痛处和疖子在代码中累积，使它变得越来越难以维护。最后，即使仅仅进行最简单的更改，也需要花费巨大的努力，以至于开发人员和一线（front-line）管理人员强烈要求重新设计。

这样的重新设计很少会成功。虽然设计人员开始时出于好意，但是他们发现自己正朝一个移动的目标射击。老系统不断地发展、变化，而新的设计必须得跟上这些变化。这样，甚至在第一次发布前，新的设计中就累积了很多的瑕疵和弊病。

① [Reeves92] 这是一篇伟大的论文，强烈推荐。我已把它列入了本书的附录 D。

7.2　设计的臭味——腐化软件的气味

当软件出现下面任何一种气味时，就表明软件正在腐化。

- 僵化性（Rigidity）：很难对系统进行改动，因为每个改动都会迫使许多对系统其他部分的其他改动。
- 脆弱性（Fragility）：对系统的改动会导致系统中和改动的地方在概念上无关的许多地方出现问题。
- 牢固性（Immobility）：很难解开系统的纠结，使之成为一些可在其他系统中重用的组件。
- 粘滞性（Viscosity）：做正确的事情比做错误的事情要困难。
- 不必要的复杂性（Needless Complexity）：设计中包含有不具任何直接好处的基础结构。
- 不必要的重复（Needless Repetition）：设计中包含有重复的结构，而该重复的结构本可以使用单一的抽象进行统一。
- 晦涩性（Opacity）：很难阅读、理解。没有很好地表现出意图。

1．僵化性

僵化性是指难以对软件进行改动，即使是简单的改动。如果单一的改动会导致有依赖关系的模块中的连锁改动，那么设计就是僵化的。必须要改动的模块越多，设计就越僵化。

大部分的开发人员都以这样或者那样的方式遇到过这种情况。他们会被要求进行一个看起来简单的改动。他们看了看这个改动并对所需的工作做出了一个合理的估算。但是过了一会儿，当他们实际进行改动时，会发现有许多改动带来的影响自己并没有预测到。他们发现自己要在庞大的代码中搜寻这个变动，并且要更改的模块数目也远远超出最初估算。最后，改动所花费的时间要远比初始估算长。当问他们为何估算得如此不准确时，他们会重复软件开发人员惯用的悲叹，“它比我想像的要复杂得多！”

2．脆弱性

脆弱性是指，在进行一个改动时，程序的许多地方就可能出现问题。常常是，出现新问题的地方与改动的地方并没有概念上的关联。要修正这些问题就又会引出更多的问题，从而使开发团队就像一只不停追逐自己尾巴的狗一样（忙得团团转）。

随着模块脆弱性的增加，改动会引出意想不到的问题的可能性就越来越大。这看起来很荒谬，但是这样的模块是非常常见的。这些模块需要不断地修补——它们从来不会被从错误列表中去掉，开发人员知道需要对它们进行重新设计（但是谁都不愿意去面对重新设计中的难以琢磨性），你越是修正它们，它们就变得越糟。

3．牢固性

牢固性是指，设计中包含了对其他系统有用的部分，但是要把这些部分从系统中分离出来所需要的努力和风险是巨大的。这是一件令人遗憾的事，但却是非常常见的事情。

4．粘滞性

粘滞性有两种表现形式：软件的粘滞性和环境的粘滞性。

当面临一个改动时，开发人员常常发现会有多种改动的方法。其中，一些方法会保持设计；而

另外一些会破坏设计（也就是生硬的手法）。当那些可以保持系统设计的方法比那些生硬手法更难应用时，就表明设计具有高的粘滞性。做错误的事情是容易的，但是做正确的事情却很难。我们希望在软件设计中，可以容易地进行那些保持设计的变动。

当开发环境迟钝、低效时，就会产生环境的粘滞性。例如，如果编译所花费的时间很长，那么开发人员就会被引诱去做不会导致大规模重编译的改动，即使那些改动不再保持设计。如果源代码控制系统需要几个小时去拆入（check in）仅仅几个文件，那么开发人员就会被引诱去做那些需要尽可能少拆入的改动，而不管改动是否会保持设计。

无论项目具有哪种粘滞性，都很难保持项目中的软件设计。我们希望创建易于保持设计的系统和项目环境。

5．不必要的复杂性

如果设计中包含有当前没有用的组成部分，它就含有不必要的复杂性。当开发人员预测需求的变化，并在软件中放置了处理那些潜在变化的代码时，常常会出现这种情况。起初，这样做看起来像是一件好事。毕竟，为将来的变化做准备会保持代码的灵活性，并且可以避免以后再进行痛苦的改动。

糟糕的是，结果常常正好相反。为过多的可能性做准备，致使设计中含有绝不会用到的结构，从而变得混乱。一些准备也许会带来回报，但是更多的不会。期间，设计背负着这些不会用到的部分，使软件变得复杂，并且难以理解。

6．不必要的重复

剪切（cut）和粘贴（paste）也许是有用的文本编辑（text-editing）操作，但是它们却是灾难性的代码编辑（code-editing）操作。时常，软件系统都是构建于众多的重复代码片断之上。例如：

Ralph 需要编写一些完成某项功能的代码。他浏览了一下他认为可能会完成类似工作的其他代码，并找到了一块合适的代码。他将那块代码拷贝到自己的模块中，并做了适当的修改。

Ralph 并不知道，他用鼠标获取的代码是由 Todd 放置在那里的，而 Todd 是从 Lilly 编写的模块中获取的。Lilly 是第一个完成这项功能的，但是她认识到完成这项功能和完成另一项功能非常类似。她从别处找到了一些完成另一项功能的代码，剪切、拷贝到她的模块中并做了必要的修改。

当同样的代码以稍微不同的形式一再出现时，就表示开发人员忽视了抽象。对于他们来说，发现所有的重复并通过适当的抽象去消除它们的做法可能没有高的优先级别，但是这样做非常有助于使系统更加易于理解和维护。

当系统中有重复的代码时，对系统进行改动会变得困难。在一个重复的代码体中发现的错误必须要在每个重复体中一一修正。不过，由于每个重复体之间都有细微的差别，所以修正的方式也不总是相同的。

7．晦涩性

晦涩性是指模块难以理解。代码可以用清晰、富有表现力的方式编写，或者可以用晦涩、费解的方式编写。代码随着时间而演化，往往会变得越来越晦涩。为了使代码的晦涩性保持最低，就需要持续地保持代码清晰并富有表现力。

当开发人员最初编写一个模块时，代码对于他们来说看起来也许是清晰的。这是由于他们使自己专注于代码的编写，并且他们对于代码非常的熟悉。在熟悉减退以后，他们或许会回过头来再去看那个模块，并想知道他们怎么会编写如此糟糕的代码。为了防止这种情况的发生，开发人员必须要站在代码阅读者的位置，共同努力对他们的代码进行重构，这样代码的阅读者就可以理解代码。他们的代码也需要被其他人评审。

7.2.1　什么激发了软件的腐化

在非敏捷环境中，由于需求没有按照初始设计预见的方式进行变化，从而导致了设计的退化。通常，改动都很急迫，并且进行改动的开发人员对于原始的设计思路并不熟悉。因而，虽然对设计的改动可以工作，但是它却以某种方式违反了原始的设计。随着改动的不断进行，这些违反渐渐地积累，设计开始出现臭味。

然而，我们不能因为设计的退化而责怪需求的变化。作为软件开发人员，我们对于需求变化有非常好的了解。事实上，我们中的大多数人都认识到需求是项目中最不稳定的要素。如果我们的设计由于持续、大量的需求变化而失败，那就表明我们的设计和实践本身是有缺陷的。我们必须要设法找到一种方法，使得设计对于这种变化具有弹性，并且应用一些实践来防止设计腐化。

7.2.2　敏捷团队不允许软件腐化

敏捷团队依靠变化来获取活力。团队几乎不进行预先（up-front）设计，因此，不需要一个成熟的初始设计。他们更愿意保持系统设计尽可能的干净、简单，并使用许多单元测试和验收测试作为支援。这保持了设计的灵活性、易于理解性。团队利用这种灵活性，持续地改进设计，以便于每次迭代结束所生成的系统都具有最适合于那次迭代中需求的设计。

7.3　“Copy”程序

观看一个设计的腐化过程会有助于阐明上述观点。比如说，你的老板周一一大早就来找你，并要求你编写一个从键盘读入字符并输出到打印机的程序。经过一番快速思考后，你断定所需的代码不会超过 10 行。设计和编码所需要的时间会远远小于一个小时。考虑到交叉功能会议、质量教育会议、日常的小组进度会议以及当前 3 个正在处理的难题，要完成这个程序应该要花费你大约一周的时间——如果你下班后仍坚持工作的话。不过，你总是把估算值乘以 3。

“需要 3 周时间。”你告诉你的老板。老板哼着走开了，把任务留给了你。

1．初始设计

现在距过程（process）评审会议开始还有一小段时间，所以你决定为那个程序做一个设计。使用结构化的设计方法，你想出了图 7.1 中所示的结构图。

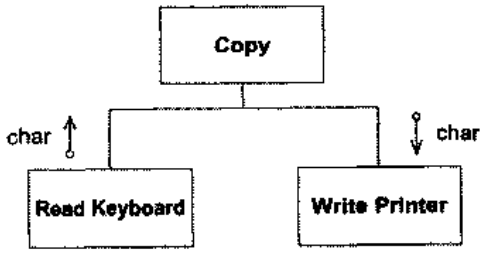


图 7.1　Copy 程序结构图

应用程序中有 3 个模块，或者子程序。Copy 模块调用另外两个模块。Copy 程序从 Read Keybord 模块中获取字符，并把字符传递给 Write Printer 模块。

你看了看设计，觉得不错。笑着离开办公室去参加评审会。至少，你能够在会上睡上一会儿。

周二，为了能够完成 Copy 程序，你提前了一点来到办公室。糟糕的是，需要处理的难题之一在昨天晚上发作了，你必须要到实验室去帮助调试一个问题。在午饭（你在下午 3 点才吃上）的间歇，你终于输入了 Copy 程序的代码。如程序 7.1 所示。

程序 7.1　Copy 程序

```
void Copy()
{
int c;
while ((c=RdKbd()) != EOF))
   WrtPrt(c);
}
```

当你正准备保存这个程序时，才想到已经延误了一个质量会议。你知道这是一个重要的会议，会议是有关零缺陷的重要性的。因此，你狼吞虎咽地吃下三明治和可乐，奔向会场。

周三，你又提前到来，并且这次好像没有任何问题发生。你打开 Copy 程序的源代码，开始编译它。你瞧！首次编译就没有错误！运气真好，因为老板让你去参加一个事先没有安排的关于激光打印机硒鼓保存必要性的会议。

周四，北卡罗来纳州洛矶山城的一个技术服务人员向你询问有关系统中一个比较难懂的组件的远程调试和错误日志命令方面的内容，经过 4 个小时的电话敷衍后，你得意地一笑，接着开始测试 Copy 程序。第一次，它就运转起来了！运气同样不错，因为你的一个新合作者刚刚删除了服务器上主要的源代码目录，你必须得找到最新的磁带备份并恢复它。最后的一次完整备份是在三个月前进行的，并且有 94 次增量备份需要在其上重建。

周五，没有任何预先安排的工作。太好了，因为一整天都可以用来把 Copy 程序成功地放进源代码控制系统中。

当然，你的程序非常的成功，并且被部署在公司各处。你作为一流程序员的名声再一次得到印证，你由于成功带来的赞誉而洋洋得意。幸运的话，也许今年你实际上只需要产出 30 行代码！

2．需求在变化

几个月后，老板来找你，说有时会希望 Copy 程序能从纸带读入机中读入信息。你咬牙切齿、翻着白眼。你想知道为何人们总是改变需求。你的程序不是为纸带读入机设计的！你警告老板像这样的改变会破坏程序的优雅性。不过，老板很固执，他说用户有时确实需要从纸带读入机中读取字符。

你叹了一口气，开始计划修改方案。你想在 Copy 函数中添加一个 boolean 变量。如果变量值为 true，那么就从纸带读入机中读取信息；如果变量值为 false，就像以前一样从键盘读取信息。糟糕的是，现在已有许多其他程序正在使用 Copy 程序，你不能改变 Copy 程序的接口。改变接口会导致长时间的重新编译和重新测试。单单系统测试工程师就会痛恨你，更别提配置控制组的 7 个家伙了。并且过程控制部门会用专门的一天时间来对所有调用了 Copy 的模块进行各种各样的代码评审。

看来，不能采用改变接口的方法。那么，如何才能使 Copy 程序知道它必须要从纸带读入机读取信息呢？你当然会使用一个全局变量！也会使用最好、最有用的 C 语言特性——?:操作符！结果如程序 7.2 所示。

程序 7.2 Copy 程序的第一次修改结果

```
bool ptFlag = false;
// remember to reset this flag
void Copy()
{
int c;
while ((c=(ptflag ? Rdpt() : RdKbd())) != EOF))
    WrtPrt(c);
}
```

想让 Copy 从纸带读入机读入信息的调用者必须要把 ptFlag 设置为 true，然后再调用 Copy 时，它就会正确地从纸带读入机中读入信息。一旦 Copy 调用返回，调用者必须要重新设置 ptFlag，否则接下来的调用者就会错误地从纸带读入机而不是键盘读入信息。为了提醒程序员记得重置这个标志，你增加了一个适当的注释。

同样，你的程序一发布，就获得了好评。它甚至比以前更成功，一大群渴望的程序员在等待机会去使用它。生活是美好的。

3. 得寸进尺

几周后，你的老板（尽管在这几个月内进行了 3 次公司范围内的重组，但他仍是你的老板）告诉你，客户有时希望 Copy 程序可以输出到纸带穿孔机上。

客户！他们总是毁坏你的设计。如果没有客户，编写软件会变得容易得多。

你告诉老板不断地变更会对你的设计的优雅性造成极度的负面影响。你警告老板如果变更继续以这样可怕的速度进行，那么在年底前，软件就会变得难以维护。老板心照不宣地点点头，接着告诉你无论如何都要进行这次改动。

这次设计的改动和上一次相似。只不过需要另外一个全局变量和?:操作符！程序 7.3 展示了你努力后的结果。

程序 7.3

```
bool ptFlag = false;
bool punchFlag = false;
// remember to reset these flags
void Copy()
{
int c;
while ((c=(ptflag ? Rdpt() : RdKbd())) != EOF))
    punchFlag ? WrtPunch© : WrtPrt(c);
}
```

尤其让你感到骄傲的是，你还记得去修改注释。可是，你对程序的结构开始变得摇摇欲坠感到担心。任何对于输入设备的再次变更肯定会迫使你对 while 循环的条件判断进行彻底的重新组织。也许你该考虑重新寻找工作了……

4. 期望变化

请读者自己去判断上面所说的有多少是讽刺性的夸大之词。故事的要点是要说明，在变化面前，程序的设计退化的速度是多么的快。Copy 程序的原始设计是简单并且优雅的。但是仅仅经历了两次变更，它就已经表现出了僵化性、脆弱性、牢固性、不必要的复杂性、不必要的重复及晦涩性的症

状。这种趋向肯定会继续下去，程序将会变得混乱不堪。

我们可以坐下来去指责变化。我们可以抱怨程序对于最初的要求是设计良好的，是因为后来对要求的改变导致了设计的退化。然而，这种抱怨忽视了软件开发中最重要的事实之一：需求总是在变化。

记住，在大多数软件项目中最不稳定的东西就是需求。需求处在一个持续变动的状态之中。这是我们作为开发人员必须得接受的事实！我们生存在一个需求不断变化的世界中，我们的工作是要保证我们的软件能够经受得住那些变化。如果我们软件的设计由于需求变化了而退化，那么我们就不是敏捷的。

7.3.1　Copy 程序的敏捷设计

使用敏捷开发方法时，一开始编写的代码和程序 7.1 中的完全一样。[1] 在老板要求敏捷开发人员使程序可以从纸带读入机中读取信息时，他们会做出这样的反应：修改设计并使修改后的设计对于那一类需求的变化具有弹性。结果可能有点像程序 7.4。

程序 7.4　Copy 的敏捷版本 2

```
class Reader
{
    public:
        virtual int read() = 0;
};

class KeyboardReader : public Reader
{
    public:
        virtual int read() { return RdKbd();}
}

KeyboardReader GdefaultReader;

void Copy(reader& reader = GdefaultReader)
{
    int c;
    while ((c=reader.read()) != EOF)
        WrtPrt(c);
}
```

在要实现新需求时，团队抓住这次机会去改进设计，以便设计对于将来的同类变化具有弹性，而不是设法去给设计打补丁。从现在开始，无论何时老板要求一种新的输入设备，团队将都能以不导致 Copy 程序退化的方式做出反应。

团队遵循了开放－封闭原则（Open-Closed Principle，简称 OCP），我们将在第 9 章学习它。这个原则指导我们设计出无需修改即可扩展的模块，这正是团队已经完成的。无需修改 Copy 程序就可以使用老板要求的每种新的输入设备。

有人会认为他们仅仅完成了一半的工作。他们在使自己免于不同的输入设备带来的麻烦时，本

① 实际上，测试驱动开发的实践很可能会促使设计足够的灵活，可以无需改动就满足老板的要求。不过，在本例中，我们会忽略这一点。

可以也使自己免于不同的输出设备带来的麻烦。然而，团队实在不知道输出设备是否会变化。现在就添加额外的保护没有任何现实意义。很明显，如果需要这种保护时，以后可以非常容易地添加。因此，实在没有现在就添加的理由。

7.3.2　敏捷开发人员如何知道要做什么

在上面例子中，敏捷开发人员构建了一个抽象类（abstract class）来使他们免于输入设备的变化带来的麻烦。他们如何知道要那样做呢？这和面向对象设计的基本原则中的一个有关。

Copy 程序最初的设计不具灵活性，是因为它的依赖关系的方向。再看一下图 7.1。请注意 Copy 模块直接依赖于 KeyboardReader 和 PrinterWriter。在这个程序中，Copy 模块是一个高层模块，它制定了应用的策略，它知道怎样去拷贝字符。糟糕的是，它也依赖于键盘和打印机的底层细节。因而，当底层细节变化时，高层策略会受到影响。

一旦暴露出了这个不灵活性，敏捷开发人员应该知道从 Copy 模块到输入设备的依赖关系需要被倒置[1]，这样 Copy 模块就不再依赖于输入设备。于是他们就应用 STRATEGY 模式[2]创建了想要的倒置关系。

因此，简而言之，敏捷开发人员知道要做什么，是因为：

（1）他们遵循敏捷实践去发现问题；
（2）他们应用设计原则去诊断问题；并且
（3）他们应用适当的设计模式去解决问题。

软件开发的这三个方面间的相互作用就是设计。

7.4　保持尽可能好的设计

敏捷开发人员致力于保持设计尽可能地适当、干净。这不是一个随便的或者暂时性的承诺。敏捷开发人员不是每几周才清洁他们的设计。而是每天、每小时、甚至每分钟都要保持软件尽可能地干净、简单并富有表现力。他们从来不说，“稍后我们会回来修正它。”他们决不让腐化出现。

敏捷开发人员对待软件设计的态度和外科医生对待消毒过程的态度是一样的。消毒过程使外科手术成为可能。没有它，被感染的风险之高是难以忍受的。敏捷开发人员对于他们的设计有同样的感觉。即使最小的腐化带来的风险也同样高到无法忍受。

设计必须要保持干净、简单，并且由于源代码是设计最重要的表示，所以它同样要保持干净。职业特性要求我们，作为软件开发人员，不能忍受代码腐化。

7.5　结　　论

那么，什么是敏捷设计呢？敏捷设计是一个过程，不是一个事件。它是一个持续的应用原则、模式以及实践来改进软件的结构和可读性的过程。它致力于保持系统设计在任何时间都尽可能得简

① 参见第 11 章“依赖倒置原则（DIP）”。
② 我们会在第 14 章学习 STRATEGY 模式。

单、干净以及富有表现力。

在随后的章节中，我们会研究软件设计的一些原则和模式。在学习它们的时候，请记住，敏捷开发人员不会对一个庞大的预先设计应用那些原则和模式。相反，这些原则和模式被应用在一次次的迭代中，力图使代码以及代码所表达的设计保持干净。

参考文献

1. Reeves, Jack. *What Is Software Design?* C++ Journal, Vol. 2, No. 2. 1992. 亦可参阅 http://www.bleading-edge.com/Publications/C++ Journal/Cpjour2.htm.

第 8 章　单一职责原则（SRP）

只有佛自己应当担负起公布玄妙秘密的职责。

——E. Cobham Brewer，1810－1897
《英语典故字典》，1898.

这条原则曾经在 Tom DeMaro[①]和 Meilir Page-Jones[②]的著作中描述过，并称之为内聚性（cohesion）。他们把内聚性定义为：一个模块的组成元素之间的功能相关性。在本章中，我们稍微改变一下它的含意，把内聚性和引起一个模块或者类改变的作用力联系起来。

8.1　单一职责原则（SRP）

就一个类而言，应该仅有一个引起它变化的原因。

考虑第 6 章中保龄球比赛的例子。在开发它的大部分时间内，Game 类一直具有两个不同的职责。一个职责是跟踪当前轮（frame）的比赛，另一个职责是计算比赛的得分。最后，RCM 和 RSK 把这两个职责分离到两个类中。Game 类保持跟踪每一轮比赛的职责，Scorer 类负责计算比赛的得分。

为何要把这两个职责分离到单独的类中呢？因为每一个职责都是变化的一个轴线（an axis of change）。当需求变化时，该变化会反映为类的职责的变化。如果一个类承担了多于一个的职责，那么引起它变化的原因就会有多个。

如果一个类承担的职责过多，就等于把这些职责耦合在了一起。一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。这种耦合会导致脆弱的（fragile）设计，当变化发生时，设计

① [DeMarco79], p. 310。
② [Page-Jones88], Chapter 6, p.82。

会遭受到意想不到的破坏。

例如，考虑图 8.1 中的设计。Retangle 类具有两个方法，如图所示。一个方法把矩形绘制在屏幕上，另一个方法计算矩形的面积。

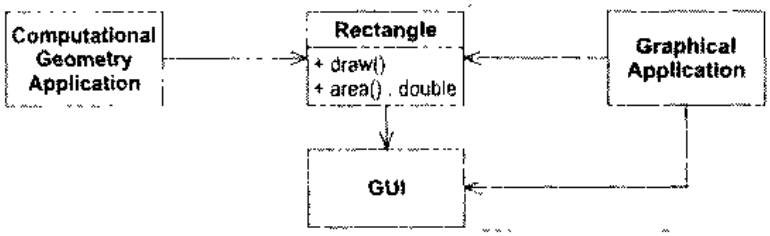


图 8.1　多于一个的职责

有两个不同的应用程序使用 Rectangle 类。一个是有关计算几何学方面的，Rectangle 类会在几何形状计算方面为它提供帮助，它从来不会在屏幕上绘制矩形。另外一个应用程序实质上是有关图形绘制方面的，它可能也会进行一些计算几何学方面的工作，但是它肯定会在屏幕上绘制矩形。

这个设计违反了单一职责原则（SRP）。Rectangle 类具有两个职责。第一个职责提供了一个矩形几何形状的数学模型；第二个职责是把矩形在一个图形用户界面上绘制出来。

对于 SRP 的违反导致了一些严重的问题。首先，我们必须在计算几何应用程序中包含进 GUI 代码。如果这是一个 C++应用程序，就必须要把 GUI 代码链接进来，这会浪费链接时间、编译时间以及内存占用。如果是一个 Java 应用程序，GUI 的.class 文件必须要被部署到目标平台。

其次，如果 GraphicalApplication 的改变由于一些原因导致了 Rectangle 的改变，那么这个改变会迫使我们重新构建、测试以及部署 ComputationalGeometryApplication。如果忘记了这样做，ComputationalGeometryApplication 可能会以不可预测的方式失败。

一个较好的设计是把这两个职责分离到图 8.2 中所示的两个完全不同的类中。这个设计把 Rectangle 类中进行计算的部分移到 GeometryRectangle 类中。现在矩形绘制方式的改变不会对 ComputationalGeometryApplication 造成影响。

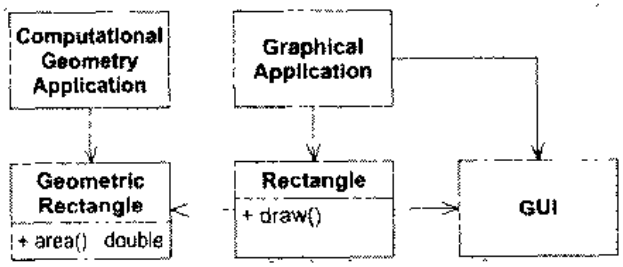


图 8.2　分离的职责

8.1.1　什么是职责

在 SRP 中，我们把职责定义为“变化的原因”（a reason for change）。如果你能够想到多于一个的动机去改变一个类，那么这个类就具有多于一个的职责。有时，我们很难注意到这一点。我们习惯于以组的形式去考虑职责。例如，考虑程序 8.1 中的 Modem 接口。大多数人会认为这个接口看起来非常合理。该接口所声明的 4 个函数确实是调制解调器所具有的功能。

程序 8.1 Modem.java—违反 SRP

```
interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public void recv();
}
```

然而，该接口中却显示出两个职责。第一个职责是连接管理；第二个职责是数据通信。dial 和 hangup 函数进行调制解调器的连接处理，而 send 和 recv 函数进行数据通信。

这两个职责应该被分开吗？这依赖于应用程序变化的方式。如果应用程序的变化会影响连接函数的签名（signature），那么这个设计就具有僵化性的臭味，因为调用 send 和 recv 的类必须要重新编译，部署的次数常常会超过我们希望的次数。在这种情况下，这两个职责应该被分离，如图 8.3 中所示。这样做避免了客户应用程序和这两职责耦合在一起。

另一方面，如果应用程序的变化方式总是导致这两个职责同时变化，那么就不必分离它们。实际上，分离它们就会具有不必要的复杂性的臭味。

在此还有一个推论。变化的轴线仅当变化实际发生时才具有真正的意义。如果没有征兆，那么去应用 SRP，或者任何其他原则都是不明智的。

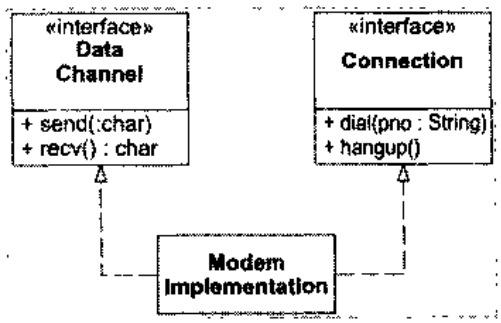


图 8.3 分离的 Modem 接口

8.1.2 分离耦合的职责

请注意，在图 8.3 中，我把两个职责都耦合进了 ModemImplementation 类中。这不是所希望的，但是或许是必要的。常常会有一些和硬件或者操作系统的细节有关的原因，迫使我们把不愿耦合在一起的东西耦合在了一起。然而，对于应用的其余部分来说，通过分离它们的接口我们已经解耦了概念。

我们可以把 ModemImplementation 类看作是一个杂凑物（kludge），或者一个瑕疵。然而，请注意所有的依赖关系都和它无关。谁也不需要依赖于它。除了 main 外，谁也不需要知道它的存在。

8.1.3 持久化

图 8.4 展示了一种常见的违反 SRP 的情形。Employee 类包含了业务规则和对于持久化的控制。这两个职责在大多数情况下决不应该混合在一起。业务规则往往会频繁的变化，而持久化的方式却不会如此频繁的变化，并且变化的原因也是完全不同的。把业务规则和持久化子系统绑定在一起的做法是自讨苦吃。

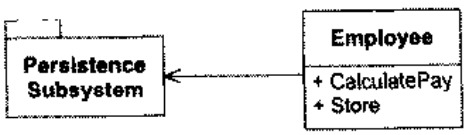


图 8.4 被耦合在一起的持久化职责

幸运的是，正如我们在第 4 章看到的，测试驱动的开发实践常常会远在设计出现臭味之前就迫使我们分离这两个职责。然而，在测试不能迫使职责分离的情况下，僵化性和脆弱性的臭味会变得很

强烈，那么就应该使用 FACADE 或者 PROXY 模式对设计进行重构，分离这两个职责。

8.2 结　　论

SRP 是所有原则中最简单的之一，也是最难正确运用的之一。我们会自然地把职责结合在一起。软件设计真正要做的许多内容，就是发现职责并把那些职责相互分离。事实上，我们将要论述的其余原则都会以这样或那样的方式回到这个问题上。

参考文献

1. DeMarco, Tom. *Structured Analysis and System Specification.* Yourdon Press Computing Series. Englewood Cliff, NJ: 1979.
2. Page-Jones, Meilir. *The Practical Guide to Structured Systems Design,* 2d ed. Englewood Cliff. HJ: Yourdon Press Computing Series, 1998.

第 9 章　开放－封闭原则（OCP）

两截门（Dutch Door）——（名词）一个被水平分割为两部分的门，这样每一部分都可以独立保持开放或者封闭。

——美国英语传统字典，第 4 版，2000 年

Ivar Jacobson 曾说过，“任何系统在其生命周期中都会发生变化。如果我们期望开发出的系统不会在第 1 版后就被抛弃，就必须牢牢地记住这一点。”[①]那么怎样的设计才能面对需求的改变却可以保持相对稳定，从而使得系统可以在第一个版本以后不断推出新的版本呢？Bertrand Meyer 在 1988 年提出的著名的开放－封闭原则（The Open-Closed Principle，简称 OCP）[②]为我们提供了指引。[③]

9.1　开放－封闭原则（OCP）

软件实体（类、模块、函数等等）应该是可以扩展的，但是不可修改的。

如果程序中的一处改动就会产生连锁反应，导致一系列相关模块的改动，那么设计就具有僵化性的臭味。OCP 建议我们应该对系统进行重构，这样以后对系统再进行那样的改动时，就不会导致更多的修改。如果正确地应用 OCP，那么以后再进行同样的改动时，就只需要添加新的代码，而不必改动已经正常运行的代码。

也许，这看起来像是众所周知的可望而不可及的美好理想——然而，事实上却有一些相对简单并且有效的策略可以帮助接近这个理想。

---

① [Jacobson92], p.21。
② 另一译法“开－闭原则”亦较通行，本书采用“开放－封闭原则”。
③ [Meyer97], p.57。

9.2　描　　述

遵循开放－封闭原则设计出的模块具有两个主要的特征。它们是：

1. “对于扩展是开放的”（Open for extension）。
   这意味着模块的行为是可以扩展的。当应用的需求改变时，我们可以对模块进行扩展，使其具有满足那些改变的新行为。换句话说，我们可以改变模块的功能。

2. “对于更改是封闭的”（Closed for modificaiton）。
   对模块行为进行扩展时，不必改动模块的源代码或者二进制代码。模块的二进制可执行版本，无论是可链接的库、DLL 或者 Java 的.jar 文件，都无需改动。

这两个特征好像是互相矛盾的。扩展模块行为的通常方式就是修改该模块的源代码。不允许修改的模块常常都被认为是具有固定的行为。

怎样可能在不改动模块源代码的情况下去更改它的行为呢？怎样才能在无需对模块进行改动的情况下就改变它的功能呢？

9.3　关键是抽象

在 C++、Java 或者其他任何的 OOPL[1] 中，可以创建出固定却能够描述一组任意个可能行为的抽象体。这个抽象体就是抽象基类。而这一组任意个可能的行为则表现为可能的派生类。

模块可以操作一个抽象体。由于模块依赖于一个固定的抽象体，所以它对于更改可以是关闭的。同时，通过从这个抽象体派生，也可以扩展此模块的行为。

图 9.1 展示了一个简单的不遵循 OCP 的设计。Client 类和 Server 类都是具体类。Client 类使用 Server 类。如果我们希望 Client 对象使用另外一个不同的服务器对象，那么就必须要把 Client 类中使用 Server 类的地方更改为新的服务器类。

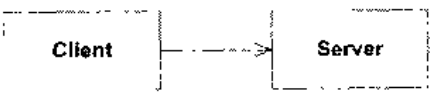


图 9.1　既不开放又不封闭的 Client

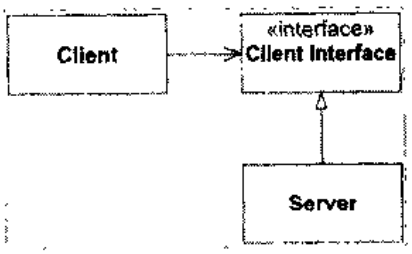


图 9.2　STRATEGY 模式：既开放又封闭的 Client

图 9.2 中展示了一个针对上述问题的遵循 OCP 的设计。在这个设计中，ClientInterface 类是一个拥有抽象成员函数的抽象类。Client 类使用这个抽象类；然而 Client 类的对象却使用 Server 类的派生类的对象。如果我们希望 Client 对象使用一个不同的服务器类，那么只需要从 ClientInterface 类派生一个新的类。无需对 Client 类做任何改动。

Client 需要实现一些功能，它可以使用 ClientInterface 抽象接口去描绘那些功能。ClientInterface 的

1) 面向对象编程语言。

子类型可以以任何它们所选择的方式去实现这个接口。这样，就可以通过创建 ClientInterface 的新的子类型的方式去扩展、更改 Client 中指定的行为。

也许你想知道我为何把抽象接口命名为 ClientInerface。为何不把它命名为 AbstractServer 呢？因为（后面将会看到）抽象类和它们的客户的关系要比和实现它们的类的关系更密切一些。

图 9.3 展示了另一个可选的结构。Policy 类具有一组实现了某种策略的公有函数。和图 9.2 中 Client 类的函数类似，这些策略函数使用一些抽象接口描绘了一些要完成的功能。不同的是，在这个结构中，这些抽象接口是 Policy 类本身的一部分。它们在 C++中表现为纯虚函数，在 Java 中表现为抽象方法。这些函数在 Policy 的子类型中实现。这样，可以通过从 Policy 类派生出新类的方式，对 Policy 中指定的行为进行扩展或者更改。

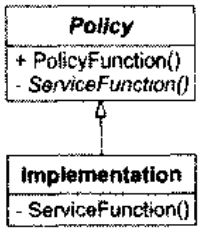


图 9.3 Template Method 模式：既开放又封闭的基类

这两个模式是满足 OCP 的最常用的方法。应用它们，可以把一个功能的通用部分和实现细节部分清晰的分离开来。

9.3.1 Shape 应用程序

下面的例子在许多讲述 OOD（面向对象的设计）的书中都提到过。它就是声名狼藉的“Shape”样例。它常常被用来展示多态的工作原理。不过，这次我们将使用它来阐明 OCP。

我们有一个需要在标准的 GUI 上绘制圆和正方形的应用程序。圆和正方形必须要按照特定的顺序绘制。我们将创建一个列表，列表由按照适当的顺序排列的圆和正方形组成，程序遍历该列表，依次绘制出每个圆和正方形。

9.3.2 违反 OCP

如果使用 C 语言，并采用不遵循 OCP 的过程化方法，我们也许会得到程序 9.1 中所示的解决方法。其中，我们看到了一组的数据结构，它们的第一个成员都相同，但是其余的成员都不同。每个结构中的第一个成员都是一个用来标识该结构是代表圆或者正方形的类型码。DrawAllShapes 函数遍历一个数组，该数组的元素是指向这些数据结构的指针，DrawAllShapes 函数先检查类型码，然后根据类型码调用对应的函数（DrawCircle 或者 DrawSquare）。

程序 9.1 Square/Circle 问题的过程化解决方案

```
-- shape.h ----------------------------------------
enum ShapeType { circle, square };
struct Shape
{
    ShapeType itsType;
}

-- circle.h ---------------------------------------
struct Circle
{
    ShapeType itsType;
    double itsRadius;
    Point itsCenter;
```

```
};

-- square.h ----------------------------------------
struct Square
{
    ShapeType itsType;
    double itsSide;
    Point itsTopLeft;
};

-- drawAllShapes.cc ---------------------------------
typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        struct Shape* s=list[i];
        switch (s->itsType)
        {
            case square:
                DrawSquare((struct Square*)s);
            break;
            case circle:
                DrawCircle((struct Circle*)s);
            Break;
        }
    }
}
```

DrawAllShapes 函数不符合 OCP，因为它对于新的形状类型的添加不是封闭的。如果希望这个函数能够绘制包含有三角形的列表，就必须得更改这个函数。事实上，每增加一种新的形状类型，都必须要更改这个函数。

当然这只是一个简单的例子。在实际程序中，类似 DrawAllShapes 函数中的 switch 语句会在应用程序的各个函数中重复不断地出现，每个函数中 switch 语句负责完成的工作差别甚微。这些函数中，可能有负责拖曳形状对象的，有负责拉伸形状对象的，有负责移动形状对象的，有负责删除形状对象的，等等。在这样的应用程序中增加一种新的形状类型，就意味着要找出所有包含上述 switch 语句（或者链式 if/else 语句）的函数，并在每一处都添加对新增的形状类型的判断。

更糟的是，并不是所有的 switch 语句和 if/else 链都像 DrawAllShapes 中的那样有比较好的结构。更有可能的情形是，if 语句中的判断条件由逻辑操作符组合而成，或者是处理方式相同的 case 语句被成组处理。在一些极端错误的实现中，会有一些函数对于 Square 的处理竟然和对于 Circle 的处理一样。在这样的函数中，甚至根本就没有 switch/case 语句或者 if/else 链。这样，要发现和理解所有的需要增加对新的形状类型进行判断的地方，恐怕就非常的困难了。

同样，在进行上述改动时，我们必须要在 ShapeType enum 中添加一个新的成员。由于所有不同种类的形状都依赖于这个 enum 的声明，所以我们必须要重新编译所有的形状模块。[1]并且也必须要重新编译所有依赖于 Shape 类的模块。

---

(1) 对 enum 的改变会导致持有该 enum 的变量在大小上的改变。所以，如果决定真的不需要重新编译其他的形状声明，一定要非常的小心。

因此，我们不但必须要更改源代码中所有的 switch/case 语句或者 if/else 链，而且还必须得改动所有使用任一个 Shape 数据结构的模块的二进制文件（通过重新编译）。更改二进制文件意味着必须要重新部署所有的 DLL、共享库或者其他类型的二进制组件。给应用程序增加一种新的形状类型这样一个简单的行为，就导致了随后对于许多模块的源代码、甚至许多模块的二进制码和二进制组件的连锁改动。可见，增加一种新的形状类型带来的影响是巨大的。

糟糕的设计

再来回顾一下。程序 9.1 中的解决方法是僵化的，这是因为增加 Triangle 会导致 Shape、Square、Circle 以及 DrawAllShapes 的重新编译和重新部署。该方法是脆弱的，因为有许多其他的即难以查找又难以理解的 switch/case 或者 if/else 语句。该方法是牢固的，因为想在另一个程序中复用 DrawAllShapes 时，都必须要附带上 Square 和 Circle，即使那个新程序不需要它们。因此，在程序 9.1 中展示了许多糟糕设计的臭味。

9.3.3 遵循 OCP

程序 9.2 中展示了一个 square/circle 问题的符合 OCP 的解决方案。在这个方案中，我们编写了一个名为 Shape 的抽象类。这个抽象类仅有一个名为 Draw 的抽象方法。Circle 和 Square 都从 Shape 类派生。

程序 9.2 问题的 OOD 解决方案

```
class Shape
{
    public:
        virtual void Draw() const = 0;
};

class Square: public Shape
{
    public:
        virtual void Draw() const;
};

class Circle: public Shape
{
    public:
        virtual void Draw() const;
};

void DrawAllShapes(vector<Shape*>& list)
{
    vector<Shape*>::iterator I;
    for (i=list.begin(); i != list.end(); i++)
        (*i)->Draw();
}
```

可以看到，如果我们想要扩展程序 9.2 中 DrawAllShapes 函数的行为，使之能够绘制一种新的形状，我们只需要增加一个新的 Shape 类的派生类。DrawAllShapes 函数并不需要改变。这样 DrawAllShapes

就符合了 OCP。无需改动自身代码，就可以扩展它的行为。实际上，增加一个 Triangle 类对于这里展示的任何模块完全没有影响。很明显，为了能够处理 Triangle 类，必须要改动系统中的某些部分，但是这里展示的所有代码都无需改动。

在实际的应用程序中，Shape 类可能会有更多的方法。但是在应用程序中增加一种新的形状类型依然非常简单，因为所需要做的工作只是创建 Shape 类的新的派生类，并实现它的所有函数。再也不需要为了找出需要更改的地方而在应用程序的所有地方进行搜寻。这个解决方案不再是脆弱的。

同时，这个方案也不再是僵化的。在增加一个新的形状类型时，现有的所有模块的源码都无需改动，并且现有的所有二进制模块都无需进行重新构建（rebuild）。只有一个例外，那就是实际创建 Shape 类新的派生类实例的模块必须被改动。通常情况下，创建 Shape 类新的派生类实例的工作要么是在 main 中或者被 main 调用的一些函数中完成，要么是在被 main 创建的一些对象的方法中完成。[①]

最后，这个方案也不再是牢固的。现在，在任何应用程序中重用 DrawAllShapes 时，都无需再附带上 Square 和 Circle。因而，这个解决方案就不再具有前面提及的任何糟糕设计的特征。

这个程序是符合 OCP 的。对它的改动是通过增加新代码进行的，而不是更改现有的代码。因此，它就不会引起像不遵循 OCP 的程序那样的连锁改动。所需要的改动仅仅是增加新的模块，以及为了能够实例化新类型的对象而进行的围绕 main 的改动。

9.3.4　是的，我说谎了

上面的例子其实并非是 100%封闭的！如果我们要求所有的圆必须在正方形之前绘制，那么程序 9.2 中的 DrawAllShapes 函数会怎样呢？DrawAllShapes 函数无法对这种变化做到封闭。要实现这个需求，我们必须要修改 DrawAllShapes 的实现，使它首先扫描列表中所有的圆，然后再扫描所有的正方形。

9.3.5　预测变化和“贴切的”结构

如果我们预测到了这种变化，那么就可以设计一个抽象来隔离它。我们在程序 9.2 中所选定的抽象对于这种变化来说反倒成为一种障碍。可能你会觉得奇怪：还有什么比定义一个 Shape 类，并从它派生出 Square 类和 Circle 类更贴切的结构呢？为何这个贴切的模型不是最优的呢？很明显，这个模型对于一个形状的顺序比形状类型具有更重要意义的系统来说，就不再是贴切的了。

这就导致了一个麻烦的结果，一般而言，无论模块是多么的“封闭”，都会存在一些无法对之封闭的变化。没有对于所有的情况都贴切的模型。

既然不可能完全封闭，那么就必须有策略地对待这个问题。也就是说，设计人员必须对于他设计的模块应该对哪种变化封闭做出选择。他必须先猜测出最有可能发生的变化种类，然后构造抽象来隔离那些变化。

这需要设计人员具备一些从经验中获得的预测能力。有经验的设计人员希望自己对用户和应用

① 这种对象就是大家熟知的厂对象，我们会在第 21 章中对此进行详述。

领域很了解，能够以此来判断各种变化的可能性。然后，他可以让设计对于最有可能发生的变化遵循 OCP 原则。

这一点不容易做到。因为它意味着要根据经验猜测那些应用程序在生长历程中有可能遭受的变化。如果开发人员猜测正确，他们就获得成功。如果他们猜测错误，他们会遭受失败。并且在大多数情况下，他们都会猜测错误。

同时，遵循 OCP 的代价也是昂贵的。创建正确的抽象是要花费开发时间和精力的。同时，那些抽象也增加了软件设计的复杂性。开发人员有能力处理的抽象的数量也是有限的。显然，我们希望把 OCP 的应用限定在可能会发生的变化上。

我们如何知道哪个变化有可能发生呢？我们进行适当的调查，提出正确的问题，并且使用我们的经验和一般常识。最终，我们会一直等到变化发生时才采取行动。

9.3.6 放置吊钩

我们怎样去隔离变化呢？在上个世纪，我们常常说的一句话是，我们会在我们认为可能发生变化的地方放置吊钩（hook）。我们觉得这样做会使软件灵活一些。

然而，我们放置的吊钩常常是错误的。更糟的是，即使不使用这些吊钩，也必须要去支持和维护它们，从而就具有了不必要的复杂性的臭味。这不是一件好事。我们不希望设计背着许多不必要的抽象。通常，我们更愿意一直等到确实需要那些抽象时再把它放置进去。

1. 只受一次愚弄

有句古老的谚语说："愚弄我一次，应感羞愧的是你。再次愚弄我，应感羞愧的是我。"这也是一种有效的对待软件设计的态度。为了防止软件背着不必要的复杂性，我们会允许自己被愚弄一次。这意味着在我们最初编写代码时，假设变化不会发生。当变化发生时，我们就创建抽象来隔离以后发生的同类变化。简而言之，我们愿意被第一颗子弹击中，然后我们会确保自己不再被同一只枪发射的其他任何子弹击中。

2. 刺激变化

如果我们决定接受第一颗子弹，那么子弹到来的越早、越快就对我们越有利。我们希望在开发工作展开不久就知道可能发生的变化。查明可能发生的变化所等待的时间越长，要创建正确的抽象就越困难。

因此，我们需要去刺激变化。我们已在第 2 章中讲述的一些方法来完成这项工作。

- 我们首先编写测试。测试描绘了系统的一种使用方法。通过首先编写测试，我们迫使系统成为可测试的。在一个具有可测试性的系统中发生变化时，我们可以坦然对之。因为我们已经构建了使系统可测试的抽象。并且通常这些抽象中的许多都会隔离以后发生的其他种类的变化。
- 我们使用很短的迭代周期进行开发——一个周期为几天而不是几周。
- 我们在加入基础结构前就开发特性，并且经常性地把那些特性展示给涉众。
- 我们首先开发最重要的特性。
- 尽早地、经常性地发布软件。尽可能快地、尽可能频繁地把软件展示给客户和使用人员。

9.3.7　使用抽象获得显式封闭

第一颗子弹已经击中我们，用户要求我们在绘制正方形之前先绘制所有的圆。现在我们希望可以隔离以后所有的同类变化。

怎样才能使得 DrawAllShapes 函数对于绘制顺序的变化是封闭的呢？请记住封闭是建立在抽象的基础之上的。因此，为了让 DrawAllShapes 对于绘制顺序的变化是封闭的，我们需要一种“顺序抽象体”。这个抽象体定义了一个抽象接口，通过这个抽象接口可以表示任何可能的排序策略。

一个排序策略意味着，给定两个对象可以推导出应该先绘制哪一个。我们可以定义一个 Shape 类的抽象方法叫作 Precedes。这个方法以另外一个 Shape 作为参数，并返回一个 bool 型结果。如果接收消息的 Shape 对象应该先于作为参数传入的 Shape 对象绘制，那么函数返回 true。

在 C++中，这个函数可以通过重载 operator<来表示。程序 9.3 中展示了添加了排序方法后的 Shape 类。

既然我们已经有了决定两个 Shape 对象的绘制顺序的方法，我们就可以对列表中的 shape 对象进行排序后依序绘制。程序 9.4 展示了 C++的实现代码。

程序 9.3　具有排序方法的 Shape 类

```
class Shape
{
    public:
        virtual void Draw() const = 0;
        virtual bool Precedes(const Shape&) const = 0;

        bool operator<(const Shape& s) {return Precedes(s);}
};
```

程序 9.4　依序绘制的 DrawAllShapes 函数

```
template <typename P>
class Lessp // utility for sorting containers of pointers.
{
    public:
        bool operator() (const P p, const P q) {return (*p) < (*q);}
};

void DrawAllShapes(vector<Shape*>& list)
{
    vector<Shape*> orderedList = list;
    sort(orderedList.begin();
         orderedList.end();
         Lessp<Shape*>());

    vector<Shape*>::const_iterator I;
    for (i=orderedList.begin(); i != orderedList.end(); i++)
        (*i)->Draw();
}
```

这给我们提供了一种对 Shape 对象排序的方法，也使得可以按照一定的顺序来绘制它们。但是我们仍然没有一个好的用来排序的抽象体。按照目前的设计，Shape 对象应该覆写 Precedes 方法来指定顺序。这究竟是如何工作的呢？我们应该在 Circle:Precedes 成员函数中编写一些什么代码，来保

证圆一定会被先于正方形绘制呢？请看程序 9.5。

程序 9.5 对 Circle 排序

```
bool Circle::Precedes(const Shape& s) const
{
    if (dynamic_cast<Square*>(s))
        return true;
    else
        return false;
}
```

显然这个函数以及所有 Shape 类的派生类中的 Precedes 函数都不符合 OCP。没有办法使得这些函数对于 Shape 类的新派生类做到封闭。每次创建一个新的 Shape 类的派生类时，所有的 Precedes() 函数都需要改动。[1]

当然，如果从来不需创建新的 Shape 类的派生类，就没有关系了。另一方面，如果需要频繁的创建新的 Shape 类的派生类，这个设计就会遭到沉重的打击。我们再次被第一颗子弹击中。

9.3.8 使用“数据驱动”的方法获取封闭性

如果我们要使 Shape 类的各个派生类间互不知晓，可以使用表格驱动的方法。程序 9.6 展示了一种可能的实现。

程序 9.6 表格驱动的形状类型排序机制

```
#include <typeinfo>
#include <string>
#include <iostream>

using namespace std;

class Shape
{
    public:
        virtual void Draw() const = 0;
        bool Precedes(const Shape&) const;
        bool operator(const Shape& s) const;
        { return Precedes(s);}
    private:
        static const char* typeOrderTable[];
};

const char* Shape::typeOperderTable[] =
{
    typeid(Circle).name(),
    typeid(Square).name(),
    0
};

// This function searches a table for the class names.
// The table defines the order in which the
```

[1] 可以使用第 29 章中描述的 ACYCLIC VISITOR 模式来解决这个问题。不过现在就展示这个解决方案还为时过早。在第 29 章结束时我会提醒你再回到这里。

```
// shapes are to be drawn. Shapes that are not
// found always precede shapes that are found.
//
bool Shape::Precedes(const Shape& s) const
{
    const char* thisType = typeid(*this).name();
    const char* argType = typeid(s).name();
    bool done = false;
    int thisOrd = -1;
    int argOrd = -1;
    for (int i=0; !done; i++)
    {
        const char* tableEntry = typeOrderTable[i];
        if (tableEntry != 0)
        {
            if (strcmp(tableEntry, thisType) == 0)
                thisOrd = i;
            if (strcmp(tableEntry, argType) == 0)
                argOrd = i;
            if ((argOrd >= 0) && (thisOrd >= 0))
                done = true;
        }
        else // table entry == 0
                done = true;
    }
    return thisOrd < argOrd;
}
```

通过这种方法，我们成功地做到了一般情况下 DrawAllShapes 函数对于顺序问题的封闭，也使得每个 Shape 派生类对于新的 Shape 派生类的创建或者基于类型的 Shape 对象排序规则的改变是封闭的。（比如，改变顺序为正方形必须最先绘制。）

对于不同的 Shapes 的绘制顺序的变化不封闭的惟一部分就是表本身。可以把表放置在一个单独的模块中，和所有其他模块隔离，因此对于表的改动不会影响到其他任何模块。事实上，在 C++中，我们可以在链接时选择要使用的表。

9.4 结　　论

在许多方面，OCP 都是面向对象设计的核心所在。遵循这个原则可以带来面向对象技术所声称的巨大好处（也就是，灵活性、可重用性以及可维护性）。然而，并不是说只要使用一种面向对象语言就是遵循了这个原则。对于应用程序中的每个部分都肆意地进行抽象同样不是一个好主意。正确的做法是，开发人员应该仅仅对程序中呈现出频繁变化的那些部分做出抽象。拒绝不成熟的抽象和抽象本身一样重要。

参考文献

1. Jacobson, Ivar, et al. *Object-Oriented Software Engineering.* Reading, MA: Addison-Wesley, 1992.
2. Meyer, Bertrand. *Object-Oriented Software Construction,* 2d ed. Upper Saddle River, NJ: Prentice Hall, 1997.

第 10 章　Liskov 替换原则（LSP）

OCP 背后的主要机制是抽象（abstraction）和多态（polymorphism）。在静态类型语言中，比如 C++和 Java，支持抽象和多态的关键机制之一是继承（inheritance）。正是使用了继承，我们才可以创建实现其基类（base class）中抽象方法的派生类。

是什么设计规则在支配着这种特殊的继承用法呢？最佳的继承层次的特征又是什么呢？怎样的情况会使我们创建的类层次结构掉进不符合 OCP 的陷阱中去呢？这些正是 Liskov 替换原则（LSP）要解答的问题。

10.1　Liskov 替换原则（LSP）

对于 LSP 可以做如下解释：

子类型（subtype）必须能够替换掉它们的基类型（base type）。

Barbara Liskov 首次写下这个原则是在 1988 年。[①]她说道，

> 这里需要如下替换性质：若对每个类型 S 的对象 o_1，都存在一个类型 T 的对象 o_2，使得在所有针对 T 编写的程序 P 中，用 o_1 替换 o_2 后，程序 P 行为功能不变，则 S 是 T 的子类型。

想想违反该原则的后果，LSP 的重要性就不言而喻了。假设有一个函数 f，它的参数为指向某个基类 B 的指针（pointer）或者引用（reference）。同样假设有 B 的某个派生类 D,如果把 D 的对象作为 B 类型传递给 f，会导致 f 出现错误的行为。那么 D 就违反了 LSP。显然，D 对于 f 来说是脆弱的。

f 的编写者会想去对 D 进行一些测试，以便于在把 D 的对象传递给 f 时，可以使 f 具有正确的行为。这个测试违反了 OCP，因为此时 f 对于 B 的所有不同的派生类都不再是封闭的。这样的测试

---

① [Liskov88]。

是一种代码的臭味，它是缺乏经验的开发人员（或者，更糟的，匆忙的开发人员）在违反了 LSP 时所产生的结果。

10.2　一个违反 LSP 的简单例子

对于 LSP 的违反常常会导致以明显违反 OCP 的方式使用运行时类型辨别（RTTI）。这种方式常常是使用一个显式的 if 语句或者 if/else 链去确定一个对象的类型，以便于可以选择针对该类型的正确行为。考虑一下程序 10.1。

程序 10.1　对 LSP 的违反导致了对 OCP 的违反

```
struct Point {double x,y;};

struct Shape {
    enum ShapeType { square, circle } itsType;
    Shape(ShapeType t):itsType(t) {}
};

struct Circle:public Shape
{
    Circle():Shape(circle) {};
    void Draw() const;
    Point itsCenter;
    double itsRadius;
};

struct Square:public Shape
{
    Square():Shape(square) {};
    void Draw() const;
    Point itsTopLeft;
    double itsSide;
};

void DrawShape(const Shape& s)
{
    if (s.itsType == Shape::square)
        static_cast<const Square&>(s).Draw();
    else if (s.itsType == Shape::circle)
        static_cast<const Circle&>(s).Draw();
}
```

很显然，程序 10.1 中的 DrawShape 函数违反了 OCP。它必须知道 Shape 类所有的派生类，并且每次创建一个从 Shape 类派生的新类时都必须要更改它，甚至很多人肯定地认为这种函数结构简直是对良好设计的诅咒。那么，是什么促使程序员编写出像这样的一个函数呢？

假设 Joe 是一个工程师。他学习过面向对象技术，并且认为多态的开销大得难以忍受。[1]因此，他定义了一个没有任何虚函数的 Shape 类。类（结构）Square 和 Circle 从 Shape 类派生，并具有 Draw() 函数，但是它们没有覆写（override）Shape 类中的函数。因为 Circle 类和 Square 类不能替换 Shape 类，所以 DrawShape 函数必须要仔细检查输入的 Shape 对象，确定它的类型，接着调用正确的 Draw 函数。

---

(1) 在一个具有相当速度的计算机中，每个方法调用的开销是 1ns 的数量级，所以 Joe 的观点是不正确的。

Square 类和 Circle 类不能替换 Shape 类其实是违反了 LSP，这个违反又迫使 DrawShape 函数违反了 OCP，因而，对于 LSP 的违反也潜在地违反了 OCP。

10.3　正方形和矩形，更微妙的违规

当然存在更为微妙的违反 LSP 的方式。考虑一个使用了程序 10.2 中描述的 Rectangle 类的应用程序。

程序 10.2　Rectangle 类

```
class Rectangle
{
    public:
        void    SetWidth(double w)  {itsWidth=w;}
        void    SetHeight(double h) {itsHeight=h;}
        double  GetHeight() const   {return itsHeight;}
        double  GetWidth() const    {return itsWidth;}
    private:
        Point   itsTopLeft;
        double  itsWidth;
        double  itsHeight;
};
```

假设这个应用程序运行得很好，并被安装在许多地方。和任何一个成功的软件一样，用户的需求不时会发生变化。某一天，用户不满足于仅仅操作矩形，要求添加操作正方形的功能。

我们经常说继承是 IS-A（“是一个”）关系。也就是说，如果一个新类型的对象被认为和一个已有类的对象之间满足 IS-A 关系，那么这个新对象的类应该从这个已用对象的类派生。

从一般意义上讲，一个正方形就是一个矩形。因此，把 Square 类视为从 Rectangle 类派生是合乎逻辑的。参见图 10.1。

IS-A 关系的这种用法有时被认为是面向对象分析（Object Oriented Analysis，简称 OOA）基本技术之一[1]。一个正方形是一个矩形，所以 Square 类就应该派生自 Rectangle 类。不过，这种想法会带来一些微妙但极为值得重视的问题。一般来说，这些问题是很难预见的，直到我们编写代码时才会发现它们。

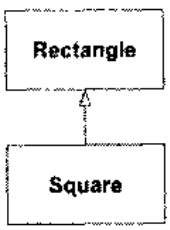


图 10.1　Square 从 Rectangle 继承

我们首先注意到出问题的地方是，Square 类并不同时需要成员变量 itsHeight 和 itsWidth。但是 Square 仍会从 Rectangle 中继承它们。显然这是浪费。在许多情况下，这种浪费是无关紧要的。但是，如果我们必须要创建成百上千个 Square 对象（比如，在 CAD/CAM 中复杂的电路的每个元件的管脚引线都作为正方形进行绘制），浪费的程度则是巨大的。

假设目前我们并不十分关心内存效率。从 Rectangle 派生 Square 也会产生其他一些问题。Square 会继承 SetWidth 和 SetHeight 函数。这两个函数对于 Square 来说是不合适的，因为正方形的长和宽是相等的。这是表明存在问题的重要标志。不过这个问题是可以避免的。我们可以按照如下方式覆写 SetWidth 和 SetHeight：

---

[1] 一个被频繁使用却很少定义的术语。

```
void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}

void Square::SetHeight(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}
```

现在，当设置 Square 对象的宽时，它的长会相应地改变。当设置长时，宽也会随之改变。这样，就保持了 Square 要求的不变性。[1] Square 对象是具有严格数学意义下的正方形。

```
Square s;
s.SetWidth(1);      // Fortunately sets the height to 1 too.
s.SetHeight(2);     // sets width and height to 2. Good thing.
```

但是考虑下面这个函数：

```
void f(Rectangle& r)
{
    r.SetWidth(32);  // calls Rectangle::SetWidth
}
```

如果我们向这个函数传递一个指向 Square 对象的引用，这个 Square 对象就会被破坏，因为它的长并不会改变。这显然违反了 LSP。以 Rectangle 的派生类的对象作为参数传入时，函数 f 不能正确运行。错误的原因是在 Rectangle 中没有把 SetWidth 和 SetHeight 声明为虚函数；因此它们不是多态的。

这个错误很容易修正。然而，如果新派生类的创建会导致我们改变基类，这就常常意味着设计是有缺陷的。当然也违反了 OCP。也许有人会反驳说，真正的设计缺陷是忘记把 SetWidth 和 SetHeight 声明为虚函数，而我们已经做了修正。可是，这很难让人信服，因为设置一个长方形的长和宽是非常基本的操作。如果不是预见到 Square 的存在，我们凭什么要把这两个函数声明为虚函数呢？

尽管如此，假设我们接受这个理由并修正这些类。修正后的代码如程序 10.3 所示。

程序 10.3　自相容的 Rectangle 类和 Square 类

```
class Rectangle
{
    public:
        virtual void    SetWidth(double w)  {itsWidth=w;}
        virtual void    SetHeight(double h) {itsHeight=h;}
        double          GetHeight() const   {return itsHeight;}
        double          GetWidth() const    {return itsWidth;}
    private:
        Point   itsTopLeft;
        double  itsWidth;
        double  itsHeight;
```

[1] 无论在什么状态下都必须为 true 的属性。

```
};

class Square:public Rectangle
{
    public:
        virtual void    SetWidth(double w);
        virtual void    SetHeight(double h);
};

void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}

void Square::SetHeight(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}
```

10.3.1 真正的问题

现在 Square 和 Rectangle 看起来都能够工作。无论对 Square 对象进行什么样的操作，它都和数学意义上的正方形保持一致。无论我们对 Rectangle 对象进行什么样的操作，它都和数学意义上的长方形保持一致。此外，可以向接受指向 Rectangle 的指针或者引用的函数传递 Square，而 Square 依然保持正方形的特性，与数学意义上的正方形一致。

这样看来该设计似乎是自相容的、正确的。可是，这个结论是错误的。一个自相容的设计未必就和所有的用户程序相容。考虑下面的函数 g：

```
void g(Rectangle& r)
{
    r.SetWidth(5);
    r.SetHeight(4);
    assert(r.Area() == 20);
}
```

这个函数认为所传递进来的一定是 Rectangle，并调用了其成员函数 SetWidth 和 SetHeight。对于 Rectangle 来说，此函数运行正确，但是如果传递进来的是 Square 对象就会发生断言错误（assertion error）。所以，真正的问题是：函数 g 的编写者假设改变 Rectangle 的宽不会导致其长的改变。

很显然，改变一个长方形的宽不会影响它的长的假设是合理的！然而，并不是所有可以作为 Rectangle 传递的对象都满足这个假设。如果把一个 Square 类的实例传递给像 g 这样做了该假设的函数，那么这个函数就会出现错误的行为。函数 g 对于 Square/Rectangle 层次结构来说是脆弱的。

函数 g 的表现说明有一些使用指向 Rectangle 对象的指针或者引用的函数，不能正确地操作 Square 对象。对于这些函数来说，Square 不能够替换 Rectangle，因此 Square 和 Rectangle 之间的关系是违反 LSP 的。

有人会对函数 g 中存在的问题进行争辩，他们认为函数 g 的编写者不能假设宽和长是独立变化的。g 的编写者不会同意这种说法的。函数 g 以 Rectangle 作为参数。并且确实有一些不变性质和原

理说明明显适用于 Rectangle 类，其中一个不变性质就是长和宽可以独立变化。g 的编写者完全可以对这个不变性质进行断言。倒是 Square 的编写者违反了这个不变性。

真正有趣的是，Square 的编写者没有违反正方形的不变性。由于让 Square 从 Rectangle 派生，Square 的编写者违反了 Rectangle 的不变性！

10.3.2　有效性并非本质属性

LSP 让我们得出一个非常重要的结论：一个模型，如果孤立地看，并不具有真正意义上的有效性。模型的有效性只能通过它的客户程序来表现。例如，如果孤立地看，最后那个版本的 Rectangle 和 Square 是自相容的且有效的。但是如果从对基类做出了一些合理假设的程序员的角度来看，这个模型就是有问题的。

在考虑一个特定设计是否恰当时，不能完全孤立地来看这个解决方案。必须要根据该设计的使用者所做出的合理假设来审视它。[①]

有谁知道设计的使用者会做出什么样的合理假设呢？大多数这样的假设都很难预测。事实上，如果试图去预测所有这些假设，我们所得到的系统很可能会充满不必要的复杂性的臭味。因此，像所有其他原则一样，通常最好的方法是只预测那些最明显的对于 LSP 的违反情况而推迟所有其他的预测，直到出现相关的脆弱性的臭味时，才去处理它们。

10.3.3　IS-A 是关于行为的

那么究竟是怎么回事？Square 和 Rectangle 这个显然合理的模型为什么会有问题呢？毕竟，Square 应该是 Rectangle。难道它们之间不存在 IS-A 关系吗？

对于那些不是 g 的编写者而言，正方形可以是长方形，但是从 g 的角度来看，Square 对象绝对不是 Rectangle 对象。为什么！因为 Square 对象的行为方式和函数 g 所期望的 Rectangle 对象的行为方式不相容。从行为方式的角度来看，Square 不是 Rectangle，对象的行为方式才是软件真正所关注的问题。LSP 清楚地指出，OOD 中 IS-A 关系是就行为方式而言的，行为方式是可以进行合理假设的，是客户程序所依赖的。

10.3.4　基于契约设计

许多开发人员可能会对“合理假设”行为方式的概念感到不安。怎样才能知道客户真正的要求呢？有一项技术可以使这些合理的假设明确化，从而支持了 LSP。这项技术被称为基于契约设计（Design By Contract，简称 DBC），Bertrand Meyer 对此进行过详细的介绍。[②]

使用 DBC，类的编写者显式地规定针对该类的契约。客户代码的编写者可以通过该契约获悉可以依赖的行为方式。契约是通过为每个方法声明的前置条件（preconditions）和后置条件（postconditions）来指定的。要使一个方法得以执行，前置条件必须要为真。执行完毕后，该方法要保证后置条件为真。

---

① 这些合理的假设常常以断言的形式出现在为基类编写的单元测试中。这是又一个要实践测试驱动开发的好理由。

② [Meyer97], Chapter 11, p.331。

Rectangle::SetWidth(double w)的后置条件可看作是：

```
assert((itsWidth == w) && (itsHeight == old.itsHeight));
```

在这个例子中，old 是 SetWidth 被调用前 Rectangle 的值。按照 Meyer 所述，派生类的前置条件和后置条件规则是：

> 在重新声明派生类中的例程（routine）时，只能使用相等或者更弱的前置条件来替换原始的前置条件，只能使用相等或者更强的后置条件来替换原始的后置条件。[①]

换句话说，当通过基类的接口使用对象时，用户只知道基类的前置条件和后置条件。因此，派生类对象不能期望这些用户遵从比基类更强的前置条件。也就是说，它们必须接受基类可以接受的一切。同时，派生类必须和基类的所有后置条件一致。也就是说，它们的行为方式和输出不能违反基类已经确立的任何限制。基类的用户不应被派生类的输出扰乱。

显然，Square::SetWidth(double w)的后置条件比 Rectangle:: SetWidth(double w)的后置条件弱(weaker)[②]，因为它不服从(itsHeight == old.itsHeight)这条约束。因而，Square 的 SetWidth 方法违反了基类订下的契约。

某些语言，比如 Eiffel，对前置条件和后置条件有直接的支持。你只需声明它们，运行时系统会去检验它们。C++和 Java 中都没有此项特性。在这些语言中，我们必须自己考虑每个方法的前置条件和后置条件，并确保没有违反 Meyer 规则。此外，为每个方法都注明它们的前置条件和后置条件是非常有帮助的。

10.3.5　在单元测试中指定契约

也可以通过编写单元测试的方式来指定契约。单元测试通过彻底的测试一个类的行为来使该类的行为更加清晰。客户代码的编写者会去查看这些单元测试，这样他们就可以知道对于要使用的类，应该做出什么合理的假设。

10.4　一个实际的例子

对正方形和矩形说的已经够多了！LSP 在实际的软件中能否发挥作用呢？我们来看一个案例研究，它来自我几年前做过的一个项目。

10.4.1　动机

在 20 世纪 90 年代初期，我购买了一个第三方的类库，其中包含有一些容器（container）类。这些容器和 Smalltalk 中的 Bags 和 Sets 略微有些关系。这些容器中有两个 Set 的变体（variety）以及两个类似的 Bag 变体。第一个变体是“有限的（bounded）”，是基于数组实现的。第二个变体是“无限的（unbounded）”，是基于链表实现的。

BoundedSet 的构造函数（constructor）指定了它能够容纳的元素的最大数目。BoundedSet 内部定

① [Meyer97], p.573, 断言重新声明规则（1）。
② 术语“弱”是一个容易混淆的概念。如果 X 没有遵从 Y 的所有约束，那么 X 就比 Y 弱。X 所遵从的新约束的数目是无关紧要的。

义了一个数组来为这些元素预分配了空间。因此，如果 BoundedSet 创建成功了，那么就可以确信它一定具有足够的存储空间。由于 BoundedSet 是基于数组的，所以是非常快速的。在正常操作期间，也不会发生内存分配动作。并且由于内存是预先分配的，所以可以确信对于 BoundedSet 的操作不会耗尽堆空间（heap）。另一方面，由于 BoundedSet 很少会完全使用预先分配的所有空间，所以在内存使用方面存在着浪费。

另一方面，UnboundedSet 对于它可以容纳的元素的数目没做限制。只要还有可用的堆内存，UnboundedSet 就可以继续接受元素。因此，它是非常灵活的。同时，它也是节约内存的，因为它仅为目前容纳的元素分配内存。另外，由于在正常的操作期间必须要分配和归还内存，所以速度较慢。最后，还存在一个危险，那就是对它进行的正常操作可能会耗尽堆空间。

我不喜欢这些第三方类的接口。我不希望自己的应用程序代码依赖于这些容器类，因为我觉得以后会用更好的来替换它们。因此，我把它们包装在我自己的抽象接口下，如图 10.2 所示。

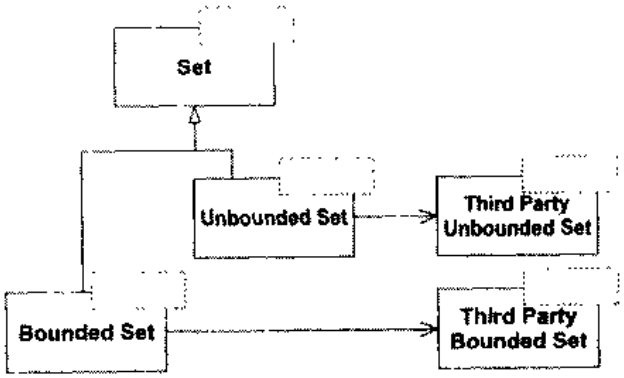


图 10.2　容器类适配器层

我创建了一个称为 Set 的抽象类，提供了 Add、Delete 以及 IsMemeber 这几个纯虚函数。如程序 10.4 所示。这个结构统一了第三方集合的两个变体：unbounded 变体和 bounded 变体，让我们通过一个公共的接口访问它们。这样，客户就可以接受类型为 Set<T>&的参数而不用关心实际使用的 Set 是 bounded 变体还是 unbounded 变体。（参见程序 10.5 中的 PrintSet 函数。）

程序 10.4　抽象 Set 类

```
template <class T>
class Set
{
    public:
        virtual void Add(const T&) = 0;
        virtual void Delete(const T&) = 0;
        virtual bool IsMember(const T&) const = 0;
};
```

程序 10.5　PrintSet

```
template <class T>
void PrintSet(const Set<T>& s)
{
    for (Iterator<T>i(s); i; i++
        cout << (*i) << endl;
}
```

不用关心所使用的 Set 的具体类型，这是一个大大的优点。这意味着程序员可以在每个具体的情况中选择所需要的 Set 种类，而不会影响到客户函数。在内存紧张而速度要求不严格时，程序员可以选择 UnboundedSet，或者在内存充裕而对速度有严格要求时，程序员可以选择 BoundedSet。客户函数是通过基类 Set 的接口来操纵这些对象的，因此也就不必关心使用的是哪种 Set。

10.4.2　问题

我想在该层次中加入 PersistentSet。所谓持久性（persistent）集合是指可以把其中的元素写入流，稍后可能由另外的程序再从流中读回其中的集合。遗憾的是，我能够访问的惟一的、同时也提供了持久化功能的第三方容器类不是一个模板类。相反，它只接受虚基类 PersistentObject 的派生对象。我创建的层次结构如图 10.3 所示。

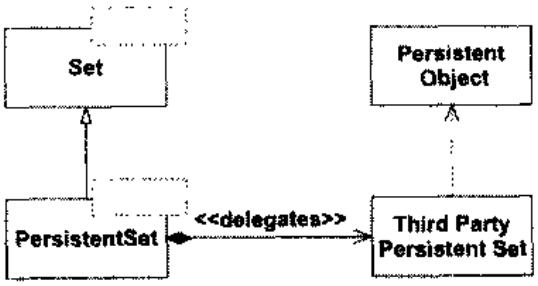


图 10.3　持久性集合层次结构

请注意，PersistentSet 包含了一个第三方持久性集合的实例，它把它的所有方法都委托（delegate）给该实例。这样，如果调用了 PersistentSet 的 Add 方法，它就简单地把该调用委托给第三方持久性集合中包含的对应方法。

表面看起来，好像没有问题。其实隐藏着一个别扭的设计问题。加入到第三方持久性集合中的元素必须得从 PersistentObject 派生。由于 PersistentSet 只是把调用委托给第三方持久性集合，所以任何要加入 PersistentSet 的元素也必须得从 PersistentObjet 派生。可是，Set 的接口没有这样的限制。

当客户程序向基类 Set 中加入元素时，客户程序不能确保该 Set 实际上是否是一个 PersistentSet。因而，客户程序没有办法知道它所加入的元素是否应该从 PersistentObject 派生。

考察程序 10.6 中 PersistentSet::Add()的代码。

程序 10.6　template <typename T>

```
void PersistentSet::Add(const T& t)
{
    PersistentObject& p =
        dynamic_cast<PersistentObject&>(t);
    itsThirdPartyPersistentSet.Add(p);
}
```

从该代码中可以明显地看出，如果任何客户企图向 PersistentSet 中添加不是从 PersistentObject 派生的对象，将会发生运行时错误。dynamic_cast 会抛出 bad_cast 异常。但是抽象基类 Set 的所有现存的客户都不会预计到调用 Add 时会抛出异常。由于 Set 的派生类会导致这些函数出现错误，所以对类层次所做的这种改动违反了 LSP。

这是个问题吗？当然是，那些以前传递 Set 的派生对象时根本没有问题的函数，现在传递给它

们 PersistentSet 对象时却会引发运行时错误。调试这种问题很困难，因为这个运行时错误发生之处距离实际的逻辑错码很远。逻辑错误可能是由于把 PersistentSet 传给了一个函数，也可能是由于向 PersistentSet 加入的对象不是派生自 PersistentObject。无论哪种情况，实际发生逻辑错误的地方可能距离调用 Add 方法的地方还有十万八千里呢！找到问题很难，解决问题更难。

10.4.3　不符合 LSP 的解决方案

怎样解决这个问题呢？几年前，我通过约定（convention）的方式解决了这个问题。也就是说没有在源代码中解决它。我约定不让 PersistentSet 和 PersistentObject 暴露给整个应用程序。它们只被一个特定的模块使用。该模块负责从持久性存储设备读出所有容器，也负责把所有容器写入到持久性存储设备。在写入容器时，该容器的内容先被复制到对应的 PersistentObject 的派生对象中，再加入到 PersistentSets，然后存入流中。在从流中读入容器时，过程是相反的。先把信息从流读到 PersistentSet 中，再把 PersistentObjects 从 PersistentSet 中移出并复制到常规的（非持久化）对象中，然后再加入到常规的 Set 中。

这个解决方案有很强的限制性，但也是我当时想到的惟一的方法，可以不让 PersistentSet 对象出现在想要在其中加入非持久性对象的函数接口中。此外，这也解除了应用程序的其余部分对整个持久化概念的依赖。

这个解决方案奏效吗？没有。有些没有理解这个约定重要性的开发人员，在应用程序的多处地方违反了这个约定。这就是使用约定方式的问题——要不断地跟每位开发人员解释。如果某位开发人员没有弄清楚或者不同意，就会违反这个约定。而一次违反就会致使整个结构的失败。

10.4.4　符合 LSP 的解决方案

现在该如何解决这个问题呢？我承认 PersistentSet 和 Set 之间不存在 IS-A 关系，它不应派生自 Set。因此我会分离这个层次结构，但不是完全的分离。Set 和 PersistentSet 之间有一些公有的特性。事实上，仅仅是 Add 方法致使在 LSP 原则下出了问题。因此，我创建了一个层次结构，其中 Set 和 PersistentSet 是兄弟关系（siblings），统一在一个具有测试成员关系、遍历等操作的抽象接口下（参见图 10.4）。这就可以对 PersistentSet 对象进行遍历以及测试成员关系等操作。但是它不能够把不是派生自 PersistentObject 的对象加入到 PersistentSet 中。

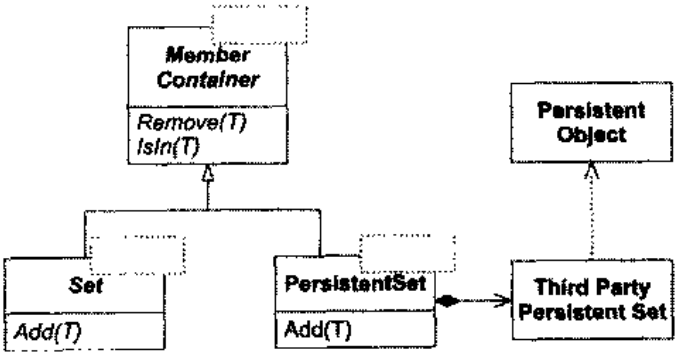


图 10.4　符合 LSP 的解决方案

10.5　用提取公共部分的方法代替继承

另一个有趣并令人迷惑的继承案例是 Line 和 LineSegment 的例子。[1]考察一下程序 10.7 和程序 10.8。最初看到这两个类时，会觉得它们之间有自然的公有继承（public inheritance）关系。LineSegment 需要 Line 中声明的每一个成员变量和每一个成员函数。此外，LineSegment 新增了一个自己的成员函数 GetLength，并覆写了 IsOn 函数。但是，这两个类还是以微妙的方式违反了 LSP。

程序 10.7　geometry/line.h

```
#ifndef GEOMETRY_LINE_H
#define GEOMETRY_LINE_H
#include "geometry/point.h"

class Line
{
    public:
        Line(const Point& p1, const Point& p2);

        double  GetSlope()  const;
        double  GetIntercept()  const;  // Y Intercept
        Point   GetP1()     const {return itsP1;};
        Point   GetP2()     const {return itsP2;};
        virtual bool IsOn(const Point&) const;

    private:
        Point itsP1;
        Point itsP2;
};
#endif
```

程序 10.8　geometry/lineseg.h

```
#ifndef GEOMETRY_LINESEGMENT_H
#define GEOMETRY_LINESEGMENT_H
class LineSegment:public Line
{
    public:
        LineSegment(const Point& p1, const Point& p2);
        double      GetLength()     const;
        virtual bool    IsOn(const Point&)  const;
};
#endif
```

Line 的使用者可以期望和该 Line 具有线性对应关系（colinear）的所有点都在该 Line 上。例如，由 Intercept 函数返回的点就是线和 y 轴的交点。由于这个点和线具有线性对应关系，所以 Line 的使用者可以期望 IsOn(Intercept()) == true。然而，对于许多 LineSegment 的实例，这条声明会失效。

这为什么是一个重要的问题呢？为什么不简单地让 LineSegment 从 Line 派生并忍受这个微妙的问题呢？这是一个需要进行判断的问题。在大多数情况下，接受一个多态行为中的微妙错误都不会比试着修改设计使之完全符合 LSP 更为有利。接受缺陷而不是去追求完美这是一个工程上的权衡问题。好的工程师知道何时接受缺陷比追求完美更有利。不过，不应该轻易放弃对于 LSP 的遵循。总是保

① 尽管本例和 Square/Rectangle 例子有相似，但是它来自一个真实的应用程序并把它作为真正的问题进行讨论。

证子类可以代替它的基类是一个有效的管理复杂性的方法。一旦放弃了这一点，就必须要单独来考虑每个子类。

有一个简单的方案可以解决 Line 和 LineSegment 的问题，该方案也阐明了一个 OOD 的重要工具。如果既要使用类 Line 又要使用类 LineSegment，那么可以把这两个类的公共部分提取出来作为一个抽象基类。程序 10.9～程序 10.11 展示了把 Line 和 LineSegment 的公共部分提取出来作为基类 LinearObject 后的结果。

程序 10.9　geometry/linearobj.h

```
#ifndef GEOMETRY_LINEAR_OBJECT_H
#define GEOMETRY_LINEAR_OBJECT_H

#include "geometry/point.h"

class LinearObject
{
    public:
        Line(const Point& p1, const Point& p2);

        double  GetSlope()  const;
        double  GetIntercept()  const;

        Point   GetP1()     const {return itsP1;};
        Point   GetP2()     const {return itsP2;};
        virtual bool IsOn(const Point&) const = 0; //abstract.

    private:
        Point itsP1;
        Point itsP2;
};
#endif
```

程序 10.10　geometry/line.h

```
#ifndef GEOMETRY_LINE_H
#define GEOMETRY_LINE_H
#include "geometry/linearobj.h"

class Line: public LinearObject
{
    public:
        Line(const Point& p1, const Point& p2);
        virtual bool IsOn(const Point&) const;
};
#endif
```

程序 10.11　geometry/linearseg.h

```
#ifndef GEOMETRY_LINESEGMENT_H
#define GEOMETRY_LINESEGMENT_H
#include "geometry/lineobj.h"

class LineSegment:public LinearObject
{
    public:
        LineSegment(const Point& p1, const Point& p2);
```

```
        double      GetLength()     const;
        virtual bool    IsOn(const Point&)  const;
};
#endif
```

LinearObject 既代表了 Line 又代表了 LineSegment。它提供了两个子类的大部分的功能和数据成员，其中不包括纯虚的 IsOn 方法。LinearObject 的使用者不得假设他们知道正在使用的对象的长度。这样，它们就可以接受 Line 或者 LineSegment 而不会出现任何问题。此外，Line 的使用者也根本不会去处理 LineSegment 的情况。

提取公共部分是一个设计工具，最好在代码不是很多时应用。当然，如果程序 10.7 中所示的类 Line 已经存在很多的客户，那么提取出 LinearObject 就不会这么轻松。不过在有可能时，它仍是一个有效的工具。如果两个子类中具有一些公共的特性，那么很可能稍后出现的其他类也会需要这些特性。关于提取公共部分，Rebecca Wirfs、Brian Wilkerson 以及 Lauren Wiener 是这样说的：

> 如果一组类都支持一个公共的职责，那么它们应该从一个公共的超类（superclass）继承该职责。如果公共的超类还不存在，那么就创建一个，并把公共的职责放入其中。毕竟，这样一个类的有用性是确定无疑的——你已经展示了一些类会继承这些职责。然而稍后对系统的扩展也许会加入一个新的子类，该子类很可能会以新的方式来支持同样的职责。此时，这个新创建的超类可能会是一个抽象类。[①]

程序 10.12 展示了一个不曾预料到的类 Ray 是如何使用 LinearObject 的属性（attribute）的。Ray 可以替换 LinearObject，并且 LinearObject 的使用者在处理 Ray 时不会有任何问题。

程序 10.12 geometry/ray.h

```
#ifndef GEOMETRY_RAY_H
#define GEOMETRY_RAY_H

class Ray: public LinearObject
{
    public:
        Ray(const Point& p1, const Point& p2);
        virtual bool    IsOn(const Point&)  const;
};
#endif
```

10.6 启发式规则和习惯用法

有一些简单的启发规则可以提供一些有关违反 LSP 的提示。这些规则都和以某种方式从其基类中去除功能的派生类有关。完成的功能少于其基类的派生类通常是不能替换其基类的，因此就违反了 LSP。

10.6.1 派生类中的退化函数

考察一下程序 10.13。在 Base 中实现了函数 f。不过，在 Derived 中，函数 f 是退化的（degenerate）。也许，Derived 的编写者认为函数 f 在 Derived 中没有用处。遗憾的是，Base 的使用者不知道它们不应

① [WirfsBrock90]，第 113 页。

该调用 f，因此就出现了一个替换违规。

程序 10.13　派生类中的一个退化函数

```
public class Base
{
    public void f()     { /* some code */ }
}

public class Derived extends Base
{
    public void f()  {}
}
```

在派生类中存在退化函数并不总是表示违反了 LSP，但是当存在这种情况时，还是值得注意一下的。

10.6.2　从派生类中抛出异常

另外一种 LSP 的违规形式是在派生类的方法中添加了其基类不会抛出的异常。如果基类的使用者不期望这些异常，那么把它们添加到派生类的方法中就会导致不可替换性。此时要遵循 LSP，要么就必须改变使用者的期望，要么派生类就不应该抛出这些异常。

10.7　结　　论

OCP 是 OOD 中很多说法的核心。如果这个原则应用得有效，应用程序就会具有更多的可维护性、可重用性以及健壮性。LSP 是使 OCP 成为可能的主要原则之一。正是子类型的可替换性才使得使用基类类型的模块在无需修改的情况下就可以扩展。这种可替换性必须是开发人员可以隐式依赖的东西。因此，如果没有显式地强制基类类型的契约，那么代码就必须良好地并且明显地表达出这一点。

术语“IS-A”的含意过于宽泛以至于不能作为子类型的定义。子类型的正确定义是“可替换性的”，这里的可替换性可以通过显式或者隐式的契约来定义。

参考文献

1. Meyer, Bertrand. *Object-Oriented Software Construction,* 2d ed. Upper Saddle River, NJ: Prentice Hall, 1997.
2. Wirfs-Brock, Rebecca, et al. *Designing Object-Oriented Software.* Englewood Cliffs, NJ: Prentice Hall, 1990.
3. Liskov, Barbara. *Data Absraction and Hierarchy.* SIGPLAN Notices, 23,5（May 1998）.

第 11 章　依赖倒置原则（DIP）

绝不能再让国家的重大利益依赖于那些会动摇人类薄弱意志的众多可能性。

——Sir Thomas Noon Talfourd（1795－1854）

11.1　依赖倒置原则（DIP）

a. 高层模块不应该依赖于低层模块。二者都应该依赖于抽象。
b. 抽象不应该依赖于细节。细节应该依赖于抽象。

在这些年中，有许多人曾经问我为什么在这条原则的名字中使用“倒置”这个词。这是由于许多传统的软件开发方法，比如结构化分析和设计，总是倾向于创建一些高层模块依赖于低层模块、策略（policy）依赖于细节的软件结构。实际上这些方法的目的之一就是要定义子程序层次结构，该层次结构描述了高层模块怎样调用低层模块。图 7.1 中 Copy 程序的初始设计就是这种层次结构的一个典型示例。一个设计良好的面向对象的程序，其依赖程序结构相对于传统的过程式方法设计的通常结构而言就是被“倒置”了。

请考虑一下当高层模块依赖于低层模块时意味着什么。高层模块包含了一个应用程序中的重要的策略选择和业务模型。正是这些高层模块才使得其所在的应用程序区别于其他。然而，如果这些高层模块依赖于低层模块，那么对低层模块的改动就会直接影响到高层模块，从而迫使它们依次做出改动。

这种情形是非常荒谬的！本应该是高层的策略设置模块去影响低层的细节实现模块的。包含高层业务规则的模块应该优先并独立于包含实现细节的模块。无论如何高层模块都不应该依赖于低层模块。

此外，我们更希望能够重用的是高层的策略设置模块。我们已经非常擅长于通过子程序库的形式来重用低层模块。如果高层模块依赖于低层模块，那么在不同的上下文中重用高层模块就会变得非常困难。然而，如果高层模块独立于低层模块，那么高层模块就可以非常容易地被重用。该原则是框架（framework）设计的核心原则。

11.2　层　次　化

Booch 曾经说过："……所有结构良好的面向对象构架都具有清晰的层次定义，每个层次通过一个定义良好的、受控的接口向外提供了一组内聚的服务。"[①] 对这个陈述的简单理解可能会致使设计者设计出类似图 11.1 的结构。图中，高层的 Policy Layer 使用了低层的 Mechanism Layer，而 Mechanism Layer 又使用了更细节的层 Utility Layer。这看起来似乎是正确的，然而它存在一个隐伏的错误特征，那就是：Policy Layer 对于其下一直到 Utility Layer 的改动都是敏感的。这种依赖关系是传递的。Policy Layer 依赖于某些依赖于 Utility Layer 的层次；因此 Policy Layer 传递性地依赖于 Utility Layer。这是非常糟糕的。

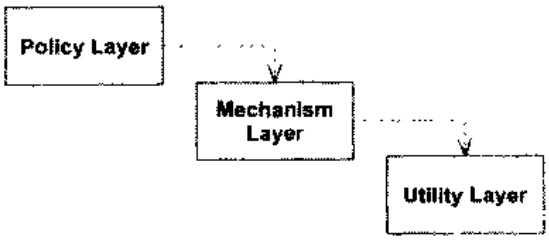


图 11.1　简单的层次化方案

图 11.2 展示了一个更为合适的模型。每个较高层次都为它所需要的服务声明一个抽象接口，较低的层次实现了这些抽象接口，每个高层类都通过该抽象接口使用下一层，这样高层就不依赖于低层。低层反而依赖于在高层中声明的抽象服务接口。这不仅解除了 PolicyLayer 对于 UtilityLayer 的传递依赖关系，甚至也解除了 PolicyLayer 对于 MechanismLayer 的依赖关系。

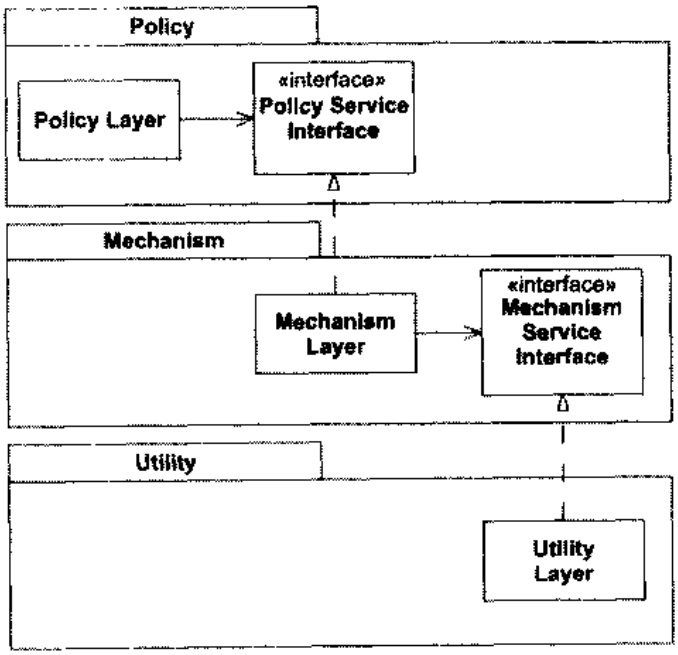


图 11.2　倒置的层次

请注意这里的倒置不仅仅是依赖关系的倒置，它也是接口所有权的倒置。我们通常会认为工具库应该拥有它们自己的接口。但是当应用了 DIP 时，我们发现往往是客户拥有抽象接口，而它们的服务者则从这些抽象接口派生。

① [Booch96]，第 54 页。

11.2.1 倒置的接口所有权

这就是著名的 Hollywood 原则："Don't call us, we'll call you."(不要调用我们，我们会调用你。)[①] 低层模块实现了在高层模块中声明并被高层模块调用的接口。

通过这种倒置的接口所有权，对于 MechanismLayer 或者 UtilityLayer 的任何改动都不会再影响到 PolicyLayer。而且，PolicyLayer 可以在定义了符合 PolicyServiceInterface 的任何上下文中重用。这样，通过倒置这些依赖关系，我们创建了一个更灵活、更持久、更易改变的结构。

11.2.2 依赖于抽象

一个稍微简单但仍然非常有效的对于 DIP 的解释，是这样一个简单的启发式规则："依赖于抽象。"这是一个简单的陈述，该启发式规则建议不应该依赖于具体类——也就是说，程序中所有的依赖关系都应该终止于抽象类或者接口。

根据这个启发式规则，可知：

- 任何变量都不应该持有一个指向具体类的指针或者引用
- 任何类都不应该从具体类派生
- 任何方法都不应该覆写它的任何基类中的已经实现了的方法

当然，每个程序中都会有违反该启发规则的情况。有时必须要创建具体类的实例，而创建这些实例的模块将会依赖于它们。[②]此外，该启发规则对于那些虽是具体但却稳定（nonvolatile）的类来说似乎不太合理。如果一个具体类不太会改变，并且也不会创建其他类似的派生类，那么依赖于它并不会造成损害。

比如，在大多数的系统中，描述字符串的类都是具体的。例如，在 Java 中，表示字符串的是具体类 String。该类是稳定的，也就是说，它不太会改变。因此，直接依赖于它不会造成损害。

然而，我们在应用程序中所编写的大多数具体类都是不稳定的。我们不想直接依赖于这些不稳定的具体类。通过把它们隐藏在抽象接口的后面，可以隔离它们的不稳定性。

这不是一个完美的解决方案。常常，如果一个不稳定类的接口必须要变化时，这个变化一定会影响到表示该类的抽象接口。这种变化破坏了由抽象接口维系的隔离性。

由此可知，该启发规则对问题的考虑有点简单了。另一方面，如果看得更远一点，认为是由客户类来声明它们需要的服务接口，那么仅当客户需要时才会对接口进行改变。这样，改变实现抽象接口的类就不会影响到客户。

11.3 一个简单的例子

依赖倒置可以应用于任何存在一个类向另一个类发送消息的地方。例如，Button 对象和 Lamp 对

① [Sweet85]。

② 事实上，如果可以通过字符串来创建类的话，那么就有一些方法可以解决该问题。在 Java 中可以这样做。还有一些其他的语言中也可以使用该方法。在这些语言中，可以把具体类的名字作为配置数据传给程序。

象之间的情形。

Button 对象感知外部环境的变化。当接收到 Poll 消息时，它会判断是否被用户“按下”。它不关心是通过什么样的机制去感知的。可能是 GUI 上的一个按钮图标，也可能是一个能够用手指按下的真正按钮，甚至可能是一个家庭安全系统中的运动检测器。Button 对象可以检测到用户激活或者关闭它。

Lamp 对象会影响外部环境。当接收到 TurnOn 消息时，它显示某种灯光。当接收到 TurnOff 消息时，它把灯光熄灭。它可以是计算机控制台的 LED，也可以是停车场的水银灯，甚至是激光打印机中的激光。

该如何设计一个用 Button 对象控制 Lamp 对象的系统呢？图 11.3 展示了一个不成熟的设计。Button 对象接收 Poll 消息，判断按钮是否被按下，接着简单地发送 TurnOn 或者 TurnOff 消息给 Lamp 对象。

图 11.3　不成熟的 Button 和 Lamp 模型

为何说它是不成熟的呢？考虑一下对应这个模型的 Java 代码（见程序 11.1）。请注意 Button 类直接依赖于 Lamp 类。这个依赖关系意味着当 Lamp 类改变时，Button 类会受到影响。此外，想要重用 Button 来控制一个 Motor 对象是不可能的。在这个设计中，Button 控制着 Lamp 对象，并且也只能控制 Lamp 对象。

程序 11.1　Button.java

```
public class Button
{
    private Lamp itsLamp;
    public void poll()
    {
        if ( /* some condition */ )
            itsLamp.turnOn();
    }
}
```

这个方案违反了 DIP。应用程序的高层策略没有和低层实现分离。抽象没有和具体细节分离。没有这种分离，高层策略就自动地依赖于低层模块，抽象就自动地依赖于具体细节。

找出潜在的抽象

什么是高层策略呢？它是应用背后的抽象，是那些不随具体细节的改变而改变的真理。它是系统内部的系统——它是隐喻（metaphore）。在 Button/Lamp 例子中，背后的抽象是检测用户的开/关指令并将指令传给目标对象。用什么机制检测用户的指令呢？无关紧要！目标对象是什么？同样无关紧要！这些都是不会影响到抽象的具体细节。

通过倒置对 Lamp 对象的依赖关系，可以改进图 11.3 中的设计。在图 11.4 中，可以看到 Button 现在和一个称为 ButtonServer 的接口关联起来了。ButtonServer 接口提供了一些抽象方法，Button 可以使用这些方法来开启或者关掉一些东西。Lamp 实现了 ButtonServer 接口。这样，Lamp 现在是依赖于别的东西了，

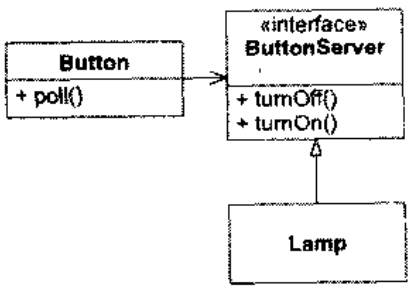


图 11.4　对 Lamp 应用依赖倒置原则

而不是被依赖了。

图 11.4 中的设计可以使 Button 控制那些愿意实现 ButtonServer 接口的任何设备。这赋予我们极大的灵活性。同时也意味着 Button 对象将能够控制还没有被创造出来的对象。

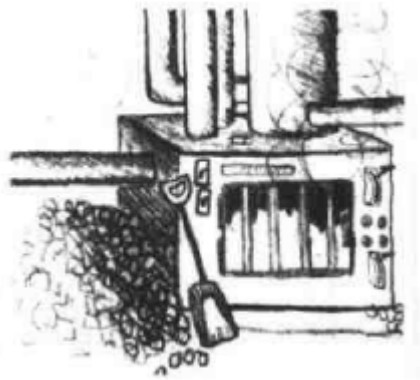
不过，这个方案对那些需要被 Button 控制的对象提出了一个约束。需要被 Button 控制的对象必须要实现 ButtonServer 接口。这不太好，因为这些对象可能也要被 Switch 对象或者一些不同于 Button 的对象控制。

通过倒置依赖关系的方向，并使得 Lamp 依赖于其他类而不是被其他类依赖，我们使 Lamp 依赖于一个不同的具体细节——Button。是这样吗？

Lamp 的确依赖于 ButtonServer，但是 ButtonServer 没有依赖于 Button。任何知道如何去操纵 ButtonServer 接口的对象都能够控制 Lamp。因此，这个依赖关系只是名字上的依赖。可以通过给 ButtonServer 起一个更通用一点的名字，比如 SwitchableDevice，来修正这一点。也可以确保把 Button 和 SwitchableDevice 被放置在不同的库中，这样对 SwitchableDevice 的使用就不必包含对 Button 的使用。

在本例中，接口没有所有者。这是一个有趣的情形，其中接口可以被许多不同的客户使用，并被许多不同的服务者实现。这样，接口就需要被放置在一个单独的组（group）中。在 C++中，可以把它放在一个单独的 namespace 和库中。在 Java 中，可以把它放在一个单独的 package 中。[①]

11.4 熔炉示例



我们来看一个更有趣的例子。考虑一个控制熔炉调节器的软件。该软件可以从一个 IO 通道中读取当前的温度，并通过向另一个 IO 通道发送命令来指示熔炉的开或者关。算法结构看起来如程序 11.2 所示。

程序 11.2 一个温度调节器的简单算法

```
#define TERMOMETER  0x86
#define FURNACE     0x87
#defina UNGAGE      1
#define DISENGAGE   0

void Regulate(double minTemp, double maxTemp)
{
    for (;;)
    {
        while (in(THERMONETER) > minTemp)
            wait(1);
        out (FURNACE, ENGAGE);

        while (in(THERMONETER) < minTemp)
            wait(1);
        out (FURNACE, DISENGAGE);
    }
}
```

① 在像 Smalltalk、Pyrhon 或者 Ruby 这样的动态语言中，接口完全不必作为显式的源码实体存在。

算法的高层意图是清楚的，但是实现代码中却夹杂着许多低层细节。这段代码根本不能重用于不同的控制硬件。

由于代码很少，所以这样做不会造成太大的损害。但是，即使是这样，使算法失去重用性也是可惜的。我们更愿意倒置这种依赖关系，结果如图 11.5 所示。

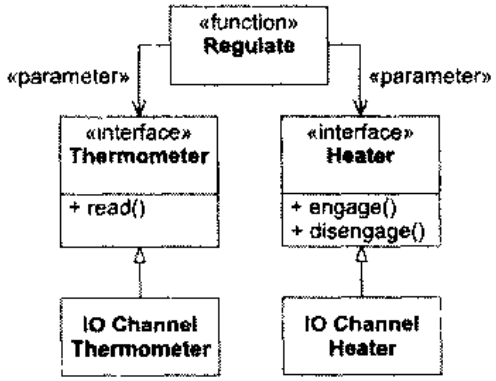


图 11.5　通用的调节器

图中显示了 Regulate 函数接受了两个接口参数。Thermoeter 接口可以读取，而 Heater 接口可以启动和停止。Regulate 算法需要的就是这些。它的实现如程序 11.3 所示

程序 11.3　通用的调节器

```
void Regulate(Thermometer& t, Heater& h,
              double minTemp, double maxTemp)
{
    for (;;)
    {
        while (t.Read() > minTemp)
            wait(1);
        h.Engage();

        while (t.Read() < minTemp)
            wait(1);
        h.Disengage();
    }
}
```

这就倒置了依赖关系，使得高层的调节策略不再依赖于任何温度计或者熔炉的特定细节。该算法具有很好的可重用性。

动态多态性与静态多态性

我们已经完成了依赖关系的倒置，并通过使用动态的多态性（也就是，抽象类或者接口）实现了通用的 Regulate。不过，还有另外一种方法。我们还可以使用由 C++模板提供的静态形式的多态性。考虑程序 11.4。

程序 11.4

```
template <typename THERMOMETER, typename HEATER>
class Regulate(Thermometer& t, Heater& h, double minTemp, double maxTemp)
{
    for (;;)
    {
        while (t.Read() > minTemp)
            wait(1);
        h.Engage();

        while (t.Read() < minTemp)
            wait(1);
        h.Disengage();
    }
}
```

这实现了同样的依赖关系的倒置，并且不具有动态多态性的开销（或者灵活性）。在 C++中，Read、Engaged 以及 Disengaged 方法都可以是非虚的。此外，任何声明了这些方法的类都可以作为模板参数使用。它们不必从一个公共基类继承。

作为模板，Regulate 不依赖于这些函数的任何特定实现。它只是要求替换 HEADER 的类要有一个 Engage 和一个 Disengage 方法，替换 THERMPMETER 的类要有一个 Read 函数。因此，这些类必须要实现模板所定义的接口。换句话说。Regualte 以及 Regulate 所使用的类都必须遵从相同的接口约定，并且它们都依赖于这个约定。

静态多态性很好地解除了源代码中的依赖关系，但是它不能解决动态多态性解决的所有问题。模板方法的缺点是：

- HEATER 和 THERMOMETER 的类型不能在运行时更改；
- 对于新类型的 HEADER 或者 THERMOMTER 的使用会迫使重新编译和重新部署。所以，除非有非常严格的速度性能要求，否则应该优先使用动态多态性。

11.5 结　论

使用传统的过程化程序设计所创建出来的依赖关系结构，策略是依赖于细节的。这是糟糕的，因为这样会使策略受到细节改变的影响。面向对象的程序设计倒置了依赖关系结构，使得细节和策略都依赖于抽象，并且常常是客户拥有服务接口。

事实上，这种依赖关系的倒置正是好的面向对象设计的标志所在。使用何种语言来编写程序是无关紧要的。如果程序的依赖关系是倒置的，它就是面向对象的设计。如果程序的依赖关系不是倒置的，它就是过程化的设计。

依赖倒置原则是实现许多面向对象技术所宣称的好处的基本低层机制。它的正确应用对于创建可重用的框架来说是必须的。同时它对于构建在变化面前富有弹性的代码也是非常重要的。由于抽象和细节被彼此隔离，所以代码也非常容易维护。

参考文献

1. Booch, Grady. *Object Solutions.* Menlo Park, CA: Addison-Wesley, 1996.
2. Gamma, et al. *Design Patterns.* Reading, MA: Addison-Wesley, 1995.
3. Sweet. Richard W. *The Mesa Programming Environment.* SIGPLAN Notices, 20（7）（July 1985）: 216-229.

第 12 章　接口隔离原则（ISP）

这个原则用来处理“胖（fat）”接口所具有的缺点。如果类的接口不是内聚的（cohesive），就表示该类具有“胖”的接口。换句话说，类的“胖”接口可以分解成多组方法。每一组方法都服务于一组不同的客户程序。这样，一些客户程序可以使用一组成员函数，而其他客户程序可以使用其他组的成员函数。

ISP 承认存在有一些对象，它们确实不需要内聚的接口；但是 ISP 建议客户程序不应该看到它们作为单一的类存在。相反，客户程序看到的应该是多个具有内聚接口的抽象基类。

12.1　接口污染

考虑一个安全系统。在这个系统中，有一些 Door 对象，可以被加锁和解锁，并且 Door 对象知道自己是开着还是关着。（参见程序 12.1。）

程序 12.1　安全系统中的 Door

```
class Door
{
    public:
        virtual void Lock() = 0;
        virtual void Unlock() = 0;
        virtual bool IsDoorOpen() = 0;
};
```

该类是抽象的，这样客户程序就可以使用那些符合 Door 接口的对象，而不需要依赖于 Door 的特定实现。

现在，考虑一个这样的实现，TimedDoor，如果门开着的时间过长，它就会发出警报声。为了做到这一点，TimedDoor 对象需要和另一个名为 Timer 的对象交互。（参见程序 12.2。）

程序 12.2

```
class Timer
{
    public:
        void Register(int timeout, TimerClient* client);
};

class TimerClient
{
    public:
        virtual void TimeOut() = 0;
};
```

如果一个对象希望得到超时通知，它可以调用 Timer 的 Register 函数。该函数有两个参数，一个是超时时间，另一个是指向 TimerClient 对象的指针，该对象的 TimeOut 函数会在超时到达时被调用。

我们怎样将 TimerClient 类和 TimedDoor 类联系起来，才能在超时时通知到 TimedDoor 中相应的处理代码呢？有几个方案可供选择。图 12.1 中展示了一个易想到的解决方案。其中 Door 继承了

TimerClient，因此 TimedDoor 也就继承了 TimeClient。这就保证了 TimerClient 可以把自己注册到 Timer 中，并且可以接收 TimeOut 消息。

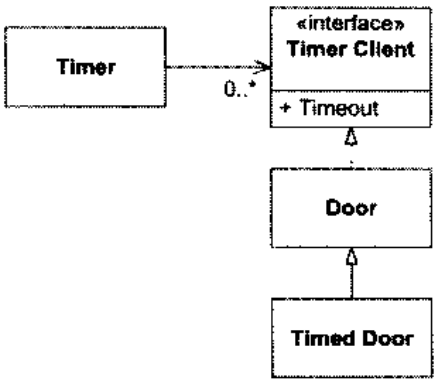


图 12.1　位于层次结构顶部的 Timer Client

虽然这个解决方案很常见，但是它也不是没有问题。最主要的问题是，现在 Door 类依赖于 TimerClient 了。可是并不是所有种类的 Door 都需要定时功能。事实上，最初的 Door 抽象类和定时功能没有任何关系。如果创建了无需定时功能的 Door 的派生类，那么在这些派生类中就必须要提供 TimeOut 方法的退化（degenerate）实现——这就有可能违反 LSP。此外，使用这些派生类的应用程序即使不使用 TimerClient 类的定义，也必须要引入（import）它。这样就具有了不必要的复杂性以及不必要的重复的臭味。

这是一个接口污染的例子，这种情况在像 C++、Java 这样的静态类型语言中是很常见的。Door 的接口被一个它不需要的方法污染了。在 Door 的接口中加入这个方法只是为了能给它的一个子类带来好处。如果持续这样做的话，那么每次子类需要一个新方法时，这个方法就会被加到基类中去。这会进一步污染基类的接口，使它变“胖”。

此外，每次基类中加入一个方法时，派生类中就必须要实现这个方法（或者定义一个缺省实现）。事实上，有一种特定的相关实践，可以使派生类无需实现这些方法，该实践的做法是把这些接口合并为一个基类，并在这个基类中提供接口中方法的退化实现。但是按照我们前面所学习的，这种实践违反了 LSP，带来了维护和重用方面的问题。

12.2　分离客户就是分离接口

Door 接口和 TimerClient 接口是被完全不同的客户程序使用的。Timer 使用 TimerClient，而操作门的类使用 Door。既然客户程序是分离的，所以接口也应该保持分离。为什么呢？因为客户程序对它们使用的接口施加有作用力。

客户对接口施加的反作用力

在我们考虑软件中引起变化的作用力时，通常考虑的都是接口的变化会怎样影响它们的使用者。例如，如果 TimerClient 的接口改变了，我们会去关心 TimerClient 的所有使用者要做什么样的改变。然而，存在着从另外一个方向施加的作用力。有时，迫使接口改变的，正是它们的使用者。

例如，有些 Timer 的使用者会注册多个超时通知请求。比如对于 TimedDoor 来说。当它检测到门被打开时，会向 Timer 发送一个 Register 消息，请求一个超时通知。可是，在超时到达前，门关上了，关闭一会儿后又被再次打开。这就导致在原先的超时到达前又注册了一个新的超时请求。最后，最初的超时到达，TimedDoor 的 TimeOut 方法被调用。Door 错误地发出了警报。

使用程序 12.3 中展示的惯用手法（convention），可以改正上面情形中的错误。在每次超时注册中都包含一个惟一的 timeoutId 码，并在调用 TimerClient 的 TimeOut 方法时，再次使用该标识码。这样 TimerClient 的每个派生类就都可以根据这个标识码知道应该响应哪个超时请求。

程序 12.3　使用 ID 的 Timer 类

```
class Timer
{
    public:
        void Register(int timeout, int timeOutID, TimerClient* client);
};

class TimerClient
{
    public:
        virtual void TimeOut(int timeOutID) = 0;
};
```

显然，这个改变会影响到 TimerClient 的所有使用者。但是由于缺少 timeOutId 是一个必须要改正的错误，所以我们接受这种改变。然而，对于图 12.1 中的设计，这个修正还会影响到 Door 以及 Door 的所有客户程序。这是僵化性和粘滞性的臭味。为什么 TimerClient 中的一个 bug 会影响到那些不需要定时功能的 Door 的派生类的客户程序呢？如果程序中一部分的更改会影响到程序中完全和它无关的其他部分，那么更改的代价和影响就变得不可预测，并且更改所附带的风险也会急剧增加。

12.3　接口隔离原则（ISP）

不应该强迫客户依赖于它们不用的方法。

如果强迫客户程序依赖于那些它们不使用的方法，那么这些客户程序就面临着由于这些未使用方法的改变所带来的变更。这无意中导致了所有客户程序之间的耦合。换种说法，如果一个客户程序依赖于一个含有它不使用的方法的类，但是其他客户程序却要使用该方法，那么当其他客户要求这个类改变时，就会影响到这个客户程序。我们希望尽可能地避免这种耦合，因此我们希望分离接口。

12.4　类接口与对象接口

再次考虑一下 TimedDoor 问题。这里有一个具有两个独立的接口，由两个独立的客户——Timer 以及 Door 所使用的对象。因为实现这两个接口需要操作同样的数据，所以这两个接口必须在同一个对象中实现。那么怎样才能遵循 ISP 呢？怎样才能分离必须在一起实现的接口呢？

该问题的答案基于这样的事实，就是一个对象的客户不是必须通过该对象的接口去访问它，也可以通过委托或者通过该对象的基类去访问它。

12.4.1　使用委托分离接口

一个解决方案是创建一个派生自 TimerClient 的对象，并把对该对象的请求委托给 TimedDoor。图 12.2 展示了这个解决方案。

当 TimedDoor 想要向 Timer 对象注册一个超时请求时，它就创建一个 DoorTimerAdapter 并且把它注册给 Timer。当 Timer 对象发送 TimeOut 消息给 DoorTimerAdapter 时，DoorTimerAdapter 把这个消息委托给 TimedDoor。

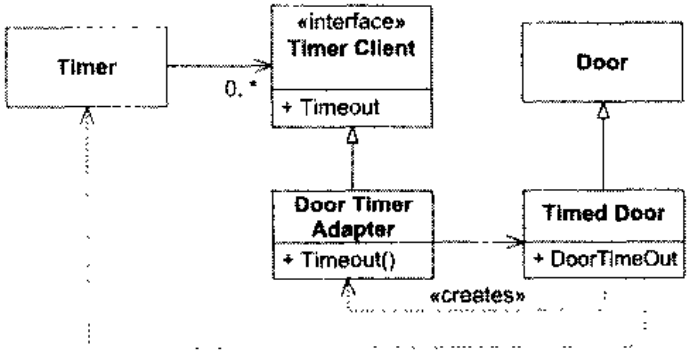


图 12.2 Door 定时器适配器

这个解决方案遵循 ISP 原则，并且避免了 Door 的客户程序和 Timer 之间的耦合。即使对程序 12.3 中所示的 Timer 进行了改变，也不会影响到任何 Door 的使用者。此外，TimedDoor 也不必具有和 TimerClient 一样的接口。DoorTimerAdapter 会将 TimerClient 接口转换成 TimedDoor 接口。因此，这是一个非常通用的解决方案。（参见程序 12.4。）

程序 12.4 TimedDoor.cpp

```
class TimedDoor: public Door
{
    public:
        virtual void DoorTimeOut(int timeOutID);
};

class DoorTimeAdapter: public TimerClient
{
    public:
        DoorTimerAdapter(TimedDoor& theDoor): itsTimedDoor(theDoor)
        {}

        vistual void TimeOut(int timeOutId)
        { itsTimedDoor.DoorTimeOut(timeOutId);}

    private:
        TimedDoor& itsTimedDoor;
}
```

不过，这个解决方案还是有些不太优雅。每次想去注册一个超时请求时，都要去创建一个新的对象。此外，委托处理会导致一些很小但仍然存在的运行时间和内存的开销。有一些应用领域，比如嵌入式实时控制系统，其中内存和运行时间都是非常宝贵的，以至于这种开销成了一个值得关注的问题。

12.4.2 使用多重继承分离接口

图 12.3 和程序 12.5 展示了如何使用多重继承来达到符合 ISP 的目标。在这个模型中，TimerdDoor 同时继承了 Door 和 TimerClient。尽管这两个基类的客户程序都可以使用 TimedDoor，但是实际上却都不再

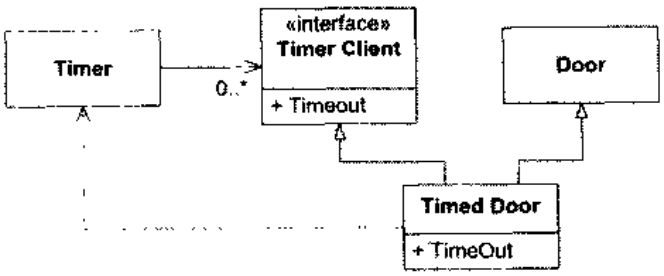


图 12.3 多重继承的 Timed Door

依赖于 TimedDoor 类。这样，它们就通过分离的接口使用同 一个对象。

程序 12.5　TimedDoor.cpp

```
class TimedDoor: public Door, public TimerClient
{
    public:
        virtual void DoorTimeOut(int timeOutID);
};
```

通常我会优先选择这个解决方案。只有当 DoorTimerAdapter 对象所做的转换是必须的，或者不同的时候会需要不同的转换时，我才会选择图 12.2 中的方案而不是图 12.3 中的方案。

12.5　ATM 用户界面的例子

现在我们来考虑一个更有意义一点的例子：传统的自动取款机（ATM）问题。ATM 需要一个非常灵活的用户界面。它的输出信息需要被转换成许多不同的语言。输出信息可能被显示在屏幕上，或者布莱叶盲文书写板上，或者通过语音合成器说出来。显然，通过创建一个抽象基类，其中具有用来处理所有不同的、需要被该界面呈现的消息的抽象方法，就可以实现这种需求。

同样可以把每个 ATM 可以执行的不同操作封装为类 Transaction 的派生类。这样，我们可以得到类 DepositTransaction、WithdrawalTransaction 以及 TransferTransaction。每个类都调用 UI 的方法。例如，为了要求用户输入希望存储的金额，DepositTransaction 对象会调用 UI 类中的 RequestDepositAmount 方法。同样，为了要求用户输入想要转账的金额，TransferTransaction 对象会调用 UI 类中的 RequestTransferAmount 方法。图 12.5 中为相应的类图。

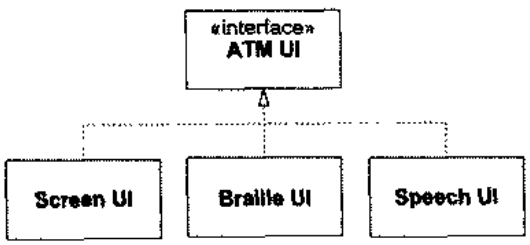


图 12.4　ATM 界面层次结构

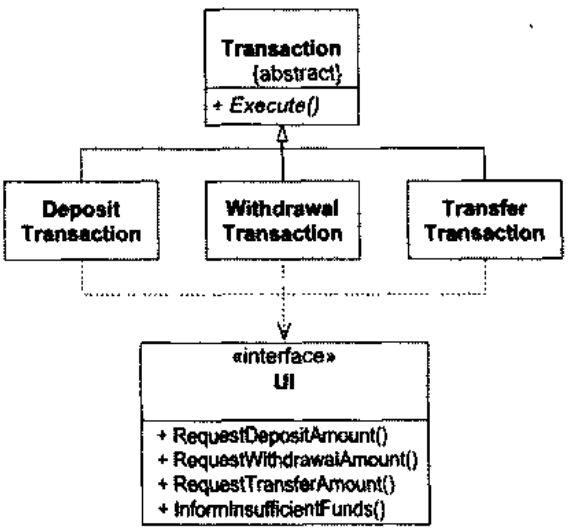


图 12.5　ATM 操作层次结构

请注意这正好是 LSP 告诉我们应该避免的情形。每个操作所使用的 UI 的方法，其他的操作类都不会使用。这样，对于任何一个 Transaction 的派生类的改动都会迫使对 UI 的相应改动，从而也影

响到了其他所有 Transaction 的派生类以及其他所有依赖于 UI 接口的类。这样的设计就具有了僵化性以及脆弱性的臭味。

例如，如果要增加一种操作 PayGasBillTransaction，为了处理该操作想要显示的特定消息，就必须要在 UI 中加入新的方法。糟糕的是，由于 DepositTransaction、WithdrawlTransaction 以及 TransferTransaction 全都依赖于 UI 接口，所以它们都需要重新编译。更糟糕的是，如果这些操作都作为不同的 DLL 或者共享库中的组件部署的话，那么这些组件必须得重新部署，即使它们的逻辑没有做过任何改动。你闻到粘滞性的臭味了吗？

通过将 UI 接口分解成像 DepositUI,WithdrawUI 以及 TransferUI 这样的单独接口，可以避免这种不合适的耦合。最终的 UI 接口可以去多重继承这些单独的接口。图 12.6 和程序 12.6 展示了这个模型。

每次创建一个 Transaction 类的新派生类时，抽象接口 UI 就需要增加一个相应的基类，并且因此 UI 接口以及所有它的派生类都必须改变。不过，这些类并没有被广泛的使用。事实上，它们可能仅被 main 或者那些启动系统并创建具体 UI 实例之类的过程所使用。因此，增加新的 UI 基类所带来的影响被减至最小。

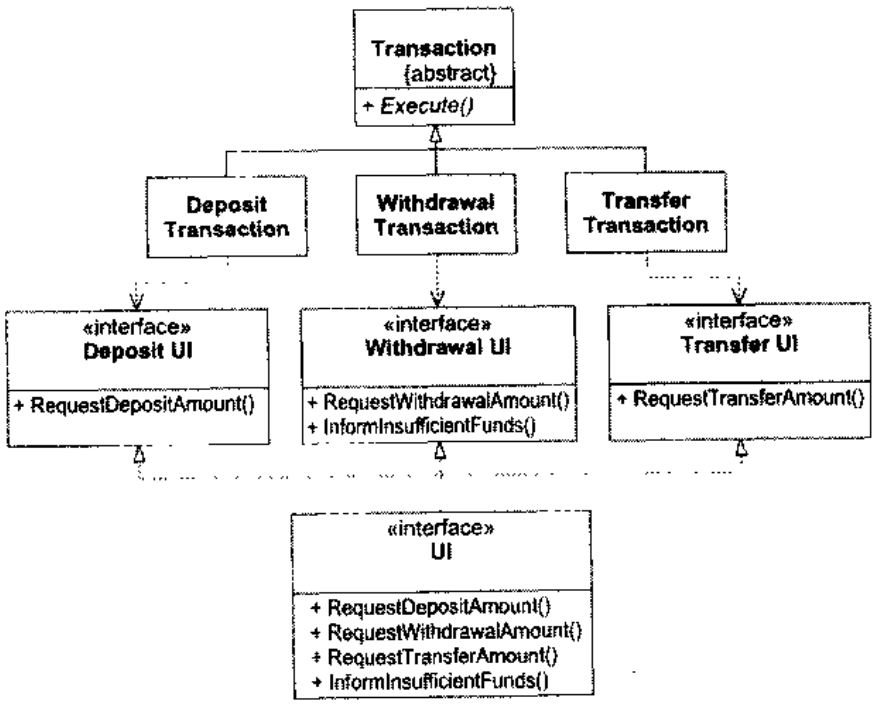


图 12.6 分离的 ATM UI 接口

程序 12.6 分离的 ATM UI 接口

```
class DepositUI
{
    public:
        virtual void RequestDepositAmount() = 0;
};

class DepositTransaction: public Transaction
{
    public:
```

```
        DepositTransaction(DepositUI& ui): itsDepositUI(ui)
        {}

        virtual void Execute()
        {
            ...
            itsDepositUI.RequestDepositAmount();
            ...
        }
    private:
        DepositUI& itsDepositUI;
}

class WithdrawalUI
{
    public:
        virtual void RequestWithDrawalAmount() = 0;
};

class WithdrawalTransaction: public Transaction
{
    public:
        WithdrawalTransaction(WithdrawalUI& ui)
        : itsWithdrawalUI(ui)
        {}

        virtual void Execute()
        {
            ...
            itsWithdrawalUI.RequestWithdrawalAmount();
            ...
        }
    private:
        WithdrawalUI& itsWithdrawalUI;
};

class TransferUI
{
    public:
        virtual void RequestTransferAmount() = 0;
};

class TransferTransaction: public Transaction
{
    public:
        TransferTransaction(TransferUI& ui)
        : itsTransferUI(ui)
        {}

        virtual void Execute()
        {
            ...
            itsTransferUI.RequestTransferAmount();
            ...
        }
    private:
        TransferUI& itsTransferUI;
}

class UI: public DepositUI
```

```
    , public WithdrawalUI
    , public TransferUI
{
    public:
        virtual void RequestDepositAmount();
        virtual void RequestWithdrawalAmount();
        virtual void RequestTransferAmount();
};
```

对程序 12.6 进行仔细的检查，就会发现这个符合 LSP 的解决方案中的一个问题，这个问题在 TimedDoor 例子中是不明显的。请注意，每个操作都必须以某种方式知晓它的特定 UI 版本。DepositTransaction 必须要知道 DepositUI，WithdrawlTransaction 必须要知道 WithdrawlUI 等。在程序 12.6 中，我使每个操作在构造时给它传入指向特定于它的 UI 的引用，从而解决了这个问题。请注意，这使我可以使用程序 12.7 中的惯用法（idiom）。

程序 12.7 接口初始化惯用法

```
UI Gui;     // global object;

Void f();
{
    DepositTransaction dt(Gui);
};
```

虽然这很方便，但是同样要求每个操作都有一个指向对应 UI 的引用成员。另外一种解决这个问题的方法是创建一组全局常量，如程序 12.8 所示。全局变量并不总是意味着拙劣的设计。在这种情况下，它们有着明显的易于访问的优点。由于它们是引用，所以不可能去改变它们。因此，对它们的操作不会出现莫名其妙的情况。

程序 12.8 分离的全局指针

```
// in some module that gets linked in
// to the rest of the app.

static UI Lui;  // non-global object;
DepositUI& GdepositUI = Lui;
WithdrawalUI& GwithdrawalUI = Lui;
TransferUI& GtransferUI = Lui;

// In the depositTransaction.h module

class WithDrawalTransaction: public Transaction
{
    public:

        virtual void Execute()
        {
            ...
            GwithdrawalUI.RequestWithdrawalAmount();
            ...
        }
};
```

在 C++中，可能有人会为了避免对全局名字空间（namespace）的污染而将程序 12.8 中的所有全局变量放到一个单独的类中。程序 12.9 展示了这种方法。然而，该方法有一个负面效果。为了使用 UiGlobals，必须要#include ui_gloabls.h。这依次又#includes depositUI.h、withdrawUI.h 以及 tranferUI.h。这

意味着，想使用任何一个 UI 接口的所有模块，会传递依赖于所有的 UI 接口——而这种情况正是 ISP 原则告诫我们要避免的。如果改动了任何一个 UI 接口，那么所有#include "ui_globals.h"的模块都必须重新编译。UIGlobal 类把我们费了很多力气才分离开的接口又合并了起来！

程序 12.9　把全局变量包装在一个类中

```
# in ui_globals.h

#include "depositUI.h"
#include "withdrawalUI.h"
#include "transferUI.h"

class UIGlobals
{
    public:
        static WithdrawalUI& withdrawal;
        static DepositUI& deposit;
        static TransferUI& transfer;
};

// in ui_globals.cc

static UI Lui;  // non-global object;
DepositUI& UIGlobals::deposit = Lui;
WithdrawalUI& UIGlobals::withdrawal = Lui;
TransferUI& UIGlobals::transfer = Lui;
```

多参数形式与单参数形式

考虑一个既要访问 DepositUI 又要访问 TransferUI 的函数 g。假设我们想把这两个 UI 传入该函数。是应该像这样来编写该函数的原型（prototype）：

```
void g(DepositUI&, TransferUI&);
```

还是应该像这样来编写呢？

```
void g(UI&);
```

以后一种形式（单参数形式）来编写该函数的诱惑是很强的。毕竟，我们知道在前一种多参数形式中，两个参数引用的是同一个对象。而且，如果使用多参数形式，它的调用看起来就像这样：

```
g(ui, ui);
```

这看起来有点有悖常理。

无论是否有悖常理，多参数形式通常都应该优先于单参数形式使用。单参数形式迫使函数 g 依赖于 UI 中包括的每一个接口。这样，如果 withdrawUI 发生了改变，那么函数 g 和 g 的所有客户程序都会受到影响。这比 g(ui, ui)更有悖常理。此外，我们不能保证传入函数 g 的两个参数总是引用同一个对象。也许以后，接口对象会因为某种原因而分离。函数 g 并不需要知道所有的接口都被合并到了单一的对象中这样的事实。因此，对于这样的函数，我更喜欢使用多参数形式。

1. 对客户进行分组

常常可以根据客户所调用的服务方法来对客户进行分组。这种分组方法使得可以为每组而不是每个客户创建分离的接口。这极大地减少了服务需要实现的接口数量，同时也避免让服务依赖于每

个客户类型。

有时，不同的客户组调用的方法会有重叠。如果重叠部分较少，那么组的接口应该保持分离。公用的方法应该在所有有重叠的接口中声明。服务者类会从这些接口的每一个中继承公用的方法，但是只实现它们一次。

2．改变接口

在维护面向对象的应用程序时，常常会改变现有的类和组件的接口。通常这些改变都会造成巨大的影响，并且迫使系统的绝大部分需要重新编译和重新部署。这种影响可以通过为现有的对象增加新接口的方法来缓解，而不是去改变现有的接口。原有接口的客户如果想访问新接口中方法，可以通过对象去询问该接口，如程序 12.10 所示。

程序 12.10

```
void Client(Service* s)
{
    if (NewService* ns = dynamic_cast<NewService *>(s))
    {
        // use the new service interface
    }
}
```

每个原则在应用时都必须小心，不能过度使用它们。如果一个类具有数百个不同的接口，其中一些是根据客户程序分离的，另一些是根据版本分离的，那么该类就是难以琢磨的，这种难以琢磨性是非常令人恐惧的。

12.6 结　论

胖类（fat class）会导致它们的客户程序之间产生不正常的并且有害的耦合关系。当一个客户程序要求该胖类进行一个改动时，会影响到所有其他的客户程序。因此，客户程序应该仅仅依赖于它们实际调用的方法。通过把胖类的接口分解为多个特定于客户程序的接口，可以实现这个目标。每个特定于客户程序的接口仅仅声明它的特定客户或者客户组调用的那些函数。接着，该胖类就可以继承所有特定于客户程序的接口，并实现它们。这就解除了客户程序和它们没有调用的方法间的依赖关系，并使客户程序之间互不依赖。

参考文献

1. Gamma, et al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

第III部分　薪水支付案例研究

本部分开始学习第一个较大的案例。我们已经学习了一些实践和原则，论述了设计的本质，也谈论了测试和计划方面的内容。现在需要做些实际的工作了。

在接下来的几章中，我们将要研究一个批量（batch）处理的薪水支付系统的设计和实现。后面会有一份关于该系统的初步规格说明。在设计和实现的过程中，我们会使用一些不同的设计模式。它们是 COMMAND, TEMPLATE METHOD, STRATEGY, NULL OBJECT, FACTORY 以及 FACADE。这些模式是下面几章的主要议题。在第 18 章，我们会完成薪水支付系统的设计和实现

在学习该案例时，有下面几种可选的方式：

- 按顺序学习，首先学习设计模式，然后再去学习这些模式是怎样应用到薪水支付系统中的。
- 如果已经了解了这些模式并且没有兴趣再复习一遍，那么可以直接学习第 18 章。
- 首先学习第 18 章，然后再返回来去学习讲述第 18 章中用到的模式的章节。
- 逐步地学习第 18 章。当其中谈论到你不熟悉的模式时，再去学习讲述那个模式的章节，接着返回第 18 章继续学习。
- 事实上，不一定非得遵循上述方式。完全可以选择，或者创造最适合自己的学习方式。

薪水支付系统的初步规格说明

下面是和客户交谈时做的一些记录。

该系统由一个公司雇员数据库以及和雇员相关的数据（比如：工作时间卡）组成。该系统必须

为每个雇员支付薪水。系统必须按照规定的方法准时地给雇员支付正确数目的薪水。同时，必须从雇员的薪水中减去各种扣款。

- 有些雇员是钟点工。会按照他们雇员记录中每小时报酬字段的值对他们进行支付。他们每天会提交工作时间卡，其中记录了日期以及工作小时数。如果他们每天工作超过 8 小时，那么超过的部分会按照正常报酬的 1.5 倍进行支付。每周五对他们进行支付。
- 有些雇员完全以月薪进行支付。每个月的最后一个工作日对他们进行支付。在他们的雇员记录中有一个月薪字段。
- 同时，对于一些带薪雇员，会根据他们的销售情况，支付给他们一定数量的酬金（commission）。他们会提交销售凭条，其中记录了销售的日期和数量。在他们的雇员记录中有一个酬金字段。每隔一周的周五对他们进行支付。
- 雇员可以选择支付方式。可以选择把支付支票邮寄到他们指定的邮政地址；也可以把支票保存在出纳人员那里随时支取；或者要求将薪水直接存入他们指定的银行账户。
- 一些雇员会加入协会。在他们的雇员记录中有一个每周应付款项字段。这些应付款必须要从他们的薪水中扣除。协会有时也会针对单个协会成员征收服务费用。协会每周会提交这些服务费用，服务费用必须要从相应雇员的下个月的薪水总额中扣除。
- 薪水支付应用程序每个工作日运行一次，并在当天为相应的雇员进行支付。系统会被告知雇员的支付日期，这样它会计算从雇员上次支付日期到规定的本次支付日期间应支付的数额。

练习

在继续学习前，现在自己来设计一下这个薪水支付系统是有好处的。你也许想画一些初步的 UML 草图。更进一步，你也许想使用测试优先（test-first）的方法去实现一些最初的用例（use case），并应用迄今为止我们已经学过的原则和实践，去创建一个平衡的、良好的设计。

如果你要做这些事情，那么可以看看下面的用例。否则可以跳过它们，我们在薪水支付案例学习的章节中会再次介绍这些用例。

用例 1：增加新雇员

使用 AddEmp 操作（transcation）可以增加新的雇员。该操作包含有雇员的名字、地址以及分配的雇员号。该操作有如下 3 种形式：

```
AddEmp <EmpID> "<name>" "<address>" H <hourly-rate>
AddEmp <EmpID> "<name>" "<address>" S <monthly-salary >
AddEmp <EmpID> "<name>" "<address>" C <monthly-salary> <commission-rate>
```

雇员记录是根据所赋予的对应字段的值来创建的。

异常情况（Alternative）：描述操作的结构中有错误。

如果描述操作的结构不正确，会在一条错误消息中把它打印出来，并且不进行处理。

用例 2：删除雇员

使用 DelEmp 操作来删除雇员。该操作使用如下形式：

```
DelEmp <EmpID>
```

当执行该操作时，会删除对应的雇员记录。

异常情况：无效或者未知的 EmpID。

如果<EmpID>字段不具有正确的结构，或者它没有引用到一条有效的雇员记录，那么会在一条错误消息中把它打印出来，并且不进行其他处理。

用例 3：登记时间卡

执行 TimeCard 操作时，系统会创建一个时间卡记录，并把该记录和对应的雇员记录关联起来。

```
TimeCard <EmpID> <date> <hours>
```

异常情况 1：所选择的雇员不是钟点工。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

异常情况 2：描述操作的结构中有错误。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

用例 4：登记销售凭条

执行 SalesReceipt 操作时，系统会创建一个新的销售凭条记录，并把该记录和对应的应支付酬金的雇员关联起来。

```
SalesReceipt <EmpID> <date> <amount>
```

异常情况 1：所选择的雇员不是应该支付酬金的。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

异常情况 2：描述操作的结构中有错误。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

用例 5：登记协会服务费

执行这个操作时，系统会创建一个服务费用记录，并把该记录和对应的协会成员关联起来。

```
ServiceCharge <memberID> <amount>
```

异常情况：描述操作的结构不是良好组织的。

如果该操作不是良好组织（well formed）的，或者<memberID>引用到一个不存在的协会成员，那么会把该操作在一条适当的错误消息中打印出来。

用例 6：更改雇员明细

执行这个操作时，系统会更改对应雇员记录的详细信息之一。该操作有几个可能的变体：

```
ChgEmp <EmpID> Name <name>                          更改雇员名
ChgEmp <EmpID> Address <address>                    更改雇员地址
ChgEmp <EmpID> Hourly <hourlyRate>                  更改每小时报酬
ChgEmp <EmpID> Salaried <salary>                    更改薪水
ChgEmp <EmpID> Commissioned <salary> <rate>         更改酬金
ChgEmp <EmpID> Hold                                 持有支票
ChgEmp <EmpID> Direct <bank> <account>              直接存款
ChgEmp <EmpID> Mail <address>                       邮寄支票
ChgEmp <EmpID> Member <memberID> Dues <rate>        使雇员加入协会
ChgEmp <EmpID> NoMember                             从协会去掉雇员
```

异常情况：操作错误。

如果描述操作的结构不正确，或者<EmpID>没有引用到真正的雇员，或者<memberID>已经引用了一个成员，那么打印一条适当的错误，并且不进行进一步的处理。

用例 7：现在运行薪水支付系统

执行 Payday 操作时，系统会找到所有应该在指定日期进行支付的雇员。接着系统确定出他们的应扣款额，并根据他们所选择的支付方式对他们进行支付。

```
Payday <date>
```

第 13 章　COMMAND 模式和 ACTIVE OBJECT 模式

没有人天生就具有命令他人的权利。

——Denis Diderot（1713－1784，法国哲学家，百科全书编者）

在近几年记述过的所有设计模式中，我认为 COMMAND 模式是最简单、最优雅的模式之一。但是我们将会看到，这种简单性是带有欺骗性的。COMMAND 模式的适用范围是非常宽广的。

如图 13.1 所示，COMMAND 模式简单得几乎可笑。程序 13.1 中的代码并没有起到削弱这种印象的作用。该模式仅由一个只有惟一方法的接口组成，这似乎是荒谬的。

程序 13.1　Command.java

```
public interface Command
{
  public void do();
}
```

图 13.1　COMMAND 模式

但是，事实上，该模式横过了一条非常有趣的界线。而这个交界处正是所有有趣的复杂性之所在。大多数类都是一组方法和相应的一组变量的结合。COMMAND 模式不是这样的。它只是封装了一个没有任何变量的函数。

从严格的面向对象意义上来讲，这种做法是被强烈反对的——因为它具有功能分解的味道。它把函数层面的任务提升到了类的层面。这简直是对面向对象的亵渎！然而，在这两个思维范式（paradigm）的碰撞处，有趣的事情发生了。

13.1　简单的 COMMAND

几年前，我为一家大型的复印机公司做顾问。我帮助他们的一个开发团队设计和实现一个嵌入

式实时软件，该软件驱动着一种新型复印机的内部工作流。我们无意中发现可以用 COMMAND 模式来控制硬件设备。我们创建了如图 13.2 中所示的层次结构。

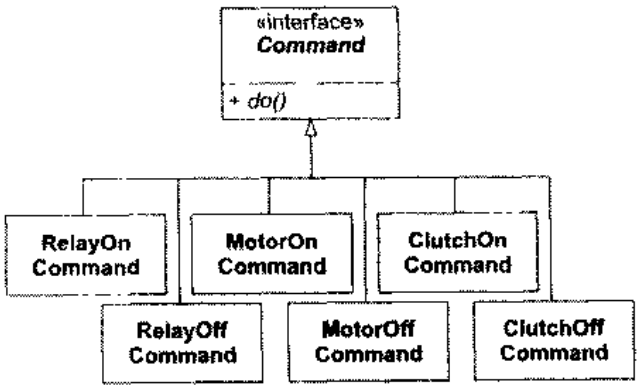


图 13.2　复印机软件中一些简单的 Command

这些类的职责很明显。如果调用 RelayOnCommand 的 do()方法，它就会开启一些继电器。如果调用 MotorOffCommand 的 do()方法，它就会关闭一些发动机。继电器或者发动机的地址作为构造函数的参数传到对象中去。

有了这种结构，我们就可以在系统中传递 Command 对象并调用它们的 do()方法，而无需明确地知道它们所代表的 Command 的种类。这会带来一些有趣的简化。

该系统是事件驱动的。继电器是开还是关，发动机是启动还是停止，离合器是使用还是未使用，都取决于系统中发生的特定事件。在这些事件中，许多是通过传感器检测的。例如，当光学传感器检测到一张纸已经到了传送路径（paper path）中的一个特定点时，就需要启用一个特定的离合器。那么，我们只要把合适的 ClutchOnCommand 绑定到控制那个光学传感器的对象上，就可以实现这个功能了。（参见图 13.3。）

图 13.3　由 Sensor 驱动的 Command

这个简单的结构具有一个巨大的优点。Sensor 不知道它所做的事情。每次当它检测到一个事件时，只需调用它所绑定的 Command 对象的 do()方法即可。这就是说 Sensor 无需知道特有的离合器或者继电器，也无需知道纸张传送装置的机械结构，这样它们的功能就变得相当简单。

当传感器检测到事件后，决定哪些继电器要被关闭的复杂逻辑被移到了一个初始化函数中。在系统初始化的某一时刻，每个传感器都被绑定到对应的 Command 对象上去。这就把所有的连接关系（wiring）[1]放置在一个地方，并使之和系统的主体部分分离。事实上，可以创建一个简单的文本文件来描述 Sensor 和 Command 之间的绑定关系。初始化程序可以读取该文件，并构建出对应的系统。这样，系统中的连接关系可以完全在程序以外确定，并且对它的调整也不会引起重新编译。

通过对命令（command）概念的封装，该模式解除了系统的逻辑互连关系和实际连接的设备之间的耦合。这是一个巨大的好处。

① Sensor 和 Command 之间的逻辑连接。

13.2　事务操作

另外一个 COMMAND 模式的常见用法是创建和执行事务操作（Transactions），该用法在薪水支付（Payroll）问题中很有用。例如，假想我们正在编写一个维护雇员数据库的软件（参见图 13.4）。用户对数据库可以执行许多操作。他们可以增加新雇员，删除老雇员，或者修改现有雇员的属性。

当用户决定增加一个新雇员时，该用户必须详细指明成功创建一条雇员记录所需要的所有信息。在使用这些信息前，系统需要验证这些信息语法和语义上的正确性。COMMAND 模式可以协助完成这项工作。Command 对象存储了还未验证的数据，实现了实施验证的方法，并且实现了最后执行事务操作的方法。

例如，在图 13.5 中。AddEmployeeTransaction 类包含有和 Employee 类相同的数据字段。同时它还持有一个指向 PayClassification 对象的指针。这些数据字段和对象是根据用户指示系统增加一个新雇员时指定的信息创建出来的。

validate 方法检查所有数据并确保数据是有意义的。它检查数据语义和语法上的正确性。它甚至会做一些确保事务操作中的数据和数据库的现有状态一致的检查。例如，它有可能要确保不存在某一雇员。

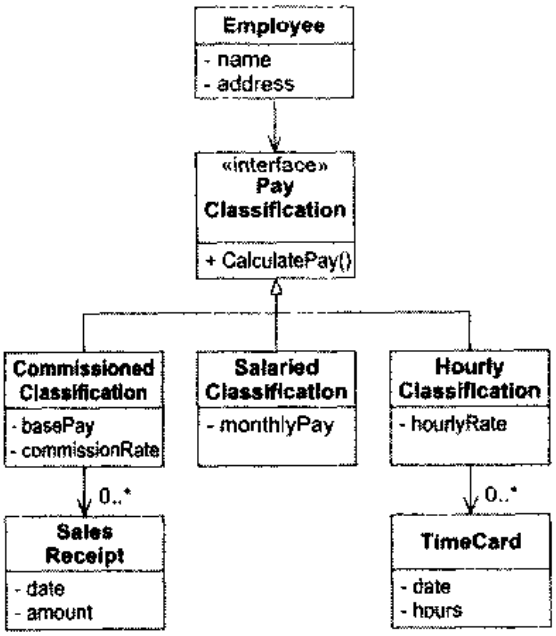


图 13.4　雇员数据库

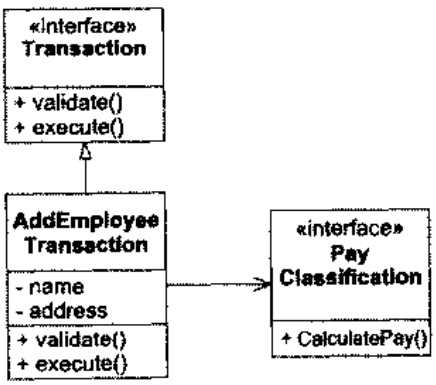


图 13.5　AddEmployee 事务操作

execute 方法用已经验证过的数据去更新数据库。在我们这个简单的例子中，AddEmployeeTransaction 对象创建新的 Employee 对象，并且初始化它的数据成员。PayClassification 对象会被移到或者拷贝到 Employee 对象中。

13.2.1　实体上解耦和时间上解耦

这给我们带来的好处在于很好地解除了从用户获取数据的代码、验证并操作数据的代码以及业

务对象本身之间的耦合关系。例如，可能会有人想通过某些 GUI 中的对话框来获取增加新雇员时需要的数据。如果 GUI 代码中包含了该操作中的验证和执行算法，那么就会很可惜。这样的耦合会使验证和执行代码无法在其他的接口中使用。通过把验证和执行代码分离到 AddEmployeeTransaction 类中，我们从实体上解除了该代码和获取数据的接口间的耦合关系。更甚者，我们也分离了知道如何操作数据库逻辑的代码和业务实体本身

13.2.2　时间上解耦

我们也以一种不同的方式解耦了验证和执行代码。一旦获取了数据，就没有理由要求验证和执行方法立即被调用。可以把事务操作对象放在一个列表中，以后再进行验证和执行。

假设我们有个数据库必须在一天之内保持不变。对数据库的修改只能在夜里 0 点和 1 点之间进行。一直等到午夜，然后匆匆忙忙在 1 点前把所有的命令都输入进去，这是不应该的。如果能够输入所有的命令并当场验证，然后在午夜时再自动执行，就非常方便了。COMMAND 模式使之成为可能。

13.3　UNDO

在图 13.6 中给 COMMAND 模式增加了 undo()方法。显而易见，如果 Command 派生类的 do()方法可以记住它所执行的操作的细节，那么 undo 方法就可以取消这些操作，并把系统恢复到原先的状态。

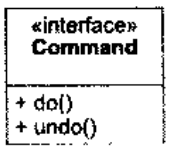
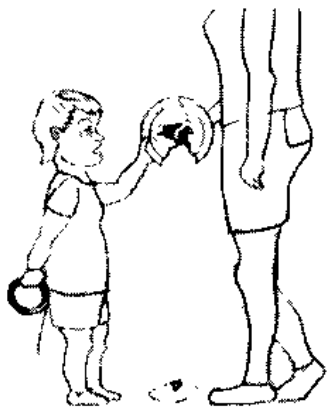


图 13.6　COMMAND 模式的 Undo 变体

例如，假想有一个允许用户在屏幕上画几何图形的应用程序。其中有一个具有一些按钮的工具条，用户可以通过这些按钮去画圆、正方形、矩形等等。用户点击了 draw circle 按钮，系统就创建一个 DrawCircleCommand 对象，并调用了该对象的 do()方法。DrawCircleCommand 对象跟踪用户的鼠标，等待在制图窗口中的一次点击。接收到该点击时，它将这个点击点作为圆心，并且开始以当前鼠标所处的位置到圆心的距离作为半径画动态变化的圆。当用户再次点时，DrawCircleCommand 对象停止动态圆的绘制，并且把相应的圆对象加入到目前在画布上显示的图形对象的列表中。同时，它把这个新圆对象的 ID 作为自己的私有变量存储起来。接着，do（）方法返回。然后，系统把这个执行过的 DrawCommand 对象压入已完成的命令堆栈中。



随后，用户点击了工具条上的 undo 按钮。系统弹出已完成命令堆栈栈顶的 Command 对象，并调用该对象的 undo()方法。接收到 undo()消息时，DrawCircleCommand 对象从当前画布上显示的对象的列表中删除和自己所保存的 ID 匹配的圆。

使用这种技术，可以容易地在几乎所有的应用程序中实现 undo 命令。知道如何去 undo 一个命令的代码几乎总是和知道如何去执行该命令的代码相似。

13.4　ACTIVE OBJECT 模式

ACTIVE OBJECT 模式[①]是我最喜欢使用 COMMAND 模式的地方之一。这是实现多线程控制的一项古老的技术。该模式有多种使用方式，为许多工业系统提供了一个简单的多任务核心。

想法很简单。考虑程序 13.2 和程序 13.3。ActiveObjectEngine 对象维护了一个 Command 对象的链表。用户可以向该引擎（engine）增加新的命令，或者调用 run()。run()函数只是遍历链表，执行并去除每个命令。

程序 13.2　ActiveObjectEngine.java

```
import java.util.LinkedList;
import java.util.Iterator;

public class ActiveObjectEngine
{
  LinkedList itsCommands = new LinkedList();
  public void addCommand(Command c)
  {
    itsCommands.add(c);
  }

  public void run()
  {
    while (!itsCommands.isEmpty())
    {
      Command c = (Command) itsCommands.getFirst();
      itsCommands.removeFirst();
      c.execute();
    }
  }
}
```

程序 13.3　Command.java

```
public interface Command
{
  public void execute() throws Exception;
}
```

这似乎没有给人太深刻的印象。但是想象一下如果链表中的一个 Command 对象会克隆自己并把克隆对象放到链表的尾部，会发生什么呢？这个链表永远不会为空，run()函数永远不会返回。

考虑一下程序 13.4 中的测试用例。它创建了一个 SleepCommand 对象。其中，它向 SleepCommand 的构造函数中传了一个 1000 ms 的延迟。接着把 SleepCommand 对象放入到 ActiveObjectEngine 中。调用了 run()后，它等待指定数目的毫秒。

程序 13.4　TestSleepCommand.java

```
import junit.framework.*;
import junit.swingui.TestRunner;

public class TestSleepCommand extends TestCase
{
  public static void main(String[] args)
```

① [Lavender96]。

```
  {
    TestRunner.main(new String[]{"TestSleepCommand"});
  }

  public TestSleepCommand(String name)
  {
    super(name);
  }

  private boolean commandExecuted = false;

  public void testSleep() throws Exception
  {
    Command wakeup = new Command()
    {
      public void execute() {commandExecuted = true;}
    };
    ActiveObjectEngine e = new ActiveObjectEngine();
    SleepCommand c = new SleepCommand(1000,e,wakeup);
    e.addCommand(c);
    long start = System.currentTimeMillis();
    e.run();
    long stop = System.currentTimeMillis();
    long sleepTime = (stop-start);
    assert("SleepTime " + sleepTime + " expected > 1000", sleepTime > 1000);
    assert("SleepTime " + sleepTime + " expected < 1100", sleepTime < 1100);
    assert("Command Executed", commandExecuted);
  }
}
```

我们来仔细看看这个测试用例。SleepCommand 的构造函数有 3 个参数。第一个是延迟的毫秒数。第二个是在其中运行该命令的 ActiveObjectEngine 对象。最后一个是名为 wakeup 的另一个命令对象。测试的意图是 SleepCommand 会等待指定数目的毫秒，然后执行 wakeup 命令。

程序 13.5 展示了 SleepCommand 的实现。在执行时，SleepCommand 检查自己以前是否已经执行过，如果没有，就记录下开始时间。如果没有过延迟时间，就把自己再加到 ActiveObjectEngine 中。如果过了延迟时间，就把 wakeup 命令对象加到 ActiveObjectEngine 中。

程序 13.5 SleepCommand.java

```
public class SleepCommand implements Command
{
  private Command wakeupCommand = null;
  private ActiveObjectEngine engine = null;
  private long sleepTime = 0;
  private long startTime = 0;
  private boolean started = false;

  public SleepCommand(long milliseconds, ActiveObjectEngine e,
                      Command wakeupCommand)
  {
    sleepTime = milliseconds;
    engine = e;
    this.wakeupCommand = wakeupCommand;
  }

  public void execute() throws Exception
  {
    long currentTime = System.currentTimeMillis();
```

```
    if (!started)
    {
      started = true;
      startTime = currentTime;
      engine.addCommand(this);
    }
    else if ((currentTime - startTime) < sleepTime)
    {
      engine.addCommand(this);
    }
    else
    {
      engine.addCommand(wakeupCommand);
    }
  }
}
```

我们可以对该程序和等待一个事件的多线程程序做一个类比。当多线程程序中的一个线程等待一个事件时，它通常使用一些操作系统调用来阻塞自己直到事件发生。程序 13.5 中的程序并没有阻塞。相反，如果所等待的((currentTime – startTime) < sleeptime)这个事件没有发生，它只是把自己放回到 ActiveObjectEngine 中。

采用该技术的变体（variations）去构建多线程系统已经是并且将会一直是一个很常见的实践。这种类型的线程被称为 run-to-completion 任务（RTC），因为每个 Command 实例在下一个 Command 实例可以运行之前就运行完成了。RTC 的名字意味着 Command 实例不会阻塞。

Command 实例一经运行就一定得完成的特性赋予了 RTC 线程有趣的优点，那就是它们共享同一个运行时堆栈。和传统的多线程系统中的线程不同，不必为每个 RTC 线程定义或者分配各自的运行时堆栈。这在需要大量线程的内存受限系统中是一个强大的优势。

继续我们的例子，程序 13.6 展示一个简单的程序，其中使用了 SleepCommand 并展示了它的多线程行为。该程序被称为 DelayedTyper。

程序 13.6 DelayedTyper.java

```
public class DelayedTyper implements Command
{
  private long itsDelay;
  private char itsChar;
  private static ActiveObjectEngine engine = new ActiveObjectEngine();
  private static boolean stop = false;

  public static void main(String args[]) throws Exception
  {
    engine.addCommand(new DelayedTyper(100,'1'));
    engine.addCommand(new DelayedTyper(300,'3'));
    engine.addCommand(new DelayedTyper(500,'5'));
    engine.addCommand(new DelayedTyper(700,'7'));

    Command stopCommand = new Command()
    {
      public void execute() {stop=true;}
    };

    engine.addCommand(new SleepCommand(20000,engine,stopCommand));
    engine.run();
  }
```

```
  public DelayedTyper(long delay, char c)
  {
    itsDelay = delay;
    itsChar = c;
  }

  public void execute() throws Exception
  {
    System.out.print(itsChar);
    if (!stop)
      delayAndRepeat();
  }

  private void delayAndRepeat() throws CloneNotSupportedException
  {
    engine.addCommand(new SleepCommand(itsDelay,engine,this));
  }

}
```

请注意 DelayedTyper 实现了 Command 接口。它的 execute 方法只是打印出在构造时传入的字符，检查 stop 标志，并在该标志没有被设置时调用 delayAndRepeat。delayAndRepeat 方法使用构造时传入的延迟构造了一个 SleepCommand 对象，再把构造后的 SleepCommand 对象插入 ActiveObjectEngine 中。

该 Command 对象的行为很容易预测。实际上，它维持着一个循环，在循环中重复地打印一个指定的字符并等待一个指定的延迟。当 stop 标志被设置时，就退出循环。

DelayedTyper 的 main 函数创建了几个 DelayedTyper 的实例并把它们放入 ActiveEngine 中，每个实例都有自己的字符和延迟。接着创建了一个 SleepCommand 对象，该对象会在一段时间后设置 stop 标志。运行该程序会打印出一个简单的由‘1’、‘3’、‘5’以及‘7’组成的字符串。再次运行该程序会打印出一个相似，但是有差别的字符串。这里是两次有代表性的运行结果：

```
13571131151137111315113171513111315173111135111371153111135 7...
13571113151317113151131171351113115173111315113171135111311 7...
```

这些字符串之所以有差别是因为 CPU 时钟和实时时钟没有完美的同步。这种不可确定的行为是多线程系统的特点。

13.5 结　论

COMMAND 模式的简单性掩盖了它的多功能性。COMMAND 模式可以应用于多种不同的美妙用途，范围涉及数据库事务操作、设备控制、多线程核心以及 GUI 的 do/undo 管理。

有人认为 COMMAND 模式不符合面向对象的思维范式（paradigm），因为它对函数的关注超过了类。这也许是真的，但是在实际的软件开发中，COMMAND 模式是非常有用的。

参考文献

1. Gamma, et al. *Design Patterns*. Reading, MA: Addision-Wesley, 1995.
2. Lavender, R. G., and D. C. Schmidt. *Active Object: An Object Behavioral Pattern for Concurrent Programming*, in "Pattern Languages of Program Design" (J.O.Coplien, J. Vlissides, and N. kerth, eds.). Reading, MA: Addison-Wesley, 1996.

第 14 章　TEMPLATE METHOD 模式和 STRATEGY 模式：继承与委托

“业精于勤”。

——中国谚语

早在 20 世纪 90 年代初期——也就是面向对象发展的初期——人们就非常看重继承这个概念。继承关系蕴涵的意义是非常深远的。使用继承我们可以基于差异编程（program by difference）！也就是说，对于某个满足了我们大部分需要的类，可以创建一个它的子类，并只改变其中我们不期望的部分。只是继承一个类，就可以重用该类的代码！通过继承，我们可以建立完整的软件结构分类，其中每一层都可以重用该层次以上的代码。这是一个美丽的新世界。

像大多数美丽新世界一样，它最终也被证明有些不切实际。直到 1995 年，人们才清楚地认识到继承非常容易被过度使用，而且过度使用的代价是非常高的。Gamma, Helm, Johnson 和 Vlissides 甚至强调，“优先使用对象组合（object composition）而不是类继承（class inheritance）。”[①]所以我们减少了对继承的使用，常常使用组合或者委托来代替它。

本章讲述了两个模式，并归纳了继承和委托之间的区别。TEMPLATE METHOD 模式和 STRATEGY 模式所要解决的问题是类似的，而且常常可以互换使用。不过，TEMPLATE METHOD 模式使用继承来解决问题，而 STRATEGY 模式使用的则是委托。

TEMPLATE METHOD 模式和 STRATEGY 模式都可以分离通用的算法和具体的上下文。在软件设计中经常会看到这样的需求。我们有一个通用的算法。为了遵循依赖倒置原则（DIP），我们想确保这个通用的算法不要依赖于具体的实现。我们更想使这个通用的算法和具体的实现都依赖于抽象。

14.1　TEMPLATE METHOD 模式

回想一下你编写过的所有程序。其中许多可能都具有如下的基本主循环结构。

```
Initialize();
while(!done()) // main loop
{
    Idle();   // do something useful.
```

① [GOF95]，第 20 页。

```
}
Cleanup();
```

首先进行初始化。接着进入主循环。在主循环中完成需要做的工作，这些工作或许是处理 GUI 事件，或许是处理数据库记录。最后，一旦完成了工作，程序就退出主循环，并且在程序终止前做些清除工作。

这种结构非常常见，所以可以把它封装在一个名为 Application 的类中。之后我们就可以在每个想要编写的新程序中重用这个类。想想！我们再也不需要去编写这个循环了！[①]

例如，在程序 14.1 中，我们看到了这种标准程序的所有组成部分。其中，初始化了 InputStreamReader 和 BufferedReader，并且有一个主循环从 BufferedReader 中读取华氏温度，并把该温度转换成摄氏温度打印出来。最后，打印出一条退出信息。

程序 14.1 ftoc raw

```
import java.io.*;
public class ftocraw
{
    public static void main(String[] args) throws Exception
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        boolean done = false;
        while (!done)
        {
            String fahrString = br.readLine();
            if (fahrString == null || fahrString.length() == 0)
                done = true;
            else
            {
                double fahr = Double.parseDouble(fahrString);
                double celcius = 5.0/9.0*(fahr-32);
                System.out.println("F=" + fahr + ", C=" + celcius);
            }
        }
        System.out.println("ftoc exit");
    }
}
```

这个程序完全符合主循环结构。它先做一些初始化，接着在主循环中完成要做的工作，最后做一些清理工作并退出。

我们可以应用 TEMPLATE METHOD 模式把这个基本结构从 ftoc 程序中分离出来。该模式把所有通用代码放入一个抽象基类（abstract base class）的实现方法中。这个实现方法完成了这个通用算法，但是将所有的实现细节都交付给该基类的抽象方法。

这样，例如，我们可以把这个主循环结构封装在一个名为 Application 的抽象基类中。（参见程序 14.2。）

程序 14.2 Application.java

```
public abstract class Application
{
```

① 我也实现了这个类，并想把它卖给你。

```
  private boolean isDone = false;

  protected abstract void init();
  protected abstract void idle();
  protected abstract void cleanup();

  protected void setDone()
  {isDone = true;}

  protected boolean done()
  {return isDone;}

  public void run()
  {
    init();
    while (!done())
      idle();
    cleanup();
  }
}
```

该类描绘了一个通用的主循环应用程序。从实现的 run 函数中，可以看到主循环。也可以看到所有的工作都被交付给抽象方法 init、idle 以及 cleanup。init 方法处理任何所需的初始化工作；idle 方法处理程序的主要工作，并且在 setDone 方法被调用之前被重复调用；cleanup 方法处理程序退出前所需的所有清理工作。

我们可以通过继承 Application 来重写 ftoc 类，只需要实现 Application 中的抽象方法即可。程序 14.3 展示了重写后的程序。

程序 14.3 ftocTemplateMethod.java

```
import java.io.*;
public class ftocTemplateMethod extends Application
{
  private InputStreamReader isr;
  private BufferedReader br;

  public static void main(String[] args) throws Exception
  {
    (new ftocTemplateMethod()).run();
  }

  protected void init()
  {
    isr = new InputStreamReader(System.in);
    br = new BufferedReader(isr);
  }

  protected void idle()
  {
    String fahrString = readLineAndReturnNullIfError();
    if (fahrString == null || fahrString.length() == 0)
      setDone();
    else
    {
      double fahr = Double.parseDouble(fahrString);
      double celcius = 5.0/9.0*(fahr-32);
      System.out.println("F=" + fahr + ", C=" + celcius);
```

```
    }
  }

  protected void cleanup()
  {
    System.out.println("ftoc exit");
  }

  private String readLineAndReturnNullIfError()
  {
    String s;
    try
    {
      s = br.readLine();
    }
    catch(IOException e)
    {
      s = null;
    }
    return s;
  }
}
```

由于进行了一些异常处理，程序显得略微长了些，但是还是能很容易地看出原先的 ftoc 应用程序是如何适配到 TEMPLATE METHOD 模式上去的。

14.1.1 滥用模式

此时，你应该考虑这样的问题，“他是认真的吗？他真希望我在所有的新应用中都使用这个 Application 类吗？它没有带来任何有价值的东西，只是使问题复杂化了。”

我之所以选择这个例子，是因为它简单，并且为展示 TEMPLATE METHOD 模式的机制提供了一个良好的平台。另一方面，我确实不推荐用这样的方法来构建 ftoc。

这是滥用模式的一个好例子。在这个特定的应用程序中，使用 TEMPLATE METHOD 模式是荒谬的。它使程序变得复杂庞大。把每个应用程序的主循环以一种通用的方式封装起来，一开始这听起来很好，但是本例中的实际应用结果却是无益的。

设计模式是很好的东西。它们可以帮助解决很多设计问题。但是它们的存在并不意味着必须要经常使用它们。本例中，虽然可以应用 TEMPLATE METHOD 模式，但是使用它是不明智的，因为使用该模式的代价要高于它所带来的好处。

来看一个稍微有用些的例子。（参见程序 14.4。）

程序 14.4 BubbleSorter.java

```
public class BubbleSorter
{
  static int operations = 0;
  public static int sort(int [] array)
  {
    operations = 0;
    if (array.length <= 1)
      return operations;
```

```
    for (int nextToLast = array.length-2; nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
        compareAndSwap(array, index);

    return operations;
  }

  private static void swap(int[] array, int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  private static void compareAndSwap(int[] array, int index)
  {
    if (array[index] > array[index+1])
      swap(array, index);
    operations++;
  }
}
```

14.1.2　冒泡排序（Bubble Sort）[①]

BubbleSorter 类知道如何运用冒泡排序算法为一个整数数组排序。BubbleSorter 类的 sort 方法包含了进行冒泡排序的算法。另外两个辅助方法，swap 和 compareAndSwap，用来处理整数和数组的细节问题以及排序算法需要的一些技术方法。

使用 TEMPLATE METHOD 模式，我们可以把冒泡排序算法分离出来，放到一个名为 BubbleSorter 的抽象基类中。BubbleSorter 中实现了 sort 函数，sort 函数调用了名为 outOfOrder 和 swap 的抽象方法。outOfOrder 方法比较数组中的两个相邻元素，如果这两个元素不是按序排列的就返回 true。swap 方法交换数组中两个相邻元素的位置。

sort 方法对数组一无所知，也不关心数组中存放的是何种类型的对象。它只需要用数组的不同下标去调用 outOfOrder 这个方法，然后决定这些下标是否应当交换。（参见程序 14.5。）

程序 14.5　BubbleSorter.java

```
public abstract class BubbleSorter
{
  private int operations = 0;
  protected int length = 0;

  protected int doSort()
  {
    operations = 0;
    if (length <= 1)
      return operations;
```

① 像 Applicaiton 一样，冒泡排序比较易于理解，所以可以作为有用的教学工具。但是，如果排序的量非常大，那么在正常情况下没有人会真的去使用冒泡排序。还有很多更好的算法可用。

```
    for (int nextToLast = length-2; nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
      {
        if (outOfOrder(index))
          swap(index);
        operations++;
      }

    return operations;
  }

  protected abstract void swap(int index);
  protected abstract boolean outOfOrder(int index);
}
```

有了 BubbleSorter 类，现在就可以创建可以为任意不同类型的对象排序的简单派生类。例如，可以创建 IntBubbleSorter 派生类去排序整数数组，创建 DoubleBubbleSorter 派生类去排序双精度型数数组。（参见图 14.1、程序 14.6 和程序 14.7。）

程序 14.6 IntBubbleSorter.java

```
public class IntBubbleSorter extends BubbleSorter
{
  private int[] array = null;
  public int sort(int [] theArray)
  {
    array = theArray;
    length = array.length;
    return doSort();
  }

  protected void swap(int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  protected boolean outOfOrder(int index)
  {
    return (array[index] > array[index+1]);
  }
}
```

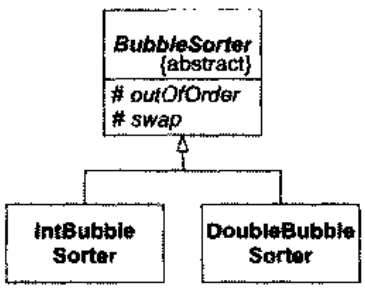


图 14.1 Bubble-Sorter 结构

程序 14.7 DoubleBubbleSorter.java

```
public class DoubleBubbleSorter extends BubbleSorter
{
  private double[] array = null;
  public int sort(double [] theArray)
  {
    array = theArray;
    length = array.length;
    return doSort();
  }

  protected void swap(int index)
  {
    double temp = array[index];
```

```
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  protected boolean outOfOrder(int index)
  {
    return (array[index] > array[index+1]);
  }
}
```

TEMPLATE METHOD 模式展示了面向对象编程中诸多经典重用形式中的一种。其中通用算法被放置在基类中，并且通过继承在不同的具体上下文中实现该通用算法。但是这项技术是有代价的。继承是一种非常强的关系。派生类不可避免地要和它们的基类绑定在一起。

例如，其他类型的排序算法确实也需要 IntBubbleSorter 中的 outOfOrder 和 swap 方法。然而，却没有办法在其他排序算法中重用 outOfOrder 和 swap。由于继承了 BubbleSorter，就注定要把 IntBubbleSorter 永远地和 BubbleSorter 绑定在一起。不过，STRATEGY 模式提供了另一种可选方案。

14.2　STRATEGY 模式

STRATEGY 模式使用了一种非常不同的方法来倒置通用算法和具体实现之间的依赖关系。再来考虑一下滥用模式的 Application 问题。

不是将通用的应用算法放进一个抽象基类中，而是将它放进一个名为 ApplicationRunner 的具体类中。我们把通用算法必须要调用的抽象方法定义在一个名为 Application 的接口中。我们从这个接口派生出 ftocStrategy，并把它传给 ApplicationRunner。之后，ApplicationRunner 就可以把具体工作委托给这个接口去完成。（参见图 14.2、程序 14.8 和程序 14.10。）

程序 14.8　ApplicationRunner.java

```
public class ApplicationRunner
{
  private Application itsApplication = null;

  public ApplicationRunner(Application app)
  {
    itsApplication = app;
  }
  public void run()
  {
    itsApplication.init();
    while (!itsApplication.done())
      itsApplication.idle();
    itsApplication.cleanup();
  }
}
```

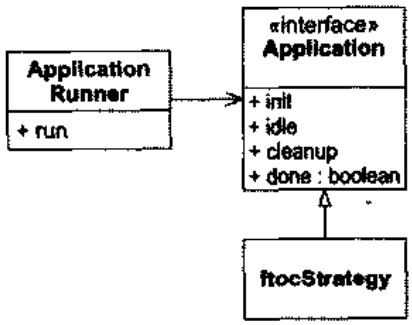


图 14.2　Application 算法策略

程序 14.9　Application.java

```
public interface Application
{
  public void init();
  public void idle();
  public void cleanup();
  public boolean done();
}
```

程序 14.10 ftocStrategy.java

```
import java.io.*;
public class ftocStrategy implements Application
{
  private InputStreamReader isr;
  private BufferedReader br;
  private boolean isDone = false;

  public static void main(String[] args) throws Exception
  {
    (new ApplicationRunner(new ftocStrategy())).run();
  }

  public void init()
  {
    isr = new InputStreamReader(System.in);
    br = new BufferedReader(isr);
  }

  public void idle()
  {
    String fahrString = readLineAndReturnNullIfError();
    if (fahrString == null || fahrString.length() == 0)
      isDone = true;
    else
    {
      double fahr = Double.parseDouble(fahrString);
      double celcius = 5.0/9.0*(fahr-32);
      System.out.println("F=" + fahr + ", C=" + celcius);
    }
  }

  public void cleanup()
  {
    System.out.println("ftoc exit");
  }

  public boolean done()
  {
    return isDone;
  }

  private String readLineAndReturnNullIfError()
  {
    String s;
    try
    {
      s = br.readLine();
    }
    catch(IOException e)
    {
      s = null;
    }
    return s;
  }
}
```

显而易见，这个结构要优于 TEMPLATE METHOD 模式的结构，而使用代价也高一些。

STRATEGY 模式比 TEMPLATE METHOD 模式涉及更多数量的类和间接层次。ApplicationRunner 中委托指针的使用招致了比继承稍微多一点的运行时间和数据空间开销。但是另一方面，如果有许多不同的应用程序要运行，就可以重用 ApplicationRunner 实例，并把许多不同的 Application 实现传递给它，从而减小了通用算法和该算法所控制的具体细节之间的耦合。

这些代价和利益都不是最重要的。在大多数情况下，它们甚至无关紧要。典型情况下，最烦人的问题是 STRATEGY 模式需要的那些额外类。然而，还有更多需要考虑的问题。

再次排序

考虑一下用 STRATEGY 模式来实现冒泡排序。（参见程序 14.11 至程序 14.13。）

程序 14.11　BubbleSorter.java

```
public class BubbleSorter
{
  private int operations = 0;
  private int length = 0;
  private SortHandle itsSortHandle = null;

  public BubbleSorter(SortHandle handle)
  {
    itsSortHandle = handle;
  }

  public int sort(Object array)
  {
    itsSortHandle.setArray(array);
    length = itsSortHandle.length();
    operations = 0;
    if (length <= 1)
      return operations;

    for (int nextToLast = length-2; nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
      {
        if (itsSortHandle.outOfOrder(index))
          itsSortHandle.swap(index);
        operations++;
      }

    return operations;
  }
}
```

程序 14.12　SortHandle.java

```
public interface SortHandle
{
  public void swap(int index);
  public boolean outOfOrder(int index);
  public int length();
  public void setArray(Object array);
}
```

程序 14.13 IntSortHandle.java

```
public class IntSortHandle implements SortHandle
{
  private int[] array = null;

  public void swap(int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  public void setArray(Object array)
  {
    this.array = (int[])array;
  }

  public int length()
  {
    return array.length;
  }

  public boolean outOfOrder(int index)
  {
    return (array[index] > array[index+1]);
  }
}
```

请注意 IntSortHandle 类对 BubbleSorter 类一无所知。它不依赖于冒泡排序的任何实现方式。这和 TEMPLATE METHOD 模式是不同的。回顾一下程序 14.6，可以看到 IntBubbleSorter 直接依赖于 BubbleSorter，而 BubbleSorter 中包含着冒泡排序算法。

由于 swap 和 outOfOrder 方法的实现直接依赖于冒泡排序算法，所以 TEMPLATE METHOD 方法部分地违反了 DIP。而 STRATEGY 方法中不包含这样的依赖。因此可以在 BubbleSorter 之外的其他 Sorter 实现中使用 IntSortHandle。

例如，可以创建冒泡排序的一个变体，如果它在一次对于数组的遍历中发现数组的元素已经是按序排列的话，就提前结束（参见程序 14.14）。QuickBubbleSorter 同样可以使用 IntSortHandle，或者任何其他从 SortHandle 派生出来的类。

程序 14.14 QuickBubbleSorter.java

```
public class QuickBubbleSorter
{
  private int operations = 0;
  private int length = 0;
  private SortHandle itsSortHandle = null;

  public QuickBubbleSorter(SortHandle handle)
  {
    itsSortHandle = handle;
  }

  public int sort(Object array)
  {
    itsSortHandle.setArray(array);
```

```
    length = itsSortHandle.length();
    operations = 0;
    if (length <= 1)
      return operations;

    boolean thisPassInOrder = false;
    for (int nextToLast = length-2; nextToLast >= 0 &&
            !thisPassInOrder; nextToLast--)
    {
      thisPassInOrder = true; //potenially.
      for (int index = 0; index <= nextToLast; index++)
      {
        if (itsSortHandle.outOfOrder(index))
        {
          itsSortHandle.swap(index);
          thisPassInOrder = false;
        }
        operations++;
      }
    }

    return operations;
  }
}
```

因此，STRATEGY 模式比 TEMPLATE METHOD 模式多提供了一个额外的好处。尽管 TEMPLATE METHOD 模式允许一个通用算法操纵多个可能的具体实现，但是由于 STRATEGY 模式完全遵循 DIP 原则，从而允许每个具体实现都可以被多个不同的通用算法操纵。

14.3 结　论

TEMPLATE METHOD 模式和 STRATEGY 模式都可以用来分离高层的算法和低层的具体实现细节。都允许高层的算法独立于它的具体实现细节重用。此外，STRATEGY 模式也允许具体实现细节独立于高层的算法重用，不过要以一些额外的复杂性、内存以及运行时间开销作为代价。

参考文献

1. Gamma, et al. *Design Patterns*, MA: Addison-Wesley, 1995.
2. Martin, Robert C., et al. *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley, 1998.

第 15 章　FACADE 模式和 MEDIATOR 模式

尊贵的符号外表下，隐藏着卑劣的梦想。

——Mason Cooley

本章中论述的两个模式有着共同的目的。它们都把某种策略（policy）施加到另外一组对象上。FACADE 模式从上面施加策略，而 MEDIATOR 模式则从下面施加策略。FACADE 模式的使用是明显且受限的，而 MEDIATOR 模式的使用则是不明显且不受限制的。

15.1　FACADE 模式

当想要为一组具有复杂且全面的接口的对象提供一个简单且特定的接口时，可以使用 FACADE 模式。例如，考虑程序 26.9 中的 DB.java。该类为 java.sql 包中复杂且全面的类接口提供了一个非常简单的、特定于 ProductData 的接口。图 15.1 展示了这个结构。

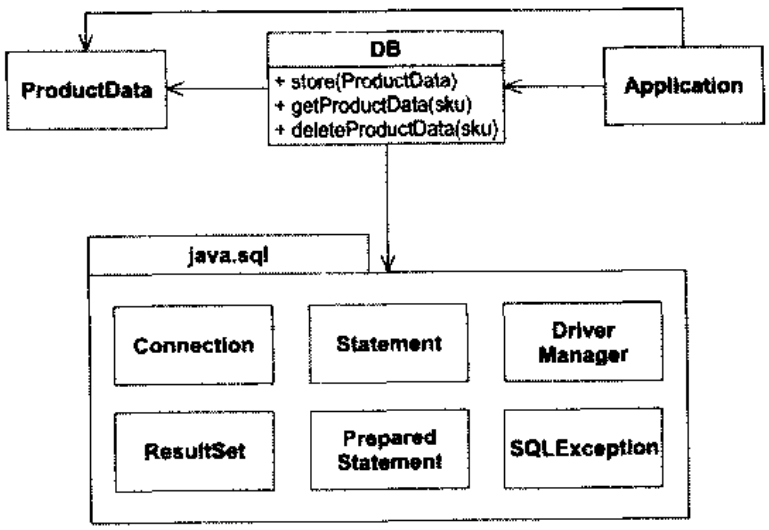


图 15.1　DB FACADE

请注意，DB 类使得 Application 类不需要了解 java.sql 包的内部细节。它把 java.sql 包的所有全面性和复杂性隐藏在一个非常简单且特定的接口后面。

像 DB 这样的 FACADE 类对 java.sql 包的使用施加了许多策略。它知道如何去初始化和关闭数据库连接。它知道如何将 ProductData 的成员变量转换成数据库字段，或反之。它知道如何去构建合适的查询和命令去操纵数据库。它对用户隐藏了所有的复杂性。在 Application 看来，java.sql 包是不存在的：它隐藏在 FACADE 后面。

使用 FACADE 模式意味着开发人员已经接受了所有数据库调用都要通过 DB 类的约定。如果 Applilcation 的任意一部分代码越过该 FACADE 直接去访问 java.sql，那么就违反了该约定。像这样，该 FACADE 对 application 施加了它的策略。基于约定，DB 类成为了 java.sql 包的惟一代理（broker）。

15.2　MEDIATOR 模式

MEDIATOR 模式同样也施加策略。不过，FACADE 模式是以明显且受限的方式来施加它的策略，而 MEDIATOR 模式则是以隐藏且不受限的方式来施加它的策略。例如，程序 15.1 中的 QuickEntryMediator 类是一个安静地呆在幕后的类，它把文本输入域绑定在 list 上。当在文本输入域中键入时，和输入匹配的 list 中的第一个元素会高亮显示。这样，无需完全输入即可快速选取 list 项。

程序 15.1　QuickEntryMediator.java

```
package utility;

import javax.swing.*;
import javax.swing.event.*;

/**
QuickEntryMediator.  This class takes a JTextField and a JList.
It assumes that the user will type characters into the JTextField
that are prefixes of entries in the JList.  It automatically selects
the first item in the JList that matches the current prefix in the
JTextField.

If the JTextField is null, or the prefix does not match any element
in the JList, then the JList selection is cleared.

There are no methods to call for this object.  You simply create it,
and forget it.  (But don't let it be garbage collected...)

Example:

JTextField t = new JTextField();
JList l = new JList();

QuickEntryMediator qem = new QuickEntryMediator(t, l);
 // that's all folks.

 @author Robert C. Martin, Robert S. Koss
 @date 30 Jun, 1999 2113 (SLAC)
 */

public class QuickEntryMediator {
  public QuickEntryMediator(JTextField t, JList l) {
    itsTextField = t;
```

```
    itsList = l;

    itsTextField.getDocument().addDocumentListener(
      new DocumentListener() {
        public void changedUpdate(DocumentEvent e) {
          textFieldChanged();
        }

        public void insertUpdate(DocumentEvent e) {
          textFieldChanged();
        }

        public void removeUpdate(DocumentEvent e) {
          textFieldChanged();
        }
      } // new DocumentListener
    ); // addDocumentListener
  } // QuickEntryMediator()

  private void textFieldChanged() {
    String prefix = itsTextField.getText();

    if (prefix.length() == 0) {
      itsList.clearSelection();
      return;
    }

    ListModel m = itsList.getModel();
    boolean found = false;
    for (int i = 0; found == false && i < m.getSize(); i++) {
      Object o = m.getElementAt(i);
      String s = o.toString();
      if (s.startsWith(prefix)) {
        itsList.setSelectedValue(o, true);
        found = true;
      }
    }

    if (!found) {
      itsList.clearSelection();
    }
  } // textFieldChanged

  private JTextField itsTextField;
  private JList itsList;

} // class QuickEntryMediator
```

图 15.2 中展示了 QuickEntryMediator 的结构。用一个 JList 和一个 JTextField 构造了一个 QuickEntryMediator 类的实例。QuickEntryMediator 向 JTextField 注册了一个匿名的 DocumentListener。每当文本发生变化时，这个 listener 就调用 textFieldChanged 方法。接着，该方法在 JList 中查找以这个文本为前缀的元素并选中它。

JList 和 JTextField 的使用者并不知道该 MEDIATOR 的存在。它安静地呆着，把它的策略施加在那些对象上，而无需它们的允许或者知晓。

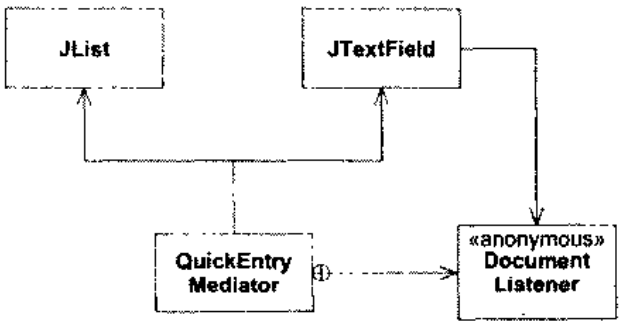


图 15.2　QuickEntryMediator

15.3　结　　论

如果策略涉及范围广泛并且可见，那么可以使用 FACACE 模式从上面施加该策略。另一方面，如果策略隐蔽并且有针对性，那么 MEDIATOR 模式是更好的选择。Facades 通常是约定的关注点。每个人都同意去使用该 facade 而不是隐藏于其下的对象。另一方面，Mediator 则对用户是隐藏的。它的策略是既成事实的而不是一项约定事务。

参考文献

1．Gamma, et al. *Design Patterns*. Reading, MA:Addison-sesley, 1995.

第 16 章　SINGLETON 模式和 MONOSTATE 模式

“这是对万物的无限祝福！除此之外再无其他”

——The point. Flatland. 爱德温·A·艾伯特

类和它们的实例间通常是一对多的关系。对于大多数的类来说，都可以创建多个实例。在需要这些实例时创建它们，在这些实例不再有用时删除它们。这些实例的来去伴随着内存的分配和归还。

然而有一些类，它们应该只有一个实例。这个实例似乎应当在程序启动时被创建出来，并且只在程序结束时才被删除。有时，这种对象是应用程序的基础（root）对象。通过这些基础对象可以得到系统中的许多其他对象。有时，它们是工厂对象（factories），用来创建系统中的其他对象。有时，这些对象是管理器（managers）对象，负责管理某些其他对象并以合适的方式去控制它们。

不管这些对象是什么，只要创建了多份，就是严重的逻辑错误。如果创建了多份基础对象，那么对应用程序中对象的访问就依赖于所选择的那个基础对象。对于不知道存在多个基础对象的程序员来说，他们可能不知道自己看到的只是应用程序对象的一个子集。如果存在多份厂对象，那么对它所创建对象的控制工作就会遭到破坏。如果存在多份管理器对象，那么本来打算串行的行为就可能会变成并发的。

那些强制对象单一性的机制似乎有些多余。毕竟，在初始化应用程序时，完全可以只创建每个对象的一个实例，然后使用该实例。事实上，这通常也是最好的方法。在没有急迫并且有意义的需要时，应该避免使用这些机制。不过，我们也希望代码能够传达我们的意图。如果强制对象单一性的机制是轻量级的，那么传达意图带来的收益就会胜过实施这些机制的代价。

本章讲述了两个强制对象单一性的模式。这两个模式有着非常不同的“代价/ 收益”权衡。在大多数情况下，它们的实施代价远低于它们的表达力带来的收益。

16.1　SINGLETON 模式[①]

SINGLETON 是一个很简单的模式。程序 16.1 中的测试用例演示了它应该如何工作。第一个测试函数表明了 Singleton 实例是通过公有的静态方法 Instance 访问的。它同样也表明了即使 Instance 方法被多次调用，每次返回的都是指向完全相同的实例的引用。第二个测试用例表明 Singleton 类没有公有构造函数，所以如果不使用 Instance 方法，就无法去创建它的实例。

程序 16.1　Singleton 测试用例

```
import junit.framework.*;
import java.lang.reflect.Constructor;

public class TestSimpleSingleton extends TestCase
{
    public TestSimpleSingleton(String name)
    {
        super(name);
    }

    public void testCreateSingleton()
    {
        Singleton s = Singleton.Instance();
        Singleton s2 = Singleton.Instance();
        assertSame(s, s2);
    }

    public void testNoPublicConstructors() throws Exception
    {
        Class singleton = Class.forName("Singleton");
        Constructor[] constructors = singleton.getConstructors();
        assertEquals("Singleton has public constructors.",
                     0, constructors.length);
    }
}
```

这个测试用例是 SINGLETON 模式的规格说明（specification）。它直接指导我们写出程序 16.2 中所示的代码。通过观察代码，应该可以很清楚地看出，在静态变量 Singleton.theInstance 的作用范围内 Singleton 类的实例决不会超过一个。

程序 16.2　Singleton 实现

```
public class Singleton
{
    private static Singleton theInstance = null;
    private Singleton() {}

    public static Singleton Instance()
    {
        if (theInstance == null)
            theInstance = new Singleton();
        return theInstance;
    }
}
```

① [GOF95]，第 127 页。

16.1.1 SINGLETON 模式的好处

- **跨平台**：使用合适的中间件（例如，RMI），可以把 SINGLETON 模式扩展为跨多个 JVM 和多个计算机工作。
- **适用于任何类**：只需把一个类的构造函数变成私有的，并且在其中增加相应的静态函数和变量，就可以把这个类变为 SINGLETON。
- **可以透过派生创建**：给定一个类，可以创建它的一个 SINGLETON 子类。
- **延迟求值**（Lazy evaluation）：如果 SINGLETON 从未使用过，那么就决不会创建它。

16.1.2 SINGLETON 模式的代价

- **摧毁方法未定义**：没有好的方法去摧毁（destroy）一个 SINGLETON，或者解除其职责。即使添加一个 decommission 方法把 theInstance 置为 null，系统中的其他模块仍然持有对该 SINGLETON 实例的引用。这样，随后对 Instance 方法的调用会创建另外一个实例，致使同时存在两个实例。这个问题在 C++中尤为严重，因为实例可以被摧毁，可能会导致去提领（dereference）一个已被摧毁的对象。
- **不能继承**：从 SINGLETON 类派生出来的类并不是 SINGLETON。如果要使其成为 SINGLETON，必须要增加所需的静态函数和变量。
- **效率问题**：每次调用 Instance 方法都会执行 if 语句。就大多数调用而言，if 语句是多余的。
- **不透明性**：SINGLETON 的使用者知道它们正在使用一个 SINGLETON，因为它们必须要调用 Instance 方法。

16.1.3 运用 SINGLETON 模式

假设有一个基于 Web 的系统，它允许用户登录进入一个 Web 服务器的受保护的区域。这样的系统会有一个包含用户名、口令以及其他用户属性的数据库。进一步假设这个数据库是通过第三方 API 进行访问的。我们可以在每个需要读写用户信息的模块中直接访问数据库。然而，这样会使得对第三方 API 的使用分散在整个代码中，并且也无法去强制实施一些访问或者结构方面的约定。

一个比较好的解决方案是运用 FACADE 模式创建一个 UserDatabase 类，该类中包含有读写 User 对象的方法。这些方法调用访问数据库的第三方 API，并执行 User 对象和数据库的表、行之间的转换工作。在 UserDatabase 类中，我们可以强制实施访问和结构方面的约定。例如，我们可以保证一条 User 记录除非拥有非空的 username，否则不予写入数据库。或者还可以把对同一条 User 记录的访问串行化，确保两个模块不会同时去读写它。

程序 16.3 和程序 16.4 中的代码展示了一个使用 SINGLETON 模式的解决方案。SINGLETON 类的名字是 UserDatabaseSource。它实现了 UserDatabase 接口。请注意静态方法 instance()并没有像惯常的那样用 if 语句来避免多次创建，而是利用了 Java 语言的初始化功能。

程序 16.3 UserDatabase 接口

```
public interface UserDatabase
{
    User readUser(String userName);
    void writeUser(User user);
}
```

程序 16.4　UserDatabaseSource Singleton

```
public class UserDatabaseSource implements UserDatabase
{
    private static UserDatabase theInstance = new UserDatabaseSource();

    public static UserDatabase instance()
    {
        return theInstance;
    }

    private UserDatabaseSource()
    {
    }

    public User readUser(String userName)
    {
        // Some Implementation
        return null;  // just to make it compile.
    }

    public void writeUser(User user)
    {
        // Some Implementation
    }
}
```

这是 SINGLETON 模式的一种非常常见的应用。它确保了所有对数据库的访问都通过 UserDatabaseSource 类的单一实例进行。这样就可以容易地在 UserDatabaseSource 类中放入检查、计数、锁等机制来强制实施前面提到的访问以及结构方面的约定。

16.2　MONOSTATE 模式[①]

MONOSTATE 模式是另外一种获取对象单一性的方法。它使用了一种完全不同的工作机制。通过学习程序 16.5 中 MONOSTATE 的测试用例，可以了解它的工作原理。

第一个测试函数只是测试了可以设置和获取一个对象的成员变量 x。但是第二个测试用例显示出同一个类的两个实例表现得就好像是一个一样。如果把一个实例中的成员变量 x 设置成某个特定值，那么可以通过获取另外一个实例的成员变量 x 来得到这个特定值。这两个实例就好像是具有不同名字的同一个对象一样。

程序 16.5　Monostate 测试用例

```
import junit.framework.*;

public class TestMonostate extends TestCase
{
    public TestMonostate(String name)
    {
        super(name);
    }

    public void testInstance()
```

① [BALL2000]。

```
    {
        Monostate m = new Monostate();
        for (int x = 0; x<10; x++)
        {
            m.setX(x);
            assertEquals(x, m.getX());
        }
    }

    public void testInstancesBehaveAsOne()
    {
        Monostate m1 = new Monostate();
        Monostate m2 = new Monostate();

        for (int x = 0; x<10; x++)
        {
            m1.setX(x);
            assertEquals(x, m2.getX());
        }
    }
}
```

如果把 Singleton 类放到这个测试用例中，并把所有的 new Monostate 语句替换为对 Singleton.Instance 的调用，这个测试用例应该仍然可以通过。所以这个测试用例描述了没有强加单一实例约束条件的 Singleton 的行为。

怎样才能使两个实例表现得象一个对象一样呢？很简单，这意味着两个对象必须共享相同的变量。这一点很容易办到，只要把所有的变量都变成静态变量即可。程序 16.6 中展示了 Monostate 的实现，该实现可以通过上面的测试用例。请注意 itsX 变量是静态的，而且所有的方法都不是静态的。这一点很重要，随后我们将会看到。

程序 16.6 Monostate 实现

```
public class Monostate
{
    private static int itsX = 0;
    public Monostate() {}

    public void setX(int x)
    {
        itsX = x;
    }

    public int getX()
    {
        return itsX;
    }
}
```

我发现这是一个令人高兴的变形（twisted）模式。无论创建了多少 Monostate 的实例，它们都表现得像一个对象一样。甚至把当前的所有实例都销毁或者解除职责，也不会丢失数据。

请注意这两个模式之间的区别，在于一个关注行为，而另一个关注结构。SINGLETON 模式强制结构上的单一性。它防止创建出多个对象实例。相反，MONOSTATE 模式则强制行为上的单一性，而没有强加结构方面的限制。为了强调这个区别，请考虑如下事实：MONOSTATE 的测试用例对 Singleton 类是有效的，但是 SINGLETON 的测试用例却远不适用于 Monostate 类。

16.2.1　MONOSTATE 模式的好处

- **透明性**：使用 Monostate 对象和使用常规（regular）对象没有什么区别。使用者不需要知道对象是 MONOSTATE。
- **可派生性**：MONOSTATE 的派生类都是 MONOSTATE。事实上，MONOSTATE 的所有派生类都是同一个 MONOSTATE 的一部分。它们共享相同的静态变量。
- **多态性**：由于 MONOSTATE 的方法不是静态的，所以可以在派生类中覆写（override）它们。因此，不同的派生类可以基于同样的静态变量表现出不同的行为。

16.2.2　MONOSTATE 模式的代价

- **不可转换性**：不能透过派生把常规类转换成 MONOSTATE 类。
- **效率问题**：因为 MONOSTATE 是真正的对象，所以会导致许多的创建和摧毁开销。
- **内存占用**：即使从未使用 MONOSTATE，它的变量也要占据内存空间。
- **平台局限性**：MONOSTATE 不能跨多个 JVM 或者多个平台工作。

16.2.3　运用 MONOSTATE 模式

考虑一下图 16.1，这是实现地铁十字转门的简单的有限状态机。十字转门开始时处于 Locked 状态。如果投入一枚硬币，它就迁移到 Unlocked 状态，开启转门，复位可能出现的任何告警状态，并把硬币放到收集箱柜中。如果此时乘客通过了转门，转门就迁移回 Locked 状态并且把门锁上。

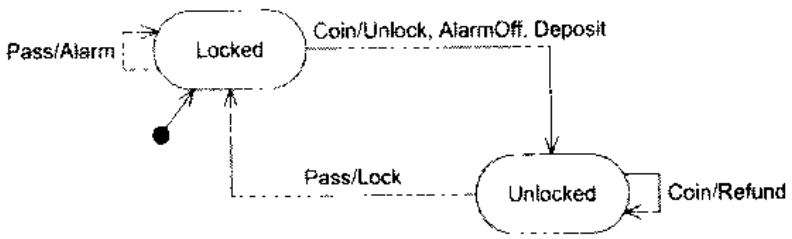


图 16.1　地铁十字转门有限状态机

有两种反常情况存在。如果乘客在通过转门前投入了两枚或者更多的硬币，那么多余的硬币会被退还，并且转门保持在 Unlocked 状态。如果乘客没有投币就想通过转门，那么会发出警报，并且转门保持在 Locked 状态。

程序 16.7 中展示了描述该操作的测试程序。请注意，测试函数假定 Turnstile 是一个 monostate。测试程序期望可以向一些 Turnstile 实例发送事件并从不同的实例中查询结果。如果 Turnstile 类根本不会有多于一个的实例，那么这种期望是非常合理的。

程序 16.7　TestTurnstile

```
import junit.framework.*;

public class TestTurnstile extends TestCase
{
    public TestTurnstile(String name)
    {
```

```
        super(name);
    }

    public void setUp()
    {
        Turnstile t = new Turnstile();
        t.reset();
    }

    public void testInit()
    {
        Turnstile t = new Turnstile();
        assert(t.locked());
        assert(!t.alarm());
    }

    public void testCoin()
    {
        Turnstile t = new Turnstile();
        t.coin();
        Turnstile t1 = new Turnstile();
        assert(!t1.locked());
        assert(!t1.alarm());
        assertEquals(1, t1.coins());
    }

    public void testCoinAndPass()
    {
        Turnstile t = new Turnstile();
        t.coin();
        t.pass();

        Turnstile t1 = new Turnstile();
        assert(t1.locked());
        assert(!t1.alarm());
        assertEquals("coins", 1, t1.coins());
    }

    public void testTwoCoins()
    {
        Turnstile t = new Turnstile();
        t.coin();
        t.coin();

        Turnstile t1 = new Turnstile();
        assert("unlocked", !t1.locked());
        assertEquals("coins",1, t1.coins());
        assertEquals("refunds", 1, t1.refunds());
        assert(!t1.alarm());
    }

    public void testPass()
    {
        Turnstile t = new Turnstile();
        t.pass();
        Turnstile t1 = new Turnstile();
        assert("alarm", t1.alarm());
        assert("locked", t1.locked());
    }
```

```
    public void testCancelAlarm()
    {
        Turnstile t = new Turnstile();
        t.pass();
        t.coin();
        Turnstile t1 = new Turnstile();
        assert("alarm", !t1.alarm());
        assert("locked", !t1.locked());
        assertEquals("coin", 1, t1.coins());
        assertEquals("refund", 0, t1.refunds());
    }

    public void testTwoOperations()
    {
        Turnstile t = new Turnstile();
        t.coin();
        t.pass();
        t.coin();
        assert("unlocked", !t.locked());
        assertEquals("coins", 2, t.coins());
        t.pass();
        assert("locked", t.locked());
    }
}
```

程序 16.8 中是 monostate 类 Turnstile 的实现。基类 Turnstile 把两个事件函数（coin 和 pass）委托给两个 Turnstile 的派生类（Locked 和 Unlocked），这两个派生类代表了有限状态机的状态。

程序 16.8　Turnstile

```
public class Turnstile
{
    private static boolean isLocked = true;
    private static boolean isAlarming = false;
    private static int itsCoins = 0;
    private static int itsRefunds = 0;
    protected final static Turnstile LOCKED = new Locked();
    protected final static Turnstile UNLOCKED = new Unlocked();
    protected static Turnstile itsState = LOCKED;

    public void reset()
    {
        lock(true);
        alarm(false);
        itsCoins = 0;
        itsRefunds = 0;
        itsState = LOCKED;
    }

    public boolean locked()
    {
        return isLocked;
    }

    public boolean alarm()
    {
        return isAlarming;
    }
```

```
    public void coin()
    {
        itsState.coin();
    }

    public void pass()
    {
        itsState.pass();
    }

    protected void lock(boolean shouldLock)
    {
        isLocked = shouldLock;
    }

    protected void alarm(boolean shouldAlarm)
    {
        isAlarming = shouldAlarm;
    }

    public int coins()
    {
        return itsCoins;
    }

    public int refunds()
    {
        return itsRefunds;
    }

    public void deposit()
    {
        itsCoins++;
    }

    public void refund()
    {
        itsRefunds++;
    }
}

  class Locked extends Turnstile
{
    public void coin()
    {
        itsState = UNLOCKED;
        lock(false);
        alarm(false);
        deposit();
    }

    public void pass()
    {
        alarm(true);
    }
}

class Unlocked extends Turnstile
{
    public void coin()
```

```
    {
        refund();
    }

    public void pass()
    {
        lock(true);
        itsState = LOCKED;
    }
}
```

这个例子展示了 MONOSTATE 模式的一些有用的特征。其中利用了 MONOSTATE 的派生对象具有多态的能力以及这些派生对象本身也是 MONOSTATE 的事实。这个例子同样也说明了有时要把 MONOSTATE 变成常规类是多么困难。该解决方案的结构非常强地依赖于 Turnstile 类的 MONOSTATE 本质。如果需要用这个状态机去控制多个十字转门，那么就需要对代码做相当大的重构。

也许，你在担心这个例子中对继承的非常规使用。让 Unlocked 和 Locked 从 Turnstile 派生似乎违反了常规的面向对象原则。不过，由于 Turnstile 是一个 MONOSTATE，所以它的实例之间没有区别。这样，Unlocked 和 Locked 实际上也不是不同的对象。相反，它们都是 Turnstile 抽象的一部分。Unlocked、Locked 以及 Turnstile 访问的是相同的变量和方法。

16.3 结　论

常常会有必要强制要求某个特定对象只能具有单一实例。本章展示了两种非常不同的技术。SINGLETON 模式使用私有构造函数，一个静态变量，以及一个静态方法对实例化进行控制和限制。MONOSTATE 模式只是简单地把对象的所有变量变成静态的。

如果希望透过派生去约束一个现存类，并且不介意它的所有调用者都必须要调用 instance()方法来获取访问权，那么 SINGLETON 是最合适的。如果希望类的单一性本质对使用者透明，或者希望使用单一对象的多态派生对象，那么 MONOSTATE 是最适合的。

参考文献

1. Gamma, et al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
2. Martin, Robert C., et al. *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley, 1998
3. Ball, Steve, and John Crawford, Monostate Classes: The Power of One. Published in *More C++ Gems*, compliles by Robert C. Martin. Cambridge, UK: Cambridge University Press, 2000, p.223

第 17 章　NULL OBJECT 模式

残缺即是完美，冷淡即是礼仪，壮观即是虚无，死亡即是圆满，没有即是更多。

——阿尔弗雷德·丁尼生（1809－1892，19 世纪英国著名诗人）

考虑如下代码：

```
Employee e = DB.getEmployee("Bob");
if (e != null && e.isTimeToPay(today))
   e.pay();
```

我们要从数据库中获取名为“Bob”的 Employee 对象。如果该对象不存在，DB 对象就返回 null；否则，就返回请求的 Employee 实例。如果雇员存在，并且到了他的发薪日，就调用 pay 方法。

我们以前都曾经编写过类似这样的代码。代码采用的惯用法（idiom）很常见，因为在 C-based 语言中，&&的第一个表达式会被首先求值，而仅当第一个表达式为 true 时才会对第二个表达式求值。大多数人也曾经由于忘记对 null 进行检查而受挫。该惯用法虽然很常见，但却是丑陋且易出错的。

通过让 DB.getEmployee 抛出一个异常而不是返回 null，可以减少出错的可能。不过，try/catch 块比对 null 的检查更加丑陋。更糟的是，使用异常就必须要在 throws 子句中声明这些异常。这使得在现有的应用程序中加入异常变得困难。

可以使用 NULL OBJECT 模式[①]来解决这些问题。通常，该模式会消除对 null 进行检查的需要，并且有助于简化代码。

图 17.1 展示了该模式的结构。Employee 变成了一个具有两个实现的接口。EmployeeImplementation 是正常的实现。它包含了 Employee 对象被期望拥有的所有方法和变量。当 DB.getEmployee 在数据库

① [PLOPD3]，第 5 页。这篇令人愉快的文章的作者是 Bobby Woolf，其中充满了智慧、反讽以及实用的建议。

中找到一个雇员时，就返回一个 EmployeeImplementation 的实例。仅当 DB.getEmployee 在数据库中没有找到雇员时才返回 NullEmployee 的实例。

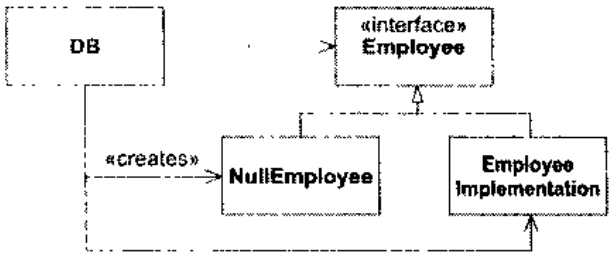


图 17.1　NULL OBJECT 模式

NullEmployee 实现了 Employee 的所有方法，方法中“什么也没做”。“什么也没做”的含意和具体的方法有关。例如，有人会期望 isTimeToPay 方法被实现为返回 false，因为根本不会为 NullEmployee 支付薪水。

使用这个模式，最初的代码可以改为这样：

```
Employee e = DB.getEmployee("Bob");
if (e.isTimeToPay(today))
   e.pay();
```

这种做法既不易于出错又不丑陋，并且具有很好的一致性。DB.getEmployee 总是会返回一个 Employee 的实例。不管是否找到雇员，都可以确保所返回的实例具有合适的行为。

当然，在许多情况下仍然想要知道是否 DB.getEmployee 没有找到雇员。在 Employee 中创建一个持有惟一 NullEmployee 实例的 static final 变量，就可以达到这个目的。

程序 17.1 展示了 NullEmployee 的测试用例。在这个测试用例中，“Bob”不存在于数据库中。请注意测试用例期望 isTimeToPay 返回 false。同样请注意，该测试用例也期望 DB.getEmployee 返回的雇员对象就是 Employee.NULL。

程序 17.1　TestEmployee.java（部分代码）

```
public void testNull() throws Exception
{
   Employee e = DB.getEmployee("Bob");
   if (e.isTimeToPay(new Date()))
     fail();
   assertEquals(Employee.NULL, e);
}
```

程序 17.2 展示了 DB 类。请注意，为了测试，getEmployee 方法只是返回 Employee.NULL。

程序 17.2　DB.java

```
public class DB
{
  public static Employee getEmployee(String name)
  {
    return Employee.NULL;
  }
}
```

程序 17.3 展示了 Employee 接口。请注意，该接口有一个名为 NULL 的静态变量持有一个匿名的 Employee 实现体。这个匿名的实现体是无效雇员(null employee)类的惟一的实例。其中，isTimeToPay 方法实现为返回 false，而 pay 方法的实现体为空。

程序 17.3　Employee.java

```
import java.util.Date;

public interface Employee
{
  public boolean isTimeToPay(Date payDate);

  public void pay();

  public static final Employee NULL = new Employee()
  {
    public boolean isTimeToPay(Date payDate)
    {
      return false;
    }

    public void pay()
    {
    }
  };
}
```

使无效雇员类成为一个匿名内嵌类是一种确保该类只有单一实例的方法。实际上并不存在 NullEmpolyee 类本身。其他任何人都无法创建无效雇员类的其他实例。这非常好因为我们希望可以这样表达：

```
if(e == Employee.NULL)
```

如果可以创建无效雇员类的多个实例，那么这种表达方式就是不可靠的。

17.1　结　　论

那些长期使用 C-based 语言的人已经习惯于函数对某种失败返回 null 或者 0。我们认为对这样的函数的返回值是需要检查的。NULL OBJECT 模式改变了这一点。使用该模式，我们可以确保函数总是返回有效的对象，即使在它们失败时也是如此。这些代表失败的对象“什么也不做”。

参考文献

1. Martin, Robert, Dirk Riehle, and Frank Buschmann. *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley, 1998.

第 18 章　薪水支付案例研究：第一次迭代开始

“那些在任何情况下都美丽的事物，其美丽是就其本性而言的，美丽的终结也是就其本性而言的，赞美并不是其本性的一部分。”

——Marcus Aurelius（大约公元 170 年，古罗马哲学家）

18.1　介　　绍

下面的案例研究描述了一个简单的批量处理薪水支付系统开发中的第一次迭代过程。你会发现这个案例研究中的用户素材（user story）是很简单的。例如，其中完全没有提及税金方面的内容。这是初期迭代的特征。其中仅仅提供了客户所需商务价值中的非常小的一部分。

本章中，我们会进行一些快速的分析和设计会话（session），这通常发生在一次正规的迭代开始时。客户已经为本次迭代挑选了素材，现在我们必须知道怎样去实现它们。这样的设计会话简短并且粗略，正像本章一样。在此处看到的 UML 图只不过是在白板上匆忙绘制的草图。在下一章中会进行实际的设计工作，到那时我们会完成单元测试和实现。

18.1.1　规格说明

下面是在和客户交谈关于第一次迭代中的素材时做的一些记录：

- 有些雇员是钟点工。会按照他们雇员记录中每小时报酬字段的值对他们进行支付。他们每天会提交工作时间卡，其中记录了日期以及工作小时数。如果他们每天工作超过 8 小时，那么超过的部分会按照正常报酬的 1.5 倍进行支付。每周五对他们进行支付。
- 有些雇员完全以月薪进行支付。每个月的最后一个工作日对他们进行支付。在他们的雇员记录中有一个月薪字段。
- 同时，对于一些带薪（salaried）雇员，会根据他们的销售情况，支付给他们一定数量的酬

金（commission）。他们会提交销售凭条，其中记录了销售的日期和数量。在他们的雇员记录中有一个酬金报酬字段。每隔一周的周五对他们进行支付。

- 雇员可以选择支付方式。可以选择把支付支票邮寄到他们指定的邮政地址；也可以把支票保存在出纳人员那里随时支取；或者要求将薪水直接存入他们指定的银行账户。
- 一些雇员会加入协会。在他们的雇员记录中有一个每周应付款项字段。这些应付款必须要从他们的薪水中扣除。协会有时也会针对单个协会成员征收服务费用。协会每周会提交这些服务费用，服务费用必须要从相应雇员的下个月的薪水总额中扣除。
- 薪水支付程序每个工作日运行一次，并在当天为相应的雇员进行支付。系统会被告知雇员的支付日期，这样它会计算从雇员上次支付日期到规定的本次支付日期间应支付的数额。

我们可以首先生成数据库模式（database schema）。显然，对于该问题可以使用某种关系数据库，并且从需求中可以清楚地知道表和字段的可能样子。可以容易地设计出一个可用的数据库模式，然后再构建一些查询。不过，在使用这种方法产生的应用程序中，数据库成为了关注的中心。

数据库是实现细节！应该尽可能地推迟考虑数据库。有太多的应用程序之所以和数据库绑定在一起而无法分离，就是因为一开始设计时就把数据库考虑在内了。请记住抽象的定义：本质部分的放大，无关紧要部分的去除。在项目的当前阶段数据库就是无关紧要的；它只不过是一项用来存储和访问数据的技术而已。

18.2 基于用例分析

我们先来考虑一下系统的行为而不是系统的数据。毕竟，别人付给我们报酬正是要我们创建系统的行为。

一种捕获、分析系统行为的方法是创建用例（user case）。按照最初 Jacobson 的描述，用例和 XP 中用户素材的概念非常相似。用例就像是用稍多一点细节详细描述的用户素材。一旦在当前迭代中要实现该用户素材，这种详尽细节就是合适的。

在进行用例分析时，我们关注用户素材和验收测试，以找出系统的用户会执行的操作种类。接着我们会努力弄清楚系统怎样去响应这些操作。

例如，这里是客户为下次迭代选取的用户素材：

（1）增加新雇员。
（2）删除雇员。
（3）登记时间卡。
（4）登记销售凭条。
（5）登记协会服务费。
（6）更改雇员明细（例如，每小时报酬，会费）。
（7）在当日运行薪水支付系统。

让我们来把这些用户素材转换为具有详细细节的用例。我们不需要陷入过多的细节——只要有助于考虑出实现每个素材的代码设计即可。

18.2.1　增加雇员

用例 1　增加新雇员

使用 AddEmp 操作（transcation）可以增加新的雇员。该操作包含有雇员的名字、地址以及分配的雇员号。该操作有 3 种形式：

```
AddEmp <EmpID> "<name>" "<address>" H <hourly-rate>
AddEmp <EmpID> "<name>" "<address>" S <monthly-salary >
AddEmp <EmpID> "<name>" "<address>" C <monthly-salary > <commission-rate>
```

雇员记录是根据对应字段的值来创建的。

异常情况（Alternative）1：描述操作的结构中有错误。

如果描述操作的结构不正确，会在一条错误消息中把它打印出来，并且不进行处理。

用例 1 中隐含着一个抽象。虽然 AddEmp 操作有 3 种形式，但是这 3 种形式中共享<EmpID>、<name>以及<address>字段。我们可以使用 COMMAND 模式创建一个具有 3 个派生类的抽象基类 AddEmployeeTransaction，这 3 个派生类是：AddHourlyEmployee-Transaction、AddSalariedEmployeeTransaction 和 AddCommissionedEmployeeTransaction。（参见图 18.1）

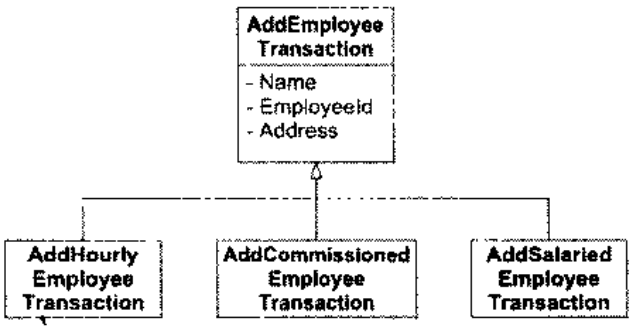


图 18.1　AddEmployeeTransaction 类层次结构

通过把每项工作划分进自己的类中，这个结构很好地遵循了单一职责原则（SRP）。另一种方法是把所有这些工作放入一个模块中。虽然这样做可以减少系统中类的数目，并且因此使系统更简单，但是这同样使所有的操作处理代码都集中在一个地方，造成了一个庞大并且很可能出错的模块。

用例 1 明确地提到了雇员记录，其中暗含着几分数据库的意味。对于数据库的倾向（predisposition）会再次引诱我们去考虑关系数据库表中的记录规划或者字段结构，但是我们应该抵御住这些欲望。用例真正要求我们做的是去创建一个雇员。雇员的对象模型是什么呢？比这更好一点的提问是，这 3 个不同的操作创建了什么呢？在我看来，它们创建了 3 个不同种类的雇员对象，其结构和 3 个不同种类的 AddEmp 操作相仿。图 18.2 展示了一个可能的结构。

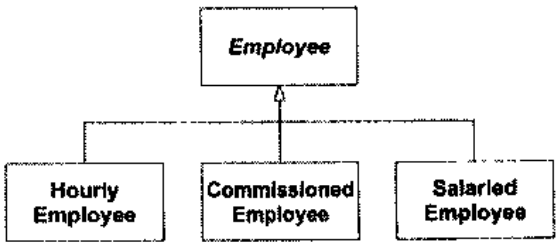


图 18.2　可能的 Employee 类层次结构

18.2.2　删除雇员

用例 2　删除雇员

使用 DelEmp 操作来删除雇员。该操作使用如下形式：

```
DelEmp <EmpID>
```

当执行该操作时，会删除对应的雇员记录。

异常情况 1：无效或者未知的 EmpID。

如果<EmpID>字段不具有正确的结构，或者它没有引用到一条有效的雇员记录，那么会在一条错误消息中把它打印出来，并且不进行其他处理。

此时，该用例没有为我们提供任何设计方面的洞察力，所以我们来看下一个。

18.2.3　登记时间卡

用例 3　登记时间卡

执行 TimeCard 操作时，系统会创建一条时间卡记录，并把该记录和对应的雇员记录关联起来。

```
TimeCard <EmpID> <date> <hours>
```

异常情况 1：所选择的雇员不是钟点雇员。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

异常情况 2：描述操作的结构中有错误。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

该用例指出，一些操作只能应用于某些种类的雇员，这加强了不同种类的雇员应该用不同的类表示的观点。在此用例中，也暗含了一个时间卡和钟点雇员之间的关联。图 18.3 展示了该关联的一个可能的静态模型。

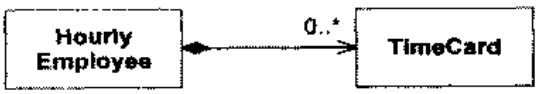


图 18.3　HourlyEmployee 和 TimeCard 间的关联

18.2.4　登记销售凭条

用例 4　登记销售凭条

执行 SalesReceipt 操作时，系统会创建一条新的销售凭条记录，并把该记录和对应的应支付酬金的雇员关联起来。

```
SalesReceipt <EmpID> <date> <amount>
```

异常情况 1：所选择的雇员不是应该支付酬金的。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

异常情况 2：描述操作的结构中有错误。

系统会打印一条适当的错误消息，并且不进行进一步的处理。

这个用例和用例 3 非常类似。它暗含的结构如图 18.4 所示。

18.2.5　登记协会服务费

用例 5　登记协会服务费

执行这个操作时，系统会创建一条服务费用记录，并把该记录和对应的协会成员关联起来。

```
ServiceCharge <memberID> <amount>
```

异常情况 1：描述操作的结构不是良好组织的。

如果该操作不是良好组织（well formed）的，或者<memberID>引用了一个不存在的协会成员，那么会把该操作在一条适当的错误消息中打印出来。

这个用例说明了不能通过雇员 ID 去访问协会成员。协会维护着它自己的针对协会成员的标识编号系统。因此，系统必须要把协会成员和雇员关联起来。有多种不同的方法可以完成这种关联，所以为了避免随意性，我们把这个决策推迟到后面进行。也许，来自系统其他部分的约束会促使我们做出某种选择。

图 18.4　应支付酬金的雇员和销售凭条间的关系

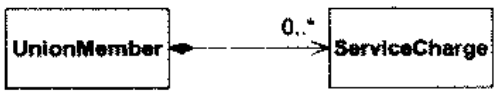


图 18.5　协会成员和服务费

18.2.6　更改雇员明细

用例 6　更改雇员明细

执行这个操作时，系统会更改对应雇员记录的详细信息之一。该操作有几个可能的变体：

`ChgEmp <EmpID> Name <name>`	更改雇员名
`ChgEmp <EmpID> Address <address>`	更改雇员地址
`ChgEmp <EmpID> Hourly <hourlyRate>`	更改每小时报酬
`ChgEmp <EmpID> Salaried <salary>`	更改薪水
`ChgEmp <EmpID> Commissioned <salary> <rate>`	更改酬金
`ChgEmp <EmpID> Hold`	持有支票
`ChgEmp <EmpID> Direct <bank> <account>`	直接存款
`ChgEmp <EmpID> Mail <address>`	邮寄支票
`ChgEmp <EmpID> Member <memberID> Dues <rate>`	使雇员加入协会
`ChgEmp <EmpID> NoMember`	从协会去掉雇员

异常情况：操作错误。

如果描述操作的结构不正确，或者<EmpID>没有引用到真正的雇员，或者<memberID>已经引用了一个协会成员，那么打印一条适当的错误，并且不进行进一步的处理。

该用例非常有启迪性。它表明了雇员信息中可以改变的内容。可以把雇员从钟点雇员改变为带薪雇员的事实意味着图 18.2 中的图示无疑是不恰当的。相反，在薪水计算中使用 STRATEGY 模式或许更加适当。Employee 类中可以持有一个名为 PaymentClassification 的策略类，如图 18.6 所示。这是有好处的因为可以无需改动 Employee 对象的任何部分即可更换 PaymentClassification 对象。把一个钟点雇员更改为带薪雇员时，相应 Employee 对象的 HourlyClassification 对象被 SalariedClassification 对象取代。

有 3 种不同的 PaymentClassification 对象。HourlyClassification 对象中保存着每小时报酬以及一个 TimeCard 对象列表。SalariedClassification 对象中保存着月薪数字。CommissionedClassification 对象中保存着月薪、酬金报酬以及一个 SalesReceipt 对象列表。其中使用了组合（composition）关系，因为我认为当雇员对象被摧毁时，TimeCard 以及 SalesReceipt 对象也应该被摧毁。

同样，支付的方式也应当可以改变。图 18.6 中使用 STRATEGY 模式实现了这个想法，并从 PaymentMethod 类派生出 3 种不同的子类。如果 Empolyee 对象中包含 MailMethod 对象，那么相应的雇员会使他的支付支票邮寄给自己。MailObject 对象中记录了支票要邮寄到的地址。如果 Employee 对象中包含 DirectMethod 对象，那么他的薪水会直接存入 DirectMethod 对象中记录的银行账户中。如果 Employee 对象中包含 HoldMethod 对象，那么他的支付支票会发送到出纳人员那里保存以便随时支取。

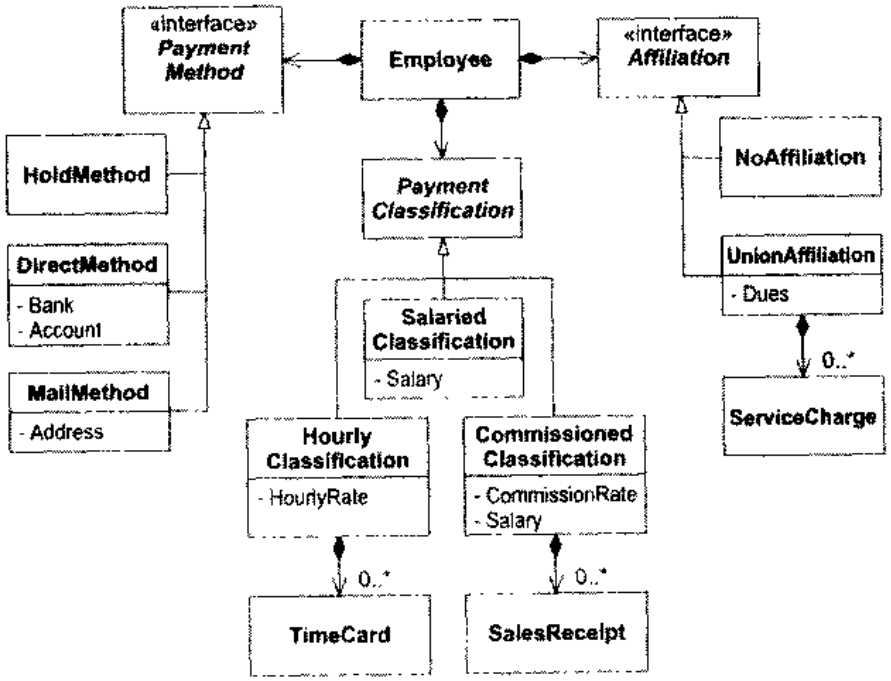


图 18.6　修订后的薪水支付系统类图——核心模型

最后，图 18.6 中对于协会成员关系应用了 NULL OBJECT 模式。每个 Employee 对象包含一个具有两种形式的 Affiliation 对象。如果 Employee 对象包含 NoAffiliation 对象，那么他的薪水除了老板外不会被任何组织调整。然而，如果 Employee 对象包含 UnionAffiliation 对象，那么该雇员就必须要支付 UnionAffiliation 对象中记录的会费和服务费。

这些模式的使用使得该系统很好地符合了开放—封闭原则（OCP）。Employee 类对于支付方式、支付类别以及协会从属关系的变化是封闭的。这样，就可以在不影响 Employee 类的情况下向系统中

增加新的支付方式、支付类别以及协会从属关系。

图 18.6 成为了我们的核心模型（core model）或者构架（architecture）。它是薪水支付系统做的所有工作的中心。在薪水支付应用程序中还有许多其他的类和设计，但是相对于这个基础结构而言，它们都是次要的。当然，这个结构也不是一成不变的：它会和其他所有部分一起演化。

18.2.7　发薪日

用例 7　现在运行薪水支付应用程序

执行 Payday 操作时，系统会找到所有应该在指定日期进行支付的雇员。接着系统确定出他们的应扣款额，并根据他们所选择的支付方式对他们进行支付。

```
Payday <date>
```

虽然该用例的意图很容易理解，但是要确定它对图 18.6 中的静态结构造成的影响就不那么简单了。我们需要回答几个问题。

首先，Employee 对象怎样知道如何计算它的薪水？当然，如果是钟点雇员，系统应当清点他的时间卡并乘以每小时报酬。如果是应支付酬金的雇员，系统应当清点他的销售凭条，乘以酬金报酬系数，并加上基础薪水。但是在哪里完成这种计算呢？PaymentClassification 的派生对象似乎是个理想的地方。这些对象保存着计算薪水所需要的数据，所以它们或许应该拥有确定薪水的方法。图 18.7 展示了一个协作（collaboration）图，描述了可能的工作方式。

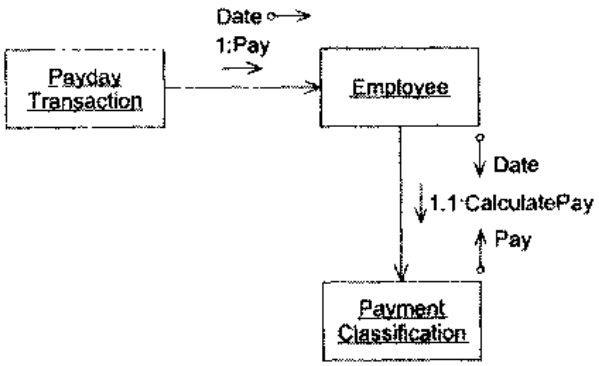


图 18.7　计算雇员薪水

在要求 Employee 对象计算薪水时，该对象把这个请求转交给它的 PaymentClassification 对象。所采用的实际计算方法依赖于 Employee 对象包含的 PaymentClassification 对象的类型。图 18.8 至图 18.10 展示了 3 种可能的情景。

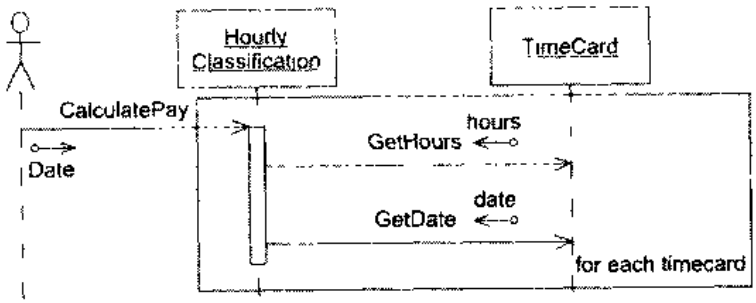


图 18.8　计算钟点雇员的薪水

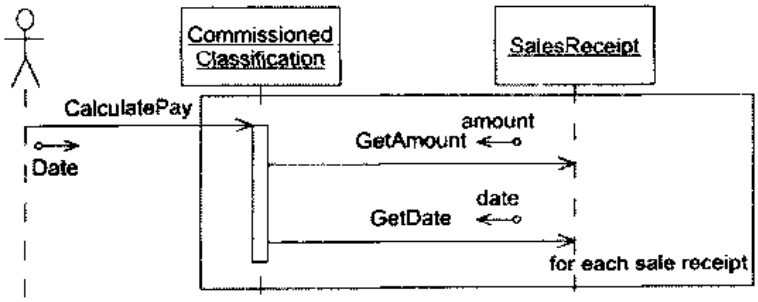


图 18.9　计算应支付酬金雇员的薪水

18.3　反思：我们学到了什么

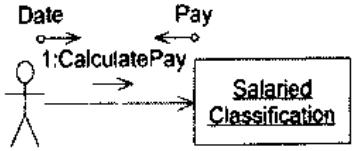
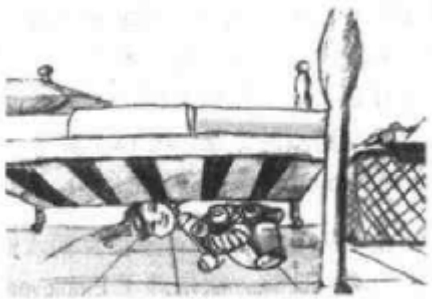


图 18.10　计算带薪雇员的薪水

我们已经知道，简单的用例分析可以提供丰富的信息以及系统设计的洞察力。图 18.6 至 18.10 就是通过思考这些用例得到的，更确切地说，是通过思考行为得到的。

18.4　找出潜在的抽象

为了有效地使用 OCP，必须要搜寻并找出隐藏于应用背后的抽象。通常，应用的需求，甚至用例不会表述，甚至间接提及这些抽象。需求和用例太关注于细节以至于不能表达潜在抽象的一般性。



薪水支付应用中的潜在抽象是什么呢？我们再来看一下需求。我们看到了像这样的陈述，“一些雇员按小时工作”，“一些雇员完全以月薪进行支付”，以及“一些[…]雇员会支付给他们一定数量的酬金”。这暗示了下面的一般性：“所有的雇员都被支付，但是对他们进行支付的策略是不同的”。这里的抽象是“所有的雇员都被支付。”图 18.7 至 18.10 中的 PaymentClassification 模型很好地表达了这个抽象。因此，通过非常简单的用例分析，就已经发现了用户素材中的抽象。

18.4.1　支付薪水时间表抽象

在寻找其他的抽象过程中，我们发现了这样的陈述，“每周五支付”，“每月的最后一个工作日支付”以及“每隔一周的周五支付”。这引导我们得到另一个一般性：“所有的雇员都是按照某种支付薪水时间表进行支付的”。这里的抽象体现为支付薪水时间表的概念(notion)。应该可以询问 Employee 对象某个日期是否是它的支付日期。用例中几乎没有提及此事。需求把雇员的支付薪水时间表和他的支付类别关联起来。更明确地说，每周支付钟点雇员，每月支付带薪雇员，以及每两周支付雇员酬金；然而，这个关联是问题的本质吗？难道不会在某天改变策略，以便于雇员可以选取一种特别的支付薪水时间表，或者以便于隶属于不同科室或者不同部门的雇员可以有不同的支付薪水时间表吗？难道支付薪水时间策略不会独立于支付策略变化吗？当然，这是可能发生的。

如果按照需求的暗示，把支付薪水时间表问题委托给 PaymentClassfication 类，那么该类对于支付薪水时间方面的变化就不是封闭的。当改变支付策略时，必须要测试和支付薪水时间相关的代码。

当改变支付薪水时间表时，同样必须也要测试支付策略。OCP 和 SRP 都被违反了。

支付薪水时间表和支付策略之间的关联会导致一些 bug，比如对于特定支付策略的更改会导致某些雇员具有不正确的支付薪水时间表。像这样的 bug 对于程序员来说是很正常的，但是管理人员以及用户会对此感到恐惧。他们害怕如果对于支付策略的更改会破坏支付薪水时间表，那么任何地方的任何更改就会导致系统的任何其他无关部分出现问题，而事实上也正是如此。他们害怕无法预测更改带来的影响。如果不能预测更改带来的影响，就会丧失信心，并且程序也会给管理人员和用户留下“危险且不稳定”的印象。

尽管存在支付薪水时间表抽象的本质特性，但是用例分析却没有给我们提供有关它的存在的任何直接线索。要发现它就需要仔细地考虑需求，并且要能够洞察出用户社团的误导。过度信赖工具和过程以及低估智力和经验都是灾难的源泉。

图 18.11 和图 18.12 展示了支付薪水时间表抽象的静态和动态模型。正如你看到的，我们再次使用了 STRATEGY 模式。Employee 类包含了抽象的 PaymentSchedule 类。PaymentSchedule 类有 3 种形式，分别对应于 3 种已知的雇员支付薪水时间表。

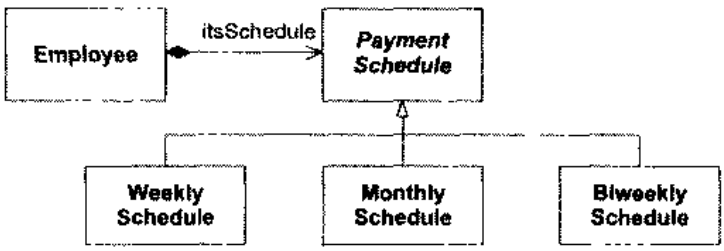


图 18.11　支付薪水时间表抽象的静态模型

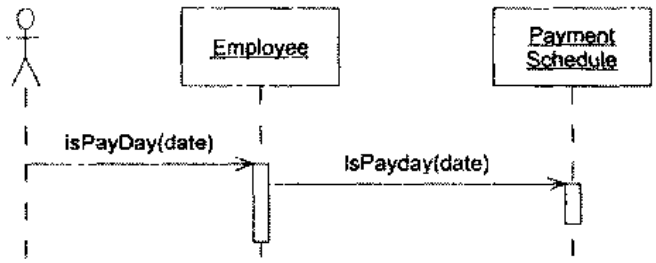


图 18.12　支付薪水时间表抽象的动态模型

18.4.2　支付方式

从需求中可以获取的另一个一般性是“所有的雇员都通过某种方式收到他们的薪水”。这个抽象就是 PaymentMethod 类。非常有趣，该抽象已经在图 18.6 中表达出来了。

18.4.3　从属关系

需求中暗示着雇员可以和一个协会有从属关系；然而协会并非惟一有权从雇员薪水中收取一些费用的组织。雇员可能想自动地为某些慈善团体捐款或者自动地交付一些专业协会的费用。因此，一般性就变成“雇员可以从属于许多组织，并应该自动地从该雇员的薪水中支付这些组织的费用。”

相应的抽象是图 18.6 中展示的 Affiliation 类。不过，图中并没有显示出 Employee 包含多个 Affiliation，并且图中显示出了 NoAffiliation 类。这个设计不是非常吻合我们现在认为需要的抽象。图 18.13 和图 18.14 展示了表示 Affilicaton 抽象的静态和动态模型。

由于使用了 Affiliation 对象列表，所以就无需对那些没有从属关系的雇员使用 NULL OBJECT 模式。现在，如果雇员没有从属关系，只要把他或者她的从属关系对象列表设置为空即可。

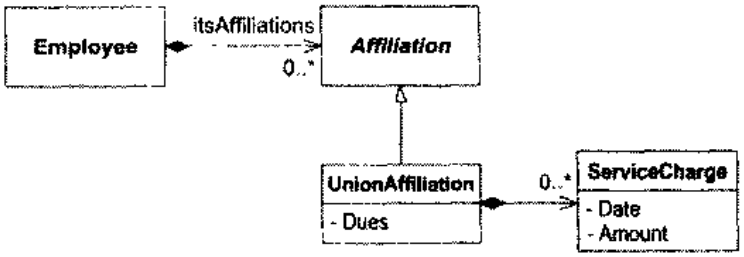


图 18.13　Affiliation 抽象的动态结构

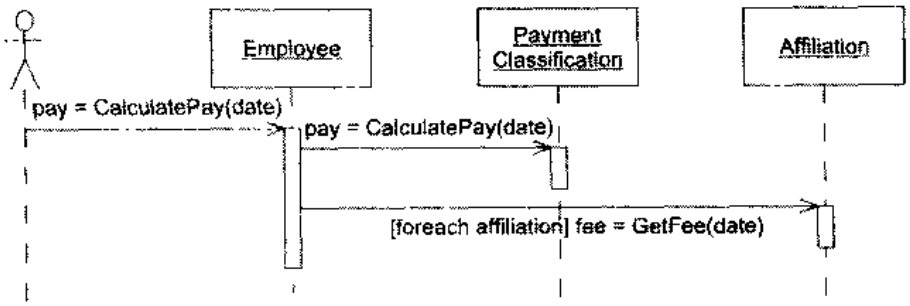


图 18.14　Affiliation 抽象的静态结构

18.5　结　　论

在一次迭代的开始，开发团队通常会聚集在一个白板前，一起思考这次迭代中要实现的用户素材的设计。这种快速设计会话持续的时间一般会小于一个小时。如果产生了一些 UML 图，那么这些图会留在白板上，或者被擦掉。通常不会书面保留这些 UML 图。会话的目的是发起思考活动，并为开发人员提供一个公共的、可以依据其展开工作的智力模型，而不是为了确定设计。

本章就是这样一个快速设计会话的原原本本的再现。

参考文献

1. Jacobson, Ivar. *Object-Oriented Software Engineering, A Use-Case-Driven Approach.* Workingham, England:Addison-Wesley, 1992.

第 19 章　薪水支付案例研究：实现

很早以前，我们就编写了支持和验证前面所讲述的设计的代码。本章中，我会以微小增量的方法来创建该代码，但是只在文中适当的地方才展示它。文中，你只会看到完整形式的代码快照（snapshot），请不要被其误导而认为代码就是以那种形式编写的。事实上，在你所看见的每块代码之间，都有许多的编辑、编译和测试用例，它们都对代码进行了微小的改进。

同样，你也会看到相当多的 UML 图。请把这些 UML 图看作是我快速在白板上勾画的草图，用来向你（我的结对同伴）展示我的想法。UML 为你我之间的交流提供了一个方便的媒介。

图 19.1 中显示出我们用一个名为 Transaction 的抽象基类来代表操作，该类具有一个名为 Execute() 的实例方法（instance method）。这当然是 COMMAND 模式。Transaction 类的实现如程序 19.1 所示。

程序 19.1　Transaction.h

```
#ifndef TRANSACTION_H
#define TRANSACTION_H

class Transaction
{
 public:
  virtual ~Transaction();
  virtual void Execute() = 0;
};

#endif
```

图 19.1　Transaction 接口

19.1　增加雇员

图 19.2 中展示了一个增加雇员操作的可能结构。请注意，正是这些操作把雇员的支付薪水时间表和他们的支付薪水类别关联起来。这样做是合适的，因为这些操作是人工发明物（contrivance）而不是核心模型的一部分。所以，核心模型不会觉察到关联；关联只是我们附加上去的内容，并且可以随时更改。例如，可以容易地增加一种更改雇员支付薪水时间表的操作。

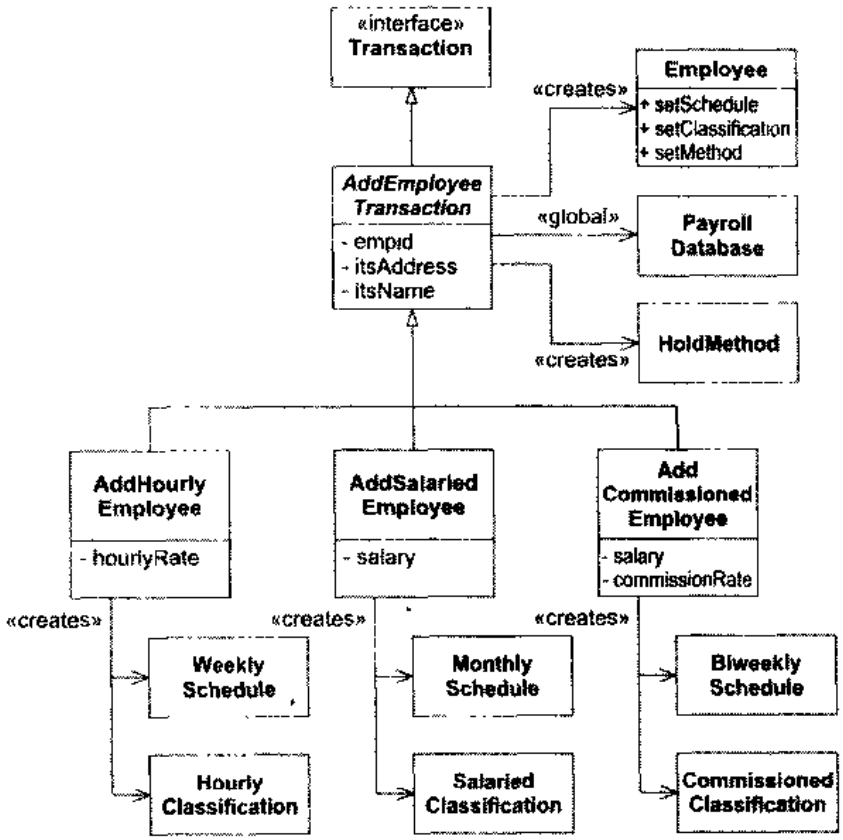


图 19.2 AddEmployeeTransavtion 的静态模型

同样请注意，缺省的支付方式是由出纳人员保存支付支票。如果雇员希望采用另一种支付方式，就必须要使用适当的 ChgEmp 操作进行更改。

如往常一样，我们使用测试优先的方法来编写代码。程序 19.2 中是一个测试用例，用来证明 AddSalariedTransaction 可以正确地工作。随后的代码将会通过该测试用例。

程序 19.2 PayrollTest::TestAddSalariedEmployee

```
void PayrollTest::TestAddSalariedEmployee()
{
  int empId = 1;
  AddSalariedEmployee t(empId, "Bob", "Home", 1000.00);
  t.Execute();

  Employee* e = GpayrollDatabase.GetEmployee(empId);
  assert("Bob" == e->GetName());

  PaymentClassification* pc = e->GetClassification();
  SalariedClassification* sc = dynamic_cast<SalariedClassification*>(pc);
  assert(sc);

  assertEquals(1000.00, sc->GetSalary(), .001);
  PaymentSchedule* ps = e->GetSchedule();
  MonthlySchedule* ms = dynamic_cast<MonthlySchedule*>(ps);
  assert(ms);
  PaymentMethod* pm = e->GetMethod();
```

```
  HoldMethod* hm = dynamic_cast<HoldMethod*>(pm);
  assert(hm);
}
```

19.1.1　薪水支付系统数据库

AddEmployeeTransaction 类使用了一个名为 PayrollDatabase 的类。PayrollDatabase 类在一个以 empID 为键值的 Dictionary 中保存着全部现有的 Employee 对象。同时，它也持有一个把协会的 memberID 映射为 empID 的 Dictionary。图 19.3 中展示了该类的结构。PayrollDatabase 是一个 FACADE 模式（第 15 章）的例子。

程序 19.3 和程序 19.4 中展示了 PayrollDatabase 类的初步实现。该实现是为了帮助通过最初的测试用例。它还没有包含把协会成员 ID 映射为 Employee 实例的 Dictionary 对象。

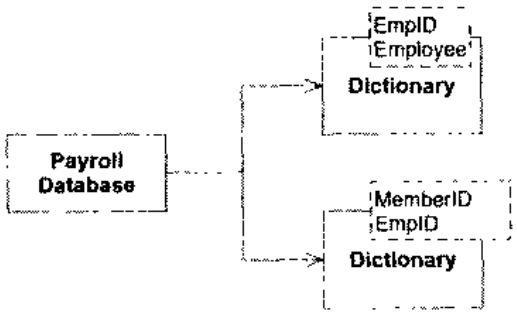


图 19.3　PayrollDatabase 的静态结构

程序 19.3　PayrollDatabase.h

```
#ifndef PAYROLLDATABASE_H
#define PAYROLLDATABASE_H

#include <map>

class Employee;

class PayrollDatabase
{
 public:
  virtual ~PayrollDatabase();
  Employee* GetEmployee(int empId);
  void AddEmployee(int empid, Employee*);
  void clear() {itsEmployees.clear(); }
 private:
  map<int, Employee*> itsEmployees;
};

#endif
```

程序 19.4　PayrollDatabase.cpp

```
#include "PayrollDatabase.h"
#include "Employee.h"

PayrollDatabase GpayrollDatabase;

PayrollDatabase::~PayrollDatabase()
{
}

Employee* PayrollDatabase::GetEmployee(int empid)
{
  return itsEmployees[empid];
}

void PayrollDatabase::AddEmployee(int empid, Employee* e)
{
  itsEmployees[empid] = e;
}
```

一般而言，我认为数据库是实现细节。应该尽可能地推迟有关这些细节的决策。不管这个特定的数据库是使用 RDBMS、平面文件（flat file）或者 OODBMS 实现的，此时都是无关紧要的。现在，我仅仅对创建为应用程序的其他部分提供数据库服务的 API 感兴趣。随后，我会发现有关数据库的合适实现。

推迟有关数据库的细节是一项不常见、但却很值得的实践。我们常常会一直等到对软件及其需要有了更多的知识时，才进行有关数据库的决策。通过等待，我们避免了把过多的基础结构放入数据库中的问题。我们更愿意仅仅实现刚好满足应用程序需要的数据库功能。

19.1.2 使用 TEMPLATE METHOD 模式来增加雇员

图 19.4 展示了增加雇员的动态模型。为了得到正确的 PaymentClassification 对象和 PaymentSchedule 对象，请注意 AddEmployeeTransaction 对象向自己发送了一条消息。AddEmployeeTransation 类的派生类实现了这些消息。这是一个 TEMPLATE METHOD 模式的应用。

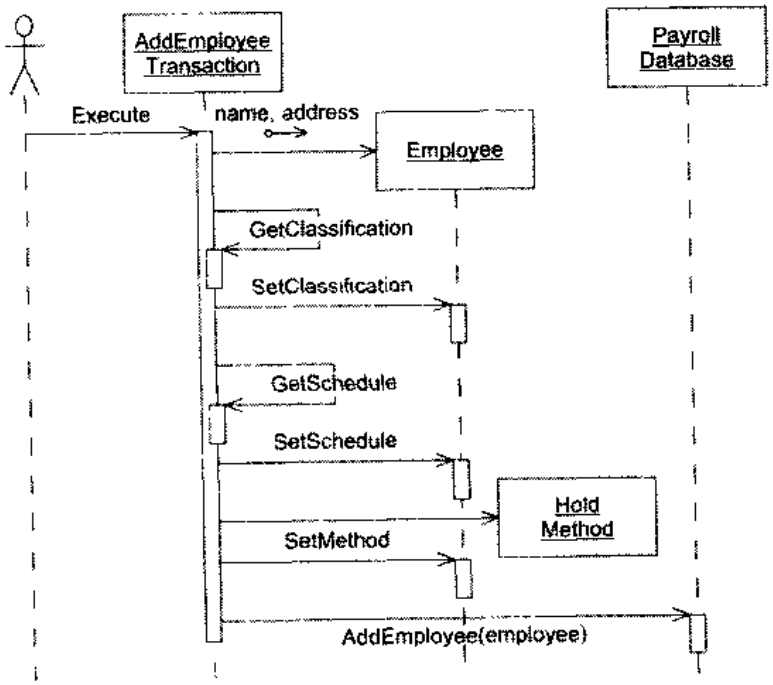


图 19.4 增加雇员的动态模型

程序 19.5 和程序 19.6 中展示了 AddEmployeeTransaction 类中 TEMPLATE METHOD 模式的实现。该类的 Execute()方法中调用了两个会在派生类中实现的纯虚函数。GetSchedule()和 GetClassification()这两个函数返回新创建的 Employee 对象所需要的 PaymentSchedule 和 PaymentClassification 对象。接着，Execute()方法把这些对象绑定到 Employee 对象上并把 Employee 对象存入 PayrollDatabase 中。

程序 19.5 AddEmployeeTransaction.h

```
#ifndef ADDEMPLOYEETRANSACTION_H
#define ADDEMPLOYEETRANSACTION_H

#include "Transaction.h"
#include <string>
```

```
class PaymentClassification;
class PaymentSchedule;

class AddEmployeeTransaction : public Transaction
{
 public:
  virtual ~AddEmployeeTransaction();
  AddEmployeeTransaction(int empid, string name, string address);
  virtual PaymentClassification* GetClassification() const = 0;
  virtual PaymentSchedule* GetSchedule() const = 0;
  virtual void Execute();

 private:
  int itsEmpid;
  string itsName;
  string itsAddress;
};
#endif
```

程序 19.6　AddEmployeeTransaction.cpp

```
#include "AddEmployeeTransaction.h"
#include "HoldMethod.h"
#include "Employee.h"
#include "PayrollDatabase.h"

class PaymentMethod;
class PaymentSchedule;
class PaymentClassification;

extern PayrollDatabase GpayrollDatabase;

AddEmployeeTransaction::~AddEmployeeTransaction()
{
}

AddEmployeeTransaction::AddEmployeeTransaction(int empid,
                string name, string address)
  : itsEmpid(empid)
  , itsName(name)
  , itsAddress(address)
{
}

void AddEmployeeTransaction::Execute()
{
  PaymentClassification* pc = GetClassification();
  PaymentSchedule* ps = GetSchedule();
  PaymentMethod* pm = new HoldMethod();
  Employee* e = new Employee(itsEmpid, itsName, itsAddress);
  e->SetClassification(pc);
  e->SetSchedule(ps);
  e->SetMethod(pm);
  GpayrollDatabase.AddEmployee(itsEmpid, e);
}
```

程序 19.7 和程序 19.8 中展示了 AddSalariedEmployee 类的实现。该类派生自 AddEmployeeTransaction 类并在 GetSchedule()方法和 GetClassification()方法的实现中传回合适的对象给 AddEmployeeTransaction::Execute()。

程序 19.7 AddSalariedEmployee.h

```
#ifndef ADDSALARIEDEMPLOYEE_H
#define ADDSALARIEDEMPLOYEE_H

#include "AddEmployeeTransaction.h"

class AddSalariedEmployee : public AddEmployeeTransaction
{
 public:
  virtual ~AddSalariedEmployee();
  AddSalariedEmployee(int empid, string name, string address, double salary);
  PaymentClassification* GetClassification() const;
  PaymentSchedule* GetSchedule() const;

 private:
  double itsSalary;
};
#endif
```

程序 19.8 AddSalariedEmployee.cpp

```
#include "AddSalariedEmployee.h"
#include "SalariedClassification.h"
#include "MonthlySchedule.h"

AddSalariedEmployee::~AddSalariedEmployee()
{
}

AddSalariedEmployee::AddSalariedEmployee(int empid, string name,
                 string address, double salary)
  : AddEmployeeTransaction(empid, name, address)
    , itsSalary(salary)
{
}

PaymentClassification* AddSalariedEmployee::GetClassification() const
{
  return new SalariedClassification(itsSalary);
}

PaymentSchedule* AddSalariedEmployee::GetSchedule() const
{
  return new MonthlySchedule();
}
```

我把 AddHourlyEmployee 和 AddCommissionedEmployee 的实现留给读者作为练习。请记住首先编写测试用例。

19.2 删除雇员

图 19.5 和图 19.6 中展现了删除雇员操作的静态和动态模型。

程序 19.9 中展示了删除雇员的测试用例。程序 19.10 和程序 19.11 展示了 DeleteEmployeeTransaction 的实现。这是一个非常典型的 COMMAND 模式的实现。构造函数保存了 Execute()方法最后会使用的数据。

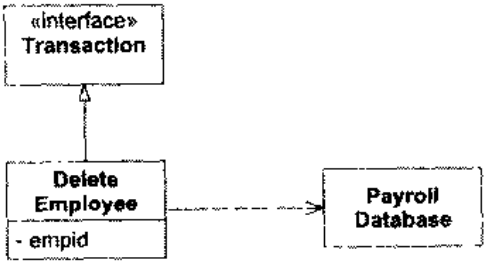


图 19.5　DeleteEmployee 操作的静态模型

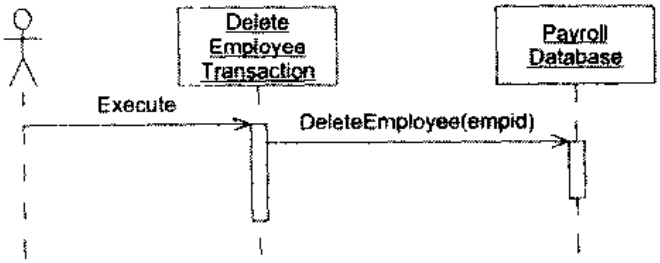


图 19.6　DeleteEmployee 操作的动态模型

程序 19.9　PayrollTest::TestDeleteEmployee()

```
void PayrollTest::TestDeleteEmployee()
{
  cerr << "TestDeleteEmployee" << endl;
  int empId = 3;
  AddCommissionedEmployee t(empId, "Lance", "Home", 2500, 3.2);
  t.Execute();
  {
    Employee* e = GpayrollDatabase.GetEmployee(empId);
    assert(e);
  }
  DeleteEmployeeTransaction dt(empId);
  dt.Execute();
  {
    Employee* e = GpayrollDatabase.GetEmployee(empId);
    assert(e == 0);
  }
}
```

程序 19.10　DeleteEmployeeTransaction.h

```
#ifndef DELETEEMPLOYEETRANSACTION_H
#define DELETEEMPLOYEETRANSACTION_H

#include "Transaction.h"

class DeleteEmployeeTransaction : public Transaction
{
 public:
  virtual ~DeleteEmployeeTransaction();
  DeleteEmployeeTransaction(int empid);
  virtual void Execute();
 private:
  int itsEmpid;
};
#endif
```

程序 19.11　DeleteEmployeeTransaction.cpp

```
#include "DeleteEmployeeTransaction.h"
#include "PayrollDatabase.h"

extern PayrollDatabase GpayrollDatabase;
DeleteEmployeeTransaction::~DeleteEmployeeTransaction()
{
}

DeleteEmployeeTransaction::DeleteEmployeeTransaction(int empid)
```

```
  : itsEmpid(empid)
{
}

void DeleteEmployeeTransaction::Execute()
{
  GpayrollDatabase.DeleteEmployee(itsEmpid);
}
```

19.2.1 全局变量

此时，你已经注意到了 GpayrollDatabase 全局变量。数十年来，教科书和教师一直都有好的理由不鼓励使用全局变量。然而，全局变量并非在本质上就是邪恶和有害的。在本案例的特定情形中，全局变量就是理想选择。PayrollDatabase 类始终只有一个实例，并且该实例需要在一个很广泛的范围中使用。

也许你会认为使用 SINGLETON 模式或者 MONOSTATE 模式可以更好地达到这个目的。这些模式确实可以达到目的。不过，它们是通过自身使用全局变量来达到这个目的的。SINGLETON 或者 MONOSTATE 本来就是全局实体。在本例中，我觉得使用 SINGLETON 模式或者 MONOSTATE 模式具有不必要的复杂性的臭味。简单地把数据库实例保存在一个全局变量中会更容易一些。

19.3 时间卡、销售凭条以及服务费用

图 19.7 中展示了向雇员中登记时间卡操作的静态结构。图 19.8 中展示了该操作的动态模型。基本的思路是，该操作从 PayrollDatabase 中得到 Employee 对象，向 Employee 对象请求它的 PaymentClassification 对象，然后创建一个 TimeCard 对象并把该对象增加到 PaymentClassification 中。

请注意，我们不能把 TimeCard 对象增加到一般的 PayClassification 对象中；我们只能把它们增加到 HourlyClassificaiont 对象中。这意味着必须要把从 Employee 对象中获取的 PaymentClassification 对象向下转型（downcast）为 HourlyClassification 对象。C++中的 dynamic_cast 操作符非常适用于这种情况，如后面的程序 19.15 所示。

程序 19.12 是一个测试用例，该测试验证了可以向钟点雇员中加入时间卡。测试代码只是创建一个钟点雇员对象并把它加入数据库中。接着，它创建一个 TimeCardTransaction 对象并调用其 Execute()。然后，它对雇员进行检查，看看 HourlyClassification 中是否包含了适当的 TimeCard。

程序 19.12 PayrollTest::TestTimeCardTransaction()

```
void PayrollTest::TestTimeCardTransaction()
{
  cerr << "TestTimeCardTransaction" << endl;
  int empId = 2;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
  t.Execute();
  TimeCardTransaction tct(Date(10,31,2001), 8.0, empId);
  tct.Execute();
  Employee* e = GpayrollDatabase.GetEmployee(empId);
  assert(e);
  PaymentClassification* pc = e->GetClassification();
  HourlyClassification* hc = dynamic_cast<HourlyClassification*>(pc);
  assert(hc);
  TimeCard* tc = hc->GetTimeCard(Date(10,31,2001));
```

```
  assert(tc);
  assertEquals(8.0, tc->GetHours());
}
```

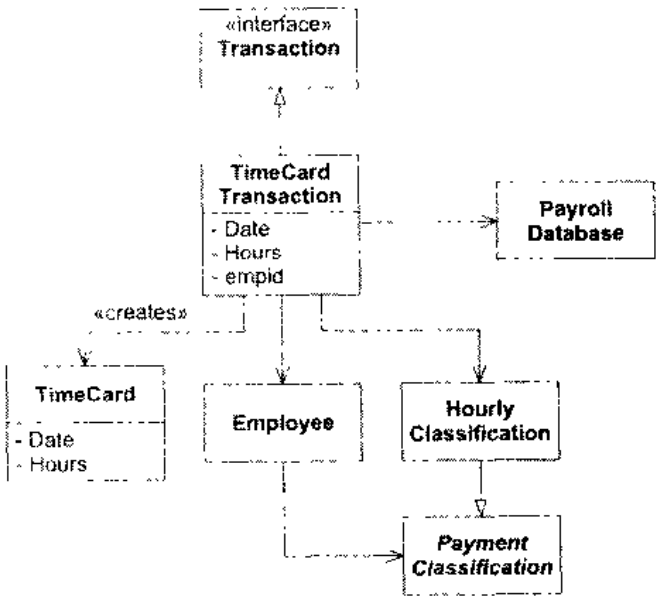


图 19.7　TimeCardTransaction 的静态结构

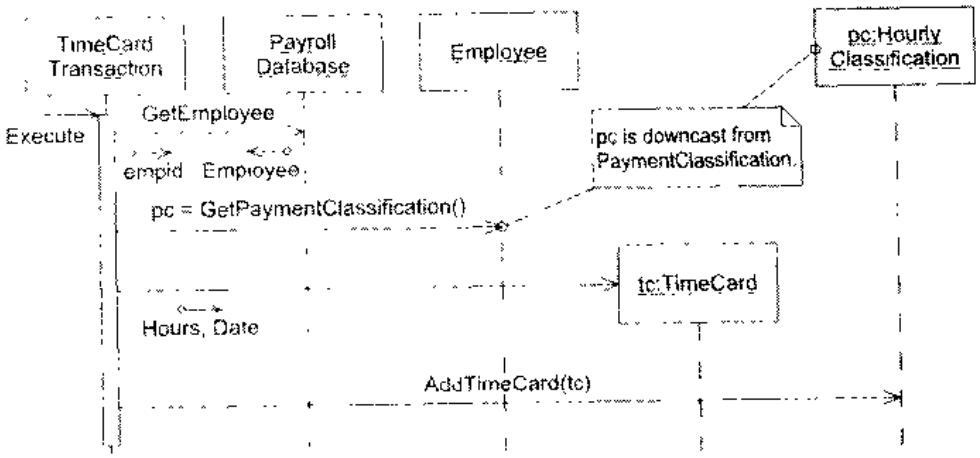


图 19.8　登记 TimeCard 的动态模型

程序 19.13 中展示了 TimeCard 类的实现。目前该类中还没有多少内容。它只是一个数据类。请注意，我用一个长整数来表示日期。这样做是因为还没有一个使于使用的 Date 类。也许很快就会需要一个，但是现在不需要。我想专注于手边的工作，通过当前的测试用例。最后，我会编写一个需要真正 Date 类的测试用例。到那时，我会回来更新 TimeCard 类。

程序 19.13　TimeCard.h

```
#ifndef TIMECARD_H
#define TIMECARD_H

#include "Date.h"

class TimeCard
{
```

```
public:
 virtual ~TimeCard();
 TimeCard(long date, double hours);
 Date GetDate() {return itsDate;}
 double GetHours() {return itsHours;}
private:
 Date itsDate;
 double itsHours;
};
#endif
```

程序 19.14 和程序 19.15 中展示了 TimeCardTransaction 类的实现。请注意，其中使用了简单的字符串异常（string exception）。这不是一个好的长期实践，但是它足以满足开发初期的需要。在对实际需要的异常有一些了解后，我们可以再回来创建有意义的异常类。同样请注意，仅当我们确信异常不会抛出时才创建了 TimeCard 实例，所以异常的抛出不会导致内存泄漏。在抛出异常时，很容易会编写出泄漏内存或者资源的代码，所以要小心。[①]

程序 19.14 TimeCardTransaction.h

```
#ifndef TIMECARDTRANSACTION_H
#define TIMECARDTRANSACTION_H

#include "Transaction.h"

class TimeCardTransaction : public Transaction
{
 public:
  virtual ~TimeCardTransaction();
  TimeCardTransaction(long date, double hours, int empid);

  virtual void Execute();

 private:
  int itsEmpid;
  Date itsDate;
  double itsHours;
};
#endif
```

程序 19.15 TimeCardTransaction.cpp

```
#include "TimeCardTransaction.h"
#include "Employee.h"
#include "PayrollDatabase.h"
#include "HourlyClassification.h"
#include "TimeCard.h"

extern PayrollDatabase GpayrollDatabase;

TimeCardTransaction::~TimeCardTransaction()
{
}

TimeCardTransaction::TimeCardTransaction(long date, double hours, int empid)
```

① 尽快去购买 Herb Sutter 的著作 *Exceptional C++*和 *More Exceptional C++*。这两本书在 C++异常学习方面会减少你大量的苦恼、伤心以及咬牙切齿。

```
  : itsDate(date), itsHours(hours), itsEmpid(empid)
{
}

void TimeCardTransaction::Execute()
{
  Employee* e = GpayrollDatabase.GetEmployee(itsEmpid);
  if (e){
    PaymentClassification* pc = e->GetClassification();
    if (HourlyClassification* hc = dynamic_cast<HourlyClassification*>(pc)){
      hc->AddTimeCard(new TimeCard(itsDate, itsHours));
    } else
      throw("Tried to add timecard to non-hourly employee");
  } else
    throw("No such employee.");
}
```

图 19.9 和图 19.10 展示了向应支付薪水的雇员中登记销售凭条操作的设计，该设计和前面的类似。我把这些类的实现留作练习。

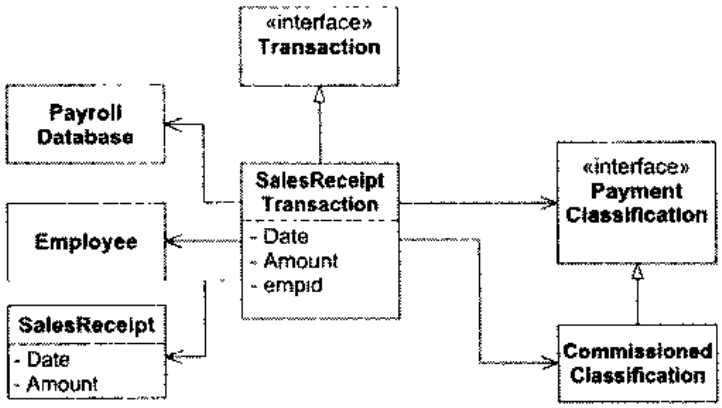


图 19.9　SalesReceiptTransaction 的静态模型

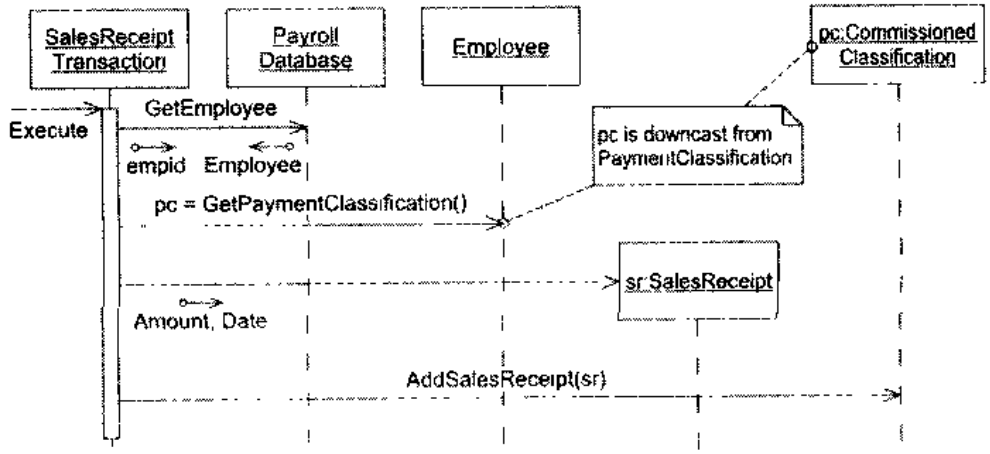


图 19.10　SalesReceiptTransaction 的动态模型

图 19.11 和图 19.12 展示了向协会成员中登记服务费用操作的设计。

这些设计指出了操作模型和已创建的核心模型间的一个失配。核心 Employee 对象可以和许多不同组织间有从属关系，但是操作模型却假定所有从属关系都是协会从属关系。因此，操作模型就无

法识别一个从属关系的明确种类。相反，它只是假定如果向一个雇员中登记了服务费用，那么该雇员就具有一个协会从属关系。

动态模型解决了这个两难问题。它在 Employee 对象包含的一组 Affiliation 对象中搜寻 UnionAffiliation 对象。然后，把 ServiceCharge 对象增加到搜寻到的 UnionAffiliation 对象中。

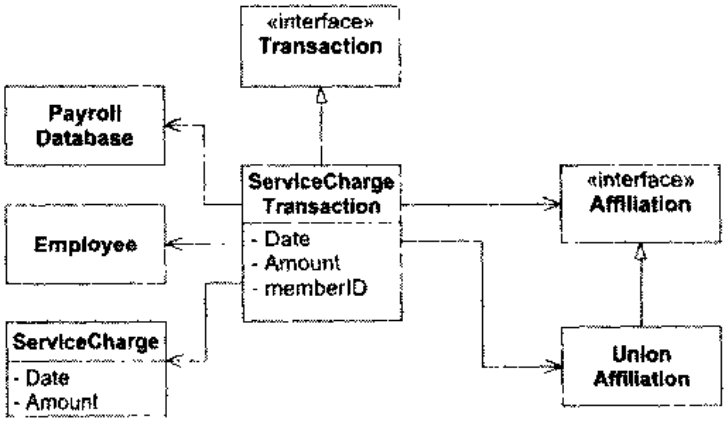


图 19.11 ServiceChargeTransaction 的静态模型

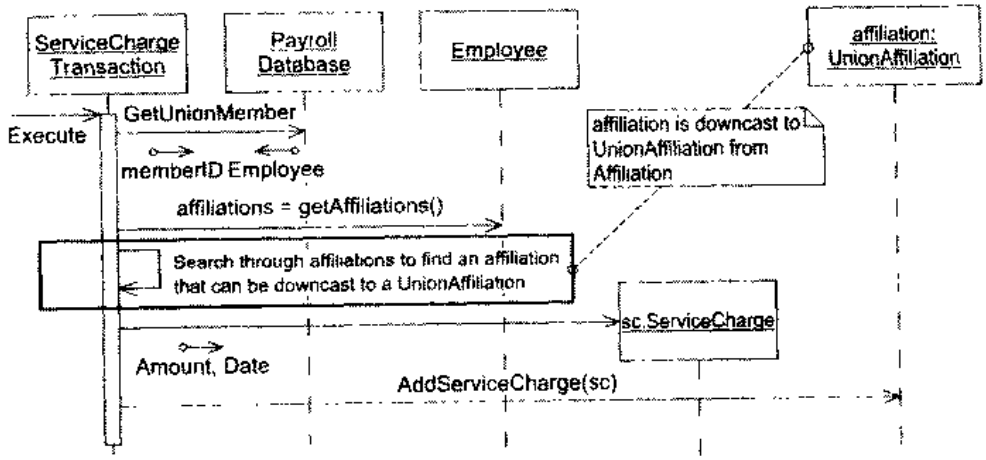


图 19.12 ServiceChargeTransaction 的动态模型

程序 19.16 展示了 ServiceChargeTransaction 的测试用例。它简单地创建一个钟点雇员对象并向其中增加一个 UnionAffiliation 对象。同时，它也确保向 PayrollDatabase 中注册了适当的协会成员 ID。接着，它创建一个 ServiceChargeTransaction 对象并执行之。最后，它证实相应的 ServiceCharge 确实被加入到 Employee 的 UnionAffiliation 中。

程序 19.16 PayrollTest::TestAddServiceCharge()

```
void PayrollTest::TestAddServiceCharge()
{
  cerr << "TestAddServiceCharge" << endl;
  int empId = 2;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
  t.Execute();
  Employee* e = GpayrollDatabase.GetEmployee(empId);
  assert(e);
  UnionAffiliation* af = new UnionAffiliation(memberId, 12.5);
```

```
  e->SetAffiliation(af);
  int memberId = 86; // Maxwell Smart
  GpayrollDatabase.AddUnionMember(memberId, e);
  ServiceChargeTransaction sct(memberId, 20011101, 12.95);
  sct.Execute();
  ServiceCharge* sc = af->GetServiceCharge(20011101);
  assert(sc);
  assertEquals(12.95, sc->GetAmount(), .001);
}
```

19.3.1　代码与 UML

在画图 19.12 中的 UML 图时，我认为用从属关系对象列表取代 NoAffiliation 是一个更好的设计。我认为这样会更加灵活、更加简单一些。毕竟，在任何想增加新的从属关系时，都可以去增加它，并且不必创建 NoAffiliation 类。然而，在编写程序 19.16 中的测试用例时，我认识到调用 Employee 的 SetAffiliation 要比调用 AddAddiliation 更好一些。毕竟，需求并没有要求雇员有多个 Affiliation，所以就没有必要使用 dynamic_cast 在众多可能的种类间进行选择。如果这样做会带来不必要的复杂性。

这个例子说明了画太多的 UML 图而没有验证它的代码是危险的。代码可以告诉你一些 UML 不能告诉你的设计的内容。这里，我在 UML 中就放入了一些不需要的结构。也许，这些结构总有一天会派上用场，但是在这期间必须得维护它们。这些结构带来的好处可能抵不上维护它们的代价。

在本例中，即使维护 dynamic_cast 的代价相对较小，我也不打算使用它。如果不使用 Affiliation 对象列表，实现起来会简单的多。所以，我会继续保持 NULL OBJECT 模式以及 NoAffiliation 类。

程序 19.17 和程序 19.18 中展示了 ServiceChargeTransaction 类的实现。没有了搜寻 UnionAffiliation 对象的循环，该类确实简单了不少。它简单地从数据库中获取 Employee 对象，把该对象的 Affiliation 向下转型为 UnionAffiliation，接着把 ServiceCharge 加入其中。

程序 19.17　ServiceChargeTransaction.h

```
#ifndef SERVICECHARGETRANSACTION_H
#define SERVICECHARGETRANSACTION_H

#include "Transaction.h"

class ServiceChargeTransaction : public Transaction
{
 public:
  virtual ~ServiceChargeTransaction();
  ServiceChargeTransaction(int memberId, long date, double charge);
  virtual void Execute();

 private:
  int itsMemberId;
  Date itsDate;
  double itsCharge;
};
#endif
```

程序 19.18　ServiceChargeTransaction.cpp

```
#include "ServiceChargeTransaction.h"
#include "Employee.h"
#include "ServiceCharge.h"
```

```
#include "PayrollDatabase.h"
#include "UnionAffiliation.h"

extern PayrollDatabase GpayrollDatabase;

ServiceChargeTransaction::~ServiceChargeTransaction()
{
}

ServiceChargeTransaction::
ServiceChargeTransaction(int memberId, long date, double charge)
:itsMemberId(memberId)
, itsDate(date)
, itsCharge(charge)
{
}

void ServiceChargeTransaction::Execute()
{
  Employee* e = GpayrollDatabase.GetUnionMember(itsMemberId);
  Affiliation* af = e->GetAffiliation();
  if (UnionAffiliation* uaf = dynamic_cast<UnionAffiliation*>(af)) {
    uaf->AddServiceCharge(itsDate, itsCharge);
  }
}
```

19.4 更改雇员属性

图 19.13 和图 19.14 展示了更改雇员属性（attribute）操作的静态结构。从用例 6 可以很容易得到这个结构。因为所有的操作都以 EmpID 作为参数，所以可以创建一个最高层次的基类 ChangeEmployeeTransaction。该基类的下面一层是修改单个属性的类，比如：ChangeNameTransaction 和 ChangeAddressTransaction。改变雇员类别的操作有一个共同的行为，它们都会修改 Employee 对象的同一个字段。因此，可以把它们一起放在抽象基类 ChangeClassifiationTransaction 之下。更改支付方式和从属关系的操作与此雷同。从 ChangeMethodTransaction 类和 ChangeAffiliationTransaction 类的结构中可以看到这一点。

图 19.15 展示了所有更改操作的动态模型。其中再次使用了 TEMPLATE METHOD 模式。对于所有的更改操作，都必须要从 PayrollDatabase 中取出对应于 EmpID 的 Employee 对象。因此，ChangeEmployeeTransaction 的 Execute 函数实现了这个行为，然后给自己发送 Change 消息。Change 方法被声明为虚的并在派生类中实现，如图 19.16 和图 19.17 所示。

程序 19.19 展示了 ChangeNameTransaction 的测试用例。该测试用例非常简单。它使用 AddHourlyEmployee 操作创建了一个名为 Bill 的钟点雇员。接着，创建并执行了一个 ChangeNameTransaction 操作，该操作应该把雇员的名字更改为 Bob。最后，从 PayrollDatabase 中取出该雇员实例并验证名字已经被更改。

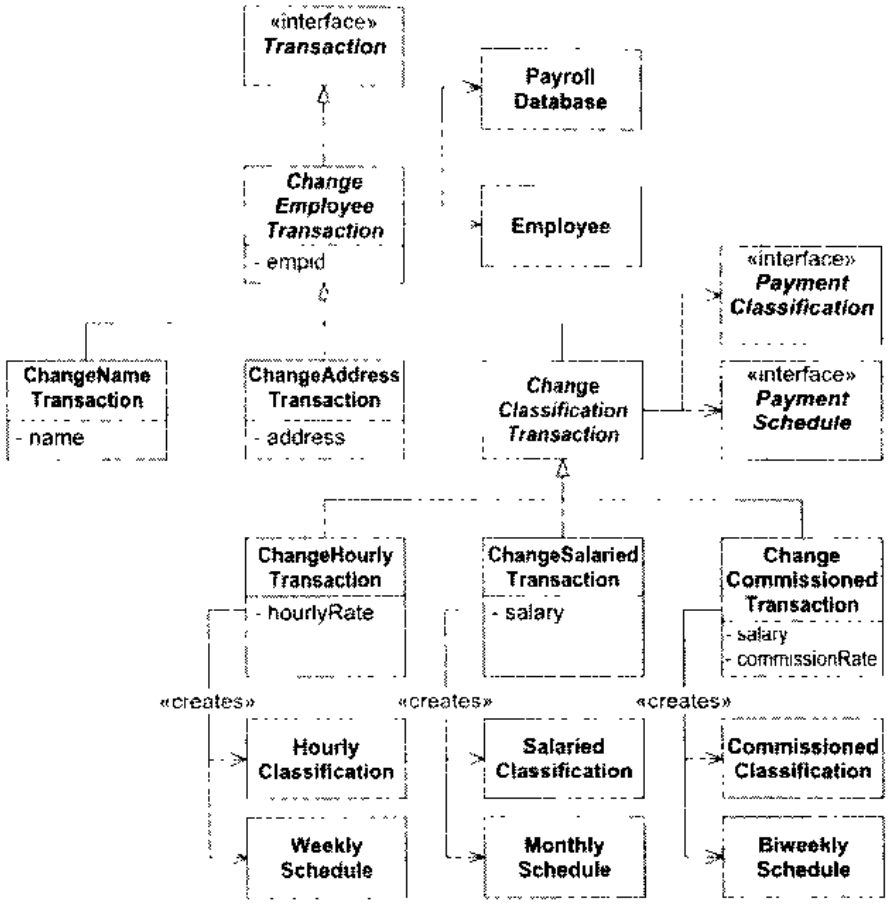


图 19.13　ChangeEmployeeTransaction 的静态模型

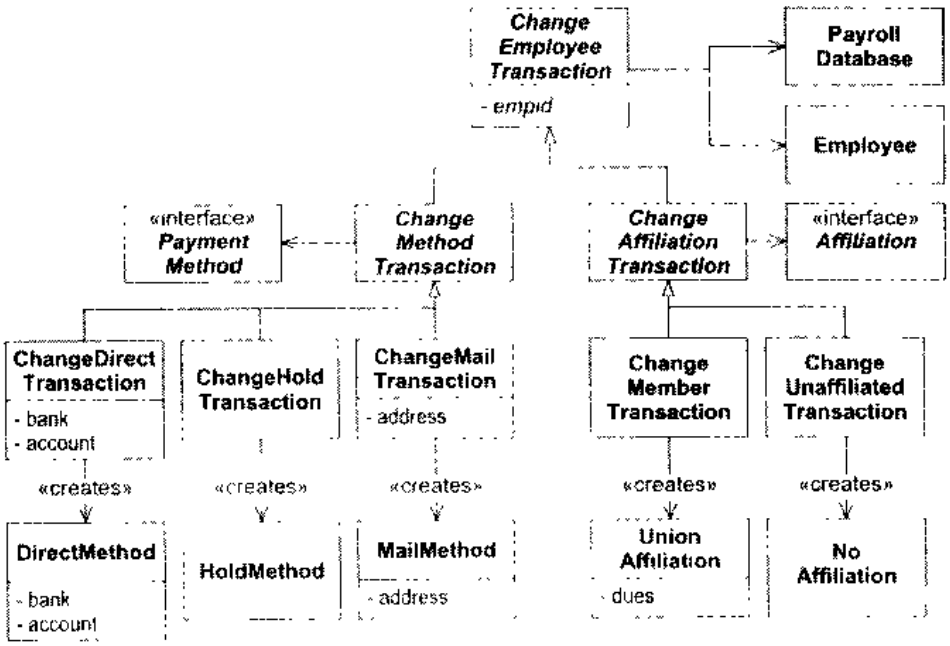


图 19.14　ChangeEmployeeTransaction 的静态模型（续）

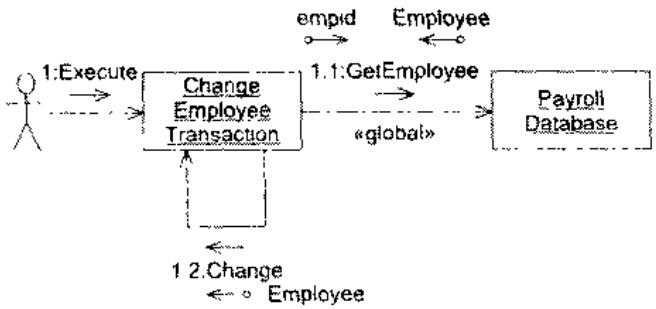


图 19.15　ChangeEmployeeTransaction 的动态模型

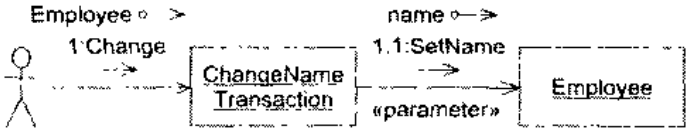


图 19.16　ChangeNameTransaction 的动态模型

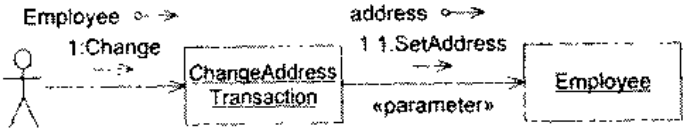


图 19.17　ChangeAddressTransaction 的动态模型

程序 19.19　PayrollTest::TestChangeNameTransaction()

```
void PayrollTest::TestChangeNameTransaction()
{
  cerr << "TestChangeNameTransaction" << endl;
  int empId = 2;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
  t.Execute();
  ChangeNameTransaction cnt(empId, "Bob");
  cnt.Execute();
  Employee* e = GpayrollDatabase.GetEmployee(empId);
  assert(e);
  assert("Bob" == e->GetName());
}
```

程序 19.20 和程序 19.21 中展示了抽象基类 ChangeEmployeeTransaction 的实现。从中可以明显地看到 TEMPLATE METHOD 模式的结构。Execute()方法只是从 PayrollDatabase 中读取适当的 Employee 实例，如果成功，就调用纯虚函数 Change()。

程序 19.20　ChangeEmployeeTransaction.h

```
#ifndef CHANGEEMPLOYEETRANSACTION_H
#define CHANGEEMPLOYEETRANSACTION_H

#include "Transaction.h"
#include "Employee.h"

class ChangeEmployeeTransaction : public Transaction
{
 public:
  ChangeEmployeeTransaction(int empid);
```

```
  virtual ~ChangeEmployeeTransaction();
  virtual void Execute();
  virtual void Change(Employee&) = 0;

private:
  int itsEmpId;
};

#endif
```

程序 19.21　ChangeEmployeeTransaction.cpp

```
#include "ChangeEmployeeTransaction.h"
#include "Employee.h"
#include "PayrollDatabase.h"

extern PayrollDatabase GpayrollDatabase;

ChangeEmployeeTransaction::~ChangeEmployeeTransaction()
{
}

ChangeEmployeeTransaction::ChangeEmployeeTransaction(int empid)
  : itsEmpId(empid)
{
}

void ChangeEmployeeTransaction::Execute()
{
  Employee* e = GpayrollDatabase.GetEmployee(itsEmpId);
  if (e != 0)
    Change(*e);
}
```

程序 19.22 和程序 19.23 中展示了 ChangeNameTransaction 类的实现。从中可以非常容易地看到 TEMPLATE METHOD 模式的另一半。Change()方法改变了作为参数传入的 Employee 对象的名字。ChangeAddressTransaction 的结构与此非常类似，把它的实现留作练习。

程序 19.22　ChangeNameTransaction.h

```
#ifndef CHANGENAMETRANSACTION_H
#define CHANGENAMETRANSACTION_H

#include "ChangeEmployeeTransaction.h"
#include <string>

class ChangeNameTransaction : public ChangeEmployeeTransaction
{
 public:
  virtual ~ChangeNameTransaction();
  ChangeNameTransaction(int empid, string name);
  virtual void Change(Employee&);

 private:
  string itsName;
};

#endif
```

程序 19.23　ChangeNameTransaction.cpp

```
#include "ChangeNameTransaction.h"

ChangeNameTransaction::~ChangeNameTransaction()
{
}

ChangeNameTransaction::ChangeNameTransaction(int empid, string name)
  : ChangeEmployeeTransaction(empid)
  , itsName(name)
{
}

void ChangeNameTransaction::Change(Employee& e)
{
  e.SetName(itsName);
}
```

19.4.1　更改雇员类别

图 19.18 展示了 ChangeClassificationTransaction 的动态行为。其中再次使用了 TEMPLATE METHOD 模式。该操作创建一个新的 PaymentClassification 对象，然后把它传给 Employee 对象。这一点是通过向自己发送 GetClassification 消息完成的。在 ChangeClassificationTransaction 的每个派生类中，都要实现 GetClassification 这个抽象方法，如图 19.19 ~图 19.21 所示。

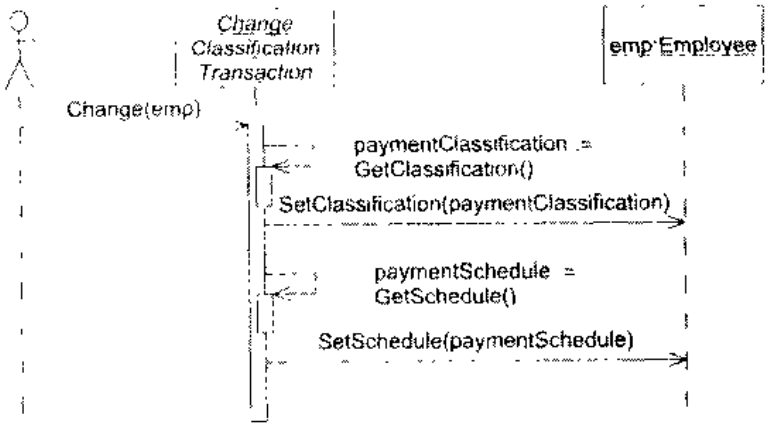


图 19.18　ChangeClassificationTransaction 的动态模型

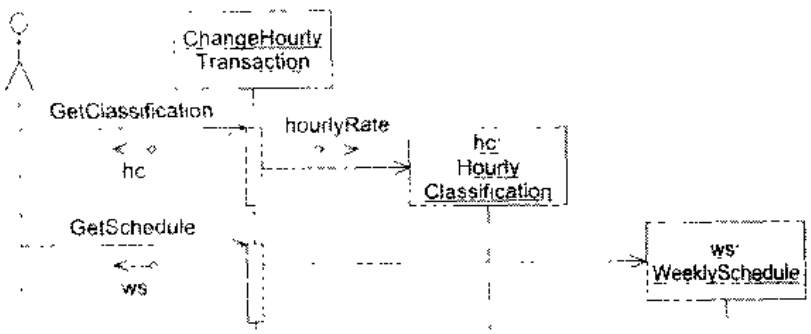


图 19.19　ChangeHourlyTransaction 的动态模型

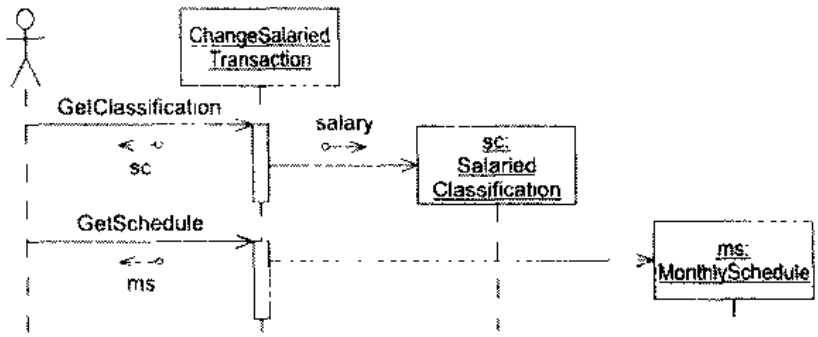


图 19.20　ChangeSalariedTransaction 的动态模型

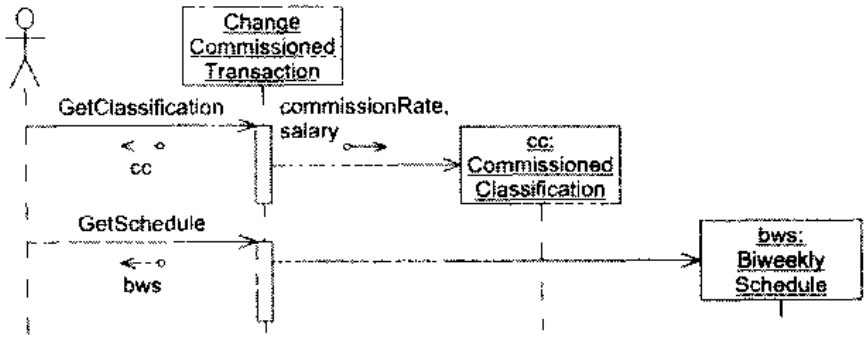


图 19.21　ChangeCommissionedTransaction 的动态模型

程序 19.24 展示了 ChangeHourlyTransaction 的测试用例。测试用例中使用 AddCommissionedEmployee 操作创建了一个应支付酬金的雇员。接着，创建一个 ChangeHourlyTransaction 对象并执行它。然后，取出已经被更改的雇员对象并验证它的 PaymentClassification 成员指向的是具有正确每小时报酬的 HourlyClassificaion 类型的对象，以及它的 PaymentSchedul 成员指向的是 WeeklySchedule 类型的对象。

程序 19.24　PayrollTest::TestChangeHourlyTransaction()

```
void PayrollTest::TestChangeHourlyTransaction()
{
  cerr << "TestChangeHourlyTransaction" << endl;
  int empId = 3;
  AddCommissionedEmployee t(empId, "Lance", "Home", 2500, 3.2);
  t.Execute();
  ChangeHourlyTransaction cht(empId, 27.52);
  cht.Execute();
  Employee* e = GpayrollDatabase.GetEmployee(empId);
  assert(e);
  PaymentClassification* pc = e->GetClassification();
  assert(pc);
  HourlyClassification* hc = dynamic cast<HourlyClassification*>(pc);
  assert(hc);
  assertEquals(27.52, hc->GetRate(), .001);
  PaymentSchedule* ps = e->GetSchedule();
  WeeklySchedule* ws = dynamic_cast<WeeklySchedule*>(ps);
  assert(ws);
}
```

程序 19.25 和程序 19.26 展示了抽象基类 ChangeClassificationTransaction 的实现。其中再一次明显使用了 TEMPLATE METHOD 模式。Change() 方法调用了两个纯虚函数 GetClassification() 和

GetSchedule()。它使用这两个函数的返回值来设置 Employee 的类别以及支付薪水时间表。

程序 19.25 ChangeClassificationTransaction.h

```
#ifndef CHANGECLASSIFICATIONTRANSACTION_H
#define CHANGECLASSIFICATIONTRANSACTION_H

#include "ChangeEmployeeTransaction.h"

class PaymentClassification;
class PaymentSchedule;

class ChangeClassificationTransaction : public ChangeEmployeeTransaction
{
 public:
  virtual ~ChangeClassificationTransaction();
  ChangeClassificationTransaction(int empid);
  virtual void Change(Employee&);
  virtual PaymentClassification* GetClassification() const = 0;
  virtual PaymentSchedule* GetSchedule() const = 0;
};
#endif
```

程序 19.26 ChangeClassificationTransaction.cpp

```
#include "ChangeClassificationTransaction.h"

ChangeClassificationTransaction::~ChangeClassificationTransaction()
{
}

ChangeClassificationTransaction::
ChangeClassificationTransaction(int empid)
  : ChangeEmployeeTransaction(empid)
{
}

void ChangeClassificationTransaction::Change(Employee& e)
{
  e.SetClassification(GetClassification());
  e.SetSchedule(GetSchedule());
}
```

程序 19.27 和程序 19.28 中展示了 ChangeHourlyTransaction 类的实现。该类实现了从 ChangeClassificationTransaction 继承的 GetClassification()和 GetSchedule()方法，从而完善了 TEMPLATE METHOD 模式。它的 GetClassification()方法返回一个新创建的 HourlyClassification 对象。它的 GetSchedule()方法返回一个新创建的 WeeklySchedule 对象。

程序 19.27 ChangeHourlyTransaction.h

```
#ifndef CHANGEHOURLYTRANSACTION_H
#define CHANGEHOURLYTRANSACTION_H

#include "ChangeClassificationTransaction.h"

class ChangeHourlyTransaction : public ChangeClassificationTransaction
{
 public:
```

```
  virtual ~ChangeHourlyTransaction();
  ChangeHourlyTransaction(int empid, double hourlyRate);
  virtual PaymentSchedule* GetSchedule() const;
  virtual PaymentClassification* GetClassification() const;

 private:
  double itsHourlyRate;
};

#endif
```

程序 19.28　ChangeHourlyTransaction.cpp

```
#include "ChangeHourlyTransaction.h"
#include "WeeklySchedule.h"
#include "HourlyClassification.h"

ChangeHourlyTransaction::~ChangeHourlyTransaction()
{
}

ChangeHourlyTransaction::ChangeHourlyTransaction(int empid,
                                                 double hourlyRate)
  : ChangeClassificationTransaction(empid)
    , itsHourlyRate(hourlyRate)
{
}

PaymentSchedule* ChangeHourlyTransaction::GetSchedule() const
{
  return new WeeklySchedule();
}

PaymentClassification* ChangeHourlyTransaction::GetClassification() const
{
  return new HourlyClassification(itsHourlyRate);
}
```

ChangeSalariedTransaction 和 ChangeCommissionedTransaction 的实现也留给读者作为练习。

ChangeMethodTransaction 的实现使用了类似的机制。用抽象方法 GetMethod 来选择适当的 PaymentMethod 派生对象，然后把该派生对象传给 Employee 对象。（参见图 19.22～图 19.25）

这些类的实现简单且不惊奇。同样把它们也留作练习。

图 19.26 展示了 ChangeAffiliationTransaction 的实现。其中再次使用了 TEMPLATE METHOD 模式来选择应该传给 Employee 对象的 Affiliation 派生对象。（参见图 19.27～图 19.29）

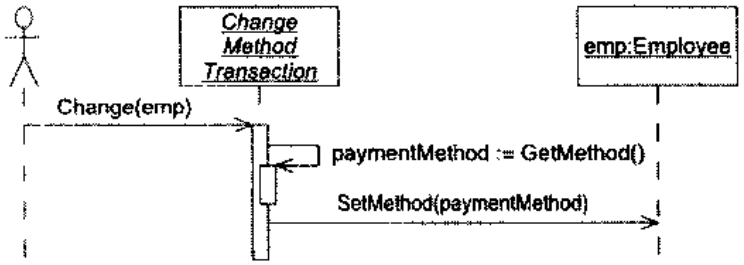


图 19.22　ChangeMethodTransaction 的动态模型

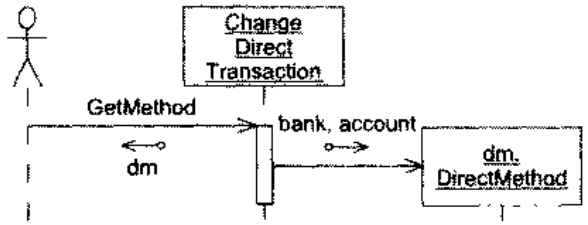


图 19.23　ChangeDirectTransaction 动态模型

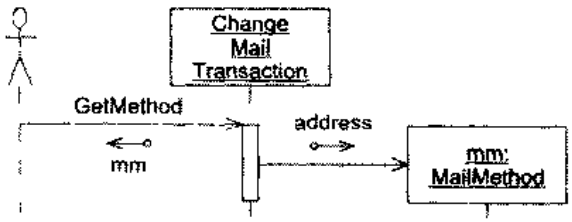


图 19.24　ChangeMailTransaction 动态模型

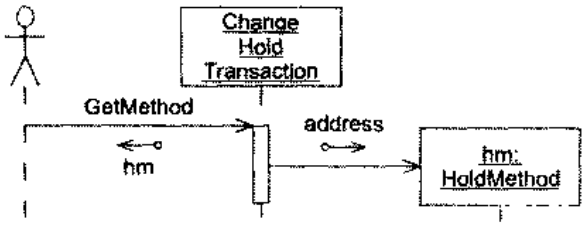


图 19.25　ChangeHoldTransaction 动态模型

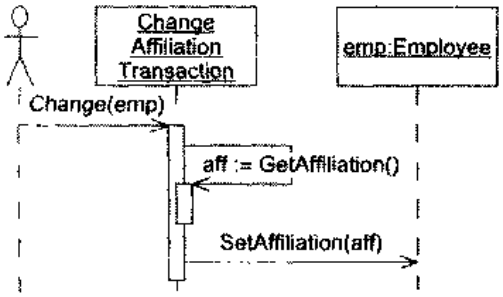


图 19.26　ChangeAffiliationTransaction 动态模型

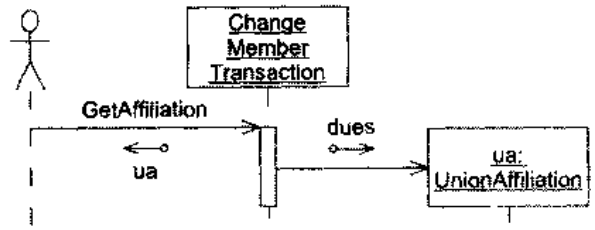


图 19.27　ChangeMemberTransaction 动态模型

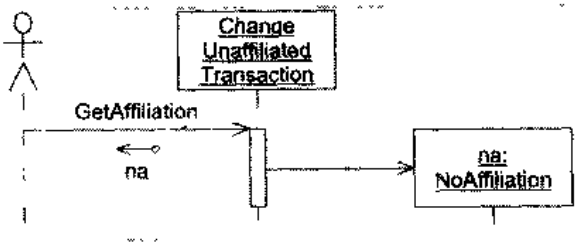


图 19.28　ChangeUnaffiliatedTransaction 动态模型

19.4.2　我当时抽什么烟了

在实现这个设计时，有一件事令我非常惊讶。请仔细看一下更改从属关系操作的动态模型。你能发现问题所在吗？

像往常一样，我通过编写 ChangeMemberTransaction 类的测试用例来实现该类。可以在程序 19.29 中看到这个测试用例。该测试用例非常简单。它创建了一个名为 Bill 的钟点雇员，然后创建并执行一个 ChangMemberTransaction 把 Bill 放入协会中。接着核实 Bill 绑定了一个 UnionAffiliation 对象而且该 UnionAffiliation 对象具有正确的会费。

程序 19.29　PayrollTest::TestChangeMemberTransaction()

```
void PayrollTest::TestChangeMemberTransaction()
{
  cerr << "TestChangeMemberTransaction" << endl;
  int empId = 2;
  int memberId = 7734;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
  t.Execute();
  ChangeMemberTransaction cmt(empId, memberId, 99.42);
  cmt.Execute();
  Employee* e = GpayrollDatabase.GetEmployee(empId);
  assert(e);
```

```
  Affiliation* af = e->GetAffiliation();
  assert(af);
  UnionAffiliation* uf = dynamic_cast<UnionAffiliation*>(af);
  assert(uf);
  assertEquals(99.42, uf->GetDues(), .001);
  Employee* member = GpayrollDatabase.GetUnionMember(memberId);
  assert(member);
  assert(e == member);
}
```

令我惊讶的事就隐藏在该测试用例的最后几行中。这些行用来证实 PayrollDatabase 记录了 Bill 的协会成员关系。现有的 UML 图中根本没有显示出这一点。UML 图仅仅关注 Employee 对象应该和适当的 Affiliation 派生对象绑定在一起。我没有注意到这个缺陷，你呢？

按照 UML 图，我愉快的编写了这些操作，然后等待单元测试失败。一旦失败发生，就可以很明显地发现所忽视的东西。但是问题的解决方案却不那么明显。如何让 ChangeMemberTransaction 记录成员关系，而让 ChangeUnaffiliatedTransaction 清除该成员关系呢？

答案是给 ChangeAffiliationTransaction 增加另外一个纯虚函数 RecordMembership(Employee*)。在 ChangeMemberTransaction 中该函数把 memberId 和 Employee 实例绑定起来。在 ChangeUnaffiliatedTransaction 中该函数清除掉成员关系记录。

程序 19.30 和程序 19.31 展示了抽象基类 ChangeAffiliationTransaction 的实现。很明显，其中再次使用了 TEMPLATE METHOD 模式。

程序 19.30　ChangeAffiliationTransaction.h

```
#ifndef CHANGEAFFILIATIONTRANSACTION_H
#define CHANGEAFFILIATIONTRANSACTION_H

#include "ChangeEmployeeTransaction.h"

class ChangeAffiliationTransaction : public ChangeEmployeeTransaction
{
 public:
  virtual ~ChangeAffiliationTransaction();
  ChangeAffiliationTransaction(int empid);
  virtual Affiliation* GetAffiliation() const = 0;
  virtual void RecordMembership(Employee*) = 0;
  virtual void Change(Employee&);
};

#endif
```

程序 19.31　ChangeAffiliationTransaction.cpp

```
#include "ChangeAffiliationTransaction.h"

ChangeAffiliationTransaction::~ChangeAffiliationTransaction()
{
}

ChangeAffiliationTransaction::ChangeAffiliationTransaction(int empid)
: ChangeEmployeeTransaction(empid)
{
}
```

```
void ChangeAffiliationTransaction::Change(Employee& e)
{
  RecordMembership(&e);
  e.SetAffiliation(GetAffiliation());
}
```

程序 19.32 和程序 19.33 展示了 ChangeMemberTransaction 的实现。该实现简单且乏味。另一方面，程序 19.34 和程序 19.35 中 ChangeUnaffiliatedTransaction 的实现显得稍微有内容一点。RecordMembership 函数必须要确定当前雇员是否为一个协会成员。如果是，那么它就从 UnionAffiliation 中获取 memberId 并清除成员关系记录。

程序 19.32 ChangeMemberTransaction.h

```
#ifndef CHANGEMEMBERTRANSACTION_H
#define CHANGEMEMBERTRANSACTION_H

#include "ChangeAffiliationTransaction.h"

class ChangeMemberTransaction : public ChangeAffiliationTransaction
{
 public:
  virtual ~ChangeMemberTransaction();
  ChangeMemberTransaction(int empid, int memberid, double dues);
  virtual Affiliation* GetAffiliation() const;
  virtual void RecordMembership(Employee*);
 private:
  int itsMemberId;
  double itsDues;
};
#endif
```

程序 19.33 ChangeMemberTransaction.cpp

```
#include "ChangeMemberTransaction.h"
#include "UnionAffiliation.h"
#include "PayrollDatabase.h"

extern PayrollDatabase GpayrollDatabase;

ChangeMemberTransaction::~ChangeMemberTransaction()
{
}

ChangeMemberTransaction::ChangeMemberTransaction(int empid,
                                                 int memberid, double dues)
: ChangeAffiliationTransaction(empid)
, itsMemberId(memberid)
, itsDues(dues)
{
}

Affiliation* ChangeMemberTransaction::GetAffiliation() const
{
  return new UnionAffiliation(itsMemberId, itsDues);
}

void ChangeMemberTransaction::RecordMembership(Employee* e)
{
  GpayrollDatabase.AddUnionMember(itsMemberId, e);
}
```

程序 19.34　ChangeUnffiliatedTransaction.h

```
#ifndef CHANGEUNAFFILIATEDTRANSACTION_H
#define CHANGEUNAFFILIATEDTRANSACTION_H

#include "ChangeAffiliationTransaction.h"

class ChangeUnaffiliatedTransaction : public ChangeAffiliationTransaction
{
 public:
  virtual ~ChangeUnaffiliatedTransaction();
  ChangeUnaffiliatedTransaction(int empId);
  virtual Affiliation* GetAffiliation() const;
  virtual void RecordMembership(Employee*);
};
#endif
```

程序 19.35　ChangeUnffiliatedTransaction.cpp

```
#include "ChangeUnaffiliatedTransaction.h"
#include "NoAffiliation.h"
#include "UnionAffiliation.h"
#include "PayrollDatabase.h"

extern PayrollDatabase GpayrollDatabase;

ChangeUnaffiliatedTransaction::~ChangeUnaffiliatedTransaction()
{
}

ChangeUnaffiliatedTransaction::ChangeUnaffiliatedTransaction(int empId)
: ChangeAffiliationTransaction(empId)
{
}

Affiliation* ChangeUnaffiliatedTransaction::GetAffiliation() const
{
  return new NoAffiliation();
}

void ChangeUnaffiliatedTransaction::RecordMembership(Employee* e)
{
  Affiliation* af = e->GetAffiliation();
  if (UnionAffiliation* uf = dynamic_cast<UnionAffiliation*>(af)) {
    int memberId = uf->GetMemberId();
    GpayrollDatabase.RemoveUnionMember(memberId);
  }
}
```

我对这个设计不是非常满意。ChangeUnaffiliatedTransaction 必须要知道 UnionAffiliation 是一件讨厌的事情。如果在 Affiliation 类中放入抽象方法 RecordMembership 和 EraseMembership 就可以解决这个问题。不过，这样做会迫使 UnionAffiliation 和 NoAffiliation 要知道 PayrollDatabase。而这同样不能令我满意。[1]

尽管如此，但是目前的实现却非常简单并且只是轻微地违反了 OCP。还好，系统中只有极少的模块知道 ChangeUnaffiliatedTransaction，所以它额外的依赖关系不会造成太大的危害。

① 我可以使用 VISITOR 模式（见第 28 章）来解决这个问题，但是这可能是一种过分工程（overengineered）的方法。

19.5　支付雇员薪水

最后，我们来考虑一下这个应用程序的核心操作：指示系统给合适的雇员支付薪水。图 19.29 展示了 PaydayTransaction 类的静态结构。图 19.30～图 19.33 描绘了相关的动态行为。

这几个动态模型显示出了大量的多态行为。CalculatePay 消息使用的算法依赖于 Employee 对象包含的 PaymentClassification 的类型。判断一个日期是否为发薪日的算法依赖于 Employee 对象包含的 PaymentSchedule 的类型。向 Emloyee 对象发送支付信息的算法依赖于 PaymentMethod 对象的类型。这种高度的抽象使得这些算法对于新类型的支付类别、支付薪水时间、从属关系，以及支付方式做到封闭。

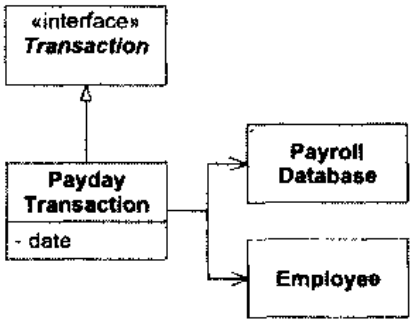


图 19.29　PaydayTransaction 静态模型

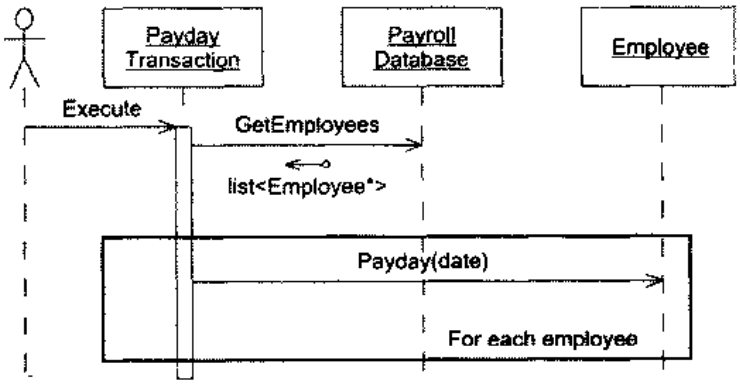


图 19.30　PaydayTransaction 动态模型

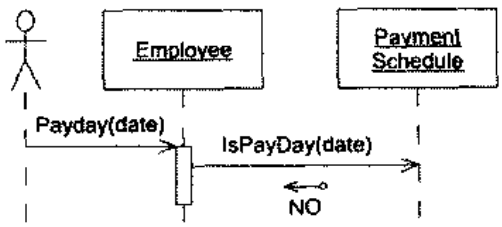


图 19.31　动态模型情景："今天不是发薪日"

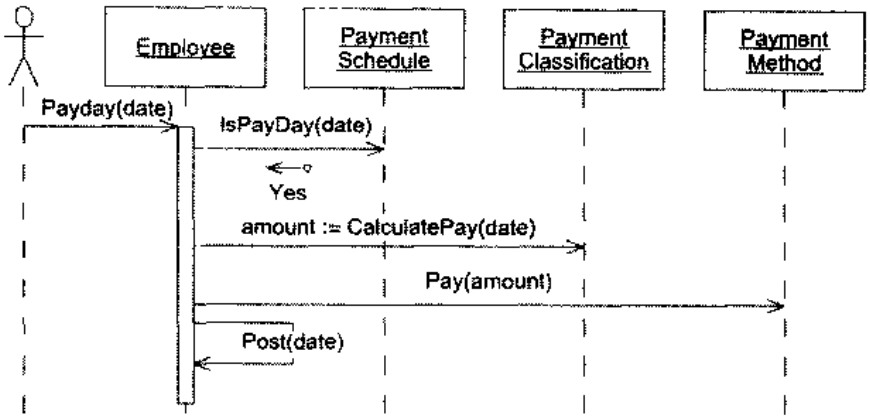


图 19.32　动态模型情景："今天是发薪日"

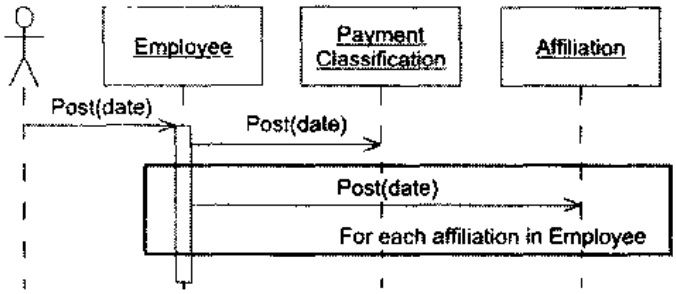


图 19.33　动态模型情景：登记支付信息

图 19.32 和图 19.33 中描绘的算法引入了登记（posting）的概念。在计算出正确的支付数额并发送到 Employee 后，会登记支付信息；也就是说，要更新涉及支付信息的记录。这样，我们就可以把 CalculatePay 方法定义为计算从最近的登记日期至指定日期间的薪水。

19.5.1　我们希望开发人员做商务决策吗

登记这个概念是从哪里来的呢？在用户素材或者用例中肯定没有提到它。我只是碰巧虚构了这么一个概念来解决我所察觉到的问题。我担心会使用同一日期，或者同一支付期内的日期多次调用 Payday 方法，所以我想确保不会出现多次支付雇员薪水的情况。我是主动这样做的，没有询问客户。这似乎就是应该要做的事情。

实际上，我做了一个商务决策。我断定多次运行薪水支付程序会产生不同的结果。关于这个问题，我本应该去询问客户或者项目管理人员，因为他们也许会有完全不同的想法。

在和客户的协商中，我发现登记的想法违反了客户的意图。[①]客户希望在运行薪水支付系统后能够再检查一下支付支票。如果有错，客户希望可以更正支付信息并再次运行薪水支付程序。客户告诉我根本不应该考虑当前支付期之外的时间卡或者销售凭条。

所以，我不得不抛弃有关登记的方案。当时它似乎像是一个好想法，但却不是客户想要的。

① 对，我就是客户。

19.5.2 支付带薪雇员薪水

程序 19.36 中有两个测试用例。它们测试是否正确地支付了一个带薪雇员薪水。第一个测试用例证实在当月的最后一天要对雇员进行支付薪水。第二个测试用例证实如果不是当月的最后一天，就不会对雇员进行支付薪水。

程序 19.36 PayrollTest::TestPaySingleSalariedEmployee & co.

```
void PayrollTest::TestPaySingleSalariedEmployee()
{
  cerr << "TestPaySingleSalariedEmployee" << endl;
  int empId = 1;
  AddSalariedEmployee t(empId, "Bob", "Home", 1000.00);
  t.Execute();
  Date payDate(11,30,2001);
  PaydayTransaction pt(payDate);
  pt.Execute();
  Paycheck* pc = pt.GetPaycheck(empId);
  assert(pc);
  assert(pc->GetPayDate() == payDate);
  assertEquals(1000.00, pc->GetGrossPay(), .001);
  assert("Hold" == pc->GetField("Disposition"));
  assert(Equals(0.0, pc->GetDeductions(), .001);
  assert(Equals(1000.00, pc->GetNetPay(), .001);
}

void PayrollTest::TestPaySingleSalariedEmployeeOnWrongDate()
{
  cerr << "TestPaySingleSalariedEmployeeWrongDate" << endl;
  int empId = 1;
  AddSalariedEmployee t(empId, "Bob", "Home", 1000.00);
  t.Execute();
  Date payDate(11,29,2001);
  PaydayTransaction pt(payDate);
  pt.Execute();
  Paycheck* pc = pt.GetPaycheck(empId);
  assert(pc == 0);
}
```

回想一下程序 19.13，在实现 TimeCard 类时，使用了一个长整数来表示日期。嗯，现在我需要一个真正的 Date 类。如果不能判断出支付日期是否是当月的最后一天，就无法通过这两个测试用例。

记得在大约 10 年前我曾经为我所教授的 C++课程编写过一个 Date 类。所以我就搜寻我的存档并在一台已经闲置的古老的 sparc 工作站上找到了它。[①]我把它移到我的开发环境中并在短时间内使之通过编译。我感到很惊讶，因为原来编写它是在 Linux 中使用的，而现在却是在 Windows 2000 中使用。其中有几个小 bug 要修正，并且还得把自己编写的字符串类替换为 STL 字符串类，但是最终所花费的努力是很小的。

程序 19.37 展示了 PaydayTransaction 的 Execute()函数。该函数遍历了数据库中的所有 Employee 对象，询问每一个 Employee 对象本次操作中指定的日期是否为它的支付日期。如果是，就为该 Employee

① 就是最初的 oma.com。这是一台 sparc 工作站，是一家公司为一个项目购买的，后来项目取消了，我就花费 6000 美元从该公司把它买了过来。在 1994 年，这确实是一笔不错的交易。目前它仍然安静地运行在 Object Mentor 的网络上，这足以证明它的优良品质。

对象创建一个新的支付支票并让 Employee 对象去填写该支付支票。

程序 19.37　PaydayTransaction::Execute()

```
void PaydayTransaction::Execute()
{
  list<int> empIds;
  GpayrollDatabase.GetAllEmployeeIds(empIds);

  list<int>::iterator i = empIds.begin();
  for (; i != empIds.end(); i++) {
    int empId = *i;
    if (Employee* e = GpayrollDatabase.GetEmployee(empId)) {
      if (e->IsPayDate(itsPayDate)) {
     Paycheck* pc = new Paycheck(itsPayDate);
     itsPaychecks[empId] = pc;
     e->Payday(*pc);
      }
    }
  }
}
```

程序 19.38 展示了 MonthlySchedule.cpp 的片段。请注意，仅当日期参数是当月的最后一天时，IsPayDate 才返回 true。这个算法指出了需要 Date 类的原因。如果没有一个好的 Date 类，进行这种简单的日期计算是非常困难的。

程序 19.38　MonthlySchedule.cpp（片段）

```
namespace
{
  bool IsLastDayOfMonth(const Date& date)
  {
    int m1 = date.GetMonth();
    int m2 = (date+1).GetMonth();
    return (m1 != m2);
  }
}

bool MonthlySchedule::IsPayDate(const Date& payDate) const
{
  return IsLastDayOfMonth(payDate);
}
```

程序 19.39 展示了 Employee::PayDay()的实现。该函数是计算并发送所有雇员支付信息的通用算法。请注意，其中大量使用了 STRATEGY 模式。所有的计算细节都被推迟到所包含的策略类：itsClassification、itsAffiliation 以及 itsPaymentMethod 中。

程序 19.39　Employee::PayDay()

```
void Employee::Payday(Paycheck& pc)
{
  double grossPay = itsClassification->CalculatePay(pc);
  double deductions = itsAffiliation->CalculateDeductions(pc);
  double netPay = grossPay - deductions;
  pc.SetGrossPay(grossPay);
  pc.SetDeductions(deductions);
  pc.SetNetPay(netPay);
  itsPaymentMethod->Pay(pc);
}
```

19.5.3 支付钟点雇员薪水

支付钟点雇员薪水的实现可以作为一个很好的示例，用来说明测试优先设计的增量性。我从一些无足轻重的测试用例开始，一步步直到编写出更加复杂的测试用例。我会在下面展示这些测试用例，然后再展示根据这些测试用例产生的产品代码。

程序 19.40 展示了最简单的测试用例。我们向数据库中增加了一个钟点雇员，然后支付他薪水。由于还没有任何的时间卡，所以我们期望支付支票上的值为 0。工具(utility)函数 ValidateHourlyPaycheck 是一次后来重构的结果。起初，该函数的代码完全隐藏在测试函数里面。无需对代码的其余部分做任何更改，就可以通过这个测试用例。

程序 19.40 TestPaySingleHourlyEmployeeNoTimeCards

```
void PayrollTest::TestPaySingleHourlyEmployeeNoTimeCards()
{
  cerr << "TestPaySingleHourlyEmployeeNoTimeCards" << endl;
  int empId = 2;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
  t.Execute();
  Date payDate(11,9,2001); // Friday
  PaydayTransaction pt(payDate);
  pt.Execute();
  ValidatePaycheck(pt, empId, payDate, 0.0);
}

void PayrollTest::ValidatePaycheck(PaydayTransaction& pt, int empid,
                   const Date& payDate, double pay)
{
  Paycheck* pc = pt.GetPaycheck(empid);
  assert(pc);
  assert(pc->GetPayPeriodEndDate() == payDate);
  assertEquals(pay, pc->GetGrossPay(), .001);
  assert("Hold" == pc->GetField("Disposition"));
  assertEquals(0.0, pc->GetDeductions(), .001);
  assertEquals(pay, pc->GetNetPay(), .001);
}
```

程序 19.41 展示了两个测试用例。第一个测试用例验证是否可以支付给具有单一时间卡的雇员薪水。第二个测试用例验证是否可以对超出 8 小时的时间卡进行支付。当然，我不是同时编写这两个测试用例的。相反，我先编写了第一个测试用例并使之能够通过，然后再编写第二个。

程序 19.41 Test...oneTimeCard

```
void PayrollTest::TestPaySingleHourlyEmployeeOneTimeCard()
{
  cerr << "TestPaySingleHourlyEmployeeOneTimeCard" << endl;
  int empId = 2;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
  t.Execute();
  Date payDate(11,9,2001); // Friday

  TimeCardTransaction tc(payDate, 2.0, empId);
  tc.Execute();
  PaydayTransaction pt(payDate);
  pt.Execute();
  ValidatePaycheck(pt, empId, payDate, 30.5);
}
```

```
void PayrollTest::TestPaySingleHourlyEmployeeOvertimeOneTimeCard()
{
  cerr << "TestPaySingleHourlyEmployeeOvertimeOneTimeCard" << endl;
  int empId = 2;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
  t.Execute();
  Date payDate(11,9,2001); // Friday

  TimeCardTransaction tc(payDate, 9.0, empId);
  tc.Execute();
  PaydayTransaction pt(payDate);
  pt.Execute();
  ValidatePaycheck(pt, empId, payDate, (8 + 1.5) * 15.25);
}
```

只要使 HourlyClassification::CalculatePay 遍历雇员的时间卡，累加工作时间，并乘以每小时报酬，就可以通过第一个测试用例。要通过第二个测试用例，我必须得对该函数进行重构，使之可以计算正常工作时间以及加班时间。

程序 19.42 中的测试用例证实了如果 PaydayTransaction 不是使用周五作为参数构造的，系统就不支付钟点雇员。

程序 19.42　TestPaySingleHourlyEmployeeOnWrongDate

```
void PayrollTest::TestPaySingleHourlyEmployeeOnWrongDate()
{
  cerr << "TestPaySingleHourlyEmployeeOnWrongDate" << endl;
  int empId = 2;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
  t.Execute();
  Date payDate(11,8,2001); // Thursday

  TimeCardTransaction tc(payDate, 9.0, empId);
  tc.Execute();
  PaydayTransaction pt(payDate);
  pt.Execute();

  Paycheck* pc = pt.GetPaycheck(empId);
  assert(pc == 0);
}
```

程序 19.43 中的测试用例证实了系统可以为具有多个时间卡的雇员计算薪水。

程序 19.43　TestPaySingleHourlyEmployeeTwoTimeCards

```
void PayrollTest::TestPaySingleHourlyEmployeeTwoTimeCards()
{
  cerr << "TestPaySingleHourlyEmployeeTwoTimeCards" << endl;
  int empId = 2;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
  t.Execute();
  Date payDate(11,9,2001); // Friday

  TimeCardTransaction tc(payDate, 2.0, empId);
  tc.Execute();
  TimeCardTransaction tc2(Date(11,8,2001), 5.0, empId);
  tc2.Execute();
  PaydayTransaction pt(payDate);
  pt.Execute();
```

```
  ValidatePaycheck(pt, empId, payDate, 7*15.25);
}
```

最后，程序 19.44 中的测试用例证实了系统只为当前支付期内的时间卡对雇员进行支付薪水。系统会忽略其他支付期内的时间卡。

程序 19.44 TestPaySingleHourlyemployeeWithTimeCardsSpanningTwoPayPeriods

```
void PayrollTest::
TestPaySingleHourlyEmployeeWithTimeCardsSpanningTwoPayPeriods()
{
  cerr << "TestPaySingleHourlyEmployeeWithTimeCards"
          "SpanningTwoPayPeriods" << endl;
  int empId = 2;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
  t.Execute();
  Date payDate(11,9,2001); // Friday
  Date dateInPreviousPayPeriod(11,2,2001);

  TimeCardTransaction tc(payDate, 2.0, empId);
  tc.Execute();
  TimeCardTransaction tc2(dateInPreviousPayPeriod, 5.0, empId);
  tc2.Execute();
  PaydayTransaction pt(payDate);
  pt.Execute();
  ValidatePaycheck(pt, empId, payDate, 2*15.25);
}
```

通过所有这些测试用例的代码是以增量方式编写的，每次通过一个测试用例。下面的代码结构是在逐个通过测试用例的过程中演化而来的。程序 19.45 展示了 HourlyClassification.cpp 的适当代码片段。我们只是简单地遍历每个时间卡，检查其是否在当前支付期内。若是，就计算它所代表的薪水。

程序 19.45 HourlyClassification.cpp（片段）

```
double HourlyClassification::CalculatePay(Paycheck& pc) const
{
  double totalPay = 0;
  Date payPeriod = pc.GetPayDate();
  map<Date, TimeCard*>::const_iterator i;
  for (i=itsTimeCards.begin(); i != itsTimeCards.end(); i++) {
    TimeCard * tc = (*i).second;
    if (IsInPayPeriod(tc, payPeriod))
      totalPay += CalculatePayForTimeCard(tc);
  }
  return totalPay;
}

bool HourlyClassification::IsInPayPeriod(TimeCard* tc,
                                         const Date& PayPeriod) const
{
    Date payPeriodEndDate = payPeriod;
    Date payPeriodStartDate = payPeriod - 5;
    Date timeCardDate = tc->GetDate();
    Return(timeCardDate >= payPeriodStartDate) &&
          (timeCardDate <= payPeriodEndDate);
}

double HourlyClassification::
CalculatePayForTimeCard(TimeCard* tc) const
{
    double hours = tc->GetHours();
```

```
    double overtime = max(0.0, hours - 8.0);
    double straightTime = hours - overtime;
    return straightTime * itsRate + overtime * itsRate * 1.5;
}
```

程序 19.46 说明了 WeeklySchedule 的支付薪水时间是周五。

程序 19.46　WeeklySchedule::IsPayDate

```
bool WeeklySchedule::IsPayDate(const Date& theDate) const
{
  return theDate.GetDayOfWeek() == Date::friday;
}
```

我把计算应支付酬金的雇员薪水的实现留给读者完成，应该不会有太大的困难。作为一个更有趣一些的练习，可以考虑一下允许周末登记时间卡，并正确计算加班时间的情况。

19.5.4　支付期：一个设计问题

现在，我们来实现计算会费和服务费的功能。我设想了一个测试用例，该测试用例增加一个带薪雇员，把它转变成一个协会成员，然后支付该雇员薪水，并确保从雇员的薪水中扣除了会费。程序 19.47 是该测试用例的代码。

程序 19.47　Payrolltest::TestSalariedUnionMemberDues

```
void PayrollTest::TestSalariedUnionMemberDues()
{
  cerr << "TestSalariedUnionMemberDues" << endl;
  int empId = 1;
  AddSalariedEmployee t(empId, "Bob", "Home", 1000.00);
  t.Execute();
  int memberId = 7734;
  ChangeMemberTransaction cmt(empId, memberId, 9.42);
  cmt.Execute();
  Date payDate(11,30,2001);
  PaydayTransaction pt(payDate);
  pt.Execute();
  ValidatePaycheck(pt, empId, payDate, 1000.00 - ??? );
}
```

请注意测试用例最后一行中的???。该用什么来替代???呢？用户素材中说会费每周提交一次，而带薪雇员却是每月支付一次。每月包含几周呢？我只要把会费乘以 4 就可以了吗？这样做不是非常准确。我要问问客户他希望如何去做。[①]

客户告诉我会费每周五累加一次。所以我只要计算支付期内包含的周五的数目并乘以每周的会费即可。2001 年 11 月（测试用例中设定的月份）有 5 个周五。所以可以对测试用例做适当地修改。

计算支付期内包含的周五的数目意味着需要知道支付期的起始和终止日期。我已经在程序 19.45 中的 IsInPayPeriod 中进行过这样的计算（你也许已经为 CommissionedClassification 写了一个类似的实现）。HourlyClassification 对象的 CalculatePay 函数使用这个函数来确保仅仅统计支付期内的时间卡。现在看来 UnionAffiliation 对象必须也要调用这个函数。

但是请等一下！在 HourlyClassification 类中这个函数做了什么？我们已经确定支付薪水时间表和支付类别之间的关联是非本质的。用来确定支付期的函数应该属于 PaymentSchedule 类，而不应该属

① 所以 Bob 再次自言自语。到 www.google.com/groups 上查询一下“Schizophrenic Robert Martin”。

于 PaymentClassification 类！

有趣的是，UML 图并没有帮助我们捕捉到这个问题。只是在我开始考虑 UnionAffiliation 的测试用例时这个问题才显现出来。这再一次说明了代码反馈对于任何设计来说是多么的必要。图示是有用的，但是在没有代码反馈的情况下去依赖它们就是冒险行为。

所以，怎样才能从 PaymentSchedule 层次结构中获取支付期间并在 PayClassification 以及 Affiliation 层次结构中使用它呢？这些层次结构之间互不知晓。可以把用于计算支付期间的日期放在 Paycheck 对象中。目前，Paycheck 中仅包含了支付期间的终止日期，还需要能够从 Paycheck 中获取起始日期。

程序 19.48 中展示了对 PaydayTransaction::Execute()所做的更改。请注意，在创建 Paycheck 时，同时给它传入了支付期间的起始和终止日期。如果提前跳到程序 19.55，你会看到计算这两个日期的是 PaymentSchedule。对 Paycheck 的更改是显而易见的。

程序 19.48 PaydayTransaction::Execute()

```
void PaydayTransaction::Execute()
{
  list<int> empIds;
  GpayrollDatabase.GetAllEmployeeIds(empIds);

  list<int>::iterator i = empIds.begin();
  for (; i != empIds.end(); i++) {
    int empId = *i;
    if (Employee* e = GpayrollDatabase.GetEmployee(empId)) {
      if (e->IsPayDate(itsPayDate)) {
     Paycheck* pc = new Paycheck(e->GetPayPeriodStartDate(itsPayDate),
                      itsPayDate);
     itsPaychecks[empId] = pc;
     e->Payday(*pc);
      }
    }
  }
}
```

HourlyClassification 和 CommissionedClassification 中用于确定 TimeCards 和 SalesReceipts 是否在支付期间的两个函数已经被合入基类 PaymentClassification 中（参见程序 19.49）。

程序 19.49 PaymentClassification::IsInPayPeriod(...)

```
bool PaymentClassfication::IsInPayPeriod(const Date& theDate,
                                         const Paycheck& pc) const
{
    Date payPeriodEndDate = pc.GetPayPeriodEndDate();
    Date payPeriodStartDate = pc.GetPayPeriodStartDate();
    Return(theDate >= payPeriodStartDate) &&
          (theDate <= payPeriodEndDate);
}
```

现在，我们可以在 UnionAffiliation::CalculateDeductions 中计算雇员的会费了。程序 19.50 展示了会费的计算方法。它从 paycheck 中获取支付期间的两个界限日期，然后把这两个日期传给一个计算它们之间周五数目的工具函数。接着，把计算结果乘以每周的会费就得到支付期间内的会费总额。

程序 19.50 UnionAffiliation::CalculateDeductions()

```
namespace
{
  int NumberOfFridaysInPayPeriod(const Date& payPeriodStart,
                 const Date& payPeriodEnd)
```

```
  {
    int fridays = 0;
    for (Date day = payPeriodStart; day <= payPeriodEnd; day++)
    {
      if (day.GetDayOfWeek() == Date::friday)
        fridays++;
    }
    return fridays;
  }
}

double UnionAffiliation::CalculateDeductions(Paycheck& pc) const
{
  double totalDues = 0;

  int fridays = NumberOfFridaysInPayPeriod(pc.GetPayPeriodStartDate(),
                        pc.GetPayPeriodEndDate());
  totalDues = itsDues * fridays;
  return totalDues;
}
```

最后两个测试用例和协会的服务费用有关。程序 19.51 中是第一个测试用例。它要证实服务费用被正确地扣除。

程序 19.51　PayrollTest::TestHourlyUnionMemberServiceCharge

```
void PayrollTest::TestHourlyUnionMemberServiceCharge()
{
  cerr << "TestHourlyUnionMemberServiceCharge" << endl;
  int empId = 1;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.24);
  t.Execute();
  int memberId = 7734;
  ChangeMemberTransaction cmt(empId, memberId, 9.42);
  cmt.Execute();
  Date payDate(11,9,2001);
  ServiceChargeTransaction sct(memberId, payDate, 19.42);
  sct.Execute();
  TimeCardTransaction tct(payDate, 8.0, empId);
  tct.Execute();
  PaydayTransaction pt(payDate);
  pt.Execute();
  Paycheck* pc = pt.GetPaycheck(empId);
  assert(pc);
  assert(pc->GetPayPeriodEndDate() == payDate);
  assertEquals(8*15.24, pc->GetGrossPay(), .001);
  assert("Hold" == pc->GetField("Disposition"));
  assertEquals(9.42 + 19.42, pc->GetDeductions(), .001);
  assertEquals((8*15.24)-(9.42 + 19.42), pc->GetNetPay(), .001);
}
```

第二个测试用例给我提出了一个问题。在程序 19.52 中可以看到这一点。该测试用例要证实当前支付期间外的服务费用没有被扣除。

程序 19.52　PayrollTest::TestServiceChargesSpanningMultiplePayPeriods

```
void PayrollTest::TestServiceChargesSpanningMultiplePayPeriods()
{
  cerr << "TestServiceChargesSpanningMultiplePayPeriods" << endl;
  int empId = 1;
  AddHourlyEmployee t(empId, "Bill", "Home", 15.24);
```

```
  t.Execute();
  int memberId = 7734;
  ChangeMemberTransaction cmt(empId, memberId, 9.42);
  cmt.Execute();
  Date earlyDate(11,2,2001); // previous Friday
  Date payDate(11,9,2001);
  Date lateDate(11,16,2001); // next Friday
  ServiceChargeTransaction sct(memberId, payDate, 19.42);
  sct.Execute();
  ServiceChargeTransaction sctEarly(memberId, earlyDate, 100.00);
  sctEarly.Execute();
  ServiceChargeTransaction sctLate(memberId, lateDate, 200.00);
  sctLate.Execute();
  TimeCardTransaction tct(payDate, 8.0, empId);
  tct.Execute();
  PaydayTransaction pt(payDate);
  pt.Execute();
  Paycheck* pc = pt.GetPaycheck(empId);
  assert(pc);
  assert(pc->GetPayPeriodEndDate() == payDate);
  assertEquals(8*15.24, pc->GetGrossPay(), .001);
  assert("Hold" == pc->GetField("Disposition"));
  assertEquals(9.42 + 19.42, pc->GetDeductions(), .001);
  assertEquals((8*15.24)-(9.42 + 19.42), pc->GetNetPay(), .001);
}
```

为了实现这一点，我想让 UnionAffiliation::CalculateDeductions 调用 IsInPayPeriod。糟糕的是，我们刚把 IsInPayPeriod 放到了 PaymentClassification 类中（参见程序 19.49）。当只有 PaymentClassification 类的派生类调用 IsInPayPeriod 时，把 IsInPayPeriod 放在 PaymentClassification 类是合适的。但是现在其他类也需要它。所以我把该函数移到 Date 类中。毕竟，该函数只是确定一个给定日期是否在其他两个指定日期之间（参见程序 19.53）。

程序 19.53 Date::IsBetween

```
static bool IsBetween( const Date& theDate, const Date& startDate,
                       const Date& endDate)
{
   return (theDate >= startDate) && (theDate <= endDate);
}
```

现在，我们可以最后完成 UnionAffiliationL::CalculateDeductions 函数。我把它留给读者作为练习。

程序 19.54 和程序 19.55 展示了 Employee 类的实现。

程序 19.54 Employee.h

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>

class PaymentSchedule;
class PaymentClassification;
class PaymentMethod;
class Affiliation;
class Paycheck;
class Date;

class Employee
{
 public:
```

```
 virtual ~Employee();
 Employee(int empid, string name, string address);
 void SetName(string name);
 void SetAddress(string address);
 void SetClassification(PaymentClassification*);
 void SetMethod(PaymentMethod*);
 void SetSchedule(PaymentSchedule*);
 void SetAffiliation(Affiliation*);

 int GetEmpid() const {return itsEmpid;}
 string GetName() const {return itsName;}
 string GetAddress() const {return itsAddress;}
 PaymentMethod* GetMethod() {return itsPaymentMethod;}
 PaymentClassification* GetClassification() {return itsClassification;}
 PaymentSchedule* GetSchedule() {return itsSchedule;}
 Affiliation* GetAffiliation() {return itsAffiliation;}

 void Payday(Paycheck&);
 bool IsPayDate(const Date& payDate) const;
 Date GetPayPeriodStartDate(const Date& payPeriodEndDate) const;

private:
 int itsEmpid;
 string itsName;
 string itsAddress;
 PaymentClassification* itsClassification;
 PaymentSchedule* itsSchedule;
 PaymentMethod* itsPaymentMethod;
 Affiliation* itsAffiliation;
};

#endif
```

程序 19.55　Employee.cpp

```
#include "Employee.h"
#include "NoAffiliation.h"
#include "PaymentClassification.h"
#include "PaymentSchedule.h"
#include "PaymentMethod.h"
#include "Paycheck.h"

Employee::~Employee()
{ delete itsClassification;
  delete itsSchedule;
  delete itsPaymentMethod;
}

Employee::Employee(int empid, string name, string address)
: itsEmpid(empid)
, itsName(name)
, itsAddress(address)
, itsAffiliation(new NoAffiliation())
, itsClassification(0)
, itsSchedule(0)
, itsPaymentMethod(0)
{
}

void Employee::SetName(string name)
{
```

```
  itsName = name;
}

void Employee::SetAddress(string address)
{
  itsAddress = address;
}

void Employee::SetClassification(PaymentClassification* pc)
{
  delete itsClassification;
  itsClassification = pc;
}

void Employee::SetSchedule(PaymentSchedule* ps)
{
  delete itsSchedule;
  itsSchedule = ps;
}

void Employee::SetMethod(PaymentMethod* pm)
{
  delete itsPaymentMethod;
  itsPaymentMethod = pm;
}

void Employee::SetAffiliation(Affiliation* af)
{
  delete itsAffiliation;
  itsAffiliation = af;
}

bool Employee::IsPayDate(const Date& payDate) const
{
  return itsSchedule->IsPayDate(payDate);
}

Date Employee::GetPayPeriodStartDate(const Date& payPeriodEndDate) const
{
  return itsSchedule->GetPayPeriodStartDate(payPeriodEndDate);
}

void Employee::Payday(Paycheck& pc)
{
  Date payDate = pc.GetPayPeriodEndDate();
  double grossPay = itsClassification->CalculatePay(pc);
  double deductions = itsAffiliation->CalculateDeductions(pc);
  double netPay = grossPay - deductions;
  pc.SetGrossPay(grossPay);
  pc.SetDeductions(deductions);
  pc.SetNetPay(netPay);
  itsPaymentMethod->Pay(pc);
}
```

19.6 主 程 序

现在，可以用一个循环来表示薪水支付应用的主程序，该循环首先解析从一个输入源到来的操作，然后执行这些操作。图 19.34 和图 19.35 描绘了主程序的静态和动态模型。思路很简单：

PayrollApplication 处在一个循环中，交替地从 TransactionSource 获取操作，然后执行这些操作对象。请注意，这和图 19.1 中显示的不同，它表明我们的理解迁移到了一个更抽象的结构。

TransactionSource 是一个抽象类，可以有多种实现方式。静态图中显示了名为 TextParseTransactionSource 的派生类，它读取输入的文本流并解析出像用例中描绘的操作。接着，该对象创建出相应的 Transaction 对象并把它们发送给 PayrollApplication。

TransactionSource 中接口和实现的分离使得操作的来源成为抽象的；例如，我们可以容易地把 PayrollApplication 和 GUITransactionSource 或者 RemoteTransactionSource 连接起来。

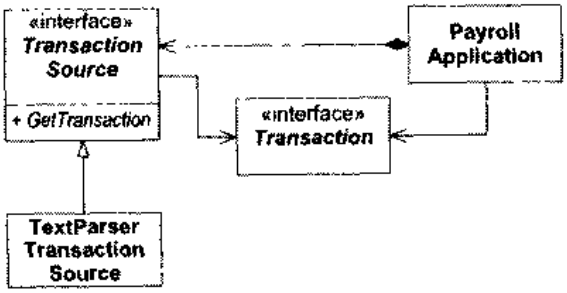


图 19.34　主程序的静态模型

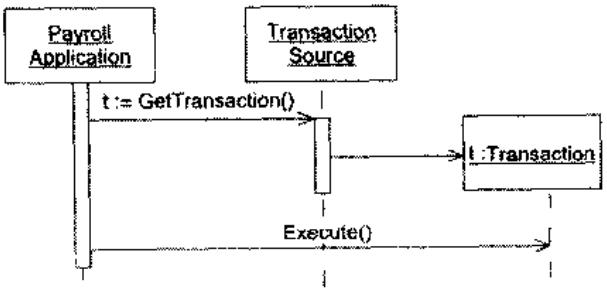


图 19.35　主程序的动态模型

19.7　数　据　库

既然已经完成了本次迭代中的分析、设计以及（大部分的）实现工作，现在可以考虑数据库的作用了。PayrollDatabase 类明显地封装了涉及持久化的工作。PayrollDatabase 所包含的对象的生存期必须要比该应用程序的任意一次单独的运行时间要长。如何实现这一点呢？显然，测试用例中使用的暂态（transient）机制对于真实系统来说是不够的。我们有数种选择。

我们可以使用面向对象数据库管理系统（OODBMS）来实现 PayrollDatabase。这可以使实际对象驻留在数据库的永久存储器中。作为设计者，我们只要做非常少量的工作，因为 OODBMS 不会向我们的设计中增加过多的新东西。OODBMS 的一个主要优点是它们对应用程序的对象模型没有（或者有很小的）影响。就设计而言，这样的数据库几乎不存在。[①]

另一种方法是使用简单的平面（flat）文件来记录数据。在初始化的时，PayrollDatabase 对象可以读取该文件并在内存中构建必需的对象。在程序结束时，PayrollDatabase 对象可以写下该文本文件的一个新版本。当然，对于拥有成百上千雇员的公司，或者对于希望实时并发访问薪水支付数据库的公司来说，这种方法是无法满足的。不过，这种方法足以应付较小的公司，并且当然可以把它作为一种机制，使用这种机制可以在无需引入大型数据库的情况下测试应用程序中其余的类。

还有一种方法是在 PayrollDatabase 对象中合入一个关系数据库管理系统（RDBMS）。此时，PayrollDatabase 对象可以对 RDBMS 进行适当的查询以在内存中临时地创建必需的对象。

关键在于，就应用程序而言，数据库只是管理存储的机制而已。通常都不应该把它们当做设计和实现的主要因素。就像我们在此所演示的那样，可以把它们留到最后并作为细节处理。[②]这样，

---

① 我对此持乐观态度。在像薪水支付一样简单的应用中，使用 OODBMS 对程序设计的影响非常小。当应用变得越来越复杂时，OODBMS 对于应用的影响就会增加。尽管如此，它还是远小于 RDBMS 对于应用程序的影响。

② 有时数据库的类型就是应用的需求之一。也许会把 RDBMS 提供的强大的查询和报表系统列为应用的需求。不过，即使这种需求是明显的，设计者仍然应该解除应用设计和数据库设计之间的耦合。应用设计不应该依赖于任何特定类型的数据库。

在实现必要的持久化功能以及创建一些机制去测试应用程序的其余部分时，我们就可以有许多有趣的方案可供选择。并且，我们也没有和任何特定的数据库技术或者产品绑定在一起。我们可以基于设计的其余部分自由选择需要的数据库，并且保留有在将来需要时更改或者替换数据库产品的自由。

19.8 薪水支付系统设计总结

我们用了大约 50 幅图以及 3300 行代码展示了薪水支付应用程序一次迭代的设计和实现。由于使用了大量的抽象和多态，使得绝大部分的设计对于薪水支付策略的更改做到了封闭。例如，可以更改应用程序去处理那些依据标准的薪水和奖金时间表每季度支付一次的雇员。这个更改需要增加一部分设计内容，但是现有的设计和代码基本上无需改动。

在这个过程中，我们很少考虑是否正在进行分析、设计或者实现。相反，我们全神贯注于清楚和封闭的问题。在任何可能的地方，我们都尽力找出潜在的抽象。结果，我们得到了一个良好的薪水支付应用的初始设计，并且拥有了一组在整体上和问题领域密切相关的核心类。

19.8.1 历史

我在 1995 年写过一本名为 *Designing Object-Oriented C++ Applications using the Booch Method* 的书，本章中的图示就衍生自该书相应章节中的 Booch 图。那些图示创建于 1994 年。在创建它们时，我同样也编写了一些实现它们的代码以确保那些图示合理。不过，那时编写的代码数量远没有本章中展示的多。因此，那些图示无法受益于代码和测试的有效反馈。这种反馈的缺乏明显可见。

我撰写本章的顺序和此处呈现的相同。其中，测试用例都是先于产品代码编写的。在多数情况下，测试都是以增量的方式创建，并和产品代码一起演化。只要图示合理，就依照图示编写产品代码。但是也有几处不合理的地方，所以我改变了代码的设计。

第一个不合理的地方在 19.3.1 节中，当时我决定不在 Employee 对象中放置多个 Affiliation 实例。另一个不合理的地方在 19.4.2 节中，当时我没有考虑到要在 ChangeMemberTransaction 中记录下雇员的协会成员关系。

这是正常的。如果在没有反馈的情况下进行设计，就必然会犯错误。正是来自测试用例以及运行代码的反馈帮助我们发现了这些错误。

19.8.2 资源

可以到 Prentice Hall 的 Web 站点，或者 www.objectmentor.com/PPP 上获取本章代码的最终版本。

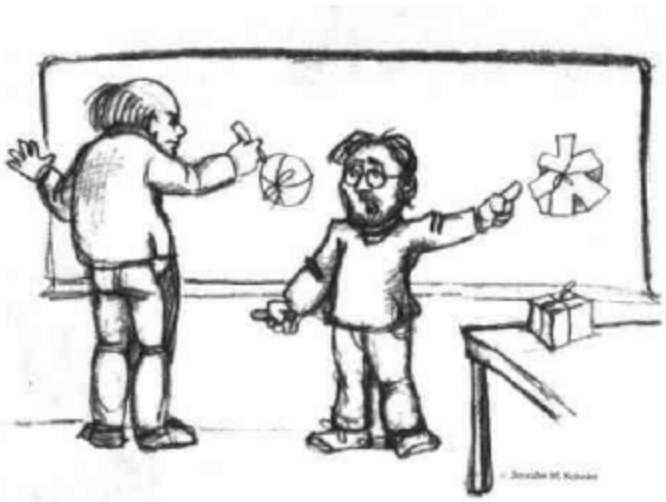
参考文献

1. Jacobson, Ivar. *Object-Oriented Software Engineering, A Use-Case-Driven Approach*. Workingham, England:Addison-Wesley, 1992.

第Ⅳ部分　打包薪水支付系统

在本部分中，我们要研究有助于把大的软件系统分割成包（package）的设计原则。第 20 章论述了这些原则，第 21 章会讲述一个有助于改进包结构的模式，第 22 章则演示了如何把这些原则和模式应用到薪水支付系统中。

第 20 章　包的设计原则



优美的包裹（package）。

——安东尼

随着应用程序规模和复杂度的增加，需要在更高层次对它们进行组织。类对于小型应用程序来说是非常方便的组织单元，但是对于大型应用程序来说，如果仅仅使用类作为惟一的组织单元，就会显得粒度过细。因此，就需要比类“大”的“东西”来辅助大型应用程序的组织。这个“东西”就是包（package）。

本章概述了 6 个原则。前 3 个原则关注包的内聚性，这些原则能够指导我们对类组包。后 3 个原则关注包的耦合性，这些原则帮助我们确定包之间的相互关系。最后两个原则还描述了一组依赖性管理度量（Dependency Management metrics）方面的内容，开发者可以据此对设计中的依赖结构进行度量和刻画。

20.1　如何进行包的设计

在 UML 的概念中，包可以用作包容一组类的容器。通过把类组织成包，我们可以在更高层次的抽象上来理解设计。我们也可以通过包来管理软件的开发和发布。目的就是根据一些原则对应用程序中的类进行划分，然后把那些划分后的类分配到包中。

但是类经常会和其他类之间存在依赖关系，这些依赖关系还经常会跨越包的边界。因此，包之间也会产生依赖关系。包之间的依赖关系展现了应用程序的高层组织结构，我们应该对这些关系进行管理。

这就提出了很多问题：

（1）在向包中分配类时应该依据什么原则？
（2）应该使用什么设计原则来管理包之间的关系？
（3）包的设计应该先于类呢（自顶向下）？还是类的设计应该先于包（自底向上）？
（4）如何实际表现出“包”？在 C++中如何表现？在 Java 中如何表现？在某种并发环境中又如何表现？
（5）包创建好后，我们应当将它们用于何种目的？

本章讲述了 6 个设计原则，涉及包的创建、相互关系的管理以及包的使用。前 3 个原则是用来

指导如何把类划分到包中的。后 3 个原则是用来处理包之间的相互关系的。

20.2　粒度：包的内聚性原则

这里要讲述的 3 个关于包的内聚性原则，可以帮助开发者决定如何把类划分到包中。这些原则依赖于这样的事实：至少已经存在一些类，并且它们之间的相互关系也已经确定。因此，这些原则是根据"自底向上"的观点对类进行划分的。

20.2.1　重用发布等价原则

重用的粒度就是发布的粒度。

当你重用一个类库时，对这个类库的作者有什么期望呢？你当然想得到好的文档，可以工作的代码，规格清晰的接口等等。但是，你还会有其他的期望。

首先，你希望代码的作者能保证为你维护这些代码，只有这样才值得你在重用这些代码上花费时间。毕竟，如果需要你亲自去维护这些代码，那将会花费你大量的时间，这些时间也许可以自己用来设计一个小些但是好些的包。

其次，你希望代码的作者在计划对代码的接口和功能进行任何改变时，提前通知你一下。但是，仅仅通知一下是不够的。代码的作者必须尊重你拒绝使用任何新版本的权力。否则，当你处在开发进度中的一个关键时刻时，他可能发布了一个新的版本，或者他对代码进行了改变，之后就干脆再也无法与你的系统兼容了。

无论在哪种情况下，如果你决定不接纳新版本，作者必须保证对于你所使用的旧版本继续提供一段时间的支持。这段时间也许只有 3 个月，或者长达 1 年，你们两个人之间必须就这些事情进行磋商。但是，他不能够和你断绝关系并且拒绝对你提供支持。如果他不同意对你使用的稍旧一点的版本提供支持，那么你就应该认真的考虑一下是否情愿忍受对方反复无常的变化，而继续使用他的代码。

这个问题主要是行政问题。如果有其他的人将要重用代码，就必须要进行行政和支持方面的工作。但是这些行政上的问题对于软件的包结构具有深刻的影响。为了给重用者提供所需的保证，代码的作者必须把他们的软件组织到一个可重用的包中，并且通过版本号对那些包进行跟踪。

REP 指出，一个包的重用粒度（granule of reuse）可以和发布粒度（granule of release）一样大。我们所重用的任何东西都必须同时被发布和跟踪。简单的编写一个类，然后声称它是可重用的做法是不现实的。只有在建立一个跟踪系统，为潜在的使用者提供所需要的变更通知、安全性以及支持后，重用才有可能。

REP 带给我们了关于如何把设计划分到包中的第一个提示。由于重用性必须是基于包的，所以可重用的包必须包含可重用的类。因此，至少，某些包应该由一组可重用的类组成。

行政上的约束力将会影响到对于软件的划分，这看上去会令人不安，但是软件不是一个可以依据纯数学规则组织起来的纯数学实体。软件是一个人的智力活动的产品。软件由人创建并被人使用，并且如果我们将要对软件进行重用，那么它肯定以一种人认为方便重用的方式进行划分。

那么，关于包的内部结构方面，我们学到了什么呢？我们必须从潜在的重用者的角度去考虑包

的内容。如果一个包中的软件是用来重用的，那么它就不能再包含不是为了重用目的而设计的软件。一个包中的软件要么都是可重用的，要么都不是可重用的。

可重用性不是惟一的标准，我们也要考虑重用这些软件的人。当然，一个容器类库是可重用的，一个金融方面的框架也是可重用的。但是，我们不希望把它们放进同一个包中。很多希望重用容器类库的人可能对于金融框架根本不感兴趣。因此，我们希望一个包中的所有类对于同一类用户来说都是可重用的。我们不希望一个用户发现包中所包含的类中，一些是他所需要的，另一些对他却完全不适合。

20.2.2 共同重用原则

一个包中的所有类应该是共同重用的。如果重用了包中的一个类，那么就要重用包中的所有类。

这个原则可以帮助我们决定哪些类应该放进同一个包中。它规定了趋向于共同重用的类应该属于同一个包。

类很少会孤立的重用。一般来说，可重用的类需要与作为该可重用抽象一部分的其他类协作。CRP 规定了这些类应该属于同一个包。在这样的一个包中，我们会看到类之间有很多的互相依赖。

一个简单的例子是容器类以及与它关联的迭代器类。这些类彼此之间紧密耦合在一起，因此必须共同重用。所以它们应该在同一个包中。

但是，CRP 告诉我们的不仅仅是什么类应该共同放入一个包中。它还告诉我们什么类不应该放入同一个包中。当一个包使用了另一个包时，它们之间会存在一个依赖关系。也许一个包仅仅使用了另外一个包中的一个类。然而，那根本不会削弱这两个包之间的依赖关系。使用者包依然依赖于被使用的包。每当被使用的包发布时，使用者包必须要进行重新验证和重新发布。即使发布的原因仅仅是由于更改了一个使用者包根本不关心的类，也必须要这样做。

此外，包也经常以共享库、DLL、JAR 等物理表示的形式出现。如果被使用的包以 JAR 的形式发布，那么使用这个包的代码就依赖于整个 JAR。对 JAR 的任何修改——即使所修改的是与用户代码无关的类，仍然会造成这个 JAR 的一个新版本的发布。这个新 JAR 仍然要重新发行，并且使用这个 JAR 的代码也要进行重新验证。

因此，我想确信当我依赖于一个包时，我将依赖于那个包中的每一个类。换句话说，我想确信我放入一个包中的所有类是不可分开的，仅仅依赖于其中一部分的情况是不可能的。否则，我将要进行不必要的重新验证和重新发行，并且会白费相当数量的努力。

因此，CRP 告诉我们更多的是，什么类不应该放在一起。CRP 规定相互之间没有紧密联系的类不应该在同一个包中。

20.2.3 共同封闭原则

包中的所有类对于同一类性质的变化应该是共同封闭的。一个变化若对一个包产生影响，则将对该包中的所有类产生影响，而对于其他的包不造成任何影响。

这是单一职责原则对于包的重新规定。正如 SRP 规定的一个类不应该包含多个引起变化的原因那样，这条原则规定了一个包不应该包含多个引起变化的原因。

在大多数的应用中，可维护性的重要性是超过可重用性的。如果一个应用中的代码必须更改，那么我们宁愿更改都集中在一个包中，而不是分布在多个包中。如果更改集中在一个单一的包中，那么我们仅仅需要发布那一个更改了的包。不依赖于那个更改了的包的其他包则不需要重新验证或者重新发布。

CCP 鼓励我们把可能由于同样的原因而更改的所有类共同聚集在同一个地方。如果两个类之间有非常紧密的绑定关系，不管是物理上的还是概念上的，那么它们总是会一同进行变化，因而它们应该属于同一个包中。这样做会减少软件的发布、重新验证、重新发行的工作量。

这个原则和开放封闭原则（OCP）密切相关。本原则中“封闭”这个词和 OCP 中的具有同样的含意。OCP 规定了类对于修改应该是封闭的，对于扩展应该是开放的。但是正如我们所学到的，100%的封闭是不可能做到的。应当进行有策略的封闭。我们所设计的系统应该对于我们经历过的最常见的变化做到封闭。

CCP 通过把对于一些确定的变化类型开放的类共同组织到同一个包中，从而增强了上述内容。因而，当需求中的一个变化到来时，那个变化就会很有可能被限制在最小数量的包中。

20.2.4　包内聚性总结

过去，我们对内聚性的认识要远比上面 3 个原则所蕴含的简单。我们习惯于认为内聚性不过是指一个模块执行一项并且仅仅一项功能。然而，这 3 个关于包内聚性的原则描述了有关内聚性的更加丰富的变化。在选择要共同组织到包中的类时，必须要考虑可重用性与可开发性（developability）之间的相反作用力。在这些作用力和应用的需要之间进行平衡不是一件简单的工作。此外，这个平衡几乎总是动态的。也就是说，今天看起来合适的划分到了明年也许就不再合适了。因此，当项目的重心从可开发性向可重用性转变时，包的组成很可能会变动并随时间而演化。

20.3　稳定性：包的耦合性原则

接下来的 3 个原则用来处理包之间的关系。这里，我们会再次碰到可开发性和逻辑设计之间的冲突力（tension）。来自技术和行政方面的作用力都会影响到包的组织结构，并且这种作用力还是易变的。

20.3.1　无环依赖原则

在包的依赖关系图中不允许存在环。

你曾经有过这样的经历吗？工作了一整天，终于完成了某项功能后回家，不料第二天早晨一来却发现那项功能不再工作了。原因是什么呢？因为有人比你走的更晚，并且更改了你所依赖的某些东西！我称其为“晨后综合症”。

如果开发环境中存在有许多开发人员都在更改相同的源代码文件集合的情况，那么就会发生晨后综合症。在仅有几个开发人员的相对小的项目中，这不是一个大问题。但是当项目和开发团队的规模增长时，晨后综合症就会带来可怕的噩梦。在缺乏纪律的团队中，几周都无法构建出一个稳定的项目版本的情况是很常见的。相反，每个人都忙于一遍遍地更改他们的代码，试图使之能够相容

于其他人所做的最近更改。

近几十年来，逐步形成了两个针对该问题的解决方案。这两个方案都来自电信业。第一个是“每周构建”，第二个是 ADP。

20.3.2　每周构建

每周构建常常应用在中等规模的项目中。它的工作方式为：在一周的前 4 天，所有的开发人员互不打扰，工作在各自私有的代码拷贝上，而不担心互相之间的集成问题。周五，他们集成进各自的更改并构建系统。

这样，开发人员每周可以单独工作 4 天，这具有极大的好处。当然，不利之处在于周五要付出巨大的集成代价。

糟糕的是，随着项目的增长，集成工作变得无法在周五完成。集成的工作量会一直增加到需要周六加班才能完成。只需几次这样的周六加班，开发人员就会认为其实应该在周四开始集成。这样，集成的起始时间就会慢慢蔓延至一周的中期。

随着开发和集成时间比率的降低，团队的效率也随之降低。最后，这会非常的令人沮丧，以至于开发人员或者项目管理者宣称应该把构建安排为每两周一次。这种做法暂时可以应付一下，但是集成的时间仍然不断地随着项目规模一起增长。

这最终会导致危机。为了保持效率，就必须要不断地延长构建周期。但是，延长构建周期会增加项目的风险。集成和测试变得越来越难进行，团队也丧失了快速反馈带来的好处。

20.3.3　消除依赖环

通过把开发环境划分成可发布的包，可以解决上述问题。这些包可以作为工作单元被一个开发人员或者一个开发团队拆出（check out）。当开发人员使一个包可以工作时，就把它发布给其他开发人员使用。他们赋予该包一个版本号并把它移到一个供其他开发人员使用的目录中。接着，他们可以在自己的私有区域中继续修改他们的包。其他所有人都使用那个已经发布的版本。

当制作了一个包的新版本时，其他开发团队可以决定是否马上采用这个新的版本。如果决定不采用，则他们完全可以继续使用老的版本。一旦觉得自己准备就绪，就可以开始使用新的版本。

因此，所有的开发团队都不会受其他开发团队支配。对一个包作的更改不必立即反应到其他开发团队中。每个开发团队独立决定何时采用目前所使用的包的新版本。此外，集成是以小规模增量的方式进行的。这样，就不会再发生所有的开发人员必须集合到一起把他们做的每一件工作集成起来的情况。

这是一个非常简单、合理的过程，并被广泛使用。不过,要使其能够工作，就必须要对包的依赖关系结构进行管理。包的依赖关系结构中不能有环。如果依赖关系结构中存在环，那么就不能避免最后综合症。

考虑图 20.1 中的包图。图中展示了组成一个应用程序的非常典型的包结构。相对于本例的意图来说，该应用程序的功能并不重要。重要的是包的依赖关系结构。请注意，该结构是一个有向图（directed graph）。其中，包是结点（node），依赖关系是有向边（directed edge）。

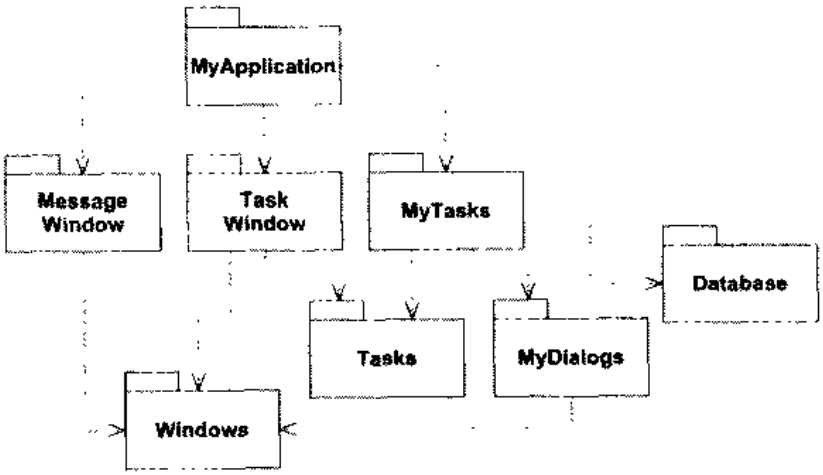


图 20.1　包结构是有向无环图

现在，请注意另外一件事情。无论从哪个包开始，都无法沿着依赖关系而绕回到这个包。该结构中没有环。它是一个有向无环图（DAG）。

当负责 MyDialogs 的团队发布了该包的一个新版本时，会很容易找出受到影响的包：只需逆着依赖关系指向寻找即可。因此，MyTasks 和 MyApplication 都会受到影响。当前工作于这两个包的开发人员就要决定何时应该和 MyDialogs 的新版本集成。

还要注意，当 MyDialogs 发布时，完全不会影响到系统中许多其他的包。它们不知道 MyDialogs，并且也不关心何时对 MyDialogs 进行了更改。这很好。这意味着发布 MyDialogs 的影响相对较小。

当工作于 MyDialogs 包的开发人员想要运行该包的测试时，只需把他们的 MyDialogs 版本和当前正使用的 Windows 包的版本一起编译、链接即可。不会涉及到系统中任何其他的包。这很好。这意味着工作于 MyDialogs 的开发人员只需较少的工作即可建立一个测试，而且他们要考虑的变数（variable）也不多。

在发布整个系统时，是自底向上进行的。首先编译、测试以及发布 Windows 包。接着是 MessageWindow 和 MyDialogs。在它们之后是 Task，然后是 TaskWindow 和 Database。接着是 MyTasks，最后是 MyApplication。这个过程非常清楚并且易于处理。我们知道如何去构建系统，因为我们理解系统各个部分之间的依赖关系。

20.3.4　包依赖关系图中环造成的影响

如果一个新需求迫使我们更改 MyDialogs 中的一个类去使用 MyApplication 中的一个类。这就产生了一个依赖关系环，如图 20.2 所示。

这个依赖关系环会导致一些直接后果。例如，工作于 MyTasks 包的开发人员知道，为了发布 MyTasks 包，他们必须得兼容 Task、MyDialogs、Database 以及 Windows。然而，由于依赖关系环的存在，他们现在必须也要兼容 MyApplication、TaskWindow 以及 MessageWindow。也就是说，现在 MyTasks 依赖于系统中所有其他的包。这就致使 MyTasks 非常难以发布。MyDialogs 有着同样的问题。事实上，该依赖关系环会迫使 MyApplication、MyTasks 以及 MyDialogs 总是同时发布。它们实际上已经变成了同一个大包。于是，在这些包上工作的所有开发人员就会再次遭受晨后综合症。他们彼此之间的发布

行动要完全一致，因为他们必须都要使用彼此间完全相同的版本。

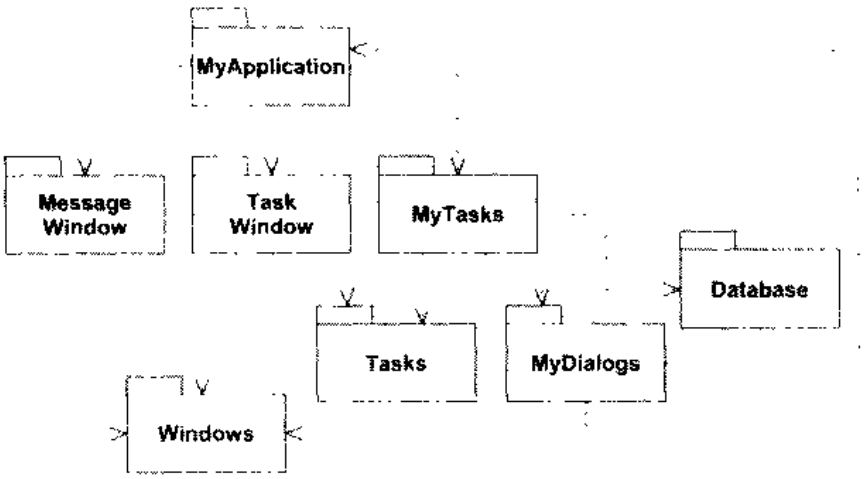


图 20.2　具有依赖环的包图

这还只是部分的问题。考虑一下在想要测试 MyDialogs 包时会发生什么。我们必须要链接进系统中所有其他的包，包括 Database 包。这意味着仅仅为了测试 MyDialogs 就必须要做一次完整的构建。这是不可忍受的。

如果想知道为何必须要链接进这么多不同的库，以及这么多其他人的代码，只需运行一个某个类的简单的单元测试即可，或许这是因为依赖关系图中存在环的缘故。这种环使得非常难以对模块进行隔离。单元测试和发布变得非常困难且易于出错。而且，在 C++中，编译时间会随模块的数目成几何级数增长。

此外，如果依赖关系图中存在环，就很难确定包构建的顺序。事实上，也许就不存在恰当的顺序。对于像 Java 一样要从编译过的二进制文件中读取它们的声明的语言来说，这会导致一些非常讨厌的问题。

20.3.5　解除依赖环

任何情况下，都可以解除包之间的依赖环并把依赖关系图恢复为一个 DAG。有两个主要的方法：

（1）使用依赖倒置原则（Dependency-Inversion Principle，DIP）。针对图 20.3 中的情况，可以创建一个具有 MyDialogs 需要的接口的抽象基类。然后，把该抽象基类放进 MyDialogs 中，并使 MyApplication 中的类从其继承。这就倒置了 MyDialogs 和 MyApplication 间的依赖关系，从而解除了依赖环。（参见图 20.3）

请注意，我们再次从客户的角度而不是服务者的角度出发来命名接口。这是接口属于客户规则的又一次应用。

（2）新创建一个 MyDialog 和 MyApplication 都依赖的包。把 MyDialog 和 MyApplication 都依赖的类移到这个新包中。（参见图 20.4）

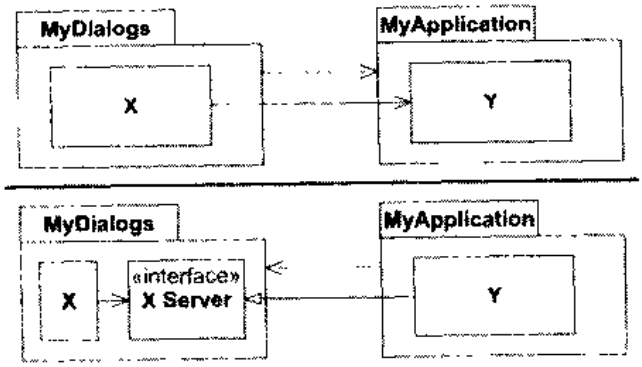


图 20.3　使用依赖倒置解除依赖环

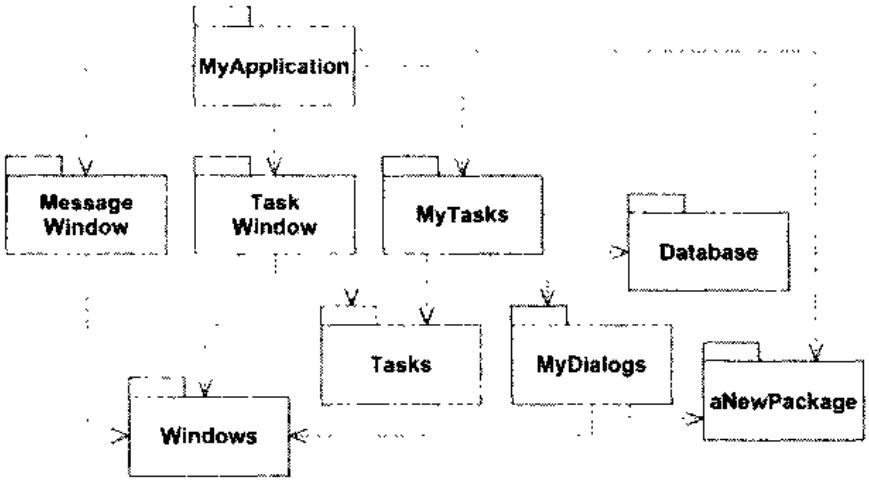


图 20.4　使用新包解除依赖环

20.3.6　抖动

第二个解决方案意味着，在需求改变面前包的结构是不稳定的。事实上，随着应用程序的增长，包的依赖关系结构会抖动（jitter）和增长。因此，必须要始终对依赖关系结构中环的情况进行监控。如果出现了环，就必须要使用某种方法把其解除。有时这意味着要创建新的包，致使依赖关系结构增长。

20.4　自顶向下设计

讨论到现在，我们可以得出一个必然的结论：不能自顶向下设计包的结构。这意味着包结构不是设计系统时首先考虑的事情之一。事实上，包结构应该是随着系统的增长、变化而逐步演化的。

也许你会认为这是违反直觉的。我们已经认为像包这样的大粒度分解同样也是高层的功能分解。当我们看到一个像包依赖关系结构这样的大粒度分组时，就会觉得包应该以某种方式描绘了系统的功能。然而，这可能不是包依赖关系图的一个属性。

事实上，包的依赖关系图和描绘应用程序的功能之间几乎没有关系。相反，它们是应用程序可构建性的映射图。这就是为何不在项目开始时设计它们的原因。在项目开始时，没有软件可构建，因此也无需构建映射图。但是，随着实现和设计初期累积的类越来越多，对依赖关系进行管理，避免项目开发中出现最后综合症的需要就不断增长。此外，我们也想尽可能地保持更改的局部化，所以我们开始关注 SRP 和 CCP，并把可能会一同变化的类放在一起。

随着应用程序的不断增长，我们开始关注创建可重用的元素。于是，就开始使用 CRP 来指导包的组合。最后，当环出现时，就会使用 ADP，从而包的依赖关系图会出现抖动以及增长。

如果在设计任何类之前试图去设计包的依赖关系结构，那么很可能会遭受惨败。我们对于共同封闭还没有多少了解，也还没有觉察到任何可重用的元素，从而几乎当然会创建产生依赖环的包。所以，包的依赖关系结构是和系统的逻辑设计一起增长和演化的。

20.5　稳定依赖原则

朝着稳定的方向进行依赖。

设计不能是完全固定的。要使设计可维护，某种程度的易变性是必要的。我们通过遵循共同封闭原则（CCP）来达到这个目标。使用这个原则，可以创建对某些变化类型敏感的包。这些包被设计成可变的。我们期望它们变化。

对于任何包而言，如果期望它是可变的，就不应该让一个难以更改的包依赖于它！否则，可变的包同样也会难以更改。

你设计了一个易于更改的包，其他人只要创建一个对它的依赖就可以使它变得难以更改，这就是软件的反常特性。没有改变你的模块中任何一行代码，可是它突然之间就变得难以更改了。通过遵循 SDP，我们可以确保那些打算易于更改的模块不会被那些比它们难以更改的模块所依赖。

20.5.1　稳定性

把一枚硬币竖立放置，在这种状态下，它是稳定的吗？你很可能会说它不稳定。不过，除非有干扰，否则它会保持这种状态很长一段时间。所以，稳定性和变化的频率没有直接关系。硬币的状态没有变化，但是却很难认为它是稳定的。

韦伯斯特认为，如果某物“不容易被移动”，就认为它是稳定的。[1]稳定性和更改所需要的工作量有关。硬币不是稳定的，因为推倒它所需的工作量是非常少的。但是，桌子是非常稳定的，因为推倒它要花费相当大的努力。

这和软件有什么关系呢？使软件包难以更改的因素有许多：它的规模、复杂性、清晰程度等等。我们会忽略所有这些因素而关注某个不同的东西。要使一个软件包难以改变，一个肯定可行的方法是让许多其他的软件包依赖于它。具有很多输入依赖关系的包是非常稳定的，因为要使所有依赖于它的包能够相容于对它所做的所有更改，往往需要非常大的工作量。

图 20.5 中展示了一个稳定的包 X。有 3 个包依赖于它；因此，就有 3 个合理的理由不去更改它。我们称 X 对这 3 个包负有责任。另外，X 不依赖于任何包，因此所有的外部影响都不会使其改变。我们称 X 是无依赖性的。

另一方面，图 20.6 展示了一个非常不稳定的包。没有任何其他的包依赖于 Y；我们称 Y 是不承担责任的。此外，Y 依赖于 3 个包，所以它具有 3 个外部更改源。我们称 Y 是有依赖性的。

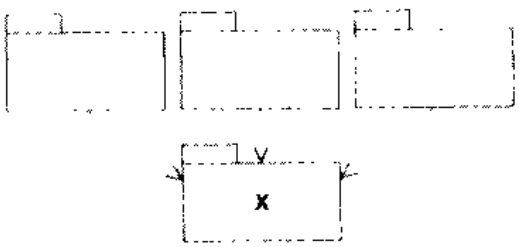


图 20.5　X：一个稳定的包

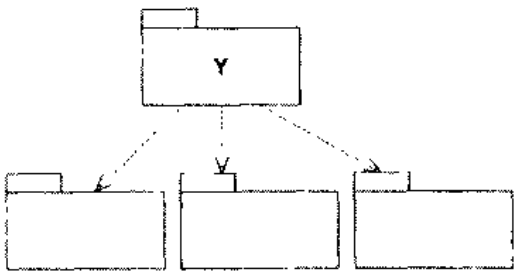


图 20.6　Y：一个不稳定的包

(1)《韦伯斯特新国际词典》第 3 部。

20.5.2　稳定性度量

如何度量一个包的稳定性呢？一种方法是计算进、出该包的依赖关系的数目。我们可以使用这些数值来计算该包的位置稳定性（positional stability）。

- (C_a) 输入耦合度（Afferent Coupling）：指处于该包的外部并依赖于该包内的类的类的数目。
- (C_e) 输出耦合度（Efferent Coupling）：指处于该包的内部并依赖于该包外的类的类的数目。
- (不稳定性 I)

$$I = \frac{C_e}{C_a + C_e}$$

该度量的取值范围是[0,1]。I=0 表示该包具有最大的稳定性。I=1 表示该包具有最大的不稳定性。

通过计算和一个包内的类有依赖关系的包外的类的数目，就可以计算出度量 C_a 和 C_e。考虑图 20.7 中的例子。

包之间的虚线箭头表示包的依赖关系。这些包的类之间的关系说明了这些依赖关系是如何形成的。其中有继承和关联关系。

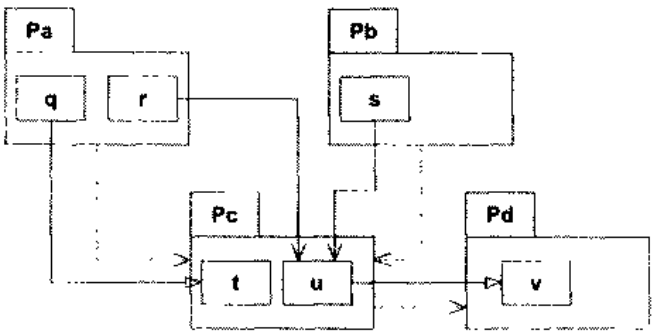


图 20.7　图表化 C_a、C_e 和 I

现在，我们来计算包 Pc 的稳定性。Pc 外部有 3 个类依赖于 Pc 内的类。所以，C_a=3。此外，Pc 外部有一个类被 Pc 内的类依赖。所以 C_e=1，I=1/4。

在 C++中，这些依赖关系一般是通过#include 语句表示的。事实上，如果把源代码组织成一个源文件中只有一个类的形式，那么计算度量 I 就会非常容易。在 Java 中，可以通过计算 import 语句以及类的修饰名称的数目来计算度量 I。

当一个包的 I 度量值为 1 时，就意味着没有任何其他的包依赖于该包（C_a=0）；而该包却依赖于其他的包（C_e>0）。这是一个包最不稳定的状态：它是不承担责任且有依赖性的。因为没有包依赖于它，所以它就没有不改变理由，而它所依赖的包会给它提供丰富的更改理由。

另一方面，当一个包的 I 度量值为 0 时，就意味着其他包会依赖于该包（C_a>0），但是该包却不依赖于任何其他的包（C_e=0）。它是负有责任且无依赖性的。这种包达到了最大程度的稳定性。它的依赖者使其难以更改，而且没有任何依赖关系会迫使它去改变。

SDP 规定一个包的 I 度量值应该大于它所依赖的包的 I 度量值（也就是说，I 度量值应该顺着依赖的方向减少）。

20.5.3　并非所有的包都应该是稳定的

如果一个系统中所有的包都是最大程度稳定的，那么该系统就是不能改变的。这不是所希望的情形。事实上，我们希望所设计出来的包结构中，一些包是不稳定的而另外一些是稳定的。图 20.8 中展示了一个具有 3 个包的系统的理想配置。

可改变的包位于顶部并依赖于底部稳定的包。把不稳定的包放在图的顶部是一个有用的约定，因为任何向上的箭头都意味着违反了 SDP。

图 20.9 展示了会违反 SDP 的做法。我们打算让 Flexible 包易于更改。我们希望 Flexible 是不稳定的。然而，一些工作于包 Stable 的开发人员，创建了一个对 Flexible 的依赖。这违反了 SDP，因为 Stable 的 *I* 度量值要比 Flexible 的 *I* 度量值小的多。结果，Flexible 就不再易于更改了。对 Flexible 的更改会迫使我们去处理该更改对 Stable 及其所有依赖者的影响。

要修正这个问题，我们就必须要以某种方式解除 Stable 对 Flexible 的依赖。为什么会存在这个依赖关系呢？我们假设 Flexible 中有一个类 C 被另一个 Stable 中的类 U 使用（参见图 20.10）。

可以使用 DIP 来修正这个问题。我们创建一个接口类 IU 并把它放进包 UInterface 中。我们确保接口 IU 中声明了 U 要使用的所有方法。接着，我们让 C 从这个接口继承（参见图 20.11）。这就解除了 Stable 对 Flexible 的依赖并促使这两个包都依赖于 UInterface。UInterface 非常稳定（*I*=0），而 Flexible 仍保持它必需的不稳定性（*I*=1）。现在所有依赖方向都是顺着 *I* 减小的方向的。

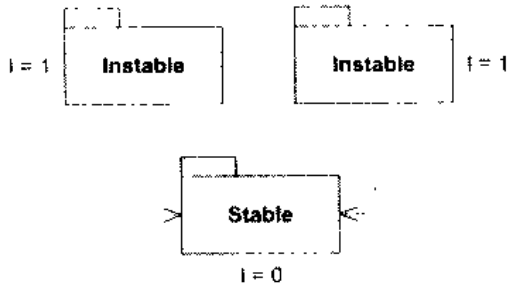


图 20.8　理想的包配置

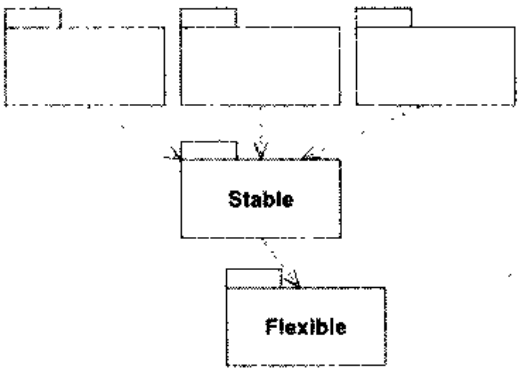


图 20.9　违反了 SDP

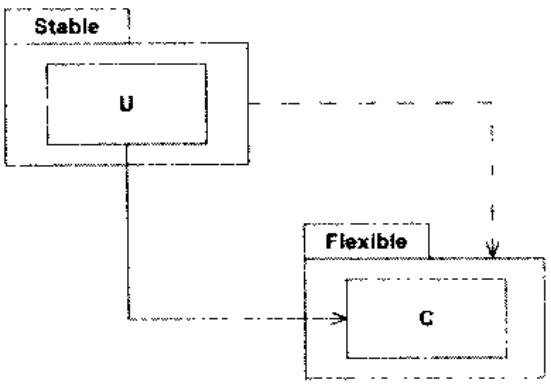


图 20.10　糟糕依赖关系的原因

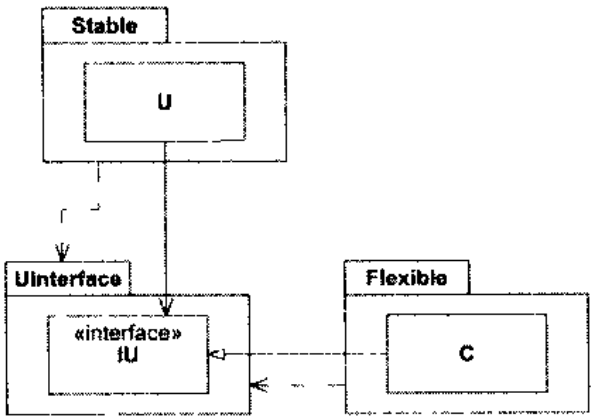


图 20.11　使用 DIP 修正稳定性违规

20.5.4　在哪里放置高层设计？

系统中的某些软件不应该经常改变。该软件代表着系统的高层构架和设计决策。我们希望这些构架决策是稳定的。因此，应该把封装系统高层设计的软件放进稳定的包中（*I*=0）。不稳定的包（*I*=1）中应该只包含那些很可能会改变的软件。

然而，如果把高层设计放进稳定的包中，那么体现高层设计的源代码就会难以更改。这会使设计变得不灵活。怎样才能让一个具有最高稳定性（I=0）的包足够的灵活，可以经受得住变化呢？在 OCP 中可以找到答案。OCP 原则告诉我们，那些足够灵活可以无需修改即可扩展的类是存在的，并且是所希望的。哪种类符合 OCP 原则呢？抽象类。

20.6　稳定抽象原则

包的抽象程度应该和其稳定程度一致。

该原则把包的稳定性和抽象性联系起来。它规定，一个稳定的包应该也是抽象的，这样它的稳定性就不会使其无法扩展。另一方面，它规定，一个不稳定的包应该是具体的，因为它的不稳定性使得其内部的具体代码易于更改。

因此，如果一个包是稳定的，那么它应该也要包含一些抽象类，这样就可以对它进行扩展。可扩展的稳定包是灵活的，并且不会过分限制设计。

SAP 和 SDP 结合在一起形成了针对包的 DIP 原则。这样说是准确的，因为 SDP 规定依赖应该朝着稳定的方向进行，而 SAP 则规定稳定性意味着抽象性。因此，依赖应该朝着抽象的方向进行。

然而，DIP 是一个处理类的原则。类没有灰度的概念（the shades of grey）。一个类要么是抽象的，要么不是。SDP 和 SAP 的结合是处理包的，并且允许一个包是部分抽象、部分稳定的。

20.6.1　抽象性度量

A 是一个测量包抽象程度的度量标准。它的值就是包中抽象类的数目和全部类的数目的比值。

$$A=\frac{N_a}{N_c}$$

N_c——包中类的总数

N_a——包中抽象类的数目。请记住，一个抽象类是一个至少具有一个纯接口（pure interface）的类，并且它不能被实例化。

A——抽象性。

度量 A 的取值范围是从 0 到 1。0 意味着包中没有任何抽象类。1 意味着包中只包含抽象类。

20.6.2　主序列

现在，我们来定义稳定性（I）和抽象性（A）之间的关系。我们可以创建一个以 A 为纵轴，I 为横轴的坐标图。如果在坐标图中绘制出两种“好”的包类型，会发现那些最稳定、最抽象的包位于左上角（0,1）处。那些最不稳定、最具体的包位于右下角（1,0）处（参见图 20.12）

并非所有的包都会落在这两个位置。包的抽象性和稳定性是有程度的。例如，一个抽象类派生自另一个抽象类的情况是很常见的。派生类是具有依赖性的抽象体。因此，虽然它是最大限度抽象的，但是它却不是最大程度稳定的。它的依赖性会降低它的稳定性。

因为不能强制所有的包都位于（0,1）或者（1,0），所以必须要假定在 *A/I* 图上有一个定义包的

合理位置的点的轨迹。我们可以通过找出包不应该在的位置（也就是，被排除的区域）来推断该轨迹的含意（参见图 20.13）。

考虑一个在（0,0）附近的包。这是一个高度稳定且具体的包。我们不想要这种包，因为它是僵化的。也无法对它进行扩展，因为它不是抽象的。并且由于它的稳定性，也很难对它进行更改。因此，通常，我们不期望看到设计良好的包位于（0,0）附近。（0,0）周围的区域被排除在外，我们称之为痛苦地带（Zone of Pain）。

应该注意，在有些情形中，包确实会落入痛苦地带。数据库模式（database schema）就是一个这样的例子。数据库模式的易变性是众所周知的，并且它还是非常具体、高度被依赖的。这就是为何面向对象应用程序和数据库之间的接口难以定义，以及数据库模式的更新通常都很痛苦的原因。

还有一个包位于（0,0）处的例子是包中含有一个具体的工具库。虽然这种包的 I 度量值是 1，但是，事实上它可能是稳定的。例如，考虑一下“string”包。即使它内部的所有类都是具体的，它也是稳定的。这种位于区域（0,0）的包不会造成损害，因为不太可能去改变它们。事实上，我们可以认为坐标图中还具有第 3 条轴线：易变性。假如这样的话，图 20.13 中展示的就是在易变性为 1 处的切面图。

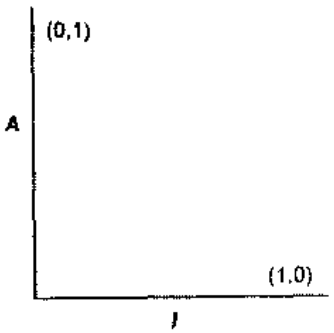


图 20.12　*A-I* 坐标图

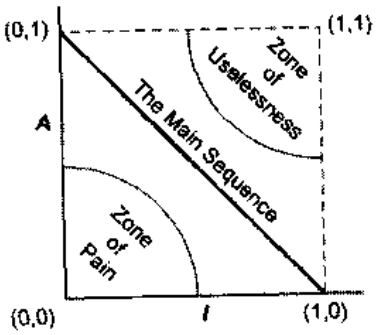


图 20.13　被排除的区域

考虑一个在（1,1）附近的包，这不是一个好位置，因为该位置处的包具有最大的抽象性却没有依赖者。这种包是无用的。因此，称这个区域为无用地带（Zone of Uselessness）。

显然，我们想让可变的包都尽可能地远离这两个被排除的区域。那些距离这两个区域最远的轨迹点组成了连接（1,0）和（0,1）的线。该线称为主序列（main sequence）。[①]

位于主序列上的包既不是太抽象，因为它具有稳定性，也不是太不稳定，因为它具有抽象性。它既不是无用的，又不是特别令人痛苦的。就其抽象性而言，它被其他的包依赖，就其具体性而言，它又依赖于其他的包。

显然，包的最佳位置位于主序列的两个端点处。不过，依据我的经验，项目中可以具有这种最佳特征的包少于一半。对其他的包来说，它们能够位于主序列上或者主序列的附近就已经很不错了。

22.6.3　到主序列的距离

为此，我们需要最后一个度量。如果希望包能够位于或者靠近主序列，那么我们可以创建一个度量来衡量包到这个理想位置的距离。

① 之所以采用“主序列”这个名字是因为我对天文学以及 HR 图（赫罗图，用于显示恒星真实亮度与其表面温度的关系）的爱好。

D——距离：

$$D=\frac{|A+I-1|}{\sqrt{2}}$$

该度量的取值范围是 [0, −0.707]。

D'——规范化的距离：

$$D'=|A+I-1|$$

这个度量使用起来要比 D 方便的一些，因为它的取值范围是[0,1]。其中 0 表示包正好位于主序列上，1 表示包到主序列的距离最远。

使用这个度量，可以全面分析一个设计和主序列间的一致性。首先计算出每个包的 D 度量值，然后对所有 D 值不在 0 附近的包进行复查和调整。事实上，这种分析非常有助于设计者确定哪些包更容易维护些，哪些包对变化更不敏感些。

同样，可以对设计进行统计分析。你可以计算出设计中所有包的 D 度量的均值和方差，并且期望一个均值和方差接近于 0 的符合主序列的设计。方差可以用来建立“控制限制”，以识别那些和所有其他的包相比显得“特别”的包（参见图 20.14）。

在这个分布图中[①]，我们看到大多数的包沿着主序列分布，但是其中一些包和均值之间的距离超过一个标准偏差（Z=1）。这些偏离的包值得留意一下。可能是某种原因导致它们非常抽象却具有很少的依赖者，或者非常具体却具有很多的依赖者。

另外一种使用该度量的方法是绘制出每个包的 D' 度量值随时间的分布图。图 20.15 中展示了一个这种图的模型。从中可以看到，一些奇怪的依赖关系已经在最近几次发布中蔓延进 Payroll 包中。图中显示了一个控制门限：D'=0.1。R2.1 已经超出了这个控制限制，这值得我们花费一些时间找出这个包远离主序列的原因。

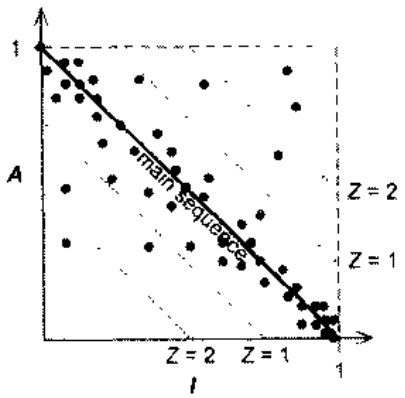


图 20.14　包的 D 值分布图

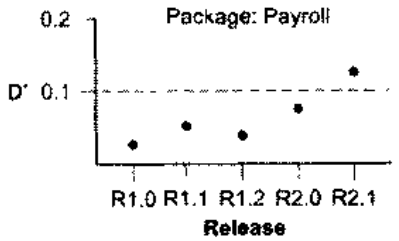


图 20.15　一个单独包的 D' 值的时间分布图

20.7　结　　论

本章中描述的依赖性管理度量可以测量一个设计与我认为“好”的依赖、抽象结构模式间的匹配程度。经验表明，依赖关系是有好坏之分的。该模式反映了这种经验。然而，度量不是万能的；它只是一个取代随意标准的测量方法。本章中选择的标准确实可能只对某些应用程序合适，而对另外一些则不合适。同样也可能存在有更好的测量设计质量的度量方法。

① 不是基于真实数据的。

第 21 章　FACTORY 模式

那个建造工厂的人建造了一座庙宇……

——凯文·库力吉〈1872－1933，
美国第 30 任总统〉

依赖倒置原则（DIP）[①]告诉我们应该优先依赖于抽象类，而避免依赖于具体类。当这些具体类不稳定时，更应该如此。因此，下面的代码片段违反了这个原则：

```
Circle c = new Circle(origin, 1);
```

Circle 是一个具体类。所以，创建 Circle 类实例的模块肯定违反了 DIP。事实上，任何一行使用了 new 关键字的代码都违反了 DIP。

有时，违反了 DIP 也是无害的。[②]一个具体类越有可能会改变，依赖于它就越有可能引发问题。但是如果这个具体类是稳定的，那么依赖于它就不会出现麻烦。

例如，创建 String 类的实例就不会带来麻烦。因为 String 类不可能会随时改变，所以依赖于它是很安全的。

另一方面，在一个正在进行的应用程序开发中，有很多具体类都是非常易变的。依赖于它们会带来问题。我们应当依赖于抽象接口，以使我们免受大多数变化的影响。

FACTORY 模式允许我们只依赖于抽象接口就能创建出具体对象的实例。所以，在正在进行的开发期间，如果具体类是高度易变的，那么该模式是非常有用的。

图 21.1 展示了一个有问题的场景。其中类 SomeApp 依赖于接口 Shape。SomeApp 完全通过 Shape 接口来使用 Shape 类的实例。它没有使用 Square 类或者 Circle 类的任何特定方法。糟糕的是，SomeApp 也创建了 Square 和 Circle 的实例，因此就不得不依赖于这些具体类。

这个问题可以通过对 SomeApp 应用 FACTORY 模式来修正（如图 21.2 所示）。其中我们看到了 ShapeFactory 接口。该接口中有两个方法：makeSquare 和 makeCircle。makeSquare 方法返回一个 Square 类的实例，而 makeCircle 方法返回一个 Circle 类的实例。不过，这两个函数返回值的类型都是 Shape。

程序 21.1 展示了 ShapeFactory 的代码，程序 21.2 展示了 ShapeFactoryImplementation 的代码。

请注意，这完全解决了对具体类的依赖问题。应用程序代码不再依赖于 Circle 或者 Square，却仍

① 第 11 章“依赖倒置原则（DIP）”。
② 这种无害的情况是相当常见的。

然可以去创建它们的实例。对这些实例的操作是通过 Shape 接口进行的，并且决不会调用特定于 Square 或者 Circle 的方法。

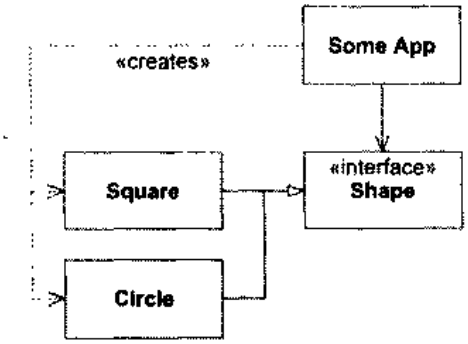


图 21.1 违反 DIP 原则创建具体类的应用程序

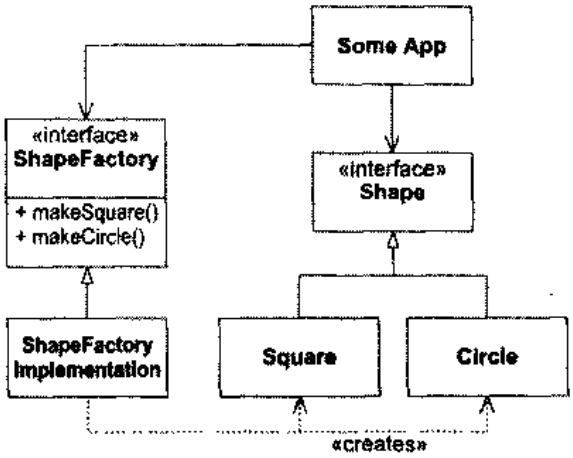


图 21.2 Shape 工厂

程序 21.1 ShapeFactory.java

```
public interface ShapeFactory
{
  public Shape makeCircle();
  public Shape makeSquare();
}
```

程序 21.2 ShapeFactoryImplementation.java

```
public class ShapeFactoryImplementation implements ShapeFactory
{
  public Shape makeCircle()
  {
    return new Circle();
  }

  public Shape makeSquare()
  {
    return new Square();
  }
}
```

依赖于具体类的问题已经解决了。虽然必须要在某个地方创建 ShapeFactoryImplementation，但是在所有其他的地方却根本不需要创建 Square 或者 Circle。ShapeFactoryImplementation 往往由 main 或者由一个隶属于 main 的初始化函数创建出来。

21.1 依赖关系环

敏锐的读者会认识到这种形式的 FACTORY 模式中存在的一个问题。针对每个 Shape 的派生类，类 ShapeFactory 都要有一个对应的方法。这就产生了一个依赖关系环，使得难以增加新的 Shape 派生类。每当增加一个新的 Shape 派生类时，都必须要向 ShapeFactory 接口中增加一个方法。在大多数的情况下，这意味着必须要重新编译、重新部署 ShapeFactory 的所有使用者。[①]

① 同样，这在 Java 中不是完全必要的。可以不重新编译和重新部署一个被更改的接口的客户，但是这是一种冒险行为。

通过牺牲一点类型安全性，可以解除这个依赖关系环。我们可以只给 ShapeFactory 提供一个以 String 作为参数的 make 函数，而不是为每个 Shape 的派生类都在 ShapeFactory 中提供一个方法。请参见程序 21.3 中示例。这项技术要求 ShapeFactoryImplementation 使用 if/else 链对传入的参数进行判断，选择出要实例化的 Shape 的派生类。如程序 21.4 和程序 21.5 所示。

程序 21.3 创建 Circle 实例的代码片段

```
public void testCreateCircle() throws Exception
{
   Shape s = factory.make("Circle");
   assert(s instanceof Circle);
}
```

程序 21.4 ShapeFactory.java

```
public interface ShapeFactory
{
  public Shape make(String shapeName) throws Exception;
}
```

程序 21.5 ShapeFactoryImplementation.java

```
public class ShapeFactoryImplementation implements ShapeFactory
{
  public Shape make(String shapeName) throws Exception
  {
    if (shapeName.equals("Circle"))
      return new Circle();
    else if (shapeName.equals("Square"))
      return new Square();
    else
      throw new Exception("ShapeFactory cannot create " + shapeName);
  }
}
```

有人也许会认为这样做是危险的，因为那些把 shape 的名字拼错的调用者会得到一个运行期错误而不是一个编译期错误。这种想法是正确的。然而，如果编写了适当数量的单元测试并且应用测试驱动的开发方法，那么远在这些运行期错误成为问题之前就可以捕获到它们。

21.2 可替换的工厂

使用工厂的一个主要好处就是可以把工厂的一种实现替换为另一种实现。这样，就可以在应用程序中替换一系列相关的对象（families of objects）。

例如，如果一个应用程序必须要适应于许多不同的数据库实现。在本例中，假设用户既可以使用平面文件也可以购买一个 Oracle 适配器。我们可以使用 PROXY 模式[①]来隔离应用程序和数据库实现。我们还可以使用工厂来实例化代理对象。图 21.3 展示了这个结构。

请注意，有两个 EmployeeFactory 的实现。一个创建与平面文件一起工作的代理，另一个创建与 Oracle 一起工作的代理。同样请注意，应用程序并不知道也不关心它在使用哪一个代理。

① 我们会在后面第 26 章中学习 PROXY 模式。现在，你只需知道 PROXY 就是知道如何从特定的数据库中读取特定对象的类。

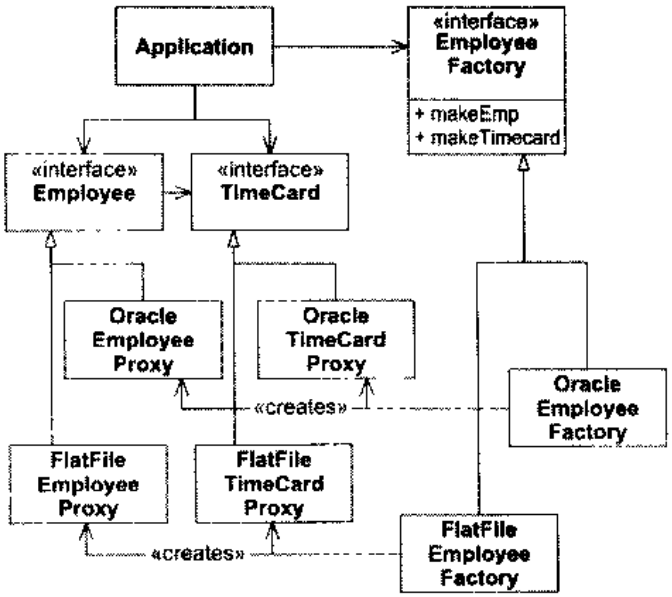


图 21.3 可替换的工厂

21.3 对测试支架使用对象工厂

在编写单元测试时，通常希望把一个模块和它所使用的模块隔离起来，单独去测试该模块的行为。例如，有一个使用数据库的 Payroll 应用程序（参见图 12.4）。我们可能希望在完全不使用数据库的情况下测试 Payroll 模块的功能。

图 21.4 Payroll 使用 Database

通过使用抽象的数据库接口，可以到达这个目标。在这个抽象接口的一个实现中使用真正的数据库。在另一个实现中，是测试代码，该测试代码模仿了数据库的行为，并且检查是否正确进行了数据库调用。图 21.5 展示了这个结构。PayrollTest 模块通过调用 PayrollModule 来测试它。它也实现了 Database 接口，所以可以捕获到 Payroll 向数据库发出的调用。这就使得 PayrollTest 可以确保 Payroll 具有正确的行为。它同样也使得 PayrollTest 可以模仿多种类型的数据库失败和问题，而以别的方式则很难引发这些失败和问题。通常，把这项技术称为欺骗（spoofing）。

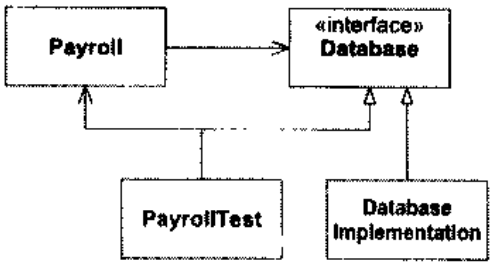


图 21.5 PayrollTest 欺骗 Database

然而，Payroll 如何获得作为 Database 的 PayrollTest 的实例呢？当然，Payroll 不会去创建 PayrollTest 的实例。显然，Payroll 必须以某种方式获得它将要使用的 Database 实现的一个引用。

在某些情况下，PayrollTest 把 Database 的引用传递给 Payroll 是相当自然的。在另一些情况下，有可能 PayrollTest 必须设置一个全局变量保存对 Database 的引用。还有一些情况下，Payroll 可能完全期望自己来创建 Database 实例。在最后一种情况中，可以使用 FACTORY 模式，通过传给 Payroll 另外一个工厂对象，来欺骗 Payroll 创建出 Database 的测试版本。

图 21.6 展示了一个可能的结构。Payroll 模块可以通过一个名为 GdatabaseFactory 的全局变量（或

者全局类中的静态变量）获取工厂。PayrollTest模块实现了DatabaseFactroy接口，并且把GdatabaseFactory设置为对自己的引用。当Payroll使用工厂去创建Database实例时，PayrollTest模块捕获这个调用并把指向自己的引用传回去。这样，Payroll确信自己已经创建了PayrollDatabase，而PayrollTest模块则可以完全欺骗Payroll模块并捕获所有的数据库调用。

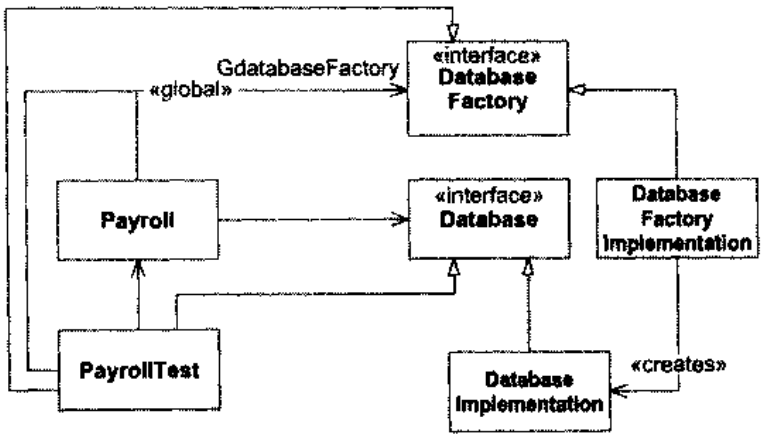


图 21.6　欺骗对象工厂

21.4　使用对象工厂有多么重要

严格按照DIP来讲，必须要对系统中所有的易变类使用工厂。此外，FACTORY模式的威力也是诱人的。这两个因素有时会诱使开发者把工厂作为缺省方式使用。我不推荐这种极端的做法。

我不是一开始就使用工厂。只是在非常需要它们的情况下，我才把它们放入到系统中。例如，如果有必要使用PROXY模式，那么就可能有必要使用工厂去创建持久化对象。或者，在单元测试期间，如果遇到了必须要欺骗一个对象的创建者的情况时，那么我很可能会使用工厂。但是我不是一开始就假设工厂是必要的。

使用工厂会带来复杂性，这种复杂性通常是可以避免的，尤其是在一个正在演化的设计的初期。如果缺省地使用它们，就会极大地增加扩展设计的难度。为了创建一个新类，就必须要创建出4个新类，这4个类是：2个表示该新类及其工厂的接口类，2个实现这些接口的具体类。

21.5　结　　论

工厂是有效的工具。在遵循DIP方面工厂有着重大的作用。它们使得高层策略模块在创建类的实例时无需依赖于这些类的具体实现。它们同样也使得在一组类的完全不同系列的实现间进行交换成为可能。然而，使用工厂会带来复杂性，这种复杂性通常是可以避免的。缺省地使用它们通常不是是最好的做法。

参考文献

1．Gamma, et al. *Design Pattern*. Reading, MA: Addison-Wesley, 1995.

第 22 章　薪水支付案例研究（第 2 部分）

“经验法则：如果你认为某样东西灵巧、精致，请小心——你可能是在放纵自我。”

——唐纳德·A. 诺曼，1990

（日常物品的设计，唐纳德·A. 诺曼，Doubleday，1990）

我们已经针对薪水支付问题做了大量的分析、设计和实现。不过，仍然还有许多决策要做。首先，解决薪水支付问题的程序员只有一个（我）。目前开发环境的结构与此一致。所有的程序文件被放在单一的目录中，除此之外没有更高级的结构。除了整个应用程序外，没有包，没有子系统，也没有可发布的单元。这种做法走不了太远。

我们必须承认，随着该程序的增长，会有更多的人员参与进来。为了便于多人开发，就必须要把源代码划分成便于拆出（check out）、修改和测试的包。

薪水支付应用程序目前有 3280 行代码组成，被分成大约 50 个不同的类和 100 个不同的源文件。虽然这不是一个巨大的数目，但是确实需要某种方式去组织这些代码。我们应该如何管理这些源文件呢？

同样，该如何对实现工作进行划分，才能使得开发过程可以平稳地进行而不会导致开发人员互相妨碍呢？我们希望把类划分成一些便于个人或者团队拆出和支持的组。

22.1　包结构和表示法

图 22.1 中的图示展示了薪水支付应用程序的一个可能的包结构。稍后，我们会解决该结构的适当性问题。现在，我们仅仅关注如何去文档化和使用这样一个结构。

附录 A 中描述了针对包的 UML 表示法。按照惯例，绘制包图时，依赖关系的方向应该向下。顶部的包依赖于其他包。底部的包被其他包依赖。

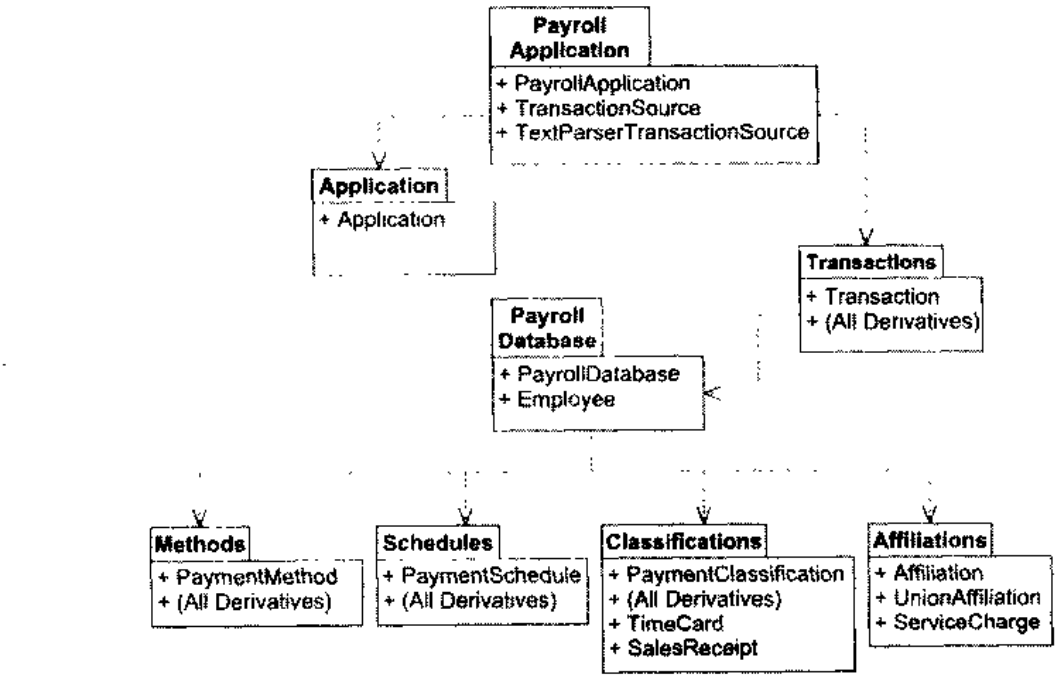


图 22.1　可能的薪水支付应用程序包图

图 22.1 把薪水支付应用程序划分成 8 个包。PayrollApplication 包中包含有 PayrollApplication 类、TransactionSource 类以及 TextParserTransaction 类。Transactions 包中包含有完整的 Transaction 类层次结构。仔细地检查图示，应该可以清楚地知道其他包中所包含的类。

依赖关系同样也是明显的。PayrollApplication 包依赖于 Transactions 包，因为 PayrollApplication 类调用了 Transaction::Execute 方法。Trasactions 包依赖于 PayrollDatabase 包，因为 Transaction 的许多派生类都直接和 PayrollDatabase 类传递消息。其他的依赖关系同理可得。

我是按照什么样的标准把这些类分组成包呢？我只不过是把那些看起来像是适合在一起的类放进相同的包中。按照我们在第 20 章所学习的，这可能不是一个好主意。

考虑一下，如果对 Classification 包做了一个更改，会发生什么呢？这个更改会迫使对 EmployeeDatabase 包进行重新编译和重新测试，这是正常的。但是，这个更改同时还会迫使对 Transaction 包进行重新编译和重新测试。确实应该对图 19.13[①]中的 ChangeClassificationTransaction 类以及它的 3 个派生类进行重新编译和重新测试，但是为什么也要重新编译和重新测试其他的类呢？

从技术角度出发，不需要去重新编译和重新测试其他的操作类。然而，如果它们是 Transaction 包的一部分，且为了适应 Classifications 包的改动要重新发布 Transaction 包，那么如果不把 Transaction 包作为一个整体重新编译、重新测试，就是不负责的行为。即使不重新编译和重新测试所有的操作类，包本身也必须重新发布、重新部署，于是它的所有客户至少需要重新验证，或许需要重新编译。

Transactions 包中的类没有共享相同的封闭性。每个类都对自己特定的变化敏感。ServiceChargeTransaction 对于 ServiceCharge 类的变化是开放的，而 TimeCardTransaction 对于 TimeCard 类的变化是开放的。从图 22.1 中可以看出，Transactions 包的某些部分实际上几乎依赖于软件的所有其

① 原书中为 19.3，应该为 19.13——译者注。

他部分。因此，这个包的发布率是非常高的。每当它下面的某一部分改变时，就必须要对 Transactions 包进行重新验证和重新发布。

PayrollApplication 包更加易受影响：对系统中任何部分的任何更改都会影响到该包，所以它的发布率肯定是非常高的。你也许会认为这是不可避免的——当一个包位于包依赖关系层次结构更高层次时，它的发布率一定会增加。幸运地是，这是不正确的，并且面向对象设计的主要目标之一就是要避免这种症状。

22.2　应用公共封闭原则（CCP）

考虑一下图 22.2。该图根据薪水支付应用程序中类的封闭性对它们进行分组。例如，PayrollApplication 包中包含有 PayrollApplication 类和 TransactionSource 类。这两个类都依赖于包 PayrollDomain 中的抽象类 Transaction。请注意，TextParseTransactionSource 类在另一个依赖于抽象类 PayrollApplication 的包中。这就创建了一个倒置的结构，其中细节依赖于通用部分，并且通用部分是无依赖性的。该结构符合 DIP。

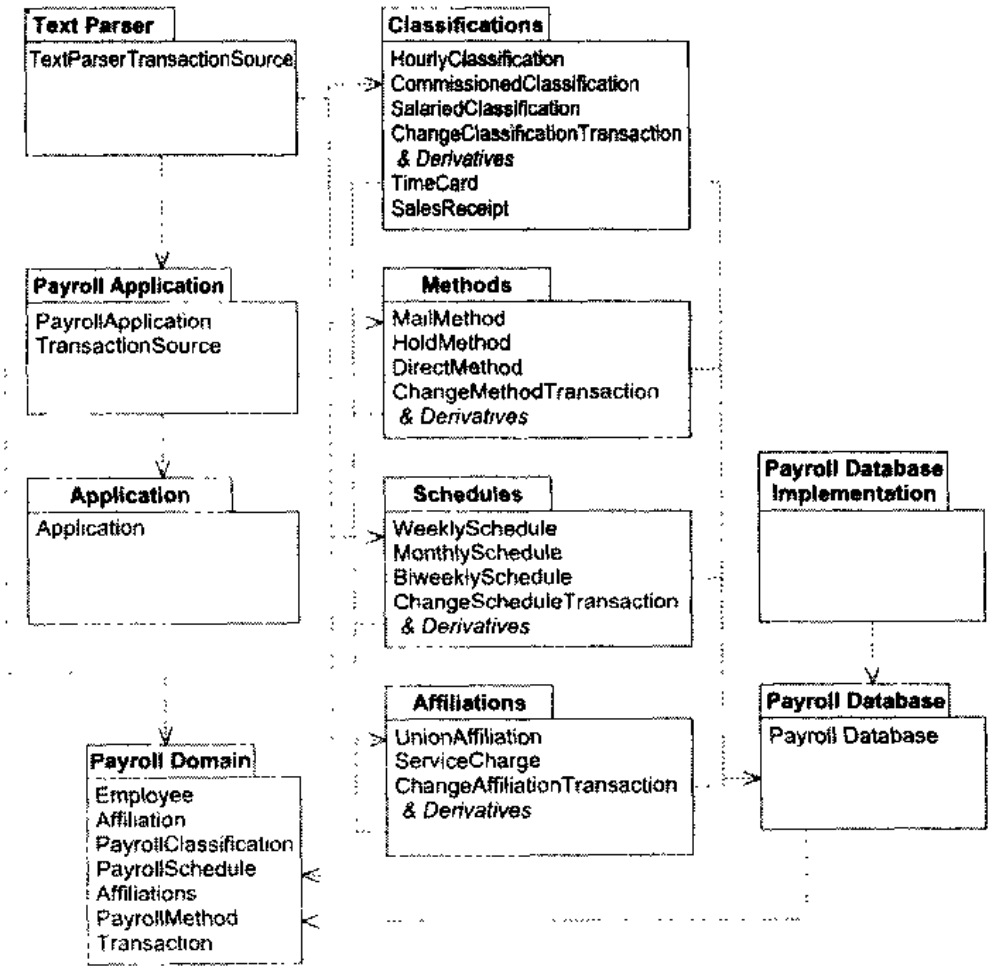


图 22.2　符合封闭性的薪水支付应用程序包层次结构

PayrollDomain 包具有最为显著的通用性和无依赖性性。该包中包含有整个系统的本质部分，却

没有依赖于任何其他的包！请仔细检查这个包。它包含了 Employee、PaymentClassification、PaymentMethod、PaymentSchedule、Affiliation 以及 Transaction。该包中包含了我们模型中所有的主要抽象，但却不依赖于任何其他的包。为什么？因为它包含的所有类几乎都是抽象的。

考虑 Classification 包，它包含了 PaymentClassification 的 3 个派生类。它同样也包含了 ChangeClassificationTransaction 类和它的 3 个派生类，以及 TimeCard 和 SalesReceipt。请注意，这 9 个类的任何变化都被隔离了；除了 TextParser 外，任何其他的包都不会受到影响！Methods 包、Schedules 包以及 Affiliations 包中同样也有这种变化隔离机制。这种隔离相当多。

请注意，大部分的可执行代码都在那些具有很少依赖者或者没有依赖者的包中。因为几乎没有包依赖于它们，所以称它们为不承担责任的。这些包中的代码是非常灵活的；对它们的更改对项目中大部分其他的包不会造成影响。同样请注意，系统中最具通用性的包所包含的可执行代码的数量是最少的。这些包被很多包所依赖，却不依赖于任何其他的包。因为有很多包依赖于它们，所以称它们为承担责任的，并且因为它们不依赖于任何其他包，所以也称它们为无依赖性的。因此，承担责任的代码（也就是说，更改它们会影响到许多其他代码）的数量是非常少的。此外，这些少量的承担责任的代码同样也是无依赖性的，这意味着任何其他的模块都不会引起它的变化。在这个倒置的结构中，底部是高度无依赖性和承担责任的包含通用部分的包，顶部是高度有依赖性和不承担责任的包含细节的包，这种结构是面向对象设计的标志。

我们来对比一下图 22.1 和图 22.2。请注意，图 22.1 中底部的细节是无依赖性并且高度承担责任的。把细节放在这里是错误的！细节应该依赖于系统的主要的构架决策，而不应该被依赖。同样请注意，那些通用的包，也就是定义系统构架的包，却是不承担责任并且高度有依赖性的。因此，定义构架决策的包就依赖于，并且因此受限于包含实现细节的包。这违反了 SAP。如果细节受限于构架的话，就会好一些。

22.3 应用重用发布等价原则（REP）

薪水支付应用程序的哪一部分可以重用呢？如果同一公司的另外一个部门想重用薪水支付系统，但是他们需要一个完全不同的策略集，他们不能重用 Classifications、Methods、Schedules 以及 Affiliations。然而，他们可以重用 PayrollDomain、PayrollApplication、Application、PayrollDatabase，也可能会重用 PDImplementation。另一方面，如果另一个部门想编写一个分析当前雇员数据库的软件，他们可以重用 PayrollDomain、Classifications、Methods、Schedules、Affiliations、PayrollDatabase 以及 PDImplementation。在每种情况下，重用的粒度都是包。

很少会出现只重用包中单一类的情况。原因很简单：一个包中的类应该是内聚的。这意味着它们之间互相依赖，很难轻易、合理地把它们分开。例如，只使用 Employee 类而不使用 PaymentMethod 类是没有意义的。事实上，为了达到这个目的，你必须要修改 Empolyee 类，把 PaymentMethod 类从其中删除。我们当然不想为了支持某种重用而强迫自己去修改要被重用的组件。因此，重用的粒度应该是包。这样，我们在试图把类分组成包时，就有了另外一个可以使用的内聚标准：一个包中的类不仅要一同封闭，而且按照 REP，它们也应该一同重用。

我们再来看一下图 22.1 种最初的包图。那些我们可能想重用包，比如：Transactions 或者 PayrollDatabase，是难以重用的，因为它们具有许多额外的负担。PayrollApplication 包是一个讨厌的依赖者（它几乎依赖于所有的包）。如果我们想创建一个新的薪水支付应用程序，其中要使用一组不同的支付薪水时间表、支付方式、从属关系以及雇员分类策略的话，我们就不能把这个包作为一个整

体重用。相反，我们必须要从 PayrollApplication、Transactions、Methods、Schedules、Classifications 以及 Affiliations 中取出单个的类来重用。以这样的方式去分解包，就破坏了它们的发布结构。我们不能够说 PayrollApplication 的 3.2 版本是可重用的。

图 22.1 违反了 CRP。因此，如果使用了不同包中的可重用片段，那么重用这些片段的人就会面临一个非常困难的管理问题：他不能依赖于我们的发布结构。Methods 的一个新版本会影响到他，因为他重用了 PaymentMethod 类。大部分情况下的更改所针对的都是他没有重用的类，但是他仍然必须要跟踪我们的新版本号并且很可能还得重新编译、重新测试他的代码。

由于很难管理所重用的代码，所以重用者很可能会把可重用的组件做一份拷贝，并使该拷贝独立于我们的组件演化。这不是重用。这两部分代码会变得不同并且需要独立地去支持它们，明显加倍了支持负担。

图 22.2 中的结构中没有这些问题。该结构中的包更容易重用。PayrollDomain 包没有过多的负担。它可以独立于 PaymentMethod、PaymentClassification、PaymentSchedule 等的任何派生类重用。

敏锐的读者会注意到图 22.2 中包图没有完全符合 CRP。特别是 PayrollDomain 中的类没有形成最小的可重用单元。Transaction 类不必和包中其余的类一起重用。我们可以设计出许多只访问 Employee 和它的域，而根本不去使用 Transaction 的应用程序。

这表明要对包图进行更改，如图 22.3 所示。该图中把操作类和它们要操作的元素分离。例如，MethodTransactions 包中的类会操作 Methods 包中的类。我们已经把 Transacion 类移到一个新的名为 TransactonApplication 的包中，该包中还包含有类 TransactionSource 和 TransactionApplication。这 3 个类形成了一个可重用的单元。PayrollApplication 包现在成了一个总的统一体。它包含了主程序以及 TransactionApplication 的一个名为 PayrollAppliaction 的派生类，该类把 TextPArserTransactionSource 绑定到 TransationApplication 上。

这些处理还给设计增加了另外一层抽象。任何从 TransactionSource 获取 Transaction 然后执行它们的应用程序现在都可以重用 TransactionApplication 包。PayrollApplication 包不再是可重用的，因为它极度地依赖于其他包。不过，TransactionApplication 包取代了它的位置，变得更加通用。现在，我们可以独立于任何 Transactions 来重用 PayrollDomain。

这确实改进了项目的可重用性以及可维护性，但是却付出了额外的 5 个包和一个更复杂的依赖关系的代价。这种交换是否合算取决于我们期望重用的类型以及我们期望应用程序演化的速度。如果应用程序保持稳定，并且很少会有客户重用它，那么就不必要做这种修改。另一方面，如果有许多应用程序会重用这个结构，或者我们期望对应用程序进行许多次的更改，那么这个新结构就是优秀的——这是一个需要判断才能做出的决定；并且判断应该基于事实而不是猜测。从简单的包开始，在必要时再去增加包结构是最好的做法。在必要时，总是可以把包结构变得更精细。

22.4　耦合和封装

正如类之间的耦合可以使用 Java 和 C++语言的封装边界来管理一样，包之间的耦合可以使用 UML 语言的引出修饰（export adornment）来管理。

如果一个包中的类被另一个包使用，那么该类必须要被引出。在 UML 中，缺省情况下类都是被引出的，但是我们可以对包进行修饰以指示其中某些类是不应该被引出的。图 22.4 中展示了一个

具有许多类的包 Classifications，其中 PaymentClassification 的 3 个派生类是被引出的，而 TimeCard 和 SalesReceipt 没有被引出。这意味着其他的包不能使用 TimeCard 和 SalesReceipt；它们是 Classifications 包的私有类。

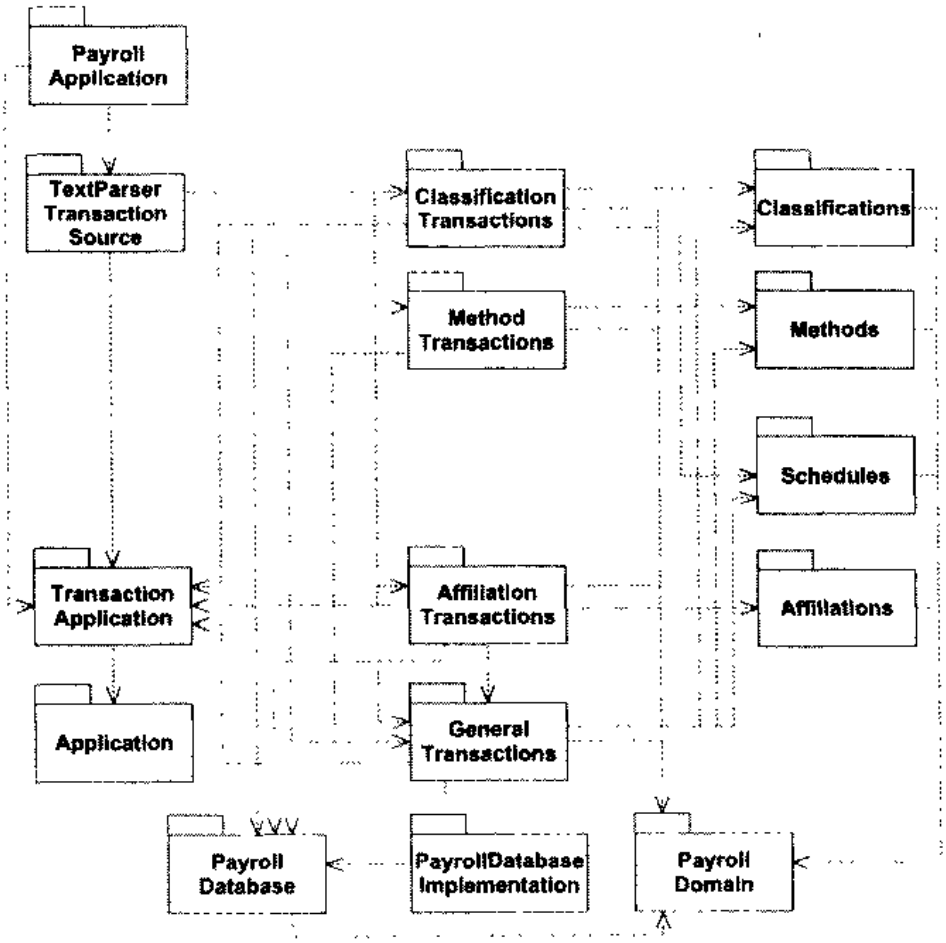


图 22.3　更新后的薪水支付应用程序包图

我们也许想隐藏包中的某些类以避免输入耦合。Classifications 是一个非常细节的类，包含了几种支付策略的实现。为了使该包保持在主序列上，我们想限制它的输入耦合，所以就隐藏了其他包无需知道的类。

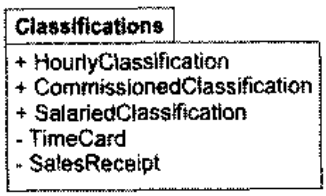


图 22.4　Classifications 包中的私有类

TimeCard 和 SalesReceipt 是非常适合作为私有类的。它们是雇员薪水计算方法的实现细节。我们希望能随时更改这些细节，所以必须要避免任何其他东西依赖于它们的结构。

快速浏览一下图 19.7 至图 19.10 以及程序 19.15（参见第 19 章），可以看出类 TimeCardTransaction 和类 SalesReceiptTransaction 已经依赖于 TimeCard 和 SlesReceipt。不过，这个问题很容易解决，如图 22.5 和图 22.6 所示。

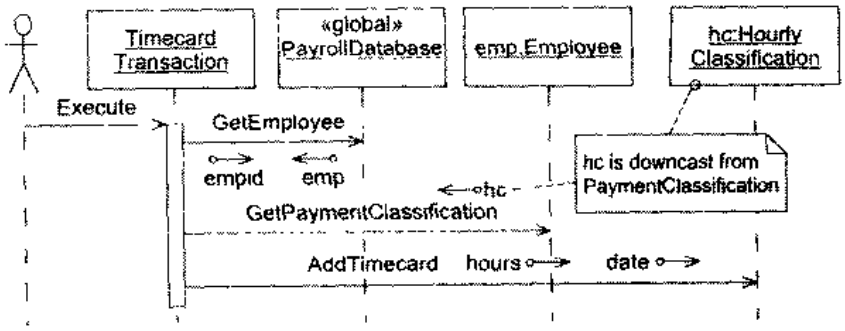


图 22.5　为保护 TimeCard 的私有性对 TimeCardTransaction 做的修正

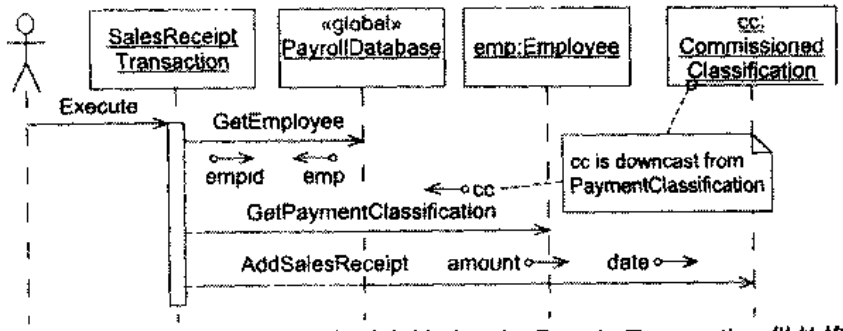


图 22.6　为保护 SalesReceipt 的私有性对 SalesReceiptTransaction 做的修正

22.5　度　　量

按照第 20 章中的论述，我们可以使用几个简单的度量值去量化内聚性、耦合性、稳定性、通用性以及和主序列的一致性等属性。但是为什么要进行量化呢？因为 Tom DeMarco 说过：无法控制的东西就无法管理，无法测量的东西就无法控制。[①]要想成为高效的软件工程师或者软件管理者，必须要能够控制软件开发的实践。如果没有测量它，无论如何都无法控制它。

通过应用下面描述的启发规则，并计算出面向对象设计的一些基本度量值，就可以把这些度量值和软件的真实特性以及开发该软件的团队的真实成效联系起来。搜集的度量值越多，就具有越多的信息，最后就能够施加越多的控制。

下面的度量值已经成功应用到自 1994 年以来的许多项目中。有一些自动工具可以完成这些度量值的计算，手工计算它们也不困难。同样，编写一个简单的 shell、python 或者 ruby 脚本对源文件进行检查并计算这些度量值也不困难。[②]

22.5.1　关系内聚性（*H*）

可以用包中每个类平均的内部关系数目作为包内聚性的一种表示方式。用 R 来表示属于包内部的类关系数目（也就是说，和包外部的类没有联系）。用 N 表示包内类的总数。公式中额外的 1 是为了避免在 N=1 时 H=0。它表示了包和它的所有类之间的关系。

$$H = \frac{R+1}{N}$$

① [DeMacro82], p.3
② 要想得到 shell 脚本的例子，可到 www.objectmentor.com 的自由软件区下载 depend.sh，或到 www.clarkware.com 了解 JDepend。

22.5.2 输入耦合度（C_a）

一个包的输入耦合度可以用对该包的类有依赖的其他包中类的数目来表示。这些依赖关系是类关系，例如：继承和关联。

22.5.3 输出耦合度（C_e）

个包的输出耦合度可以用被该包的类所依赖的其他包中类的数目来表示。像上面一样，这些依赖关系也是类关系。

22.5.4 抽象性或者通用性（A）

一个包的抽象性或者通用性可以用该包中抽象类（或者接口）的数目和该包中类（和接口）的总数的比值来表示。[①]该度量的取值范围是 0 到 1。

$$A = \frac{\text{抽象类的数目}}{\text{类的总数}}$$

22.5.5 不稳定性（I）

一个包的不稳定性可用输出耦合度和总耦合度的比值来表示。该度量的取值范围也是从 0 到 1。

$$I = \frac{C_e}{C_e + C_a}$$

22.5.6 到主序列的距离（D）

理想的主序列是由 $A+I=1$ 所表示的线。D 的计算公式可以计算任何特定的包到主序列的距离。它的范围是从 0～.7[②]，越接近 0 越好。

$$D = \frac{|A + I - 1|}{\sqrt{2}}$$

22.5.7 到主序列的规范化距离（D'）

该度量把度量 D 的取值范围规范化为[0,1]。这样计算和解释起来也许会方便一些。值 0 表示包和主序列是重合的，值 1 表示包到主序列的距离最大。

$$D' = |A + I - 1|$$

22.6 度量薪水支付应用程序

表 22.1 展示了薪水支付模型中包和类之间的对应关系，图 22.7 展示了计算出所有度量值后的薪水支付应用程序包图，表 22.2 展示了单个包的所有度量值。

---

① 你也许会认为把 A 的计算公式改为包中纯虚函数的数目和总成员函数的数目的比值会更好一些。但是，我发现这个计算公式过多地削弱了抽象性度量。即使只有一个纯虚函数也会使类成为抽象的，并且这个抽象的力量要比该类可能具有许多具体函数的事实重要得多，在遵循 DIP 时更是如此。

② 任何包都不可能被绘制在 A/I 坐标图上的单元正方形之外。这是因为 A 和 I 都不会超过 1。主序列从（0，1）到（1，0）把这个正方形等分为两部分。正方形中距离主序列最远的点是两个顶点（0，0）和（1，1）。它们到主序列距离是

$$\frac{\sqrt{2}}{2} = 0.70710678$$

图 22.7 中每个包的依赖关系都有两个数字来修饰。最靠近依赖者包的数字表示该包中对被依赖包中的类有依赖的类的数目。最靠近被依赖包的数字表示该包中被依赖者包所依赖的类的数目。

图 22.7 中的每个包都带有表示它的度量值的修饰。其中许多度量值是鼓舞人心的。例如：PayrollApplication、PayrollDomain 以及 PayrollDatabase 都具有高度的关系内聚性，并且都接近或者位于主序列上。但是，包 Classifications、Methods 以及 Schedules 的关系内聚性却普遍比较糟糕，并且都几乎最大程度地远离主序列。

这些数字表明，我们把类划分成包的方式不是很有效。如果我们找不到改进这些数字的方法，那么开发环境就会易受变化的影响，从而会导致一些不必要的重新发布和重新测试。尤其严重的是，一些低抽象性的包，比如：ClassificatiunTransactions，严重依赖于其他一些低抽象性的包，比如：Classifications。低抽象性的类中包含着大部分的细节代码，因此很可能会改变，从而会迫使重新发布那些依赖于它们的包。因此，包 ClassificationTransactions 将会具有非常高发布率，因为它自己的高更改率以及 Classificaions 的高更改率都会影响到它。我们希望尽可能地限制开发环境对变化的敏感性。

显然，如果总共只有两、三个开发人员，那么可以在“他们的大脑”中管理开发环境，在这种情况下，把包维持在主序列上的需要不是非常强烈。但是，开发人员的数目越多，就越难以保持一个良好的开发环境。此外，即使一次重新测试和重新发布的工作量都远大于获取这些度量值的工作量。[①]因此，计算这些度量值的工作是否是短期的损失或者收益，是需要进行判断才能做出的决定。

表 22.1

包	包中的类		
Affiliations	ServiceCharge	UnionAffiliation	
AffiliationTransactions	ChangeAffiliationTransaction	ChangeUnaffiliatedTransaction	ChangeMemberTransaction
	ServiceChargeTransaction		
Application	Application		
Classifications	CommissionedClassification	HourlyClassification	SalariedClassification
	SalesReceipt	Timecard	
ClassificationTransaction	ChangeClassificationTransaction	ChangeCommissionedTransaction	ChangeHourlyTransaction
	ChangeSalariedTransaction	SalesReceiptTransaction	TimecardTransaction
GeneralTransaction	AddCommissionedEmployee	AddEmployeeTransaction	AddHourlyEmployee
	AddSalariedEmployee	ChangeAddressTransaction	ChangeEmployeeTransaction
	ChangeNameTransaction	DeleteEmployeeTransaction	PaydayTransaction
Methods	DirectMethod	HoldMethod	MailMethod
MethodTransactions	ChangeDirectTransaction	ChangeHoldTransaction	ChangeMailTransaction
	ChangeMethodTransaction		
PayrollApplication	PayrollApplication		
PayrollDatabase	PayrollDatabase		
PayrollDatabaseImplementation	PayrollDatabaseImplementation		
PayrollDomain	Affiliation	Employee	PaymentClassification
	PaymentMethod	PaymentSchedule	
Schedules	BiweeklySchedule	MonthlySchedule	WeeklySchedule
TextParserTransactionSource	TextParserTransactionSource		
TransactionApplication	TransactionApplication	Transaction	TransactionSource

① 我花了约两个小时来手工搜集统计数字并计算薪水支付例子中的度量值。如果使用普通工具的话，几乎不需要花费时间。

表 22.2

包名	N	A	Ca	Ce	R	H	I	A	D	D'
Affiliations	2	0	2	1	1	1	.33	0	.47	.67
AffiliationTransactions	4	1	1	7	2	.75	.88	.25	.09	.12
Application	1	1	1	0	0	1	0	1	0	0
Classifications	5	0	8	3	2	.06	.27	0	.51	.73
ClassificationTransaction	6	1	1	14	5	1	.93	.17	.07	.10
GeneralTransaction	9	2	4	12	5	.67	.75	.22	.02	.03
Methods	3	0	4	1	0	.33	.20	0	.57	.80
MethodsTransaction	4	1	1	6	3	1	.86	.25	.08	.11
PayrollApplication	1	0	0	2	0	1	1	0	0	0
PayrollDatabase	1	1	11	1	0	1	.08	1	.06	.08
PayrollDatabaseImpl...	1	0	0	1	0	1	1	0	0	0
PayrollDomain	5	4	26	0	4	1	0	.80	.14	.20
Schedules	3	0	6	1	0	.33	.14	0	.61	.86
TextParserTransactionSource	1	0	1	20	0	1	.95	0	.03	.05
TransactionApplication	3	3	9	1	2	1	.1	1	.07	.10

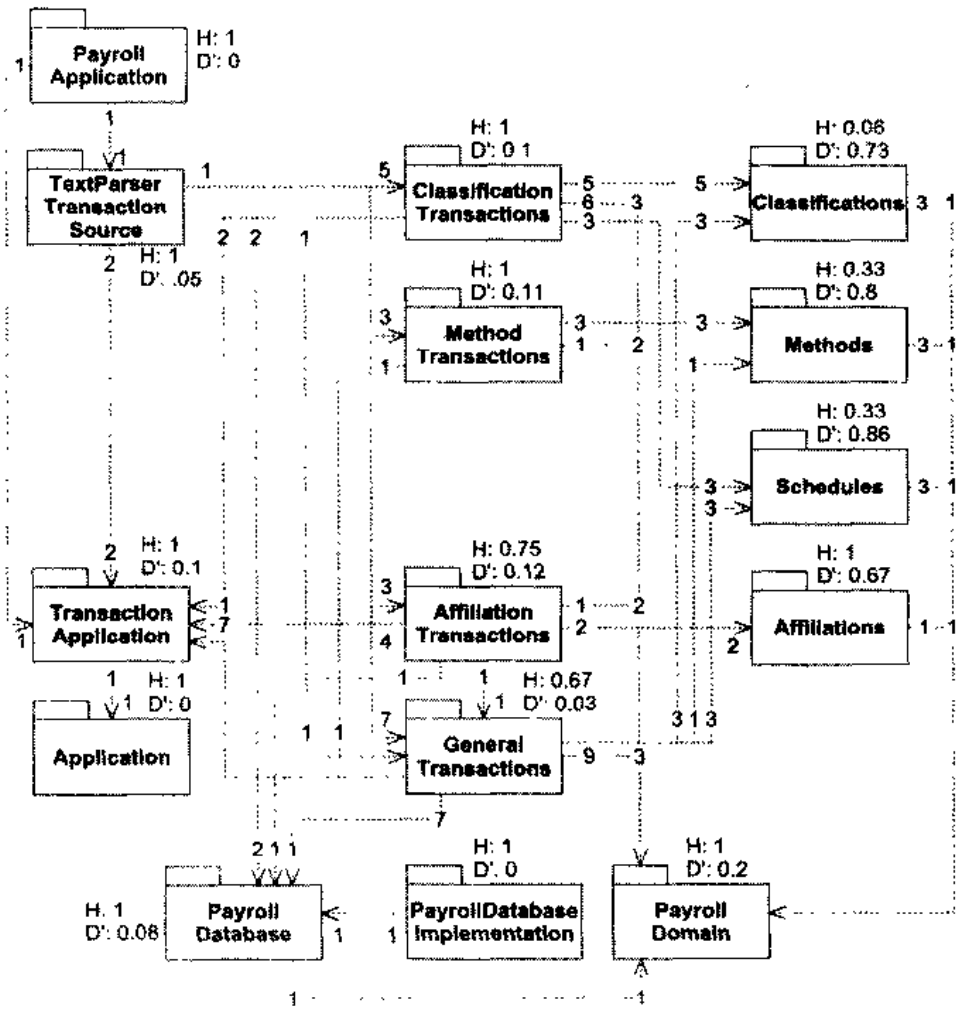


图 22.7　带有度量值的包图

22.7　对象工厂

Classification 和 ClassificationTransactions 之所以被严重地依赖是因为它们中的类必须要被实例化。例如，TextParserTransactionSource 类必须能够创建 AddHourlyEmployeeTransaction 对象；因此，就产生了一个从 TextParserTransactionSource 包到 ClassificationTransactions 包的输入耦合。同样，ChangeHourleTransaction 类必须能够创建 HourlyClassification 对象，所以就产生了一个从 ClassificationTransactions 包到 Classification 包的输入耦合。

对这些包中对象的几乎所有其他的使用方式都是通过抽象接口进行的。如果不需要去创建每个具体对象，那么这些包的输入耦合就不会存在。例如，如果 TestParserTransactionSource 不需要去创建不同的操作，那么它就不会依赖于包含操作实现的 4 个包了。

使用 FACTORY 模式可以显著地缓和这个问题。每个包都提供一个对象工厂，该工厂负责创建该包中所有的公有对象。

22.7.1　TransactionImplementation 包的对象工厂

图 22.8 中展示了如何去构建 TransactionImplementation 包的对象工厂。TransactionFactory 包中包含着抽象基类，这些抽象基类定义了用来描绘具体操作对象构造器的纯虚函数。TransactionImplementation 包中包含着 TransactionFactory 类的具体派生类，并且使用了所有要创建的具体操作对象。

TransactionFactory 类有一个声明为指向 TransactionFactory 的指针的静态成员。这个成员必须要在主程序中被初始化为指向一个具体 TransactionFactoryImplementation 对象的实例。

22.7.2　初始化对象工厂

为了使用对象工厂去创建对象，抽象对象工厂的静态成员必须要被初始化为指向适当的具体对象工厂。这项工作必须要在任何使用者试图去使用对象工厂前完成。通常主程序最适合完成这项工作，这就意味着主程序要依赖于所有的对象工厂以及所有的具体包。因此，每个具体包都至少具有一个来自主程序的输入依赖。这会迫使具体包稍微偏离主序列一点，但这是无法避免的。[①]这意味着每当对任何具体包进行更改时，就必须要重新发布主程序。当然，对于每个更改无论如何都应该重新发布主程序，因为不管怎样都要对它进行测试。

图 22.9 和图 22.10 展示了主程序和对象工厂间关系的静态和动态结构。

22.7.3　重新思考内聚的边界

最初，我们在图 22.1 中把 Classificatons、Methods、Schedules 以及 Affiliations 分开。当时，这看起来像是一个合理划分方法。毕竟，其他的使用者也许希望在重用支付薪水时间表类时不要带上 Affiliation 类。在把操作类分割到自己的包中，创建了一个双重层次结构后，仍然维持着这种划分方法。也许，这样做有些过分了。图 22.7 中的包结构非常的复杂。

复杂的包图使手工管理包的发布变得困难。虽然使用自动的项目计划工具可以很好地处理复杂的包图，但是我们中的大多数都不具有这种奢侈品。因此，需要尽可能地保持包图实用、简单。

---

① 在实际的做法中，我通常会忽略来自主程序的耦合。

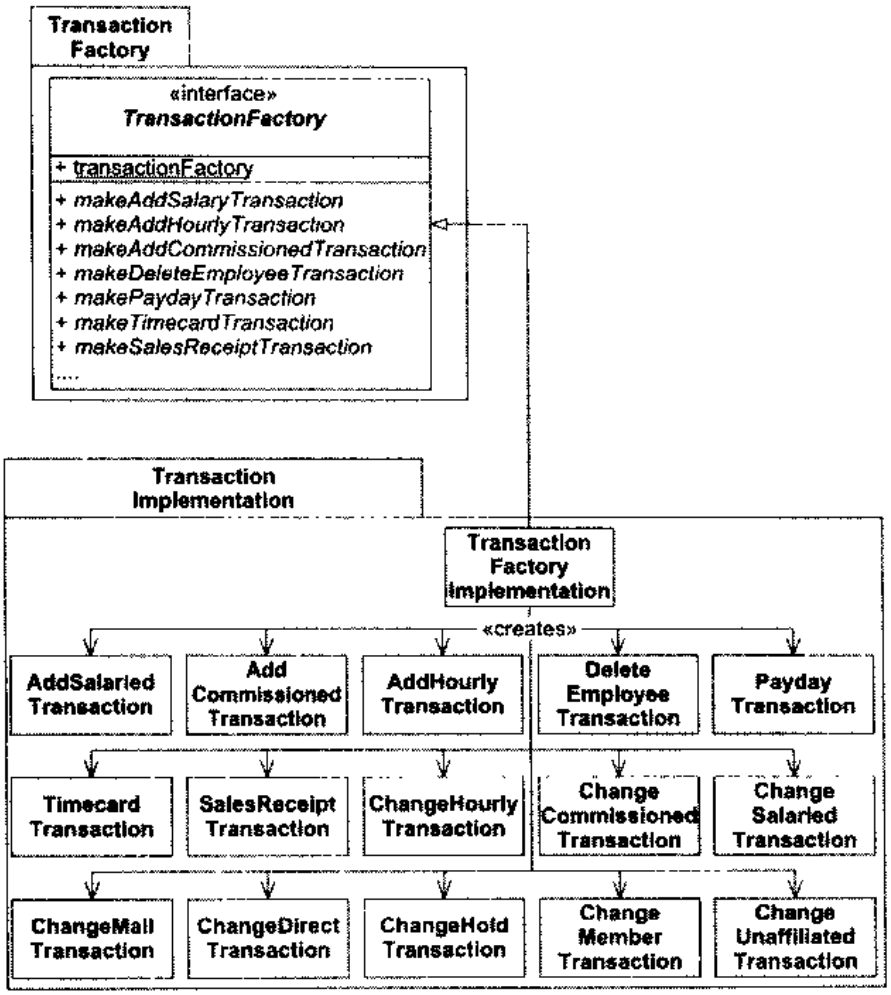


图 22.8　Transaction 的对象工厂

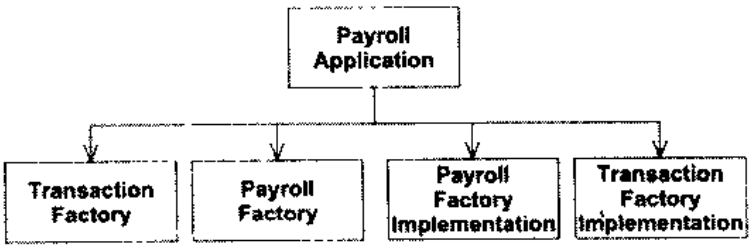


图 22.9　主程序和对象工厂的静态结构

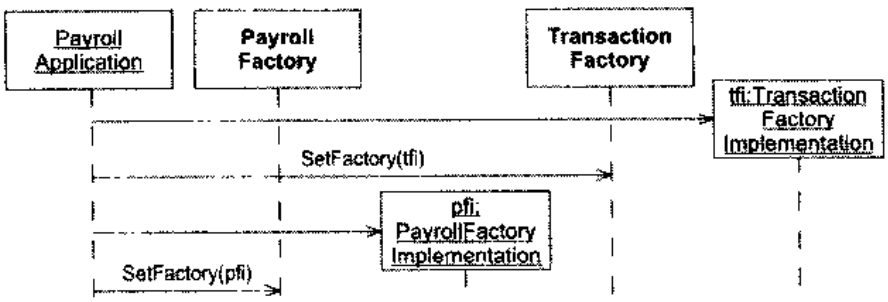
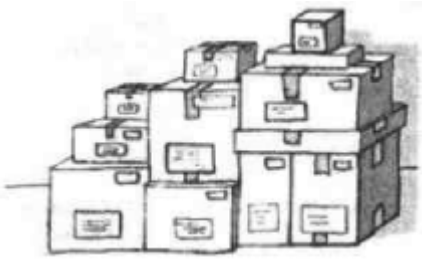


图 22.10　主程序和对象工厂的动态结构

在我看来，基于操作的划分要比基于功能的划分重要。所以，我们把所有的操作并入一个单一的包 TransactionImplementation 中（图 22.11）。我们同样也把 Classifications、Schedules、Methods 以及 Affilliations 包并入一个单一的包 PayrollImplementation 中。



22.8　最终的包结构

表 22.3 中展示了最终的类到包的分配结果。表 22.4 中包含着度量数据表。图 22.11 中展示了最终的包结构，该结构中使用了对象工厂使具体包位于主序列的附近。

该图中的度量值是令人振奋的。其中具有高度的关系内聚性（部分原因是具体对象工厂和它们创建的对象间的关系），并且没有严重的和主序列的背离情况。因此，包之间的耦合合乎一个良好开发环境的要求。抽象的包是封闭的、可重用的，并且被严重依赖，同时它们很少依赖于其他包。具体的包被基于可重用性分离开来，它们严重地依赖于抽象包，并且没有严重地被自己依赖。

表 22.3

包	包中的类		
AbstractTransactions	AddEmployeeTransaction	ChangeAffiliationTransaction	ChangeEmployeeTransaction
	ChangeClassificationTransaction	ChangeMethodTransaction	
Application	Application		
PayrollApplication	PayrollApplication		
PayrollDatabase	PayrollDatabase		
PayrollDatabaseImplementation	PayrollDatabaseImplementation		
PayrollDomain	Affiliation	Employee	PaymentClassification
	PaymentMethod	PaymentSchedule	
PayrollFactory	PayrollFactory		
PayrollImplementation	BiweeklySchedule	CommissionedClassification	DirectMethod
	HoldMethod	HourlyClassification	MailMethod
	MonthlySchedule	PayrollFactoryImplementation	SalariedClassification
	SalesReceipt	ServiceCharge	Timecard
	UnionAffiliation	WeeklySchedule	
TextParserTransactionSource	TextParserTransactionSource		
TransactionApplication	Transaction	TransactionApplication	TransactionSource
TransactionFactory	TransactionFactory		
TransactionImplementation	AddCommissionedEmployee	AddHourlyEmployee	AddSalariedEmployee
	ChangeAddressTransaction	ChangeCommissionedTransaction	ChangeDirectTransaction
	ChangeHoldTransaction	ChangeHourlyTransaction	ChangeMailTransaction
	ChangeMemberTransaction	ChangeNameTransaction	ChangeSalariedTransaction
	ChangeUnaffiliatedTransaction	DeleteEmployee	PaydayTransaction
	SalesReceiptTransaction	ServiceChargeTransaction	TimecardTransaction
	TransactionFactoryImplementation		

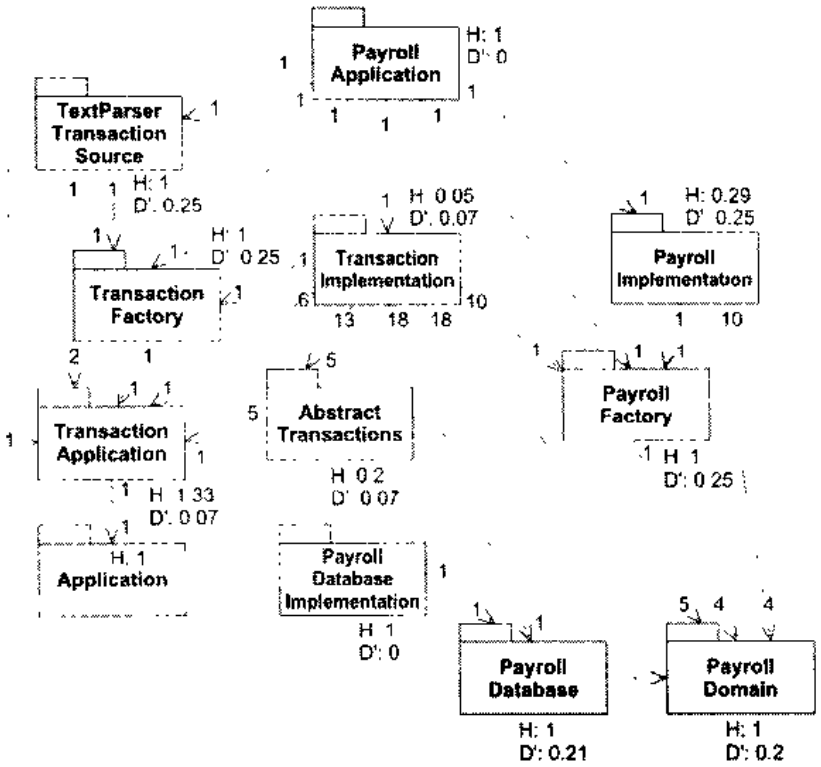


图 22.11　薪水支付应用程序的最终包结构

表 22.4

包名	N	A	Ca	Ce	R	H	I	A	D	D'
AbstractTransaction	5	5	13	1	0	.20	.07	1	.05	.07
Application	1	1	1	0	0	1	0	1	0	0
PayrollApplication	1	0	0	5	0	1	1	0	0	0
PayrollDatabase	1	1	19	5	0	1	.21	1	.15	.21
PayrollDatabaseImpl...	1	0	0	1	0	1	1	0	0	0
PayrollDomain	5	4	30	0	4	1	0	.80	.14	.20
PayrollFactory	1	1	12	4	0	1	.25	1	.18	.25
PayrollImplementation	14	0	1	5	3	.29	.83	0	.12	.17
TextParserTransactionSource	1	0	1	3	0	1	.75	0	.18	.25
TransactionApplication	3	3	14	1	3	1.33	.07	1	.05	.07
TransactionFactory	1	1	3	1	0	1	.25	1	.18	.25
TransactionImplementation	19	0	1	14	0	.05	.93	0	.05	.07

22.9　结　　论

是否需要对包结构进行管理，取决于程序的规模以及开发团队的规模。即使小的团队，也需要对源代码进行划分，以便于开发人员间可以互不干扰。如果没有某种形式的划分结构，大程序就会变成晦涩难懂的源文件堆积。

参考文献

1. Benjamin/Cummings. *Object-Oriented Analysis and Design with Applications*, 2d ed.,1994
2. DeMacro,Tom. *Controlling Software Projects*. Yourdon Press, 1982.

第V部分　气象站案例研究

在接下来的几章中，我们会对一个简单的天气监控系统案例进行深入的研究。虽然这个案例是虚构的，但是它的构建过程却非常贴近真实情况。我们会碰到时间压力、遗留代码、贫乏以及变化的规格说明、未尝试过的新技术等一系列问题。我们的目标是展示一下如何在实际的软件开发中使用我们学过的原则、模式以及实践。

像前面一样，在探讨气象站应用的开发中，我们会遇到一些有用的设计模式。在该案例研究的预备章节中，会对这些模式进行描述。

第 23 章　COMPOSITE 模式

COMPOSITE 模式是一个非常简单但具有深刻内涵的模式。图 23.1 中展示了 COMPOSITE 模式的基本结构。图中是一个形状类层次结构。基类 Shape 有两个派生类：Circle 和 Square。第 3 个派生类是一个组合体。CompositeShape 持有一个含有多个 Shape 实例的列表。当调用 CompositeShape 的 draw() 方法时，它就把这个方法委托给列表中的每一个 Shape 实例。

因此，对系统来说，一个 CompositeShape 实例就像是一个单一的 Shape。可以把它传递给任何使用 Shape 的函数或者对象，并且它表现得就像是一个 Shape。不过，实际上它只是一组 Shape 实例的代理（proxy）。[①]

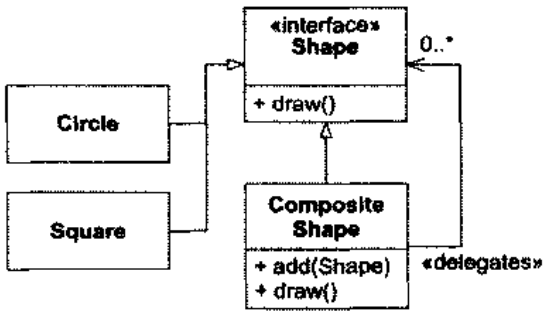


图 23.1　Composite 模式

程序 23.1 和程序 23.2 展示了 CompositeShape 的一个可能实现。

程序 23.1　Shape.java

```
public interface Shape
{
  public void draw();
}
```

程序 23.2　CompositeShape.java

```
import java.util.Vector;

public class CompositeShape implements Shape
{
```

① 请注意在结构上和 PROXY 模式的相似性。

```
  private Vector itsShapes = new Vector();
  public void add(Shape s)
  {
    itsShapes.add(s);
  }

  public void draw()
  {
    for (int i = 0; i < itsShapes.size(); i++)
    {
      Shape shape = (Shape) itsShapes.elementAt(i);
      shape.draw();
    }
  }
}
```

23.1 示例：组合命令

请回顾一下我们在第 13 章中所讨论的 Sensor 对象和 Command 对象。图 13.3 中展示了一个使用 Command 类的 Sensor 类。当 Sensor 检测到对它的刺激时，就调用 Command 的 do()方法。

在那次讨论中，我没有提及一个 Sensor 必须执行多个 Command 的情况，这种情况是经常发生的。例如，当纸到达传送路径上的一个特定点时，就会启动一个光学传感器。接着，这个传感器停止一个发动机，启动另一个，然后启用一个特定的离合器。

起初，我认为这意味着每个 Sensor 类必须要维持一个 Command 对象的列表（参见图 23.2）。然而，我们很快意识到每当 Sensor 需要执行多个 Command 时，它总是以一致的方式去对待这些 Command 对象。也就是说，它只是遍历列表并调用每个 Command 对象 do()方法。这种情况最适合使用 COMPOSITE 模式。

这样，我们就不去改动 Sensor 类，而创建一个如图 23.3 所示的 CompositeCommand 类。

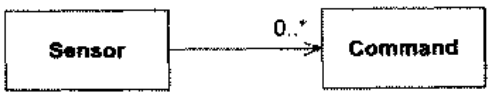


图 23.2 包含许多 Command 的 Sensor

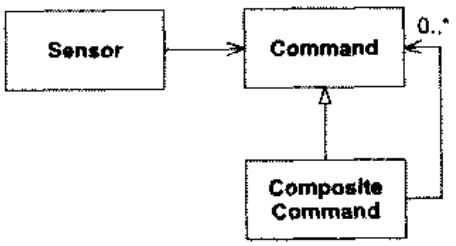


图 23.3 组合命令

这意味着我们无需去更改 Sensor 以及 Command。我们可以在不改变 Sensor 类和 Command 类的情况下向 Sensor 对象中增加多个 Command 对象。这里应用了 OCP 原则。

23.2 多重性还是非多重性

这导致了一个有趣的问题。我们可以在不修改 Sensor 的情况下，就能够使其表现得好像包含了多个 Command 一样。在通常的软件设计中，肯定会有许多与此类似的其他情形。你肯定也多次碰到过这样的情况，其中使用 COMPOSITE 模式要胜于构建一个对象列表或者向量。

我换一种方式来说明这个问题。Sensor 和 Command 之间的关联是一对一的。我们非常想把该关

联更改为一对多的。但是，我们找到了一种无需一对多的关系即可获得一对多的行为的替代方法。一对一的关系要比一对多的关系更容易理解、编码以及维护；所以，这种设计权衡显然是正确的。如果使用了 COMPOSITE 模式，在你当前的项目中，有多少一对多的关系可以转变成一对一的呢？

当然，使用 COMPOSITE 模式并不能把所有的一对多关系都转变成一对一关系。只有那些以一致的方式对待列表中的每个对象的情况才具备转换的可能性。例如，如果你持有一个雇员对象列表，并在列表中搜寻发薪日在今天的雇员，你或许就不应该使用 COMPOSITE 模式，因为你不是以一致的方式去对待所有的雇员对象。

尽管如此，还是有相当一部分一对多的关系适合于使用 COMPOSITE 模式。并且好处还是相当大的。列表管理和遍历的代码只是在组合类中出现一次，而不是在每个客户代码中重复出现。

第 24 章　OBSERVER 模式——回归为模式

本章有一个特别的目的。其中，我将会讲解 OBSERVER 模式[①]，但是这只是一个次要的目的。本章的主要目的是向你展示一下，代码和设计是怎样演化成使用模式的。

在前面的章节中，我们已经使用了许多模式。我们常常是直接使用它们，而没有展示代码是怎样演化成使用模式的。这可能会让你认为模式就是一些以完整的形式插入到代码和设计中的东西。这不是我所建议的使用方式。我更喜欢朝着正在编写的代码需要的方向去演化代码。当我去重构代码以解决耦合性、简单性以及表达性的问题时，可能会发现代码已经接近于一个特定的模式了。此时，我把类和变量的名字改成使用模式的名字，并且把代码的结构更改为以更正规的形式使用模式。这样，代码就回归为模式。

本章首先提出一个简单的问题，然后展示设计和代码是如何演化并最终解决这个问题的。演化的最终结果是 OBSERVER 模式。在演化的每一个阶段，我都会先描述要解决的问题，然后展示解决这些问题的步骤。

24.1　数字时钟

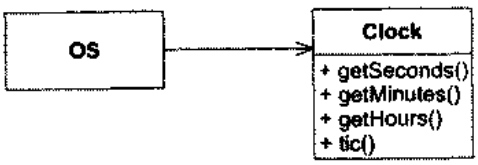


图 24.1　时钟

我们有一个时钟对象。该对象捕获来自操作系统的毫秒中断（即时钟滴答），并把它们转换成时间。该对象知道如何从微秒数计算出秒数，如何从秒数计算出分钟数，如何从分钟数计算出小时数，如何从小时数计算出天数等等。它知道每个月有多少天，以及每年有几个月。它知道有关闰年的所有信息，并且可以判断出什么时候是闰年，而什么时候不是闰年。它也知道时间的概念（参见图 24.1）。

我们想创建一个数字时钟，可以把它摆放在桌面上，并且连续地显示时间。最简单的实现方法

① [GOF95]，第 293 页。

是什么呢？我们可以编写下面的代码：

```
public void DisplayTime
{
    while (1)
    {
    int sec = clock.getSeconds();
    int min = clock.getMinutes();
    int hour = clock.getHours();
    showTime(hour, min, sec);
    }
}
```

显然，这不是最好的方法。为了重复地显示时间，它消耗了所有可用的 CPU 周期。其中，大部分的显示都是多余的，因为时间并没有变化。也许，这个解决方案完全可以应用于数字手表或者数字挂钟中，因为在这些系统中，节省 CPU 周期不是非常重要。不过，我们不希望这个独占 CPU 的家伙运行在我们的桌面上。

最根本的问题是如何高效地把数据从 Clock 传给 DigitalClock。假设 Clock 对象和 DigitalClock 对象都存在。我所关心的是如何把它们连接起来。要测试该连接，只要证实从 Clock 取出的数据和发送给 DigitalClock 的数据是相同的即可。

有一个简单地实现该测试的方法：首先创建两个接口，一个充当 Clock，另一个充当 DigitalClock，然后编写实现这两个接口的特殊测试对象并核实它们之间的连接是按期望的方式工作。

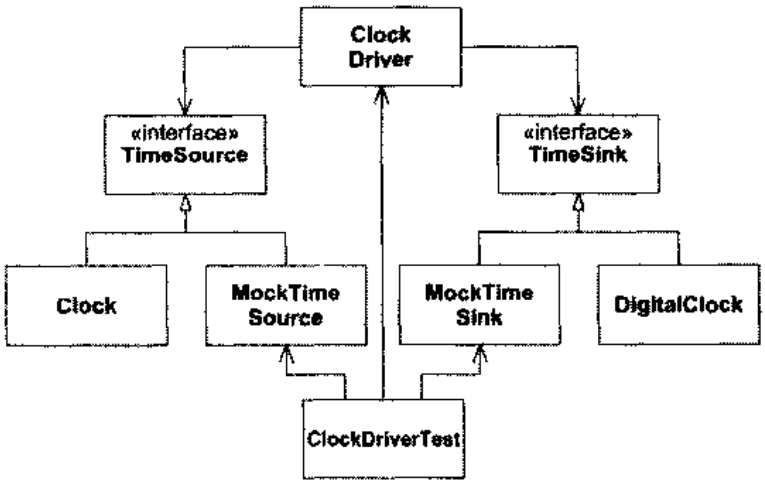


图 24.2　测试数字时钟

ClockDriverTest 对象通过 TimeSource 和 TimesSink 接口把 ClockDriver 和两个模仿（mock）对象连接起来。接着，它会去检查每个模仿对象以确保 ClockDriver 已经把时间数据从源传到了接收端。如果有必要的话，ClockDriverTest 也要保证效率得到了提高。

完全是出于测试的考虑，结果却向设计中增加了接口，我认为这很有趣。为了测试一个模块，你必须能够把它与系统中的其他模块隔离开，就像我们把 ClockDriver 与 Clock、DigitalClock 隔离开一样。优先考虑测试有助于把设计中的耦合减至最少。

那么，ClockDriver 如何工作呢？显然，为了高效，ClockDriver 必须要检测 TimeSource 对象中的时间何时发生改变。只有在时间发生改变的那一刻，它才应该把时间数据移至 TimeSink 对象中。ClockDriver 怎样才能知道时间在何时发生了改变呢？它可以轮询 TimeSource，但是这只会再现独占

CPU 的问题。

让 ClockDriver 知道何时时间发生变化的最简单的方法是让 Clock 对象告诉它。我们可以通过 TimeSource 接口把 ClockDriver 传给 Clock，这样，当时间变化时，Clock 对象就可以更新 ClockDriver。接着，ClockDriver 再把时间设置到 ClockSink 中（参见图 24.3）。

请注意从 TimeSource 到 ClockDriver 的依赖关系。产生这个依赖关系的原因是因为 setDriver 方法的参数是一个 ClockDriver 对象。我对此不是很满意，因为这意味着在任何情况下 TimeSource 对象都必须使用 ClockDriver 对象。不过，在该程序能够工作之前，我不会进行任何有关依赖关系的处理。

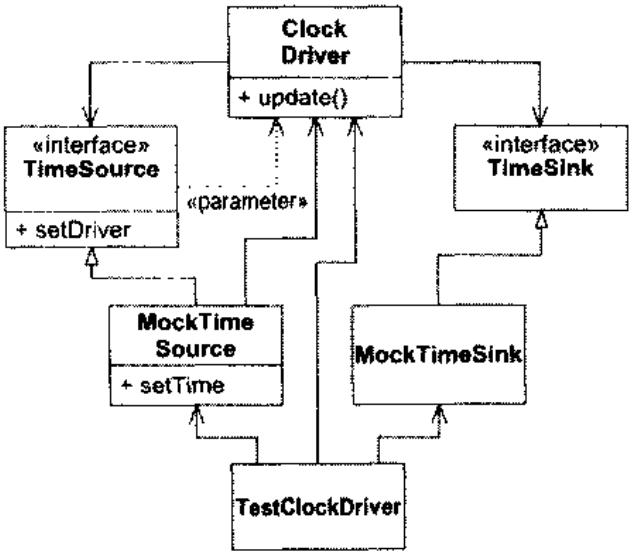


图 24.3　让 TimeSource 去更新 ClockDriver

程序 24.1 展示了 ClockDriver 的测试用例。请注意，它创建了一个 ClockDriver 对象并在其上绑定了 MockTimeSource 和 MockTimeSink。接着，它在 source 对象中设置了时间，并期望这个时间能够神奇地到达 sink 对象。程序 24.2 至 24.6 中为其余的代码。

程序 24.1　ClockDriverTest.java

```
import junit.framework.*;

public class ClockDriverTest extends TestCase
{
    public ClockDriverTest(String name)
    {
        super(name);
    }

    public void testTimeChange()
    {
        MockTimeSource source = new MockTimeSource();
        MockTimeSink sink = new MockTimeSink();
        ClockDriver driver = new ClockDriver(source, sink);
        source.setTime(3,4,5);
        assertEquals(3, sink.getHours());
        assertEquals(4, sink.getMinutes());
        assertEquals(5, sink.getSeconds());

        source.setTime(7,8,9);
        assertEquals(7, sink.getHours());
        assertEquals(8, sink.getMinutes());
        assertEquals(9, sink.getSeconds());
    }
}
```

程序 24.2　TimeSource.java

```
public interface TimeSource
{
```

```
    public void setDriver(ClockDriver driver);
}
```

程序 24.3 TimeSink.java

```
public interface TimeSink
{
    public void setTime(int hours, int minutes, int seconds);
}
```

程序 24.4 ClockDriver.java

```
public class ClockDriver
{
    private TimeSink itsSink;

    public ClockDriver(TimeSource source, TimeSink sink)
    {
        source.setDriver(this);
        itsSink = sink;
    }

    public void update(int hours, int minutes, int seconds)
    {
        itsSink.setTime(hours, minutes, seconds);
    }
}
```

程序 24.5 MockTimeSource.java

```
public class MockTimeSource implements TimeSource
{
    private ClockDriver itsDriver;

    public void setTime(int hours, int minutes, int seconds)
    {
        itsDriver.update(hours, minutes, seconds);
    }

    public void setDriver(ClockDriver driver)
    {
        itsDriver = driver;
    }
}
```

程序 24.6 MockTimeSink.java

```
public class MockTimeSink implements TimeSink
{
    private int itsHours;
    private int itsMinutes;
    private int itsSeconds;

    public int getSeconds()
    {
        return itsSeconds;
    }

    public int getMinutes()
    {
        return itsMinutes;
    }
```

```
    public int getHours()
    {
        return itsHours;
    }

    public void setTime(int hours, int minutes, int seconds)
    {
        itsHours = hours;
        itsMinutes = minutes;
        itsSeconds = seconds;
    }
}
```

好极了，既然测试通过了，我就可以考虑去整理它了。我不喜欢从 TimeSource 到 ClockDriver 的依赖关系，因为我希望 TimeSource 接口可以被任何对象使用，而不仅仅是 ClockDriver 对象。通过创建一个 TimeSource 可以使用，而 ClockDriver 可以继承的接口，就可以修正这个问题。我们称这个接口为 ClockObserver。请参见程序 24.7 到 24.10。其中，粗体的部分是更改过的代码。

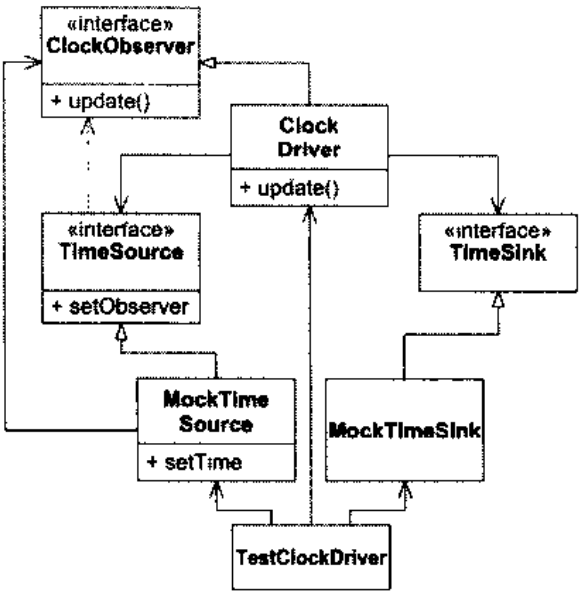


图 24.4　解除 TimeSource 对 ClockDriver 的依赖

程序 24.7　ClockObserver.java

```
public interface ClockObserver
{
    public void update(int hours, int minutes, int seconds)
}
```

程序 24.8　ClockDriver.java

```
public class ClockDriver implements ClockObserver
{
    private TimeSink itsSink;

    public ClockDriver(TimeSource source, TimeSink sink)
    {
        source.setObserver(this);
        itsSink = sink;
    }

    public void update(int hours, int minutes, int seconds)
    {
        itsSink.setTime(hours, minutes, seconds);
    }
}
```

程序 24.9　TimeSource.java

```
public interface TimeSource
{
    public void setObserver(ClockObserver observer);
}
```

程序 24.10　MockTimeSource.java

```
public class MockTimeSource implements TimeSource
{
    private ClockObserver itsObserver;
```

```
    public void setTime(int hours, int minutes, int seconds)
    {
        itsObserver.update(hours, minutes, seconds);
    }

    public void setObserver(ClockObserver observer)
    {
        itsObserver = observer;
    }
}
```

这就好多了。现在任何对象都可以使用 TimeSource。它们只要实现 ClockObserver 接口，并把自己作为参数调用 SetObserver 方法即可。

我想让多个 TimeSink 都能够获得时间。有人可能要实现数字时钟。另外有人可能想使用所提供的时间实现一个提醒服务。还有人可能想启动每晚备份功能。简而言之，我希望一个单一的 TimeSource 对象能够为多个 TimeSink 对象提供时间。

所以我修改了 ClockDriver 的构造函数，使之只具有一个参数 TimeSource，然后增加一个方法 addTimeSink，该方法允许你在任何需要的时候都可以增加 TimeSink 实例。

这种做法中有一点我不喜欢，那就是现在出现了两个间接关系。我必须要调用 setObserver 来告诉 TimeSource 谁是 ClockServer，同样还必须要告诉 ClockDriver 谁是 TimeSink 实例。这个双重间接关系真的是必需的吗？

仔细检查了 ClockObserver 和 TimeSink 后，我发现它们都有 setTime 方法。TimeSink 好像也可以实现 ClockObserver。如果这样做了，那么测试程序就可以创建一个 MockTimeSink 并把它作为参数去调用 TimeSource 的 setObserver。这样，就可以完全去掉 ClockDriver（和 TimeSink）了！程序 24.11 展示了对 ClockDriverTest 的更改。

程序 24.11 ClockDriverTest.java

```
import junit.framework.*;

public class ClockDriverTest extends TestCase
{
    public ClockDriverTest(String name)
    {
        super(name);
    }

    public void testTimeChange()
    {
        MockTimeSource source = new MockTimeSource();
        MockTimeSink sink = new MockTimeSink();
        Source.setObserver(sink);

        source.setTime(3,4,5);
        assertEquals(3, sink.getHours());
        assertEquals(4, sink.getMinutes());
        assertEquals(5, sink.getSeconds());

        source.setTime(7,8,9);
        assertEquals(7, sink.getHours());
        assertEquals(8, sink.getMinutes());
```

```
        assertEquals(9, sink.getSeconds());
    }
}
```

这意味着 MockTimeSink 应该实现 ClockObserver 而不是 TimeSink。请参见程序 24.12。这些更改很有效。为什么一开始我会认为需要一个 ClockDriver 呢？图 24.5 中展示了相应的 UML 图。

程序 24.12　MockTimeSink.java

```
public class MockTimeSink implements ClockObserver
{
    private int itsHours;
    private int itsMinutes;
    private int itsSeconds;

    public int getSeconds()
    {
        return itsSeconds;
    }

    public int getMinutes()
    {
        return itsMinutes;
    }

    public int getHours()
    {
        return itsHours;
    }

    public void update(int hours, int minutes, int seconds)
    {
        itsHours = hours;
        itsMinutes = minutes;
        itsSeconds = seconds;
    }
}
```

显然，这简单多了。

好，现在我们把 setObserver 函数改成 registerObserver，并确保所有注册的 ClockObserver 实例都被保存在一个列表中并被适时更新，这样就可以处理多个 TimeSink 对象了。这需要对测试程序做另外的更改。程序 24.13 中展示了这些更改。此外，我还对测试程序做了少许重构以使它更小、更易读一些。

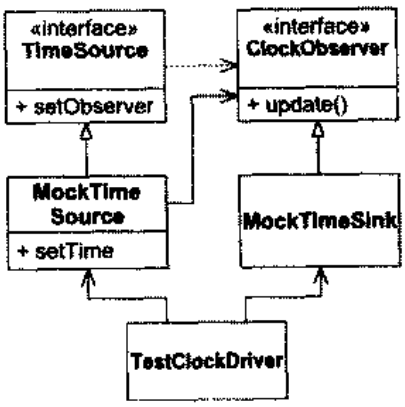


图 24.5　去除了 ClockDriver 和 TimeSink

程序 24.13　ClockDriverTest.java

```
import junit.framework.*;

public class ClockDriverTest extends TestCase
{
    private MockTimeSource source;
    private MOckTimeSink sink;

    public ClockDriverTest(String name)
    {
        super(name);
```

```
    }

    public void setup()
    {
        source = new MockTimeSource();
        sink new MockTimeSink();
        source.registerObserver(sink);
    }

    private void assertSinkEquals(
            MockTimeSink sink, int hours, int minutes, int seconds)
    {
        assertEquals(hours, sink.getHours());
        assertEquals(minutes, sink.getMinutes());
        assertEquals(seconds, sink.getSeconds());
    }

    public void testTimeChange()
    {
        source.setTime(3,4,5);
        assertSinkEquals(sink, 3,4,5);

        source.setTime(7,8,9);
        assertSinkEquals(sink, 7,8,9);
    }

    public void testMultipleSinks()
    {
      MockTimeSink sink2 =
           new MockTimeSink();
      source.registerObserver(sink2);

      source.setTime(12,13,14);
      assertSinkEquals(sink, 12,13,14);
      assertSinkEquals(sink2, 12,13,14);
    }
}
```

要通过这个测试，只需对程序做非常简单的更改。我们修改了 MockTimeSource，让它把所有已经注册的观察者（observer）保存在一个 Vector 中。这样，当时间变化时，我们就遍历 Vector 并且调用所有已注册的 ClockObserver 对象的 update 方法。程序 24.14 和程序 24.15 展示所做的更改，图 24.6 展示相应的 UML 图。

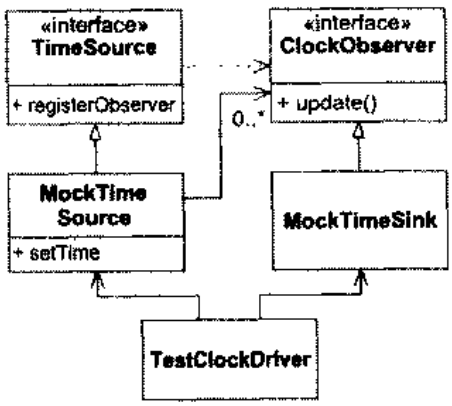


图 24.6　处理多个 TimeSink 对象

程序 24.14　TimeSource.java

```
public interface TimeSource
{
    public void registerObserver(ClockObserver observer);
}
```

程序 24.15　MockTimeSource.java

```
import java.util.*;

public class MockTimeSource implements TimeSource
{
    private Vector itsObservers = new Vector();
```

```
    public void setTime(int hours, int minutes, int seconds)
    {
        Iterator i = itsObservers.iterator();
        observer.update(hours, minutes, seconds);
    }

    public void registerObserver(ClockObserver observer)
    {
        itsObservers.add(observer);
    }
}
```

这非常不错，不过有一点我不喜欢，那就是 MockTimeSource 必须要处理注册和更新。这意味着 Clock 以及每一个 TimeSource 的其他派生类都必需重复注册和更新部分的代码。我认为 Clock 不应该处理注册和更新。此外，我也不喜欢出现代码重复。所以，我想把所有的注册和更新逻辑移到 TimeSource 中。当然，这意味着 TimeSource 要从接口变为类。这同样也意味着 MockTimeSource 会缩小到几乎没有。程序 24.16 和程序 24.17 以及图 24.7 展示了所做的更改。

程序 24.16 TimeSource.java

```
import java.util.*;

public class TimeSource
{
    private Vector itsObservers = new Vector();

    protected void notify(int hours, int minutes, int seconds)
    {
        Iterator i = itsObservers.iterator();
        while (i.hasNext())
        {
            ClockObserver observer = (ClockObserver) i.next();
            observer.update(hours, minutes, seconds);
        }
    }

    public void registerObserver(ClockObserver observer)
    {
        itsObservers.add(observer);
    }
}
```

程序 24.17 MockTimeSource.java

```
public class MockTimeSource extends TimeSource
{
    public void setTime(int hours, int minutes, int seconds)
    {
        notify(hours, minutes, seconds);
    }
}
```

这相当棒。现在，任何类都可以从 TimeSource 派生。它们只要调用 notify 方法就可以更新观察者。但是其中仍有一些我不喜欢的东西。MockTimeSource 直接从 TimeSource 继承。这意味着 Clock 也必须从 TimeSource 派生。Clock 为什么要依赖于注册和更新逻辑呢？Clock 只是一个知道时间的类。让它依赖于 TimeSource 似乎是必要的，但不是所希望的。

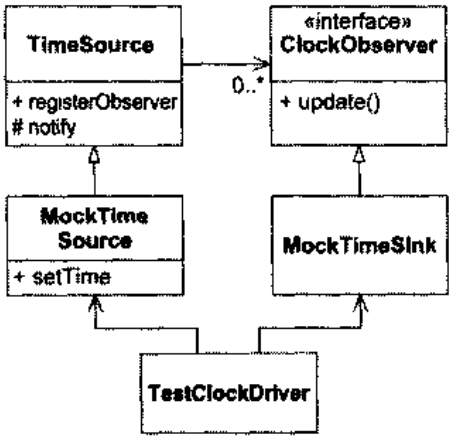


图24.7 把注册和更新逻辑移到TimeSource

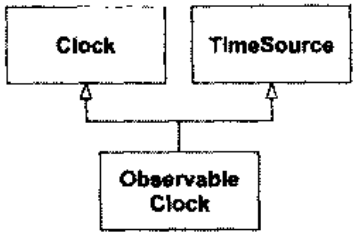


图24.8 用C++的多重继承分离Clock和TimeSource

我知道在 C++中如何解决这个问题。我会创建一个 TimeSource 和 Clock 的共同子类 ObservableClock。我会覆写（override）ObservableClock 的 tic 和 setTime 方法，让它们去调用 Clock 的 tic 或者 setTime 方法，然后再调用 TimeSource 的 notify 方法。请参见程序 24.18 和图 24.8。

程序 24.18 ObservableClock.cc (C++)

```
class ObservableClock: public Clock, public TimeSource
{
    public:
        virtual void tic()
        {
            Clock::tic();
            TimeSource::notify(getHours(), getMinutes(), getSeconds());
        }

        virtual void setTime(int hours, int minutes, int seconds)
        {
            Clock::setTime(hours, minutes, seconds);
            TimeSource::notify(hours, minutes, seconds);
        }
};
```

糟糕的是，无法在 Java 中使用这种方法，因为 Java 语言不支持类的多重继承。所以，在 Java 中，我们要么顺其自然，要么使用委托方法。程序 24.19 至程序 24.21 以及图 24.9 展示了委托方法。

程序 24.19 TimeSource.java

```
public interface TimeSource
{   public void registerObserver(ClockObserver observer);
}
```

程序 24.20 TimeSourceImplmentation.java

```
import java.util.*;

public class TimeSourceImplementation
{
    private Vector itsObservers = new Vector();

    protected void notify(int hours, int minutes, int seconds)
    {
        Iterator i = itsObservers.iterator();
```

```
        while (i.hasNext())
        {
            ClockObserver observer = (ClockObserver) i.next();
            observer.update(hours, minutes, seconds);
        }
    }

    public void registerObserver(ClockObserver observer)
    {
        itsObservers.add(observer);
    }
}
```

程序 24.21　MockTimeSource.java

```
public class MockTimeSource implements TimeSource
{
    TimeSourceImplementation tsImp =
            New TimeSourceImplementation();

    public void registerObserver(ClockObserver observer);
    {
        tsImp.registerObserver(observer);
    }

    public void setTime(int hours, int minutes, int seconds)
    {
        tsImp.notify(hours, minutes, seconds);
    }
}
```

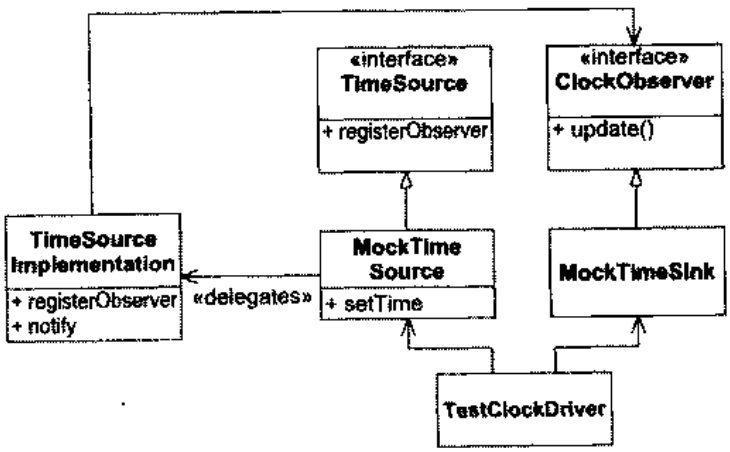


图 24.9　在 Java 中使用委托方法实现 Observer 模式

请注意，MockTimeSource 类实现了 TimeSource 并且包含一个指向 TimeSourceImplementation 实例的引用。同样请注意，所有对 MockTimeSource 的 registerObserver 方法的调用都被委托给那个 TimeSourceImplementation 对象。此外，MockTimeSource.setTime 还调用了 TimeSourceImplementetion 实例的 notify 方法。

这虽然很丑陋，但是它有一个优点，就是 MockTimeSource 没有去继承（extend）一个类。这意味着如果我们要去创建 ObservableClock，它就可以继承 Clock，实现（implement）TimeSource，并委托给 TimeSourceImplementation（参见图 24.10）。这就以微小的代价解决了 Clock 依赖于注册和更新逻辑

的问题。

好了，我们结束这个无尽的讨论，再回到图 24.7 中展示的内容。我们完全接受 Clock 必须依赖于所有的注册和更新逻辑的事实。

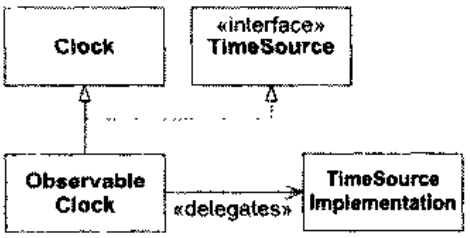


图 24.10　ObservableClock 的委托方法实现

TimeSource 这个名字无法清楚地表达出该类要做的事情。一开始，在还有 ClockDriver 时，这个名字还不错。但是从那以后，这个名字就变得非常糟糕了。我们应该对名字进行更改，使人看到它就会想到注册和更新。OBSERVER 模式把这个类称为 Subject。在我们的情形中，它是特定于时间的，所以称它为 TimeSubject，不过这个名字不太直观。我们可以使用以前 Java 中的命名：Observable，但是它也不能令我满意。TimeObservable？——也不好。

也许，“推模型（push model）”OBSERVER 模式的特殊性才是问题的关键。[1]如果改成“拉模型（pull model）”的话，我们就可以使这个类具有一般性。这样，我们可以把 TimeSource 的名字改为 Subject，每一个熟悉 OBSERVER 模式的人都会明白它的含意。

这是一个不错的选择。我们不是把时间传递给 notify 和 update 方法，而是让 TimeSink 向 MockTimeSource 索要时间。我们不想让 MockTimeSink 知道 MockTimeSource，所以我们创建一个接口，MockTimeSource 可以使用这个接口来获得时间。MockTimeSource（和 Clock）会实现这个接口。我们称这个接口为——嗯——TimeSource。

图 24.11 以及程序 24.22 至 24.27 中为最终的代码和 UML 图。

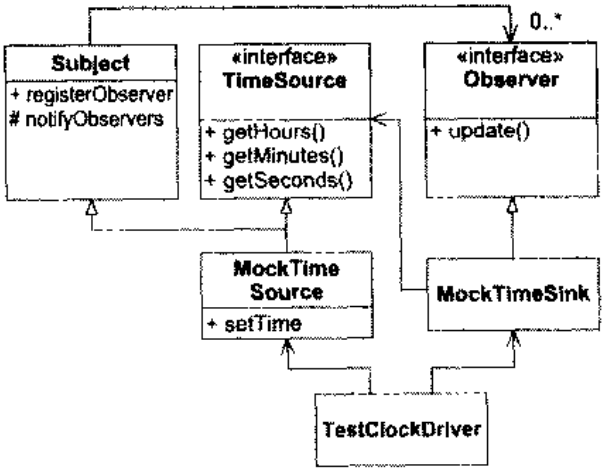


图 24.11　在 MockTimeSource 和 MockTimeSink 上应用 OBSERVER 模式的最终版本

程序 24.22　ObserverTest.java

```
import junit.framework.*;
public class ObserverTest extends
TestCase
{
  private MockTimeSource source;
  private MockTimeSink sink;

  public ObserverTest(String name)
  {
    super(name);
  }

  public void setUp()
  {
    source = new MockTimeSource();
    sink =
        new MockTimeSink(source);
    source.registerObserver(sinkprivate void assertSinkEquals(sink));
  }

  private void assertSinkEquals(
            MockTimeSink sink, int hours, int minutes, int seconds)
```

① 在 OBSERVER 模式的“推模型”实现中，是通过把数据传给 notify 和 update 方法从而把数据从目标（subject）推给观察者（observer）的。在 OBSERVER 模式的“拉模型”实现中，没有给 notify 和 update 方法传递任何数据，数据是在观察者对象收到更新消息后，查询被观察者对象得到的，请参见[GOF95]。

```
  {
    assertEquals(hours, sink.getHours());
    assertEquals(minutes, sink.getMinutes());
    assertEquals(seconds, sink.getSeconds());
  }

  public void testTimeChange()
  {
    source.setTime(3,4,5);
    assertSinkEquals(sink, 3,4,5);

    source.setTime(7,8,9);
    assertSinkEquals(sink, 7,8,9);
  }

  public void testMultipleSinks()
  {
    MockTimeSink sink2 = new MockTimeSink();
    source.registerObserver(sink2);

    source.setTime(12,13,14);
    assertSinkEquals(sink, 12,13,14);
    assertSinkEquals(sink2, 12,13,14);
  }
}
```

程序 24.23 Observer.java

```
public interface Observer
{
    public void update();
}
```

程序 24.24 Suject.java

```
import java.util.*;

public class Subject
{
    private Vector itsObservers = new Vector();

    protected void notifyObserver()
    {
        Iterator i = itsObservers.iterator();
        while (i.hasNext())
        {
            Observer observer = (Observer) i.next();
            observer.update();
        }
    }

    public void registerObserver(Observer observer)
    {
        itsObservers.add(observer);
    }
}
```

程序 24.25 TimeSource.java

```
public interface TimeSource
{
    public int getHours();
```

```
    public int getMinutes();
    public int getSeconds();
}
```

程序 24.26 MockTimeSource.java

```
public class MockTimeSource extends Subject implements TimeSource
{
    private int itsHours;
    private int itsMinutes;
    private int itsSeconds;

    public void setTime(int hours, int minutes, int seconds)
    {
        itsHours = hours;
        itsMinutes = minutes;
        itsSeconds = seconds;
        notifyObservers();
    }

    public int getHours()
    {
        return itsHours;
    }

    public int getMinutes()
    {
        return itsMinutes;
    }

    public int getSeconds()
    {
        return itsSeconds;
    }
}
```

程序 24.27 MockTimeSink.java

```
public class MockTimeSink implements Observer
{
    private int itsHours;
    private int itsMinutes;
    private int itsSeconds;

    public MOckTimeSink(TimeSource source)
    {
        itsSource = source;
    }

    public int getSeconds()
    {
        return itsSeconds;
    }

    public int getMinutes()
    {
        return itsMinutes;
    }

    public int getHours()
    {
```

```
        return itsHours;
    }

    public void update()
    {
        itsHours = itsSource.getHours();
        itsMinutes = itsSource.getMinutes();
        itsSeconds = itsSource.getSeconds();
    }
}
```

24.2 结　论

好了，本章到此结束。我们从一个设计问题开始，经过合理的演化，最后得到了一个规范的 OBSERVER 模式。你可能会抱怨，因为我想得到 OBSERVER 模式，所以本章的内容完全是以可以得到 OBSERVER 模式的方式安排的。我不否认这一点。但这不是真正的问题。

如果你熟悉设计模式，那么在面临一个设计问题时，你的脑海中很可能会浮现出一个模式。随后的问题就是是直接实现这个模式呢，还是通过一系列小步骤不断地去演化代码。本章展示了第二种方案的过程。我不是直接断定 OBSERVER 模式就是手边问题的最佳选择，而是持续地一个接一个地解决问题。最后，代码非常明显地朝着 OBSRVER 模式的方向前进，所以我更改了名字，并把代码整理成规范的形式。

在演化过程中的每一刻，我都可以发现问题已经解决并停止演化。或者，我也可能发现可以通过改变路线并朝着另一个方向发展来解决问题。

24.2.1 本章中图的使用

有些图是为读者而绘制的。我觉得如果用一个概括视图来展示一下我所做的工作的话，会让读者更容易理解一些。如果不是为了展示和说明，我不会去创建它们。不过，有几幅图是为我自己绘制的。有时，我确实需要凝视着我所创建的结构，这样才能知道下一步该如何走。

如果不是在写书，我会把这些图手工地画在一片纸或者一个白板上。我不会在使用画图工具上花费时间。我还不知道在哪种情形中使用画图工具会比使用一小片餐巾纸更快。

在这些图完成了帮助演化代码的任务后，我就不会保留它们。在任何情况下，那些为自己画的图都是中间步骤。

描绘这种层次细节的图有保存的价值吗？显然，如果你试图去展示你的推理，就像我在本书中做的那样，它们是非常能派得上用场的。但是通常我们不会试图去文档化几个小时编码的演化过程。这些图通常都是暂时性的，最好丢弃。对于这种层次的细节来说，通常代码就足以充当自己的文档。对于更高的层次来说，这并不总是正确的。

24.3 OBSERVER 模式

好，既然我们已经完成了样例并把代码演化到了 OBSERVER 模式，那么来彻底研究一下 OBSERVER 模式可能会比较有趣。图 24.12 中展示了 OBSERVER 模式的规范形式。在本例中，Clock 被 DigitalClock 观察。DigitalClock 通过 Subject 接口注册到 Clock 中。无论任何原因，只要时间一改变，

Clock 就调用 Subject 的 notify 方法。而 Subject 的 notify 方法会调用每个已注册 Observer 对象的 update 方法。因此，每当时间发生变化时，DigitalClock 都会接收到一个 update 消息。此时，它会向 Clock 请求时间，然后把时间显示出来。

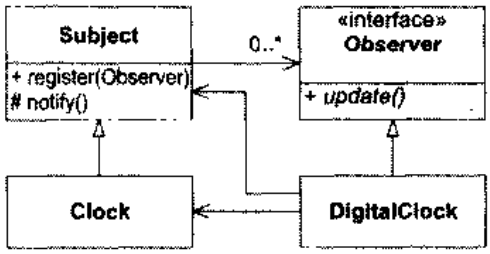


图 24.12 拉模型 OBSERVER 模式的规范形式

OBSERVER 模式是那种一旦你理解了，就会觉得到处都可以使用它的模式之一。这种间接关系非常好。你可以向各种对象注册观察者，而不用让这些对象显式地调用你。虽然这种间接关系是一种有用的管理依赖关系的方法，但是它很容易会被过分使用。过度使用 OBSERVER 模式往往会导致系统难以理解和跟踪。

24.3.1 推我拉你

OBSERVER 模式有两种主要模型。图 24.12[①]展示了拉模型 OBSERVER 模式。因为 DigitalClock 在收到 update 消息后，必须要从 Clock 对象中“拉出”时间信息，所以就给它起了这个名字。

拉模型的优点是它实现起来比较简单，并且 Subject 类和 Observer 类可以成为库中的标准可重用元素。然而，想像一下，如果你正在观察一个具有一千个字段的雇员记录，并且刚好收到了一个更新消息，那么你如何知道是哪个字段发生了变化呢？

当调用 ClockObserver 的 update 方法时，响应方式显而易见。ClockObserver 需要从 Clock 中“拉出”时间并显示它。但是当调用 EmployeeObserver 的 update 方法时，响应方式就不那么明显了。我们不知道发生了什么，也不知道要做什么。也许是雇员的名字改变了，或者是他的薪水改变了。也许是他换了一个新老板，或者是他的银行帐户改变了。我们需要帮助。

推模型形式的 OBSERVER 模式可以为我们提供这个帮助。图 24.13 中展示了推模型 OBSERVER 模式的结构。请注意，notify 方法和 update 方法都带有一个参数。该参数是一个提示（hint），它是通过 notify 方法和 update 方法从 Employee 传到 SalaryObserver 的。这个提示让 SalaryObserver 知道了雇员记录遭受了哪种变化。

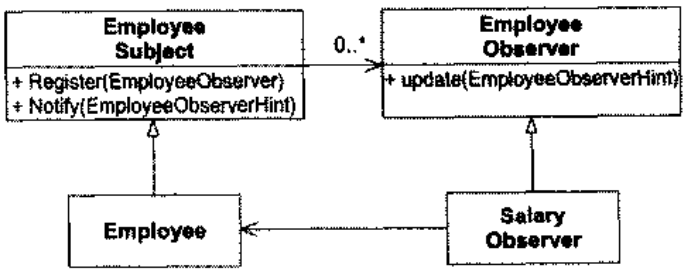


图 24.13 推模型 OBSERVER 模式

notify 和 update 的 EmployeeObserverHint 参数可能是某种枚举、一个字符串或者一个包含了某个字段新、老值的复杂数据结构。不管它是什么，它的值都被推到观察者中。

要选择哪种 OBSERVER 模型完全取决于被观察对象的复杂性。如果被观察对象比较复杂，并且观察者需要一个提示，那么推模型是合适的。如果被观察的对象比较简单，那么拉模型就很合适。

① 原书中为 24.13，有误——译者注。

24.3.2　OBSERVER 模式如何运用面向对象设计的原则

OBSERVER 模式的最大推动力来自开放封闭原则（OCP）。使用这个模式的动机就是为了在增加新的观察对象时可以无需更改被观察的对象。这样，被观察对象就可以保持封闭。

请回顾一下图 24.12，显然，Clock 可以替换 Subject，并且 DigitalClock 可以替换 Observer。因此，本例中也运用了 Liskov 替换原则（LSP）。

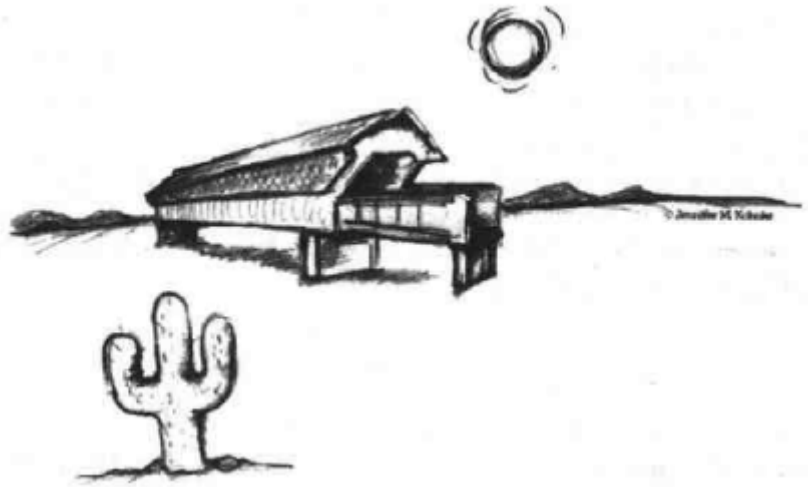
Observer 是一个抽象类，具体的 DigitalClock 依赖于它。Subject 的具体方法也依赖于它。因此，依赖倒置原则（DIP）在本例中也被运用了。你可能会认为，由于 Subject 不具有抽象方法，所以 Clock 和 Subject 之间的依赖关系违反了 DIP。但是，Subject 是一个绝不应该被实例化的类。它只在派生类的上下文中才有意义。所以，尽管 Subject 不具有抽象方法，但它是逻辑抽象的。在 C++中，我们可以通过使 Subejct 的析构函数是纯虚的或者使它的构造函数是受保护的来强制它的抽象性。

从图 24.11 中可以看出接口隔离原则（ISP）的迹象。Subject 和 TimeSource 类为 MockTimeSource 的每个客户提供了特定的接口，从而分离了它的客户。

参考文献

1. Gamma, et al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
2. Martin, Robert C., et al. *Pattern Languages of Program Design 3*. Reading,MA:Addison-Wesley, 1998.

第 25 章　ABSTRACT SERVER 模式、ADAPTER 模式和 BRIDGE 模式



政治家都是完全一样的。即使一个地方没有河，他们也会许诺在那里建造一座桥。

——尼基塔·赫鲁晓夫（前苏联共产党第一书记）

在 20 世纪 90 年代中期，我深深地沉浸到在 comp.object 新闻组上进行的讨论中。我们在新闻组中张贴消息，激烈地争论有关分析和设计的不同策略的问题。在讨论当中，我们觉得一个具体的例子会有助于评价彼此的观点。所以我们就选择一个非常简单的设计问题，然后开始提出各自认可的解决方案。

这个设计问题非常简单。我们选择设计运行在简易台灯中的软件。台灯由一个开关和一盏灯组成。你可以询问开关是开着还是关着，也可以让灯打开或关闭。一个不错的简单问题。

争论激烈地持续了好几个月。每个人都认为自己独特的设计风格优于所有其他人。有些人使用了只有一个开关对象和灯对象的简单方法。另一些人认为应该有一个包含开关和灯的台灯对象。还有一些人认为电流（electricity）也应该是一个对象。居然还有人提出了电线对象。

尽管这些争论中大多数是荒谬的，但是对这个设计模型进行探索还是很有趣的。请考虑一下图 25.1。我们当然可以使这个设计工作起来。Switch 对象可以轮询真实开关的状态，并且可以发送相应的 turnOn 和 turnOff 消息给 Light。

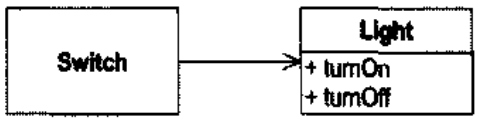


图 25.1　简易的台灯

我们为何不喜欢这个设计呢？

这个设计违反了两个设计原则：依赖倒置原则（DIP）和开放封闭原则（OCP）。对 DIP 的违反是明显的，Switch 依赖了具体类 Light。DIP 告诉我们要优先依赖于抽象类。对 OCP 的违反虽然不那

么明显，但是更加切中要害。我们之所以不喜欢这个设计是因为它迫使我们在任何需要 Switch 的地方都要附带上 Light。我们不能容易地扩展 Switch 去管理除 Light 外的其他对象。

25.1　ABSTRACT SERVER 模式

你也许认为可以从 Switch 继承一个子类，这样就可以控制除灯外的其他东西了，如图 25.3 所示。但是这没有解决问题，因为 FanSwitch 仍然继承了对 Light 的依赖。只要你使用了 FanSwitch，就必须要附带上 Light。无论如何，这个特定的继承关系都会违反 DIP。

为了解决这个问题，我们使用了一个最简单的设计模式：ABSTRACT SERVER 模式（参见图 25.2）。我们在 Switch 和 Light 之间引入一个接口，这样就使得 Switch 能够控制任何实现了这个接口的东西。这立即就满足了 DIP 和 OCP。

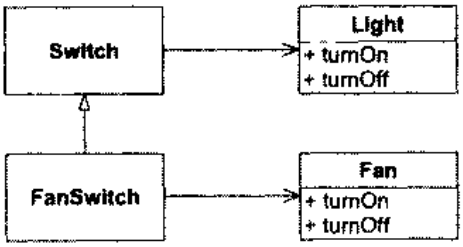


图 25.2　扩展 Switch 的糟糕方法

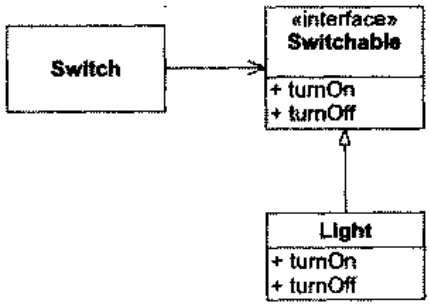


图 25.3　用 ABSTRACT SERVER 模式解决台灯问题

25.1.1　谁拥有这个接口

作为一个有趣的插入语，请注意接口的名字是从它的客户的角度起的。它被称为 Switchable 而不是 ILight。我们在前面已经谈论过这个问题，并且可能还会再次看到它。接口属于它的客户，而不是它的派生类。客户和接口之间的逻辑绑定关系要强于接口和它的派生类之间的逻辑绑定关系。它们之间的关系强到在没有 Switchable 的情况下就无法使用 Switch；但是，在没有 Light 的情况下却完全可以使用 Switch。逻辑关系的强度和实体（physical）关系的强度是不一致的。继承是一个比关联强得多的实体关系。

在 20 世纪 90 年代初期，我们通常认为实体关系支配着一切。有很多名著都建议把继承层次结构一起放到同一个实体包中。这似乎是合理的，因为继承是一种非常强的实体关系。但是在最近的 10 年中，我们已经认识到继承的实体强度是一个误导，并且继承层次结构通常也不应该被打包在一起。相反，往往是把客户和它们控制的接口打包在一起。

这种逻辑和实体关系强度的不一致性是静态类型语言（像 C++和 Java）的一个产物。动态类型语言（像 Smalltalk、Python 和 Ruby）不具有这种不一致性，因为它们没有用继承去实现多态行为。

25.2　ADAPTER 模式

图 25.3 中的设计有一个问题。它可能会违反单一职责原则（SRP）。我们把 Light 和 Switchable 绑定在一起，而它们可能会因为不同的原因改变。如果无法把继承关系加到 Light 上该怎么办呢？如果从第三方购买了 Light，而没有源代码该怎么办呢？或者如果想让 Switch 去控制其他一些类，但是却

不能让它们从 Switchable 派生该怎么办呢？引入 ADAPTER 模式。[1]

图 24.5 中展示了使用 ADAPTER 模式的解决方案。适配器从 Switchable 派生并委托给 Light。问题被优美地解决了。现在，Switch 就可以控制任何能够被打开或者关闭的对象。我们所需要做的只是创建一个合适的适配器。事实上，对象甚至不需要具有和 Switchable 中一样的 turnOn 和 turnOff 方法。适配器会适配到对象的接口。

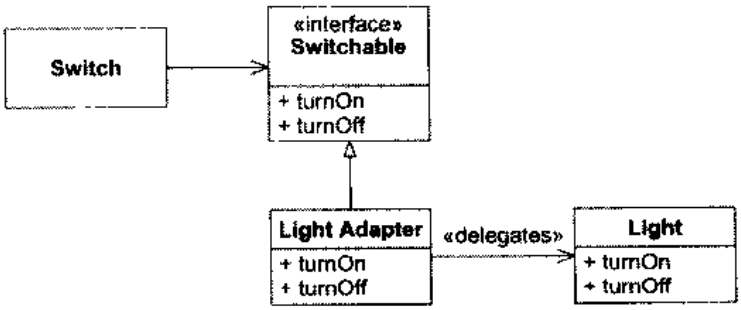


图 25.4　使用 ADAPTER 模式解决台灯问题

25.2.1　没有免费的午餐

使用适配器是有代价的。你需要编写新的类，需要实例化适配器并把要适配的对象和它绑定起来。然后，每当你调用适配器时，必须要付出委托所需的时间和空间代价。所以，你显然不想始终都使用适配器。对大多数情况来说，ABSTRACT SERVER 解决方案就非常合适了。事实上，就是图 25.1 中最初的解决方案也是相当好的，除非你正好知道还有其他对象需要 Switch 去控制。

25.2.2　类形式的 ADAPTER 模式

图 25.4 中的 LightAdapter 类被称为对象形式的适配器。还有一种被称为类形式的适配器的方法，如图 25.5 所示。在这种形式中，适配器对象同时继承了 Switchable 接口和 Light 类。这种形式比对象方式稍微高效一点，也易于使用一些，但是却付出了使用高耦合度的继承关系的代价。

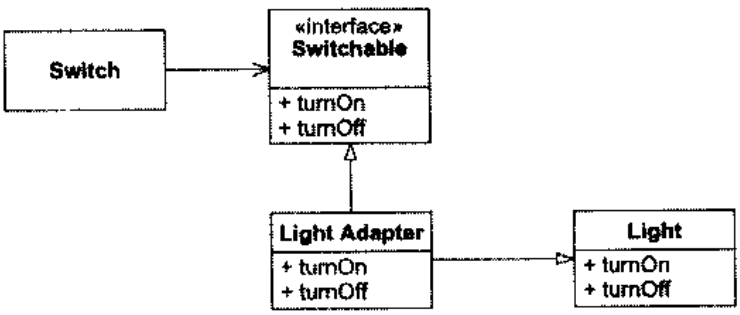


图 25.5　使用 ADAPTER 模式解决台灯问题

25.2.3　调制解调器问题、适配器以及 LSP

请考虑一下图 25.6 中的情形。我们有大量的调制解调器客户程序，它们都使用 Modem 接口。Modem 接口被几个派生类 HayesModem、USRoboticsModem 和 EariniesModem 实现。这是常见的方案，它很好地遵循了 OCP、LSP 和 DIP。当增加新种类的调制解调器时，调制解调器的客户

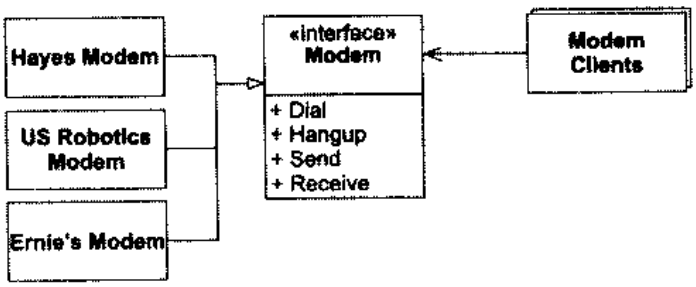


图 25.6　调制解调器问题

[1] 我们已经在前面看到过 ADAPTER 模式，请参见第 10 章中的图 10.2 和图 10.3。

程序不会受影响。假定这种情形持续了几年。假定有许多调制解调器的客户程序都在愉快地使用着 Modem 接口。

现在假定客户提出了一个新的需求。有某些种类的调制解调器是不拨号的。它们被称为专用调制解调器，因为它们位于一条专用连接的两端。[1] 有几个新应用程序使用这些专用调制解调器，它们无需拨号。我们称这些使用者为 DedUser。但是，客户希望当前所有的调制解调器客户程序都可以使用这些专用调制解调器。他们不希望去更改许许多多的调制解调器客户应用程序，所以完全可以让这些调制解调器客户程序去拨一些假（dummy）电话号码。

如果能选择的话，我们会把系统的设计更改为如图 25.7 所示的那样。我们会使用 ISP 把拨号和通信功能分离为两个不同的接口。原来的调制解调器实现这两个接口，而调制解调器客户程序使用这两个接口。DedUser 只使用 Modem 接口，而 DedicateModem 只实现 Modem 接口。糟糕的是，这样做会要求我们更改所有的调制解调器客户程序——这是客户不允许的。

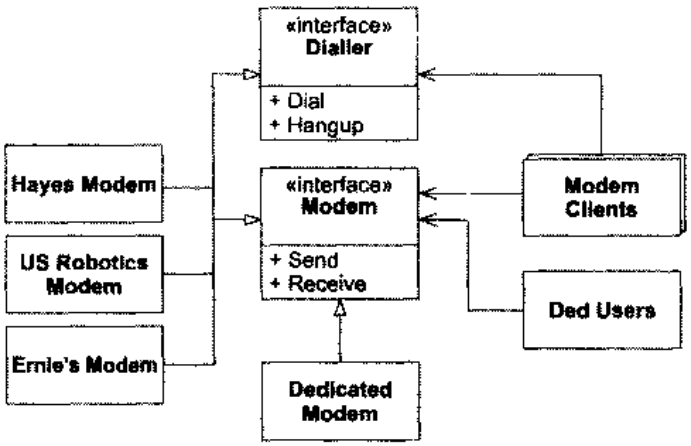


图 25.7　调制解调器问题的理想解决方案

那么我们该怎么办呢？我们不能像希望的那样去分离接口，可是还得找到一个让所有的调制解调器客户程序使用 DedicatedModem 的方法。一个可能的解决方案是让 DedicatedModem 从 Modem 派生并且把 dial 方法和 hangup 方法实现为空，就像下面这样：

这两个退化函数预示着我们可能违反了 LSP。基类的使用者可能期望 dial 和 hangup 会明显地改变调制解调器的状态。DedicatedModem 中的退化实现可能会违背这些期望。

假定调制解调器客户程序期望在调用 dial 方法前调制解调器处于休眠状态，并且当调用 hangup 时返回休眠状态。换句话说，它们期望不会从没有拨号的调制解调器中收到任何字符。DedicatedModem 违背了这个期望。在调用 dial 之前，它就会返回字符，并且在调用 hangup 调用之后，仍会不断地返回字符。所以，DedicatedModem 可能会破坏某些调制解调器的使用者。

现在你可能会认为问题是由调制解调器的客户程序引起的。如果它们因为不期望的输入而崩溃，是因为它们做的不够好。我同意这个观点。但是如果仅仅是因为我们增加了一种新调制解调器的原因，就让那些维护调制解调器客户程序的人去更改他们的软件，这是很难令他们信服的。这不但违反了 OCP，而且同样是令人沮丧的。此外，我们的客户已经明确地禁止更改调制解调器的客户程序。

25.2.4　使用杂凑的方法来修正这个问题

我们可以在 DedicatedModem 的 dial 方法和 hangup 方法中模拟一个连接状态。如果还没有调用 dial，或者已经调用了 hangup，就可以拒绝返回字符。如果这样做的话，那么所有的调制解调器客户程序

① modem 过去通常都是专用的。只是在近期，modem 才有了拨号能力。在以前，你得从电话公司租用一台面包箱大小的 modem 并通过专线把它和另一个也是你从电话公司租用的 modem 连接起来（那时电话公司的生意是不错的）。如果想拨号，你要从电话公司租用另一个面包箱大小的称为自动拨号器的设备。

都可以正常工作并且也不必更改。只要让 DedUser 去调用 dial 和 hangup 即可（参见图 25.8）。

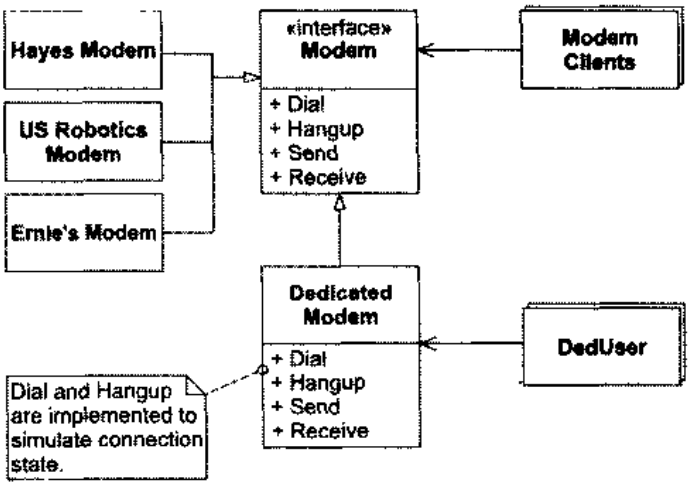


图 25.8　通过临时让 DedicatedModem 模仿连接状态来解决调制解调器问题

你可能认为这种做法会令那些正在实现 DedUser 的人觉得非常沮丧。他们明明在使用 DedicatedModem。为什么他们还要去调用 dial 和 hangup 呢？不过，他们的软件还没有开始编写，所以还比较容易让他们按照我们的想法去做。

25.2.5　混乱的依赖关系网

几个月后，已经有了大量的 DedUser，此时客户提出了一个新的更改。这些年来，我们的程序似乎都没有拨过国际电话号码。这就是为什么在 dial 中使用 char[10]而没有出问题的原因。但是，现在，客户希望能够拨打任意长度的电话号码。他们需要去拨打国际电话、信用卡电话、PIN 标识电话，等等。

显然，所有的调制解调器客户程序都必须更改。在它们中是用 char[10]来表示电话号码的。客户同意了对调制解调器客户程序的更改，因为他们别无选择，我们把大量的程序员投入到这个任务中。同样显然的是，调制解调器层次结构中的类都必须更改以容纳新电话号码的长度。我们的小开发团队可以处理这个问题。糟糕的是，现在我们必须要去告诉 DedUser 的编写者，他们必须要更改他们的代码！你可以想象他们听到这个会有多高兴。本来他们是不用调用 dial 的。他们之所以调用了 dial 是因为我们告诉他们必须要这样做。现在，他们将要遭受高代价的维护工作，因为他们做了我们让他们做的事情。

这就是许多项目都会具有的那种有害的混乱依赖关系。系统某一部分中的一个杂凑体（kludge）创建了一个有害的依赖关系，最终导致系统中完全无关的部分出现问题。

25.2.6　用 ADAPTER 模式来解决

如果使用 ADAPTER 模式解决最初的问题的话（参见图 25.9），就可以避免这个严重问题。在这种方案中，DedicatedModem 不从 Modem 继承。调制解调器客户程序通过 DedicatedModemAdapter 间接地使用 DedicatedModem。在这个适配器的 dial 和 hangup 的实现中去模拟连接状态。它把 send 和 receive 调用委托给 DedicatedModem。

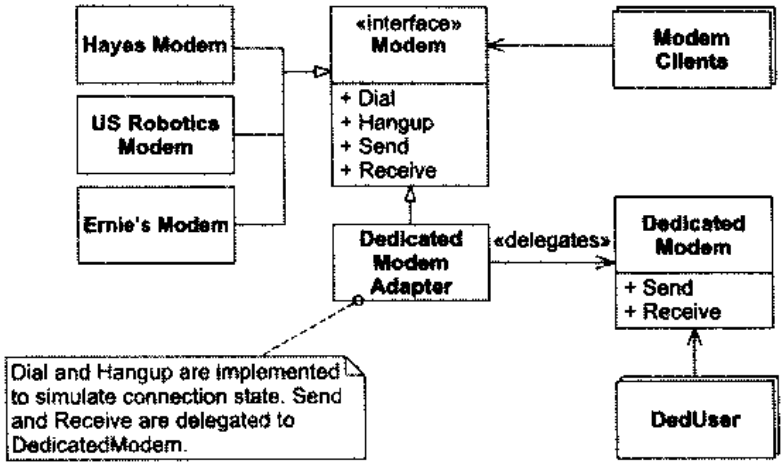


图 25.9　使用 ADAPTER 模式解决调制解调器问题

请注意，这消除了我们以前遇到的所有困难。调制解调器的客户程序看到的是它们期望的连接行为，并且 DedUser 也不必去调用 dial 和 hangup。当改变有关电话号码的需求时， DedUser 不会受到影响。因此，通过在适当的位置放置适配器，我们修正了对于 LSP 和 OCP 的违反。

请注意，杂凑体仍然存在。适配器仍然要模拟连接状态。你可能认为这很丑陋，我当然同意你的观点。然而，请注意，所有的依赖关系都是从适配器发起的。杂凑体和系统隔离，藏身于几乎无人知晓的适配器中。只有在某处的某个工厂才可能会实际依赖于这个适配器。[①]

25.3　BRIDGE 模式

看待这个问题，还有另外一个方式。对于专用调制解调器的需要向 Modem 类型层次结构中增加了一个新的自由度。在最初构思 Modem 类型时，它只是一组不同硬件设备的接口。因此，我们让 HayesModem、USRModem 和 ErniesModem 从基类 Modem 派生。但是，现在，出现了另外一种切分 Modem 层次结构的方式。我们可以让 DialModem 和 DedicatedModem 从 Modem 派生。

可以像图 25.10 中的那样把这两个独立的层次结构合并起来。类型层次结构的每一个叶子节点要么向它所控制的硬件提供拨号行为；要么提供专用行为。DedicatedHayesModem 对象以专用的方式控制着 Hayes 品牌的调制解调器。

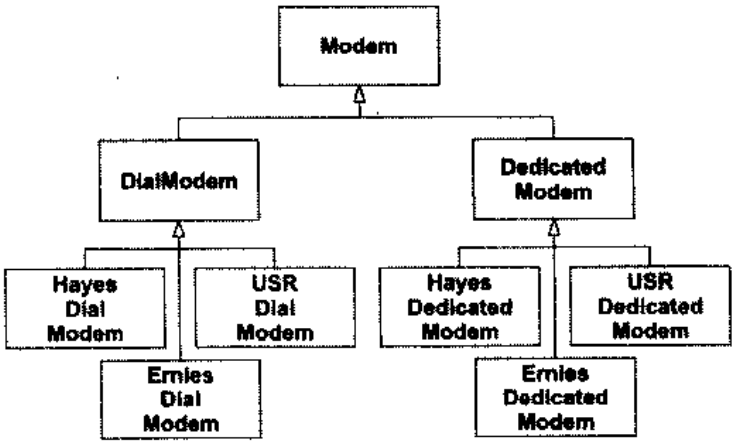


图 25.10　通过合并类型层次结构解决调制解调器问题

这不是一个理想的结构。每当增加一款新硬件时，就必须创建两个新类——一个针对专用的情况，一个针对拨号的情况。每当增加一种新连接类型时，就必须创建 3 个新类，分别对应 3 款不

① 请参见第 21 章：FACTORY 模式。

同的硬件。如果这两个自由度根本就是不稳定的，那么不用多久，就会出现大量的派生类。

在类型层次结构具有多个自由度的情况中，BRIDGE 模式通常是有用的。我们可以把这些层次结构分开并通过桥把它们结合到一起，而不是把它们合并起来。

图 25.11 展示了这个结构。我们把调制解调器类层次结构分成两个层次结构。一个表示连接方法，另一个表示硬件。

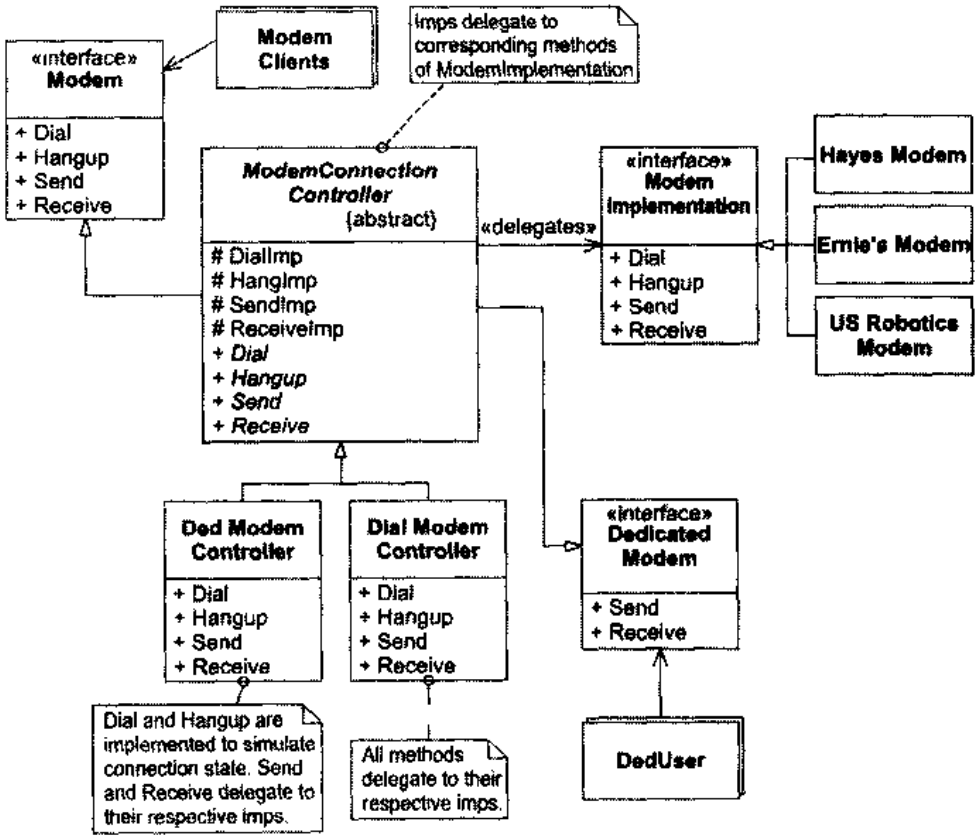


图 25.11　使用 BRIDGE 模式解决调制解调器问题

调制解调器的使用者继续使用 Modem 接口，ModemConnectionController 实现了 Modem 接口。ModemConnectionController 的派生类控制着连接机制。DialModemController 的 dial 方法和 hangup 方法只是转调用基类 ModemConnectionController 中的 dialImp 和 hangImp。接着，这两个方法把调用委托给类 ModemImplementation，在那里它们会被分派到适当的硬件控制器。DedModemController 把 dial 和 hangup 实现为模仿连接状态。它的 send 和 receive 会转调用 sendImp 和 receiveImp，并像前面一样再委托给 ModemImplementation 层次结构。

请注意，ModemConnectionController 基类中的 4 个 imp 方法都是受保护的（protected）。这是因为他们只被 ModemConnectionController 的派生类使用。其他任何类都不应当调用它们。

这个结构虽然复杂，但是很有趣。它的创建不会影响到调制解调器的使用者，并且还完全分离了连接策略和硬件实现。ModemConnectController 的每个派生类代表了一个新的连接策略。在这个策略的实现中可以使用 sendImp、receiveImp、dialImp 和 hangImp。新 imp 方法的增加不会影响到使用者。可以使用 ISP 来给连接控制类增加新的接口。这种做法可以创建出一条迁移路径，调制解调器的客户程序可以沿着这条路径慢慢地得到一个比 dial 和 hangup 层次更高的 API。

25.4　结　　论

有人可能非常想说，调制解调器场景中的真正问题是最初的设计者设计错了。他们本应该知道连接和通讯是不同的概念。如果他们稍稍多做一些分析，就会发现这个问题并且改正它。所以，很容易把问题归结为不充分的分析。

胡说！根本不存在充分分析这种东西。无论花多少时间试图去找出完美的软件结构，客户总是会引入一个变化破坏这个结构。

这种情况是无法避免的。不存在完美的结构。只存在那些试图去平衡当前的代价和收益的结构。随着时间的过去，这些结构肯定会随着系统需求的改变而改变。管理这种变化的诀窍是尽可能地保持系统简单、灵活。

使用 ADAPTER 模式的解决方案是简单和直接的。它让所有的依赖关系都指向正确的方向，并且实现起来非常简单。BRIDGE 模式稍稍有些复杂。我建议在开始时不要使用 BRIDGE 模式，直到你明显可以看出需要完全分离连接策略和通信策略并且需要增加新的连接策略时，才使用这种方法。

向往常一样，这里要讲的是，模式是既能带来好处又具有代价的东西。你应该使用那些最适合手边问题的模式。

参考文献

1．Gamma,et al.*Design Patterns*,Reading, MA:Addison-Wesley,1995

第 26 章　PROXY 模式和 STAIRWAY TO HEAVEN 模式：管理第三方 API

还有人记得笑声吗？

——罗伯特·布朗特（美国著名摇滚歌手），*The Song Remains the Same*

软件系统中存在很多障碍。当把数据从程序移到数据库中去时，我们正在跨越数据库障碍。当把消息从一台计算机发送到另一台计算机时，我们正在跨越网络障碍。

跨越这些障碍可能是复杂的。如果不小心，那么我们的软件就更多的是在处理有关障碍的问题而不是本来要解决的问题。本章中的模式会有助于我们在跨越这些障碍的同时，仍然保持程序关注于本身要解决的问题

26.1　PROXY 模式

假设我们为一个网站编写一个购物车系统。这样的系统中会有一些关于客户、订单（购物车）及订单上的商品的对象。图 26.1 展示了一个可能的结构。这个结构虽然简单，但是符合我们的需要。

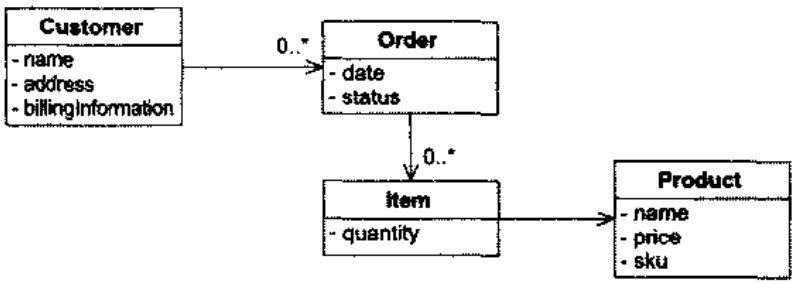


图 26.1　简单的购物车应用对象模型

如果我们考虑一下向订单中增加新商品条目的问题，就可能会得到程序 26.1 中的代码。Order 类的 addItem 方法只是创建一个新的 Item，该 Item 拥有适当的 Product 和数量。然后，它把这个 Item 增加到自己内部的 Item 向量中。

程序 26.1　向对象模型中增加一个商品项

```
public class Order
{
    private Vector itsItems = new Vector();
    public void addItem(Product p, int qty)
    {
        Item item = new Item(p, qty);
        itsItems.add(item);
    }
}
```

现在，假设这些对象所代表的数据保存在一个关系数据库中的。图 26.2 展示了可能代表这些对象的表和键。为了得到一个指定客户的订单，你就找出所有具有该客户 cusid 的订单。为了得到一个指定订单中的所有商品条目，你就找出具有该订单 orderId 的那些商品条目。为了得到商品条目上提及的商品，你就使用商品的 sku。

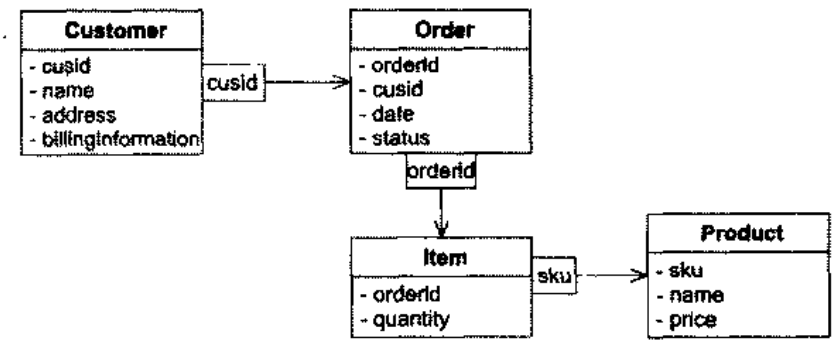


图 26.2　购物车应用的关系数据模型

如果我们想把一个商品条目增加到一个特定的订单中，我们会使用类似程序 26.2 中的代码。该代码使用 JDBC 直接去操纵关系数据模型。

程序 26.2　向关系数据模型中增加一个条目

```
public class AddItemTransaction extends Transaction
{
    public void addItem(int orderId, String sku, int qty)
    {
        Statement s = itsConnection.CreateStatement();
        s.executeUpdate("insert into items values(" +
                        orderId + "," + sku + "," +
                        qty + ")");
    }
}
```

虽然这两个代码片段非常不同，但是它们执行的却是相同的逻辑功能。它们都是把商品条目和订单联系起来。第一个忽略了数据库的存在，而第二个则完全依赖于数据库。

显然，购物车程序就是关于订单、商品条目和商品的。糟糕的是，如果我们使用程序 26.2 中的代码，就使得该程序去关注 SQL 语句、数据库连接以及拼凑在一起的查询字符串。这严重违反了

SRP，并且还可能违反 CCP。程序 26.2 把两个具有不同更改原因的概念混合在一起。它把商品条目和订单的概念与关系模式（schema）和 SQL 的概念混合在了一起。无论任何原因造成其中的一个概念需要更改，另一个概念就会受到影响。程序 26.2 也违反了 DIP，因为程序的策略依赖于存储机制的细节。

PROXY 模式是解决这些问题的一种方法。为了说明这一点，我们来编写一个测试程序，该测试中创建了一个订单并且计算出该订单的总价。程序 26.3 中展示了该程序的重要部分。

程序 26.3 创建订单并验证价钱计算的测试程序

```
public void testOrderPrice()
{
    OrderImp o = new OrderImp("Bob");
    Product toothpaste = new Product("Toothpaste", 129);
    o.addItem(toothpaste, 1);
    assertEquals(129, o.total());
    Product mouthwash = new Product("Mouthwash", 342);
    o.addItem(mouthwash, 2);
    assertEquals(813, o.total());
}
```

程序 26.4 至 26.6 中展示了通过该测试的简单代码。它使用了图 26.1 中的简单对象模型。它根本就没有考虑数据库的存在。同时，它在许多方面也是不完善的。它只是正好能让测试通过的代码。

程序 26.4 order.java

```
public class Order
{
    private Vector itsItems = new Vector();

    public Order(String cusid)
    {
    }

    public void addItem(Product p, int qty)
    {
        Item item = new Item(p, qty);
        itsItems.add(item);
    }

    public int total()
    {
        int total = 0;
        for (int i=0; i<itsItems.size(); i++)
        {
            Item item = (Item) itsItems.elementAt(i);
            Product p = item.getProduct();
            int qty = item.getQuantity();
            total += p.getPrice() * qty;
        }
        return total;
    }
}
```

程序 26.5 product.java

```
public class Product
{
    private int itsPrice;
```

```
    public Product(String name; int price)
    {
        itsPrice = price;
    }

    public int getPrice()
    {
        return itsPrice;
    }
}
```

程序 26.6 item.java

```
public class Item
{
  private Product itsProduct;
  private int itsQuantity;

  public Item(Product p, int qty)
  {
    itsProduct = p;
    itsQuantity = qty;
  }

  public Product getProduct()
  {
    return itsProduct;
  }

  public int getQuantity()
  {
    return itsQuantity;
  }
}
```

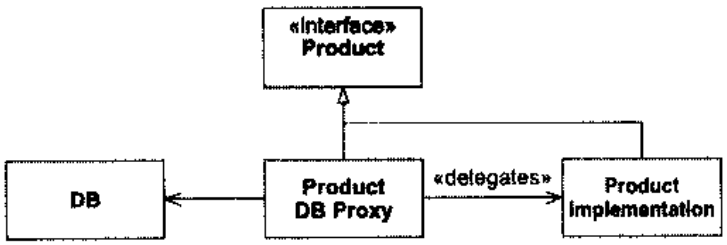


图 26.3 PROXY 模式的静态模型

图 26.3 和图 26.4 展示了 PROXY 模式的工作原理。每个要被代理的对象都被分成 3 个部分。第一部分是一个接口，该接口中声明了客户要调用的所有方法。第二部分是一个类，该类在不涉及数据库逻辑的情况下实现了接口中的方法。第 3 部分是一个知晓数据库的代理。

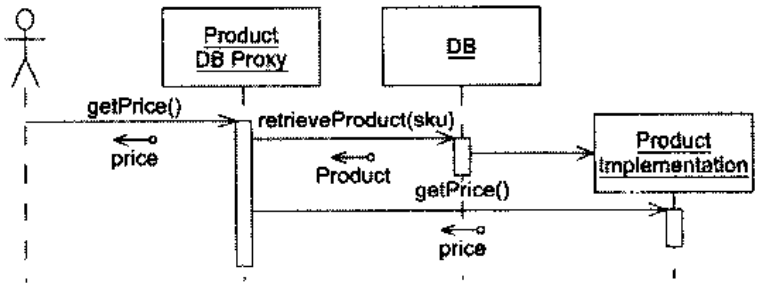


图 26.4 PROXY 模式的动态模型

请考虑一下 Product 类。我们通过用一个接口来代替它实现了对它的代理。这个接口具有 Product 类的所有方法。ProductImplementation 类几乎和原先一样地实现这个接口。ProductDBProxy 实现了 Product 中的所有方法，这些方法从数据库中取出产品，创建一个 ProductImplementation 实例，然后再把消息委托给这个实例。

图 26.4 中的顺序图（sequence diagram）展示了这是如何工作的。客户向一个它认为是 Product，但实际上是 ProductDBProxy 的对象发送 getPrice 消息。ProductDBProxy 从数据库中获取 Productimplementation，然后把 getPrice 方法委托给它。

客户和 ProductImplementation 都不知道所发生的事情。数据库在这两者都不知道的情况下被插入到应用程序中。这正是 PROXY 模式的优点。理论上，它可以在两个协作的对象都不知道的情况下被插入到它们之间。因此，使用它可以跨越像数据库或者网络这样的障碍，而不会影响到任何一个参与者。

事实上，使用代理并不是一件简单的事情。为了能够认识到其中的某些问题，让我们试着在简单的购物车应用中使用 PROXY 模式。

26.1.1 代理化购物车应用

Product 类的代理创建起来是最简单的。在我们的应用中，商品表代表了一个简单的字典（dictionary）。在某个地方，它里面会被放入所有的商品。其他任何地方都不会操作该表，因此该代理显得比较简单。

我们需要一个用来存储和取回商品数据的简单数据库工具来作为出发点。代理将会使用这个接口去操作数据库。程序 26.7 中展示了我设想的测试程序。程序 26.8 和程序 26.9 中的代码通过了这个测试。

程序 26.7 DBTest.java

```
import junit.framework.*;
import junit.swingui.TestRunner;

public class DBTest extends TestCase
{
  public static void main(String[] args)
  {
    TestRunner.main(new String[]{"DBTest"});
  }

  public DBTest(String name)
  {
    super(name);
  }

  public void setUp() throws Exception
  {
    DB.init();
  }

  public void tearDown() throws Exception
  {
    DB.close();
  }

  public void testStoreProduct() throws Exception
  {
    ProductData storedProduct = new ProductData();
    storedProduct.name = "MyProduct";
    storedProduct.price = "1234";
    storedProduct.sku = "999";
    DB.store(storedProduct);
    ProductData retrievedProduct = DB.getProductData("999");
    DB.deleteProductData("999");
```

```
    assertEquals(storedProduct, retrievedProduct);
  }
 }
```

程序 26.8　ProductData.java

```
public class ProductData
{
  public String name;
  public int price;
  public String sku;

  public ProductData()
  {
  }

  public ProductData(String name, int price, String sku)
  {
    this.name = name;
    this.price = price;
    this.sku = sku;
  }

  public boolean equals(Object o)
  {
    ProductData pd = (ProductData)o;
    return name.equals(pd.name) &&
           sku.equals(pd.sku) &&
           price==pd.price;
  }
}
```

程序 26.9　DB.java

```
import java.sql.*;

public class DB
{
  private static Connection con;

  public static void init() throws Exception
  {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    con = DriverManager.getConnection("jdbc:odbc:PPP Shopping Cart");
  }

  public static void store(ProductData pd) throws Exception
  {
    PreparedStatement s = buildProductInsertionStatement(pd);
    executeStatement(s);
  }

  private static PreparedStatement
        buildProductInsertionStatement(ProductData pd) throws SQLException
  {
    PreparedStatement s = con.prepareStatement(
                          "INSERT into Products VALUES (?, ?, ?)");
    s.setString(1, pd.sku);
    s.setString(2, pd.name);
    s.setInt(3, pd.price);
```

```
  return s;
}

public static ProductData getProductData(String sku) throws Exception
{
  PreparedStatement s = buildProductQueryStatement(sku);
  ResultSet rs = s.executeQueryStatement(s);
  ProductData pd = extractProductDataFromResultSet(rs);
  rs.close();
  s.close();
  return pd;
}

private static PreparedStatement
              buildProductQueryStatement(String sku) throws SQLException
{
  PreparedStatement s = con.prepareStatement(
              "SELECT * FROM Products WHERE sku = ?;");
  s.setString(1, sku);
  return s;
}

private static ProductData
      extractProductDataFromResultSet(ResultSet rs) throws SQLException
{
  ProductData pd = new ProductData();
  pd.sku = rs.getString(1);
  pd.name = rs.getString(2);
  pd.price = rs.getInt(3);
  return pd;
}

public static void deleteProductData(String sku) throws Exception
{
  executeStatement(buildProductDeleteStatement(sku));
}

private static PreparedStatement buildProductDeleteStatement(String sku)
                  throws SQLException
{
   PreparedStatement s = con.prepareStatement(
          "DELETE from Products where sku = ?");
   s.setString(1, sku);
   return s;
}

private static void executeStatement(PreparedStatement s)
                  throws SQLException
{
  s.execute();
  s.close();
}

private static ResultSet executeQueryStatement(PreparedStatement s)
                  throws SQLException
{
   ResultSet rs = s.executeQuery();
   rs.next();
   Return rs;
}
```

```
  public static void close() throws Exception
  {
    con.close();
  }

}
```

下一步，我们编写一个测试来展示一下代理是如何工作的。这个测试向数据库中增加一个商品。然后它创建一个具有被存储商品 sku 的 ProductProxy 并且试图使用 Product 的访问方法（accessor）从代理中获取数据（参见程序 26.10）。

程序 26.10　ProxyTest.java

```
import junit.framework.*;
import junit.swingui.TestRunner;

public class ProxyTest extends TestCase
{
  public static void main(String[] args)
  {
    TestRunner.main(new String[]{"ProxyTest"});
  }

  public ProxyTest(String name)
  {
    super(name);
  }

  public void setUp() throws Exception
  {
    DB.init();
    ProductData pd = new ProductData();
    pd.sku = "ProxyTest1";
    pd.name = "ProxyTestName";
    pd.price = 456;
    DB.store(pd);
  }

  public void tearDown() throws Exception
  {
     DB.deleteProductData("ProxyTest1");
     DB.close();
  }

  public void testProductProxy() throws Exception
  {
    Product p = new ProductProxy("ProxyTest1");
    assertEquals(456, p.getPrice());
    assertEquals("ProxyTestName1", p.getName());
    assertEquals("ProxyTest1", p.getSku());
  }
```

为了使这种做法可行，我们必须把 Product 的接口和它的实现分离。所以我把 Product 更改成一个接口并且创建了实现该接口的 ProductImp（参见程序 26.11 和程序 26.12）。

请注意，我在 Product 接口中增加了异常。这是因为我是在编写 Product、ProductImp 和 ProxyTest 的同时编写 ProductProxy（参见程序 26.13）的。我在实现它们时都是按照每次一个访问方法的方式

进行的。我们会看到，ProductProxy 类调用了数据库，该调用会抛出异常。我不想让代理去捕获并隐藏这些异常，所以就决定让它们从接口中暴露出来。

程序 26.11 Product.java

```
public interface Product
{
  public int getPrice() throws Exception;
  public String getName() throws Exception;
  public String getSku() throws Exception;
}
```

程序 26.12 ProductImp.java

```
public class ProductImp implements Product
{
  private int itsPrice;
  private String itsName;
  private String itsSku;

  public ProductImp(String sku, String name, int price)
  {
    itsPrice = price;
    itsName = name;
    itsSku = sku;
  }

  public int getPrice()
  {
    return itsPrice;
  }

  public String getName()
  {
    return itsName;
  }

  public String getSku()
  {
    return itsSku;
  }
}
```

程序 26.13 ProductProxy.java

```
public class ProductProxy implements Product
{
  private String itsSku;
  public ProductProxy(String sku)
  {
    itsSku = sku;
  }
  public int getPrice() throws Exception
  {
    ProductData pd = DB.getProductData(itsSku);
    return pd.price;
  }

  public String getName() throws Exception
```

```
  {
    ProductData pd = DB.getProductData(itsSku);
    return pd.name;
  }

  public String getSku() throws Exception
  {
    return itsSku;
  }
}
```

这个代理的实现非常简单。事实上，它和图 26.3 和图 26.4 中展示的模式的规范形式并不是完全匹配。这个结果出乎意料。我原本是要实现 PROXY 模式。但是当这个实现最终完成时，规范形式的模式就没有意义了。

如下所示，在规范的模式中，ProductProxy 会在每个方法中都创建一个 ProductImp，然后再把那个方法委托给 ProductImp。

```
public int getPrice() throws Exception
{
    ProductData pd = DB.getProductData(itsSku);
    ProductImp p = new ProductImp(pd.sku, pd.name, pd.price);
    return p.getPrice();
}
```

ProductImp 的创建对于程序员和计算机资源来说完全是一种浪费。ProductProxy 已经具有了 ProductImp 的访问方法会返回的数据。所以创建 ProductImp，然后再委托给它的做法是没有必要的。这也是另外一个代码是可以引导你偏离你所期望的模式和模型的例子。

请注意，程序 26.13 中 ProductProxy 的 getSku 方法在这个问题上更进了一步。它根本没有从数据库中获取 sku。它为何可以这样做呢？因为它已经具有了 sku。

你可能会认为 ProductProxy 的实现是非常低效的。在每个访问方法中，它都会去使用数据库。如果它把 ProductData 条目进行缓存来避免访问数据库不是会更好一些吗？

虽然这个更改非常简单，但是促使我们这样做的惟一原因就是我们的恐惧。此时，还没有数据显示出这个程序具有性能问题。此外，数据库引擎本身也会做一些缓存处理。所以建立自己的缓存会给我们带来什么好处并不明显。在做这些麻烦的工作前，我们应该等待，直到我们看到性能问题的迹象。

1. 代理化关系

下一步，我们来创建 Order 的代理。每个 Order 实例都包含有许多 Item 实例。在关系模式（schema）中（见图 26.2），这个关系保存在 Item 表中。Item 表的每一行中都含有包含它的 Order 的键值。然而，在对象模型中，这个关系是用 Order 中的一个 Vector 来实现的（参见程序 26.4）。代理必须要以某种方法在这两种形式间进行转换。

我们首先编写一个代理必须要通过的测试用例。这个测试先向数据库中增加几个虚构的商品，然后取得这些商品的代理，并使用它们去调用 OrderProxy 的 addItem 方法。最后，它向 OrderProxy 索要总价（参见程序 26.14）。该测试用例的意图是要展示一下：OderProxy 具有和 Order 一样的行为，但是它是从数据库而不是内存中获取它的数据的。

程序 26.14 ProxyTest.java

```
public void testOrderProxyTotal() throws Exception
{
    DB.store(new ProductData("Wheaties", 349, "wheaties"));
    DB.store(new ProductData("Crest", 258, "crest"));
    ProductProxy wheaties = new ProductProxy("wheaties");
    ProductProxy crest = new ProductProxy("crest");
    OrderData od = DB.newOrder("testOrderProxy");
    OrderProxy order = new OrderProxy(od.orderId);
    order.addItem(crest, 1);
    order.addItem(wheaties, 2);
    assertEquals(956, order.total());
}
```

要通过这个测试用例，我们必须实现几个新的类和方法。首先要解决的是 DB 中的 newOrder 方法。看起来这个方法好像返回了一个称为 OrderData 的类的实例。OrderData 和 ProductData 非常相似。它是一个表示 Order 数据库表中的一行的简单数据结构。程序 26.15 展示了该结构。

程序 26.15 OrderData.java

```
public class OrderData
{
  public String customerId;
  public int orderId;

  public OrderData()
  {
  }

  public OrderData(int orderId, String customerId)
  {
    this.orderId = orderId;
    this.customerId = customerId;
  }
}
```

不要因为使用了公共的数据成员而觉得不舒服。这本来就不是一个真实意义上的对象。它只是一个数据容器。它没有什么有意义的行为需要封装。让数据变量私有并且提供获取和设置方法完全是一种不必要的复杂化。

现在我们需要编写 DB 的 newOrder 方法。请注意，我们在程序 26.14 中调用它时，给它提供了拥有它的客户的 ID，却没有提供 orderId。每个 Order 都需要一个 orderId 来充当它的键值。此外，在关系模式中，每个 Item 都以引用到该 orderId 来表明它和 Order 之间的联系。显然，orderId 必须是惟一的。如何产生它呢？我们编写一个测试来展示我们的意图（参见程序 26.16）。

程序 26.16 DBTest.java

```
public void testOrderKeyGeneration() throws Exception
{
    OrderData o1 = DB.newOrder("Bob");
    OrderData o2 = DB.newOrder("Bill");
    int firstOrderId = o1.orderId;
    int secondOrderId = o2.orderId;
    assertEquals(firstOrderId+1, secondOrderId);
}
```

这个测试表明我们期望每次创建一个新 Order 时，orderId 都会以某种方式自动加 1。这一点很容易实现，只要查询数据库获得当前正在使用的 orderId 的最大值，并在其上加 1 即可（参见程序 26.17）。

程序 26.17　DB.java

```
public static OrderData newOrder(String customerId) throws Exception
{
    int newMaxOrderId = getMaxOrderId() + 1;
    PreparedStatement s = con.prepareStatement(
                    "Insert into Orders(orderId,cusid) Values(?,?);");
    s.setInt(1, newMaxOrderId);
    s.setString(2,customerId);
    executeStatement(s);
    return new OrderData(newMaxOrderId, customerId);
}

private static int getMaxOrderId() throws SQLException
{
    Statement qs = con.createStatement();
    ResultSet rs = qs.executeQuery("Select max(orderId) from Orders;");
    rs.next();
    int maxOrderId = rs.getInt(1);
    rs.close();
    return maxOrderId;
}
```

现在我们可以开始编写 OrderProxy 了。和 Product 一样，我们需要把 Order 的接口和实现分开。所以 Order 变成了接口，而 OrderImp 变成了实现（参见程序 26.18 和程序 26.19）。

程序 26.18　Order.java

```
public interface Order
{
  public String getCustomerId();
  public void addItem(Product p, int quantity);
  public int total();
}
```

程序 26.19　OrderImp.java

```
import java.util.Vector;

public class OrderImp implements Order
{
  private Vector itsItems = new Vector();
  private String itsCustomerId;

  public String getCustomerId()
  {
    return itsCustomerId;
  }

  public OrderImp(String cusid)
  {
    itsCustomerId = cusid;
  }

  public void addItem(Product p, int qty)
  {
```

```
    Item item = new Item(p,qty);
    itsItems.add(item);
  }

  public int total()
  {
    try
    {
      int total = 0;
      for (int i = 0; i < itsItems.size(); i++)
      {
        Item item = (Item) itsItems.elementAt(i);
        Product p = item.getProduct();
        int qty = item.getQuantity();
        total += p.getPrice() * qty;
      }
      return total;
    }
    catch (Exception e)
    {
      throw new Error(e.toString());
    }
  }
}
```

我必须向 OrderImp 中增加一些异常处理，因为 Product 接口会抛出异常。我对所有这些异常感到很沮丧。接口背后的代理实现不应该对接口造成影响，但是代理抛出的异常却通过接口传播了出去。所以我决定把所有的 Exceptions 改为 Errors，这样我就不必用 throws 子句来污染接口，也不必用 try/catch 块来污染这些接口使用者。

如何在代理中实现 addItem 方法呢？显然，代理不能委托给 OrderImp.addItem！相反，代理必须要在数据库中插入一个 Item 行。另一方面，我非常想把 OrderProxy.total 委托给 OrderImp.total，因为我想把业务规则（也就是计算总价的策略）封装在 OrderImp 中。创建代理完全就是为了分离数据库实现和业务规则。

为了委托 total 函数，代理必须要构建完整的 Order 对象以及它所包含的所有 Item。因此，在 OrderProxy.total 中，我们必须从数据库中读入所有的 Item，把找到的每个 Item 都加入到一个空的 OrderImp 中（通过调用其 addItem 方法），然后调用这个 OrderImp 的 total 方法。这样，OrderProxy 的实现看上去应该像程序 26.20。

程序 26.20 OrderProxy.java

```
import java.sql.SQLException;

public class OrderProxy implements Order
{
  private int orderId;

  public OrderProxy(int orderId)
  {
    this.orderId = orderId;
  }

  public int total()
  {
    try
```

```
    {
      OrderImp imp = new OrderImp(getCustomerId());
      ItemData[] itemDataArray = DB.getItemsForOrder(orderId);
      for (int i = 0; i < itemDataArray.length; i++)
      {
        ItemData item = itemDataArray[i];
        imp.addItem(new ProductProxy(item.sku), item.qty);
      }
      return imp.total();
    }
    catch (Exception e)
    {
      throw new Error(e.toString());
    }
  }

  public String getCustomerId()
  {
    try
    {
      OrderData od = DB.getOrderData(orderId);
      return od.customerId;
    }
    catch (SQLException e)
    {
      throw new Error(e.toString());
    }
  }

  public void addItem(Product p, int quantity)
  {
    try
    {
      ItemData id = new ItemData(orderId, quantity, p.getSku());
      DB.store(id);
    }
    catch (Exception e)
    {
      throw new Error(e.toString());
    }
  }

  public int getOrderId()
  {
    return orderId;
  }

}
```

这意味着还需要一个 ItemData 类和几个操作 ItemData 行的 DB 函数。程序 26.21 至 26.23 中展示了它们。

程序 26.21 ItemData.java

```
public class ItemData
{
  public int orderId;
  public int qty;
  public String sku = "junk";
```

```
  public ItemData()
  {
  }

  public ItemData(int orderId, int qty, String sku)
  {
    this.orderId = orderId;
    this.qty = qty;
    this.sku = sku;
  }

  public boolean equals(Object o)
  {
    ItemData id = (ItemData)o;
    return orderId == id.orderId &&
           qty == id.qty &&
           sku.equals(id.sku);
  }
}
```

程序 26.22 DBTest.java

```
public void testStoreItem() throws Exception
{
    ItemData storedItem = new ItemData(1, 3, "sku");
    DB.store(storedItem);
    ItemData[] retrievedItems = DB.getItemsForOrder(1);
    assertEquals(1, retrievedItems.length);
    assertEquals(storedItem, retrievedItems[0]);
}

public void testNoItems() throws Exception
{
    ItemData[] id = DB.getItemsForOrder(42);
    assertEquals(0, id.length);
}
```

程序 26.23 DB.java

```
public static void store(ItemData id) throws Exception
{
    PreparedStatement s = buildItemInsersionStatement(id);
    executeStatement(s);
}

private static PreparedStatement
buildItemInsersionStatement(ItemData id) throws SQLException
{
    PreparedStatement s = con.prepareStatement(
                "Insert into Items(orderId,quantity,sku)" +
                " VALUES (?, ?, ?);");
    s.setInt(1,id.orderId);
    s.setInt(2,id.qty);
    s.setString(3, id.sku);
    return s;
}

public static ItemData[] getItemsForOrder(int orderId) throws Exception
{
```

```
    PreparedStatement s = buildItemsForOrderQueryStatement(orderId);
    ResultSet rs = s.executeQuery();
    ItemData[] id = extractItemDataFromResultSet(rs);
    rs.close();
    s.close();
    return id;
}

private static PreparedStatement
buildItemsForOrderQueryStatement(int orderId) throws SQLException
{
    PreparedStatement s = con.prepareStatement(
                "SELECT * FROM Items WHERE orderid = ?;");
    s.setInt(1, orderId);
    return s;
}

private static ItemData[] extractItemDataFromResultSet(ResultSet rs)
                throws SQLException
{
    LinkedList l = new LinkedList();
    for (int row = 0; rs.next(); row++)
    {
      ItemData id = new ItemData();
      id.orderId = rs.getInt("orderid");
      id.qty = rs.getInt("quantity");
      id.sku = rs.getString("sku");
      l.add(id);
    }
    return (ItemData[]) l.toArray(new ItemData[l.size()]);
}

public static OrderData getOrderData(int orderId) throws SQLException
{
    PreparedStatement s = con.prepareStatement(
                "Select cusid from orders where orderid = ?;");
    s.setInt(1, orderId);
    ResultSet rs = s.executeQuery();
    OrderData od = null;
    if (rs.next())
      od =  new OrderData(orderId, rs.getString("cusid"));
    rs.close();
    s.close();
    return od;
}
```

26.1.2 PROXY 模式总结

这个例子应该已经消除了所有关于使用代理是简单和优雅的错误认识。使用代理是有代价的。规范模式中所隐含的简单委托模型很少能够被优美地实现。相反，我们经常会取消对于繁琐的获取和设置方法的委托。对于那些管理 1:N 关系的方法来说，我们会推迟委托并把它移到其他方法中，就像把对 addItem 的委托移到 total 方法中一样。最后，我们还要面临缓存的困扰。

在本例中，我们没有进行任何缓存。所有的测试都在一秒内运行完成，所以无需过多的担心性能问题。但是在真实的应用程序中，很可能就要考虑性能问题并且很可能会需要智能缓存机制。我不赞成因为惧怕如果没有缓存会导致性能降低，而机械地去实现一个缓存策略的做法。事实上，我

已经发现过早地加入缓存反而会有效的降低性能。如果你担心性能可能是个问题，我建议你做一些试验去证明它确实是一个问题。当且仅当得到证实时，你才应该考虑如何去提速。

PROXY 模式的好处

虽然代理会带来很多讨厌的问题，但是它们具有一个非常大的好处：重要关系的分离（separation of concerns）。在我们的例子中，业务规则和数据库就被完全分开了。OrderImp 对于数据库没有任何依赖。如果想更改数据库模式或者数据库引擎，我们可以在不影响 Order、OrderImp 以及任何其他业务领域类的情况下完成它。

在那些把业务规则和数据库实现分离显得非常重要的情况中，PROXY 模式是很适用的。就此而言，PROXY 模式可以用来分离业务规则和任何种类的实现问题。它可以用来防止业务规则被诸如：COM、CORBA、EJB 等东西污染。这是当前流行的一种保持项目的业务规则逻辑和实现机制分离的方法。

26.1.3　处理数据库、中间件以及其他第三方接口

软件工程师在实际工作中肯定会用到第三方 API。我们会购买数据库引擎、中间件引擎、类库、线程库等等。最初，我们通过在应用程序中直接调用这些 API 的方式去使用它们（参见图 26.5）。

然而，随着时间的推移，我们发现我们的应用程序已经越来越多地被这样的 API 调用污染了。例如，在一个数据库应用程序中，我们会发现越来越多的 SQL 字符串把那些同样也包含业务规则的代码弄的一团糟。

当第三方 API 发生变化时，这就变成了问题。对数据库应用来说，当数据库模式发生变化时，它也会成为问题。随着新版本的 API 或者数据库模式的发布，越来越多的应用程序代码需要重写去适应这些变化。

最后，开发者决定必须要把这些变化隔离起来。因此他们就想出了用一个层来隔离应用业务规则和第三方 API（参见图 26.6）。他们把所有使用第三方 API 的代码以及所有和 API 而不是应用的业务规则有关的概念都集中到这个层中。

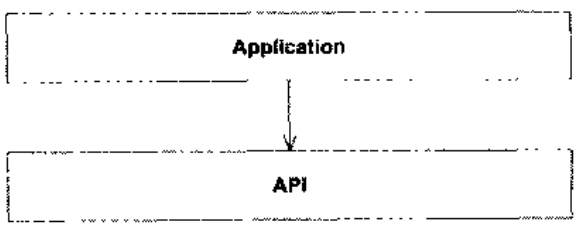


图 26.5　应用程序和第三方 API 之间的最初关系

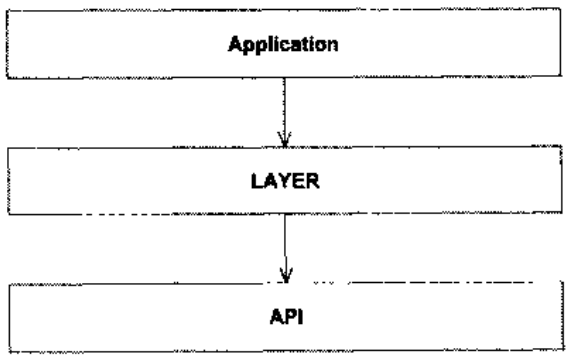


图 26.6　引入一个隔离层

这种层往往可以购买。ODBC 或者 JDBC 就是这样的层。它们分离了应用程序代码和实际的数据库引擎。当然，它们本身也是第三方 API，所以，应用甚至可能也需要和它们分离。

请注意，Application 和 API 之间有一个传递依赖关系。在某些应用程序中，这个间接的依赖关系仍足以引起问题。例如，JDBC 就没有把应用和数据库模式的细节隔离。

为了更好地隔离，我们需要倒置应用程序和该层之间的依赖关系（参见图 26.7）。这就使得应用程序对于第三方 API 没有任何依赖，不管是直接的还是间接的。在使用数据库的应用中，它使得应用程序无需直接知道数据库模式的知识。在使用中间件引擎的应用中，它使得应用程序无需知道任何有关中间件处理器所使用的数据类型。

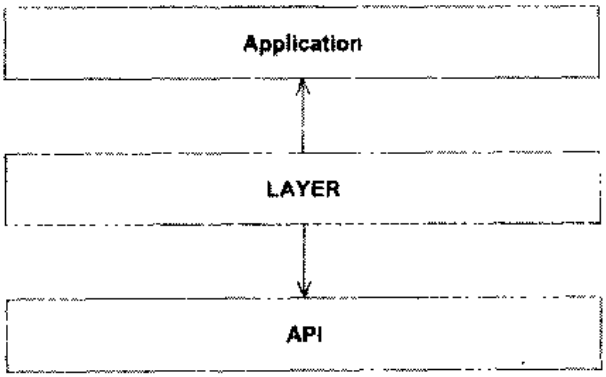


图 26.7　倒置应用程序和层之间的依赖关系

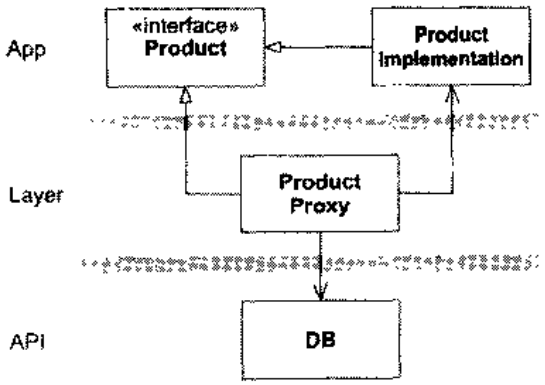


图 26.8　PROXY 模式是如何倒置应用程序和层之间的依赖关系的

PROXY 模式正好可以实现这种形式的依赖关系。应用程序完全没有依赖于代理。相反，代理依赖于应用程序以及 API。这就把所有关于应用程序和 API 之间的映射关系的知识都集中到代理中。

这种对知识的集中意味着代理会成为噩梦。每当 API 改变时，代理就得改变。每当应用程序改变时，代理也要改变。代理会变得非常难以处理。

知道噩梦会出现在哪里是件好事。如果没有代理，噩梦就会遍布到应用程序代码的各个地方。

大多数应用程序不需要代理。代理是一个非常重型的解决方案。当我看见使用了代理的解决方案时，在大多数情况下，我都会建议去掉它们，并使用简单一些的方案。但是存在一些情况，其中代理所提供的应用程序和 API 的极端分离是有益的。这些情况几乎总是出现在那些遭受着频繁的数据库模式或者 API 变更的非常大型的系统中；或者出现在可以运行在许多不同的数据库引擎或者中间件引擎之上的系统中。

26.2　STAIRWAY TO HEAVEN 模式[①]

STAIRWAY TO HEAVEN 模式是另一个可以完成和 Proxy 模式一样的依赖关系倒置的模式。它使用了类形式（class form）的 ADAPTER 模式的一个变体（参见图 26.9）。

PersistentObject 是一个知道数据库的抽象类。它提供了两个抽象方法：read 和 write。它同时还提供了一组实现方法作为实现 read 和 write 所需要的工具。例如，在 PersistentProduct 的 read 和 write 的实现中，会使用这些工具把 Product 的所有数据字段从数据库中读出或者写入到数据库。同样，在 PersistentAssembly 的 read 和 write 的实现中，会对 Assembly 中对额外字段做相同的工作。它从

① [Martin97]。

PersistentProduct 中继承了读写 Product 字段的能力并且把 read 和 write 方法组织成可以利用这个能力的形式。

这个模式只能在支持多重继承的语言中使用。请注意，PersistentProduct 和 PersistentAssembly 都继承自两个已经实现的基类。此外，PersistentAssembly 和 Product 之间是一个菱形继承关系。在 C++中，我们使用虚继承来避免 PersistentAssembly 继承了 Product 的两个实例。

对虚继承或者是其他语言中类似关系的需要，意味着这个模式会带来一些干扰。虽然它和 Product 层次结构纠结在一起，但是带来的干扰却是极小的。

这个模式的好处是它把有关数据库的知识和应用程序的业务规则完全分离开来。应用程序中那些需要调用 read 和 write 的少量代码可以使用下面的应急方法：

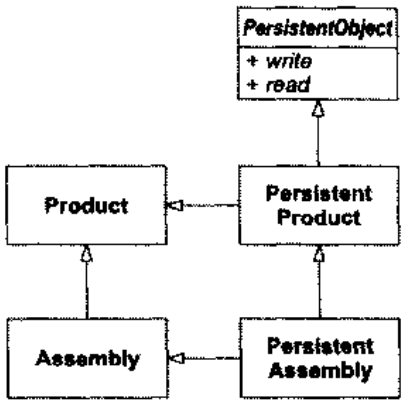


图 26.9　STAIRWAY TO HEAVEN 模式

```
PersistentObject* o = dynamic_cast<PersistentObject*>(product);
if(o)
    o->write();
```

换句话说，我们询问应用对象是否符合 PersistentObject 接口，如果符合，我们就调用 read 或者 write。这就使得应用程序中不需要知道读写逻辑的部分完全独立于层次结构中有关 PersistentObject 的部分。

26.2.1　STAIRWAY TO HEAVEN 模式的例子

程序 26.24 至 26.34 展示了一个使用 C++语言的 STAIRWAY TO HEAVEN 模式实例。像往常一样，最好的做法是从测试用例开始。如果完整地展示 CppUnit[①]，会有些冗长，所以我在程序 26.24 中只包含了测试用例方法。第一个测试用例证明了 PersistentProduct 可以在系统中作为 Product 传送，然后再被转换成一个 PersistentObject 并被随意写入。我们假设 PersistentProduct 会把自己写成一个简单的 XML 格式。第二个测试用例对 PersistentAssembly 所做的证明和第一个测试用例相同，惟一不同之处是 Assembly 对象中多了第二个字段

程序 26.24　ProductPersistenceTestCase.cpp（节选）

```
void ProductPersistenceTestCase::testWriteProduct()
{
  ostrstream s;
  Product* p = new PersistentProduct("Cheerios");
  PersistentObject* po = dynamic_cast<PersistentObject*>(p);
  assert(po);
  po->write(s);
  char* writtenString = s.str();
  assert(strcmp("<PRODUCT><NAME>Cheerios</NAME></PRODUCT>",
                writtenString) == 0);
}
```

① XUnit 家族中的一个单元测试框架。要获取更多的信息可以参见 www.junit.org 和 www.xprograming. com。

```
void ProductPersistenceTestCase::testWriteAssembly()
{
  ostrstream s;
  Assembly* a = new PersistentAssembly("Wheaties", "7734");
  PersistentObject* po = dynamic_cast<PersistentObject*>(a);
  assert(po);
  po->write(s);
  char* writtenString = s.str();
  assert(strcmp("<ASSEMBLY><NAME>Wheaties"
                "</NAME><ASSYCODE>7734</ASSYCODE></ASSEMBLY>",
                writtenString) == 0);
}
```

紧接着，在程序 26.25 至 26.28 中，可以看到 Product 和 Assembly 的定义和实现。为了节省空间，我们的例子中的这些类几乎都是退化的。在正常的应用程序中，这些类会包含有实现业务规则的方法。请注意，这些类中都没有持久化的迹象。没有任何从业务规则到持久化机制的依赖关系。这就是该模式的关键所在。

虽然 STAIRWAY TO HEAVEN 模式具有良好的依赖关系特征，但是程序 26.27 中却有一个完全是因为该模式而人为添加的东西。Assembl 在继承 Product 时使用了 virtual 关键字。为了避免 PersistentAssembly 对 Product 的重复继承，这是必需的。如果回顾一下图 26.9 的话，就会看到 Product 是包括 Assembly、PersistentProduct 以及 PersistentObject 在内的菱形[①]继承关系的顶点。为了避免对 Product 的重复继承，它必须被虚拟继承。

程序 26.25　product.h

```
#ifndef STAIRWAYTOHEAVENPRODUCT_H
#define STAIRWAYTOHEAVENPRODUCT_H

#include <string>

class Product
{
 public:
  Product(const string& name);
  virtual ~Product();
  const string& getName() const {return itsName;}
 private:
  string itsName;
};

#endif
```

程序 26.26　product.cpp

```
#include "product.h"

Product::Product(const string& name)
  : itsName(name)
{
}

Product::~Product()
{
}
```

① 有时开玩笑地称之为“可怕的死亡菱形”。

程序 26.27 assembly.h

```
#ifndef STAIRWAYTOHEAVENASSEMBLY_H
#define STAIRWAYTOHEAVENASSEMBLY_H

#include <string>
#include "product.h"

class Assembly : public virtual Product
{
 public:
  Assembly(const string& name, const string& assyCode);
  virtual ~Assembly();

  const string& getAssyCode() const {return itsAssyCode;}
 private:
  string itsAssyCode;
};

#endif
```

程序 26.28 assembly.cpp

```
#include "assembly.h"

Assembly::Assembly(const string& name, const string& assyCode)
  :Product(name), itsAssyCode(assyCode)
{
}

Assembly::~Assembly()
{
}
```

程序 26.29 和程序 26.30 展示了 PersistentObject 的定义和实现。请注意，尽管 PersistentObject 对 Product 层次结构一无所知，但是它知道如何去写 XML。至少，它知道在写一个对象时，应该先写头，然后是字段，再接着是尾。

PersistentObject 的 write 方法使用 TEMPLATE METHOD 模式[①]控制它的所有派生类的写操作。所以，STAIRWAY TO HEAVEN 模式中的持久化部分使用了 PersistentObject 基类的功能。

程序 26.29 persistentOjbect.h

```
#ifndef STAIRWAYTOHEAVENPERSISTENTOBJECT_H
#define STAIRWAYTOHEAVENPERSISTENTOBJECT_H

#include <iostream>

class PersistentObject
{
 public:
  virtual ~PersistentObject();
  virtual void write(ostream&) const;

 protected:
```

① 参见第 14 章：TEMPLATE METHOD 模式与 STRATEGY 模式：继承与委托。

```
 virtual void writeFields(ostream&) const = 0;

private:
 virtual void writeHeader(ostream&) const = 0;
 virtual void writeFooter(ostream&) const = 0;
};

#endif
```

程序 26.30　persistentObject.cpp

```
#include "persistentObject.h"

PersistentObject::~PersistentObject()
{
}

void PersistentObject::write(ostream& s) const
{
 writeHeader(s);
 writeFields(s);
 writeFooter(s);
 s << ends;
}
```

程序 26.31 和程序 26.32 展示了 PersistentProduct 的实现。这个类实现了 writeHeader、writeFooter 以及 writeFields 函数为 Product 创建一个合适的 XML 文件。它从 Product 中继承了字段和访问方法，并且受控于它的基类 PersistentObject 的 write 方法。

程序 26.31　persistentProduct.h

```
#ifndef STAIRWAYTOHEAVENPERSISTENTPRODUCT_H
#define STAIRWAYTOHEAVENPERSISTENTPRODUCT_H

#include "product.h"
#include "persistentObject.h"

class PersistentProduct : public virtual Product, public PersistentObject
{
public:
 PersistentProduct(const string& name);
 virtual ~PersistentProduct();

protected:
 virtual void writeFields(ostream& s) const;

private:
 virtual void writeHeader(ostream& s) const;
 virtual void writeFooter(ostream& s) const;
};

#endif
```

程序 26.32　persistentProduct.cpp

```
#include "persistentProduct.h"

PersistentProduct::PersistentProduct(const string& name)
:Product(name)
```

```
{
}

PersistentProduct::~PersistentProduct()
{
}

void PersistentProduct::writeHeader(ostream& s) const
{
  s << "<PRODUCT>";
}

void PersistentProduct::writeFooter(ostream& s) const
{
  s << "</PRODUCT>";
}

void PersistentProduct::writeFields(ostream& s) const
{
  s << "<NAME>" << getName() << "</NAME>";
}
```

最后，程序 26.33 和程序 26.34 展示了 PersistentAssembly 是如何把 Assembly 和 PersistentProduct 合成一体的。就像 PersistentProduct 一样，它覆写了 writeHeader、writeFooter 以及 writeFields。不过，在 writeFields 的实现中，调用了 PersistentProduct::writeFileds。因此，它从 PersistentProduct 中继承了写 Assembly 中 Product 部分的能力，并且从 Assembly 中继承了 Product 和 Assembly 的字段和访问方法。

程序 26.33 persistentAssembly.h

```
#ifndef STAIRWAYTOHEAVENPERSISTENTASSEMBLY_H
#define STAIRWAYTOHEAVENPERSISTENTASSEMBLY_H

#include "assembly.h"
#include "persistentProduct.h"

class PersistentAssembly : public Assembly, public PersistentProduct
{
 public:
  PersistentAssembly(const string& name, const string& assyCode);
  virtual ~PersistentAssembly();

 protected:
  virtual void writeFields(ostream& s) const;

 private:
  virtual void writeHeader(ostream& s) const;
  virtual void writeFooter(ostream& s) const;
};

#endif
```

程序 26.34 persistentAssembly.cpp

```
#include "persistentAssembly.h"

PersistentAssembly::PersistentAssembly(const string& name,
                                       const string& assyCode)
  :Assembly(name, assyCode), PersistentProduct(name), Product(name)
{
```

```
}

PersistentAssembly::~PersistentAssembly()
{
}

void PersistentAssembly::writeHeader(ostream& s) const
{
  s << "<ASSEMBLY>";
}

void PersistentAssembly::writeFooter(ostream& s) const
{
  s << "</ASSEMBLY>";
}

void PersistentAssembly::writeFields(ostream& s) const
{
  PersistentProduct::writeFields(s);
  s << "<ASSYCODE>" << getAssyCode() << "</ASSYCODE>";
}
```

26.2.2 结论

我们已经看到，在许多不同的情形下使用 STAIRWAY TO HEAVEN 模式都有很好的结果。这个模式相对比较容易创建并且对包含业务规则的对象具有最小的影响。但是，它需要一种实现了对多重继承支持的语言，比如：C++。

26.3 可以用于数据库的其他模式

26.3.1 Extension Object 模式

假定一个扩展对象（extension object）知道如何把被扩展的对象写入数据库中。为了写入这种对象，你会向它请求一个和“database”键值匹配的扩展对象，把它转型为 DatabaseWriterExtension，然后调用 write 函数。

```
Product p = /* some function that return a Product */
ExtensionObject e = p.getExtension("Database");
if(e != null)
{
    DatabaseWriterExtension dwe = (DatabaseExtension) e;
    e.write();
}
```

26.3.2 Visitor 模式[①]

假定一个访问者（visitor）类层次结构知道如何把被访问的对象写入数据库中。你会通过创建一个合适类型的访问者，然后调用要被写入对象的 accept 方法来把对象写入到数据库中。

① 参见 28.2 中的“VISITOR 模式”。

```
Product p = /* some function that returns a Product */
DatabaseWriteVisitor dwv = new DatabaseWriteVisitor();
p.accept(dwv);
```

26.3.3 Decorator[①]

有两种使用装饰者（decorator）实现数据库的方法。你可以装饰一个业务对象并赋予它 read 和 write 方法；或者可以装饰一个知道如何读写自身的数据对象并赋予它业务规则。后一种方法在使用面向对象数据库时是很常用的。可以把业务规则放到 OODB 模式（schema）之外并且通过装饰者把它加进来。

26.3.4 Facade

这是我最喜欢的出发点。它简单有效。不好的一面是，它把业务规则对象和数据库耦合在了一起。图 26.10 展示了它的结构。DatabaseFacade 类只是提供了读写所有必要对象的方法。这就把对象和 DatabaseFacade 互相耦合到了一起。对象知道 facade 因为它们经常调用 read 和 write 函数。Facade 知道对象因为它必须使用对象的访问方法和改变属性的方法（mutator）去实现 read 和 write 函数。

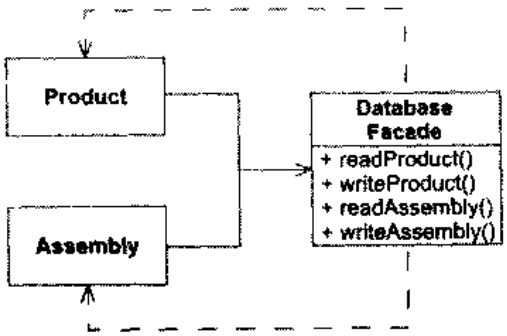


图 26.10 Database Facade

该耦合在稍大一些应用程序中会引起很多问题；但是在较小的或者刚刚开始的应用程序中，它却是一项非常有效的技术。如果在开始时使用了 facade，后来决定改变到一个可以减小耦合的其他模式，facade 也是非常易于重构的。

26.4 结　论

远在真正需要 PROXY 模式或者 STAIRWAY TO HEAVEN 模式前，就去预测对于它们的需要是非常有诱惑力的。但这几乎从来都不是一个好主意，特别是对于 Proxy 模式。我建议在开始时先使用 FACADE 模式，然后在必要时进行重构。如果这样做的话，就会为自己节省时间并省去麻烦。

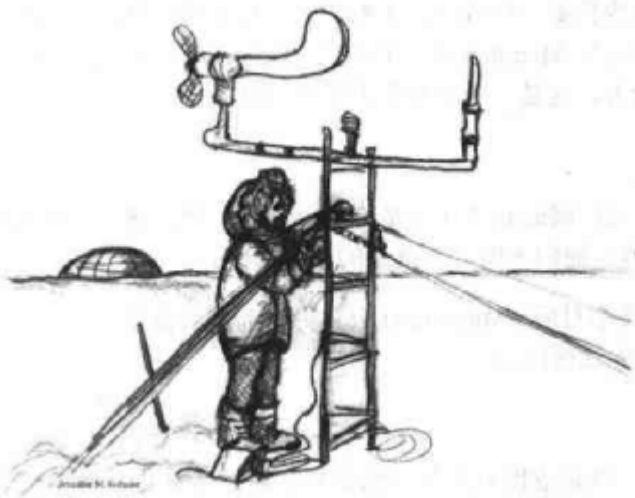
参考文献

1. Gamma.et al. *Design Patterns*.MA:Addison-Wesley,1995
2. Martin.Robert C.Design Patterns for Dealing with Dual Inheritance.C++ *Report*（April）:1997

① 参见 28.4 中的“DECORATOR 模式”。

第 27 章　案例研究：气象站

本章与 Jim Newkirk 合作完成



虽然下面的内容是虚构的，但是你仍会发现其中有许多和你自己的经历相似的内容。

27.1　Cloud 公司

在过去的几年中，Cloud 公司在供工业使用的天气监控系统（WMS）领域一直处于领先地位。作为其旗舰产品的 WMS 可以跟踪温度、湿度、气压、风速以及风向，等等。系统把测量结果实时地显示在一个显示器上。此外，系统也以小时和天为单位保存了有关天气情况的历史信息。用户可以在显示器上察看这些历史信息。

Cloud 公司的主要客户是航空、海运、农业以及广播行业。对于这些行业来说，WMS 是完成重要工作的应用程序。Cloud 公司在构建安装于较难控制环境中的高可靠性产品方面有很好的声誉。

这些系统的高昂价格使 Cloud 公司失去了那些不需要，或者买不起它们的高可靠性系统的客户。Cloud 公司的管理者认为这是一个很大的潜在市场，并且想要开辟这个市场。

1．问题

Cloud 公司的一个竞争对手 Microburst 公司声称，他们拥有一条产品线，客户可以先使用其中的低端产品，并且可以逐渐升级至更高的可靠性。这种威胁可能会使 Cloud 公司丧失虽然数量不多但是却在不断增长的客户。这些客户在增长至有能力使用 Cloud 公司产品的规模时，他们已经正在使用 Microburst 公司的产品了。

还有更大的威胁，Microburst 公司自称他们的产品具有和高端产品互联的能力。也就是说，可以把高端的升级产品通过网络连接在一起形成一个广域的天气监控系统。这种威胁会侵蚀 Cloud 公司的当前客户。

2. 对策

虽然 Microburst 公司已经在展览会上成功地演示了其低端产品，但是在至少 6 个月内，他们的产品还无法批量发售。这意味着 Microburst 公司可能还有一些工程或者产品方面的问题没有解决。此外，Microburst 公司目前还无法提供其承诺的产品线中的高可靠性的升级产品。看来，Microburst 公司过早地宣布了其产品。

如果 Cloud 公司宣布一款具有升级和互联能力的低端产品，并在 6 个月内发售的话，那么他们也许能够赢得，或者至少拖延一些客户，不然的话，这些客户就会去购买 Microburst 公司的产品。通过延迟市场的启动从而使 Microburst 公司丧失一部分订单，可能会削弱 Microburst 公司解决其工程和制造方面问题的能力，这是一个非常令人期望的结果。

3. 左右为难

建立一条低成本、可扩展的新产品线是一项巨大的工程。硬件工程师断然拒绝了 6 个月的开发限期。他们认为 12 个月后他们才能看到批量的产品部件。

市场管理者认为，12 个月后，Microburst 公司的产品就会批量发售，并会赢得部分 Cloud 公司的客户，而且这些客户是无法挽回的。

4. 计划

Cloud 公司的管理者决定立即宣布他们的新产品线，并且开始接受在 6 个月内发货的订单。他们把新产品命名为 Nimbus-LC 1.0。他们计划把原先昂贵的、高可靠性的硬件重新包装进一个具有精美 LCD 触摸屏的外壳中。这些装置具有很高的制造成本，所以每卖出一件产品，公司其实都要亏损。

同时，硬件工程师开始开发真正的低成本硬件，这需要 12 个月的时间。具有这种配置的产品被称为 Nimbus-LC 2.0。当该产品可以批量生产时，Nimbus-LC 1.0 就会被逐渐停产。

当 Nimbus-LC 1.0 的客户希望使用更高级的服务时，无需附加的成本就可以把他的设备替换为 Nimbus-LC 2.0。因此，为了赢得至少是拖延潜在的 Microburst 公司的客户，Cloud 公司甘愿在 6 个月内在这款产品上亏损。

27.1.1 WMS-LC 的软件

Nimbus-LC 项目中的软件部分是复杂的。开发人员必须要创建既能在现有的硬件上，又能在低成本的 2.0 硬件上使用的软件产品。2.0 硬件的原型设备要在 9 个月后才能使用。此外，2.0 电路板上使用的处理器很可能和 1.0 的不同。尽管如此，系统仍然必须要能完全一致的运转，而不管使用的是哪一个硬件平台。

硬件工程师会编写底层的硬件驱动程序，他们需要应用软件工程师去设计这些驱动程序的 API。该 API 必须要在随后的 4 个月内提供给硬件工程师。软件必须要在 6 个月内具备产品化的能力，必须要在 12 个月内能够在 2.0 硬件上运行。软件工程师希望有至少 6 周的时间去熟悉 1.0 设备，这样他们实际的开发时间就只有 20 周。因为 2.0 版本的硬件平台是新的，所以他们需要 8 到 10 周的熟悉时间。这又耗去最初原型到最终发售之间 3 个月的大部分时间。这样，软件工程师就必须要在很短的时间内使新硬件能够工作起来。

软件计划文档

开发和市场人员编写了一些描述 Nimbus-LC 项目的文档：

（1）“Nimbus-LC 需求概述”（参见 27.4 节）：描述了在项目开始时所理解的 Nimbus-LC 系统的操作需求。[1]

（2）“Nimbus-LC 用例”（参见 27.5 节）：描述了从需求中得到的参与者以及用例。

（3）“Nimbus-LC 发布计划”（参见 27.6 节）：描述了软件的发布计划。该计划试图在项目生命周期的初期解决主要的风险，同时确保软件在必需的限期内完成。

语言选择

对语言最重要的限制是可移植性。因为开发时间很短，并且软件工程师熟悉 2.0 硬件的时间甚至更短，所以就要求 1.0 和 2..0 版本使用相同的软件。也就是说，源代码必须是一样的，或者几乎是一样的。如果语言不能满足可移植的约束，那么要在 12 个月的限期内完成 2.0 版本的发布是非常危险的。

还好，除此之外几乎没有其他的限制。软件的规模不是非常大，所以存储空间不是大问题。并且没有短于 1 秒的硬实时限制，所以速度也不是大问题。事实上，由于对于实时的限制非常弱，所以一个具有中等速度垃圾回收机制的语言就是适用的。只有可移植性的限制，而没有任何其他严格的限制，就使得 Java 成为了非常合适的选择。

27.2　Nimbus-LC 软件设计

根据发布计划，第 I 阶段的主要目标是要创建一个可以使大部分的软件独立于它所控制的硬件的构架。事实上，我们希望把气象站应用的抽象行为和它的具体实现分离开来。

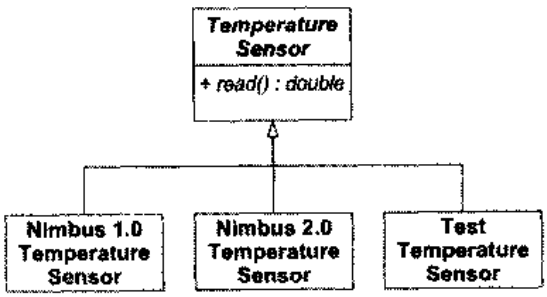


图 27.1　初始的温度传感器设计

例如，不管是哪种硬件配置，软件必须都要能够显示当前的温度。设计如图 27.1 中所示。

抽象类 TemperatureSensor 中有一个多态函数 read()。该基类的派生类可以提供 read()函数的不同实现。

1．测试类

请注意，对两种已知硬件平台中的每一个，都有一个派生类。此外，还有一个名为 TestTemperatureSensor 的特殊派生类。使用该类可以在一台没有连接 Nimbus 硬件的工作站上对软件进行测试。这样，软件工程师就可以在即使没有 Nimbus 系统的情况下为他们的软件编写单元测试和验收测试。

此外，我们必须要在很短的时间内把 Nimbus 2.0 硬件和软件集成在一起。由于时间很短，所以 Nimbus 2.0 版本是有风险的。通过使 Nimbus 软件同时适用于 Nimbus 1.0 以及测试类，我们就可以让 Nimbus 软件在多个平台上运行。这样就减少了向 Nimbus 2.0 移植的风险。

① 我们都知道，在任何软件项目中，需求文档都是最易变化的文档。

测试用例同样也使我们可以测试那些难以在软件中捕获的特性和情况。例如，我们可以让测试类产生一些难以用硬件模拟的故障。

2．定期测量

Nimbus 系统最公共的行为是何时去显示当前的天气监控数据。每一个监测值都以她们自己的特定速率刷新。每一分钟刷新一次温度，而每 5 分钟才刷新一次大气压。显然，我们需要某种调度器来触发这些读数并把它们呈现给使用者。图 27.2 展示了一个可能的结构。

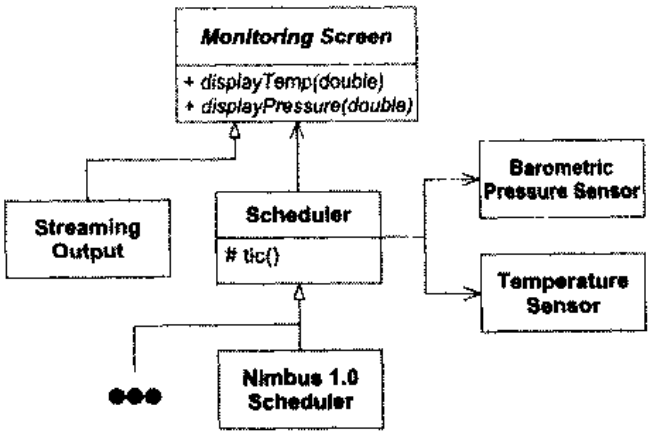


图 27.2　初始的调度和显示构架

我们把 Scheduler 作为具有许多可能实现的基类，针对每个硬件和测试平台，都有一份自己的实现。Scheduler 有一个 tic 函数，该函数期望每 10ms 会被调用一次。派生类要负责进行这个调用（参见图 27.3）。Scheduler 会计算 tic()调用的次数。每到一分钟，它就调用 TemperatureSensor 的 read()函数并把返回的温度值传递给 MonitoringScreen。在第 I 阶段中，我们不需要在 GUI 上显示温度，所以 MonitoringScreen 的派生类只是把结果发送到一个输出流中。

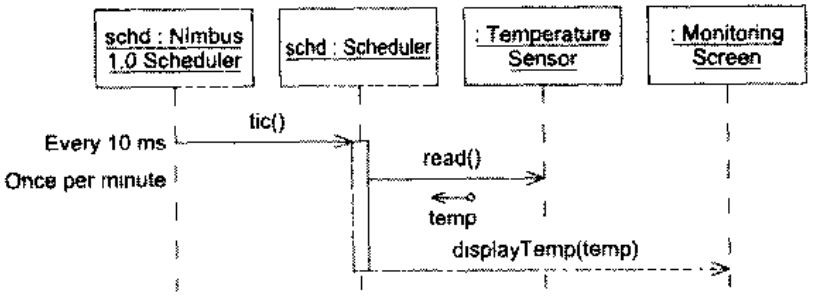


图 27.3　初始的调度器顺序图（sequence diagram）

3．大气压的趋势

需求文档要求我们必须报告出大气压的趋势。这是一个具有 3 个状态的值：上升、下降或者保持稳定。我们怎样来确定该变量的值呢？

根据联邦气象手册[①]，大气压趋势的计算方法如下：

> 如果大气压以每小时 0.06 英寸[②]的速率上升或者下降并且在观测的时刻（每 3 小时进行一次观测）压力的变化合计为 0.02 英寸或者更多，那么就应该报告一次压力变化指示。

我们把这个算法放在哪里呢？如果把它放在 BarometricPressureSensor 类中，那么该类就需要知道每个读数的时间，并且它必须要记录 3 个小时前的一组读数。我们目前的设计没有考虑到这一点。通过给 BarometricPressureSensor 类的 Read 函数增加一个当前时间参数并保证定期地调用这个函数，可以修正这个问题。

① 联邦气象手册 No.1，第 11 章，11.4.6 小节（http://www.nws.noaa.gov）。
② 此处的英寸为气压测量单位，一大气压相当于地面温度为 0 度时的一英寸水银柱的压力——译者著。

不过，这种做法把趋势的计算和使用者更新的频率耦合起来了。这样对用户界面更新方式的更改就有可能会影响到大气压趋势算法。此外，要求必须定期地去读一个传感器才能保证它正确工作的做法也是非常不友好的。我们需要找出一个更好的解决方案。

我们可以让 Scheduler 来记录大气压的历史记录并在需要的时候计算趋势。但是，我们接着也会把温度和风速的历史记录放进 Scheduler 类中吗？每一种新的传感器或者历史记录需求都会导致对 Scheduler 的更改。这样，我们就面临着维护的噩梦。

4. 重新考虑 Scheduler

再看图 27.2。请注意，Scheduler 和每种传感器和用户界面都有连接。当增加更多的传感器和用户界面时，它们必须也要增加到 Scheduler 中。因此，Scheduler 对新传感器和用户界面的增加没有做到封闭。这违反了 OCP。我们希望把 Scheduler 设计为可以独立于传感器以及用户界面的增加和变化。

5. 解除和用户界面之间的耦合

用户界面是容易变化的。客户、市场人员以及几乎所有接触产品的人的突然的想法都可能会导致用户界面的改变。如果说系统中有些部分很可能会遭受需求的严重影响的话，那就是用户界面。因此，我们应该首先解除和它之间的耦合。

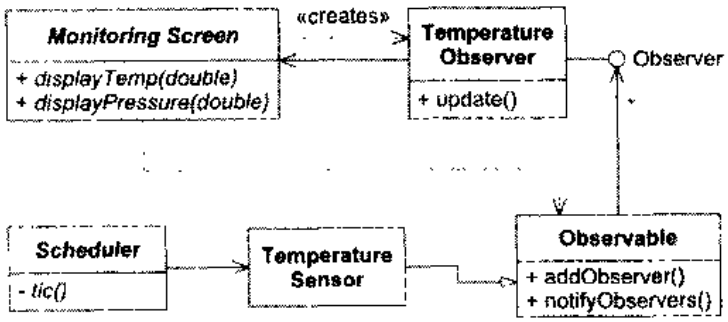


图 27.4　Observer 模式解除了 UI 和 Scheduler 之间的耦合

图 27.4 和图 27.5 中展示了一个使用 OBSERVER 模式的新设计。我们让 UI 依赖于传感器，这样当传感器的读数变化时，就会自动通知 UI。请注意，该依赖关系是间接的。实际的观察者是一个名为 TemperatureObserver 的 ADAPTER[①] 对象。当温度读数变化时，TemperatureSensor 会通知该对象。作为响应，TemperatureObserver 调用 MonitoringScreen 对象的 DisplayTemp 函数。

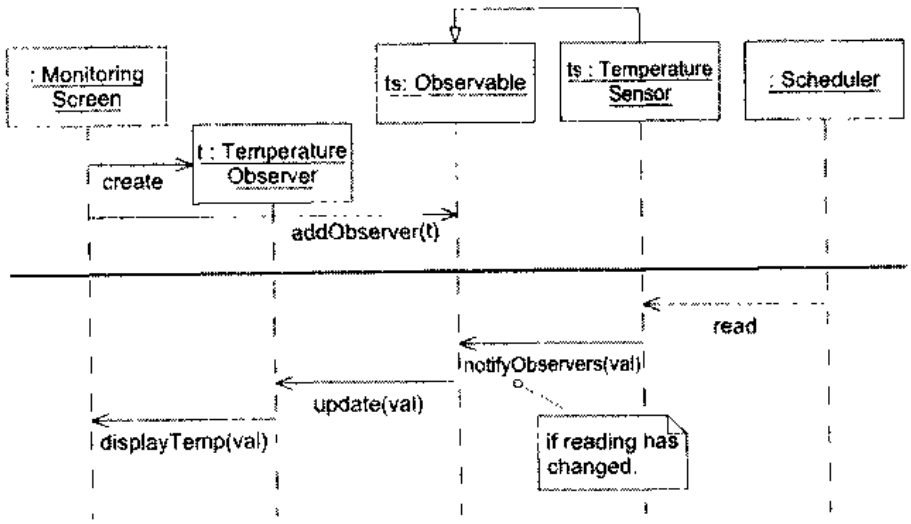


图 27.5　解除和 UI 之间的耦合后的顺序图

① [GOF95]，第 139 页。

这个设计很好地解除了 UI 和 Scheduler 之间的耦合。现在，Scheduler 对 UI 一无所知，而仅仅关注于告诉传感器何时去读。UI 把自己和传感器绑定在一起，以期望传感器向它报告变化。但是，UI 并不知道传感器本身。它只是知道一组实现了 Observable 接口的对象。这样，我们就可以在无需对 UI 的这一部分进行重大更改的情况下去增加传感器。

我们同样已经解决了大气压趋势的问题。现在，可以让一个对 BarometricPressureSensor 进行观察的独立的 BarometricPressureTrendSensor 来计算这个值（参见图 27.6）。

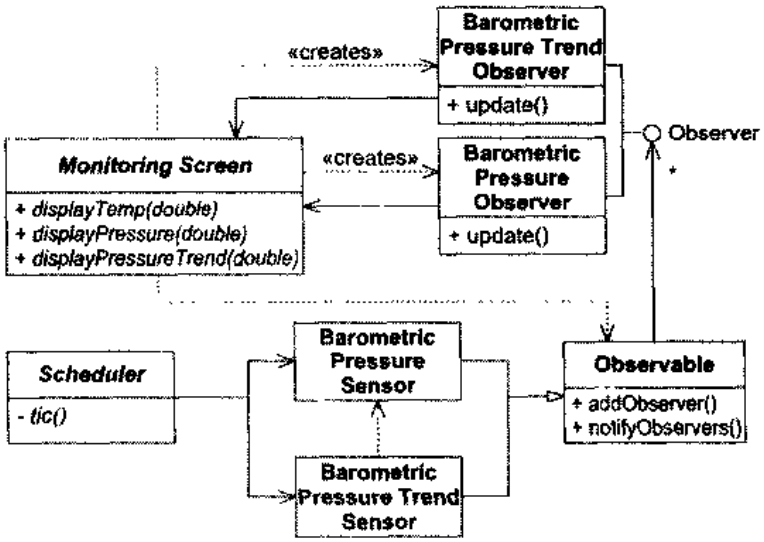


图 27.6　大气压力观察者

6. 再次考虑 Scheduler

Scheduler 的主要任务是告诉每一个传感器它们何时应该去读取一个新值。然而，如果以后的需求迫使我们去增加或者删除一个传感器的话，就需要去更改 Scheduler。事实上，即使我们只是想改变传感器的读取速率，Scheduler 也必须得更改。这令人遗憾地违反了 OCP。看来，传感器的读取速率应该由传感器自己掌握，而不是系统中的任何其他部分。

我们可以使用 Java 类库中的 Listener[①]范例（paradigm）来解除 Scheduler 和传感器之间的耦合。它和 OBSERVER 模式相似，因为你也得注册一些要被通知的东西；但是在本例中，我们希望在某个事件（时间）发生时会通知我们（参见图 27.7）。

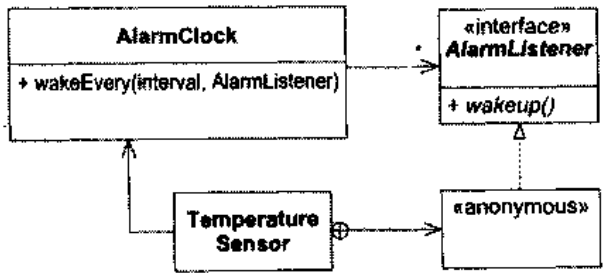


图 27.7　被解除耦合的 AlarmClock

传感器创建了实现 AlarmListener 接口的匿名 ADAPTER 类。接着，传感器把这些适配器注册到 AlarmClock（也就是前面的 Scheduler 类）中。作为注册的的一部分，它们也告诉 AlarmClock 它们希望多长时间被通知一次（例如：每秒或者每 50 毫秒）。当那段时间结束时，AlarmClock 就向适配器发送 wakeup 消息，接着适配器向传感器发送 read 消息。

① [JAVA98]，第 360 页。

这完全改变了 Scheduler 类的本质。在图 27.2 中，它构成了系统的中心并知道大部分的其他组件。但是现在它在系统中完全处于次要位置。它对其他的组件一无所知。它现在只完成一项工作——时间调度（scheduling）——这和天气监控没有任何关系，从而符合了 SRP。事实上，它可以在许多不同种类的应用程序中重用。其实，正是因为改动过大，所以我们才把名字改为 AlarmClock。

7. 传感器的结构

既然已经能够解除了传感器和系统其他部分之间的耦合，我们该来看一下它们的内部结构了。现在传感器要完成 3 个不同功能。首先，它们必须要创建并注册 AlarmListener 的匿名派生类。其次，它们必须要确定它们的读数是否发生了变化并调用 Observable 类的 notifyObservers 方法。第三，它们必须要和 Nimbus 硬件交互以读取适当的值。

图 27.1 展示了这些关系是如何被分离的。图 27.8 中是把该设计和我们已经做的其他更改集成后的结果。TemperatureSensor 基类处理前两个关系，因为它们是通用的。然后，TemperatureSensor 的派生类就可以处理和硬件相关的事务并执行实际的读取工作。

为了分离 TemperatureSensor 的通用部分和特定部分，图 27.8 中使用了 TEMPLATE METHOD 模式。从 TemperatureSensor 的 check 和 read 函数中可以看出该模式的结构。当 AlarmClock 调用匿名类的 wakeup 方法时，这个匿名类就把该调用转发给 TemperatureSensor 的 check 函数。接着，check 函数调用 TemperatureSensor 的抽象函数 read。该函数会被派生类实现为真正地去和硬件交换并获取传感器读数。然后，check 函数确定新的读数是否和上一次的读数不同。如果检测到不同，它就通知正在等待的观察者。

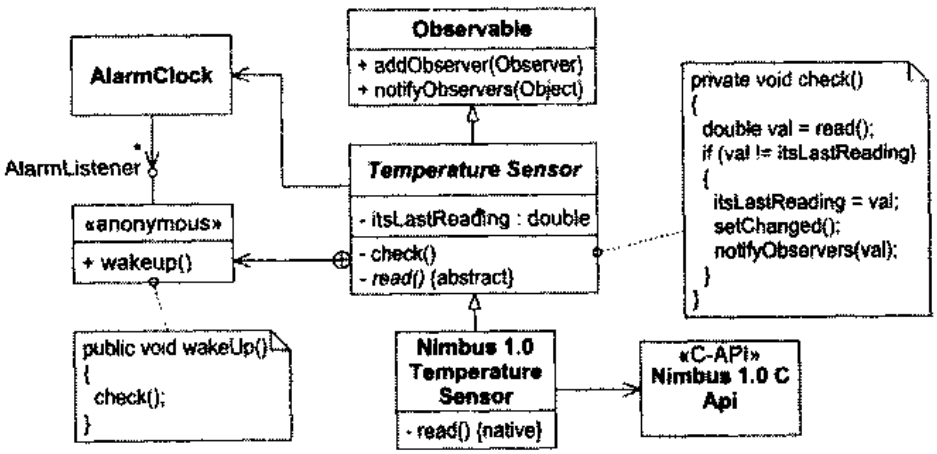


图 27.8　传感器结构

这很好的实现了我们需要的关系分离。对于每个新的硬件或者测试平台，我们都可以创建一个适用于它的 TemperatureSensor 的派生类。此外，那个派生类只需要覆写一个非常简单的函数：read()。传感器其余的功能由它所属的基类来完成。

8. API 在哪里

第 II 阶段的目标之一就是要为 Nimbus 2.0 硬件创建一套新的 API。这套 API 应该是用 Java 编写，可以扩展并且提供简单和直接的对 Nimbus 2.0 硬件的访问。此外，它必须也要适用于 Nimbus 1.0 硬件。如果没有这套 API，那么当采用新的电路板时，我们为这个项目编写的所有简单的调试和校准工具都必须要更改。在我们当前的设计中，这套 API 在哪里呢？

事实上，迄今为止我们所创建的所用东西都无法充当一套简单的 API。我们所期望的东西有点像这样：

```
public interface TemperatureSensor
{
    public double read();
}
```

我们希望编写一些可以直接访问这套 API 的工具，而不必关心注册观察者的逻辑。我们同样也希望在这个层面上传感器不要自动地去轮询（polling）自己，或者和 AlarmClock 交互。我们希望的是某些非常简单并且独立的，可以充当和硬件的直接接口的东西。

我们似乎推翻了前面所有的讨论。毕竟，图 27.1 正是我们要寻找的东西。但是，图 27.1 以后我们所做的更改也是合理的。我们所需要的就是这两种方案中最好部分的结合体。

图 27.9 中使用 BRIDGE 模式把真正的 API 从 TemperatureSensor 中提取出来。该模式的意图是把实现和抽象分离，以便于二者可以独立变化。在我们的例子中，TemperatureSensor 是抽象，而 TemperatureSensorImp 是实现。请注意，“实现”这个词是用来描述抽象接口的，而 “实现”本身则被 Nimbus1.0TemperatureSensor 类实现。

9. 创建问题

请再看一下图 27.9。为了使该图可以工作，就必须要创建一个 TemperatureSensor 对象并把它和一个 Nimbus1.0TemperatureSensor 对象绑定起来。谁来完成这个工作呢？当然，无论软件中哪一部分负责这个工作，它都不会是平台独立的。因为它必须要明确地知道平台相关的 Nimbus1.0TemperatureSensor。

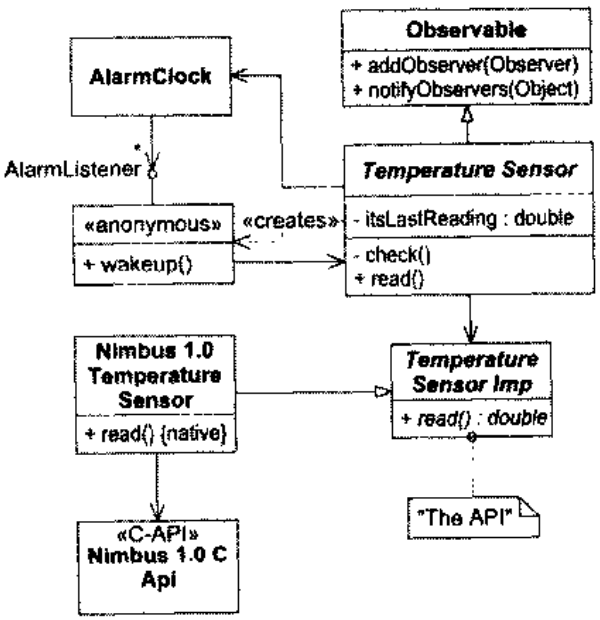


图 27.9　具有 API 的温度传感器

我们可以让主程序来做所有这种工作。程序 27.1 为主程序的一种实现方法。

程序 27.1　WeatherStation

```
public class WeatherStation
{
    public static void main
            (String[] args)
    {
        AlarmClock ac = new Alarm(
            new Nimbus1_0AlarmClcok);

        TemperatureSensor ts =
            new TemperatureSensor(ac,
                new Nimbus1_0TemperatureSensor);

        BarometricPressureSensor bps =
            new BarometricPresssureSensor(ac,
                new Nimbus1_0BarometricPressureSensor);

        BarometricPressureTrend bpt =
            New BarometricPressureTrend(bps)
    }
}
```

这是一个可用的解决方案，但是编写起来却非常的繁杂、丑陋。我们可以改为使用 FACTORIES 来处理大部分和创建有关的杂务。图 27.10 展示了这个结构。

我们把工厂叫做 StationToolkit。这是一个接口，它所定义的方法是用来创建 API 类的实例的。每个平台中都有自己的 StationToolkit 的派生类，并且该派生类会创建出 API 类的合适派生对象。

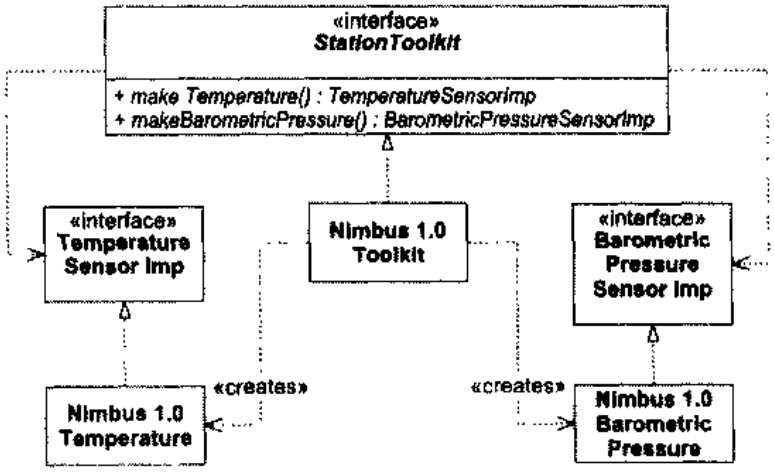


图 27.10　StationToolkit

现在，我们可以重新编写 main 函数，如程序 27.2 所示。请注意，要使主程序运行于另一个平台，我们只需更改创建 Nimbus1.0AlarmClock 和 Nimbus1.0Toolkit 的两行代码。而程序 27.1 中却需要更改它所创建的每一个传感器，这是一个显著的改善。

程序 27.2　WeatherStation

```
public class WeatherStation
{
    public static void main(String[] args)
    {
        AlarmClock ac = new AlarmClock(
            new Nimbus1_0AlarmClcok);

        StationToolkit st = new Nimbus1_0Toolkit();

        TemperatureSensor ts =
            new TemperatureSensor(ac, st);

        BarometricPressureSensor bps =
            new BarometricPresssureSensor(ac, st);

        BarometricPressureTrend bpt =
            New BarometricPressureTrend(bps)
    }
}
```

请注意，StationToolkit 被传递给每一个传感器。这使得传感器可以创建它们自己的实现。程序 27.3 展示了 TemperatureSensor 的构造函数。

程序 27.3　TemperatureSeneor

```
public class TemperatureSensor extends Observable
{
    public TemperatureSensor(AlarmClock ac, StationToolkit st)
    {
```

```
        itsImp = st.makeTemperature();
    }
    private TemperatureSensorImp itsImp;
}
```

10. 让 StationToolkit 创建 AlarmClock

我们可以进一步改善程序，让 StationToolkit 去创建相应的 AlarmClock 派生类。我们再次使用 BRIDGE 模式来分离针对天气监控应用的 AlarmClock 抽象和支持硬件平台的实现。

图 27.11 展示了新的 AlarmClock 结构。现在，AlarmClock 通过它的 ClockListener 接口来接收 tic()消息。这些消息是由 API 中 AlarmClockImp 类适当的派生类发送的。

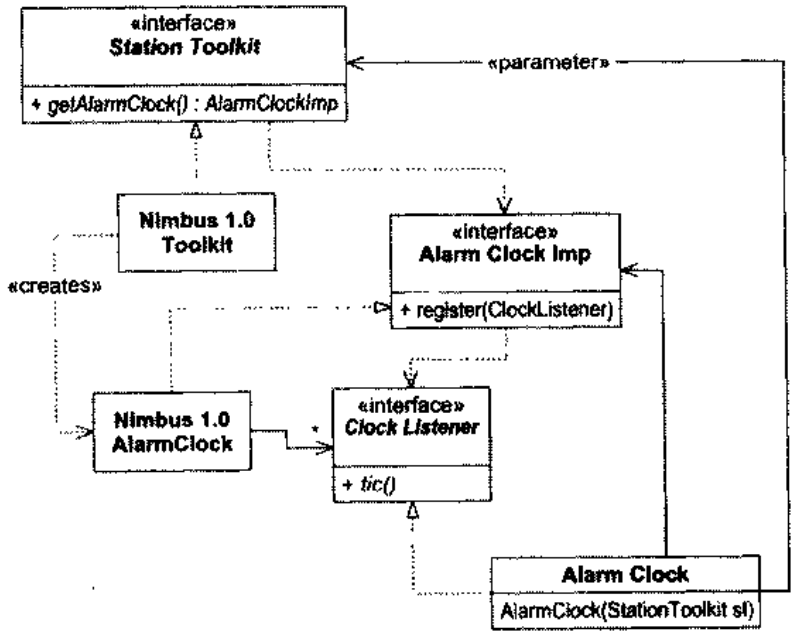


图 27.11　StationToolkit 和 AlarmClock

图 27.12 中展示了 AlarmClock 的创建过程。适当的 StationToolkit 派生对象被传递给 AlarmClock 的构造函数。AlarmClock 使用它创建出合适的 AlarmClockImp 的派生对象。该对象被传回给 AlarmClock，并且 AlarmClock 会向它注册，这样 AlarmClock 就可以接收来自它的 tic()消息。

这再次会影响到程序 27.4 中的主程序。请注意，现在只有一行代码是平台相关的。只要更改那一行代码，整个系统就使用了一个不同的平台。

程序 27.4　WeatherStation

```
public class WeatherStation
{
    public static void main(String[] args)
    {
        StationToolkit st = new Nimbus1_0Toolkit();

        AlarmClock ac = new AlarmClock(st);

        TemperatureSensor ts =
            new TemperatureSensor(ac, st);

        BarometricPressureSensor bps =
```

```
            new BarometricPresssureSensor(ac, st);

        BarometricPressureTrend bpt =
            New BarometricPressureTrend(bps)
    }
}
```

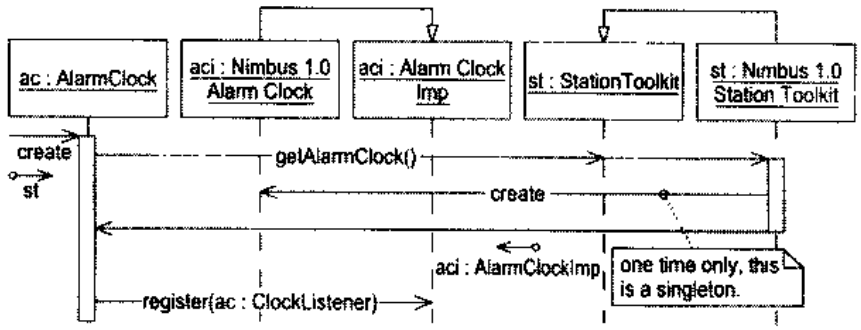


图 27.12　创建 AlarmClock

这相当不错，但是在 Java 中我们可以做得更好。Java 允许我们根据名字创建对象。要使程序 27.5 中的主程序工作于一个新平台，可以不用对它进行更改。只要把 StationToolkit 派生类的名字作为命令行参数传递即可。如果指定的名字正确，那么就会创建出合适的 StationToolkit，并且系统的其余部分都会具有正确的行为。

程序 27.5　WeatherStation

```
public class WeatherStation
{
    public static void main(String[] args)
    {
        try
        {
            Class tkClass = Class.forName(args[0]);
            StationToolkit st = (StationToolkit)tkClass.NewInstance();

            AlarmClock ac = new AlarmClock(st);

            TemperatureSensor ts =
                new TemperatureSensor(ac, st);

            BarometricPressureSensor bps =
                new BarometricPresssureSensor(ac, st);

            BarometricPressureTrend bpt =
                New BarometricPressureTrend(bps)
        }
        catch(Exception e)
    }
}
```

11. 把类放到包中

我们想对该软件的几个部分分别地进行发布（release）和分发（distribute）。API 以及它的每个实现都可以独立于应用程序的其余部分重用并且可以被测试和质量保证团队使用。UI 和传感器应该被分开，这样它们就可以独立变化。毕竟，下一代产品可能会在相同的系统构架上面使用更好的 UI。事实上，版本 II 就是第一个这样的例子。

图 27.13 中展示了第 I 阶段的包结构。这个包结构中几乎没有包含迄今为止我们所设计的类。每个平台都有一个对应的包，这些包中的类都派生自 API 包中的类。API 包惟一的一个客户是 WeatherMonitoringSystem 包，该包中包含了所有其他的类。

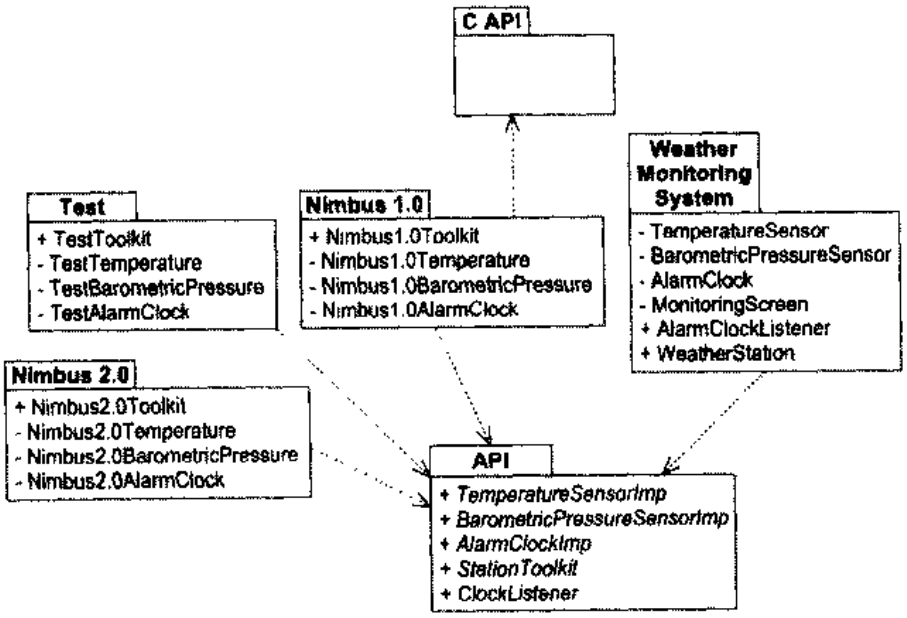


图 27.13　第 I 阶段的包结构

尽管版本 I 的 UI 非常小，但是糟糕的是，它仍然和 WeatherMontoringSystem 混合在一起。把这个类放在单独的包中会更好一些。然而，我们碰到了一个问题。在当前的实现中，WeatherStation 对象创建了 MonitoringScreen 对象，但是 MonitoringScreen 对象为了通过传感器的 Observable 接口增加它的观察者，就必须要知道所有的传感器。因此，如果我们把 MonitoringScreen 拿出来放到自己的包中，那么这个包和 WeatherMonitoringSystem 包之间就会有一个环状的依赖关系。这违反了无环依赖原则（ADP）并且会使这两个包不能相互独立地发布。

把主程序从 WeatherStation 类中剥离出来就可以修正这个问题。WeatherStation 仍然创建 StationToolkit 和所用的传感器，但是它不创建 MonitoringScreen。主程序会创建 MonitoringScreen 以及 WeatherStation。然后，主程序会把 WeatherStation 传递给 MonitoringScreen，这样 MonitoringScreen 就可以把它的观察者增加到传感器中了。

MonitoringScreen 怎样从 WeatherStation 中获取传感器呢？我们需要向 WeatherStation 中增加一些完成这项工作的方法。程序 27.6 中为所增加的方法。

程序 27.6　WeatherStation

```
public class WeatherStation
{
    public WeatherStation(String tkName)
    {
        //create station toolkit and sensors as before.
    }

    public void addTempObserver(Observer o)
    {
        itsTS.addObserver(o);
```

```
    }

    public void addBPObserver(Observer o)
    {
        itsBPS.addObserver(o);
    }

    public void addBPTObserver(Observer o)
    {
        itsBPT.addObserver(o);
    }

    //private variables ...
    private TemperatuerSensor itsTS;
    private BarometricPressureSensor itsBPS;
    private BarometricPressureTrend itsBPT;
}
```

现在，我们可以重新绘制包图，如 27.14 所示。我们忽略了大部分的和 MonitoringScreen 没有关系的包。这看上去很好。UI 完全可以独立于 WeatherMonitoringSystem 而变化。但是，由于 UI 依赖于 WeatherMonitoringSystem，所以每当 WeatherMonitoringSystem 变化时都会导致问题。

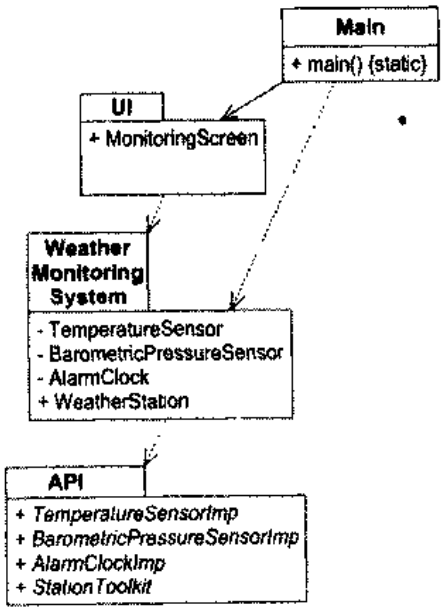


图 27.14　解除了依赖环的包图

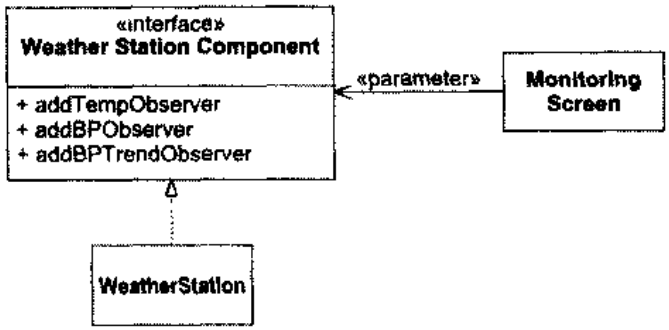


图 27.15　WeatherStation 抽象接口

UI 和 WeatherMonitoringSystem 都是具体的。当一个具体包依赖于另一个具体包时，就违反了依赖倒置原则（DIP）。在本例中，如果 UI 依赖于某个抽象的东西而不是 WeatherMonitoringSystem 就会好一些。

这个问题可以通过创建一个由 MonitoringScreen 使用，而 WeatherStation 从其派生的接口来修正（参见图 27.15）。

现在，如果我们把 WeatherStationComponent 接口放到它自己的包中，就可以得到期望的分离性（参见图 27.16）。请注意，现在 UI 和 WeatherMonitoringSystem 之间完全没有耦合关系。它们都可以独立于对方变化。这是一件好事。

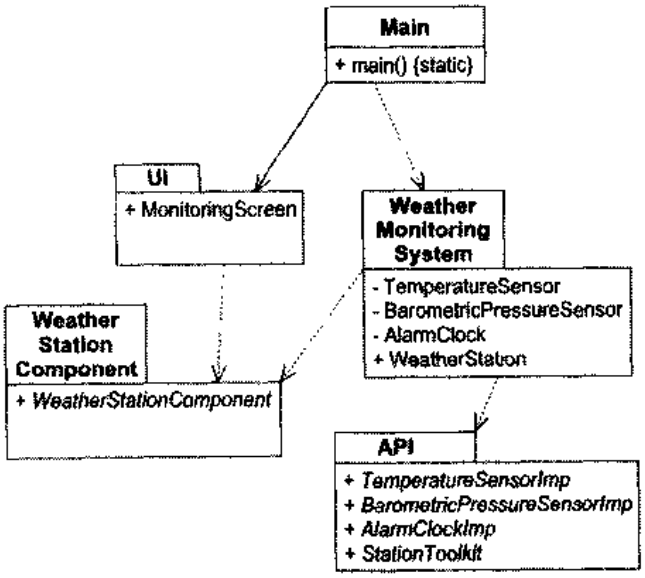


图 27.16　气象站应用组件包图

27.2.1　24 小时历史数据和持久化

版本 1 中要交付的产品小节中的第 4 点和第 5 点（参见 27.6.2.2）谈到了需要保存持久化的 24 小时历史数据。我们知道 Nimbus 1.0 和 Nimbus 2.0 的硬件中都有某种非易失性存储器（NVRAM）。另一方面，测试平台会通过使用磁盘来模拟非易失性存储器。

我们需要创建一个独立于特定平台的持久化机制，并且仍能提供必需的功能。我们同样也需要把该机制和保存 24 小时历史数据的机制连接起来。

显然，低层的持久化机制应该被作为接口定义在 API 包中。这个接口应该采用哪种形式呢？NimbusI 的 C-API 提供了一些可以从非易失性存储器的指定偏移地址读写字节块的调用。这虽然很有效，但是却有些原始（primitive）。还有更好的方法吗？

1. 持久化 API

Java 环境中提供了可以让任何对象直接转换为一串字节的机制。这个过程被称为序列化（serialization）。这样的一串字节可以通过反序列化（deserialization）过程重组回一个对象。如果低层的 API 允许我们指定一个对象以及该对象的名字的话，那么就会比较便于使用。程序 27.7 中展示了这个 API 的可能样子。

程序 27.7　PersistentImp

```
package api;
import java.io.Serializable;
import java.util.AbstractList;

public interface PersistentImp
{
  void store(String name, Serializable obj);
  Object retrieve(String name);
  AbstractList directory(String regExp);
}
```

PersistentImp 接口允许你通过名字来存储（stroe）和取回（retrieve）完整的对象。惟一的限制是这种对象必须要实现 Serializable 接口，这是一个很小的限制。

2．24 小时历史数据

在确定了存储持久化数据的低层机制后，我们来看一下将要持久化的数据的种类。规格说明书中规定我们必须要保存前 24 小时内的最高和最低测量值。图 27.23 中展示了具有这些数据的曲线图。该图看起来似乎没有多大意义。最高和最低的测量值具有很多的冗余，这非常糟糕。更糟糕的是，这些数据是基于时钟的最近 24 小时内的，而不是基于日历的最近 24 小时内的。通常，当我们想要最近 24 小时的最高和最低测量值时，都是指上一个历日（calendar day）的。

这是规格说明中的问题，还是我们理解上的问题呢？如果规格说明中的东西不是客户真正想要的，那么按照它进行实现对我们是没有好处的。

经过涉众的一个快速证实，表明我们的直觉是正确的。我们的确需要一个可滚动的（rolling）最近 24 小时的历史数据。但是，历史的最高和最低测量值是基于历日的。

3．24 小时的最高和最低测量值

每天的最高和最低测量值是基于传感器的实时读数的。例如，每当温度变化时，24 小时的最高和最低温度都会相应地更新。显然，这符合 OBSERVER 模式中的关系。图 27.17 中展示了这个静态结构，图 27.18 中展示了相应的动态情景。

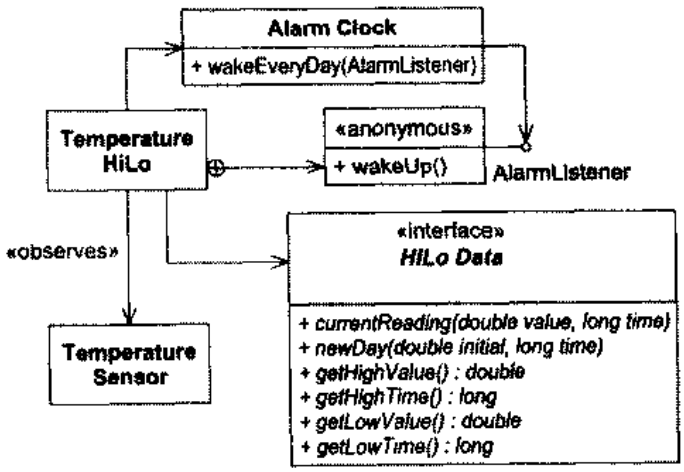


图 27.17　TemperatureHilo 结构

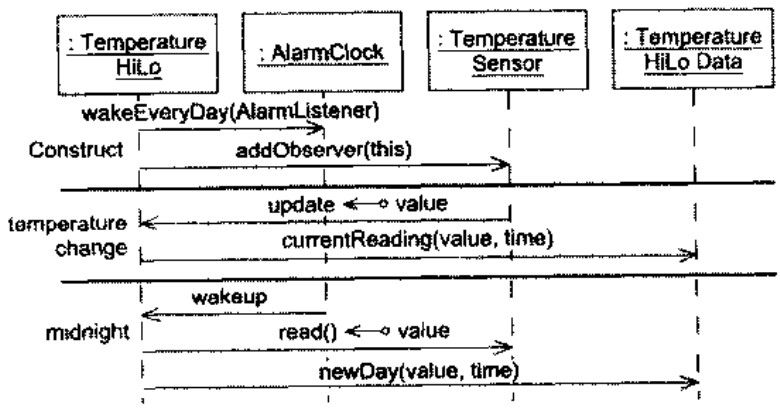


图 27.18　HiLo 的情景图

我们选择了使用一个标记有《observes》构造型（stereotype）的关联来指示 OBSERVER 模式。我们创建了一个称为 TemperatureHiLo 的类，它会在每天晚上午夜时分被 AlarmClock 通知。请注意，WakeEveryDay 方法已经被增加到了 AlarmClock 中。

在 TemperatureHiLo 对象构造时，它会同时向 AlarmClock 和 TemperatureSensor 注册。每当温度变化时，按照 OBSERVER 模式的结构，都会通知 TemperatureHiLo 对象。然后，TemperatureHiLo 使用 currentReading 方法通知 HiLoData 接口。HiLoData 必须要被某些知道如何存储当前 24 小时历日内的最高和最低值的类实现。

因为两个原因，我们分离了 TemperatureHiLo 类和 HiLoData 类。首先，我们希望把 TemperatureSensor 和 AlarmClock 的逻辑 与 确定每天的最高和最低测量值的算法分离开。其次，也是更重要的一个原因，是因为确定每天最高和最低值的算法可以重用于大气压力、风速、露点等上面。因此，虽然我们会需要 BarometricPressureHiLo、DewPointHiLo、WindSpeedHiLo 等去观察它们对应的传感器，但是它们却都可以使用 HiLoData 类来计算和存储数据。

在午夜时分，AlarmClock 向 TemperatureHiLo 对象发送 wakeup 消息。作为响应，TemperatureHiLo 从 TemperatureSensor 中获取当前温度并把它转发给 HiLoData 接口。HiLoData 的实现体必须要使用 PersistenImp 接口去存储前一个历日的温度值，并且还得创建一个具有初始值的新历日。

PersistentImp 根据一个字符串来访问持久化存储设备中的对象。这个字符串是一个访问键值。存储和取回 HiLoData 对象的字符串具有如下的形式：“<type>+HiLo+<MM><dd><yyyy>”，例如，“temperatureHiLo04161998”。

27.2.2 实现 HiLo 算法

如何实现 HiLoData 类呢？似乎很简单。程序 27.8 展示了这个类的 Java 代码。

程序 27.8 HiLoDataImp

```
public class HiLoDataImp implements HiLoData,java.io.Serializable
{
  public HiLoDataImp(StationToolkit st, String type,
                     Date theDate, double init,
                    long initTime)
  {
    itsPI = st.getPersistentImp();
    itsType = type;
    itsStorageKey = calculateStorageKey(theDate);
    try
    {
        HiLoData t = (HiLoData)itsPI.retrieve(itsStorageKey);
        itsHighTime = t.getHighTime();
        itsLowTime = t.getLowTime();
        itsHighValue = t.getHighValue();
        itsLowValue = t.getLowValue();
        currentReading(init, initTime);
    }
    catch(RetrieveException re)
    {
       itsHighValue = itsLowValue = init;
       itsHighTime = itsLowTime = initTime;
    }
```

```
public long getHighTime()     { return itsHighTime; }
public double getHighValue()  { return itsHighValue; }
public long getLowTime()      { return itsLowTime; }
public double getLowValue()   { return itsLowValue; }

// Determine if a new reading changes the
// hi and lo and return true if reading changed.
public void currentReading(double current, long time)
{
  if(current > itsHighValue)
  {
    itsHighValue = current;
    itsHighTime = time;
    store();
  }
  else if(current < itsLowValue)
  {
    itsLowValue = current;
    itsLowTime = time;
    store();
  }
}

public void newDay(double initial, long time)
{
   store();
   // now clear it out and generate a new key.
   itsLowValue = itsHighValue = initial;
   itsHighTime = itsLowTime = time;

   // now calculate a new storage key based on
   // the current date, and store the new record.
   itsStorageKey = calculateStorageKey(new Date());
   store()
}

private store()
{
   try
   {
       itsPI.store(itsStorageKey, this);
   }
   catch(StoreException)
   {
       // log the error somehow.
   }
}

private String calculateStorageKey(Date d)
{
   SimpleDateFormat df = new SimpleDateFormat("Mmddyyyy");
   Return(itsType + "Hello" + df.format(d));
}
private double    itsLowValue;
private long      itsLowTime;
private double    itsHighValue;
private long      itsHighTime;
private String    itsType;
// we don't want to store the following.
```

```
  transient private String itsStorageKey;
  transient private api.PersistentImp itsPI;
}
```

嗯，也许没有那么简单。我们来浏览一下该代码，看看它做了什么。

该类的底部，是一些私有成员变量。头 4 个变量是期望的。它们记录了最高和最低值以及这些值出现的时间。itsType 变量是用来指示该 HiLoData 保存的测量值的类型的。它的值为“Temp”时表示温度，为“BP”时表示大气压力，为“DP”时表示露点，等等。最后两个变量被声明为暂时的（transient）。这意味着它们不会被存储到持久存储器中。它们记录了当前的存储键值以及指向 PersistentImp 的引用。

构造函数有 5 个参数。StationToolkit 是用来获取对 PersistentImp 的访问的。type 和 Date 参数被用来生成存储键值，使用该键值可以存储和取回对象。最后，init 和 initTime 参数是用来在 PersistentImp 无法找到存储键值时初始化对象的。

构造函数试图从 PersistentImp 获取数据。如果得到数据，它就把这些持久的数据拷贝到自己的成员变量中。接着它用初始值和时间作为参数去调用 currentReading 以确保记录了这些测量值。最后，如果 currentReading 发现最高或者最低数据发生了变化，就返回 true，并调用 Store 函数以确保更新了持久存储器。

currentReading 方法是这个类的核心。它把新读入的测量值和原来的最高和最低值进行比较。如果新的测量值比原来的最高值高或者比原来的最低值低，那么它就替换掉相应的值，记录下对应的时间，并把更改记录到持久存储器中。

newDay 方法会在午夜时被调用。它先把当前的 HiLoData 存储到持久存储器中。接着把 HiLoData 的值重新设置为新的一天的开始。它为新日期重新计算存储键值，然后把新的 HiLoData 存储到持久存储器中。

Store 函数只是使用当前的存储键值，通过 PersistantImp 对象把 HiLoData 对象写入持久存储器中。

最后，calculataStorageKey 方法根据 HiLoData 的类型以及日期参数产生一个存储键值。

1．丑陋的实现

显然，程序 27.8 中的代码理解起来并不困难。但是，它因为另外一个原因而变得丑陋。currentReading 函数和 newDay 函数中所表达的策略和管理最高、最低数据值有关，而和持久化无关。另一方面，方法 store、calculateStorageKey、构造函数以及暂时性的变量都是特定于持久化的，而和最高和最低值的管理没有任何关系。这个实现违反了 SRP。

在当前这种职责混合的状态中，这个类是会导致维护噩梦的。如果持久化机制的某些基础部分发生了变化，达到了 calculateStorageKey 以及 store 函数不再适用的程度，那么新的持久化机制就必须要被替换到该类中。为了使用新的持久化功能，像 newDay 和 currentReading 这样的函数就必须要被更改。

2．解除持久化和策略之间的耦合

通过使用 PROXY 模式来解除最高、最低值的管理策略和持久化机制之间的耦合，就可以避免这些潜在的问题。请回顾一下图 26.7（参见 26.1.3）。请注意是如何解除策略层（application）和机制层（API）之间的耦合的。

图 27.19 中使用 PROXY 模式来达到必要的解除耦合的目的。它和 27.2.1 节中的图 27.17 的区别在于增加了一个 HiLoDataProxy 类。TemperatureHilo 对象中实际持有的正是对这个代理类的引用。而代理类又持有一个对 HiLoDataImp 对象的引用，并把调用委托给它。程序 27.9 中展示了 HiLoDataProxy 和 HiloDataImp 中关键函数的实现。

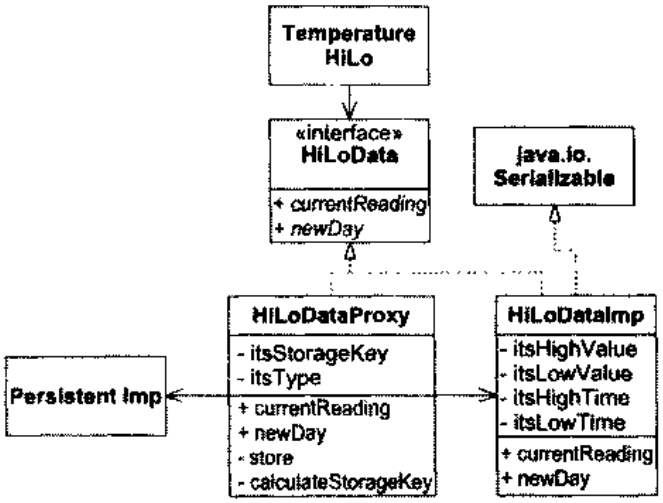


图 27.19　在 HiLo 持久化中应用 PROXY 模式

程序 27.9　PROXY 模式解决方案的代码片段

```
class HiLoDataProxy implements HiLoData
{
  public boolean currentReading(double current, long time)
  {
    boolean change;
    change = itsImp.currentReading(current, time);
    if (change)
        store();
    return change;
  }

  public void newDay(double initial, long time)
  {
    store();
    itsImp.newDay(initial.time);
    calculateStorageKey(new Date(time));
    store();
  }

  private HiLoDataImp itsImp;
}

class HiLoDataImp implements HiLoData,java.io.Serializable
{
  public boolean currentReading(double current, long time)
  {
    boolean changed =  false;
    if(current > itsHighValue)
    {
      itsHighValue = current;
      itsHighTime = time;
      changed =  true;
```

```
    }
    else if(current < itsLowValue)
    {
      itsLowValue = current;
      itsLowTime = time;
      changed = true;
    }
  }

  public void newDay(double initial, long time)
  {
    itsHighTime = itsLowTime = time;
    itsLowValue = itsHighValue = initial;
  }
}
```

现在，HiLoDataImp 对持久化一无所知。而且，HiLoDataProxy 类处理了所有丑陋的持久化逻辑，然后才委托给 HiLoDataImp。这很好。此外，代理类同时依赖于 HiLoDataImp（策略层）以及 PersistentImp（机制层）。这正是我们想要得到的。

但是，事情并非十全十美。敏锐的读者会发现我们对 currentReading 方法所做的改变。我们让它返回一个布尔值。根据这个布尔值，代理类就可以知道何时去调用 store。为什么不在每次调用 currentReading 时都调用 store 呢？是因为 NVRAM 的种类很多。有些 NVRAM 对于写入的次数是有上限的。因此，为了延长 NVRAM 的寿命，我们只在值发生了变化时才存储进 NVRAM 中。现实情况又一次左右了我们。

3. 对象工厂和初始化

显然，我们不想让 TemperatureHiLo 知道代理对象。它只应该知道 HiLoData（参见图 27.19）。但是必须要有某个东西去创建 HiLoDataProxy 让 TemperatureHiLo 对象使用。此外，也必须要有某个东西去创建代理对象要委托的 HiLoDataImp。

我们需要一种不用知道对象的确切类型就可以创建对象的方法。我们需要一种方法来让 TemperatureHiLo 创建 HiLoData 而不知道它实际上创建的是 HiLoDataProxy 和 HiLoDataImp。我们需要再次求助于 FACTORY 模式（参见图 27.20）。

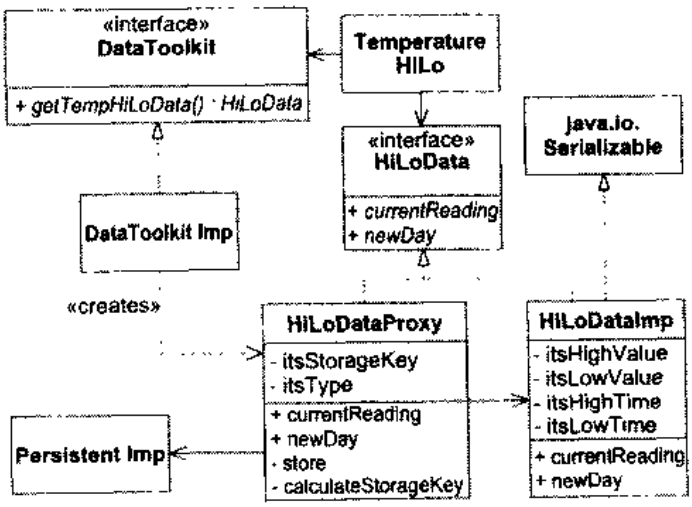


图 27.20　使用抽象工厂（Abstract Factory）来创建代理

TemperatureHiLo 使用 DataToolkit 接口创建了一个符合 HiLoData 接口的对象。getTempHiLoData 方法会被分派到一个 DataToolkitImp 对象上，该对象创建了一个类型码为“Temp”的 HiLoDataProxy 对象，并把它作为 HiLoData 对象返回。

这很好地解决了有关创建的问题。TemperatureHiLo 不必为了创建 HiLoDataProxy 而依赖于它。但是 TemperatureHiLo 怎样才能访问到 DataToolkitImp 对象呢？我们不想让 TemperatureHiLo 和 DataToolkitImp 之间有任何关系，因为这会创建一个从策略层到机制层的依赖关系。

4．包结构

为了回答这个问题，我们先来看一下图 27.21 中的包结构。WMS 是天气监控系统（Weather Monitoring System）包的缩写，在 27.2.1 中的图 27.16 中描述过该包。

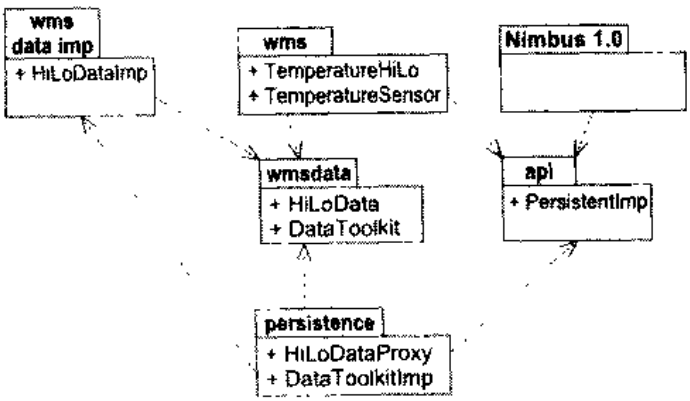


图 27.21　使用 PROXY 模式和 FACTORY 模式的包结构图

图 27.21 进一步强化了我们让持久化层依赖于策略层和机制层的期望。它同时也展示了类和包之间的配置关系。请注意，抽象工厂 DataToolkit 和 HiLoData 一起被定义在 WMSData 包中。HiLoData 的实现是在 WMSDataImp 包中，而 DataToolKit 的实现却是在 persistence 包中。

5．谁来创建工厂

现在，我们再一次问这个问题。为了能够调用 getTempHiLoData 方法并创建 persistence.HiLoDataProxy 的实例，wms.TemperatureHiLo 的实例怎样才能访问到 persistence.DataTollkitImp 的实例呢？

我们所需要的是可以被 wmsdata 中的类访问的某个静态分配的变量，该变量被声明为持有一个 wmsdata.DataToolkit 类型的引用，但却被初始化为一个 persistence.DataToolkitImp 实例。因为 Java 中所有的变量，包括静态变量，都必须声明在某种类中，所以我们可以创建一个命为 Scope 的类，其中含有我们需要的静态变量。我们把这个类放到 wmsdata 包中。

程序 27.10 和程序 27.11 中展示了实现代码。wmsdata 中的 Scope 类声明了一个静态的 DataToolkit 类型的成员变量。persistence 包中的 Scope 类声明了一个 init()函数，该函数创建一个 DataToolkitImp 实例并把它存入 wmsdata.Scope.itsDataToolkit 变量中。

程序 27.10　wmsdata.Scope

```
package wmsdata;

public class Scope
{
```

```
    public static DataToolkit itsDataToolkit;
}
```

程序 27.11　persistence.Scope

```
package persistence;

public class Scope
{
    public static void init()
    {
        wmsdata.Scope.itsDataToolkit =
                new DataToolkit();
    }
}
```

在包和 Scope 类之间有一个有趣的对称。wmsdata 包中除了 Scope 以外的所有类都是只含有抽象方法，不含有变量的接口。但是 wmsdata.Scope 类却含有一个变量，没有函数。另一方面，persistence 包中除了 Scope 以外的所有类都是含有变量的具体类。但是 persistence.Scope 却含有一个函数，没有变量。

图 27.22 中是描述这种情况的一个可能类图。Scope 类是《utility》类。这种类的所有成员，不管是变量还是函数，都是静态的——这是造成对称的决定性因素。看起来，似乎那些包含抽象接口的包中往往会包含有数据而没有函数的工具类，而那些包含具体类的包中往往会包含有函数而没有数据的工具类。

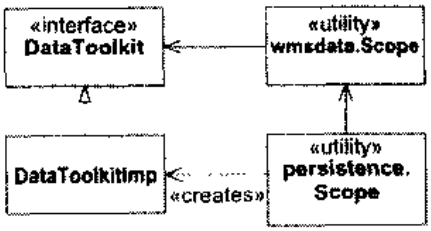


图 27.22　Scope 工具类

6. 那么，谁来调用 persistence.Scope.init()

也许是 main()函数。包含 main 函数的类必须被放在一个不介意依赖于 persistence 的包中。我们通常称包含 main 的包为 root 包。

7. 但是，你说……

持久化实现层不应该依赖于策略层。然而，如果仔细检查图 27.21 的话，就会发现一个从 persistence 到 wmsDataImp 的依赖关系。这个依赖关系可以追溯到图 27.20，其中 HiLoDataProxy 依赖于 HiLoDataImp。存在了这个依赖关系，HiLoDataProxy 就可以创建它所依赖的 HiLoDataImp。

在大多数情况下，代理对象是不必创建实现对象的，因为代理对象可以从持久化存储设备中读取实现对象。也就是说，调用 PersistentImp.retrieve 就会把 HiLoDataImp 对象返回给代理对象。但是，在那些不常发生的情况中，如：retrieve 函数没有在持久化存储设备中找到对象，此时 HiLoDataProxy 就必须创建一个空的 HiLoDataImp。

所以，看起来好像我们还需要另外一个知道如何创建 HiLoDataImp 实例的工厂，并且代理对象可以调用它。这意味着更多的包、更多的类以及其他东西。

8. 这真的是必要的吗

或许在其他的案例中，因为我们希望 TemperatureHiLo 能够使用许多不同的持久化机制，所以我们会创建代理对象的工厂。这样，我们就有充分的理由去证明 DataToolkit 工厂是适当的。但是，在 HiLoDataProxy 和 HiLoDataImp 之间放入一个工厂能带来什么好处呢？如果有许多不同的 HiLoDataImp 实

现，并且如果我们希望它们都能被代理使用，那么这样做也许是适当的。

然而，我们认为需求并不是真的那么易变。包含天气监控策略和业务规则的包已经很长一段时间保持不变了。看起来它们似乎在以后也不太可能会变化。这听起来像是惯用的结束语，但是你必须在某处划定最后界限。在本例中，我们认为从代理类到实现类的依赖关系并不代表着很大的维护风险，我们不需要工厂。

27.3 结　　论

Jim Newkirk 和我在 1998 年初期编写了本章。Jim 完成了大部分的编码，我把代码转换为 UML 图并在周围写上文字。如今，代码早已不在了。但是正是这些代码驱动着你在本章中所看到的设计。大部分的图都是在代码完成后才绘制的。

1998 年时，Jim 和我还都不曾听说过极限编程。所以本章中的设计不是通过结对编程和测试驱动的开发方法产生的。然而，Jim 和我一直都是高度协作的。我们一起审视他编写的代码，在合适的地方去运行它，一起更改设计，然后一起编制了本章中的 UML 图和文字。

所以，虽然本章的设计是在 XP 出现前完成的，但是它仍然是以高度协作、以代码为中心的方式创建的。

参考文献

1. Gamma, et al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
2. Jacobson, Ivar, et al. *Object-Oriented Software Engineering*. Reading, MA: Addison- Wesley, 1992.
3. Arnold, Ken, and James Gosling. *The Java Programming language, 2md ed* . Reading,MA: Addison-Wesley, 1998.

27.4 Nimbus-LC 需求概述

27.4.1 使用需求

该系统应该提供各种天气情况的自动监控功能。特别地，它必须要测量以下变量：

- 风速和风向
- 温度
- 大气压力
- 相对湿度
- 风寒指数
- 露点温度

系统也应该提供当前大气压测量值的趋势。该趋势有 3 个可能的值：稳定、上升和下降。例如，当前大气压力为 29.95 英寸汞柱（IOM）并且呈下降趋势。

系统应该有一个显示器，其上持续地显示所有的测量值以及当前的时间和日期。

27.4.2　24 小时历史数据

通过触摸屏，使用者可以指示系统显示下面任何一个测量值的 24 小时历史数据：

- 温度
- 大气压力
- 相对湿度

历史数据应该以曲线图的形式展示给使用者。

27.4.3　用户设置

在安装期间，系统应该为使用者提供下面的配置功能：

- 设置当前的时间、日期以及时区
- 设置显示单位（英制或者公制）

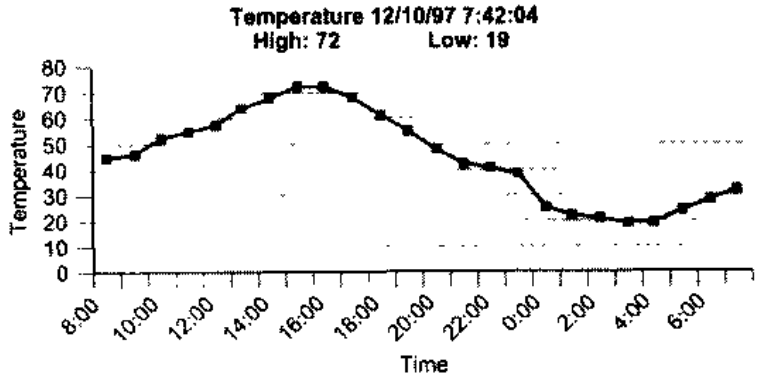


图 27.23　温度历史数据

27.4.4　管理需求

系统应该对气象站应用程序管理功能的使用提供安全机制。这些功能如下：

- 把传感器校对到已知值
- 复位系统

27.5　Nimbus-LC 用例

27.5.1　参与者

在这个系统中，使用者可以具有两种不同的角色。

1. 用户

用户观察系统测量的实时天气信息。他们也和系统交互以显示和某个单独的传感器关联的历史

数据。

2. 管理者

管理者对系统的安全问题进行管理，包括：校正单个传感器、设置时间/日期、设置测量单位以及在需要时复位系统。

27.5.2　用例

用例#1：监控天气数据

系统会显示当前的温度、大气压力、相对湿度、风速、风向、风寒温度、露点以及大气压趋势。

27.5.3　测量的历史数据

系统会显示一个描绘从系统传感器中读出的前 24 小时的测量值的曲线图。除了这个曲线图外，系统还会显示当前的时间和日期以及前 24 小时中的最高和最低测量值。

用例#2：观看温度历史数据

用例#3：观看大气压力历史数据

用例#4：观看相对湿度历史数据

27.5.4　设置

用例#5：设置单位

使用者设置显示使用的单位的类型。可以在英制和公制中做出选择。缺省使用公制。

用例#6：设置日期

用户会设置当前日期。

用例#7：设置时间

用户会为系统设置当前的时间和时区。

27.5.5　管理

用例#8：复位气象站应用程序

管理者能够把气象站系统复位到它出厂时的缺省设置。有很重要的一点要注意，这会清除掉存储在系统中的所有历史数据并且移去任何已经设置的校对值。作为最后的检查，它会告知管理者这样做的后果并询问是否进行系统的复位动作。

用例#9：校对温度传感器

管理者把一个已知正确的温度值输入到系统中。系统应该接受该值并在内部使用它去把传感器

的当前测量值调整到这个实际值上。如果想详细地看一下有关传感器校对的内容，请参见硬件描述文档。

用例#10：校对大气压传感器

用例#11：校对相对湿度传感器

用例#12：校对风速传感器

用例#13：校对风向传感器

用例#14：校对露点传感器

用例#15：校对日志

系统会向管理者展示对设备进行校对的历史记录。该历史记录包括：校对的时间和日期、校对的传感器以及校对传感器所使用的值。

27.6 Nimbus-LC 发布计划

27.6.1 介绍

气象站应用的实现会在一系列的迭代中完成。每个迭代都以上一次迭代完成的工作为基础，直到我们完成了需要发布给客户的功能。这份文档概述了这个项目的 3 次发布

27.6.2 发布 I

本次发布有两个目标。第一个是要创建一个使大部分应用程序独立于 Nimbus 硬件平台的构架。第二个目标是管理两个最大的风险：

（1）使原来的 Nimbus 1.0 API 可以工作于使用新操作系统的处理器板上。这当然是可行的，但是因为我们无法预测所有的不兼容性，所以很难估算出这要花费多长时间。

（2）Java 虚拟机。我们以前从来没有在嵌入式电路板上使用过 JVM。我们不知道它是否能够和我们的操作系统一起工作，或者它是否真的正确实现了所有的 Java 字节码（byte code）。我们的供应商向我们保证一切都没有问题，但是我们仍然感觉这是一个重大的风险。

JVM 与触摸屏以及图形子系统的集成和这次发布同时进行。我们期望在第 2 阶段开始之前这些工作能够完成。

1. 风险

（1）操作系统升级——我们目前使用的是操作系统的一个老版本。为了使用 JVM，我们需要把操作系统升级到一个最新的版本。这也要求我们使用最新版本的开发工具。

（2）操作系统销售商提供了这个版本操作系统上的最新的 JVM 版本。为了跟上形势，我们想使用 JVM 的 1.2 版本。但是，V1.2 目前正在 beta 测试并且会在项目开发期间发生变化。

（3）使用板级“C”API 的 Java 本地接口需要在新构架中进行验证。

2. 要交付的产品

（1）运行着新操作系统和最新版本 JVM 的硬件。

（2）一个流输出，它会显示当前的温度以及大气压力测量值。（这些代码会被丢弃，在最后的发布中不需要它们。）

（3）当大气压力有变化时，系统会通知我们有关压力的上升、下降或者平稳的情况。

（4）每个小时，系统会显示过去 24 小时的温度和大气压力测量值。这些数据会被持久化，这样我们可以关闭并打开设备的电源而数据会被保存下来。

（5）每天上午的 12:00，系统会显示前一天的最高和最低的温度以及大气压力。

（6）所有的测量值都以公制表示。

27.6.3　发布 II

在项目的这个阶段，在第一次发布的基础上增加用户界面的基础部分。不再增加另外的测量种类。对测量本身所做的惟一更改是增加了校对机制。这个阶段主要关注于系统的显示部分。主要的风险是和液晶屏/触摸屏接口的软件。此外，因为这是首次以 UI 的方式展示给用户看的版本，所以可能会造成一些需求的变动。除了软件外，我们会交付一份有关新硬件的规格说明书。这也是在项目的这个阶段才增加校对功能的主要原因。所提供的 API 是用 Java 来详细说明的。

1. 用例实现

- #2——观看温度历史数据
- #3　观看大气压力历史数据
- #5——设置单位
- #6——设置日期
- #7——设置时间/时区
- #9——校准温度传感器
- #10——校准大气压力传感器

2. 风险

（1）液晶屏/触摸屏和 Java 虚拟机的接口需要在实际的硬件上测试。

（2）需求变化

（3）JVM 以及 Java 基础类在从 beata 版本到发布版本时的变化。

3. 要交付的产品

（1）提供并且能够执行上面列出的所有指定功能的系统。

（2）用例#1 中有关温度、大气压力以及时间/日期的部分也要实现。

（3）软件构架中的 GUI 部分要作为这个阶段的一部分完成。

（4）要实现支持对温度和大气压力传感器校准的管理部分软件。

（5）新硬件 API 的规格说明要用 Java 语言而不是用“C”语言描述。

27.6.4　发布III

这是客户部署产品前的一次发布。

1. 用例实现

- #1——监控天气数据
- #4——观看相对湿度历史数据
- #8——复位气象站应用系统
- #11——校准相对湿度传感器
- #12——校准风速传感器
- #13——校准风向传感器
- #14——校准露点传感器
- #15——记录校准日志

2. 风险

（1）需求变化——一般认为，在产品的不断完善中，总会出现一些需求变化。
（2）完成整个产品意味着要对发布 II 结束时规定的硬件 API 进行改动。
（3）硬件限制——当产品完成时，可能会碰到一些硬件限制（如：内存、CPU，等等）。

3. 要交付的产品

（1）运行在原来硬件平台上的新软件。
（2）在这次实现中被验证过的新硬件的规格说明。

第VI部分　ETS案例研究

在美国或者加拿大，要想成为一个有资质的建筑师，必须要通过一个考试。如果通过考试，就可以得到国家资质委员会颁发的资质证书，这是进行实际的建筑设计所必需的。考试是由国家注册建筑师委员会（NCARB）授权的教育考试中心（ETS）来承办的。目前，诚希国际集团（Chauncey Group International）在管理着这项考试。

过去，应试者都是用铅笔和纸来完成考试的。然后，由一个评审中心来对这些完成的试卷进行评分。评审中心由经验非常丰富的设计师组成，他们会仔细地评阅试卷并决定是否能够通过考试。

1989年，NCARB授权ETS去研究一下是否可以在一个自动的系统中解答考试的部分内容并对其评分。本部分中的章节描述了由此而来的项目的部分内容。像以前一样，在设计这个软件的过程中我们会碰到许多有用的设计模式，所以我们在案例学习前，先用一些章节来描述这些模式。

第 28 章　VISITOR 模式

“T 是某个来访者，”我低声说着，“轻敲着我的房门；仅此而已”

——埃德加·爱伦·坡（美国小说家），《乌鸦》

问题：需要向类层次结构中增加新的方法，但是增加起来会很费劲或者会破坏设计。

这是一个很常见的问题。例如，假设你有一个 Modem 对象的层次结构。基类中具有对于所有调制解调器来说公共的通用方法。派生类代表着针对许多不同调制解调器厂商和类型的驱动程序。同样假设你有一个需求，要向该层次结构中增加一个新方法，名为 configureForUnix。这个方法会对调制解调器进行配置，使之可以工作于 UNIX 操作系统中。在每个调制解调器派生类中，该函数的实现都不相同，因为每个不同的调制解调器在 UNIX 中都有自己独特的配置方法和行为特征。

糟糕的是，增加 configureForUnix 方法其实回避了非常讨厌的一组问题。对于 Windows 该怎么办？对于 MacOS 该怎么办呢？对于 Linux 又该怎么办呢？我们真的必须针对所使用的每一种新操作系统都要向 Modem 层次结构中增加一个新方法吗？这种做法显然是丑陋的。我们将永远无法封闭 Modem 接口。每当出现一种新操作系统时，我们就必须更改该接口并重新部署所有的调制解调器软件。

28.1　VISITOR 设计模式系列

VISITOR 模式系列允许在不更改现有类层次结构的情况下向其中增加新方法。

该系列中的模式如下：

- VISITOR 模式
- ACYCLIC VISITOR 模式
- DECORATOR 模式
- EXTENSION OBJECT 模式

28.2　VISITOR 模式[①]

请考虑一下图 28.1 中的 Modem 层次结构。Modem 接口包含了所有调制解调器都能实现的通用方法。图中展示了 3 个派生类——一个驱动着 Hayes

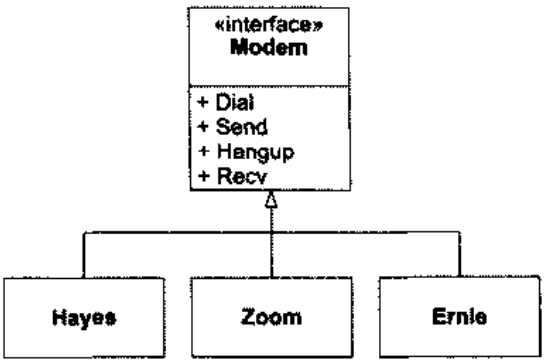


图 28.1　Modem 层次结构

① [GOF95]，第 331 页。

调制解调器，另一个驱动着 Zoom 调制解调器，第 3 个驱动着我们的一个硬件工程师 Ernie 制作的调制解调器卡。

如果不在 Modem 接口中增加 ConfigureForUnix 方法，那么我们怎么才能把这些调制解调器配置为可以在 UNIX 中使用呢？我们可以使用一项名为双重分发（dual dispatch）的技术，这项技术是 VISITOR 模式的核心机制。

图 28.2 展示了 VISITOR 模式的结构，程序 28.1 至 28.6 展示了相应的 Java 代码。程序 28.7 展示了测试代码，该测试代码既验证了 VISITOR 模式可以工作又演示了其他的程序员该如何使用它。

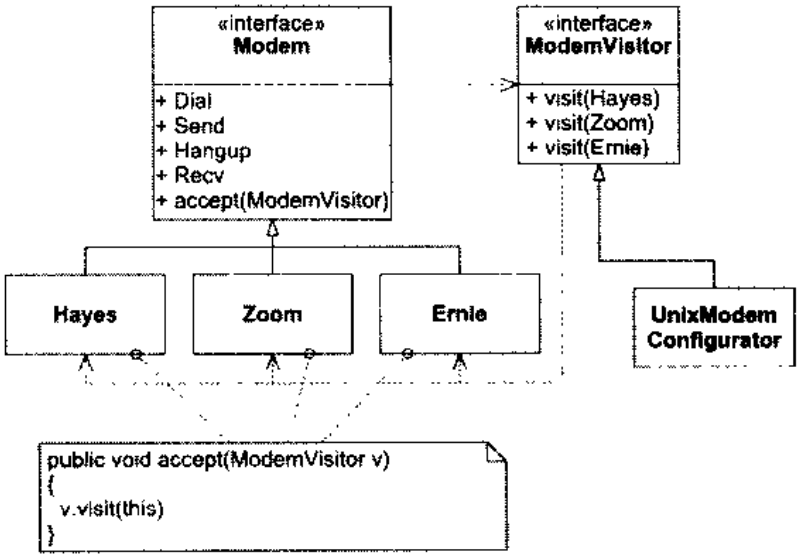


图 28.2　VISITOR 模式

程序 28.1　Modem.java

```
public interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
    public void accept(ModemVisitor v);
}
```

程序 28.2　ModemVisitor.java

```
public interface ModemVisitor
{
    public void visit(HayesModem modem);
    public void visit(ZoomModem modem);
    public void visit(ErnieModem modem);
}
```

程序 28.3　HayesModem.java

```
public class HayesModem implements Modem
{
    public void dial(String pno) {}
    public void hangup() {}
    public void send(char c) {}
    public char recv() {return 0;}
    public void accept(ModemVisitor v) {v.visit(this);}
```

```
    String configurationString = null;
}
```

程序 28.4 ZoomModem.java

```
public class ZoomModem implements Modem
{
    public void dial(String pno) {}
    public void hangup() {}
    public void send(char c) {}
    public char recv() {return 0;}
    public void accept(ModemVisitor v) {v.visit(this);}

    int configurationValue = 0;
}
```

程序 28.5 ErineModem.java

```
public class ErnieModem implements Modem
{
    public void dial(String pno) {}
    public void hangup() {}
    public void send(char c) {}
    public char recv() {return 0;}
    public void accept(ModemVisitor v) {v.visit(this);}

    String configurationPattern = null;
}
```

程序 28.6 UnixModemConfigurator.java

```
public class UnixModemConfigurator implements ModemVisitor
{
    public void visit(HayesModem m)
    {
        m.configurationString = "&s1=4&D=3";
    }

    public void visit(ZoomModem m)
    {
        m.configurationValue = 42;
    }

    public void visit(ErnieModem m)
    {
        m.configurationPattern = "C is too slow";
    }
}
```

程序 28.7 TestModemVisitor.java

```
import junit.framework.*;

public class TestModemVisitor extends TestCase
{
    public TestModemVisitor(String name)
    {
        super(name);
    }

    private UnixModemConfigurator v;
    private HayesModem h;
```

```
    private ZoomModem z;
    private ErnieModem e;

    public void setUp()
    {
        v = new UnixModemConfigurator();
        h = new HayesModem();
        z = new ZoomModem();
        e = new ErnieModem();
    }

    public void testHayesForUnix()
    {
        h.accept(v);
        assertEquals("&s1=4&D=3", h.configurationString);
    }

    public void testZoomForUnix()
    {
        z.accept(v);
        assertEquals(42, z.configurationValue);
    }

    public void testErnieForUnix()
    {
        e.accept(v);
        assertEquals("C is too slow", e.configurationPattern);
    }
}
```

请注意，对于被访问（Modem）层次结构中的每一个派生类，访问者（visitor）层次结构中都有一个对应的方法。这是一种从派生类到方法的 90 度旋转。

测试代码显示出，为了把调制解调器配置为可以在 UNIX 中使用，程序员创建了 UnixModemConfigurator 的一个实例，并把它传给 Modem 的 accept 函数。接着，相应的 Modem 派生对象会调用 UnixModemConfigurator 的基类 ModemVisitor 的 visit(this)。如果这个派生对象是一个 Hayes，那么 visit(this)就会调用 public void visit(Hayes)。这个调用会被分发到 UnixModemConfigurator 中的 public void visit(Hayes)函数，接着该函数把 Hayes 调制解调器配置为可以在 UNIX 中使用。

构建了这个结构后，就可以通过增加新的 ModemVisitor 派生类来增加新的操作系统配置函数，而完全不用对 Modem 层次结构进行更改。所以，VISITOR 模式使用 ModemVisitor 的派生类替代了 Modem 层次结构中的方法。

这之所以被称为双重分发是因为它涉及了两个多态分发。第一个分发是 accept 函数。该分发辨别出所调用的 accept 方法所属对象的类型。第二个分发是 visit 方法，它辨别出要执行的特定函数。这两个分发赋予了 VISITOR 模式非常快的执行速度。

28.2.1 VISITOR 模式如同一个矩阵

VISITOR 模式中的两次分发形成了一个功能矩阵。在调制解调器的例子中，矩阵的一条轴是不同类型的调制解调器。另一条轴是不同类型的操作系统。该矩阵的每个单元都被一项功能填充，该功能描绘了如何把特定的调制解调器初始化为可以在特定的操作系统中使用。

28.3　ACYCLIC VISITOR 模式

请注意，被访问（Modem）层次结构的基类依赖于访问者层次结构（ModemVisitor）的基类。同样请注意，访问者层次结构的基类中对于被访问层次结构中的每个派生类都有一个对应函数。因此，就有一个依赖环把所有被访问的派生类（所有的调制解调器）绑定在一起。这样，就很难实现对访问者结构的增量编译，并且也很难向被访问层次结构中增加新的派生类。

如果程序中要更改的层次结构不需要经常地增加新的派生类，那么 VISITOR 模式工作的很好。如果我们很可能只需要 Hayes、Zoom 以及 Ernie，或者很少会去增加新的 Modem 派生类，那么 VISITOR 模式将会非常合适。

另一方面，如果被访问层次结构非常不稳定，经常需要创建许多新的派生类，那么每当向被访问层次结构中增加一个新的派生类时，就必须要更改并且重新编译 Visitor 基类（也就是，ModemVisitor）以及它的所有派生类。在 C++中，情况甚至更糟。每当增加任何一个新的派生类时，整个被访问层次结构就必须要被重新编译、重新部署。

可以使用一个称为 ACYCLIC VISITOR 模式的变体来解决这个问题[①]（参见图 28.3）。该变体把 Visitor 基类（ModemVisitor）变成退化[②]的，从而解除了依赖环。这个类中没有任何方法意味着它没有依赖于被访问层次结构的派生类。

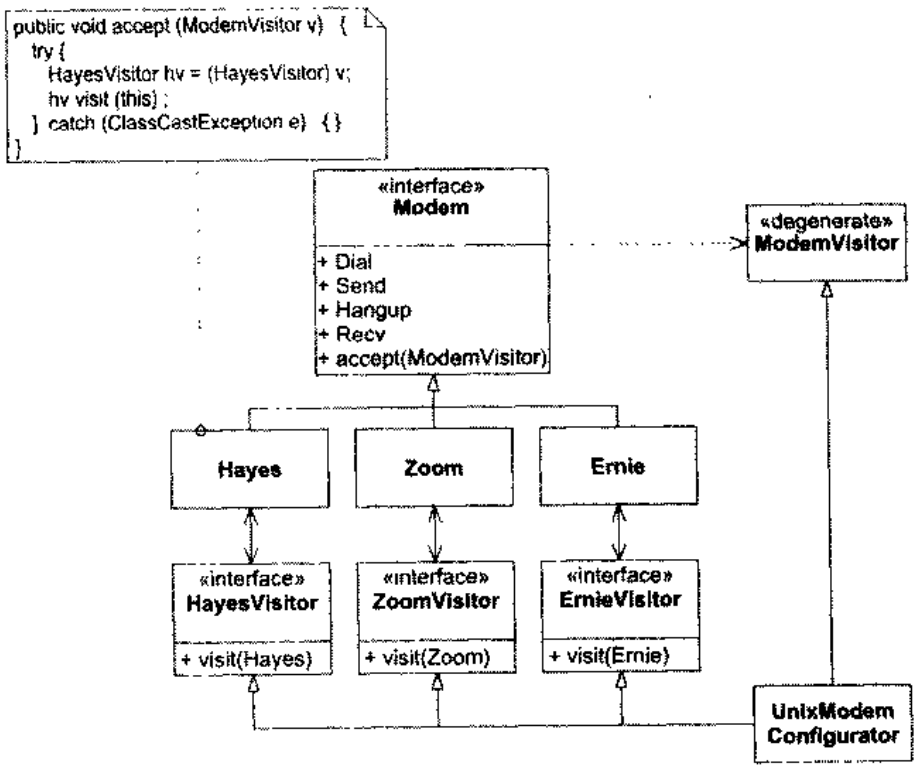


图 28.3　ACYCLIC VISITOR 模式

① [PLOPD3]，第 93 页

② 退化类就是没有任何方法的类。在 C++中，它会具有一个纯虚析构函数。在 Java 中，这种类被称为“标记接口（Marker Interface）”。

访问者派生类同样派生自访问者（visitor）接口。对于被访问层次结构的每个派生类，都有一个对应的访问者接口。这是一个从派生类到接口的 180 度旋转。被访问派生类中的 accept 函数把 Visitor 基类转型（cast）[1]为适当的访问者接口。如果转型成功，该方法就调用相应的 visit 函数。程序 28.8 至 28.16 中为对应的代码。

这种做法解除了依赖环，并且更易于增加被访问的派生类以及进行增量编译。糟糕的是，它同样也使得解决方案更加复杂了。更糟糕的是，转型花费的时间依赖于被访问层次结构的宽度和深度，所以很难进行测定。

由于转型需要花费大量的执行时间，并且这些时间是不可预测的，所以 ACYCLLIC VISITOR 模式不适用于严格的实时系统。该模式的复杂性可能同样会使它不适用于其他的系统。但是，对于那些被访问的层次结构不稳定，并且增量编译比较重要的系统来说，该模式是一个不错的选择。

程序 28.8　Modem.java

```
public interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
    public void accept(ModemVisitor v);
}
```

程序 28.9　ModemVisitor.java

```
public interface ModemVisitor
{
}
```

程序 28.10　ErnieModemVisitor.java

```
public class ErnieModemVisitor
{
    public void visit(ErnieModem m);
}
```

程序 28.11　HayesModemVisitor.java

```
public class HayesModemVisitor
{
    public void visit(HayesModem m);
}
```

程序 28.12　ZoomModemVisitor.java

```
public class ZoomModemVisitor
{
    public void visit(ZoomModem m);
}
```

程序 28.13　ErnieModem.java

```
public class ErnieModem implements Modem
{
    public void dial(String pno) {}
    public void hangup() {}
    public void send(char c) {}
    public char recv() {return 0;}
```

(1) 在 C++中，使用 dynamic_cast。

```
    public void accept(ModemVisitor v)
    {
        try
        {
            ErnieModemVisitor ev = (ErnieModemVisitor)v;
            ev.visit(this);
        }
        catch (ClassCastException e)
        {
        }
    }

    String configurationString = null;
}
```

程序 28.14 HayesModem.java

```
public class HayesModem implements Modem
{
    public void dial(String pno) {}
    public void hangup() {}
    public void send(char c) {}
    public char recv() {return 0;}
    public void accept(ModemVisitor v)
    {
        try
        {
            HayesModemVisitor hv = (HayesModemVisitor)v;
            hv.visit(this);
        }
        catch (ClassCastException e)
        {
        }
    }

    String configurationString = null;
}
```

程序 28.15 ZoomModem.java

```
public class ZoomModem implements Modem
{
    public void dial(String pno) {}
    public void hangup() {}
    public void send(char c) {}
    public char recv() {return 0;}
    public void accept(ModemVisitor v)
    {
        try
        {
            ZoomModemVisitor zv = (ZoomModemVisitor)v;
            zv.visit(this);
        }
        catch (ClassCastException e)
        {
        }
    }

    int configurationValue = 0;
}
```

程序 28.16 TestModemVisitor.java

```
import junit.framework.*;

public class TestModemVisitor extends TestCase
{
    public TestModemVisitor(String name)
    {
        super(name);
    }
            v
    private UnixModemConfigurator v;
    private HayesModem h;
    private ZoomModem z;
    private ErnieModem e;

    public void setUp()
    {
        v = new UnixModemConfigurator();
        h = new HayesModem();
        z = new ZoomModem();
        e = new ErnieModem();
    }

    public void testHayesForUnix()
    {
        h.accept(v);
        assertEquals("&s1=4&D=3", h.configurationString);
    }

    public void testZoomForUnix()
    {
        z.accept(v);
        assertEquals(42, z.configurationValue);
    }

    public void testErnieForUnix()
    {
        e.accept(v);
        assertEquals("C is too slow", e.configurationPattern);
    }
}
```

28.3.1 ACYCLIC VISITOR 模式如同一个稀疏矩阵

正像 VISITOR 模式创建了一个功能矩阵（一个轴是被访问的类型，另一个轴是要执行的功能）一样，ACYCLIC VISITOR 模式创建了一个稀疏矩阵。访问者类不需要针对每一个被访问的派生类都实现 visit 函数。例如，如果 Erniet 调制解调器不可以配置在 UNIX 中，那么 UnixModemConfigurator 就不会实现 ErineVisitor 接口。因此，ACYCLIC VISITOR 模式允许我们忽略某些派生类和功能的组合。有时，这可能是一个有用的优点。

28.3.2 在报表生成器中使用 VISITOR 模式

VISITOR 模式的一个非常常见的应用是，遍历大量的数据结构并产生报表。这使得数据结构对象中不含有任何产生报表的代码。如果想增加新报表，只需增加新的访问者，而不需要更改数据结构中的代码。这意味着报表可以被放置在不同的组件中，并且仅被那些需要它们的客户单独使用。

请考虑一个表示材料单的简单数据结构（见图 28.4）。从该数据结构可以生成无数的报表。例如，我们可以生成一张一个组合件总成本的报表，或者生成一张列出了一个组合件中所有零件的报表。

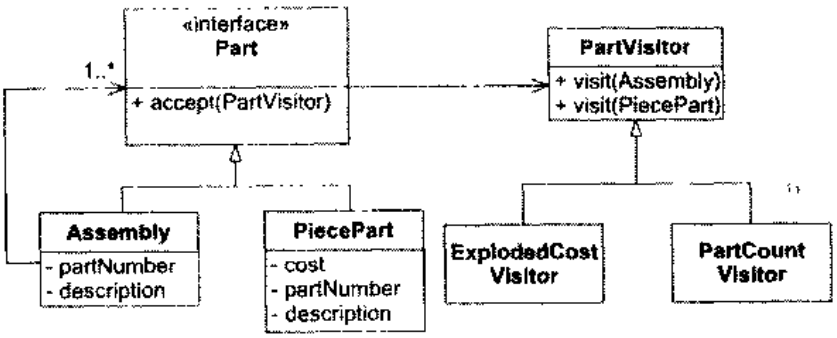


图 28.4　材料单报表生成器结构

每个报表都可以通过 Part 类中的方法生成。例如，可以把 getExplodedCost 和 getPieceCount 增加到 Part 类中。这两个方法会在 Part 的每个派生类中实现，这样就能生成相应的报表。糟糕的是，这意味着每当客户想要一种新报表时，我们都必须要更改 Part 层次结构。

单一职责原则（SRP）告诉我们要分离那些因为不同原因而改变的代码。Part 层次结构可能会因为需要新类型的零件而改变。但是，它不应该因为需要新类型的报表而改变。因此，我们想把报表和 Part 层次结构分离。我们在图 28.4 中看到的 VISITOR 模式结构展示了解决的方法。

每种新报表都可以作为一个新的访问者编写。在 Assembly 的 accept 函数的实现中，会调用访问者的 visit 方法以及它所包含的所有 Part 实例的 accept 方法。这样，就遍历了整个层次结构树。对于树中的每个结点，都会调用报表对象的相应 visit 函数。报表对象收集了必要的统计数据，然后就可以向报表对象询问感兴趣的数据并把它们呈现给使用者。

该结构允许我们在完全不影响 Part 层次结构的情况下创建任意数目的报表。此外，每个报表类都可以独立于所有其他报表类编译和分发。这很好。程序 28.17 至 28.23 展示了该方案的 Java 代码实现。

程序 28.17　Part.java

```
public interface Part
{
    public String getPartNumber();
    public String getDescription();
    public void accept(PartVisitor v);
}
```

程序 28.18　Assembly.java

```
import java.util.*;

public class Assembly implements Part
{

    public Assembly(String partNumber, String description)
    {
        itsPartNumber = partNumber;
        itsDescription = description;
    }

    public void accept(PartVisitor v)
```

```
    {
        v.visit(this);
        Iterator i = getParts();
        while (i.hasNext())
        {
            Part p = (Part)i.next();
            p.accept(v);
        }
    }

    public void add(Part part)
    {
        itsParts.add(part);
    }

    public Iterator getParts()
    {
        return itsParts.iterator();
    }

    public String getPartNumber()
    {
        return itsPartNumber;
    }

    public String getDescription()
    {
        return itsDescrition;
    }

    private List itsParts = new LinkedList();
    private String itsPartNumber;
    private String itsDescription;
}
```

程序 28.19 PiecePart.java

```
public class PiecePart implements Part
{

    public PiecePart(String partNumber, String description, double cost)
    {
        itsPartNumber = partNumber;
        itsDescription = description;
        itsCost = cost;
    }

    public void accept(PartVisitor v)
    {
        v.visit(this);
    }

    public String getPartNumber()
    {
        return itsPartNumber;
    }

    public String getDescription()
    {
        return itsDescrition;
    }
```

```
    public double getCost()
    {
        return itsCost;
    }

    private String itsPartNumber;
    private String itsDescription;
    private double itsCost;
}
```

程序 28.20 PartVisitor.java

```
public interface PartVisitor
{
    public void visit(PiecePart pp);
    public void visit(Assembly a);
}
```

程序 28.21 ExplodedCostVisitor.java

```
public class ExplodedCostVisitor implements PartVisitor
{
    private double cost = 0;
    public double cost() {return cost;}

    public void visit(PiecePart p)
    { cost += p.getCost();}

    public void visit(Assembly a) {}
}
```

程序 28.22 PartCountVisitor.java

```
import java.util.*;

public class PartCountVisitor implements PartVisitor
{

    public void visit(PiecePart p)
    {
        itsPieceCount++;
        String PartNumber = p.getPartNumber();
        int partNumberCount = 0;
        if (itsPieceMap.containsKey(partNumber))
        {
            Integer carrier = (Integer)itsPieceMap.get(partNumber);
            partNumberCount = carrier.intValue();
        }
        partNumberCount++;
        itsPieceMap.put(partNumber, new Integer(partNumberCount));
    }

    public void visit(Assembly a)
    {
    }

    public int getPiectCount() { return itsPieceCount; }
    public int getPartNumberCount() { return itsPieceMap.size(); }
    public int getCountForPart(String partNumber)
    {
        int partNumberCount = 0;
```

```
        if (itsPieceMap.containsKey(partNumber))
        {
            Integer carrier = (Integer)itsPieceMap.get(partNumber);
            partNumberCount = carrier.intValue();
        }
        return partNumberCount;
    }

    private int itsPieceCount = 0;
    private HashMap itsPieceMap = new HashMap();

}
```

程序 28.23 TestBOMReport.java

```
import junit.framework.*;
import java.util.*;

public class TestBOMReport extends TestCase
{

    public TestBOMReport(String name)
    {
        super(name);
    }

    private PiecePart p1;
    private PiecePart p2;
    private Assembly a;

    public void setUp()
    {
        p1 = new PiecePart("997624", "MyPart", 3.20);
        p2 = new PiecePart("7734", "Hell", 666);
        a = new Assembly("5879", "MyAssembly");
    }

    public void testCreatePart()
    {
        assertEquals("997624", p1.getPartNumber());
        assertEquals("MyPart", p1.getDescription());
        assertEquals(3.20, p1.getCost(), .01);
    }

    public void testCreateAssembly()
    {
        assertEquals("5879", a.getPartNumber());
        assertEquals("MyAssembly", a.getDescription());
    }

    public void testAssembly()
    {
        a.add(p1);
        a.add(p2);
        Iterator i = a.getParts();
        PiecePart p = (PiecePart)i.next();
        assertEquals(p, p1);
        p = (PiecePart)i.next();
        assertEquals(p, p2);
        assertEquals(i.hasNext() == false);
    }
```

```
public void testAssemblyOfAssemblies()
{
    Assembly subAssembly = new Assembly("1324", "SubAssembly");
    subAssembly.add(p1);
    a.add(subAssembly);

    Iterator i = a.getParts();
    assertEquals(subAssembly, i.next());
}

private boolean p1Found = false;
private boolean p2Found = false;
private boolean aFound = false;

public void testVisitorCoverage()
{
    a.add(p1);
    a.add(p2);
    a.accept(new PartVisitor() {
        public void visit(PiecePart p)
        {
            if (p==p1)
                p1Found = true;
            else if (p==p2)
                p2Found = true;
        }

        public void visit(Assembly assy)
        {
            if (assy==a)
                aFound = true;
        }
    });
    assert(p1Found);
    assert(p2Found);
    assert(aFound);
}

private Assembly cellphone;

void setUpReportDatabase()
{
    cellphone = new Assembly("CP-7734", "Cell Phone");
    PiecePart display = new PiecePart("DS-1428", "LCD Display", 14.37);
    PiecePart speaker = new PiecePart("SP-92", "Speaker", 3.50);
    PiecePart microphone = new PiecePart("MC-28", "Microphone", 5.30);
    PiecePart cellRadio = new PiecePart("CR-56", "Cell Radio", 30);
    PiecePart frontCover = new PiecePart("FC-77", "Front Cover", 1.4);
    PiecePart backCover = new PiecePart("RC-77", "RearCover", 1.2);
    Assembly keypad = new Assembly("KP-62", "Keypad");
    Assembly button = new Assembly("B52", "Button");
    PiecePart buttonCover = new PiecePart("CV-15", "Cover", .5);
    PiecePart buttonContact = new PiecePart("CN-2", "Contact", 1.2);
    button.add(buttonCover);
    button.add(buttonContact);
    for (int i=0; i<15; i++)
        keypad.add(button);
    cellphone.add(display);
    cellphone.add(speaker);
```

```
        cellphone.add(microphone);
        cellphone.add(cellRadio);
        cellphone.add(frontCover);
        cellphone.add(backCover);
        cellphone.add(keypad);
    }

    public void testExplodedCost()
    {
        setUpReportDatabase();
        ExplodedCostVisitor v = new ExplodedCostVisitor();
        cellphone.accept(v);
        assertEquals(81.27, v.cost(), .001);
    }

    public void testPartCount()
    {
        setUpReportDatabase()
        PartCountVisitor v = new PartCountVisitor();
        cellphone.accept(v);
        assertEquals(36, v.getPieceCount());
        assertEquals(8, v.getPartNumberCount());
        assertEquals("DS-1428", 1, v.getCountForPart("DS-1428"));
        assertEquals("SP-92", 1, v.getCountForPart("SP-92"));
        assertEquals("MC-28", 1, v.getCountForPart("MC-28"));
        assertEquals("CR-56", 1, v.getCountForPart("CR-56"));
        assertEquals("RC-77", 1, v.getCountForPart("RC-77"));
        assertEquals("CV-15", 1, v.getCountForPart("CV-15"));
        assertEquals("CN-2", 1, v.getCountForPart("CN-2"));
        assertEquals("Bob", 0, v.getCountForPart("Bob"));
    }
}
```

28.3.3 VISITOR 模式的其他用途

一般来说，如果一个应用程序中存在有需要以多种不同方式进行解释的数据结构，就可以使用 Visitor 模式。编译器通常创建一些中间数据结构来表示那些语法上正确的源代码。然后，这些数据结构被用来生成经过编译的代码。有人会设想出针对每种不同的处理器或者优化方案的访问者。同样也有人会设想出把中间数据转换成交叉引用列表，甚至 UML 图的访问者。

很多应用程序都使用配置数据结构。有人会设想让不同的应用程序子系统通过使用它们自己特定的访问者遍历配置数据来对自己进行初始化。

在每个使用访问者的情况中，所使用的数据结构都独立于它的用途。可以创建新的访问者，可以更改现有的访问者，并且可以把所有的访问者重新部署到安装地点而不会引起现有数据结构的重新编译和重新部署。这就是 VISITOR 模式的威力。

28.4 DECORATOR 模式[①]

VISITOR 模式给我们提供一种方法，使用这种方法可以在不改变现有类层次结构的情况下向其中增加新方法。另外一个可以达到这个目标的模式是 DECORATOR 模式。

① [GOF95]。

请再考虑一下图 28.1 中的 Modem 层次结构。假设我们有一个具有很多使用者的应用程序。每个使用者都可以坐在他的计算机前，要求系统使用该计算机的调制解调器呼叫另一台计算机。有些用户希望听到拨号声，有些用户则希望他们的调制解调器保持安静。

我们可以通过在代码中每一处对调制解调器拨号的地方询问使用者的优先选择来实现这一点。如果使用者希望听到拨号声，我们就将扬声器的音量设高。否则，我们就把它关掉。

```
...
Modem m = user.getModem();
if(user.wantsLoudDial())
    m.setVolume(11);     // its more than 10, isn't it?
m.dial(...);
...
```

看到这段代码幽灵般成百上千次地遍布于应用程序中，就会在我们的心中浮现出每周工作 80 个小时以及正在进行着可恨的调试的景象。这种做法是要避免的。

另一种方法是在调制解调器对象内部设置一个标志，让 dial 方法检测这个标志并相应地设置音量。

```
...
public class HayesModem implements Modem
{
    private boolean wantsLoudDial = false;
    public void dial(...)
    {
        if(wantsLoudDial)
        {
            setVolume(11);
        }
        ...
    }
    ...
}
```

这样做虽然好了一些，但是仍然必须在 Modem 每个的派生类中重复这一段代码。Modem 新派生类的编写者必须要记着复制这段代码。依赖于程序员的记忆力是相当冒险的事。

我们可以使用 TEMPLATE METHOD 模式[①]来解决这个问题，方法如下：把 Modem 从接口变成一个类，让它持有 wantsLoudDial 变量，并且在它的 dial 函数中先检测完该变量后再去调用 dialForReal 函数。

```
...
public abstract class Modem
{
    private boolean wantsLoudDial = false;
    public void dial(...)
    {
        if(wantsLoudDial)
        {
            setVolume(11);
        }
        dialForReal(...);
```

① 参见第 14 章的“TEMPLATE METHOD 模式”

```
    }
    public abstract void dialForReal(...);
}
```

这虽然更好了一些，但是为什么使用者突然的想法就应该以这种方式影响到 Modem 呢？Modem 为什么应该知道大声拨号呢？每当使用者提出一些其他的古怪要求时（比如：在挂断前先注销），就必须对它进行更改吗？

我们要再次使用共同封闭原则（CCP）。我们想分离那些由于不同的原因而改变的东西。我们同样也可以使用单一职责原则（SRP），因为大声拨号的需要和调制解调器的内在功能没有任何关系，所以也不应该成为调制解调器的一部分。

DECORATOR 模式通过创建一个名为 LoudDialModem 的全新类来解决这个问题。LoudDialModem 派生自 Modem ，并且委托给一个它包含的 Modem 实例。它捕获对 dial 函数的调用并在委托前把音量设高。图 28.5 展示了这个结构。

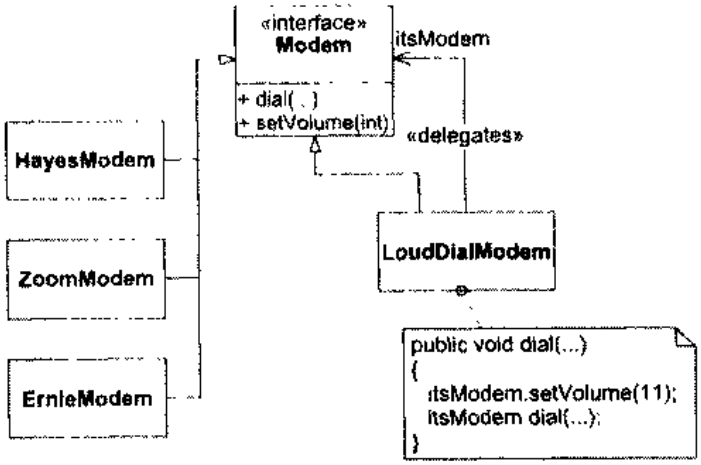


图 28.5 DECORATOR: LoudDialModem

现在对大声拨号的决定是在一个地方进行的。如果使用者要求大声拨号，那么在代码中设置使用者优先选择的地方可以创建一个 LoudDialModem 对象，并把使用者的调制解调器对象传给它。LoudDialModem 会把对它的所有调用都委托给使用者的调制解调器，所以使用者不会察觉到任何不同。不过，dial 方法会在委托给使用者的调制解调器前，先把音量设高。于是，LoudDialModem 就在不影响系统中任何其他东西的情况下，变成了使用者的调制解调器。程序 28.24 至 28.27 展示了实现代码。

程序 28.24 Modem.java

```
public interface Modem
{
    public void dial(String pno);
    public void setSpeakerVolume(int volume);
    public String getPhoneNumber();
    public int getSpeakerVolume();
}
```

程序 28.25 HayesModem.java

```
public class HayesModem implements Modem
{
    public void dial(String pno)
```

```
    {
        itsPhoneNumber = pno;
    }

    public void setSpeakerVolume(int volume)
    {
        itsSpeakerVolume = volume;
    }

    public String getPhoneNumber()
    {
        return itsPhoneNumber;
    }

    public int getSpeakerVolume()
    {
        return itsSpeakerVolume;
    }

    private String itsPhoneNumber;
    private int itsSpeakerVolume;
}
```

程序 28.26 LoudDialModem.java

```
public class LoudDialModem implements Modem
{
    public LoudDialModem(Modem m)
    {
        itsModem = m;
    }

    public void  dial(String pno)
    {
        itsModem.setSpeakerVolume(10);
        itsModem.dial(pno);
    }

    public void setSpeakerVolume(int volume)
    {
        itsModem.setSpeakerVolume(volume);
    }

    public String getPhoneNumber()
    {
        return itsModem.getPhoneNumber();
    }

    public int getSpeakerVolume()
    {
        return itsModem.getSpeakerVolume()
    }

    private Modem itsModem;
}
```

程序 28.27 ModemDecoratorTest.java

```
import junit.framework.*;

public class ModemDecoratorTest extends TestCase
```

```
{
    public ModemDecoratorTest(String name)
    {
        super(name);
    }

    public void testCreateHayes()
    {
        Modem m = new HayesModem();
        assertEquals(null, m.getPhoneNumber());
        m.dial("5551212");
        assertEquals("5551212", m.getPhoneNumber());
        assertEquals(0, m.getSpeakerVolume());
        m.setSpeakerVolume(10);
        assertEquals(10, m.getSpeakerVolume());
    }

    public void testLoudDialModem()
    {
      Modem m = new HayesModem();
      Modem d = new LoudDialModem(m);
      assertEquals(null, d.getPhoneNumber());
      assertEquals(0, d.getSpeakerVolume());
      d.dial("5551212");
      assertEquals("5551212", d.getPhoneNumber());
      assertEquals(10,
        d.getSpeakerVolume());
    }
}
```

28.4.1 多个 Decorator

有时，在同一个类层次结构中可能存在两个或者更多的装饰器（decorator）。例如，我们可能希望用 LogoutExitModem 来装饰 Modem 层次结构，每当 Hangup 方法被调用时，它就会发送字符串 'exit'。这个（第 2 个）装饰器必须要重复我们已经在 LoudDialModem 中编写过的所有委托代码。要消除该重复代码，我们可以创建一个名为 ModemDecorator 的新类，该类提供了所有的委托代码。于是，实际的装饰器就只需从 ModemDecorator 派生并仅仅覆写那些它们需要的方法即可。图 28.6、程序 28.28 以及程序 28.29 展示了这个结构。

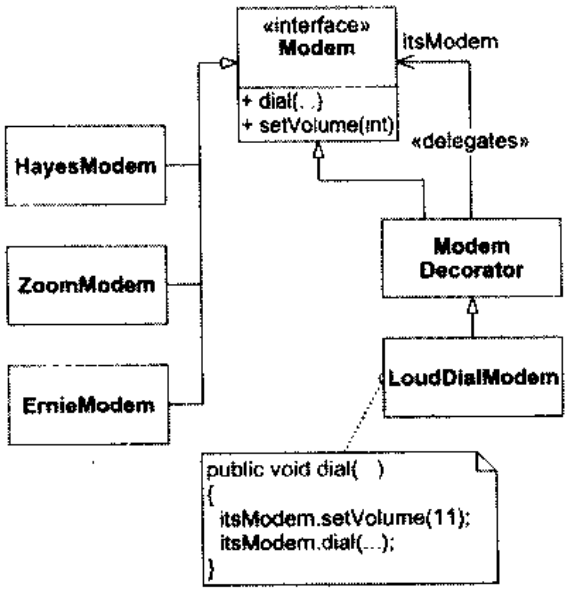


图 28.6 ModemDecorator

程序 28.28 ModemDecorator.java

```
public class ModemDecoratorTest implements Modem
{
    public ModemDecorator(Modem m)
    {
        itsModem = m;
    }

    public void dial(String pno)
```

```
    {
        itsModem.dial(pno);
    }

    public void setSpeakerVolume(int volume)
    {
        itsModem.setSpeakerVolume(volume);
    }

    public String getPhoneNUmber()
    {
        return itsModem.getPhoneNumber();
    }

    public int getSpeakerVolume()
    {
        return itsModem.getSpeakerVolume();
    }

    protected Modem getModem()
    {
        return itsModem;
    }

    private Modem itsModem;
}
```

程序 28.29

```
public class LoudDialModem extends ModemDecorator
{
    public LoudDialModem(Modem m)
    {
        super(m);
    }

    public void dial(String pno)
    {
        getModem().setSpeakerVolume(10);
        getModem().dial(pno);
    }
}
```

28.5 EXTENSION OBJECT 模式

还有另外一种方法可以在不更改类层次结构的情况下向其中增加功能，那就是使用 EXTENSION OBJECT 模式。[1]这个模式虽然比其他的模式复杂一些，但是它也更强大、更灵活一些。层次结构中的每个对象都持有一个特定扩展对象（extension object）的列表。同时，每个对象也提供一个通过名字查找扩展对象的方法。扩展对象提供了操作原始层次结构对象的方法。

例如，再次假设我们有一个材料单系统。我们想让该层次结构中的每个对象都具有创建表示自身的 XML 的能力。我们可以把 toXML 方法放到层次结构中，但是这会违反 SRP。我们不希望把有关 XML 的内容和有关 BOM 的内容放到同一个类中。虽然我们可以使用 VISITOR 模式来创建 XML，但是这无法使我们把针对每种不同类型 BOM 对象的 XML 生成代码分离。在 VISITOR 模式中，针

[1] [PLOPD3]，第 79 页。

对每个 BOM 类的所用 XML 生成代码会在同一个 VISITOR 对象中。如果我们想把针对每种不同 BOM 对象的 XML 生成代码分离到它自己的类中，该怎么办呢？

EXTENSION OBJECT 模式提供了一个实现这个目标的优雅方案。程序 28.30 至程序 28.41 中的代码展示了具有两个不同类型扩展对象的 BOM 层次结构。一种扩展对象把 BOM 对象转换成 XML。另一种扩展对象 BOM 对象转换成 CSV（以逗号分隔的值）字符串。第一种扩展对象通过 `getExtension("XML")` 获得，第二种扩展对象通过 `getExtension("CSV")` 获得。图 28.7 中展示了相应的结构，并且该结构图是根据已经完成的代码绘制的。图中的《marker》构造型表示一个标记接口（也就是没有任何方法的接口）。

我不是完全从零开始编写程序 28.30 至 28.41 中的代码的，知道这一点非常重要。相反，代码是随着一个个测试用例演化而来的。第一个源代码文件（程序 28.30）中展示了所有的测试用例。它们是按照所展示的顺序编写的。每个测试用例都是在还没有任何使之通过的代码的情况下编写的。一旦每个测试用例编写完成并失败了，就去编写使之通过的代码。代码决不会比使现有的测试用例通过所需要的更复杂。这样，代码就以微小增量的方式，从一个可工作的基点演化到另一个可工作的基点。我知道我正在试图构建 EXTENSION OBJECT 模式，并且使用它来指导代码的演化。

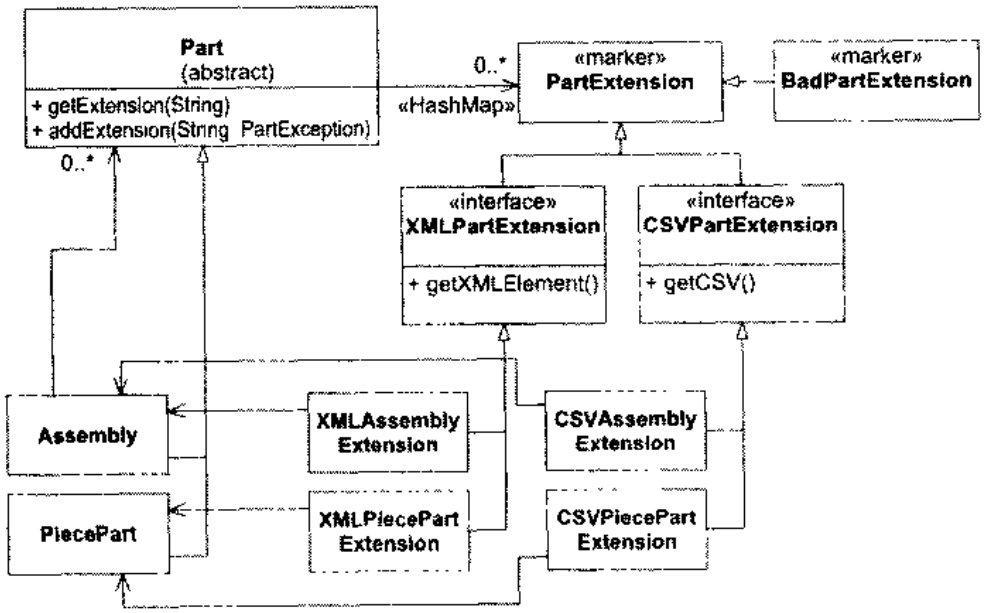


图 28.7　Extension Object

程序 28.30　TestBOMXML.java

```
import junit.framework.*;
import java.util.*;
import org.jdom.*;

public class TestBOMXML extends TestCase
{

    public TestBOMXML(String name)
    {
        super(name);
    }

    private PiecePart p1;
```

```
private PiecePart p2;
private Assembly a;

public void setUp()
{
    p1 = new PiecePart("997624", "MyPart", 3.20);
    p2 = new PiecePart("7734", "Hell", 666);
    a = new Assembly("5879", "MyAssembly");
}

public void testCreatePart()
{
    assertEquals("997624", p1.getPartNumber());
    assertEquals("MyPart", p1.getDescription());
    assertEquals(3.20, p1.getCost(), .01);
}

public void testCreateAssembly()
{
    assertEquals("5879", a.getPartNumber());
    assertEquals("MyAssembly", a.getDescription());
}

public void testAssembly()
{
    a.add(p1);
    a.add(p2);
    Iterator i = a.getParts();
    PiecePart p = (PiecePart)i.next();
    assertEquals(p, p1);
    p = (PiecePart)i.next();
    assertEquals(p, p2);
    assertEquals(i.hasNext() == false);
}

public void testAssemblyOfAssemblies()
{
    Assembly subAssembly = new Assembly("1324", "SubAssembly");
    subAssembly.add(p1);
    a.add(subAssembly);

    Iterator i = a.getParts();
    assertEquals(subAssembly, i.next());
}

public void testPiecePart1XML()
{
    PartExtension e = p1.getExtension("XML");
    XMLPartExtension xe = (XMLPartExtension)e;
    Element xml = xe.getXMLElement();
    assertEquals("PiecePart", xml.getName());
    assertEquals("997624", xml.getChild("PartNumber").getTextTrim());
    assertEquals("MyPart", xml.getChild("Description").getTextTrim());
    assertEquals(3.2,
        Double.parseDouble(xml.getChild("Cost").getTextTrim()),.01);
}

public void testPiecePart2XML()
{
    PartExtension e = p2.getExtension("XML");
```

```
        XMLPartExtension xe = (XMLPartExtension)e;
        Element xml = xe.getXMLElement();
        assertEquals("PiecePart", xml.getName());
        assertEquals("7734", xml.getChild("PartNumber").getTextTrim());
        assertEquals("Hell", xml.getChild("Description").getTextTrim());
        assertEquals(666,
            Double.parseDouble(xml.getChild("Cost").getTextTrim()), .01);
    }

    public void testSimpleAssemblyXML()
    {
        PartExtension e = a.getExtension("XML");
        XMLPartExtension xe = (XMLPartExtension)e;
        Element xml = xe.getXMLElement();
        assertEquals("Assembly", xml.getName());
        assertEquals("5879", xml.getChild("PartNumber").getTextTrim());
        assertEquals("MyAssembly",
            xml.getChild("Description").getTextTrim());
        Element parts = xml.getChild("Parts");
        List partList = parts.getChildren();
        assertEquals(0, partList.size());
    }

    public void testAssemblyWithPartsXML()
    {
        a.add(p1);
        a.add(p2);
        PartExtension e = a.getExtension("XML");
        XMLPartExtension xe = (XMLPartExtension)e;
        Element xml = xe.getXMLElement();
        assertEquals("Assembly", xml.getName());
        assertEquals("5879", xml.getChild("PartNumber").getTextTrim());
        assertEquals("MyAssembly",
            xml.getChild("Description").getTextTrim());

        Element parts = xml.getChild("Parts");
        List partList = parts.getChildren();
        assertEquals(2, partList.size());

        Iterator i = partList.iterator();
        Element partElement = (Element)i.next();
        assertEquals("PiecePart", partElement.getName());
        assertEquals("997624",
            partElement.getChild("PartNumber").getTextTrim());

        partElement = (Element)i.next();
        assertEquals("PiecePart", partElement.getName());
        assertEquals("7734",
            partElement.getChild("PartNumber").getTextTrim());
    }

    public void testPiecePart1CSV()
    {
        PartExtension e = p1.getExtension("CSV");
        CSVPartExtension ce = (CSVPartExtension)e;
        String csv = ce.getCSV();
        assertEquals("PiecePart, 997624, MyPart, 3.2", csv);
    }

    public void testPiecePart2CSV()
```

```
    {
        PartExtension e = p2.getExtension("CSV");
        CSVPartExtension ce = (CSVPartExtension)e;
        String csv = ce.getCSV();
        assertEquals("PiecePart, 7734, Hell, 666.0", csv);
    }

    public void testAssemblyCSV()
    {
        PartExtension e = a.getExtension("CSV");
        CSVPartExtension ce = (CSVPartExtension)e;
        String csv = ce.getCSV();
        assertEquals("Assembly, 5879, MyAssembly", csv);
    }

    public void testAssemblyWithPartsCSV()
    {
        a.add(p1);
        a.add(p2);
        PartExtension e = a.getExtension("CSV");
        CSVPartExtension ce = (CSVPartExtension)e;
        String csv = ce.getCSV();

        assertEquals("Assembly, 5879, MyAssembly", +
                "{PiecePart, 997624, Mypart, 3.2}," +
                "{PiecePart, 7734, Hell, 666.0}"
                , csv);
    }

    public void testBadExtension()
    {
        PartExtension pe =
            p1.getExtension("ThisStringDoesn'tMatchAnyException");
        assert(pe instanceof BadPartExtension);
    }
}
```

程序 28.31 Part.java

```
import java.util.*;

public abstract class Part
{
    HashMap itsExtensions = new HashMap();

    public abstract String getPartNumber();
    public abstract String getDescription();

    public void addExtension(String extensionType, PartExtension extension)
    {
        itsExtensions.put(extensionType, extension);
    }

    public PartExtension getExtension(String extensionType)
    {
        PartExtension pe = (PartExtension) itsExtensions.get(extensionType);
        if (pe == null)
            pe = new BadPartExtension();
        return pe;
    }
}
```

程序 28.32 PartExtension.java

```
public interface PartExtension
{
}
```

程序 28.33 PiecePart.java

```
public class PiecePart extends Part
{

    public PiecePart(String partNumber, String description, double cost)
    {
        itsPartNumber = partNumber;
        itsDescription = description;
        itsCost = cost;
        addExtension("CSV", new CSVPiecePartExtension(this));
        addExtension("XML", new XMLPiecePartExtension(this));
    }

    public String getPartNumber()
    {
        return itsPartNumber;
    }

    public String getDescription()
    {
        return itsDescrition;
    }

    public double getCost()
    {
        return itsCost;
    }

    private String itsPartNumber;
    private String itsDescription;
    private double itsCost;
}
```

程序 28.34 Assembly.java

```
import java.util.*;

public class Assembly extends Part
{

    public Assembly(String partNumber, String description)
    {
        itsPartNumber = partNumber;
        itsDescription = description;
        addExtension("CSV", new CSVPiecePartExtension(this));
        addExtension("XML", new XMLPiecePartExtension(this));
    }

    public void add(Part part)
    {
        itsParts.add(part);
    }

    public Iterator getParts()
    {
```

```
        return itsParts.iterator();
    }

    public String getPartNumber()
    {
        return itsPartNumber;
    }

    public String getDescription()
    {
        return itsDescrition;
    }

    private List itsParts = new LinkedList();
    private String itsPartNumber;
    private String itsDescription;
}
```

程序 28.35 XMLPartExtension.java

```
import org.jdom.*;

public interface XMLPartExtension extends PartExtension
{
    public Element getXMLElement();
}
```

程序 28.36 XMLPiecePartExtension.java

```
import org.jdom.*;

public class XMLPiecePartExtension implements XMLPartExtension
{
    public XMLPiecePartExtension(PiecePart part)
    {
        itsPiecePart = part;
    }

    public Element getXMLElement()
    {
        Element e = new Element("PiecePart");
        e.addContent(
            new Element("PartNumber").setText(
                itsPiecePart.getPartNumber()));
        e.addContent(
            new Element("Description").setText(
                itsPiecePart.getDescription()));
        e.addContent(
            new Element("Cost").setText(
                Double.toString(itsPiecePart.getCost())));
        return e;
    }

    private PiecePart itsPiecePart = null;
}
```

程序 28.37 XMLAssemblyExtension.java

```
import org.jdom.*;
import java.util.*;

public class XMLAssemblyExtension implements XMLPartExtension
```

```
{
    public XMLAssemblyExtension(Assembly assembly)
    {
        itsAssembly = assembly;
    }

    public Element getXMLElement()
    {
        Element e = new Element("Assembly");
        e.addContent(
            new Element("PartNumber").setText(
                itsAssembly.getPartNumber()));
        e.addContent(
            new Element("Description").setText(
                itsAssembly.getDescription()));
        Element parts = new Element("Parts");
        e.addContent(parts);
        Iterator i = itsAssembly.getParts();
        while (i.hasNext())
        {
            Part p = (Part)i.next();
            PartExtension pe = p.getExtension("XML");
            XMLPartExtension xpe = (XMLPartExtension)pe;
            parts.addContent(xpe.getXMLElement());
        }
        return e;
    }

    private Assembly itsAssembly = null;
}
```

程序 28.38 CSVPartExtension.java

```
public interface CSVPartExtension extends PartExtension
{
    public String getCSV();
}
```

程序 28.39 CSVPiecePartExtension.java

```
public class CSVPiecePartExtension implement CSVPartExtension
{
    private PiecePart itsPiecePart = null;

    public CSVPiecePartExtension(PiecePart part)
    {
        itsPiecePart = part;
    }

    public String getCSV()
    {
        StringBuffer b = new StringBuffer("PiecePart,");
        b.append(itsPiecePart.getPartNumber());
        b.append(",");
        b.append(itsPiecePart.getDescription());
        b.append(",");
        b.append(itsPiecePart.getCost());
        return b.toString();
    }
}
```

程序 28.40 CSVAssemblyExtension.java

```
import java.util.Iterator;

public class CSVAssemblyExtension implement CSVPartExtension
{
    private Assembly itsAssembly = null;

    public CSVAssemblyExtension(Assembly assy)
    {
        itsAssembly = assy;
    }

    public String getCSV()
    {
        StringBuffer b = new StringBuffer("Assembly,");
        b.append(itsAssembly.getPartNumber());
        b.append(",");
        b.append(itsAssembly.getDescription());

        Iterator i = itsAssembly.getParts();
        while (i.hasNext())
        {
            Part p = (Part)i.next();
            CSVPartExtension ce = (CSVPartExtension)p.getExtension("CSV");
            b.append(",{");
            b.append(ce.getCSV());
            b.append("}");
        }
        return b.toString();
    }
}
```

程序 28.41 BadPartExtension.java

```
public class BadPartExtension implements PartExtension
{
}
```

请注意，扩展对象是通过每个 BOM 对象的构造函数装入该对象中的。这意味着，在某种程度上 BOM 对象仍然依赖于 XML 类和 CSV 类。如果即使这个轻微的依赖也需要解除的话，我们可以创建一个 FACTORY[①]对象去创建 BOM 对象并装入其扩展对象。

可以向对象中装入扩展对象的能力带来了很大的灵活性。根据系统的状态，可以把某些扩展对象插入到对象中，或者从对象中删除。这种灵活性会很容易使我们失去自制力。在大多数的情况下，你可能都不必去使用它。事实上，PiecePart.getExtension(String extensionType)的最初实现是这样：

```
public PartExtension getExtension(String extensionType)
{
    if (extensionType.equals("XML"))
        return new XMLPiecePartExtension(this);
    else if (extensionType.equals("CSV"))
        return new XMLAssemblyExetnsion(this)

    return new BadPartExtension();;
}
```

① 参见第 21 章中的“FACTORY 模式”。

我对此并不是非常满意，因为它和 Assemblu.getExtension 中的代码实质上是一样的。Part 中使用的 HashMap 方案消除了这个重复并且也更简单一些。任何读过代码的人都可以很清楚地知道是如何获取扩展对象的。

28.6 结　论

VISITOR 模式系列给我们提供了许多无需更改一个层次结构中的类即可修改其行为的方法。因此，它们有助于我们保持 OCP。此外，它们也提供了用来分离不同种类的功能的机制，从而使类不会和很多其他的功能混杂在一起。它们也同样有助于我们保持 CCP。也应该可以清楚地看出，VISITOT 模式系列的结构中也使用了 SRP、LSP 以及 DIP。

VISITOR 模式是有诱惑力的。在它们面前很容易会失去自制力。如果它们有用就去使用它们，但是请对它们的必要性保持健康的怀疑。通常，可以使用 VISITOR 模式解决的问题往往也可以使用更简单的方法解决。

28.6.1 提示

既然已经学习完了本章，你可能想回到第 9 章去解决 shape 排序的问题。

参考文献

1. Gamma.et al.*Design Patterns*.Reading,MA:Addison-Wesley,1995.
2. Martin,Robert C.et al. *Pattern Languages of Program Design 3*. Reading, MA: Addison- Wesley, 1998.

第 29 章　STATE 模式

一个无法改变的状态就无法保持。

——埃德蒙·柏克（1729—1797，英国著名的政治家和保守主义政治理论家）

有限状态自动机是软件宝库中最有用的抽象之一。它们提供了一个简单、优雅的方法去揭示和定义复杂系统的行为。它们同样也提供了一个易于理解、易于修改的有效实现策略。我在系统的各个层面，从控制高层逻辑的 GUI[①]到最低层的通讯协议，都会使用它们。它们几乎适用于任何地方。

29.1　有限状态自动机概述

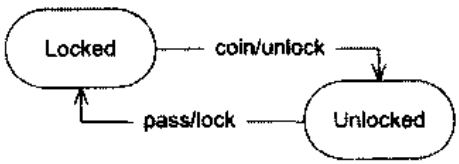


图 29.1　简单的十字转门状态机

在地铁十字转门的操作中，可以找到一个简单的有限状态机（FSM）。正是这个装置控制着乘客通往乘坐地铁列车道路的大门。图 29.1 中展示了控制地铁十字转门的初步 FSM。该图被称为状态迁移图，或者 STD。[②]

STD 至少由 4 部分组成。图中的圆形被称为状态。连接状态的箭头被称为迁移。迁移被用一个后面跟有动作名的事件名做了标记。图 29.1 中的 STD 的含意如下：

- 若状态机在 Locked 状态收到了一个 coin 事件，则迁移到 Unlocked 状态并执行 unlock 动作。
- 若状态机在 Unlocked 状态收到了一个 pass 事件，则迁移到 Locked 状态并执行 lock 动作。

这两句话完全描绘了图 29.1 中的状态图。每句话都用 4 个元素描述了一个迁移箭头：起始状态、触发迁移的事件、终止状态以及要执行的动作。事实上，这些描绘迁移的语句可以被简化为一张称为状态迁移表（STT）的表格。该表格看起来可能像这样：

```
Locked      coin    Unlocked    unlock
Unlocked    pass    Locked      lock
```

① 参见第 30 章中的 30.5 节“TASKMASTER 构架”。
② 参见附录 B 中的“状态和内部迁移”、“状态间的迁移”以及“嵌套状态”小节。

这个状态机如何运转呢？假设 FSM 一开始处于 Locked 状态。一个乘客走向地铁十字转门并投入一枚硬币。这致使软件收到 coin 事件。STT 中的第一项迁移指出，如果在 Locked 状态收到到了 coin 事件，那么就迁移到 Unlocked 状态并执行 unlock 动作。因此，软件把它的状态改为 Unlocked 并且调用 unlock 函数。接着，乘客就通过转门，这又致使软件检测到 pass 事件。因为 FSM 现在处于 Unlocked 状态，所以就会使用第二项变迁，使机器回到 Locked 状态并调用 lock 函数。

显然，STD 和 STT 都对状态机的行为进行了简单、优雅的描述。同时，它们也是非常有效的设计工具。它们可以带来许多好处，其中之一就是使用它们，设计人员可以很容易检测到那些未知的以及没有处理的情况。例如，请检查图 29.1 中每个状态，并对之应用两个已知的事件。请注意，在 Unlocked 状态下没有处理 coin 事件的迁移，并且在 Locked 状态下也没有处理 pass 事件的迁移。

这些遗漏是非常严重的逻辑错误，也是程序员错误的一个非常常见的来源。通常，程序员对正常事件过程的考虑要比对可能出现的异常情况的考虑更彻底一些。STD 或者 STT 给程序员提供了一种方法，使用这种方法可以容易地核实设计在每个状态下都处理了所有的事件。

我们可以通过增加必要的迁移来对 FSM 进行修正。图 29.2 中展示了这个新版本。从中可以看出，如果乘客在首次投币后又多投了一些硬币，那么状态机就保持在 Unlocked 状态并把一个小“thank-you”灯点亮来鼓励乘客继续投币。[①]同样，如果乘客在转门锁着时想通过它（多半会使用一个大铁锤），那么 FSM 会保持在 Locked 状态并响起警报。

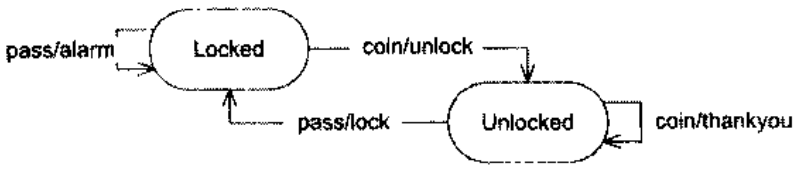


图 29.2　包含有异常事件的十字转门 FSM

29.2　实现技术

29.2.1　嵌套 switch/case 语句

有许多不同的实现 FSM 的策略。第一个，也是最直接的一个策略是使用嵌套 switch/case 语句。程序 29.1 展示了一个这样的实现。

程序 29.1　Turnstile.java（嵌套 Switch Case 实现）

```
package com.objectmentor.PPP.Patterns.State.turnstile;

public class Turnstile
{
  // States
  public static final int LOCKED = 0;
  public static final int UNLOCKED = 1;

  // Events
  public static final int COIN = 0;
  public static final int PASS = 1;
```

① ;^)

```
  /*private*/ int state = LOCKED;

  private TurnstileController turnstileController;

  public Turnstile(TurnstileController action)
  {
    turnstileController = action;
  }

  public void event(int event)
  {
    switch (state)
    {
    case LOCKED:
      switch (event)
      {
      case COIN:
        state = UNLOCKED;
        turnstileController.unlock();
        break;
      case PASS:
        turnstileController.alarm();
        break;
      }
      break;
    case UNLOCKED:
      switch (event)
      {
      case COIN:
        turnstileController.thankyou();
        break;
      case PASS:
        state = LOCKED;
        turnstileController.lock();
        break;
      }
      break;
    }
  }
}
```

嵌套 switch/case 语句把代码分成了 4 个互斥的区域，每个区域对应 STD 中的一项迁移。每个区域在需要时都会更改软件的状态，然后调用相应的动作。例如，关于 Locked 和 Coin 的区域会把状态改为 Unlocked 并调用 unlock

代码中有一些有趣的特征，这些特征和嵌套 switch/case 语句无关。为了更清楚的理解它们，你需要去看一下用来验证该代码的单元测试（参见程序 29.2 和程序 29.3）。

程序 29.2 TurnstileController.java

```
package com.objectmentor.PPP.Patterns.State.turnstile;

public interface TurnstileController
{
  public void lock();
  public void unlock();
  public void thankyou();
  public void alarm();
}
```

程序 29.3 TestTurnstile.java

```
package com.objectmentor.PPP.Patterns.State.turnstile;

import junit.framework.*;
import junit.swingui.TestRunner;

public class TestTurnstile extends TestCase
{
  public static void main(String[] args)
  {
    TestRunner.main(new String[]{"TestTurnstile"});
  }

  public TestTurnstile(String name)
  {
    super(name);
  }

  private Turnstile t;
  private boolean lockCalled = false;
  private boolean unlockCalled = false;
  private boolean thankyouCalled = false;
  private boolean alarmCalled = false;


  public void setUp()
  {
    TurnstileController controllerSpoof = new TurnstileController()
    {
      public void lock() {lockCalled = true;}
      public void unlock() {unlockCalled = true;}
      public void thankyou() {thankyouCalled = true;}
      public void alarm() {alarmCalled = true;}
    };
    t = new Turnstile(controllerSpoof);
  }

  public void testInitialConditions()
  {
    assertEquals(Turnstile.LOCKED, t.state);
  }

  public void testCoinInLockedState()
  {
    t.state = Turnstile.LOCKED;
    t.event(Turnstile.COIN);
    assertEquals(Turnstile.UNLOCKED, t.state);
    assert(unlockCalled);
  }

  public void testCoinInUnlockedState()
  {
    t.state = Turnstile.UNLOCKED;
    t.event(Turnstile.COIN);
    assertEquals(Turnstile.UNLOCKED, t.state);
    assert(thankyouCalled);
  }
```

```
  public void testPassInLockedState()
  {
    t.state = Turnstile.LOCKED;
    t.event(Turnstile.PASS);
    assertEquals(Turnstile.LOCKED, t.state);
    assert(alarmCalled);
  }

  public void testPassInUnlockedState()
  {
    t.state = Turnstile.UNLOCKED;
    t.event(Turnstile.PASS);
    assertEquals(Turnstile.LOCKED, t.state);
    assert(lockCalled);
  }
}
```

1．包范围内有效的状态变量

请读者注意单元测试中的 4 个测试函数：testCoinInLockedState、testCoinInUnlockedState、testPassInLockedState 以及 testPassInUnlockedState。这些函数分别测试了 FSM 的 4 个迁移。在实现中，它们把 Turnstile 的 state 变量强制设为想要检查的状态，然后引发想要验证的事件。为了使测试程序能够访问 state 变量，它就不能是私有的。所以，我让它成为包范围内有效的，并且增加了一个注释来指出我的意图是让这个变量私有。

面向对象法则主张，类的所有实例变量都应该是私有的。我明显没有遵循这个原则，因此我破坏了 Turnstile 的封装。

不这样做的话，该怎么做呢？

毫无疑问，我本可以让 state 变量是私有的。然而，这样做会使测试代码不能强制设置它的值。我当然可以创建相应的包范围内有效的 setState 和 getState，但是这似乎很荒谬。我不想把 state 变量暴露给除 TestTurnstile 以外的其他类，那么我为什么要创建一个设置和一个获取方法，而这两个方法却意味着包范围内有效的任何类都可以获取并设置该变量呢？

Java 中有一个令人遗憾的弱点，它缺少类似 C++中 friend 的概念。如果 Java 中具有 friend 声明，那么我就可以让 state 保持私有并把 TestTurnstile 声明为 Turnstile 的友元。不过，在目前情况下，我认为让 state 成为包范围内有效的并用注释来表明我的意图是最好的选择。

2．测试动作

请注意程序 29.2 中的 TurnstileController 接口。使用该接口就是为了使 TestTurnstile 类可以确保 Turstile 类以正确的顺序调用了正确的动作。如果没有这个接口，那么要确保状态机正确地工作就会非常困难。

这是一个测试影响设计的例子。如果我仅仅去编写状态机而不考虑测试。那么很可能就不会创建 TurnstileController 接口。那样就会很可惜。TurnstileController 接口优雅地解除了有限状态机逻辑和它要执行的动作之间的耦合。这样，另外一个具有完全不同逻辑的 FSM 就可以在没有任何影响的情况下使用 TurnstileController。

如果我们需要隔离地去验证每个功能单元，那么在创建测试代码时，就会迫使我们以在其他情况下可能不会想到的方式解除代码间的耦合。因此，可测试性可以促使设计中具有更少的耦合。

3. 嵌套 Switch/Case 实现的代价和收益

对于简单的状态机来说，嵌套 switch/case 实现既简单又优雅。所有的状态和事件都出现在一、两页代码中。然而，对于大型的 FSM 来说，情况就不同了。在一个具有大量状态和事件的状态机中，代码就退化成一页页的 case 语句。并且没有方便的定位工具帮助你了解正在阅读的是状态机的哪一部分。维护冗长、嵌套的 switch/case 语句是一项非常困难并且容易出错的工作。

嵌套 switch/case 语句实现的另一个代价是在有限状态机的逻辑和实现动作的代码之间没有被很好地分离。在程序 29.1 中明显地显示出了这个分离，因为动作是在 TurnstileController 的一个派生类中实现的。然而，在我见过的大多数使用嵌套 switch/case 实现的 FSM 中，动作的实现都被隐藏在 case 语句中。事实上，在程序 29.1 中也是有可能这样做的。

29.2.2 解释迁移表

一个很常见的实现 FSM 的技术就是创建一个描绘迁移的数据表。该表被一个处理事件的引擎解释。引擎查找与事件匹配的迁移，调用相应的动作，并更改状态。

程序 29.4 展示了创建迁移表的代码，程序 29.5 展示了迁移引擎。这两个程序都是从本章最后的完整实现（程序 29.12）中截取的片段。

程序 29.4 创建十字转门迁移表

```
public Turnstyle(TurnstyleController action)
{
    turnstyleController = action;
    addTransition(LOCKED,   COIN, UNLOCKED, unlock()  );
    addTransition(LOCKED,   PASS, LOCKED,   alarm()   );
    addTransition(UNLOCKED, COIN, UNLOCKED, thankyou());
    addTransition(UNLOCKED, PASS, LOCKED,   lock()    );
}
```

程序 29.5 迁移引擎

```
public void event(int event)
{
    for (int i = 0; i < transitions.size(); i++)
    {
      Transition transition = (Transition) transitions.elementAt(i);
      if (state == transition.currentState && event == transition.event)
      {
        state = transition.newState;
        transition.action.execute();
      }
    }
}
```

解释迁移表的代价和收益

这种实现方法的有一个很大的好处，那就是构建迁移表的代码读来就像一个规范的状态迁移表。其中的 4 行 addTransaction 语句非常易于理解。状态机的逻辑全部集中在一个地方并且没有被动作的实现污染。

和嵌套 switch/case 语句相比，维护这样的一个有限状态机是非常容易的。要增加新的迁移，只

需向 Turnstile 的构造函数中增加一行 addTransaction 语句即可。

该方法的另一个好处是迁移表可以容易地在运行时改变。这样就允许动态地改变状态机逻辑。我曾经使用过类似这样的机制来为复杂的有限状态机打热补丁（hot patching）。

还有另外一个好处是可以创建多个迁移表，每个都代表一个不同的状态机逻辑。这些表可以根据启动条件在运行时进行选择。

该方法的代价主要是速度。对迁移表的遍历要花费时间。对于大型的状态机来说，所花费的时间就会变得相当可观。另外一个代价是要编写大量的代码去支持迁移表。如果你仔细检查程序 29.12，就会看到数量众多的小支持函数，这些函数的目的是为了使程序 29.4 中状态迁移表的简单表达成为可能。

29.3　STATE 模式[①]

还有另外一项实现有限状态机的方法：STATE 模式。该模式既具有嵌套 swith/case 语句的效率又具有解释迁移表的灵活性。

图 29.3 展示该解决方案的结构。Turnstile 类拥有关于事件的公有方法以及关于动作的受保护方法。它持有一个指向 TurnstileState 接口的引用。TurnstileState 的两个派生类代表 FSM 的两个状态。

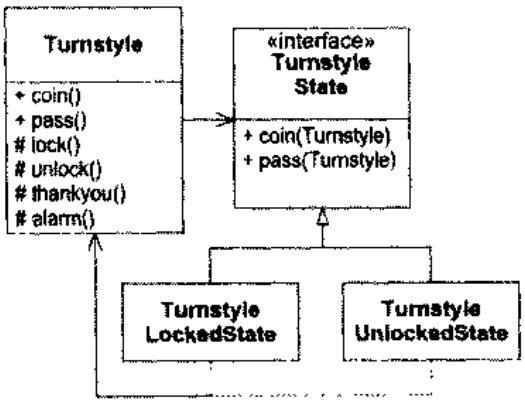


图 29.3　Turnstile 的 STATE 模式解决方案

当 Turnstile 的两个事件方法中的一个被调用时，它就把这个事件委托给 TurnstileState 对象。TurnstileLockedState 的方法实现了 LOCKED 状态下的相应动作。TurnstileUnlockedState 的方法实现了 UNLOCKED 状态下的相应动作。为了改变 FSM 的状态，就要把这两个派生类之一的实例赋给 Turnstile 对象中的引用。

程序 29.6 展示了 TurnstileState 接口以及它的两个派生类。在这两个派生类的 4 个方法中可以容易地对状态机进行访问。例如，LockedTurnstileState 的 coin 方法让 Turnstile 对象把状态改变到 unlocked，然后再调用 Turnstile 的 unlock 动作函数。

程序 29.6　TurnstileState.java

```
interface TurnstileState
{
  void coin(Turnstile t);
  void pass(Turnstile t);
}

class LockedTurnstileState implements TurnstileState
{
  public void coin(Turnstile t)
  {
    t.setUnlocked();
    t.unlock();
  }
```

① [GOF95]，第 305 页。

```
  public void pass(Turnstile t)
  {
    t.alarm();
  }
}

class UnlockedTurnstileState implements TurnstileState
{
  public void coin(Turnstile t)
  {
    t.thankyou();
  }

  public void pass(Turnstile t)
  {
    t.setLocked();
    t.lock();
  }
}
```

程序 29.7 中展示了 Turnstile 类。请注意那些保存 TurnstileState 的派生类实例的静态变量。这些类不具有任何变量，所以永远不会需要有多个实例。把 TurnstileState 的派生类实例保存到成员变量中，是为了避免每次状态变化时，都去创建新的实例。把这些变量声明成静态的，是为了当我们需要多个 Turnstile 实例时，不必去创建新的派生类实例。

程序 29.7 Turnstile.java

```
public class Turnstile
{
  private static TurnstileState lockedState = new LockedTurnstileState();
  private static TurnstileState unlockedState = new UnlockedTurnstileState();

  private TurnstileController turnstileController;
  private TurnstileState state = lockedState;

  public Turnstile(TurnstileController action)
  {
    turnstileController = action;
  }

  public void coin()
  {
    state.coin(this);
  }

  public void pass()
  {
    state.pass(this);
  }

  public void setLocked()
  {
    state = lockedState;
  }

  public void setUnlocked()
  {
    state = unlockedState;
  }
```

```
  public boolean isLocked()
  {
    return state == lockedState;
  }

  public boolean isUnlocked()
  {
    return state == unlockedState;
  }

  void thankyou()
  {
    turnstileController.thankyou();
  }

  void alarm()
  {
    turnstileController.alarm();
  }

  void lock()
  {
    turnstileController.lock();
  }

  void unlock()
  {
    turnstileController.unlock();
  }
}
```

1. STATE 模式和 STRATEGY 模式

图 29.3 中的图示很容易让我们回想起 STRATEGY 模式。[①]这两个模式都有一个上下文类，都委托给一个具有几个派生类的多态基类。不同之处（参见图 29.4）在于，在 STATE 模式中，派生类持有回指向上下文类的引用。派生类的主要功能是使用这个引用选择并调用上下文类中的方法。在 STRATEGY 模式中，不存在这样的限制以及意图。STRATEGY 的派生类不必持有指向上下文类的引用，并且也不需要去调用上下文类的方法。所以，所有的 STATE 模式实例同样也是 STRATEGY 模式实例，但是并非所有的 STRTEGY 模式实例都是 STATE 模式实例。

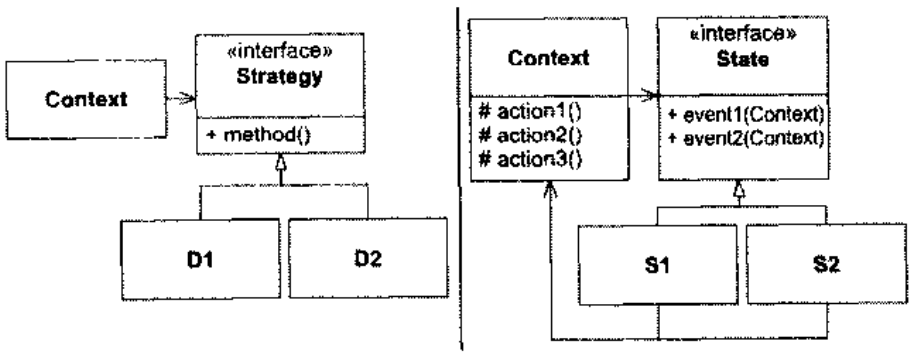


图 29.4 STATE 模式和 STRATEGY 模式

① 参见 14.2 节“STRATEGY 模式”。

2. STATE 模式的代价和收益

STATE 模式彻底地分离了状态机的逻辑和动作。动作是在 Context 类中实现的，而逻辑则是分布在 State 类的派生类中。这就使得二者可以非常容易地独立变化、互不影响。例如，只要使用 State 类的另外一个派生类，就可以非常容易地在一个不同的状态逻辑中重用 Context 类的动作。此外，我们也可以在不影响 State 派生类逻辑的情况下创建 Context 子类来更改或者替换动作实现。

该方法的另外一个好处就是它非常高效。它基本上和嵌套 switch/case 实现的效率完全一样。因此，该方法既具有表驱动方法的灵活性，又具有嵌套 switch/case 方法的效率。

这项技术的代价体现在两个方面。第一，State 派生类的编写完全是一项乏味的工作。编写一个具有 20 个状态的状态机会使人精神麻木。第二，逻辑分散。无法在一个地方就看到整个状态机逻辑。因此，就使得代码难以维护。这会使人想起嵌套 switch/case 方法的晦涩性。

29.3.1　SMC——状态机编译器

为了省去编写派生状态类的乏味工作并把状态机的逻辑放在一个地方表达，我就编写了一个编译器，该编译器可以把一个用文本描述的状态迁移表转变成实现 STATE 模式所必必需的类。该编译器是免费的，可以从 http://www.objectmentor.com 下载。

程序 29.8 中展示了编译器的输入。语法如下所示：

```
currentState
{
    event newState action
    ...
}
```

最上面的 4 行描述了状态机的名字、上下文类的名字、初始状态以及在出现非法事件时会抛出的异常的名字。

程序 29.8　Turnstile.sm

```
FSMName Turnstyle
Context TurnstyleActions
Initial Locked
Exception FSMError
{
    Locked
    {
        coin    Unlocked    unlock
        pass    Locked      alarm
    }
    Unlocked
    {
        coin    Unlocked    thankyou
        pass    Locked      lock
    }
}
```

为了使用这个编译器，你必须编写一个声明了动作函数的类。Context 行中指定了这个类的名字。我把它称为 TurnstileActions（参见程序 29.9）。

程序 29.9 TurnstileActions.java

```
public abstract class TurnstileActions
{
  public void lock() {}
  public void unlock() {}
  public void thankyou() {}
  public void alarm() {}
}
```

编译器生成一个从上下文类派生的类。FSMName 行中指定了生成类的名字。我把它称为 Turnstile。

我本可以在 TurnstileActions 中实现动作函数，不过，我更倾向于编写另外一个类，该类从所生成的类派生并且实现了动作函数。程序 29.10 展示了这种做法。

程序 29.10 TurnstileFSM.java

```
public class TurnstileFSM  extends Turnstile
{
  private TurnstileController controller;
  public TurnstileFSM(TurnstileController controller)
  {
    this.controller = controller;
  }

  public void lock()
  {
    controller.lock();
  }

  public void unlock()
  {
    controller.unlock();
  }

  public void thankyou()
  {
    controller.thankyou();
  }

  public void alarm()
  {
    controller.alarm();
  }
}
```

我们只需要编写这些，SMC 会生成其余的代码。图 29.5 展示了最后得到的结构。我们称其为 3 层有限状态机（THREE-LEVEL FINITE STATE MATCHINE）。[①]

这 3 个层次以非常低的代价提供了最大的灵活性。我们可以创建许多不同的有限状态机，而所需要做的只是让它们从 TurnstileActions 派生。同样，只需从 Turnstile 继承，我们就可以以许多不同的方式实现动作。

请注意，生成的代码完全和你要编写的代码隔离。你根本不必修改生成的代码，甚至都不必去查看它们。你可以把它们看作是二进制代码。

① [PLoPD1]，第 383 页。

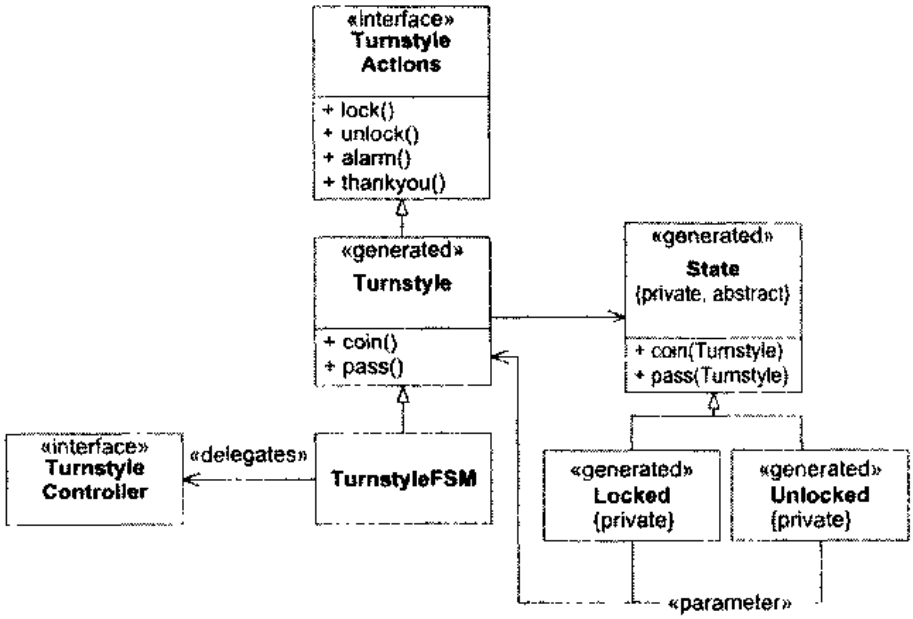


图 29.5　3 层 FSM

在本章最后的程序区中，可以看到针对本例所生成的代码以及其他的支持代码（程序 29.13 至程序 29.15）。

用 SMC 方法生成 STATE 模式的代价和收益

显然，我们已经得到了各种不同方法的最大好处。对有限状态机的描述全部包含在一个地方并且非常易于维护。有限状态机的逻辑彻底和动作实现隔离，使得二者可以独立变化。解决方案高效、优雅并且所需要的编码量最小。

代价在于对 SMC 的使用上。你必须获取另外一个工具并学习如何去使用它。不过，在本例中，所使用的工具非常易于安装和使用（参见程序 29.16 和前一节）。并且它是免费的！

29.4　应该在哪些地方使用状态机

我会把状态机（以及 SMC）用在应用程序的几个不同级别中。

29.5　作为 GUI 中的高层应用策略

20 世纪 80 年代发生了一场图形革命，其目标之一就是要创造出无状态的界面供人们使用。那时候，计算机界面基本上都是文本化的分层菜单。在使用这种界面时，很容易迷失在菜单结构中，而不知道屏幕当前所处的状态。GUI 则通过最小化屏幕状态的变化次数来缓解这个问题。在一些现代的 GUI 中，为了能够把公共特性总是保持在屏幕上，并确保使用者不被隐藏的状态所迷惑，人们做了大量的工作。

具有讽刺意味的是，实现这些“无状态”GUI 的代码本身正是完全由状态驱动的。在这样的 GUI 中，代码必须指出哪些菜单项和按钮要灰色显示，哪个子窗口应该显现出来，哪个标签（tab）要被激活，焦点应被放置在何处，等等。所有这些决策都和界面的状态相关。

很久以前我就认识到，如果不把这些要素组织成单一的控制结构，那么对它们的控制就是一场噩梦。这个控制结构最好被表示为 FSM。从那时起，我几乎在所有 GUI 的编写中都使用由 SMC（或者它的前期版本）生成的 FSM。

请考虑程序 29.11 中的状态机。该状态机控制着应用程序中的用户登录部分。当收到一个启动事件时，状态机就提供一个登录屏幕。一旦使用者敲击了回车键，状态机就去检查口令。如果口令正确，它就进入 loggedIn 状态并启动用户处理过程（没有在此显示）。如果口令错误，它就显示一个屏幕通知使用者口令错误。如果使用者想要再试一次，可以点击确定按钮；否则，就点击取消按钮。如果口令连续输入错误 3 次（thridBadPassword 事件），那么状态机就锁定屏幕直到输入了管理员口令。

程序 29.11 login.sm

```
Initial init
{
    init
    {
        start logginIn displayLoginScreen
    }

    logginIn
    {
        enter checkingPassword checkPassword
        cancel init clearScreen
    }

    checkingPassword
    {
        passwordGood loggedIn startUserProcess
        passwordBad notifyingPasswordBad displayBadPasswordScreen
        thirdBadPassword screenLocked displayLockScreen
    }

    notifyingPasswordBad
    {
        OK checkingPassword displayLoginScreen
        cencel init clearScreen
    }

    screenLocked
    {
        enter checkingAdminPassword checkAdminPassword
    }

    checkingAdminPassword
    {
        passwordGood init clearScreen
        passwordBad screenLocked displayLockScreen
    }
}
```

此处我们所做的就是在状态机中捕获了应用程序的高层策略。这个高层策略集中在一个地方并且易于维护。它极大地简化了系统中的其余代码，因为那些代码不再和策略代码混合在一起。

显然，这个方法也可以用于除 GUI 以外的其他界面处。事实上，我曾经在文本界面以及机器——机器界面上也使用过类似的方法。但是 GUI 往往比它们更加复杂，所以对状态机的需要和使用量也更多些。

29.5.1　GUI 交互控制器

假设你想让使用者在屏幕上画矩形。他们的操作步骤如下：首先，在工具窗口中点击矩形图标。然后，在画布窗口上用鼠标定位出矩形的一个角。接着，按下鼠标键并把鼠标拖拽到所希望的第二个角。在拖拽鼠标时，屏幕上会显示一个可能矩形的活动图示。只要在拖曳鼠标时保持鼠标键按下，就可以把矩形拖曳成想要的形状。当矩形合适时，就释放鼠标键。此时，程序就停止显示活动图并在屏幕上绘制一个固定的矩形。

当然，在任何时候，只要使用者点击一个不同的工具图标，就可以中止这次绘制。如果使用者把鼠标拖曳到画布窗口以外，活动图示就会消失。如果鼠标又回到画布窗口，活动图就会再次出现。

最后，画完了一个矩形后，使用者只要在画布窗口中点击并拖曳就可以画另外一个矩形，而不需要到工具窗口中去点击矩形图标。

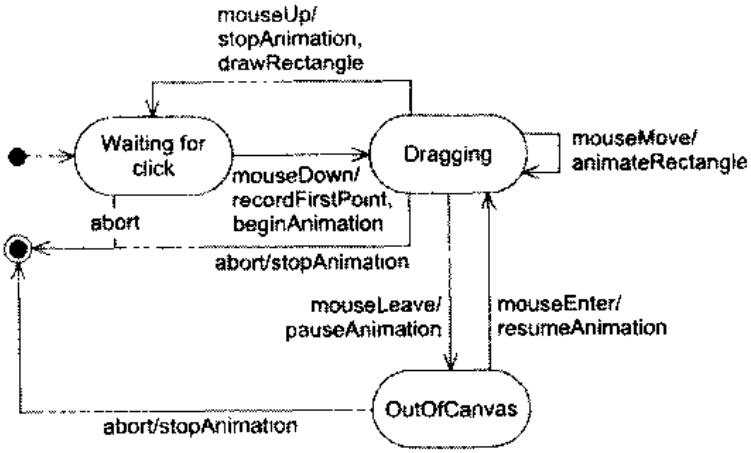


图 29.6　矩形绘制交互状态机

上面所描绘的正是一个有限状态机。图 29.6 中展示了状态迁移图。具有箭头的实心圆表示状态机[①]的起始状态。被空心圆环绕的实心圆是状态机的最终状态。

GUI 交互中有大量有限状态机。它们由使用者的输入事件驱动。这些事件引起交互状态的改变。

29.5.2　分布式处理

还有另外一种情形，其中系统的状态会基于输入的事件而改变，那就是分布式处理。例如，假设你要把一大块信息从网络上的一个节点传送到另一个节点。同样假设网络的响应时间很宝贵，所以需要把信息块分割成一组小包发送。

图 29.7 展示了描述这个场景的状态机。它从请求一个传输会话开始，接着发送每个包并且等待一个确认，最后以终止会话而结束。

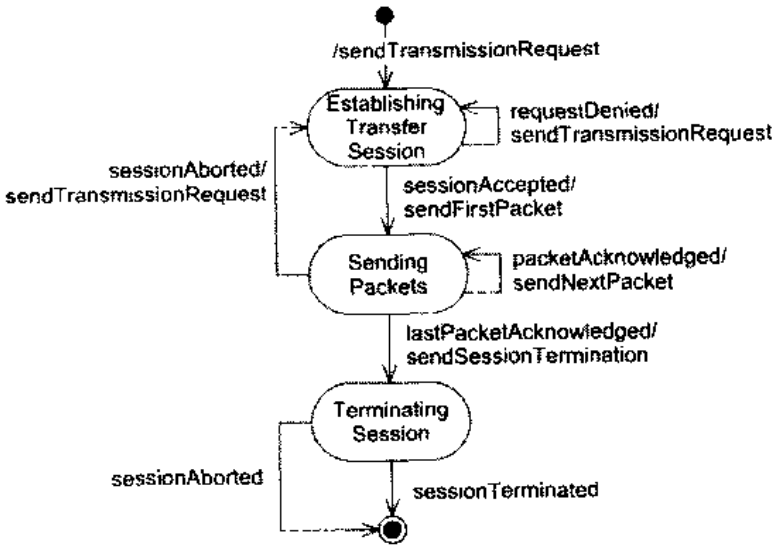


图 29.7　把大包分成多个小包发送

① 参见附录 B 中的“状态和内部迁移”小节。

29.6 结　论

有限状态机并没有被充分使用。在许多情形中，使用它们都会有助于创建更清楚、更简单、更灵活以及更准确的代码。使用 STATE 模式以及根据状态迁移表生成代码的简单工具，可以给我们提供很大的帮助。

29.7 程　序

29.7.1 使用解释表的 Turnstile.java

这个程序展示了如何通过解释一个状态迁移数据结构向量（vector）来实现有限状态机。它和程序 29.2 中的 TurnstileController 以及程序 29.3 中的 TurnstileTest 完全兼容。

程序 29.12　使用解释表的 Turnstile.java

```
import java.util.Vector;

public class Turnstile
{
  // States
  public static final int LOCKED = 0;
  public static final int UNLOCKED = 1;

  // Events
  public static final int COIN = 0;
  public static final int PASS = 1;

  /*private*/ int state = LOCKED;
  private TurnstileController turnstileController;
  private Vector transitions = new Vector();

  private interface Action
  {
    void execute();
  }

  private class Transition
  {
    public Transition(int currentState, int event, int newState, Action action)
    {
      this.currentState = currentState;
      this.event = event;
      this.newState = newState;
      this.action = action;
    }

    int currentState;
    int event;
    int newState;
    Action action;
  }

  public Turnstile(TurnstileController action)
  {
```

```
    turnstileController = action;
    addTransition(LOCKED,   COIN, UNLOCKED, unlock()  );
    addTransition(LOCKED,   PASS, LOCKED,   alarm()   );
    addTransition(UNLOCKED, COIN, UNLOCKED, thankyou());
    addTransition(UNLOCKED, PASS, LOCKED,   lock()    );
  }

  private void addTransition(int currentState, int event,
                             int newState, Action action)
  {
    transitions.add(new Transition(currentState, event, newState, action));
  }

  private Action lock()
  {
    return new Action(){public void execute(){doLock();}};
  }

  private Action thankyou()
  {
    return new Action(){public void execute(){doThankyou();}};
  }

  private Action alarm()
  {
    return new Action(){public void execute(){doAlarm();}};
  }

  private Action unlock()
  {
    return new Action(){public void execute(){doUnlock();}};
  }

  private void doUnlock()
  {
    turnstileController.unlock();
  }

  private void doLock()
  {
    turnstileController.lock();
  }

  private void doAlarm()
  {
    turnstileController.alarm();
  }

  private void doThankyou()
  {
    turnstileController.thankyou();
  }

  public void event(int event)
  {
    for (int i = 0; i < transitions.size(); i++)
    {
      Transition transition = (Transition) transitions.elementAt(i);
      if (state == transition.currentState && event == transition.event)
      {
```

```
        state = transition.newState;
        transition.action.execute();
      }
    }
  }

}
```

29.7.2　由 SMC 生成的 Turnstile.java 以及其他支持文件

程序 29.13 至 29.16[①]是十字转门例子中，使用 SMC 生成的完整代码。Turnstile.java 由 SMC 生成。虽然生成器制造了一点混乱，但是代码却不坏。

程序 29.13　Turnstile.java（由 SMC 生成）

```
//----------------------------------------------
//
// FSM:      Turnstile
// Context:   TurnstileActions
// Exception: FSMError
// Version:
// Generated: Thursday 09/06/2001 at 12:23:59 CDT
//
//----------------------------------------------


//----------------------------------------------
//
// class Turnstile
//    This is the Finite State Machine class
//
public class Turnstile extends TurnstileActions
{
  private State itsState;
  private static String itsVersion = "";

  // instance variables for each state
  private static Locked itsLockedState;
  private static Unlocked itsUnlockedState;

  // constructor
  public Turnstile()
  {
    itsLockedState = new Locked();
    itsUnlockedState = new Unlocked();

    itsState = itsLockedState;

    // Entry functions for: Locked
  }

  // accessor functions

  public String getVersion()
  {
```

① 原书中为 31.13、31.16，有误——译者注。

```
  return itsVersion;
}

public String getCurrentStateName()
{
  return itsState.stateName();
}

// event functions - forward to the current State

public void pass() throws FSMError
{
  itsState.pass();
}

public void coin() throws FSMError
{
  itsState.coin();
}

//-----------------------------------------
//
// private class State
//    This is the base State class
//
private abstract class State
{
  public abstract String stateName();

  // default event functions

  public void pass() throws FSMError
  {
    throw new FSMError( "pass", itsState.stateName());
  }

  public void coin() throws FSMError
  {
    throw new FSMError( "coin", itsState.stateName());
  }

}

//-----------------------------------------
//
// class Locked
//    handles the Locked State and its events
//
private class Locked extends State
{
  public String stateName()
    { return "Locked"; }

  //
  // responds to coin event
  //
  public void coin()
  {
    unlock();
```

```
    // change the state
    itsState = itsUnlockedState;
  }

  //
  // responds to pass event
  //
  public void pass()
  {
    alarm();

    // change the state
    itsState = itsLockedState;
  }
}

//-----------------------------------------------
//
// class Unlocked
//   handles the Unlocked State and its events
//
private class Unlocked extends State
{
  public String stateName()
    { return "Unlocked"; }

  //
  // responds to pass event
  //
  public void pass()
  {
    lock();

    // change the state
    itsState = itsLockedState;
  }

  //
  // responds to coin event
  //
  public void coin()
  {
    thankyou();

    // change the state
    itsState = itsUnlockedState;
  }
}

}
```

如果出现了非法事件，我们会让 SMC 抛出 FSMError 异常。十字转门的例子非常简单，不可能出现非法事件，所以异常是无用的。然而，在大型的状态机中，有一些事件是不应该出现在某些状态中的。这些迁移根本没有在 SMC 的输入中提及。因此，如果出现了这样的事件，生成的代码就会抛出异常。

程序 29.14 FSMError.java

```
public class FSMError extends Exception
```

```
{
  public FSMError(String event, String state)
  {
    super("Invalid event:" + event + " in state:" + state);
  }
}
```

用来测试由 SMC 生成的状态机的代码与我们在本章中已经编写的所有其他测试程序是非常类似的。只有一些微小的差别。

程序 29.15 SMCTurnstileTest.java

```
import junit.framework.*;
import junit.swingui.TestRunner;

public class SMCTurnstileTest extends TestCase
{
  public static void main(String[] args)
  {
    TestRunner.main(new String[]{"SMCTurnstileTest"});
  }

  public SMCTurnstileTest(String name)
  {
    super(name);
  }

  private TurnstileFSM t;
  private boolean lockCalled = false;
  private boolean unlockCalled = false;
  private boolean thankyouCalled = false;
  private boolean alarmCalled = false;


  public void setUp()
  {
    TurnstileController controllerSpoof = new TurnstileController()
    {
      public void lock() {lockCalled = true;}
      public void unlock() {unlockCalled = true;}
      public void thankyou() {thankyouCalled = true;}
      public void alarm() {alarmCalled = true;}
    };

    t = new TurnstileFSM(controllerSpoof);
  }

  public void testInitialConditions()
  {
    assertEquals("Locked", t.getCurrentStateName());
  }

  public void testCoinInLockedState() throws Exception
  {
    t.coin();
    assertEquals("Unlocked", t.getCurrentStateName());
    assert(unlockCalled);
  }
```

```
  public void testCoinInUnlockedState() throws Exception
  {
    t.coin(); // put in Unlocked state
    t.coin();
    assertEquals("Unlocked", t.getCurrentStateName());
    assert(thankyouCalled);
  }

  public void testPassInLockedState() throws Exception
  {
    t.pass();
    assertEquals("Locked", t.getCurrentStateName());
    assert(alarmCalled);
  }

  public void testPassInUnlockedState() throws Exception
  {
    t.coin(); // unlock
    t.pass();
    assertEquals("Locked", t.getCurrentStateName());
    assert(lockCalled);
  }
}
```

TurnstileController 类和本章中所有其他例子中使用的一样。可以参见程序 29.2。

程序 29.16 中展示了用来生成 Turnstile.java 代码的 ant 文件。请注意，它并不是非常重要。如果只是想在 DOS 窗口中键入构建命令，可以键入如下内容：

```
java smc.Smc -f TurnstileFSM.sm
```

程序 29.16 build.xml

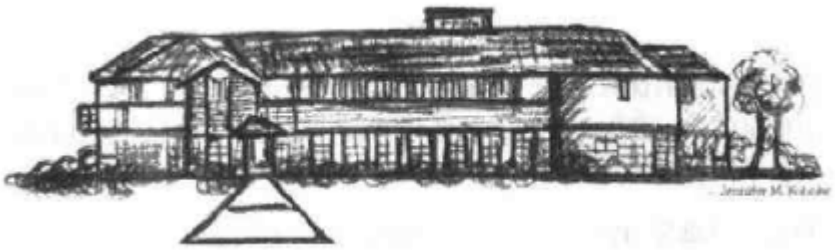
```
<project name="SMCTurnstile" default="TestSMCTurnstile" basedir=".">

    <property environment="env" />

    <path id="classpath">
        <pathelement path="${env.CLASSPATH}"/>
    </path>

    <target name="TurnstileFSM">
        <java classname="smc.Smc">
            <arg value="-f TurnstileFSM.sm"/>
            <classpath refid="classpath"/>
        </java>
    </target>

</project>
```

参考文献

1. Gamma, et al. *Design Patterns*, MA: Addison-Wesley, 1995.
2. Coplien and Schmidt. *Pattern Languages of Program Design* . Reading, MA: Addison- Wesley, 1995.

第 30 章　ETS 框架

本章由 Robert C. Martin 和 James Newkirk 合写



本章描述了一个非常有意义的软件项目，该项目开始于 1993 年 3 月，直到 1997 年末才开发完成。该软件是由教育考试中心（ETS）委托开发的，我们两个以及另外几个 Object Mentor 公司的开发人员参与了开发工作。

在本章中，我们主要关注用于创建可重用框架（framework）的方法，包括技术方面的和管理方面的。创建这样一个框架对于该项目的成功来说是非常重要的一步，并且它的设计以及开发历程同样也具有教育意义。

任何软件项目的开发环境都不可能是完美的，这个项目也不例外。为了从技术角度去理解设计，环境问题的考虑是很重要的。因此，在深入研究项目的软件工程方面的内容前，我们会先介绍一点项目的背景以及它的开发环境。

30.1　介　　绍

30.1.1　项目概述

在美国或者加拿大，要想成为一个有资质的建筑师，必须要通过一个考试。如果通过了考试，就可以得到国家资质委员会颁发的资质证书，这是进行实际的建筑设计所需的。考试是由国家注册建筑师委员会（NCARB）授权的教育考试中心（ETS）来承办的。目前，诚希国际集团（Chauncey Group International）在管理着这项考试。

考试分为 9 个部分，要持续几天的时间。在 3 个绘图部分的考试中，要求应试者在一个类似 CAD 的环境中，通过绘制或者摆放实物来给出他们的答案。例如，可能会要求他们做下面的内容：

- 设计某种建筑的楼层平面图
- 为一个已有的建筑设计合适的屋顶
- 把提供的建筑放置在一片土地上，并设计适合该建筑的停车场、道路系统及人行道系统

过去，应试者对这些问题的解答是使用铅笔和纸绘制出来的。然后，由一个评审中心来对这些解答文档进行评分。评审中心由经验非常丰富的设计师组成，他们会仔细地评阅应试者的答案并决定是否能够通过考试。

1989 年，NCARB 授权 ETS 去研究一下是否可以开发一个自动系统，在这个系统中可以解答考试中的绘图部分内容并对其进行评分。到 1992 年，ETS 和 NCARB 认为这样的系统确实是可以开发出来的。此外，他们觉得由于需求会经常改变，所以使用面向对象方法会比较合适。因此，他们就和 Object Mentor 公司（OMI）联系去帮助他们进行设计。

1993 年 3 月，OMI 获准开发部分的考试软件。一年后，OMI 成功地完成了该部分软件，并又获准开发剩余的大部分软件。

1．程序结构

ETS 所决定的结构相当的优雅。绘图测验被分成 15 个不同的问题，称为绘图题（vignette）。每个绘图题测试一个特定的知识范围。某一个可能会测试应试者对于屋顶设计的理解，而另一个可能会测试他们对于楼层平面设计的理解。

每个绘图题被进一步细分为两部分。“答题（delivery）”部分提供了一个图形用户界面，应试者可以使用该界面“绘制”手边问题的解决方案。“评分（scoring）”部分会读取答题部分做出的解答并给它评分。答题部分会被安置在应试者附近的地点。考试完毕后，解答会被传送到评分中心。

2．试卷脚本

虽然只有 15 个绘图题，但是每个绘图题都会有许多可能的“脚本（script）”。脚本确定了应试者要解决的问题的要旨。例如，对楼层平面绘图题来说，它可能会有一个要求应试者设计一个图书馆的脚本，并且可能还会有一个要求应试者设计一个食品杂货商店的脚本。因此，必须要以通用的方式去编写绘图题程序。这些绘图题的解答以及评分必须要在脚本的控制下进行。

3．平台

答题和评分程序都运行在 Windows3.1 中（后来升级至 Win95/NT）。程序是用 C++语言采用面向对象方法编写的。

4．首次签约

1993 年 3 月，OMI 获准开发所有绘图题中最复杂部分的答题和评分软件：“建筑设计”。之所以做出这个决定，是因为考虑到 Booch 的建议：首先开发最高风险的部分，可以作为一种管理风险以及校对团队评估过程的方法。

5．建筑设计

建筑设计要测试应试者设计一个相对简单的两层建筑的楼层平面图的能力。会给应试者提供一个要设计的建筑，包括需求和约束。然后，应试者要使用答题程序去摆放房间、门、窗、走廊、楼梯以及电梯来进行解答。

接着，评分程序会用能够评估出应试者知识的大量“考察点（feature）”去核对答案。这些考察点的性质是保密的，但是大体上，它们会评估诸如下面的一些内容：

- 建筑满足客户的需求吗？
- 符合建筑规范吗？
- 应试者表明了设计逻辑吗？
- 建筑以及它的房间的朝向正确吗？

30.1.2　早期历程：1993—1994

在开发的早期，只有我们两个人（Martin 和 Newkirk）工作于这个项目。我们和 ETS 的合同规定我们要开发出建筑设计的答题程序和评分程序。不过，我们想在开发建筑设计软件的同时也开发一个可重用的框架。

到 1997 年，必须要有 15 个绘图题的答题和评分程序投入使用。这项任务要在 4 年内完成。我们觉得一个可重用的框架对于完成这个目标会有很大的帮助。这样的一个框架在保持绘图题软件的一致性和质量方面同样也会大有帮助。毕竟，我们不想让不同绘图题软件中的相似特性操作起来会有细微的差别。

因此，在 1993 年 3 月，我们开始开发建筑设计的两个部分，以及一个可以被剩余 14 个绘图题重用的框架。

成功

1993 年 9 月，我们完成了答题和评分程序的第一个版本，并向 NCARB 和 ETS 的代表演示了这些程序。演示获得成功，并计划在 1994 年 1 月进行现场试验（field trial）。

和大多数项目一样，一旦使用者看到了实际运行的程序，他们就会意识到他们所要求的并不是他们真正想要的。在整个 1993 年中，我们每周都会给 ETS 发送一个绘图题程序的中间版本，并且到 9 月份的演示时，我们已经做了大量的更改和增加。

演示结束后，现场试验迫在眉睫，所以更改和增加的次数也因此而激增。我们两人一直专职忙于更改和测试，为现场测试作准备。

现场测试的结果更进一步加剧了关于建筑设计的规格说明书的变动，也使得我们在整个 1994 年第一季度更加繁忙。

1993 年 12 月，我们开始谈判有关构建其余绘图题软件的合同。谈判持续了 3 个月。1994 年 3 月，ETS 同意 OMI 开发一个框架以及另外 10 个绘图题软件。ETS 自己的工程师会基于我们的框架开发剩余的 5 个绘图题软件。

30.1.3　框架

1993 年末，也是建筑设计的需求变动最频繁的时期，为了让 ETS 的一个工程师能够完成即将签署的合同中规定的工作，我们中的一个（Newkirk）花费了一周时间和该工程师一起为之作准备。目标是演示一下如何重用具有 60 000 行代码的可重用 C++框架去构建其他绘图题程序。然而，事情进展的并不顺利。到周末时，结果已经很清楚，重用框架的惟一方法就是把它的源代码零零碎碎地剪切并粘贴到新绘图题程序中。这显然不是一个好方法。

后来我们认识到，有两个导致创建可用的框架失败的原因。第一，我们仅仅关注了建筑设计而没有考虑所有其他的绘图题。第二，在这几个月内，我们一直经受着需求的变动以及时间压力。这两件事情一起把那些特定于建筑设计的概念混入到框架中。

在某种意义上，我们想当然地认为只要使用面向对象技术就肯定会带来好处。我们觉得，在使用 C++语言并仔细地做一个面向对象的设计情况下，创建一个可重用的框架会很容易。我们错了。我们从中学到的东西数年前大家就已经知道了——构建可重用的框架是困难的。

30.2 框 架

1994 年 3 月，在签署了新合同后，我们向项目又增加了两个工程师并开始开发新的绘图题软件。我们仍然相信我们需要一个框架，并且深信目前的框架无法满足要求。显然，我们需要改变策略。

1994 年时的团队组成：

- Robert C.Martin：构架师、主要设计人员，20 年以上经验
- James W.Newkirk：设计人员、项目领导者，15 年以上经验
- Bhama Rao：设计人员、程序员，12 年以上经验
- William Mitchell：设计人员、程序员，15 年以上经验

30.2.1 最后期限

考试软件要在 1997 年投入使用，最后期限就是据此设定的。应试者要在 2 月份参加考试，并在 5 月份进行评分。这个要求必须被满足。

30.2.2 策略

为了按时完成项目并且为了确保可以维持程序的质量和一致性，我们采用了一个新的构建框架的策略。我们虽然部分保留了原来 60 000 行框架的代码，但是大部分的代码被丢弃。

一个被舍弃的方案

一种做法是试图先对框架进行重新设计并在开始开发任何绘图题程序前完成该框架。事实上，许多人会认为这就是构架驱动的方法。然而，我们放弃了这个方案，因为这会导致我们产生大量的框架代码，而这些代码却不能在实际使用的绘图题程序中测试。简而言之，我们认为必须得尽快在实际使用的绘图题程序中使用构架，以对它进行验证。我们不想猜测。

Rebecca Wirfs-Brock 曾经说过：“一个框架必须要在至少 3 个或者更多的应用程序中锤炼（然后把它们丢弃）之后，你对已经为那个领域构建了正确的构架的自信才是理性的”。[①]在经历了一次构建框架的失败后，我们深有同感。因此，我们决定在开发框架的同时也开发几个新的绘图题程序。这样，我们就可以对绘图题程序中的相似特性进行比较，并且以通用和可重用的方式去设计这些特性。

开始时，我们并行开发 4 个绘图题程序。在开发它们的过程中，发现了某些相似的部分。此时，就把这些部分重构成更通用的形式并对其他 3 个绘图题程序进行修整。因此，任何代码除非已经成功重用于至少 4 个绘图题程序，否则是不可能成为框架的一部分的。

此外，我们从建筑设计绘图题程序中拿出了部分代码并以相似的方式对它们进行了重构。一旦这些部分可以在其他 3 个绘图题程序中使用，就把它们放到框架中。

被加入到框架中的一些公共特性如下：

① [BOOCH-OS]，第 275 页。

- UI 屏幕的结构——消息窗口、绘图窗口、按钮工具板，等等
- 图形元素的创建、移动、调整、识别以及删除
- 缩放和滚动
- 简单草图元素的绘制，比如：线、圆以及折线
- 绘图题时间设定以及自动中止
- 答案文件的保存和复原，包括错误恢复
- 许多几何元素的数学模型：线、射线、线段、点、正方形和长方形、圆、弧、三角形、多边形，等等。这些模型包含有一些方法，如：交叉、面积、IsPointIn、IsPointOn，等等
- 个别评分考察点的评估和权重

在接下来的 8 个月中，框架逐渐增长至 60 000 行 C++代码，这是稍微多于一个人年的直接努力的结果。 但是，这个框架却重用在 4 个不同的绘图题程序中。

30.2.3 结果

1. 丢弃一个

我们如何来处理原来的建筑设计绘图题程序呢？随着框架的增长以及在新绘图题程序中的成功使用，原来的建筑设计绘图题程序越来越显得异样。它不同于所有其他的绘图题程序，必须要以不同的方式去维护和演化。尽管建筑设计绘图题程序花费了大约一个人年的努力，我们还是决定狠下心来，完全丢弃老的版本。我们决定在项目开发的后期对它进行重新设计和重新实现。

2. 漫长的初期开发

我们采用的框架策略的一个负面作用，是在开发头几个绘图题程序时花费了相对较长的时间。头 4 个绘图题交付程序几乎需要 4 个人年去开发。

3. 重用效率

在最初的几个绘图题程序完成时，框架已经具有 60 000 行 C++代码，而绘图题交付程序却相当小。每个程序大约具有 4 000 行的样板（boiler-plate）代码（也就是对每个绘图题程序都相同的代码）。此外，每个程序还平均具有 6 000 行的特定于应用的代码。最小的绘图题程序具有 500 行特定于应用的代码，最大的具有 12 000 行。平均起来，每个绘图题程序中几乎 5/6 的代码是框架代码，我们觉得这是非常出色的。这些程序中只有 1/10 的代码是特定的。

4. 开发生产率

在完成头 4 个绘图题程序后，开发时间就急剧下降。仅用了 18 个人月就完成了另外 7 个交付程序（包括重写建筑设计）。这些新绘图题程序的代码行生产率基本和头 4 个相同。

此外，建筑设计程序，第一次开发时花费了我们大约一个人年的工作量，而再次使用框架重新从头编写时，只花费了 2.5 个人月。生产率几乎是 6:1 的增长。

还有另外一种看待这些结果的方法，头 5 个绘图题程序，包括建筑设计，每个都需要一个人年的努力。然而，随后的绘图题程序每个却只需要 2.6 个人月——几乎增长了 400%。

5. 每周交付

从项目一开始，并且在整个开发活动中，我们每周都会给 ETS 提供一些中间版本。ETS 会测试

并评价这些版本，然后发送给我们一个更改列表。我们会对这些更改做出评估，然后和 ETS 一起来确定它们会在哪周交付。通常，我们会优先处理那些高优先级的更改，而推迟处理那些难的或者不太重要的更改。因此，ETS 自始至终都保持对项目以及时间安排的控制。

6. 健壮和灵活的设计

我们对该项目最满意的一个方面，是构架和框架经受住了需求的剧烈变化。在开发的高潮期间，每周都会有一个长长的更改和调整列表要确认。有些更改是要修正 bug，但是更多的是由于实际需求的变化引起的。但是，在艰苦的开发中期，尽管有这么多的修改、干预以及调整，但是“软件的设计却没有被破坏”。[①]

7. 最终成果

到 1997 年 2 月，建筑师应试者开始使用交付的程序进行他们的注册考试。到 1997 年 5 月，程序开始对他们的考试结果评分。系统从那时起一直使用到现在，并且工作良好。现在，每个北美洲的建筑师应试者都使用这个软件进行他们的考试。

30.3　框架设计

30.3.1　评分应用程序的公共需求

请考虑一下有关如何测试某人的知识和技能的问题。ETS 为 NCARB 程序所采用的方案相当的精细。在此，我们可以通过研究一个简单的虚构例子来说明这一点，该例子是基础数学测试。

在我们的基础数学测试中，会给学生出 100 道数学题，这些题的范围包括简单的加、减法以及大数乘法、长除法。我们会根据他们对这些问题的回答来确定其在基础数学方面的能力和技巧。我们的目标是要授予他们一个通过/未通过等级。“通过”意味着我们确信他们已经具备了基础数学所需要的知识和技能。“未通过”意味着我们确信他们不具备这种知识和技能。对于那些无法肯定的情形，我们会把成绩设定为“不确定”。

不过，还有另外一个目标。我们希望能够列举出学生的强项和弱项。我们想把基础数学的主要内容划分成一些子内容，然后针对每个子内容对学生进行评测。

例如，如果一个学生所学习的乘法口诀（fact）是不正确的。也许，他总是把 7×8 误算为 42。这个学生会把大部分的乘法和除法问题弄错。该学生当然不应通过测试。另一方面，假设该学生把其他的所有部分都做对了！该学生在长乘法中有部分的步骤是正确的并且正确地组织了长除法问题。事实上，该学生所犯的惟一错误就是把 7×8 算成了 42。我们当然想知道这一点。事实上，由于很容易改正这种学生所犯的错误，所以我们也许非常有理由让该学生通过并附上一些改正指导。

那么，我们怎样来构造测试的评分方法，才能确定学生具有基础数学中的哪些部分的专业知识以及不具有哪些部分的专业知识呢？请考虑一下图 30.1 中的图示。图中的矩形表示我们想要测试的专业知识的范围，线表示层次化的依赖关系。因此，基础数学的知识依赖于项（term）和因数（factor）的知识。项的知识依赖于加法口诀的知识以及加、减法机制方面的知识。加法的知识依赖于加法的交换、结合性质方面的知识以及进位的机制。

① Pete Brittingham，ETS NCARB 项目的管理者。

叶子矩形被称为“考察点”。考察点是可以被考察的知识单元，根据考察的结果可以赋予它们合格（A）、不合格（U）或者不确定（I）。因此，对于我们的 100 道题以及学生的回答，我们想把每个考察点应用于每个问题中并确定出成绩。对于“进位”考察点来说，我们会察看每道加法题并和学生的回答做比较。如果学生把所有的加法题都做对了，“进位”考察点的考察结果肯定是“A”。但是，对于学生做错的每一道加法题，我们都会试图确定错误是否发生在进位上。我们会试着组合不同的进位错误来确定是否某个错误会导致学生给出的答案。如果具有很高的概率可以确定是一个进位错误，那么我们就会相应地调整进位考察点的得分。最后，进位考察点所返回的得分是一个统计结果，该结果是基于由于进位错而导致的错误答案的总数得出的。

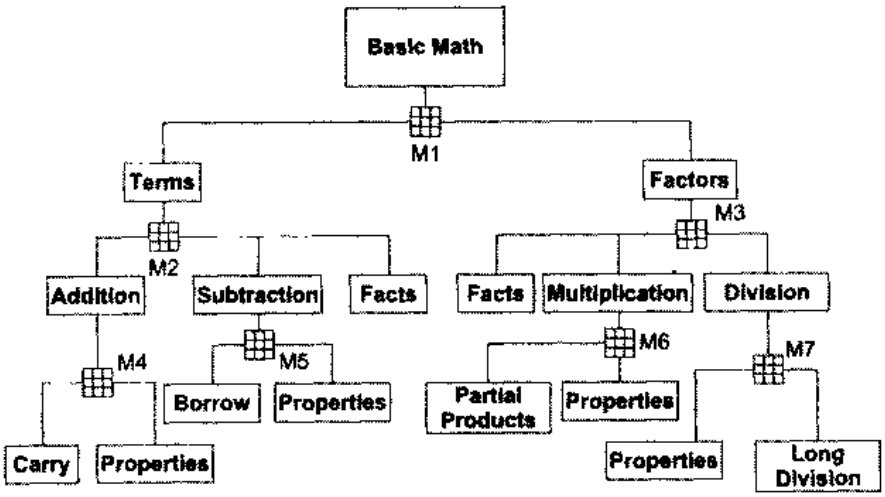


图 30.1 基础数学的考察点层次结构

例如，如果学生做错一半的加法题，并且大部分的错误都可以定位到进位错，那么对于进位考察点我们肯定会给出“U”。另一方面，如果只有 1/4 的错误能被定位为进位错，那么我们会给出“I”。

最后，所有的考察点都以这种方式进行评测。每个考察点都对测试答案进行检查并得出一个关于该特定考察点的成绩。各种不同考察点的成绩代表了对学生基础数学知识的分析。

下一步是要从分析中得出最后的等级。为了完成这一点，我们使用权重和矩阵来沿着层次结构向上合并考察点的成绩。请注意，在图 30.1 中，层次结构的层之间的结合处都存在有矩阵图标。该矩阵把一个加权系数和每个考察点的成绩关联起来，并且提供了一个层次结构中关于该层的成绩的映射关系。例如，紧挨着加法节点下面的矩阵会设立应用于进位和性质考察点成绩的权重，并且会描述用来计算加法的综合成绩的映射关系。

图 30.2 中展示了这些矩阵中的一个的样子。来自进位的输入被认为比来自性质的输入更重要一些，所以赋予了它两倍的权重。然后把加权后的成绩加在一起，并把结果应用于矩阵。

U

I	0	1	2	3
0	A	I	U	U
1	A		U	
2	I	U		
3	U			

Inputs:
Carry X 2
Properties X 1

Output:
Addition

图 30.2 Addition 矩阵

例如，我们假设来自进位的成绩为‘I’，来自性质的成绩为‘A’。因为没有成绩‘U’，所以使用矩阵的最左边的列。‘I’的加权成绩为 2，所以使用矩阵的第 3 行，结果为‘I’。请注意，矩阵中有一些空单元。它们表示不可能出现的情形。对于给定的当前权值来说，不会出现选中矩阵中空单元的成绩的组合。

这种加权矩阵的方案会在层次结构的每一层中重复应用，直到得出最后的成绩。因此，最后成绩是各种不同的考察点成绩的合并以及再合并的结果。这种层次结构非常适合于 ETS 的心理测试专家对其局部进行非常精细的调整。

30.3.2　评分框架的设计

图 30.3 中展示了评分框架的静态结构。可以把结构划分成两个主要部分。右边以不同的字体显示的 3 个类不是框架的组成部分。它们表示必须要针对每个特定的评分应用程序编写的类。图 30.3 中其余的类都是框架类，所有的评分应用程序都共用这些类。

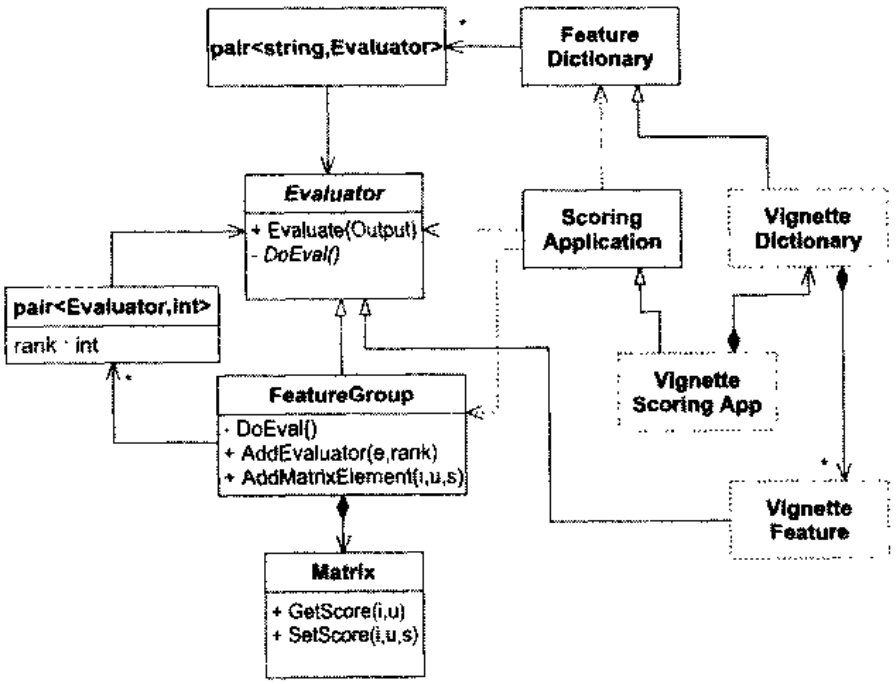


图 30.3　评分框架

评分框架中最重要的类是 Evaluator。该类是一个抽象类，它既代表了评分树中的叶子结点又代表了评分树中的矩阵结点。当要计算评分树中一个结点的成绩时，就调用 Evaluate(ostream&)。为了提供一种标准的方法把成绩记录到输出设备中，该函数使用了 TEMPLATE METHOD 模式。[①]

程序 30.1　Evalutator

```
class Evaluator
{
    public:
        enum Score {A,I,U,F,X};
        Evaluator();
        virtual ~Evaluator();

        Score Evaluate(ostream& scoreOutput);
        void SetName(const String& theName) {itsName = theName;}
        const String& GetName() {return itsName;}
```

① [GOF95]，第 325 页。

```
    private:
        virtual Score DoEval() = 0;

        String itsName;
};
```

请看程序 30.1 和程序 30.2。其中 Evaluate()函数调用了一个私有的、纯虚函数 DoEval()。该函数会被覆写来对评分树中的结点进行实际的评分。Evaluate()把它返回的评分结果以标准的形式输出。

程序 30.2 Evaluator::Evaluate

```
Evaluator::Score Evaluator::Evaluate(ostream& o)
{
    static char scoreName[] = {'A', 'I', 'U', 'F', 'X'};
    o << itsName << ":";
    score = DoEval();
    o << scoreName[score] << endl;
    return score;
}
```

图 30.3 中的 VignetteFeature 类代表着评分树中的叶子结点。事实上，在每个评分应用程序中会有许多这样的类。每个类都会覆写 DoEval()来为自己特定的评分考察点计算成绩。

图 30.3 中的 FeatureGroup 类代表着评分树中的矩阵结点。程序 30.3 中展示了该类的代码。在 FeatureGroup 对象的创建中，有两个函数会提供帮助。第一个是 AddEvaluator，第二个是 AddMatrixElement。

程序 30.3 FeatureGroup

```
class FeatureGroup : public Evaluator
{
    public:
        FeatureGroup(const RWCString& name);
        virtual ~FeatureGroup();

        void AddEvaluator(Evaluator* e, int rank);

        void AddMatrixElement(int i, int u, Score s);
    private:
        Evaluator::Score DoEval();
        Matrix      itsMatrix;
        Vector<pair<Evaluator*, int> > itsEvaluators;
};
```

AddEvaluator 函数可以把子结点增加到 FeatureGroup 中，例如，请回顾一下图 30.1，Addition 结点是一个 FeatureGroup，并且我们要调用两次 AddEvaluator 才能把 Carry 和 Properties 结点加入其中。在 AddEvaluator 函数中可以指定评分器（evaluator）的等级。等级是一个系数，它会应用于评分器的评分结果。因此，当调用 AddEvaluator 把 Carry 增加到 Addition FeatureGroup 中时，我们会指定一个值为 2 等级，因为 Carry 考察点的权重是 Properties 考察点权重的两倍。

AddMatricElement 函数向矩阵中增加了一个单元。必须针对每个需要填充的单元调用该函数。例如，图 30.2 中的矩阵要使用程序 30.4 中的调用序列来创建。

程序 30.4 创建 Addition 矩阵

```
addition.AddMatrixElement(0,0,Evaluator::A);
addition.AddMatrixElement(0,1,Evaluator::I);
```

```
addition.AddMatrixElement(0,2,Evaluator::U);
addition.AddMatrixElement(0,3,Evaluator::U);
addition.AddMatrixElement(1,0,Evaluator::A);
addition.AddMatrixElement(1,2,Evaluator::U);
addition.AddMatrixElement(2,0,Evaluator::I);
addition.AddMatrixElement(2,1,Evaluator::U);
addition.AddMatrixElement(3,0,Evaluator::U);
```

DoEval 函数只是遍历评分器列表，把它们的成绩乘以等级并把乘积增加到相应的关于 I 和 U 分数的累加器中。遍历完成后，它把这些累加器作为矩阵的索引去获取最后的分数（参见程序 30.5）。

程序 30.5 FeatureGroup::DoEval

```
Evaluator::Score FeatureGroup::DoEval()
{
    int sumU, sumI;
    sumU = sumI = 0;
    Evaluator::Score s, rtnScore;
    Vector<Pair<Evaluator*, int> >::iterator ei;
    ei = itsEvaluators.begin();

    for(; ei != itsEvaluators.end(); ei++)
    {
        Evaluator* e = (*ei).first;
        int rank = (*ei).second;

        s = e.Evaluate(outputStream);

        switch(s)
        {
            case I:
                sumI += rank;
            break;
            case U:
                sumU += rank;
            break;
        }
    } // for ei
    rtnScore = itsMatrix.GetScore(sumI, sumU);
    return rtnScore;
}
```

还有最后一个问题。如何构建评分树呢？很明显，ETS 的心理测验专家会希望能够在无需更改实际应用程序的情况下，就可以更改评分树的结构和权重。因此，评分树是由 VignetteScoringApp 类构建的。

每个评分应用程序对于该类都有自己的实现。该类的一个职责是构建一个 FeatureDictionary 的派生对象。该类包含有一个从字符串到 Evaluator 指针的映射。

当评分应用程序启动时，评分框架就获取控制权。它调用 ScoringApplication 类的方法来创建适当的 FeatureDictionary 派生对象。然后，它读取一个描述评分树结构及其权重的专门文本文件。该文本文件中使用专用名字来标识考察点。FeatureDictionary 中和相应的 Evaluator 指针关联的正是这些名字。

因此，在最简单的情况下，评分应用程序只是一组考察点以及一个构造 FeatureDictionary 的方法。评分树的构建和评分工作是由框架处理的，因此对所有的评分应用程序来说是公用的。

30.4　TEMPLATE METHOD 模式的一个例子

有一个绘图题是测试应试者设计建筑的楼层平面的能力，比如：图书馆或警察局。在此绘图题中，应试者必须绘制房间、走廊、门、窗、墙壁开口、楼梯、电梯，等等。程序把绘制的图转换成评分程序可以理解的数据结构。对象模型看起来像图 30.4。

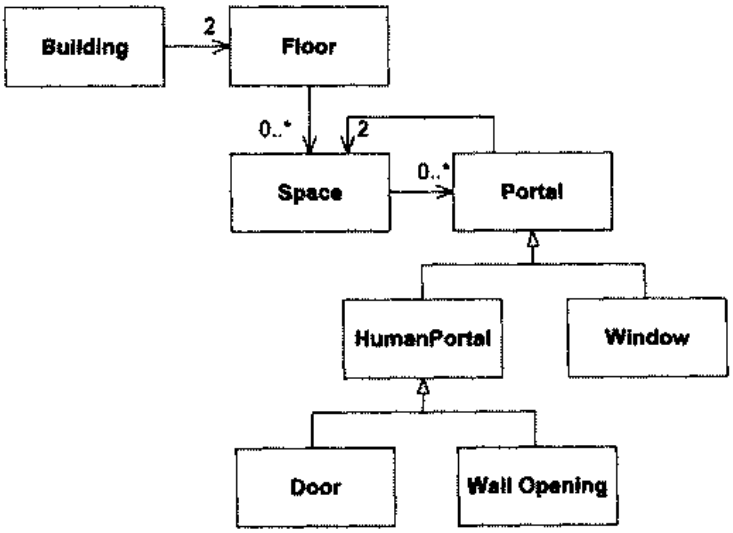


图 30.4　楼层平面数据结构

毫无疑问，该数据结构中的对象具有微乎其微的功能性。它们根本不是多态对象。相反地，它们是一些简单的数据载体——即纯粹的表示模型（representational model）。

图中的建筑有两个楼层组成。每个楼层都有许多空间。每个空间都包含许多入口，每个入口都把两个空间分开。入口可以是窗户或者可以让人通过。人能通过的入口要么是墙壁开口；要么是门。

评分是通过检查针对一组考察点的解答进行的。考察点是像下面的一些东西：

- 应试者画出所有要求的空间了吗？
- 每个空间都具有合格的纵横比吗？
- 每个空间都有入口吗？
- 对外的空间有窗户吗？
- 有门把男女卫生间连接起来吗？
- 从主管办公室可以看到山的全景吗？
- 从厨房可以容易地到达后巷（back alley）吗？
- 从餐厅可以容易地到达厨房吗？
- 可以通过走廊系统到达每个房间吗？

ETS 的心理测试专家希望能够容易地对评分矩阵进行调整。他们希望能够更改权重、把考察点重新组织到不同的子层次中，等等。他们希望能够去掉他们认为没有价值的考察点或者增加新考察点。大部分的操作只需要通过更改一个文本配置文件即可。

出于对性能的考虑，我们只想计算矩阵中所包含的考察点。因此，我们针对每个考察点都创建了类。每个 Feature 类都有一个 Evaluate 方法，该方法会遍历图 30.4 中的数据结构并计算出成绩。这意味着我们具有许许多多的 Feature 类，它们都遍历相同的数据结构。代码重复是令人恐怖的。

30.4.1　只编写一次循环

为了消除代码循环，我们开始使用 TEMPLATE METHOD 模式。这发生在 1993 和 1994 年，那时我们还远不知道什么是模式。我们称我们的做法为“只编写一次循环”（参见程序 30.6 和程序 30.7）。这些是从该程序中取出的实际 C++模块。

程序 30.6 solspcft.h

```
/* $Header: /Space/src_repository/ets/grande/vgfeat/
solspcft.h,v 1.2 1994/04/11 17:02:02 rmartin Exp $ */

#ifndef FEATURES_SOLUTION_SPACE_FEATURE_H
#define FEATURES_SOLUTION_SPACE_FEATURE_H

#include #scoring/eval.h"

template <class T> class Query;

class SolutionSpace;
//-------------------------------------------------
// Name
//  SolutionSpaceFeature
//
// Description
//  This class is a base class which provides a loop which
//  scans through the set of solution spaces and then
//  finds all the solution spaces that match it. Pure virtual
//  functions are provided for when a solution space are found.
//

class SolutionSpaceFeature: public Evaluator
{
    public:
        SolutionSpaceFeature(Query<SolutionSpace*>&);

        virtual ~SolutionSpaceFeature();
        virtual Evaluator::Score DoEval();
        virtual void NewSolutionSpace(const SolutionSpace&) = 0;
        virtual Evaluator::Score GetScore() = 0;

    private:
        SolutionSpaceFeature(const SolutionSpaceFeature&);
        SolutionSpaceFeature& operator= (const SolutionSpaceFeature&);

        Query<SolutionSpace*>& itsSolutionSpaceQuery;
}
#endif
```

程序 30.7 splspcft.cpp

```
/* $Header: /Space/src_repository/ets/grande/vgfeat/
solspcft.cpp,v 1.2 1994/04/11 17:02:00 rmartin Exp $ */

#include "componen/set.h"

#include "vgsolut/solspc.h"
#include "componen/query.h"
#include "vgsolut/scfilter.h"
#include "vgfeat/solspcft.h"

extern ScoringFilter* GscoreFilter;

SolutionSpaceFeature::SolutionSpaceFeature(Query<SolutionSpace*>& q)
: itsSolutionSpaceQuery(q) {}

SolutionSpaceFeature::~SolutionSpaceFeature() {}

Evaluator::Score SolutionSpaceFeature::DoEval()
{
    Set<SolutionSpace*>& theSet = GscoreFilter->GetSolutionSpaces();
```

```
    SelectiveIterator<SolutionSpace*>ai(theSet,itsSolutionSpaceQuery);

    for (; ai; ai++)
    {
        SolutionSpace& as = **ai;
        NewSolutionSpace(as);
    }
    return GetScore();
}
```

从注释头中可以看出，该代码编写于 1994 年。因此，对那些习惯于使用 STL 的人来说，它看起来有些奇怪。不过，如果忽略那些无关的信息（cruft）和奇异的迭代器，就会从中看出标准的 TEMPLATE METHOD 模式。DoEval 函数循环遍历所有的 SolutionSpace 对象。然后，它调用纯虚函数 NewSolutionSpace。SolutionSpaceFeature 的派生类实现了 NewSolutionSpace 并根据特定的评分标准度评测每个空间。

SolutionSpaceFeature 的派生类包含了用来评测答案中的空间是否合适，空间是否具有合适的面积和纵横比，电梯摆放是否合适等等的考察点。

这种做法的优雅之处在于遍历数据结构的循环被放置在一个地方。所有的评分考察点都是通过继承得到它而不是重新实现它。

有些考察点必须评测依附于一个空间的入口的特征。所以，我们再次使用该模式，并创建了派生自 SolutionSpaceFeature 的类 Portalfeature。在 PortalFeature 的 NewSolutionSpace 的实现中遍历了 SolutionSpace 参数中的所有入口，并调用了纯虚函数 NewPortal(const Portal&)（参见图 30.5）。

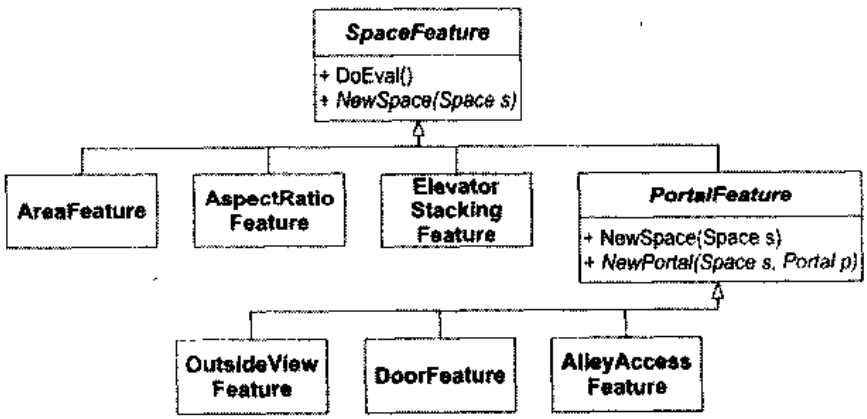


图 30.5 评分考察点的 TEMPLATE METHOD 模式结构

使用该结构，我们可以创建许多不同的评分考察点，每个考察点都可以在不知道楼层平面数据结构的情况下去遍历它们。如果楼层平面数据结构的细节变化了（例如，我们决定使用 STL 来代替自己的迭代器），那么我们只需更改 2 个类，而不是几十个。

为什么我们选择了 TEMPLATE METHOD 模式而不是 STRATEGY 模式呢？[①]请想一想，如果使用了 STRATEGY 模式，耦合就会弱多少呀（参见图 30.6）！

使用 TEMPLATE METHOD 结构时，如果必须更改遍历数据结构的算法，那么就必须更改 SpaceFeature 和 PortalFeature。这很可能还会迫使我们重新编译所有的考察点类。然而，使用 STRATEGY 模式时，更改就会被限制在两个 Driver 类中。事实上，根本不需要重新编译考察点类。

① 显然，我们不是 这些术语来思考的。当我们做这个决定时，模式的名字还没有被创造出来。

那么我们为什么选择了 TEMPLATE METHOD 模式呢？因为它更简单一些。因为数据结构并不会频繁的改变。还因为编译所有的考察点类只需要花费几分钟的时间。

因此，即使 TEMPLATE METHOD 模式中继承关系的使用会导致设计中的耦合关系更紧一些，并且即使 STRATEGY 模式比 TEMPLATE METHOD 模式更好地符合 DIP，最后我们还是觉得不值得创建两个额外的类去实现 STRATEGY 模式。

30.4.2　要交付的应用程序的公共需求

要交付的程序有相当一部分的重叠。例如，表示屏幕的结构在所有绘图题程序中都是相同的。屏幕的左边是一个只包含一列按钮的窗口。该窗口被称为“命令窗口”。命令窗口中的按钮可以用来控制应用程序。按钮上面被标注着一些词语，比如：“放置项（place item）”、“擦除”、“移动/调整”、“缩放”以及“完成”。在这些按钮上单击会驱动应用程序完成所希望的行为。

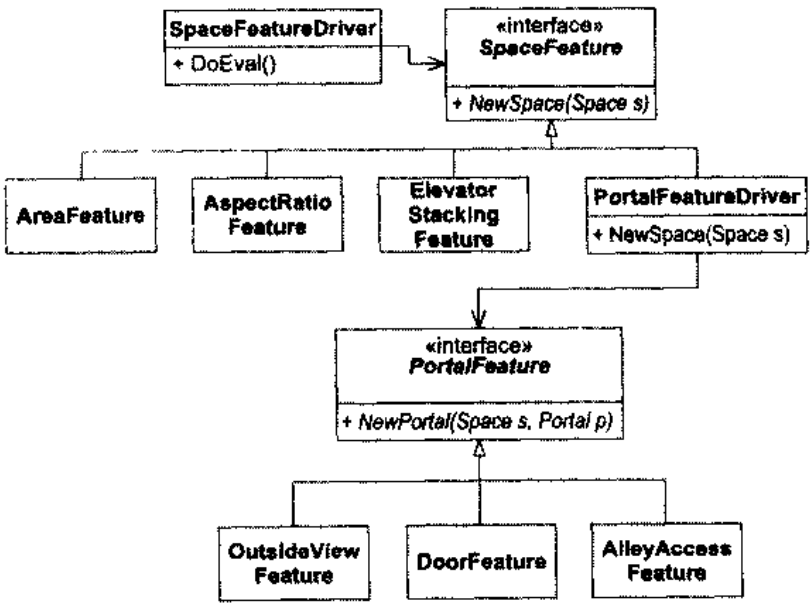


图 30.6　使用 STRATEGY 模式的楼层平面评分结构

命令窗口的右边是任务（Task）窗口。这是一个可滚动、可缩放的大区域，使用者可以在其中绘制他的答案。一般来说，在命令窗口中所触发的命令都会被用于修改任务窗口中的内容。事实上，大多数在命令窗口中触发的命令都需要和任务窗口进行大量的交互。

例如，为了在楼层平面中放置一个房间，使用者要单击命令窗口中的放置项按钮。此时，会弹出一个可选房间的菜单。使用者会选择他想放置在楼层平面中的房间种类。然后，使用者会把鼠标移进任务窗口并在他想放置房间的位置单击。对某些绘图题来说，这也许会把房间的左上部锚定在使用者单击的地方。接着，会出现一个可拉伸的房间，它的左下部会随着鼠标在任务窗口中移动，直到使用者第二次单击，把左下部锚定在那个位置上。

在每个绘图题程序中，这些动作虽然不完全相同，但它们是相似的。有些绘图题不涉及房间，但却涉及登高线、地界线或者屋顶。虽然有些不同，但是在绘图题程序中大体上的操作样式（paradigm）却非常的相似。

这种相似性意味着我们具有一个相当大的重用机会。我们应该能够创建一个面向对象框架，该框架捕获了大部分的相似性并且可以方便地表达不同之处。我们在这个方面取得了成功。

30.4.3　交付框架的设计

ETS 框架最后增长至差不多 75 000 行代码。显然，我们不能在此展示该框架的所有细节。因此，我们选择了框架中两个最具说明性的部分来进行探讨：事件模型以及 taskmaster 构架

1．事件模型

使用者所采取的每个动作都会引发一个事件。如果使用者单击一个按钮，那么就会产生一个以该按钮命名的事件。如果使用者选择了一个菜单项，那么就会产生一个以该菜单项命名的事件。对于框架来说，对这些事件进行编列（marshal）是一个非常重要的问题。

之所以称之为问题，是因为非常大一部分事件可以由框架类处理，但是每个单独的绘图题程序却需要覆写框架对一个特定事件的处理方法。因此，我们需要找到一个方法来让绘图题程序具有在需要时覆写事件处理的能力。

事件列表不封闭的事实使问题变得复杂。每个绘图题程序都可以选择它自己特定的命令窗口中的按钮集以及自己特定的菜单项集。因此，框架需要对所有绘图题程序共有的事件进行编列，同时还要允许每个绘图题程序可以覆写缺省的处理；并且需要允许绘图题程序编列特定于绘图题程序本身的事件。这不是一件容易完成的任务。

作为例子，请考虑图 30.7。该图[①]展示了有限状态机的一小部分内容，该部分内容用来编列出现在绘图题程序命令窗口中的事件。对于这个有限状态机，每个绘图题程序都有自己特定的版本。

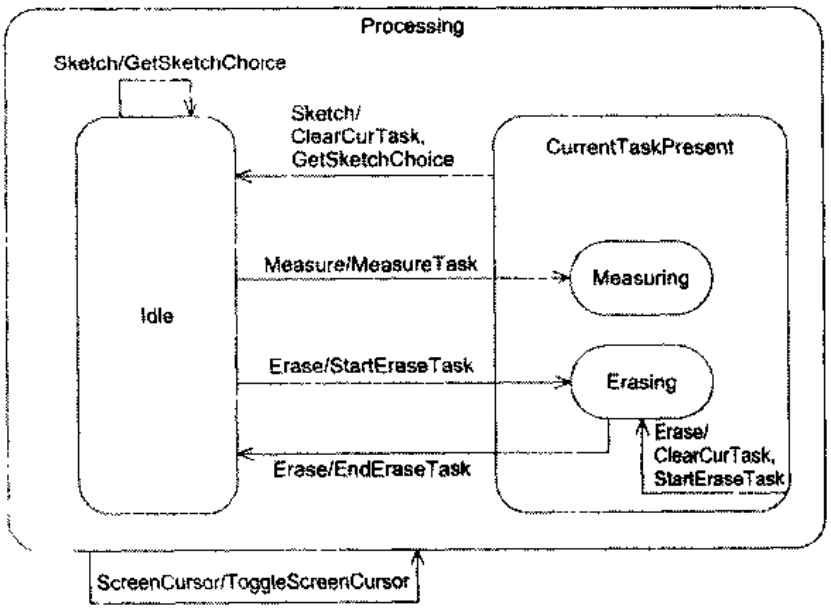


图 30.7　命令窗口事件处理器

图 30.7 展示了 3 种不同事件的行为方式。我们首先来考虑一下最简单的情况，ScreenCursor 事件。在使用者单击更改光标按钮时会产生这个事件。每当使用者单击这个按钮时，任务窗口中的光

① 附录 B 中彻底描述了类似这样的状态图表示法。

标就在箭头和全屏的十字交叉线之间转换。因此，虽然光标的状态改变了，但是事件处理器中的状态却没有发生任何变化。

当使用者想删除一个他已经绘制的物体时，他就单击擦除按钮。然后，他在任务窗口中想要删除的一个或者多个项上面单击。最后，他再次单击擦除按钮提交删除操作。图 30.7 中的状态机展示了命令窗口事件处理器对此是如何处理的。第一次擦除事件引发一个从 Idle 状态到 Erasing 状态的迁移，并且启动擦除任务。我们会在下一小节更详细的讲述有关任务的内容。现在，你只要知道擦除任务会处理发生在任务窗口中的任何事件就可以了。

请注意，即使在 Erasing 状态下，ScreenCursor 事件依然会正确的工作并且不会影响到擦除操作。同样请注意，有两个离开 Erasing 状态的方法。如果出现了另外一个擦除事件，那么就结束擦除任务、提交擦除结果并且状态机迁移回 Idle 状态。这是结束擦除操作的正常方法。

另外一种结束擦除操作的方法是，单击命令窗口中的其他一些按钮。当擦除行为正在进行时，如果单击命令窗口中启动不同任务的按钮（如草图按钮），则擦除任务就会中止，并且会取消删除。

图 30.7 中展示了在发生 Sketch 事件时取消过程是如何完成的，但是还有许多工作方式相同的其他事件并没有在图中展示出来。如果使用者点击了画草图（Sketch）按钮，此时不管系统是处于 Erasing 状态还是处于 Idle 状态，系统都会迁移到 Idle 状态，并调用 GetSketchChoices 函数。该函数弹出草图菜单，其中包含有一个用户可以执行的操作列表。其中的一个操作就是测量。

当使用者从草图菜单中选择了测量项时，会出现 Measure 事件。这会启动测量任务。在测量时，使用者可以在任务窗口中单击两个点。这两个点会被标记上细小的十字线，并且它们之间的距离被报告在屏幕底部的一个小消息窗口中。然后，使用者可以点击另外两个点，再点击另外两个点，再点击另外两个点，等等。没有退出测量任务的正常方法。相反，使用者必须单击可以启动另外一个任务的按钮，比如：擦除或者画草图。

2. 事件模型设计

图 30.8 中展示了实现命令窗口事件处理器的类的静态模型。右边的层次结构表示 CommandWindow，而左边的层次结构表示把事件转换成动作的有限状态机。

在图 30.8 中，CommandWindow、StandardCommandWindow 以及 StandardFsm 是框架类。其余的是特定于绘图题的类。CommandWindow 提供了标准动作的实现，比如，MeasureTask 以及 EraseTask。

事件被 VignetteCommandView 接收。它们被传递给有限状态机，有限状态机把它们转换成动作。然后动作被传回 CommandWindow 类层次结构，在该层次结构中实现了这些动作。

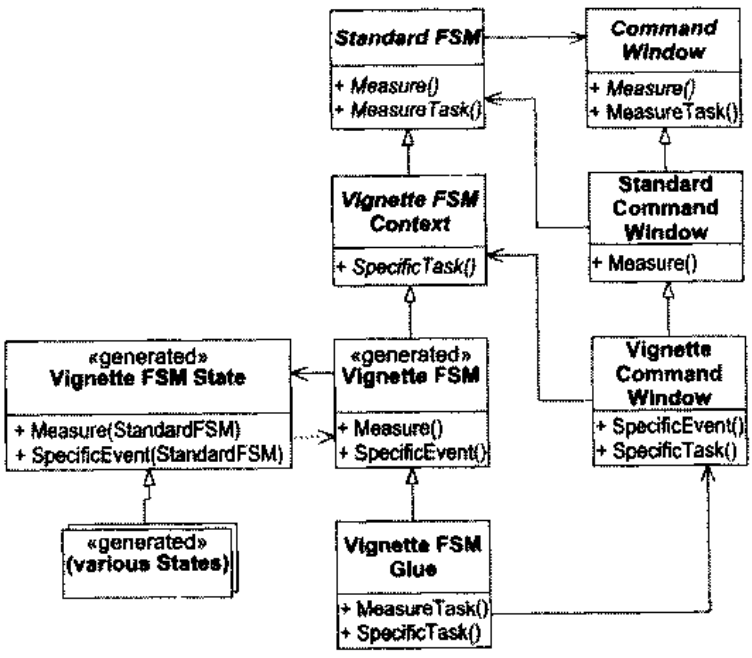


图 30.8　命令窗口事件处理器的静态模型

CommandWindow 类提供了标准动作的实现，比如：MeasureTask 以及 EraseTask。“标准动作”是指所有绘图题程序共有的操作。StandardCommandWindow 把完成了针对有限状态机接收的标准事件的编列。VignetteCommandWindow 是特定于绘图题的，并且它既提供了特定动作的实现又提供了对特定事件的编列。此外，它也允许对标准的实现和编列方法进行覆写。

因此，框架提供了所有任务的缺省实现和编列。但是，绘图题程序可以覆写这些实现或者编列中的任何一个。

3. 跟踪标准事件

图 30.9 中展示了如何把一个标准事件编列进有限状态机并转换成标准的动作。消息 1 是一个 Measure 事件。它由 GUI 产生并且被传递给 VignetteCommandWindow。由于 StandardCommandWindow 提供了对于该事件的缺省编列方法，所以它在消息 1.1 中把该事件转发给 StandardFsm。

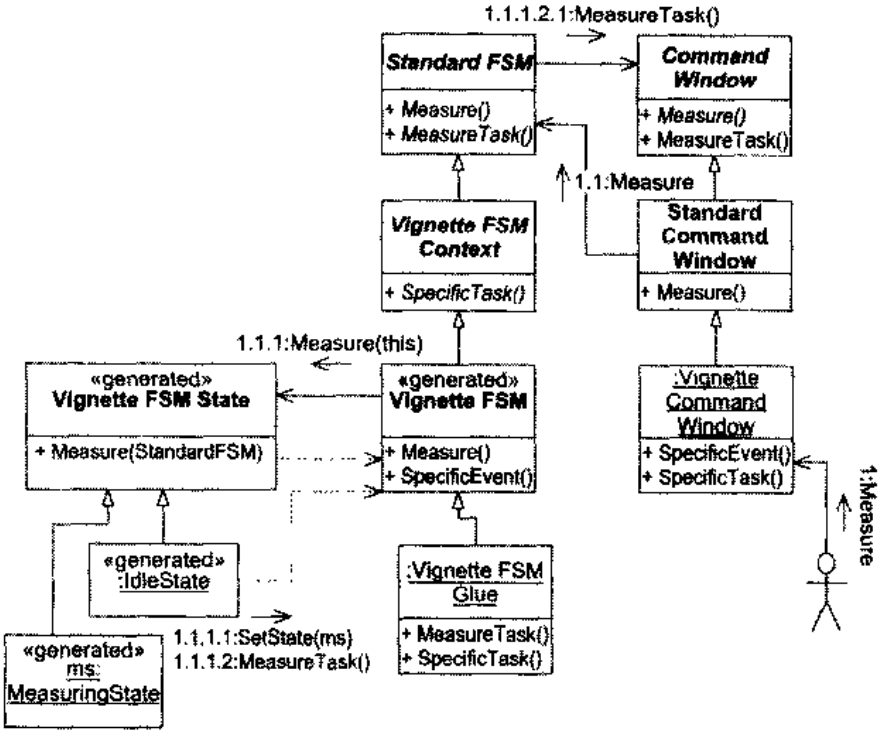


图 30.9　Measure 事件的处理

StandardFsm 是一个框架类，它提供了一个针对所有输入的标准事件以及所有输出的标准动作的接口。在这个层面，这些函数都没有实现。VignetteFSMContext 类向接口中增加了一些针对特定绘图题的事件和动作，但是仍然没有实现。

在 VignetteFSM 类以及 VignetteFSMState 类中完成了把事件转换成动作的实际工作。VignetteFSM 中包含了所有事件函数的实现。于是，1.1:Measure 消息就被向下分发到这个层面。作为响应，VignetteFSM 向 VignetteFSMState 对象发送 1.1.1:Measure(this)消息。

VignetteFSMState 是一个抽象类。对于有限状态机中的每个状态，都会有这个类一个派生类与之对应。在图 30.9 中，我们假设 FSM 的当前状态是 Idle（参见图 30.7）。于是，1.1.1:Measure(this)消息就被分发给 IdleState 对象。作为响应，该对象回送给 VignetteFSM 两个消息。第一个消息是 1.1.1.1:SetState(ms)，它把 FSM 的状态改变到 Measuring 状态。第二个消息是 1.1.1.2:MeasireTask()，它是

Idle 状态下收到 Measure 事件时要执行的动作。

MeasureTask 消息最后在 VignetteFSMGlue 类中实现，该类把该动作当做一个在 CommandWindow 中声明的标准动作，因此在消息 1.1.1.2.1:MeasureTask 中把它发送给 CommandWindow，从而结束环游。

把事件转换成动作采用的机制是 STATE 模式。我们在框架中的许多地方都充分利用了该模式，在下面的小节中会看到这一点。STATE 模式的类中出现的《generated》构造型表示这些类是由 SMC 自动生成的。

4．跟踪特定于绘图题的事件

图 30.10 中展示了当出现一个特定于绘图题的事件时发生的情况。VignetteCommandWindow 再一次捕获了消息 1:SpecificEvent。然而，由于对特定事件的编列是在这个层面实现的，所以就由 VignetteCommandWindow 来发送消息 1.1:SpecificEvent，并且把它发送给最初声明 SpecificEvent 方法的 VignetteFSMContext 类。

事件再一次被分发给 VignetteFSM，该类通过消息 1.1.1:SpecificEv 和当前的状态对象协商产生出对应的动作。像前面一样，状态对象以两条消息：1.1.1.1:SetState 以及 1.1.1.2:SpecificTask 作为回应。

同样，消息被向下分发给 VignetteFSMGlue。不过，这次，该消息被认为是一个绘图题程序的特定动作，因此被直接发送到实现特定动作的 VignetteCommandWindow。

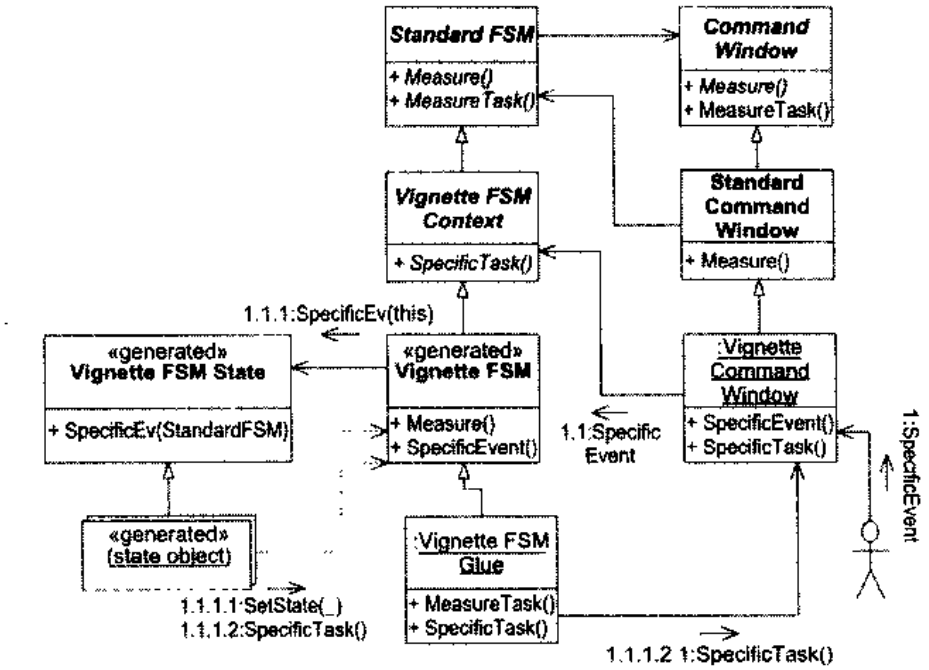


图 30.10　特定事件的处理

5．产生并重用命令窗口状态机

此时，你也许会觉得奇怪，为什么要用这种方式来编列事件和动作呢？它需要的类太多了。但是，请考虑一下，虽然需要很多的类，但是需要的对象却非常少。事实上，被实例化的对象只有 VignetteCommandWindow、VignetteFSMGlue 及各种不同的状态对象，它们都微不足道且是被自动生成的。

消息流看起来似乎很复杂，但是实际上却相当简单。窗口检测到一个事件，把它传递给 FSM，FSM 把事件转换成动作，并把动作传回给窗口。其余的部分只是为了把框架中的标准动作和绘图题

程序中的特定动作分离。

我们决定以这种方式来划分类，还受另外一个因素的影响，那就是我们使用 SMC 来自动地生成有限状态机的类。请考虑一下下面的描述，并回头查阅一下图 30.7：

```
Idle
{
    Measure Measuring   MeasureTask
    Erase   Erasing     StartEraseTask
    Sketch  Idle        GetSketchChoice
}
```

请注意，这个简单的文本描述了状态机在 Idle 状态下可能会发生的所有迁移。大括号中的 3 行指出了触发迁移的事件、迁移的目的状态以及迁移执行的动作。

SMC[1]接受这种形式的文本并产生标记有《generated》的类。无需对 SMC 产生代码进行编辑和任何检查。

使用 SMC，可以非常简单地创建出绘图题程序事件处理器中的状态机代码。开发者必须编写 VignetteCommandWindow 以及其中对应于特定事件和动作的实现。此外，开发者也必须编写 VignetteFSMContext，该类只是声明了关于特定事件和动作的接口。然后，开发者要编写 VignetteFSMGlue 类，该类只是把动作发回给 VignetteCommandWindow。所有这些任务都不是特别复杂。

开发者还必须做另外一件事情。他必须为 SMC 编写有限状态机的描述。实际上，这个状态机相当复杂。图 30.7 中的图示完全和实际不符。一个真正的绘图题程序必须处理许多不同的事件，每个事件都可以具有非常不同的行为。

幸运的是，大多数绘图题程序的行为方式大致相同。因此，我们可以把一个标准的有限状态机描述作为模型，并对每个绘图题程序都做相对较小的修改。这样，每个绘图题程序就具有它自己的 FSM 描述。

这种方法稍微有些不令人满意，因为 FSM 的描述文件间非常的相似。事实上，有些时候，我们必须得更改通用的状态机，这意味着我们必须得对每个 FSM 描述文件都做出相同或者几乎相同的更改。这项工作既单调乏味又容易出错。

我们本可以想出另外的方案来分离 FSM 描述的通用部分和特定部分，但是最终我们认为不值得在这上面花费精力。因为这个决定，我们已经不止一次责怪自己了。

30.5 TASKMASTER 构架

我们已经看到事件是如何被转换成动作的，并且看到了这个转换是如何通过一个相对复杂的有限状态机完成的。现在，我们要关注一下如何来处理动作本身。每个动作的核心部分也是由有限状态机驱动的，对于这一点，我们不应该感到奇怪。

我们来研究一下上一小节中讨论的 MeasureTask。当使用者想测量两点间的距离时就启动这个任务。当该任务启动时，如果使用者点击 TaskWindow 中的一个点，该点处会出现一个小的十字交叉线。接着，当使用者来回移动鼠标时，就会绘制一条从点击点到鼠标当前位置的可拉伸的线。此外，该线的当前长度被显示在一个不同的消息窗口中。当使用者第 2 次点击时，会出现另外一个十字交叉

① SMC（状态机编译器）是一个免费软件，可以到 http://www.objectmentor.com 下载。

线，可拉伸的线消失了，并且这两点间最后的距离被显示在消息框中。如果使用者接着再次点击，该过程就重新开始。

一旦事件处理器选取了 MeasureTask（如图 30.9 中所示），CommandWindow 就创建实际的 MeasureTask 对象，接着该对象就启动有限状态机（如图 30.11 所示）。

要启动 MeasureTask，可以调用它的 init 函数，接着它就进入 GetFirstPoint 状态。TaskWindow 中发生的 GUI 事件被发送到当前正在运行的任务。因此，当使用者在 TaskWindow 中移动鼠标时，当前任务就会接收到 MovePoint 消息。请注意，在 GetFirstPoint 状态中，该事件不会引发任何动作（这也是所期望的）。

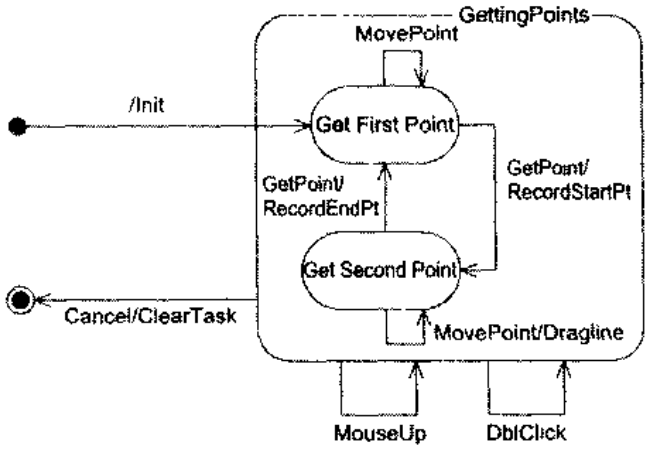


图 30.11　MeasureTask 的有限状态机

当使用者最终在 TaskWindow 中点击时，会发生一个 GetPoint 事件。这致使状态机迁移到 GetSecondPoint 状态并执行 RecordStartPt 动作。该动作会绘制第一个十字交叉线，并且也会把鼠标的点击点记为起始点。

在 GetPointState 状态中，MovePoint 事件会引起 Dragline 动作的执行。该动作把显示模式设置为 XOR[①]，并绘制一条从已经记下的起始点到鼠标当前位置的线。同时，它也计算这两点之间的距离并把计算结果显示在一个消息窗口中。

只要鼠标在 TaskWindow 中移动，MovePoint 事件就会频繁出现，因此只要鼠标在运动，移动的线以及消息窗口中显示的长度就会不断地更新。

当使用者第二次点击时，状态机就迁移回 GetFirstPoint 状态，并调用 RecordEndPt 动作。这个动作会取消掉 XOR 模式、擦除第一个点和当前鼠标位置间的线、在点击点绘制一个十字交叉线并把起始点和点击点之间的距离显示在消息窗口中。

只要使用者愿意，就可以重复该事件序列。仅当该任务被 CommandWindow 取消时才终止，取消的原因可能是为了响应使用者对命令按钮的点击。

图 30.12 中展示了一个稍微复杂一点的任务——绘制一个“两点”方框。

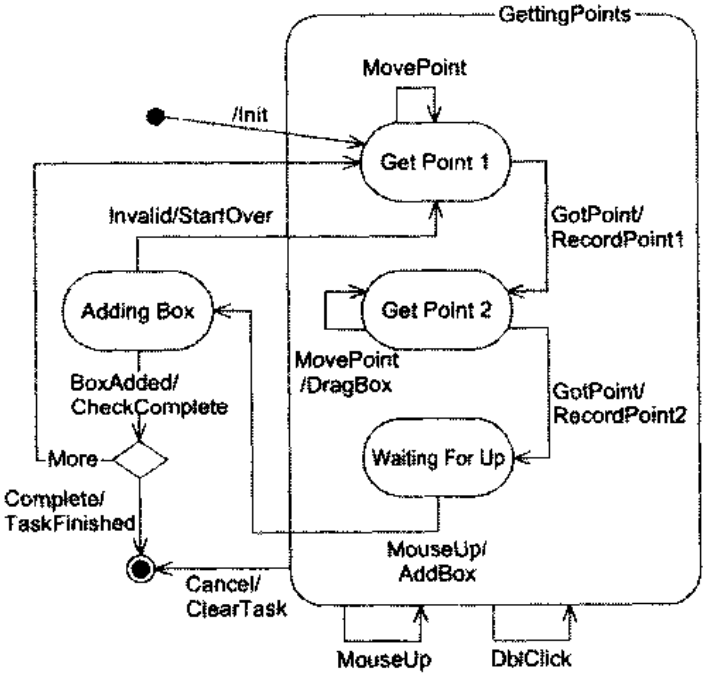


图 30.12　两点方框

① XOR 模式是 GUI 可以设置的一种模式。它极大地简化了在屏幕中已有形状之上拖曳可拉伸的线或者形状的问题。如果你不理解这一点，请不要为此担心。

两点方框是一个矩形，可以通过在屏幕上点击两次把它绘制出来。第一次点击锚定方框的一个角。接着，就会出现一个随鼠标移动而拉伸的方框。当使用者第二次点击时，方框就被固定下来。

像前面一样，在调用 Init 后，任务就开始于 GetPoint1 状态。在该状态中，鼠标的移动被忽略。当点击鼠标时，状态机就迁移到 GetPoint2 状态并调用 RecordPoint1 动作。该动作把点击点记录下来作为起始点。

在 GetPoint2 状态中，移动鼠标会引起 DragBox 动作的调用。该函数把模式设置为 XOR 并从起始点到鼠标的当前位置绘制可拉伸的方框。

当鼠标第 2 次按下鼠标时，状态机就迁移到 WaitingForUp 状态并调用 RecordPoint2。该函数只是记录下方框的第 2 个点。它没有取消 XOR 模式、没有擦除可拉伸的方框、也没有绘制实际的方框，因为我们还不确信方框是否有效。

此时，鼠标仍被按下，并且使用者正打算把手指从鼠标按键上移开。我们要等待它的发生，否则，鼠标键放开事件就会被其他的任务得到，这样会引起混乱。在等待的过程中，我们忽略任何鼠标的移动，并把方框锚定在最后的点击点。

一只鼠标键被放开，状态机就迁移到 AddingBox 状态并调用 AddBox 函数。该函数检查方框是否有效。有许多原因会导致方框无效。可能是退化方框（也就是，第一个点和第 2 个点相同），或者可能和图形中其他一些部分冲突。每个绘图题程序都有权拒绝某些使用者试图绘制的东西。

如果发现方框无效，就产生一个 Invalid 事件，并且状态机迁移回 GetPoint1 状态并调用 StartOver 函数。然而，如果方框有效，那么就产生 BoxAdded 事件。这会引起 CheckComplete 函数的调用。这是另外一个和特定于绘图题的函数。它确定是否应该允许使用者继续绘制另外的方框，以及是否应该结束任务。

事实上，框架中有许多这样的任务。每个任务都用 Task 类的一个派生类来表示（参见图 30.13）。每个任务内部都有一个有限状态机，并且它们相当复杂，无法在此进行展示。每个状态机同样都是由 SMC 产生的。

图 30.13 中展示了 Taskmaster 的构架。该构架把 CommandWindow 和 TaskWindoe 连接起来，并且创建和管理着使用者选择的任务。

图中展示了两个任务，图 30.11 和图 30.12 中描绘了它们的状态机。请注意其中 STATE 模式的运用以及每个任务中被生成的那些类。所有这些类，从上一直到 MeasureTaskImplementation 和 TwoPointBoxImplementation 都是框架的组成部分。事实上，开发者必须编写的类只有 VignetteTaskWindow 以 Task 类的特定派生类。

MeasureTaskImplementation 类和 TwoPointBoxImplementation 类是框架中包含的许多不同任务的代表。但是请注意，这些类是抽象的。其中有几个函数没有实现，比如：AddBox 以及 CheckComplete。在需要时，每个绘图题程序必须根据自己的情况实现这些函数。

因此，框架中包含的任务控制着所有绘图题程序中的大部分交互。每当开发者要绘制一个方框，或者一个和方框有关的对象时，这个开发者可以从 TwoPoiontBoxImplementation 派生一个新的任务类。或者每当他要只用一次点击就能把某些对象放置在屏幕上时，他可以覆写 SinglePointPlacementTask。或者如果他要基于折线绘制一些东西，他可以覆写 PolylineTask。并且，这些任务管理着交互、执行着任何需要的拖曳，并为开发工程师提供了一些吊钩，开发工程师可以使用这些吊钩完成所需的对象验证和创建工作。

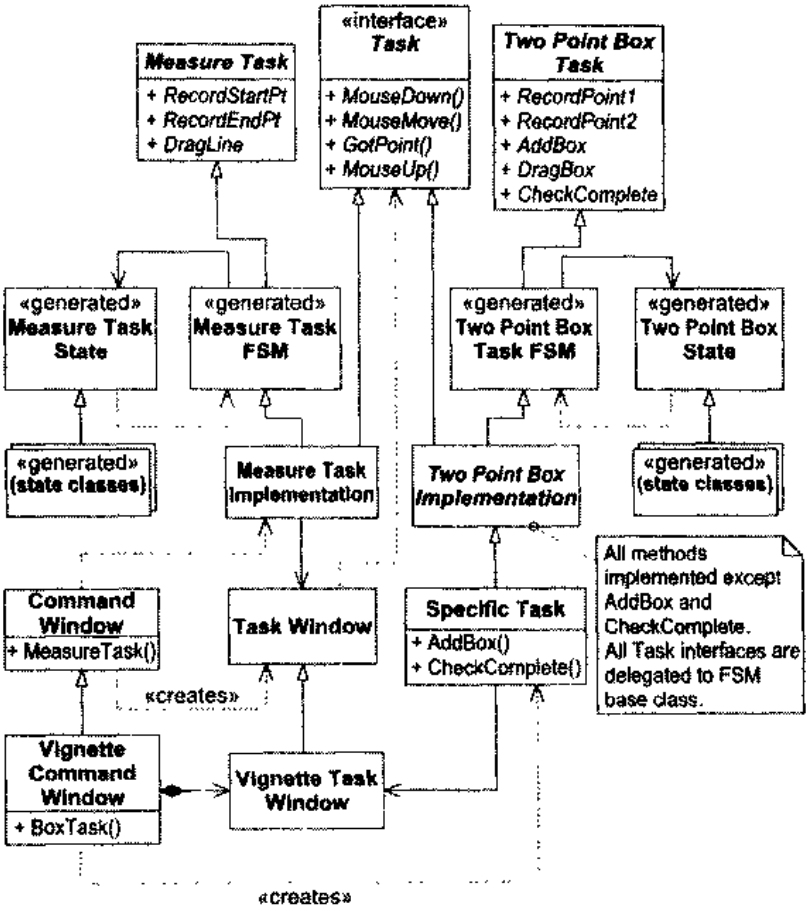


图 30.13 Taskmaster 构架

30.6 结　论

在本章中，我们当然可以谈论的更多一些。我们可以谈论框架中处理计算几何的部分，或者谈论有关答案文件存取的部分。我们可以谈论参数文件的结构，正是这些参数文件才使得每个绘图题应用程序可以驱动自身的许多不同变体。糟糕的是，空间和时间都不允许我们这样做。

不过，我们认为本章中已经包含了框架中最有教育意义的部分。我们在这个框架中使用的策略可以被其他人用在他们自己的可重用框架开发中。

参考文献

1. Booch, Grady. *Object-Oriented Design with Applications.* Redwood City, CA: Benjamin Cummings, 1991.
2. Booch, Grady.*Object Solutions.* Menlo Park, CA: Addison-Wesley, 1996.
3. Gamma, et al. *Design Patterns.* Reading, MA: Addison-Wesley, 1995

附　录

附录 A　UML 表示法 I：CGI 示例

在软件的分析和设计的过程中，迫切需要某种表示法（notation）。为了创建这样的表示法，人们进行了许多的尝试，流程图、数据流图、实体关系图等等都是这种尝试的结果。

面向对象编程的出现引起了表示法种类的一次激增，用来表示面向对象分析和设计的具有竞争关系的表示法可谓数量众多。

这些表示法中，最流行的一些如下：

- Booch 94[①]
- OMT（对象建模技术）——由 Rumbaugh 等定义[②]
- RDD（职责驱动设计）——由 Wrifs-Brock 等定义[③]
- Coad/Yourdon——由 Peter Coad 和 Ed Yourdon 定义[④]

在这些表示法中，Booch 94 和 OMT 是到目前为止最重要的。通常认为，作为一种设计表示法，Booch 94 更强，而作为一种分析表示法，OMT 更强。

这种两分的方法很有趣。在 20 世纪 80 年代晚期和 90 年代早期，人们认为分析和设计能够用相同的表示法表示是面向对象的一个优势。也许，这是对结构化分析和结构化设计被明显地分开做出的反应。众所周知，跨越结构化分析和结构化设计之间的鸿沟是困难的。

当面向对象表示法最初出现时，在分析和设计中可以使用同样的表示法。然而，在几十年发展中，分析师和设计者开始转向他们各自喜欢的表示法。分析师往往更喜欢 OMT，而设计者往往更喜欢 Booch 94。于是，看来一种表示法似乎不足以满足需要。适合于分析的表示法不适合于设计，反之亦然。

UML 是一种单一的表示法，但是它的应用范围很广。表示法中的一些部分可以用于分析；另一些部分可用于设计。因此，分析师和设计者都可以使用 UML。

本章中，我们将从这两者的视角介绍 UML 表示法。首先我们描述一个分析，接着我们会继续描述一个设计。该描述会以小型的案例研究的形式进行。

请注意，这种先分析，然后设计的顺序是人为的，而且也不打算作为一个推荐方法。事实上，本书中所有其他的案例研究都没有在这两者之间作出区分。在此，我用这种方法进行介绍，完全是

---

① [BOOCH94]。
② [RUMBAUGH91]。
③ [WIRFS901]。
④ [COAD91A]。

为了阐明如何在不同的抽象层次中使用 UML。在实际的项目中，所有的抽象层次是同时产生的——不是按顺序的。

A.1　课程登记系统：问题描述

假设我们为一个提供面向对象分析和设计的专业培训课程的公司工作。这个公司需要一个系统来管理要教授的课程以及登记的学生。请参见“课程登记系统”补充内容。

课程登记系统

用户应当能够查看所提供的可选课程清单，并且能够选择他们想登记学习的课程。一旦进行选择，应当弹出一个窗体让用户输入如下信息：

- 姓名
- 电话号码
- 传真号码
- 电子邮件地址

还应该为用户提供一种方法让他们选择希望的课程支付方式。这些方式可以是下面的一种：

- 支票
- 定课单（purchase order）
- 信用卡

如果用户希望用支票支付，那么窗体应该提示他们输入支票号码。

如果用户希望用信用卡支付，那么窗体应该提示他们输入信用卡号码、有效期限以及卡上显示的名字。

如果用户希望用定课单支付，那么窗体应该提示用户输入定课单号码（PO#）、公司的名字以及应付帐款部门中某个人的名字和他的电话号码。

一旦填好了所有这些信息，用户会点击“提交”按钮。另一个屏幕会弹出来，上面汇总了用户输入的所有信息。它会指示用户把屏幕打印出来，在打印拷贝上签名，并把它传真到登记中心。

它也应该把登记汇总通过电子邮件发送给我们的登记处工作人员和用户。

系统知道每个班级的最多学生数目，一旦达到了这个限制，系统会自动地为这个班级标记上“已满”。

通过调出一个特殊的窗体并选择一门课程，登记处工作人员就可以通过电子邮件向所有登记该课程的学生发送消息。该窗体允许工作人员键入一条消息，然后点击一个按钮，就可以把消息发送给当前所有登记学习所选择课程的学生。

登记处工作人员也可以调出一个窗体，上面显示了对于已经教授过的班级，所有学生的状态。这个状态表明学生是否出席以及是否已经收到学生交付的费用。这个窗体可以以课程为基础被调出。或者登记处工作人员也可以要求看一下余额没有结清的所有学生的名单。

识别参与者和用例

需求分析的一个任务就是识别参与者（actor）和用例。值得注意的是，在一个实际的系统中，这些未必是合适的首要任务。但是出于本章的目的，我就选择了从它们入手。事实上，你开始时做什么要比从哪里开始更重要。

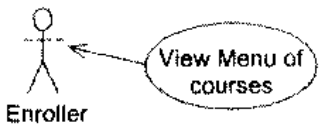
A.1.1 参与者

参与者是那些和系统进行交互，但是又位于系统之外的实体。系统的用户通常担任这些角色。然而，有时，它们可以是其他一些系统。在本例中，所有的参与者都指得是人。

- **登记者**：这个参与者把学生登记到一门课程中。它和系统交互以选择合适的课程，并输入关于学生的信息以及支付方式。
- **登记处工作人员**：这个参与者收到每个登记的电子邮件通知。它还给学生发送电子邮件通知，并接收关于登记和支付的报告。
- **学生**：这个参与者收到登记的电子邮件确认以及来自登记处工作人员的电子邮件通知。学生参加它所登记的课程。

A.1.2 用例

确定了参与者后，我们来详细说明一下这些参与者和系统之间的交互。这些详细描述被称为“用例”。用例从参与者的视角描述了参与者和系统之间的交互。其中不涉及任何系统内部的工作方式，也没有用户界面的任何细节描述。

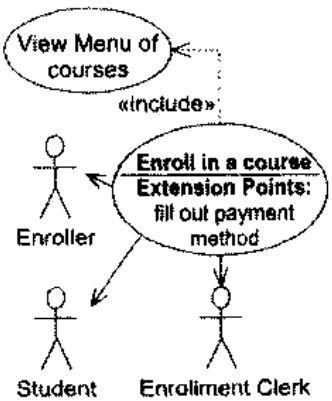


用例#1：查看课程清单

登记者请求一个课程目录中当前可提供的课程清单。系统显示这个课程清单。清单上包含有课程的名字、时间、地点以及费用。清单上还显示有课程所允许的学生数目以及这个课程当前是否“已满”。

用例表示法

上面的图展示了一个用例图中的一个参与者和一个用例。参与者用人形表示，用例用椭圆形表示。这两者被一个显示了数据流方向的关联绑定在一起。



用例#2：登记一门课程

登记者首先查看课程清单（用例#1）。登记者从清单中选择一门要登记的课程。系统提示登记者输入学生的名字、电话号码、传真号码，电子邮件地址。系统还提示登记者选择希望的支付方式。

扩展点（extension point）：填写支付方式

登记者提交登记表单。系统向学生和登记处工作人员发送确认登记的电子邮件。系统向登记者展示登记确认并且要求登记者打印确认信息、在上面签名，并把它传真到一个指定的号码。

扩展以及使用用例

用例＃2 有一个扩展点。这意味着其他用例将会扩展这个用例。扩展（extending）用例会在下面进行介绍，它们是： #2.1、#2.1 和#2.3。对它们的描述被插入到前面用例的扩展点处。它们描述了和所选择的支付方式有关的可能需要被输入的数据。

用例＃2 和用例＃1 之间还有一个《include》关系：“查看课程清单”。这意味着用例＃1 的描述被插入到用例＃2 的适当位置。

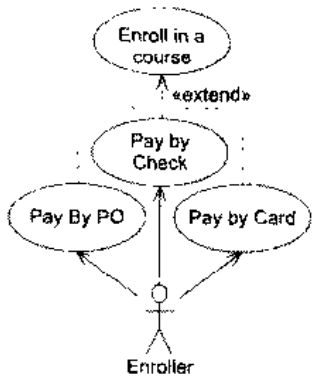
请注意扩展和包含之间的区别。当一个用例包含另一个时，包含用例就引用被包含的用例。但是，当一个用例扩展另一个时，它们之间互不引用。相反，会根据上下文选择一个扩展用例并把它插入到被扩展用例的合适位置。

当我们想让用例结构更加高效时，就把重复的操作分解成小一些的用例，这些用例可以在许多其他用例间共享，此时我们就使用《include》关系。目的就是为了管理变更，消除冗余。把许多用例的公共部分移到一个单独的被包含用例中，这样当公共部分的需求发生变化时，就只有这个单独的被包含用例需要改变。

当我们知道一个用例中有许多代替物或者选择时，我们就使用《extend》关系。我们把用例中的不变部分和可变部分分开。不变的部分就成为被扩展的用例，而可变的部分就变成扩展用例。这样做的目的同样是为了管理变更。在当前的例子中，如果要增加新的支付选择，就必须要创建新的扩展用例，但是任何已有的用例都无需更改。

表示《include》关系的表示法

用例＃2 的图中显示出：“登记一门课程”用例和“查看课程清单”用例被一条以开放箭头结尾的虚线连接起来。箭头指向被包含的用例并且具有构造型《include》。



用例＃2.1：使用定课单支付

登记者被提示输入 PO#、公司的名字以及应付帐款部门中某个人的名字和电话号码。

用例＃2.2：使用支票支付

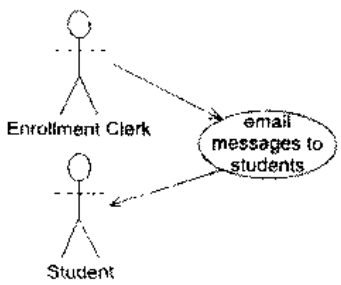
登记者被提示输入支票号码。

用例＃2.3：使用信用卡支付

登记者被提示输入信用卡号码、有效期限以及卡上显示的名字。

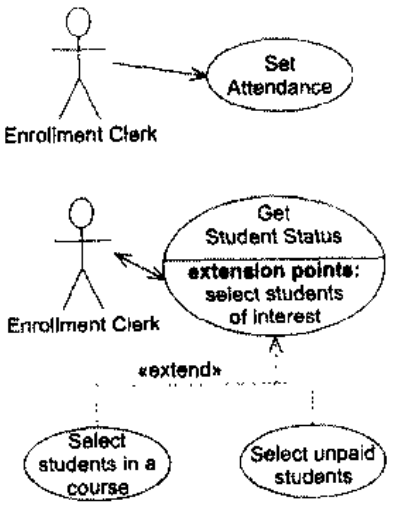
表示用例扩展的表示法

上面显示的扩展用例通过《extend》关系和被扩展用例连接起来。这个关系被绘制成一条连接这两个用例的线，同样也是带有开放箭头的虚线。箭头指向被扩展的用例并且具有构造型《extend》。



用例＃3：通过电子邮件向学生发送消息

登记处工作人员选择一门课程并且输入消息文本。系统把这个消息发送到当前登记这门课程的所有学生的电子邮件地址。



用例＃4：设置出席状态

登记处工作人员选择一个班级及当前登记在这个班级中的一个学生。系统显示出该学生的情况，并显示他是否出席以及是否收到了该学生支付的费用。接着，登记处工作人员可能会修改出席和支付状态。

用例＃5：获取学生状态

登记处工作人员选择感兴趣的学生。

扩展点："选择感兴趣的学生"。

系统在一张报表中显示所选择的学生的出席和支付状态。

扩展

有两个扩展了获取学生状态的用例

用例＃5.1：选择参加一门课程的所有学生

系统显示一张所有课程的清单。登记处工作人员选择一门课程。系统选择出参加这门课程的所有学生。

用例＃5.2：选择未支付费用的学生

登记处工作人员指示系统应该选择出所有未付费的学生。系统把那些被标记为出席，而支付状态却显示出没有收到费用的所用学生选择出来。

再谈用例

我们在这里创建的用例，描述了用户所期望的系统行为。请注意，这些用例没有谈到用户界面的细节。它们没有提到图标、菜单项、按钮或者滚动列表。事实上，即使最初的规格说明对于用户界面的描述也要比用例多。这样做是有意的。我们希望用例是轻量级的并且易于维护。在编写时，用例应该对于非常人的一组可能实现来说都是有效的。

系统边界图

完整的用例集可以通过图 A.1 中的系统边界（system-boundary）图来汇总显示。该图显示了被一个表示系统边界的矩形包围起来的系统中的所有用例。参与者被放置在系统外部，并且被带有数据流向的关联和用例连接起来。

我们用这些图来做什么

用例图，包括系统边界图，都不是软件结构图。它们没有给我们提供有关要创建的系统的软件元素划分的任何信息。这些图是用来进行人和人之间的交流的，主要是用于分析师和涉众（stakeholder）之间的交流。它们有助于按照不同类型的系统用户来组织系统的功能。

此外，当为各种不同的用户描述系统时，这些图会相当有用。每种不同的用户主要关心的都是他自己的用例。参与者和用例之间的连接可以使每种不同的用户关注于他或者她将要使用的用例。在非常大型的系统中，你可能希望按照参与者的类型来组织系统边界图，这样所有不同类型的用户

就都可以看到他们所关心的用例子集。

最后，请记住 Martin 文档第一定律：直到迫切需要并且意义重大时，才来编制文档。

这些图可能是有用的，但是它们通常不是必需的。你不应该认为它们是必需的或者必要的。如果你需要它们，那么就绘制它们。否则，就一直等待，直到你有这个需要为止。

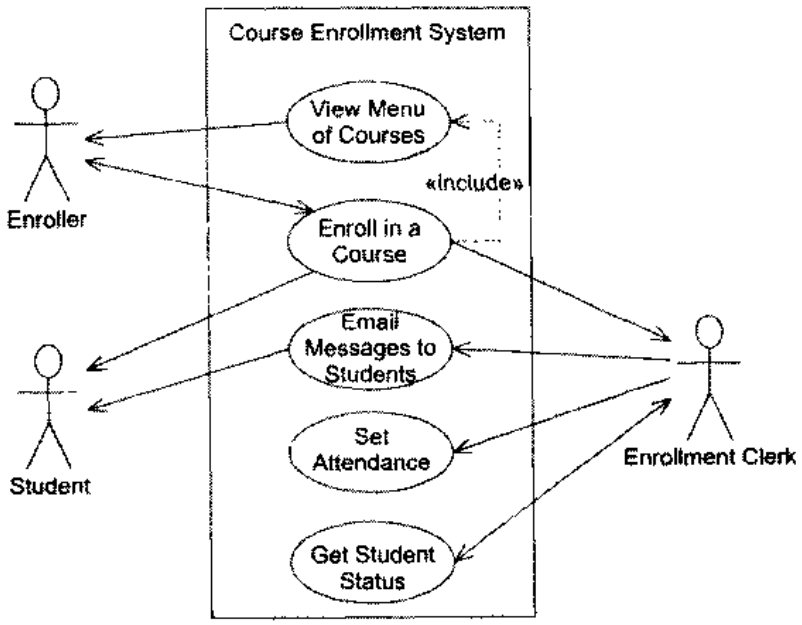


图 A.1　系统边界图

A.1.3　领域模型

领域（domain）模型是一组图，这些图有助于定义出现在用例中的术语。这些图显示了问题中的关键对象以及它们之间的关系。人们往往认为这个模型同样也是要构建的软件的模型，这已经导致了许多的痛苦和损害。对于分析师和设计者来说，认识到领域模型是一种描述工具，用来帮助人们记录他们的决策以及相互之间的交流是非常重要的。领域模型中的对象未必对应于软件的面向对象设计，这样的对应也没有多大的价值。[①]

在 Booch94 和 OMT 中，领域模型图与表示软件结构和设计的图是没有区别的。更糟糕的情形是，领域模型图被作为高层的设计文档，因此被用来建立软件本身的高层结构。

为了避免这类错误，我们可以利用 UML 特性（feature）的优势，在领域模型中使用一种特殊的称为《type》的实体。《type》代表了对象可以担当的角色。《type》可以具有操作和属性，也可以具有和其他《type》实体的关联。但是，《type》不代表设计意义上的类或者对象，不代表软件元素，

① [JACOBSON]，第 133 页，“我们认为最好的（最稳定的）系统不是只使用和真实世界中的实体对应的对象构建起来的……”并且在 p.167 写道，“在[其他]一些方法中，这个[领域]模型同样也成为了实际实现的基础；也就是说，在实现期间，对象直接被映射成类。然而，在 OOSE 中却并非如此，[……]。我们使用这种方法的经验表明了一些不同的内容。我们不是使用问题领域模型来充当设计和实现的基础，而是开发了一个在未来改变面前更加健壮、更加可维护的分析模型。”[BOOCH96]，第 108 页，“……在一个不成熟的项目中，往往会认为可以基于从分析中得到领域模型进行编码[……]，从而省略了任何进一步的设计。健康的项目知道仍然还有许多工作要做，包括诸如：并发、序列化、安全性、分布，等问题，并且最终的设计模型在许多细微的方面看起来都有很大的不同。”

并且也不直接映射到代码。它表示了一个用于问题描述的概念实体。

课程目录

我们首先要考虑的领域抽象是课程目录。这个抽象代表着所提供的所有课程的清单。我们在课程目录的领域模型（请参见图 A.2）中展示了这一点，其中描绘了代表领域中的抽象的两个实体：CourseCatalog 实体和 Course 实体。CourseCatalog 实体提供了许多 Course 实体。

领域模型表示法

图 A.2 中使用的表示法把两个领域抽象体描绘成具有《type》构造型的 UML 类（请参见补充内容，“UML 类表示法和语义概述”）。这表明这些类是问题领域中的概念元素，和软件类没有直接的关系。请注意， CourseCatalog 实体有 2 个操作：AddCourse 和 RemoveCourse。在《type》中，操作对应于职责。所以，CourseCatalog 具有可以增加和去除课程的职责。同样，这些也只是概念，不是真实类中成员函数的规格说明。我们使用它们来和用户交流，而不是为了明确说明软件结构。

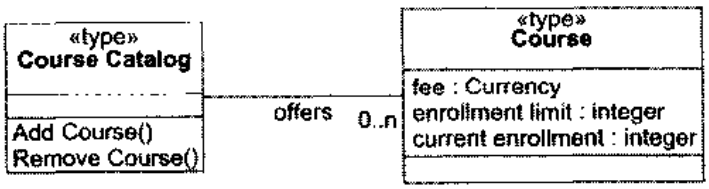


图 A.2　课程目录实体的领域模型

出于同样原因，Course 实体中显示的属性也只是概念。它们表明 Course 实体应当负责记住它的学费、登记限额以及当前的登记人数。

概念与实现以及云形图标的使用

我多次强调了概念层次（也就是，《type》）上的类与设计或者实现层次上类之间的区别。我觉得这样做是恰当的，因为把概念图误认为是软件的结构和构架的规格说明是危险的。请记住，概念图是用来帮助和涉众进行交流的，不具有任何软件结构技术方面的内容。

在对这些种类的图进行区分方面，构造型的作用可能被忽视了。领域模型中的 UML 类看上去很像设计和实现中的 UML 类。所以，为了进一步突出这些种类的图之间的不同点，我们从现在起将使用云形图标来表示《type》类。于是，图 A.2 中课程目录实体的领域模型就变成如图 A.3 中的那样。

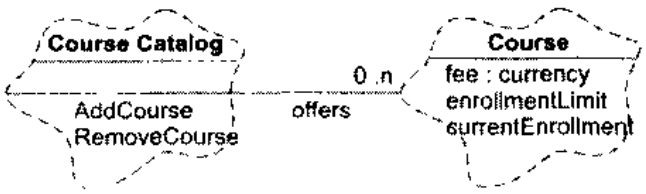
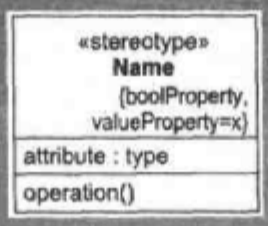


图 A.3　用云形图标表示的课程目录领域模型

UML 类表示法和语义概述

在 UML 中，类被绘制成具有 3 个被分隔开的区域的矩形。第一个分隔区域指明了类的名字。第二个指明了它的属性，第三个指明了它的操作。

在名字分隔区内部，名字可以用一个构造型以及一些特性（property）修饰。构造型出现在名字的上面，用书名号（《》）括了起来。特性出现在名字的右下方，用大括号括了起来（见下图）。



构造型指得是代表 UML 类的"种类"的名字。在 UML 中，类只是一个具有属性和操作的有名字的实体。缺省的构造型是《implementation class》，在这种情况下，UML 类直接对应于语言（如：C++、Java、Smalltalk 或者 Eiffel）中类的软件概念，属性对应于成员变量，而操作对应于成员函数。

然而，如果构造型是《type》，那么 UML 类就完全不对应于软件实体，而是对应于存在于问题领域中的一个概念实体。属性代表着逻辑上属于该概念实体的信息，而操作则代表着概念实体的职责。

在本章的后面，我们会讨论一些其他的预定义的构造型。你可以自由创建你自己的构造型。然而，构造型不仅仅是一种注释，它还规定了 UML 类中的所有元素应该按照什么样的方式去解释。所以如果你创造了一种新的构造型，就把它定义好。

特性是主要的结构化注释。特性被表示为大括号之间用逗号分隔的列表。每个特性都是一个用等号（=）分隔的"名字=值"对。如果等号被省略了，那么就认为这个特性是布尔型的并被赋值为"true"。否则值的类型就是字符串。

在本章的后面，我们会讨论一些其他的预定义特性。不过，你在任何时候都可以自由增加自定义的特性。例如，你可能会像这样创建特性：

```
{author=Robert C. Martin, date=12/6/97, SPR=4033}
```

完成领域模型

迄今为止，领域模型中展示了包含所提供的全部课程的课程目录。不过，在此，我们有一个问题。课程的用意是什么呢？相同的课程可以在许多不同的时间和地点开设，并且可以被许多不同的讲师教授。显然，我们需要两个不同的实体。我们称第一个为 Course。它表示课程本身，但是不是时间、地点和教授者。我们称第二个为 Session。它表示一门特定课程的时间、地点以及教授者（请参见图 A.4）。

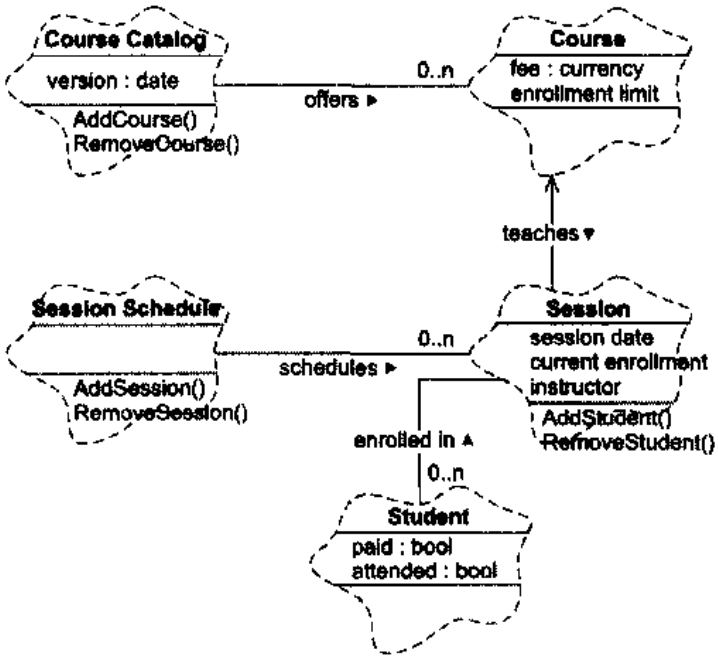


图 A.4　Course 和 Session

表示法

把实体连接起来的线被称为关联。图 A.4 中的所有关联都具有名字，尽管这不是一个规则。请注意，名字都是动词或者动词短语。名字旁边的黑色小三角指向由两个实体以及它们之间的关联组成的句子的谓语。所以，“Course Catalog 提供了很多 Course”，“Session Schedule 安排许多 Session 的时间表”以及“许多学生可以登记到一个 Session 中”。

在前面的句子中，只要相应的关系是用 “0..*” 图标表示的，我都使用了单词“许多”。这个图标是几种不同的多重性（multiplicity）图标中的一种，它们可以被放置在关联的末端。它们指出了参与关联的实体的数目。缺省的重数为“1”（请参见补充内容，“多重性”）。

多重性

有几个多重性图标可以被用来修饰关联，包括以下这些：

- 0..*　0 个到多个
- *　0 个到多个
- 1..*　1 个到多个
- 0..1　0 个或者 1 个
- 6　正好 6 个
- 3..6　在 3 个和 6 个之间
- 3,5　3 个或者 5 个

任何非负整数都可以用在点的两边或者用逗号分隔开来。

关联都被认为是双向的，除非其上带有一个箭头。双向关联允许两个实体互相知晓。例如，显然，CourseCataiog 实体应该知道它的 Course 实体，并且每个 Course 实体都能知道它被列入的 CourseCatalog 实体似乎也是合理的。对于 SessionSchedule 和 Session 来说，同样如此。

箭头的出现把知识限制在箭头所指的方向。所以，Session 知道 Course，但是 Courses 却对 Session 一无所知。

用例迭代

从这个图中，我们立即可以知道两件事情。首先，用例在许多地方使用了错误的语言。在它们谈论课程目录和课程的地方，它们应该谈论上课时间表和课时（session）。其次，有许多用例被遗漏了。CourseCatalog 和 SessionSchedule 需要被维护。Course 需要增加到 CourseCatalog 中以及从 CourseCatalog 中去除，并且 Sessions 需要增加到 SessionSchedule 中以及从 SessionSchedule 中去除。

因此，通过创建一个领域模型，我们更好地理解了手边的问题。这种更好的理解有助于我们对用例进行改善和补充。这两者之间的这种迭代是自然的，也是必要的。

如果我们继续这个案例研究直到得到最终结果，我们就会展示上面所暗含的那些变化。但是为了高效地介绍表示法，我们将跳过用例的迭代。

A.1.4　构架

现在，我们开始来考虑软件设计。构架指的是构成应用程序的骨架（skeleton）的软件结构。构架中的类以及关系和代码之间有非常紧密的映射关系。

决定软件平台

然而，在我们开始前，我们必须要了解应用程序运行所在的软件平台。我们有许多选择。

（1）基于 Web 的 CGI 应用程序。通过 Web 浏览器可以访问登记以及其他一些窗体（form）。数据驻留在 Web 服务器上，Web 浏览器会调用 CGI 脚本去访问和操作这些数据。

（2）数据库应用程序。我们可以购买一个关系数据库并且使用窗体包和 4GL 去编写应用程序。

（3）VisualXXX。我们可以购买一个可视化（visual）编程语言。人机界面可以使用可视化构造工具创建。这些工具会调用存储、获取以及操作数据所需的软件函数。

当然，还有其他选择。我们可以用 C 语言来编写整个应用程序，而无需除了编译器之外的任何库或者工具的支持。不过，这样做很愚蠢。工具就在手边，并且也都好用。我们应该使用它们。

为了我们的例子的缘故，我们将选择基于 Web 的应用程序。这个选择是有意义的，因为这样的话登记者就可以位于世界上的任何地方，并且我们也可以在 Internet 上提供登记服务。

Web 构架

我们需要决定 Web 应用程序的总体构架。将会有多少个 Web 页面，它们会调用什么样的 CGI 程序呢？图 A.5 展示了我们开始时可能的设想。

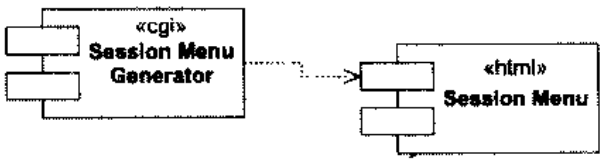


图 A.5　Session Menu 构架

表示法

图 A.5 是一个组件图。图标描绘的是实体（physical）软件组件。我们使用了构造型来指明组件的种类。图中展示出，Session Menu 是作为 HTML Web 页面显示的，该页面是被一个称为 Session Menu Generator 的 CGI 程序产生的。两个组件之间的虚箭头线表示一个依赖关系。依赖关系指明了哪个组件具有对其他组件的知识。在本例中，Session Menu Generator 程序创建了 Session Menu Web 页面，所以具有对它的知识。不过，Ssession Menu Web 页面却对生成器本身一无所知。

定制图标

图 A.5 中有两种不同的组件。为了使它们看上去不同，我们对 UML 进行了扩展，创建了两个新图标——一个用于 CGI 程序，一个用于 HTML 页面。图 A.6 中展示了用例#2：登记一门课程中所涉及的组件。Web 页面被绘制为页形，其中有一个‘W’。

CGI 程序被绘制为圆形，其中有‘CGI’。

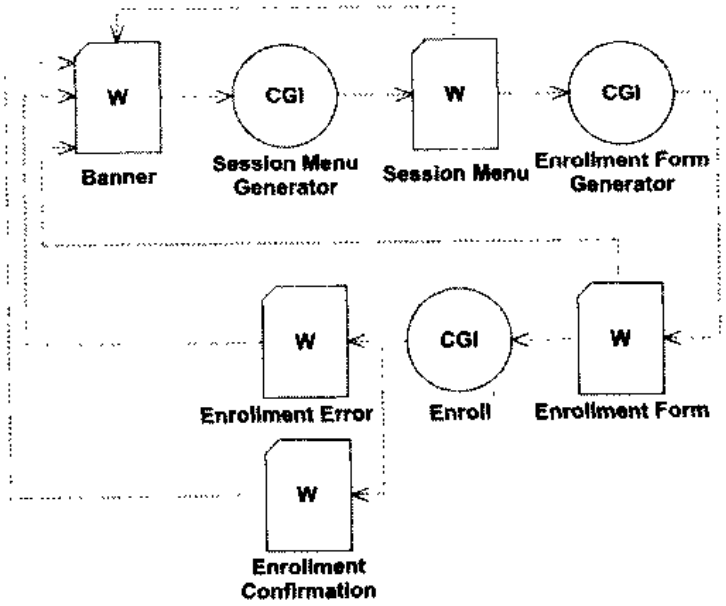


图 A.6　登记组件图

组件流程

图 A.6 中引入了两个新的 Web 页面和一个新的 CGI 程序。我们已经决定应用程序应该以某种标题页面开始。这个页面上可能会有一些使用者可以执行各种各样操作的链接。标题页面调用 Session Menu Generator 来生成 Session Menu 页面。Session Menu 页面上可能会具有一些使得用户可以登记一门课程的链接或者按钮。Session Menu 调用 Enrollment Form Generator CGI 来创建登记所选课程必需的窗体。一旦使用者填写完成，就会调用 Enroll CGI。该程序验证并记录窗体上的信息。如果窗体上的信息无效，它就生成一个 Enrollment Error 页面；否则，它就生成一个确认页面并且通过电子邮件发送一些必要的消息（请参考用例#2）。

提高灵活性

敏锐的读者会发现，这个组件模型中存在有相当严重的不灵活性。CGI 程序生成了大部分的

Web 页面。这意味着，表示 Web 页面的 HTML 文本必须被包含在 CGI 程序中。这样，在更改 Web 页面时，就必须要更改和重新构建 CGI 程序。我们宁愿使用一个好的 HTML 编辑器来创建这些数量众多的被生成的 Web 页面。

因此，CGI 程序应该读入一个它们将要生成的 Web 页面的模板（template）。模板应该被做上特殊的标记，这些标记会被 CGI 程序必须生成的 HTML 替换掉。这意味着 CGI 程序之间共享了一些公共的东西。它们都读取模板 HTML 文件，并且在其中加入自己的内容。[1]

图 A.7 展示了由此得到的组件图。请注意，我增加了一个 WT 图标。它代表了一个 HTML 模板。它是一个具有特殊标记的 HTML 格式的文本文件，CGI 程序把这些标记作为它们生成的 HTML 的插入点。同样请注意 CGI 程序和 HTML 模板之间依赖关系的方向。它们似乎和你想的相反。但是请记住，它们是依赖关系，不是数据流。CGI 程序知道（依赖于）HTML 模板。

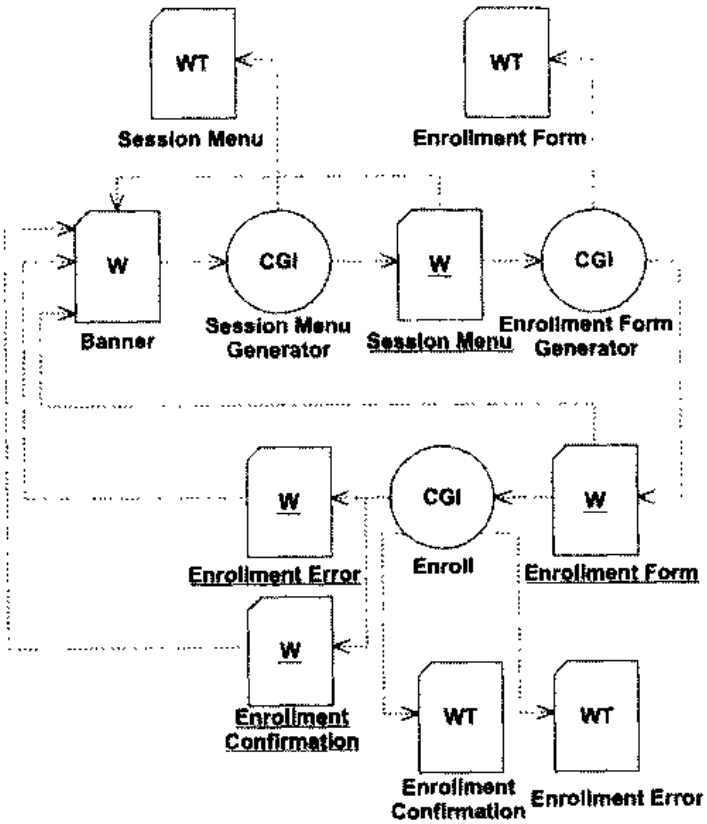


图 A.7　在 Enrollment 组件图中加入了 HTML 模板

规格说明/实例二分法

在图 A.7 中，被生成的 Web 页面的名字加了下划线。这是因为这些页面只存在于运行时。它们是 HTML 模板的实例。在 UML 中，约定给实例加下划线。实例是根据某种规格说明（源文档）生成的软件元素。我们稍后会对此做详细介绍。目前，我们所要知道的只是：名字下面没有下划线的元素表示必须用手工编写并充当规格说明的那些元素，名字下面有下划线的元素是根据规格说明生

[1] 本章的编写时间远在 XSLT 出现之前。现在，我们很可能会这样来解决问题：让 CGI 脚本（或者 servlets）生成 XML，然后调用 XSLT 脚本把它转换为 HTML。不过，用 XSLT 来生成 HTML 仍然无法让我们使用一个 WYSIWYG（所见即所得）的编辑器来设计 Web 页面。有时，我认为本章中提到的模板方案在许多情况中都会更好一些。

成它们的某个过程的产品。

使用 HTML 模板

HTML 模板为这个应用程序的构架提供了极大的灵活性。它们如何工作呢？CGI 程序如何把它们生成的输出物放到 HTML 模板的合适位置呢？

我们也许会考虑把一个特殊的 HTML 标签（tag）放入 HTML 模板文件中。这样的标签会在生成的 HTML 文件中标记出位置，CGI 程序可以在这个位置上插入它的输出物。然而，这些生成的 Web 页面可能具有多个部分（section），每一部分也许都需要它自己的 CGI 可以插入 HTML 的插入点。因此，每个 HTML 模板可能具有多个插入标签，CGI 程序必须能够以某种方式指明哪个输出物对应哪个标签。

标签看上去可以像这样：<insert name>，其中“name”是标识 CGI 插入点的任意字符串。像<insert header>这样的标签使得 CGI 指定名字“header”，然后用生成的输出物完全替代这个标签。

显然，每个标签都是被一串字符替换的，所以每个标签都代表了一种被命名的字符流。可能存在于 CGI 中的这种 C++代码如下所示：

```
HTMLTemplate myPage("mypage.htmp");
myPage.insert("header",
        "<h1> this is a header <h1>\n");
cout << myPage.Generate();
```

这段程序会把从模板 mypage.html 生成的 HTML 发送到 cout，模板中的<insert header>标签被字符串"<h1> this is a header <h1>\n"代替。

图 A.8 中展示了我们所设计的 HTMLTemplate 类。该类中有一个持有模板文件名的属性。它同样也具有在被明确命名的插入点插入替换字符串的方法。HTMLTemplate 的实例会包含一个把插入点的名字和替换字符串联系起来的 map。

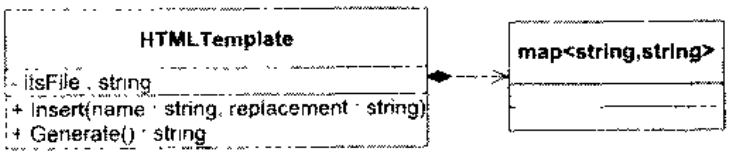


图 A.8 HTMLTemplate 设计

表示法

这是我们第一个真正的类图。它向我们展示了两个具有组合（composition）关系的类。用来表示 HTMLTemplate 类和 map<string,string>类的图标对我们并不陌生。在补充内容，“UML 类表示法和语义概述”中已经作过介绍。在下面的补充内容“属性和操作”中，我们来描述属性和操作的语法。

图 A.8 中连接两个类的关联上的箭头表明，HTMLTemplate 知道 map<string,string>，但 map<string,string>却不知道 HTMLTemplate。关联中靠近 HTMLTemplate 类一端的黑色菱形把该关联确定成一种特殊情形的关联：组合（请参见补充内容，“关联、聚合和组合”）。它表明，HTMLTemplate 要负责 map 类的生存期。

属性和操作

属性和操作可以用下面的封装说明符来修饰：

+　公有的
-　私有的
#　受保护的

属性的类型可以用跟在属性之后，用冒号分隔的标识符来指定（例如，count : int）。

同样的，函数的参数类型也是使用相同的冒号表示法来指定（例如，SetName(name : string)）。

最后，操作的返回类型可以用跟在操作名字和参数列表之后，用冒号分隔的标识符来指定。（例如，Distance(from : Point) :float）。

数据库接口层

每个 CGI 程序都必须能够访问表示课程、班级以及学生等等的数据。我们称之为培训数据库。至今，我们还没有确定数据库的形式。数据可以保存在一个关系数据库或者一组平面文件中。我们不想让应用程序的构架依赖于数据存储的形式。我们希望应用程序的大部分代码在数据库形式发生变化时仍然能够保持不变。因此，我们通过引入一个数据库接口层（DIL）使得数据库对应用程序隐蔽起来。

为了达到这个效果，DIL 必须具有图 A.9 所示的特殊的依赖关系特征。DIL 依赖于应用程序，并且 DIL 依赖于数据库。应用程序和数据库相互之间一无所知。这就使得我们改变数据库时，不必去改变应用程序。同样使得我们在改变应用程序时，也不必去改变数据库。我们可以在不影响应用程序的情况下，完全替换掉数据库格式或者引擎。

图 A.9　数据库接口层的依赖关系特征

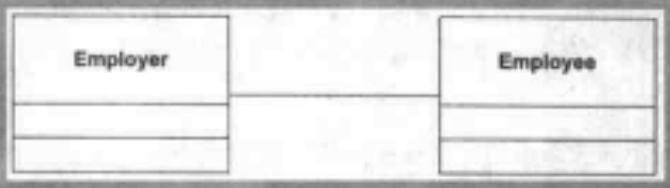
表示法

图 A.9 展示了一种特殊的类图："包图"。图中的图标表示包。它们的形状让人联想到文件夹。和文件夹一样，包是一个容器（请参见补充内容，"包和子系统"）。图 A.9 中的包包含有诸如类、HTML 文件，CGI 主程序文件等等软件组件。连接包的虚箭头线代表一个依赖关系。箭头指向依赖的目标。包之间的依赖关系意味着，一个包必须和它所依赖的包一起使用。

关联、聚合和组合

关联是两个类之间的一种关系，这种关系允许从这两个类创建的实例可以互相发送消息（也就是说，被关联起来的类的对象间会存在有链接（link））。它被表示为一条连接两个类的线。关联最常见的实现方式为：一个类中的实例变量指向或者引用到另外一个类。

关联

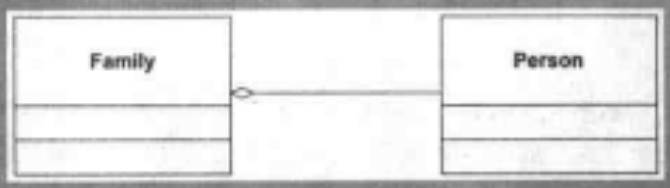


通过向关联中增加箭头可以限制关联的导航性（navigability）。当带有一个箭头时，关联就只能够朝着箭头的方向前进。这意味着箭头指向的类不知道它的关联者。

可导航的关联

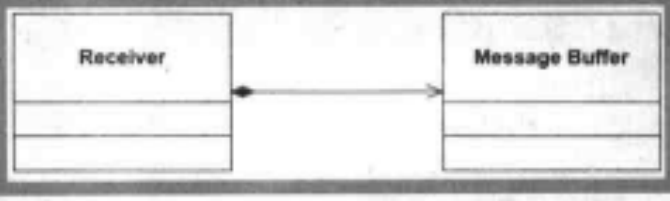
聚合是一种特殊形式的关联。它被表示为聚集（aggregate）类上的一个白色菱形。聚合意味着"整体/部分"关系。和白色菱形相邻的类是"整体"，另外一个类是"部分。"整体/部分"关系完全是隐含的；和关联没有语义上的不同。[①]

聚合



组合是一种特殊形式聚合。它被表示为一个黑色菱形。它意味着"整体"负责它的"部分"的生存期。这个职责并不是指创建或删除的责任。更确切的说，它指得是"整体"必须注意到"部分"以某种方式被删除了。这可以通过直接删除"部分"或者把"部分"传递给另一个负责删除的实体来实现。

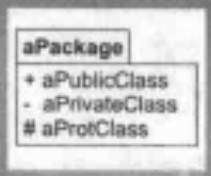
组合



① 有一个例外。对象之间的自反和环形聚合关系是不允许的。也就是说，实例不能参与到一个聚合环中。如果没有这条规则，那么环中的所有实例就都是它们自身的一部分。也就是说，部分可以包含其整体。
请注意，这条规则并没有禁止类参与到聚合环中；它只是限制它们的实例。

包和子系统

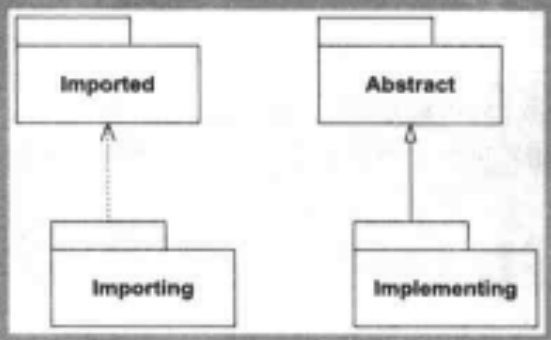
包被绘制为一个大的矩形，其左上角带有一个小一些的矩形标签。包的名字通常放在大矩形中。



包也可以这样绘制：其名字放在“标签”中，而在大矩形中放置内容。内容可以是类、文件或者其他的包。每项内容前面都可以加上（-、+、#）封装图标前缀来指明它们在包中是私有的、公有的还是受保护的。

包之间可以用两种不同关系中的一种连接起来。虚依赖箭头线被称为输入（import）依赖。它是一个具有构造型《import》的依赖关系。当在包上使用依赖关系时，缺省的就是这种构造型。箭头的底部和输入包相连，而箭头则和被输入的包相连。输入依赖意味着被输入的包中的所有公有元素对输入包都是可见的。这意味着输入包的元素可以使用被输入的包中的所有公有元素。

包之间还可以用泛化（Generalization）关系来连接。空心三角形箭头和通用的（general）或者抽象的包相连，而关系的另一端则和实现包相连。抽象包中的所有公有和受保护的元素对实现包都是可见的。



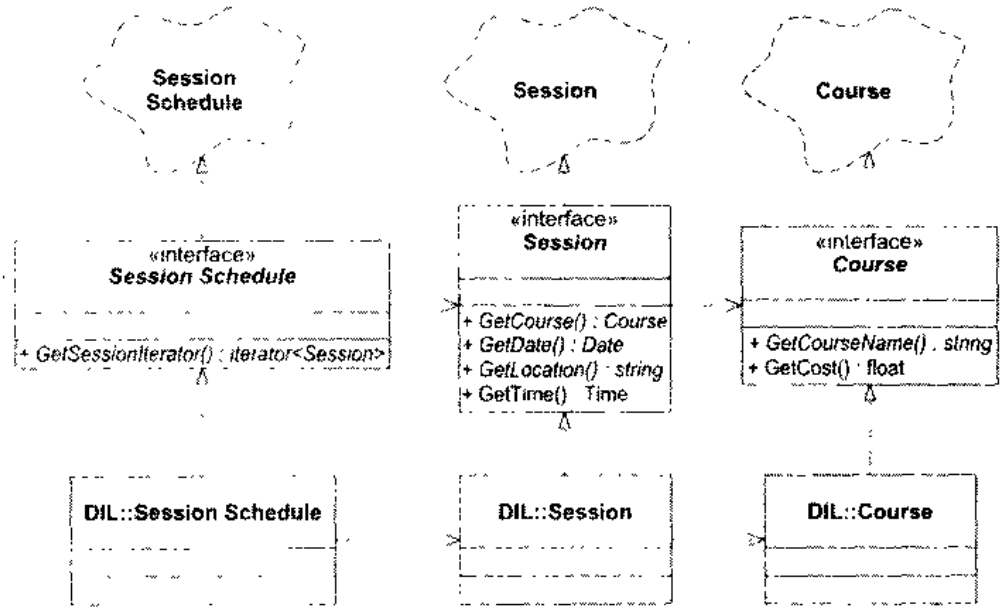
包有一些已定义的构造型。缺省是《package》，表明一个没有任何特殊限制的容器。它可以包含任何用 UML 建模的东西。一般用它来表示一个可发布的实体单元。我们会在配置管理和版本控制系统中跟踪这样一个包。它可以被表示为一个文件系统中的子目录或者一种语言中的模块系统（例如，Java 包或者 JAR 文件）。包代表了一种对系统的划分，这种划分增强了系统的可开发性和可发布性。

包的《 subsystem 》构造型表示了一个逻辑元素，该逻辑元素除了包含有模型元素外，还指明了它们的行为。子系统可以被赋予操作。包内部的协作或者用例必须要支持这些操作。子系统表示了一个系统或者应用程序在行为上的划分。

这两种包相互之间是正交的。增强可开发性和可发布性的划分与基于行为的划分几乎根本没有什么相似之处。前者通常被软件工程师用作配置管理和版本控制的单元。后者通常被分析师使用，目的是为了以一种直观的方式描述系统，并且在特性改变或者增加时进行影响分析。

数据库接口

Training Application包中的类需要一些访问数据库的方法。这可以通过Training Application包中的一组接口完成（请参见图A.10）。这些接口代表了图A.4中的领域模型中的类型。这些接口被DIL包中的类实现。Training Application包中的其他类会使用它们去访问数据库中的数据。请注意，图A.10中依赖关系的方向与图A.9中包之间的输入关系的方向对应。



图A.10 Training Application包中的数据库接口类

表示法

当类的名字被用斜体书写时，就表明这个类是抽象的。[1]因为接口是完全的抽象类，所以就应该以斜体来书写它们的名字。操作也被书写成斜体以表明它们是抽象的。DIL中的类通过实现（realizes）关系被绑定到接口上，这些关系被绘制为带有指向接口的空心三角箭头的虚线。在Java中，它们代表着“implements”关系。在C++中，它们代表着继承。

接口是实体结构。在像C++和Java这样的语言中，它们有着源代码对等物。但是，类型不是实体上的，并且也不代表具有源代码等价物的东西。在图A.10中，我们绘制出了从接口到它们所代表的类型的实现关系。这并不代表一个实体关系，也没有对应的源代码。在本例中，实现关系展示了实体设计结构和领域模型之间的对应关系。这种对应关系很少会像在这里描述的一样清晰。

图A.9中的包之间的输入关系通过使用双冒号展示在图A.10中。DIL::Session类是一个存在于DIL包中名为Session的类并且对（已经被输入到）TrainingApplication包是可见的。因为两个名为Session的类在不同的包中，所以是可以接受的。

Session Menu Generator

请回顾一下图A.7，我们看到，第一个CGI程序是SessionMenuGenerator。这个程序对应于最开始的用例。这个CGI程序的设计看起来是什么样子呢？

---

① 抽象类至少具有一个纯虚（抽象）方法。

显然，它必须构建一个课时安排表的 HTML 表示。因此，我们需要一个把课时安排表样板文件（boiler plate）和实际的课时安排表数据合并起来的 HTMLTemplate。同样，程序将会使用 SessionSchedule 接口去访问数据库中的 Session 和 Course 实例以获得它们的名字、时间、地点以及费用。图 A.11 中展示了一个描绘这个过程的顺序图（sequence diagram）。

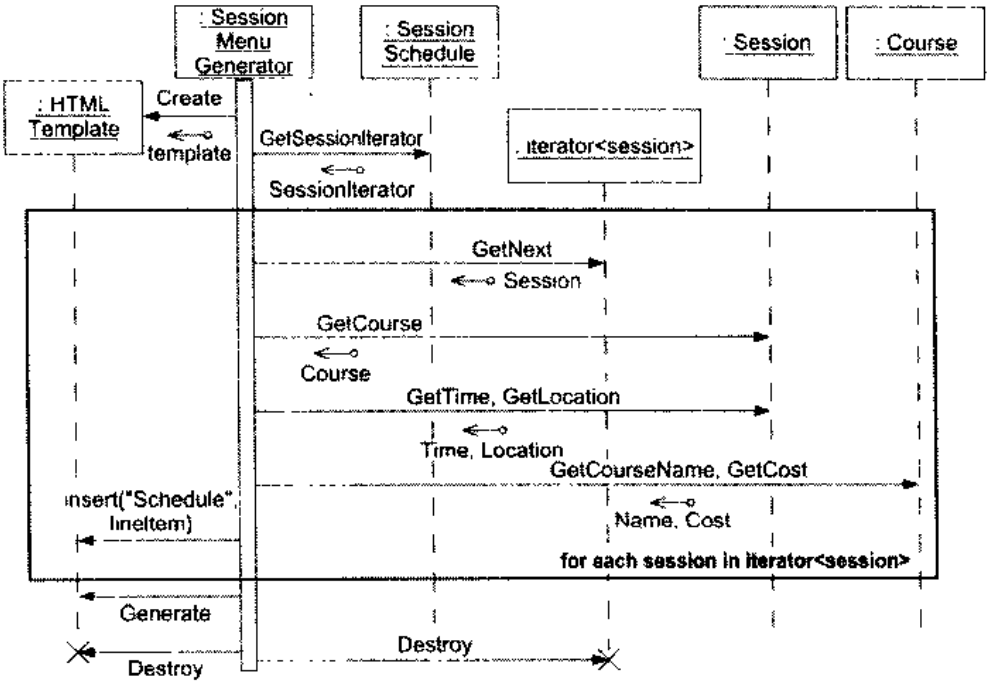


图 A.11　SessionMenuGenerator 的顺序图

SessionMenuGenerator 对象由 main 创建，并且控制着整个应用程序。它以模板文件的名字作为参数创建了一个 HTMLTemplate 对象。接着它从 SessionSchedule 接口中获取一个 iterator<Session>[①]对象。它遍历 iterator<Session>中的每个 Session，请求每个 Session 的 Course。它从 Session 对象中获取课程的时间和地点，并从 Course 对象中获取课程的名字和费用。遍历的最后一步是根据所有这些信息创建一条行式项目（line item）并把它插入到 HTMLTemplate 对象的 Schedule 插入点处。当循环结束时，SessionMenuGenerator 调用 HTMLTemplate 的 Generator 方法，然后销毁它。

表示法

在图 A.11 中，顺序图中矩形内部的名字都被加上了下划线。这表明它们表示的是对象而不是类。对象名字由使用冒号隔开的两个元素组成。冒号前面是对象的简单名字，冒号后面是这个对象实现的类或者接口的名字。在图 A.11 中，简单的对象名字都被省略了，所以所有的名字都以冒号开头。

对象下面悬挂的虚线被称为生存线（lifeline），它们代表对象的生存期。图 A.11 中除了 HTMLTmplate 和 iterator<Session>外的所有对象的生存线都是开始于顶部，结束于底部。按照惯例，这意味着这些对象在场景开始前就已经存在了，并且在场景结束后仍然存在。另一方面，HTMLTemplate 是由 SessionMenuGenerator 显式地创建和销毁的。从终止于 HTMLTemplate 从而创建了它的箭头线，以及底部结束其生存线的“X”，可以明显看出这一点。SessionMenuGenerator 同样也销毁了

① 本章编写于 STL 流行之前。当时，我使用的是我自己的容器库，它具有模板化的迭代器。

Iterator<Session>；然而，并不清楚是什么对象创建了它。这个创建者可能是 SessionSchedule 的一个派生对象。因此，尽管图 A.11 中没有显式地展示 Iterator<session>对象的创建，但是这个对象的生存线起始位置仍然暗示了它大约是在 GetSessionIterator 消息被送到 SessionSchedule 对象时被创建出来的。

连接生存线的箭头线是消息。时间是自顶到底流逝的，因此该图展示了对象之间传递的消息的顺序。靠近箭头线的标签是消息的名字。尾端带有圆形的短箭头线被称为“数据表示（data token）”。它们代表着在消息的上下文中传递的数据元素。如果它们指向消息传递的方向，它们就是消息的参数。如果它们和消息传递的方向相反，它们就是消息的返回值。

SessionMenuGenerator 生存线上长狭形的矩形被称为激活（activation）。激活代表着一个方法或函数执行的持续时间。在本例子中，没有显示启动方法的消息。图 A.11 中其他的生存线不具有激活，因为它们的方法的执行时间都非常短暂并且没有发出消息。

图 A.11 中包围着某些消息的粗体矩形框定义了一个循环。循环的结束条件被标在矩形框的底部。在本例子中，被包围的消息一直重复执行，直到检查完了 iterator<Session>中的所有 Session 对象。

请注意，这些消息箭头线中有两个携带了不只一条消息。这只是一种减少箭头线的数目的简略表达方法。消息按照书写的顺序发送，并且返回值也按照同样的顺序返回。

A.1.5 顺序图中的抽象类和接口

敏锐的读者会注意到，图 A.11 中的某些对象是从接口实例化而来的。例如，SessionSchedule 就是一个数据库接口类。这似乎违反了对象不能从接口或抽象类实例化的原则。

顺序图中对象的类名不必是对象的实际类型的名字。只要对象符合被命名的类的接口就足够了。在像 C++、Java 或者 Eiffel 这样的静态语言中，对象要么属于在顺序图中命名的类；要么属于派生自顺序图中的类或者接口的类。在像 Smalltalk 或者 Objective-C 这样的动态语言中，对象只要符合在顺序图中命名的接口就足够了。[①]

因此，图 A.11 中的 SessionSchedule 对象引用了一个其类实现了或者派生自 SessionSchedule 接口的对象。

Session Menu Gnerator 的静态模型

图 A.11 展示的动态模型隐含着图 A.12 中展示的静态模型。请注意，其中的关系要么是依赖关联；要么是构造型关联。这是因为图中展示的所有类都没包含引用到其他类的实例变量。所有的关系都是短暂的，因为它们没有图 A.11 中 SessionMenuGenerator 生存线上激活矩形的执行期长。

SessionMenuGenerator 和 SessionSchedule 之间的关系应该受到特别的关注。请注意，SessionSchedule 带有{singleton}特性。这表明应用程序中只能存在一个 SessionSchedule 对象并且它在全局范围内都是可访问的（请参见第 16 章中的 SINGLETON 模式）。

SessionMenuGenerator 和 HTMLTemplate 之间的依赖关系具有构造型《creates》。这只是表明 SessionMenuGenerator 实例化了 HTMLTemplate 的实例。

---

① 如果你不理解这一点，请不必担心。在像 Smalltalk 以及 Objective-C 这样的动态语言中，你可以向任何你喜欢的对象发送任何你喜欢的消息。编译器不检查对象是否接受这个消息。如果在运行时消息被发送到一个不认识它的对象上，那么会出现一个运行时错误。因此，两个完全不同并且无关的对象是能够接受相同的消息的。这种对象被称为符合相同的接口。

带有《parameter》构造型的关联表示，对象通过方法参数或者返回值来获取彼此之间的信息。

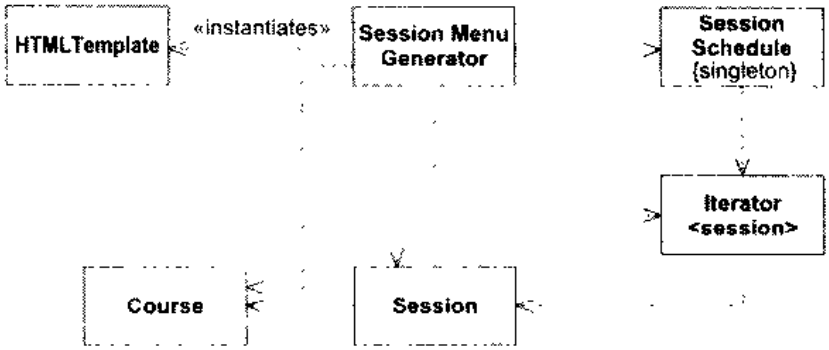


图 A.12　Session Menu Gnerator 应用程序的静态模型

SessionMenuGenerator 对象如何得到控制权呢？

图 A.11 中 SessionMenuGenerator 生存线上的激活矩形没有展示它是如何被启动的。也许是某个像 main 这样的高层实体调用了 SessionMenuGenerator 的一个方法。我们称这个方法为 Run()。这是有意义的，因为我们还需要去编写一些其他的 CGI 程序，并且它们都会需要以某种方式被 main()启动。也许，有一个定义了 Run()方法的称为 CGIProgram 的基类或者接口，SessionMenuGenerator 也许会从它派生。

把用户输入传给 CGI 程序

CGIProgram 类在另外一个问题上也会帮助我们。CGI 程序通常是在用户填写完浏览器屏幕上的表单后，被浏览器调用的。然后，用户输入的数据通过标准输入被传给 CGI 程序的 main()函数。因此，main()可以向 CGIProgram 对象传递一个指向标准输入流的引用，而该对象就可以使它的派生对象方便的使用数据了。

数据是以什么样的形式从浏览器传递到 CGI 程序的呢？它是一组的名字-值对。表单中每个由用户填写的字段都被赋予了一个名字。从概念上来说，我们希望 CGIProgram 的派生对象能够仅仅通过名字就可以请求一个特定字段的值。例如：

```
String course = GetValue("course");
```

因此，main()创建了 CGIProgram 并且把标准输入流传给它的构造函数来为它预先准备好所需要的数据。接着，main()函数调用 CGIProgram 的 Run()方法，使之开始工作。CGIProgram 的派生对象调用 GetValue(string)去访问表单中的数据。

但是这给我们带来了一个两难问题。我们希望 main()函数是通用的，但是它又必须创建 CGIProgram 的适当的派生对象，并且像这样的派生对象有很多。我们如何避免有多个 main()函数呢？

我们可以使用连接时多态来解决这个问题。也就是说，我们在 CGIProgram 类的实现文件中（也就是说，cgiProgram.cc）实现 main()函数。在 main()的实现体中，我们声明一个名为 CreateCGI 的全局函数。但是，我们不实现这个函数。更确切的说，我们会在 CGIProgram 派生类的实现文件（例如，SessionMenuGenerator.cc）中实现这个函数（请参见图 A.13）。

CGI 程序的编写者不再需要去编写一个 main()程序。不过，在 CGIProgram 的每个派生类中，它们必须提供对于全局函数 CreateCGI 的实现。这个函数把派生对象返回给 main()，接着 main()在必要时就可以对它进行操作了。

图 A.13 中展示了我们如何使用带有构造型《function》的组件来表示自由的全局函数。图中同样也演示了使用特性来说明函数实现所在的文件。请注意，CreateCGI 函数用特性{file=sessionMenuGenerator.cc}作为其注解。

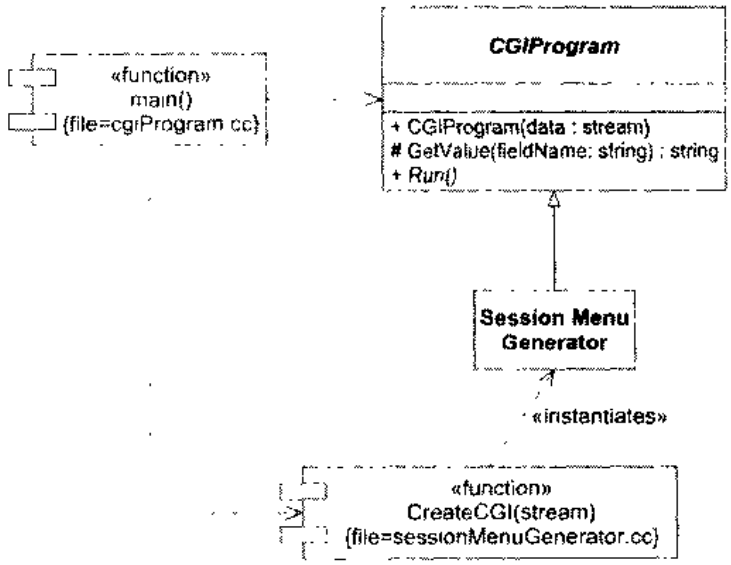


图 A.13 CGI 程序的框架

A.2 小 结

在本附录中，我们在一个简单例子的上下文中遍历了 UML 表示法的大部分内容。我们展示了在软件开发的不同阶段中使用的各种不同的表示法惯用法。我们展示了如何使用用例和类型构成一个应用领域模型来分析问题。我们展示了类、对象以及组件是如何被结合到描述软件构架和构造的静态以及动态图中的。我们展示了所有这些概念的 UML 表示法，并且演示了如何去使用这些表示法。还有最重要的一点，我们展示了所有这些概念和表示法是如何参与到软件设计的思考中的。

关于 UML 和软件设计还有许多要学习的内容。下一章会使用另外一个例子来探索更多的 UML 内容以及一些不同的分析和设计权衡（tradeoff）。

参考文献

1. Booch,Grady.Object Oriented Analysis and Design with Application,2nd ed. Benjamin Cumming: 1994
2. Rumbaugh,et al.*Object Oriented Modeling and Design*. Prentice Hall:1991
3. Wirfs-Brock,Rebecca,et al.*Designing Object-Oriented Software*. Prentice Hall:1990
4. Coad,peter,and Ed Yourdon.*Object Oriented Analysis*.Ypurdon Press:1991
5. Jacobson,Ivar.Object Oriented Software Engineering a Use Case Driven Approach. Addison-Wesley,1992
6. Cockburn,Alistair.Structuring Use Case with Goals.http://members.aol/com/acockburn/ papers/usecase.htm
7. Kennedy, Edward. Object Practitioner's Guide. http://www.zoo.co.uk/~z0001039/ PracGuides. Novenber 29,1997.
8. Booch,Grady.*Object Solutions*.Addison-Wesley,1995
9. Gamma,et al.*Design Pattrens*.Addison-Wesley,1995

附录 B　UML 表示法 II：统计多路复用器

在本章中，我们继续 UML 表示法的探索，这次我们关注一些更细节方面的内容。作为这次探索的上下文，我们会研究一个统计多路复用器的问题。

B.1　统计多路复用器的定义

统计多路复用器是一种设备，该设备允许多个串行数据流在一条单一的通信线路上传输。例如，考虑一台包含有一个 56K 调制解调器并且具有 16 个串口的设备。当通过电话线把两台这样的设备连接在一起时，从一台设备的串口 1 进入的字符就会从另一台设备的串口 1 中出来。这种设备可以在一个单一的调制解调器上同时支持 16 路全双工的通信会话。

图 B.1 中展示了 20 世纪 80 年代这种设备的一种典型的应用。在芝加哥我们有一些 ASCII 终端和打印机，我们想把它们连接到位于底特律的一台 VAX 机上。我们有一条连接这两个地方的 56K 租用线路。统计多路复用器在这两个地方之间创建了 16 条虚拟串行通道。

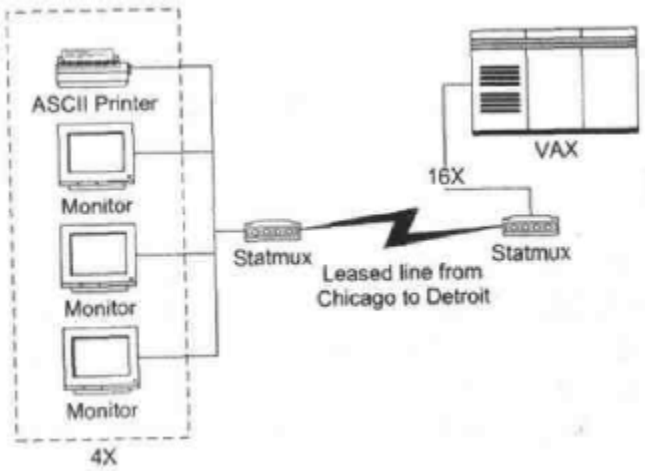


图 B.1　典型的统计多路复用器应用

显然，如果 16 条通道全部同时运转，那么它们会平均分配 56K 的吞吐量，于是每台设备的有效位速率就会略小于每秒 3500 比特。然而，大多数的终端和打印机并不是所有时间都使用的。事实上，在许多应用中，它们的有效利用率都低于 10%。因此，虽然线路是共享的，但是统计地来看，每个用户都会感觉到几乎 56K 的性能。

我们在本章中要研究的问题是统计多路复用器中的软件。该软件控制着调制解调器和串口硬件。同时，它还决定用来在所有串口之间共享通信线路的多路复用协议。

B.1.1　软件环境

图 B.2 中是一个方块图（block diagram）[①]，该图展示了软件在统计多路复用器系统中的位置。它位于 16 个串口和调制解调器之间。

① 方块图是 Kent Beck 的 GML（大型建模语言（galactic modeling language））的一种形式。GML 图由线、矩形、圆、椭圆以及理解要点所必需的任何其他形状。

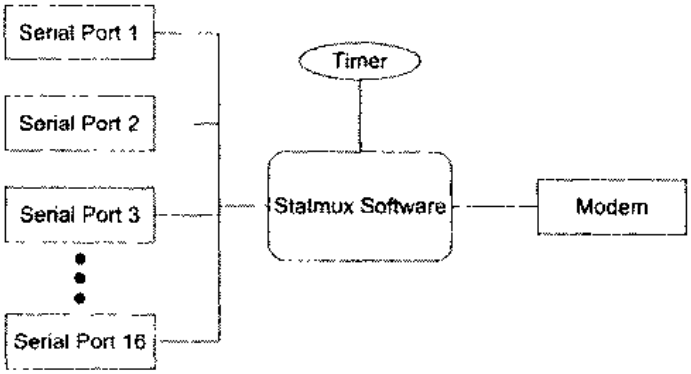


图 B.2 统计多路复用器系统方块图

每个串口都向主处理器产生两个中断——一个是在准备好发送一个字符时产生的，一个是在接收到一个字符时产生的。调制解调器也会产生类似的中断。因此，有 34 个中断进入系统。调制解调器中断的优先级高于串口中断。这保证了即使其他的串口必须具有休眠周期，调制解调器也能够以 56K 全速运行。

最后，有一个每毫秒产生一个中断的定时器。这个定时器使得软件可以安排事件在某一时间内出现特定的次数。

B.1.2 实时约束

一个小小的计算就会显示出该系统面临的问题。在任何给定的时刻，34 个中断源都会以每秒 5 600 个中断的速率来请求服务——加上来自定时器的额外的每秒 1000 个中断。总计为每秒 191 400 个中断。这样，软件服务每个中断的时间就只有 5.2us。这非常的严格（tight），我们需要一个相当快的处理器以确保不会丢掉任何一个字符。

更糟糕的是，系统要做的工作不仅仅是进行中断服务。它还得处理调制解调器之间的通信协议，收集从串口进来的字符，以及把要发送到串口的字符分配出去。所有这些都需要一些处理，这些处理必须要以某种方式安排在中断之间。

还好，系统的最大持续吞吐量只有每秒 11 200 个字符（也就是说，调制解调器可以同时收、发的字符数）。这意味着，平均来说在两个字符之间我们拥有几乎 90us。[①]

因为 ISR（中断服务例程）的持续时间不能超过 5.2us，所以在中断之间至少还有 94%的处理时间对我们可用。这意味着我们不需要过多地考虑 ISR 之外的效率问题。

B.1.3 输入中断服务例程

这段例程必须要使用汇编语言编写。输入 ISR 的主要目标是把字符从硬件中取出，并把它存储到某个非 ISR 软件在方便时可以对它进行处理的地方。处理这种情况的典型方法是使用一个环形缓冲区（ring buffer）。

图 B.3 中为一个类图，该类图展示了输入 ISR 及其环形缓冲区结构。我们发明了一些新的构造型和属性来描述一些涉及中断服务例程的非常独特的问题。

① 很长的一段时间。

最先具有的是 InputISR 类。该类具有构造型《ISR》，该构造型表明类是一个中断服务例程。这种类是用汇编语言编写的，并且只有一个方法。这个方法没有名字，并且只在中断发生时才被调用。

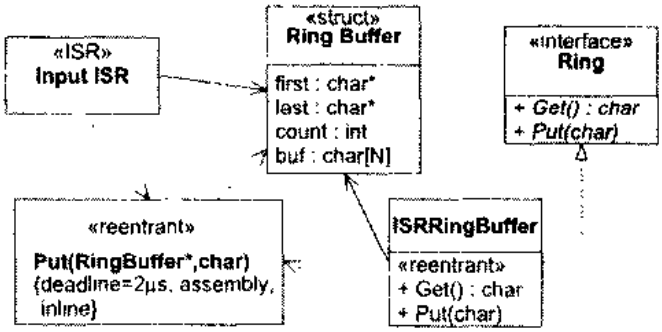


图 B.3 输入中断服务例程

InputISR 和它的 RingBuffer 间有一个关联。RingBuffer 类上的《struct》构造型表明这是一个没有方法的类。它只是一个数据结构。我们这样做是因为我们期望使用汇编语言编写的函数可以对它进行访问，这些函数无法对类的方法进行访问。

我们使用带有《reentrant》构造型的类图标来表示 Put(RingBuffer*, char)函数。以这种方式使用该构造型表示了一个中断安全的自由函数（free function）。[①]该函数把字符添加到环形缓冲区中。在接收到字符时，InputISR 会调用该函数。

关于函数的属性表明：它有一个 2us 的实时限制时间，它应该使用汇编语言编写，以及无论在什么情况下它都应该被编写成内联的。后两个属性是为了要满足第一个属性。

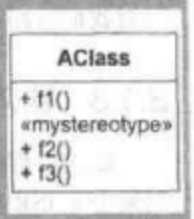
ISRRingBuffer 类是一个常规类，它的方法在中断服务例程之外执行。它使用 RingBuffer 结构并为它提供了一个 facade 类。它的方法都遵循《reentrant》构造型，因此都是中断安全的。

ISRRingBuffer 类实现了 Ring 接口。运行在中断服务例程之外的客户可以使用该接口访问存储在环形缓冲区中的字符。该类代表了中断和系统其余部分之间的接口边界。

列表框中的构造型

当构造型出现在类的列表分割栏（compartment）中时，它们具有特别的含意。那些出现在构造型下面的元素要遵循该构造型。

在本例中，函数 f1()不具有显式的构造型。但是，函数 f2()和 f3()要遵循《mystereotype》构造型。



对于像这样出现在列表框（list box）中的构造型的数目是没有限制的，每个新的构造型都会覆写前面的一个。位于两个构造型之间的所有元素都要遵循它们上面的构造型。

空构造型《》可以被用于列表框的中间，它表示在它下面的元素没有显式的构造型。

① 可重入是一个复杂的主题，超出了本章讨论的范围。读者可以参考一些优秀的有关实时和并发编程方面的书籍，比如 Doug Lea 的：*Concurrent Programming in Java*. Addison-Wesley,1997。

环形缓冲区的行为

可以使用一个“简单的”状态机（如图 B.4 中所示）来描绘环形缓冲区。状态机展示了当调用类 ISRRingBuffer 的对象的 Get()和 Put()方法时，所发生的事情。

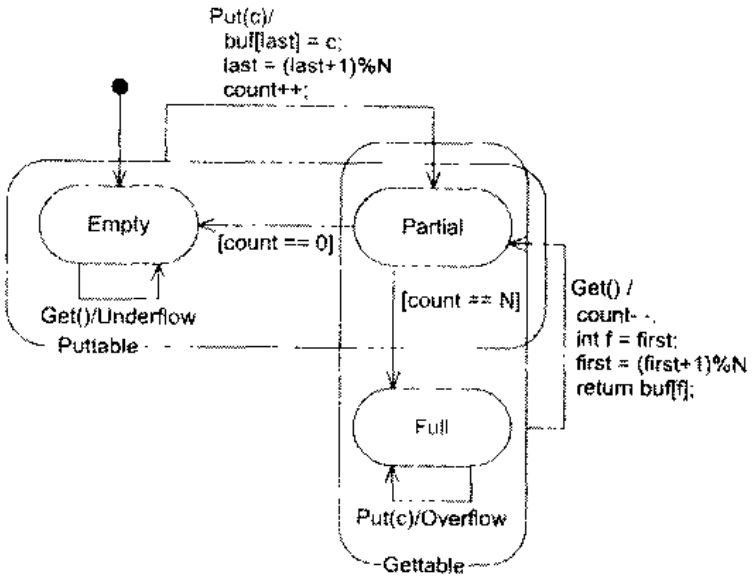


图 B.4 环形缓冲区状态图

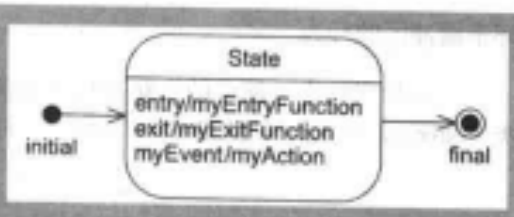
图中展示了环形缓冲区可以具有的 3 个状态。状态机开始于 Empty 状态。在 Empty 状态中，环形缓冲区中没有任何字符。在该状态下，调用 Get()方法会导致下溢（underflow）。在此，我们没有定义在下溢或者上溢期间所发生的事情——这些决策留待以后定义。Empty 和 Partial 这两个状态都是父状态（superstate）Puttable 的子状态。Puttalbe 父状态代表了那些在其中执行 Put()方法时不会引起上溢的状态。每当在 Puttable 状态的一个子状态中调用 Put 方法时，输入的字符就会被存储在缓冲区中，并且记数和索引会被相应地调整。Partial 和 Full 状态都是父状态 Gettable 的子状态。Gettalbe 父状态代表了那些在其中执行 Get()方法时不会引起下溢的状态。当在这些状态中调用 Get()时，下一个字符就会从环中去除，并返回给调用者。记数和索引会被相应地进行调整。在 Full 状态中，Put()函数会导致上溢。

两个离开 Partial 状态的迁移由监护条件（guard condition）控制。每当 count 变量的值变为 0 时，状态机就从 Partial 状态迁移到 Empty 状态。同样，每当 count 变量的值达到缓冲区的大小 N 时，状态机就从 Partial 状态迁移到 Full 状态。

B.1.4 输出中断服务例程

对输出中断的处理和对输入中断的处理非常类似。不过，还是有一些差别。系统中不涉及中断的部分把要发送的字符放到输出环形缓冲区中。每当串口准备好发送下一个字符时，就会产生一个中断。中断服务例程从环形缓冲区中取出下一个字符并把它发送到串口。

状态和内部迁移



在UML中，状态是用圆角矩形表示的。这个矩形可以具有两个分割栏。

顶部的分割栏只是对状态进行命名。如果没有指定名字，那么状态就是匿名的。所有的匿名状态都不相同。

底部的分割栏列出了该状态的内部迁移。内部迁移使用 事件名/动作 来表示。事件名必须是一个当状态机处于指定的状态时可以发生的事件的名字。状态机通过保持状态不变并执行指定的动作来响应该事件。

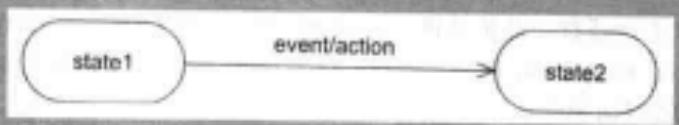
在内部迁移中，可以使用两个特殊的事件。在上面的图标中对它们进行了描述。当进入该状态时，出现 entry 事件，当离开该状态时，出现 exit 事件（即使迁移立刻又回到那个状态）。

动作可以是另外一个具有初始和终结状态的有限状态机的名字，或者是用某种计算机语言或者伪码编写的过程表达式。如果状态机中包含有对象的话，这个过程可以使用这些对象的操作和变量。动作还可以是^object.message(arg1, arg2, ...)的形式，在这种形式中，动作会把指定的消息发送给指定的对象。

在前面的图中，有两个特殊的伪状态图标。在左边，有一个表示初始伪状态的黑色圆。在右边，有一个表示终结伪状态的靶心状的圆（bull's-eye）。当状态机最开始被调用时，它会从初始伪状态迁移到和初始伪状态连接的状态。这样，初始伪状态只能具有一个离开它的迁移。当一个事件致使状态机迁移到终结伪状态时，状态机就停止运行并不再接受事件。

状态之间的迁移

有限状态机是一张由迁移连接起来的状态网。迁移是把一个状态和另一个状态连接起来的箭头线。迁移上标注有触发该迁移的事件的名字。



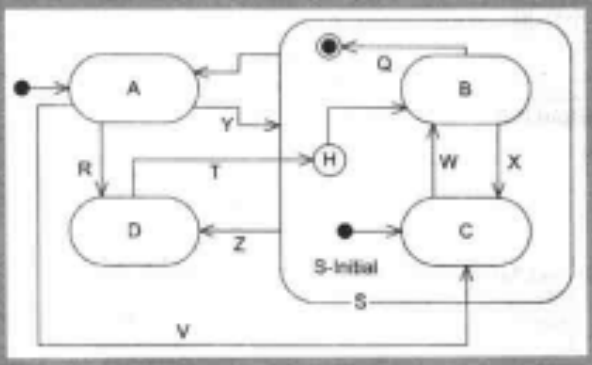
图中我们看到了被一个单一迁移连接的两个状态。如果状态机处于 state1 状态并且发生了 event 事件，迁移就会被激发（fired）。当迁移被激发时，状态机就退出 state1 状态，并且会执行所有的 exit 动作。然后，该迁移上的 action 会被执行。接着状态机进入 state2 状态，并且执行它的 entry 动作。

迁移上的事件可以用一个监护条件修饰。仅当事件发生并且监护条件为真时，才会激发迁移。监护条件是布尔表达式，它们出现在事件名后面的方括号中（例如，myEvent[myGuardCondition]）。

迁移上的动作和状态内部迁移上的动作完全相同（请参见“状态和内部迁移”）。

嵌套状态

当一个状态图标完全包围了一个或者多个其他状态图标时，被包围的状态被称为包围它们的父状态的子状态。



在上面的图中，状态 B 和 C 是父状态 S 的子状态。状态机开始于状态 A，如图所示它和初始伪状态相连。如果激发了迁移 V，那么父状态 S 中的子状态 C 就会成为活动状态。这会导致 S 和 C 的 entry 函数都被调用。

如果在状态 A 中激发了迁移 Y，那么状态机就进入父状态 S。一个到父状态的迁移必然导致它的一个子状态变成活动的。如果迁移停止在父状态矩形的边上，就像迁移 Y 那样，那么在父状态内会有一个源自初始伪状态的迁移。因此，迁移 Y 触发了从 S_Initial 伪状态到子状态 C 的迁移。

当迁移 Y 被激发时，就同时进入了父状态 S 和子状态 C。此时，S 和 C 的任何 entry 动作都会被调用。父状态的 entry 动作先于子状态的 entry 动作被调用。现在，迁移 W 和 X 可以被激发，使状态机在子状态 B 和 C 之间移动。exit 和 entry 动作的执行和往常一样，但是因为没有离开父状态 S，所以父状态的 exit 函数不会被调用。

最终，迁移 Z 会被激发。请注意，Z 是从父状态矩形的边上离开的。这意味着，不管是子状态 B 还是 C 处于活动状态，迁移 Z 都把状态机移到状态 D。这等同于两个单独的迁移。一个是从 C 到 D，另一个是从 B 到 D，这两个迁移都用 Z 来标注。当 Z 被激发时，相应的子状态的 exit 动作就被执行，然后执行父状态的 exit 动作。接着，进入状态 D，并执行它的 entry 动作。

请注意，迁移 Q 终止在一个终结伪状态上。如果激发了迁移 Q，那么父状态 S 就终止。这会激发一个未标注的从 S 到 A 的迁移。终止父状态还会像后面所述的那样复位所有的历史信息。

迁移 T 终止在父状态 S 中的一个特殊的图标。这个图标被称为 history（历史）标记。当迁移 T 被激发时，S 中最近一个活动的状态会再次变成活动的。这样，如果在 C 为活动状态时发生了迁移 Z，那么迁移 T 会导致 C 再次成为活动状态。

如果在历史标记非活动时激发了迁移 T，那么就激发从历史标记到子状态 B 的未标注迁移。这是一个没有历史信息可用时的缺省迁移。如果 S 从未被进入过，或者 S 刚刚被迁移 Q 终止之后，历史标记就是非活动的。

因此，事件序列 Y-Z-T 会使状态机停留在子状态 C。而事件序列 R-T 和 Y-W-Q-R-T 都会使状态机停留在子状态 B。

如果当串口准备好时，环形缓冲区中没有等待发送的字符，那么中断服务例程就无事可做。如果串口已经通知过它做好了接受一个新字符准备，那么它不会再次这样做直到给它一个字符发送并且完成了发送。于是，中断流就停止了。因此，我们需要一种策略来在新字符到达时重新启动输出中断。图 B.5 中展示了输出 ISR 的结构。很明显，它和图 B.3 中的输入中断是相似的。但是，请注意从 ISRRingBuffer 到 OutputISR 的《calls》依赖关系。这表明 ISRRingBuffer 对象可以引起 OutputISR 的执行，就好像接收到了一个中断一样。

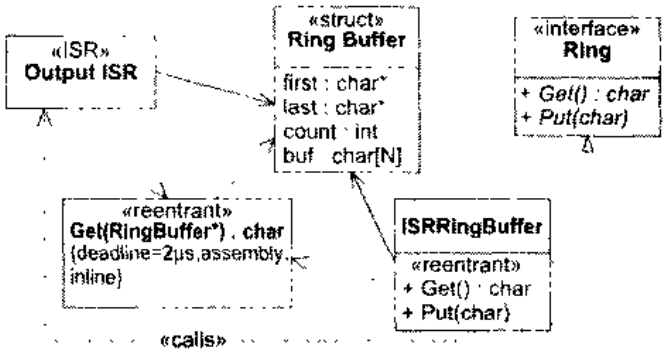


图 B.5 输出中断程序例程

图 B.6 中展示了对输出环形缓冲区的有限状态机进行的必要更改。请把该图和图 B.4 做个比较。请注意，Puttable 父状态被去掉了并且其中有两个 Put 迁移。第一个 Put 迁移从 Empty 状态到 Partial 状态。这会引起执行 Gettable 父状态的 entry 动作，该动作会产生一个触发 OutputISR 的人为中断。[①]

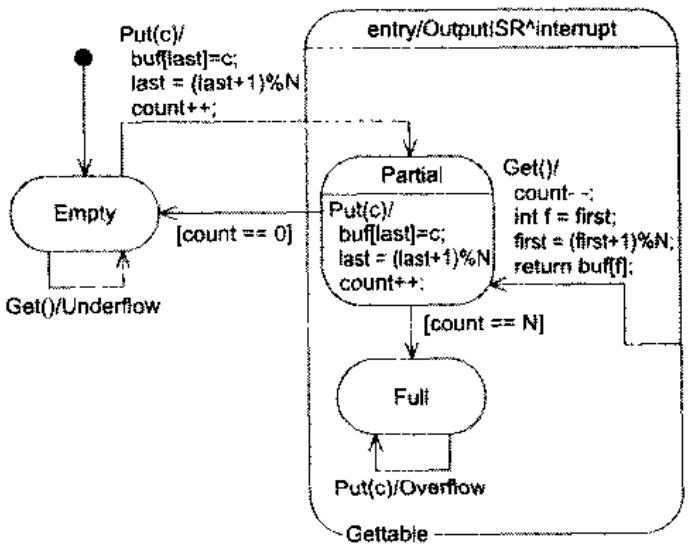


图 B.6 输出服务中断状态机

B.1.5 通信协议

两台统计多路复用器通过它们的调制解调器进行通信。每一台都通过一条通信线路向另外一台发送数据包。我们必须认为线路是非理想的，会出现一些错误和信息丢失。在传输期间，字符可能

① 以这种方式产生人为中断的机制和平台有非常强的相关性。在一些机器上，完全可以把 ISR 当做函数来调用。而另外一些机器则需要更复杂的协议来人为地调用 ISR。

会被丢失或者篡改，并且一些放电作用或者其他的电磁干扰可能会创建一些虚假的字符。因此，在两个调制解调器之间必须要有一个通信协议。这个协议必须能够验证包的完整性和正确性，并且它必须能够重新传送被篡改或者丢失的包。

图 B.7 是一个活动图（请参见“活动图”补充内容）。它展示了我们的统计多路复用器使用的通信协议。该协议是一个相对比较简单的具有管道（pipelining）和搭载（piggybacking）功能的滑动窗口协议。[1]

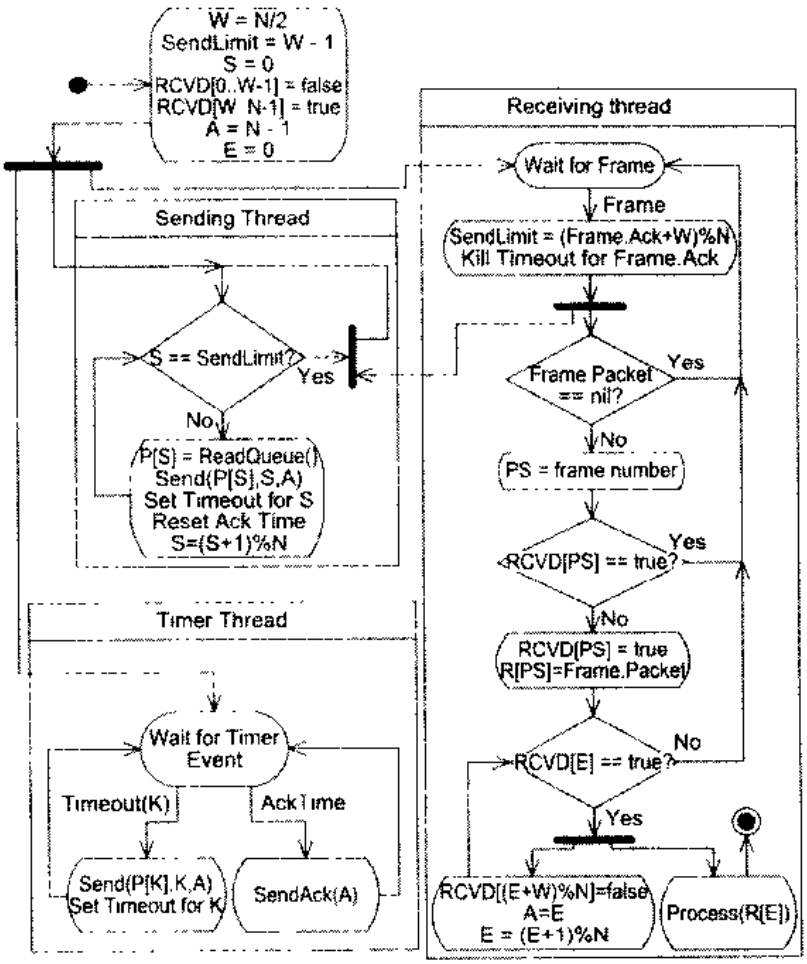


图 B.7　通信协议活动图

协议开始于初始伪状态，它初始化一些变量，接着创建了 3 个独立的线程。变量会在后面进行介绍，本小节中介绍一下依赖于这些变量的线程。如果在允许的时间段内没有接收到确认包，那么在对包进行重新传送时会用到“定时线程”。同时，它还用来确保对正确接收到的包的确认可以被及时地发送出去。“发送线程”用来发送放置在发送队列中的包。“接收线程”用来接收、检验以及处理包。我们来依次研究它们。

[1] 请参见 *Computer Networks*，2d.ed. Tanenbaum, Prentice Hall, 1998, Sec.4.4，可以获得关于这种通信协议的更多信息。

发送线程

变量 S 中包含着标记在下一个要发送出去的包上的序号（serial number）。每个包都会被编上范围在 0..N 的一个序号。发送线程会持续地发送包，不等待对它们的确认，直到没有确认的包的数目达到 W。W 被设置为 N/2，这样，在任何给定的时刻，都决不会有超过一半的序号等待确认。在正常情况下，SendLimit 变量持有超出窗口（W）范围的包的序号，因此它是目前不能被发送的最小的序号。

发送线程在持续发送包的过程中，它会以 N 为模来增加 S。当 S 到达 SendingLimit 时，发送线程就会阻塞，直到接受线程更改了 SendingLimit。如果 S 还没有到达 SendingLimit，那么 ReadQueue 函数就会从队列中取出一个新包。该包被放置在数组 P 的位置 S 处。我们把包保存在这个数组中是为了以后重传的需要。接着，该包被标记上它的序号（S）以及搭载的确认号（A）并被发送出去。A 是我们收到的最近一个包的序号。这个变量由接收线程更新。

发送完包后，我们会为所发送的包设定一个超时时间。如果在包被确认之前，超时发生了，那么定时线程就认为包或者确认包丢失了，它会重新传送包。

此时，我们还会复位 Ack 定时器。当 Ack 定时器超时时，定时线程就认为自我们上次发送一个确认以来已经过去了很长时间，于是它就认为上一个正确的包被接收到了。

接收线程

这个线程开始时会初始化一些变量。RCVD 是一个以序号作为索引的布尔标志数组。当收到包时，就在 RCVD 中把它们标记为 true。一旦处理了一个包，比该包的序号多 W 的序号位置就被标记为 false。E 是我们正在期待的并且也是下一个要被处理的包的序号；它总是等于（A+1）模 N。作为初始化的一部分，我们把 RCVD 中后面一半的值设置为 true，这意味着这些序号位于所允许的窗口之外。如果收到了它们，会把它们作为重复包丢弃。

接收线程在等待一个帧（frame），帧可能是一个包或者只是一个普通的确认。无论哪种情况，它都包含有对上一个正确包的确认。我们更新 SendingLimit 并通知发送线程。请注意，SendingLimit 被设置为最近一次确认帧的序号加上 W。于是，就只允许发送者使用从最近一次确认包开始的半数的序号空间，这样发送者和接收者就商定出了序号空间的哪一半是当前有效的。

如果帧包含了一个包，那么我们就从该包中取出序号，并检查 RCVD 数组看一下我们是否已经收到了具有该序号的包。如果收到的话，我们把它作为重复包丢弃。否则，我们就更新 RCVD 数据以表示现在已经收到了包，并把包保存在数组 R 中。

虽然包是以序号顺序发送的，但是在接收它们时却可能失序。之所以如此完全是因为包可能丢失并被重新传送。因此，即使我们刚刚收到的包序号为 PS，它可能也不是我们正期望的（E）包。如果不是所期望的，我们就只是等待直到收到了 E。然而，如果 PS == E，那么我们就创建一个新线程来处理这个包。同时，我们还把 RCVD 数组的 E+W 处的标记设置 false 来移动允许的序号窗口。最后，我们把 A 设置为 E 来表示 E 是最近一个正确收到的序号，接着我们增加 E。

定时线程

定时器只是等待一个定时器事件。有两种可能发生的事件。Timeout(K)事件表示发送线程发送了一个包，但是还没有收到任何确认。因此，定时器线程就重新发送序号为 K 的包并重新启动它的定时器。

AckTime事件由一个反复的、可重新触发的定时器产生。该定时器每X毫秒就发送一个AckTime事件。但是，可以重新触发它，使它重新定时为X。每当发送线程发送一个包时，它都会重新触发这个定时器。这样做是合适的，因为每个包都搭载一个确认信息。如果在X毫秒内没有发送任何包，那么就会发生AckTime事件，此时定时线程会发送一个确认帧。

哎呀！

你可能觉得这里的讨论理解起来有点费劲。请想一下如果没有这些说明性文本会是什么样子。图可能表达了我的意图，但是附加的文字肯定是有帮助的。图很少能够独立说明问题。

我们如何知道图是正确的呢？我们不知道！如果有读者发现图中存在问题，我一点也不奇怪。通常，无法像验证代码那样去直接验证图是正确的。因此，我们必须要等待到编码时，才知道这个算法是否真的正确。

这两个问题对这种图的有用性提出了疑问。它们可以作为很好的教育工具，但不应认为它们具有足够的表达力、足够的精确，从而成为设计的惟一规格标准。文本、代码和测试同样也是需要的。

通信协议软件的结构

3个控制线程都共享相同的变量。因此，被这些线程调用的函数应该是同一个类的方法。然而，当前的大多数线程系统都把线程当作一个对象。也就是说，每个线程都有一个控制它的对象。在UML中，它们被称为主动对象（active object）（请参见“主动对象”）。因此，含有协议方法的类同样也需要创建控制线程的主动对象。

定时线程不只是在协议中使用，所以它的线程应该在系统的其他地方创建。于是，协议对象就只需创建发送线程和接收线程。

图B.8展示了一个对象图（请参见“对象图”），该图描绘了刚刚初始化完CommunicationsProtocol对象后的情形。CommunicationsProtocol创建了两个Thread对象并负责它们的生存期。为了启动新近创建的执行线程，Thread对象使用了COMMAND[①]模式。每个Thread都持有一个遵循Runnable接口的对象实例（请参见“接口棒棒糖”）。然后，使用ADAPTER[②]模式把Thread绑定到CommunicationsProtocol对象的相应方法上。

Timer和CommunicationsProtocol之间具有类似的关系。不过，在它们的关系中，Timer对象的生存期不是由CommunicationsProtocol对象控制的。

之所以使用《friend》关系，是因为我们想让适配器调用的方法是CommunicationsProtocol的私用方法。我们不想让除了适配器之外的其他对象调用它们。[③]

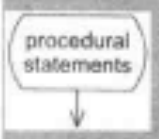
① 第15章。
② 第25章。
③ 还有其他一些方法可以实现这个目的。在Java中我们可以使用内部类。

活动图

活动图（activity diagram）是状态迁移图、流程图以及 petri 网的混合产物。它们在描绘事件驱动的多线程算法时特别有效。

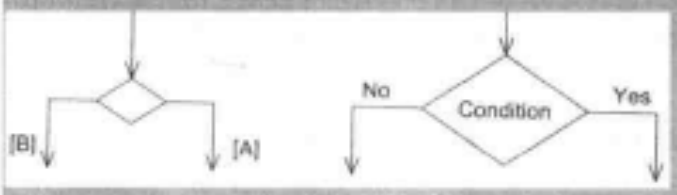
活动图是状态图。它仍然是一个由迁移连接起来的状态图。但是，在活动图中，有一些特殊种类的状态和迁移。

动作状态



动作状态（action state）被绘制成一个顶、底是平的并且两边为圆形的长方形。该图标和正常的状态图标的不同之处在于：它的角是尖的，而状态图标的角是圆的（请参见“状态和内部迁移”）。动作状态的内部含有一个或者多个表示它的 entry 动作的过程语句（这就像流程图中的一个处理框）。

当进入一个动作状态时，立即会执行它的 entry 动作。一旦完成了这些动作，就退出动作状态。从动作状态向外的迁移一定不具有事件标签，因为“事件”就是 entry 动作的完成。然而，却可能会有多个离开迁移，每个离开迁移都具有一个互斥的监护条件。所有的监护条件合起来必须总为 true（也就是说，不能被卡在动作状态中）。



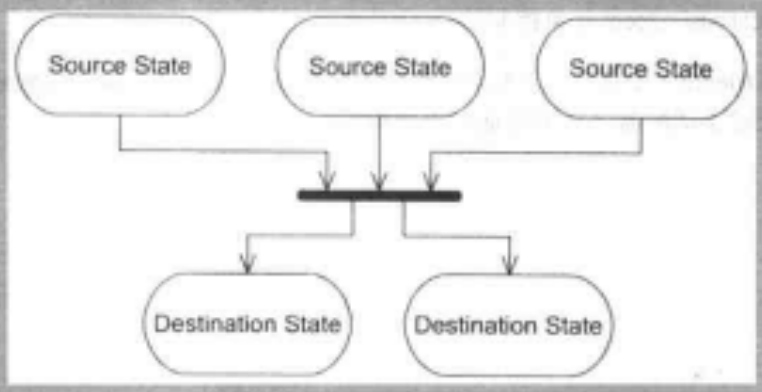
决策

一个动作状态可以具有多个离开的受监护的迁移，这就意味着它可以充当一个决策环节。不过，使用惯用的菱形图标来明确地表示决策，通常会更好一些。

一个迁移进入菱形，而多个受监护的迁移离开它。同样，所有受监护的布尔值合起来必须为 true。

在图 B.7 中，我们使用了一个更类似于流程图的菱形变体。在菱形中声明了一个布尔条件，两个离开迁移上标注有“Yes”和“No”。

复杂的迁移



复杂的迁移表现了多个控制线程（thread of control）的分开和合并。它们被表示为一个称为异步棒（asynchronization bar）的黑色棒条。状态被箭头线连接到异步棒上。通向棒条的状态被称为源

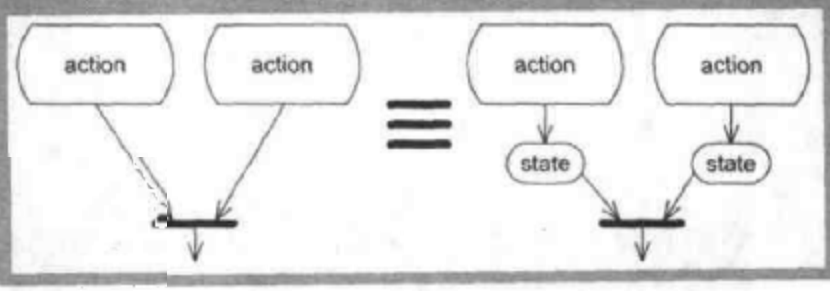
状态，从棒条离开的状态被称为目的状态。

所有通向和离开异步棒的箭头线组形成了一个单一的迁移。这些箭头线上既没有标注事件也没有标注监护条件。当所有的源状态都被占用（occupied）时（也就是说，当这 3 个独立的线程处于适当的状态时），迁移就被激发。此外，源状态必须是真正的状态而不是动作状态（也就是说，它们必须能够等待）。

迁移被激发后，会退出所有的源状态，并进入所有的目的状态。如果目的状态的数目多于源状态的数目，那么我们就创建新的控制线程。如果源状态的数目多一些，那么我们就合并一些线程。

每当进入一个源状态时，就对进入记数。每当激发了复杂迁移时，源状态中的记数值就会被减少。只要源状态的记数值不为 0，就认为它被占用。

为了在使用表示法时方便一些，真正的状态和动作状态都可以用作异步棒的源（参见图 B.7）。在这种情况下，我们假定迁移实际上终止于一个未命名的真正状态，该状态是异步棒的源状态。



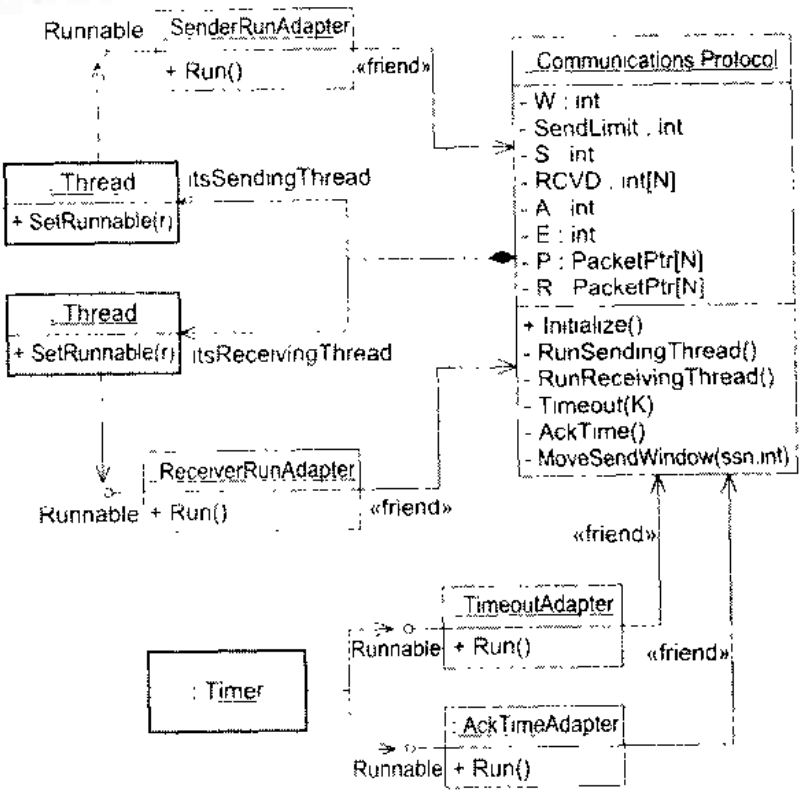
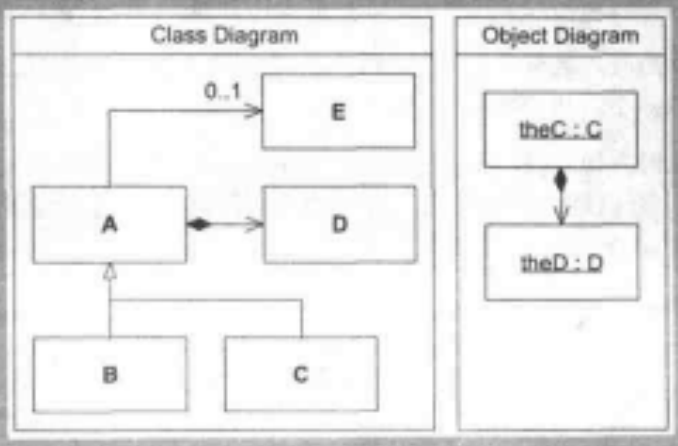


图 B.8　对象图：协议对象刚刚初始化后的情形

对象图

对象图描绘了在某个特定时刻一组对象之间存在的静态关系。它们和类图有两方面的不同。第一，它们描绘的是对象以及对象之间的连接（link），而不是类以及类之间的关系。第二，类图展示了源代码中的关系和依赖，而对象图仅仅展示对象图定义的那一时刻的运行时关系和依赖。因此，对象图展示了系统处在某个特定状态时存在的对象和连接。



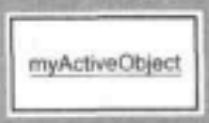
在前面的图中，我们看到了一个类图和一个表示对象的一个可能状态和连接的对象图，这些对象和连接均来自类图中的类和关系。请注意，对象的画法和顺序图中的一样。它们是带有名字的矩形，名字由两部分构成，并且带有下划线。同样请注意，对象图中的关系的画法和类图中的一样。

两个对象间的关系被称为连接。连接允许消息沿着导航（navigability）的方向流动。在本例中，消息可以从theC流动到theD。之所以存在这个连接是因为在类A和类D之间有一个组合（composition）关系，而类C派生自类A。因此，类C的实例就可以具有派生自它的基类的连接。

同样请注意，类A和类E之间的关系没有表现在对象图中。这是因为对象图描绘的是系统的一个特定状态，在这期间，C的对象还没有和E的对象关联起来。

主动对象

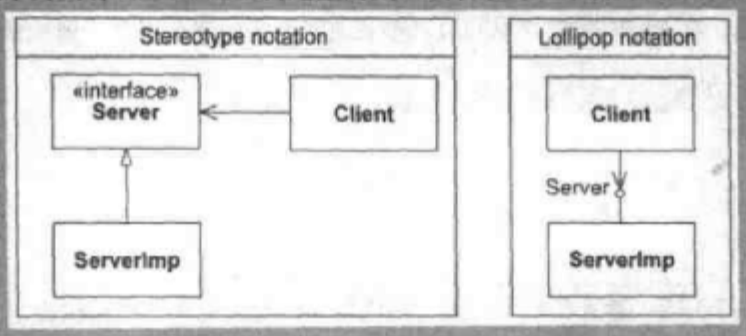
主动对象是负责一个单一执行线程（thread of execution）的对象。执行线程不需要在主动对象方法的内部运行。事实上，主动对象一般会调用其他的对象。主动对象只是一个执行线程从其中产生的对象。它也是提供管理接口的对象，比如：Terminate、Suspend以及ChangePriority。



主动对象的画法和普通对象一样，但是轮廓线是粗体。如果主动对象也拥有一些在它的控制线程中执行的其他对象，那么你可以把这些对象画在活动对象的边界内部。

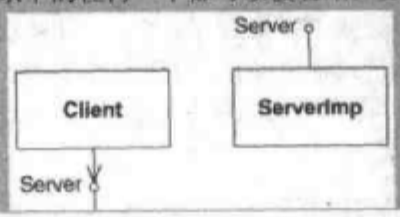
接口棒棒糖

接口可以被表示为具有《interface》构造型的类，或表示为一个特殊的棒棒糖（lollipop）图标。



方框中的两个图具有完全相同的含意。类 Client 的实例使用 Server 接口。ServerImp 类实现 Server 接口。

连接到棒棒糖图标的两个关系中的任何一个都可以被省略，如下图所示：



初始化过程

用来初始化 CommunicationsProtocol 对象的单独处理步骤如图 B.9 所示。这是一个协作图（请参见“协作图”）。

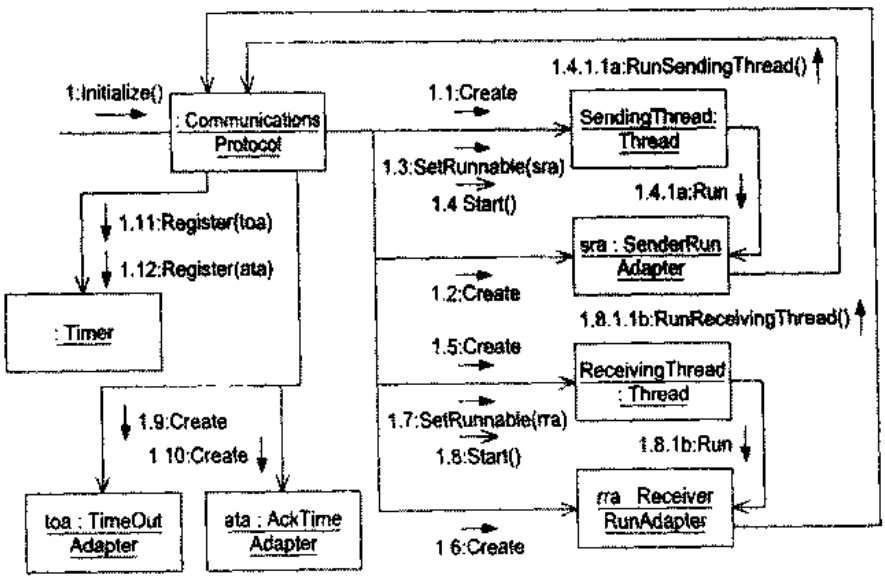
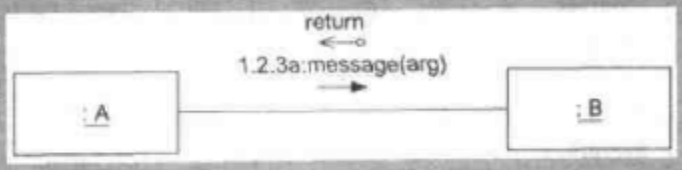


图 B.9　协作图：初始化 CommunicationsProtocol 对象

协作图

协作图和对象图相似，只是它们还展示了系统的状态是如何随时间演化的。协作图中显示了对象之间发送的消息以及它们的参数和返回值。每条消息上都标注有一个序列号来表明和其他消息之间的顺序。



消息被画为小箭头线，放置在两个对象之间连接的附近。箭头线指向接收消息的对象。消息上标注有名字和消息的序列号。

序列号和消息的名字之间用冒号分隔。消息名后面跟随有圆括号，圆括号中包含有被逗号分隔的消息的参数列表。序列号是一个用点分隔的数字列表，其后跟有可选的线程标识符。

序列号中的数字既表示了消息的顺序又表示了在它在调用层次中的深度。编号为 1 的消息是发送的第一条消息。如果消息 1 调用的过程调用了另外两条消息，那么它们会被分别编号为 1.1 和 1.2。一旦它们返回并且消息 1 完成了，下一条消息就被编号为 2。通过使用这种点分隔的方法，我们就可以完全描绘出消息的顺序和嵌套。

线程标识符是消息执行所在的线程的名字。如果线程标识符被省略了，那么就意味着消息在一个未命名的线程中执行。如果消息 1.2 创建了一个名为“t”的新线程，那么该新线程的第一条消息就会被编号为 1.2.1t。

可以使用数据表示符号（data token symbol）（在尾部有一个圆圈的小箭头线）来表示返回值和参数。此外，返回值可以在消息名中使用赋值语法来表示，如下：

```
1.2.3 : c:=message(a,b)
```

在本例中，“message”的返回值会被保存在名为“c”的变量中。

使用填充箭头的消息，如左边所示，表示一个同步函数调用。该消息直到它的过程所调用的所有其他消息都返回时，它才返回。这是 C++、Smalltalk、Eiffel 或者 Java 等的正常消息类型。

左边所示的棒状箭头表示一个异步消息。这种消息创建一个新的控制线程来执行调用的方法，然后立即返回。因此，消息是在方法被执行前返回的。方法发出的消息应该带有一个线程标识符，因为它们是在一个不同于调用线程的线程中执行的。

初始化过程开始于编号为 1 的消息。CommunicationsProtocol 从某个未知的源收到 Initialize 消息。作为响应，它在消息 1.1 和 1.2 中创建了 SendingThread 对象以及和它关联的 SenderRunAdapter。接着，在消息 1.3 和 1.4 中，它把适配器和线程绑定起来并启动线程。

请注意，消息 1.4 是异步的，所以初始化过程会继续执行消息 1.5 至 1.8，这个过程和上面是完全重复的，不过创建的是 ReceivingRhread。期间，一个独立的执行线程在消息 1.4.1a 中开始执行，它会调用 SenderRunAdapter 中的 Run 方法。结果，适配器向 CommunicationsProtocol 对象发送消息 1.4.1.1a:RunSendingThread。这启动了发送线程的处理过程。启动接收线程的事件链与此类似。最后，

消息 1.9 至 1.12 创建了定时器适配器并把它们注册到定时器中。

协议中的竞争条件

图 B.7 中描述的协议中有许多有趣的竞争条件（race condition）。当无法预知两个不同事件的顺序，而系统的状态却受这个顺序的影响时，就会出现竞争条件。于是，系统的状态就依赖于哪个事件赢得了竞争。

程序员试图确保无论事件以哪种顺序出现系统都具有正确的行为。但是，竞争条件是难以识别的。没有被发现的竞争条件可以导致不稳定的并且难以诊断的错误。

作为一个条件竞争的例子，请考虑一下当发送线程发送一个包时会发生什么（请参见图 B.10）。这种图被称为消息顺序图（请参见“消息顺序图”）。本地的发送者发送了包 S 并设定一个超时时间。远端的接收者收到这个包并让远端的发送者知道 S 被正确接收了。远端的发送者发送一个显式的 ACK 或者把 ACK 搭载到下一个包中。本地的接收者得到这个 ACK 比并终止定时器。

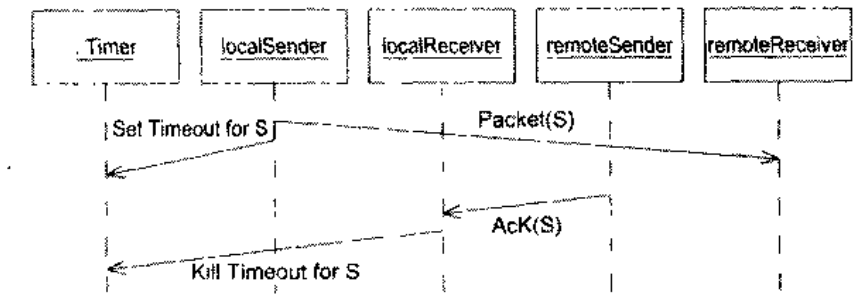


图 B.10　对包的确认：正常流程

有时，会收不到 ACK。在这种情况下，定时器就会超时，包就会被重新传送。图 B.11 展示了所发生的情况。

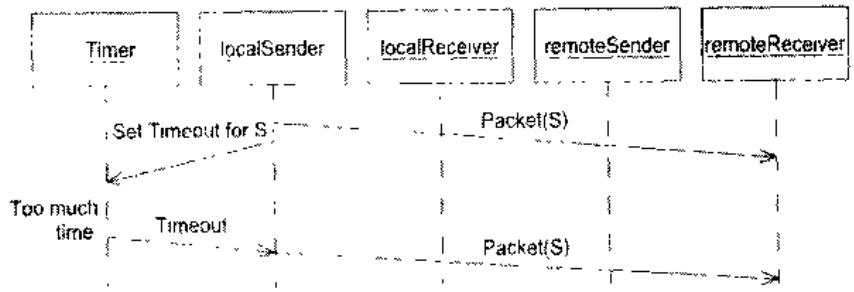


图 B.11　确认丢失：重传流程

在这两个极端情况之间存在一个竞争条件。定时器可能会在 ACK 刚刚被发出时超时。图 B.12 展示了这种情景。请注意图中的交叉线。它们代表着竞争。包 S 被发送出去并被正确地接收到。而且，一个 ACK 被传送回来。然而，ACK 是在超时发生后到达的。因此，即使收到了 ACK，包仍然被重新传送。

图 B.7 中的逻辑正确地处理了这个竞争。远端的接收者会认识到第 2 次到达的包 S 是一个重复包并把它丢弃掉。

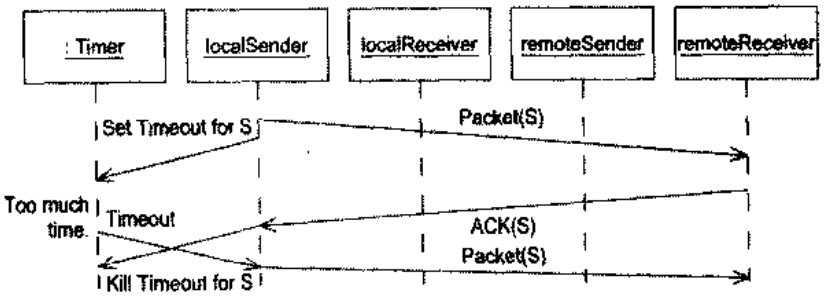


图 B.12 ACK/重传竞争条件

消息顺序图

消息顺序图是一种特殊形式的顺序图。主要的区别在于：消息箭头线向下倾斜了一个角度来表示在消息的发送和接收之间经历的时间。顺序图的所有其他部分都可以在其中使用，包括激活和序列号。

消息顺序图的主要用途是发现和文档化竞争条件。这些图非常善于显示某些事件的相对时序关系以及两个独立的处理怎样才能具有不同的事件顺序视图。

请考虑一下图 B.12。Timer 对象认为 Timeout 事件在 Kill Timeout 事件之前发生。但是，localSender 却发现这两个事件的顺序是相反的。

这个对事件顺序理解上的不同可能会导致对时序非常敏感的逻辑错误，并且非常难以重现和诊断。消息顺序图是一种非常好的工具，它们可以在这些逻辑错误在系统中造成危害之前就发现这些错误。

B.2 结　论

在本章中，我们介绍了 UML 的大多数动态建模技术。我们看到了状态机、活动图、协作图以及消息顺序图。我们同样也看到了这些图是如何处理单控制线程以及多控制线程问题的。

参考文献

1. Gamma, et al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

附录 C　两个公司的讽刺小品

“我要加入一个俱乐部，并用它来让你就范。”

——-Rufus T. Firefly

Rufus 公司
项目开始

你叫 Bob。日期是 2001 年 1 月 3 日。刚刚渡过了新千年的狂欢，你的头还疼着。你和几个管理人员以及一些同事坐在会议室中。你是一个项目团队的领导。你的上司也在其中，他叫来了归他管的所有的团队领导。他的老板召集了这次会议。

“我们有一个新项目要开发，”你老板的老板（我们叫他 BB）说。他的发尖是那么的高，都擦到天花板了。你的上司的发尖才刚开始长出，他急切地等着有一天他也可以把 Brylcream 护发香波的痕迹沾染在吸音瓦上。BB 描述了他们调查过的新市场的基本情况，以及他们想开发的用来开拓市场的产品。

“到第 4 季度，10 月 1 日时，我们必须完成这个新产品，并使之可用”，BB 要求道。“任何事情都没有它的优先级高，因此要取消你们当前的项目”。

大家的反应出奇的安静。数月的工作就这样完全被丢弃。慢慢地，低沉的反对声开始在会议室中传播。

当 BB 和房间中每个人目光相遇时，他的发尖发出邪恶的绿光。和每个人相视时的阴险目光使每个在场者不寒而栗。显然，他不容许在这个事情上进行讨论。

大家一恢复安静，BB 就说，“我们要马上开始。你们需要多长时间来进行分析呢？”

你举起了手。你的上司试图阻止你，但是他投掷的小东西没能击中你，你没有觉察到他的举动。

“先生，在得到一些需求前，我们无法告诉你分析会花费多长时间。”

“需求文档要在 3 或者 4 周后才能准备好，”BB

Rupert 工业公司
项目：~Alpha~

你叫 Robert。日期是 2001 年 3 月。在假期中你和家人所度过的轻松时光使你恢复了精神，准备投入工作。你和你的开发团队坐在会议室中。部门的管理者召集了这次会议。

“我们有一些关于一个新项目的想法，”部门管理者（称他为 Russ）说。他是一个容易激动的英国人，他的精力比聚变反应堆还要旺盛。他雄心勃勃并具有紧迫感，但是他了解团队的价值所在。

Russ 描述了公司了解的新市场机遇的基本情况，并把你介绍给 Jay，负责定义用来抓住这个机遇的产品的市场管理者。

和你打过招呼后，Jay 说，“我们希望尽快开始定义我们首次要提供的产品。你和你的团队什么时候能和我谈谈呢？”

你回答道，“本周五我们会完成我们项目的当前一次迭代。在这之间，我们可以抽出几个小时和你谈谈。迭代完成后，我们会从团队中抽出一些人专门投入到你的项目。我们会立即开始招聘一些人来接替他们并为你的团队招聘新人。”

“好极了，”Russ 说，“但是我希望你明白，7 月份我们要在产品展览会上拿出一些东西展示，这很重要。如果我们到时不能拿出一些有意义的东西，我们就会失去机会。”

“我明白，”你回答道。“虽然我还不知道你打算做的是什么，但是到 7 月时肯定可以拿出一些东西来。我还不能马上告诉你那个东西什么。无论如何，你和 Jay 将完全控制着开发人员的开发方向，所以你可以放心，到 7 月要去展示时，你会拿到可以完成的最重要的东西。”

说，他的发尖由于沮丧而震动着。“假如现在需求文档就在你面前。你需要多长时间进行分析呢？”

大家都屏住呼吸。每个人都环顾着其他人，看看他们是否有一些注意。

“如果分析需要的时间超过了 4 月 1 日，那么就会出现问题。到那时，你们能够完成分析吗？”

你的上司明显地鼓足勇气，突然说道，“先生，我们会找到办法的！”他的发尖增长了 3mm，而你的头痛也增加了，需要服两片去痛片才行。

“好。”BB 露出微笑。“现在，设计要花费多长时间呢？”

“先生，”你说。你的上司明显脸色苍白。显然，他在担心他那 3mm 会有危险。“没有分析，是不可能告诉你设计会花费多长时间的。”

BB 的表情难以置信的严厉。“假如你已经做过了分析！”他说，同时还用他那透露着无知的小圆眼注视着你。“那么，设计会花费你多长时间呢？”

两片去痛片都不能减少疼痛。你的老板，不顾一切的想保住他新增长的发尖，插嘴说道，“嗯，先生，只剩下 6 个月的时间来完成项目了，设计最好不超过 3 个月。”

“你能同意，我很高兴，Smithers！”BB 面带喜色地说。你的上司放松了一些。他知道他的发尖保住了。过了一会儿，他开始轻轻地哼起 Brylcream 的广告语。

BB 继续说道，“好，4 月 1 日前完成分析，7 月 1 日前完成设计，那么你们有 3 个月的时间实现项目。这次会议是一个榜样，它表明了我们新的协商和授权程序工作的有多么好。现在，大家可以离开，开始工作了。我期望在下周前，在我的办公桌上看到 TQM（全面质量管理）计划以及 QIT（质量改进团队）任命情况。哦，别忘了在下个月的质量审计中，你们交叉功能团队要开会并进行报告。”

“忘了去痛片，”当你返回小卧室时，心中想到。“我需要波旁威士忌酒。”

你的上司过来找你，明显带着兴奋，说道，“天哪，多么美妙的一个会议呀。我认为关于这个项目，Russ 满意地点点头。他知道这种工作方式。你的团队总是能让他了解并把握开发情况。对于你的团队会首先着手于最重要的工作并产生出高质量的产品这一点，他极有信心。

~ ~ ~

“那么 Robert，”Jay 在第一次会面时说道，“你的团队对被分开是怎么看的？”

“我们会怀念在一起工作的日子，”你回答道，“但是，我们中的一些人对于最近的一个项目相当厌倦了，并且期望有一些变化。你那边在做些什么呢？”

Jay 微笑着说，“你知道我们的客户当前遇到了很大的麻烦……”接着他用大约半个小时的时间描述了问题以及可能的解决方案。

“好，请稍等一会儿”你说道。“我需要把这搞清楚。”因此，你和 Jay 谈论了系统可能的工作方式。Jay 的一些想法并没有完全成形。你提出了一些可能的方案。他对其中的一些表示赞同。你们继续讨论。

在讨论期间，对于所提出的每个新主题，Jay 都编写了相应的用户素材卡。每张卡片都描述了新系统必须要做的事情。卡片堆积在桌子上，展开在你们面前。当你们讨论这些素材时，你和 Jay 都会指着它们，把它们拿起来，并在上面作一些记录。这些卡片是有效的助记工具，你们使用它们来描述一些刚刚成形的复杂想法。

在会谈结束时，你说，“好，我对你想要什么已经有了一个大体的认识。我和我的团队会对它进行讨论。我想他们会对各种不同的数据库结构以及表示格式进行一些试验。下次我们会面时，就会有一个团队，并且我们会开始确定系统最重要的特性。”

一周后，你新建的团队和 Jay 会面。他们把现有的用户素材卡在桌子上铺开并开始研究系统的某些细节。

会议非常有生气。Jay 把素材按照它们的重要性排好。对于每一个素材都进行了大量的讨论。开发人员关心的是要保持素材足够的小，这样便于估算和测试。所以他们不断地要求 Jay 把一个素材分成几个小一些的素材。Jay 关心的是每个素材都要有一个

我们真的会做出一些震惊世界的事情。”你愤怒得只能点头同意。

“哦，”你的上司继续说道，“我几乎忘了。”他交给你一份30页的文档。“记住，SEI下周要过来做一次评估。这是评估指南。你要把它读一遍，记住它，然后把它撕碎。它告诉你如何回答SEI审计师问你的任何问题。它还告诉你在构建过程中可以使用哪些部分内容以及避免使用哪些部分内容。到6月份时，我们会被确定为CMM3级机构。”

*　*　*

你和你的同事开始对新项目进行分析。这很困难，因为你们没有需求。但是，从BB在那个决定命运的早上所做的10分钟介绍中，你们对于产品应该做什么有了一些认识。

公司的过程要求，开始时要编写一份用例文档。你和你的团队开始列举用例并绘制椭圆图以及参与者标识图。

团队中爆发起来了争论。对于某些用例之间是用《extends》还是《includes》关系连接起来，大家有不同的意见。虽然不同的模型都被创建出来，但是没有人知道如何去评价它们。争论在继续着，明显拖延了进度。

一周后，有人找到一个网站：iceberg.com，上面推荐完全不要使用《extends》和《includes》，应该用《precedes》和《uses》来代替它们。该网站上由Don Sengrojux所写的文档描述了一个叫做Stalwart分析的方法，该方法声称可以逐步地把用例转换成设计图。

使用这个新方案，更多不同的用例模型被创建出来；但是，同样，大家对如何评价它们仍无法达成一致。争论仍在继续。

用例会议越来越多的是被情绪而不是理性所驱动。要不是因为没有需求，你早就会因为没有任何进展而心烦意乱了。

在2月15日拿到了需求文档。接着，20日、25日及此后的每周都有需求文档到来，每次新版本的需求都和前面的有冲突。虽然编写需求文档的市场人员负责此事，但是显然他们没有得到一致的意见。清晰的商业价值和优先级，倘若他对素材进行了分割，他就会保证这一点。

素材堆积在桌子上。Jay在编写它们，不过，在需要时开发人员会在上面写上注释。没有人试图去捕获所谈论的每一件事情。素材卡不必捕获所有的东西；它们只不过在会谈中起提示作用。

当开发人员对素材较满意时，他们就开始编写对它们的估算。这些估算很粗糙并且只是一种预算，但是它们却使Jay对素材的代价有了一个概念。

会谈结束时，明显还有许多素材可以讨论。同样，也清楚地知道已经明确了最重要的素材，实现这些素材需要几个月。Jay结束了会议，他带走了素材卡并承诺明天上午会拿出一份关于第一次发布的方案。

~　~　~

第二天早上，你又召集了会议。Jay挑选了5个素材卡，把它们摆放在桌子上。

“根据你们的估算，这些素材卡代表着大约50个点的工作量。上一个项目的最后一次迭代在3周内完成了50个点。如果我们可以在3周内完成这5个素材，我们就能够把它们演示给Russ看。这样，他就会对我们的进度感到特别满意。”

Jay在催促着。你从他脸上腼腆的表情可以看出他也知道这一点。你回答道，“Jay，这是一个新团队，从事的是一个新项目。期望我们和前一个团队具有同样的开发速度有点专横了。不过昨天中午我和团队谈过，事实上，我们都同意把最初的速度设定为每3周50个点。所以在这个事情上你非常幸运。”

“还请记住，”你继续说，“现在，有关素材的估算以及设定的速度都是推测出来的。在做计划时我们了解得会多一些，在实现时了解得还要再多一些。”

Jay透过他的眼镜看着你，好像要说，“在这里到底谁是上司？”接着他笑着说，“好的，不必担心，现在我已经知道了规则（drill）。”

Jay接着把另外15张素材卡放到桌子上。他说，

同时，许多团队成员又提出了一些新的不同的用例模板。每个人都以自己特有的方式来延缓进度。争论愈发激烈了。

3月1日，过程监控人员 Percival Putrigence 成功地把所有不同的用例形式和模板合成为一个单一的包含一切的形式。仅仅空白表格就有15页之多。他把所有不同模板中出现的每一个不同的地方都包含进去。同时，他还提供了一份159页的文档，描述如何填写用例表格。所有当前的用例都要按照新的标准重写。

令你大为惊奇的是，现在要回答"当用户敲击回车键时，系统应该做什么？"这个问题，需要填写15页的表格和问答题。

公司的过程（由 L.E.Ott 制订，他是"全盘分析：软件工程进步辩证法"的知名作者）坚决要求你们必须找出所有的首要用例，87%的次要用例以及36.274%的第3级用例之后，才能算完成分析并进入设计阶段。你们根本就不知道什么是第3级用例。所以，为了满足这个要求，你们就让市场部检查你们的用例文档。也许，他们知道什么是第3级用例。

糟糕的是，市场人员正忙于销售支持而无法和你们讨论。事实上，自从项目开始，你们就没能召开过任何有关市场的会议。他们所能做的最好的就是提供一份不停变化并且矛盾的需求文档。

当一个团队纠缠于无穷无尽的用例文档时，另一个团队在开发领域模型。不停变化的 UML 文档淹没了这个团队。每周，模型都要重做。团队的成员无法决定在模型中是使用《interfaces》还是使用《types》。关于 OCL 的适当语法以及应用方面，出现了很大的不同意见。团队中的有些人完全违背了5天课程中关于"分解"的内容，他们创建的图不可思议的详细和晦涩，所有其他人都无法理解。

3月27日，距离分析完成还有一周时间，你们已经产生了大量的文档和图示，但是你们对问题的分析却和1月3日时一样的浅薄。

*　*　*

接着，奇迹发生了。

"如果我们到3月底能够完成所有这些素材卡，我们就可以把系统移交给我们的 beta 版测试客户。那么我们会从他们那里得到很好的反馈。"

你回答说，"好，我们已经定义了首次迭代，并且具有了此后下3次迭代的素材。这4次迭代可以完成我们的首次发布。"

"那么，"Jay 说，"你们真的能够在接下来的3周内完成这5个素材吗？"

"我确实不知道，Jay，"你回答道，"我们来把它们分解成任务，看看能得到些什么。"

于是，在接下来的几个小时中，Jay、你和你的团队把 Jay 为首次迭代挑选的5个素材中的每一个都分解成小任务。开发人员很快就认识到某些任务可以在素材间共享，并且其他一些任务具有一些可以加以利用的公共点。很明显，开发人员的头脑中已经出现了一些可能的设计。他们不时地结成讨论小组并在一些卡片上勾勒出 UML 图。

很快，白板就被任务充满了，一旦实现了这些任务，就完成了本次迭代中的5个素材。你开始了签订过程，说道，"好，我们来签订这些任务吧。"

"我做初始的数据库生成，"Pete 说，"在最近的一个项目我做的就是这个，这看起来并不困难。我估计它需要两天时间。"

"好，那么我做登录屏幕。"Joe 说。

"噢，该死，"Elmo，团队的一个新成员，说道，"我从来没有做过 GUI，我有点想做这一个。"

"哦，年轻人真急躁。"Joe 贤明地说，并朝你使了个眼色，"你可以来协助我，年轻人"，他对 Joe 说，"我认为我需要大约3天完成它。"

开发人员一个接一个的签订了任务并估算了它们。你和 Joe 都知道让开发人员自愿选择任务要比把任务分配给他们好。你也充分地知道你不敢质疑任何一个开发人员的估算。你了解这些人，并且信任他们。你知道他们会尽最大努力的。

开发人员知道，他们签订的任务不能超过在他们参与的最近一次迭代中所完成的任务。一旦开发人员关于本次迭代的时间表安排满了，他就不再签

*　*　*

4月1日，星期天，你在家中检查你的电子邮件，看到了一封你的上司发给BB的便函。上面明确地写道你已经完成了分析！

你给上司打电话并抱怨道，“你怎么能告诉BB我们已经完成了分析呢？”

“喂，你看日历了吗？”他说，“今天是4月1日！”

你没有忘记这个日期的讽刺意味，“可是，我们还有很多问题要考虑，很多东西要分析！我们甚至还没有决定是用《extends》还是用《precedes》！”

“你凭什么说你们还没有完成？”你的上司不耐烦的问到。

“我……”

但是他打断了你，“分析永远也做不完，必须要停在某个点上。因为今天就是计划要结束的日期，所以它就停在今天。星期一，我希望你把现有的所有分析资料收集起来放到一个公共的文件夹中。把该文件开放给Percival，这样他就可以在星期一中午前把它记入CM系统。接下来就开始设计工作吧。”

当你挂断电话时，你开始思考在写字台底部的抽屉中保存一瓶波旁威士忌酒的好处。

*　*　*

他们举行了一个宴会来庆祝分析阶段的按时完成。BB发表了一通有关授权的激动人心的讲话。你的上司，另外又增长了3mm，也祝贺他的团队所表现出的不可思议的团结和团队协作。最后，CIO登台并告诉大家SEI的审计工作进行的非常顺利，并且感谢大家学习并撕碎了所发的评估指南。看来，在6月肯定可以被授予CMM3级。

（有传言说，一旦被SEI授予CMM3级，和BB同层以及更高层的管理者就可以得到丰厚的奖金）

几周过去了，你和你的团队一直在进行系统的设计。当然，你发现设计基于的假想分析是有缺陷的……不，毫无用处……不，比无用还糟。但是，当你告诉你的上司你需要返回去再多做一些分析工作以加固分析中的薄弱部分时，他只是说道，“分析订任务。

最后，所有的开发人员都停止签订任务。但是，当然，白板上仍剩有任务。

“我就担心这会发生，”你说。“好，现在只有一件事情要做，Jay。我们在这次迭代中做的过多了。我们可以去掉哪个素材或者任务呢？”

Jay叹了口气。他知道这是惟一的选择。在项目一开始就加班工作是非常愚蠢的，并且出现这种情况的项目也不会成功。

于是，Jay开始去掉最不重要的功能。“嗯，此时，我们还不是真正需要登录界面。我们完全可以在登录后的状态中启动系统。”

“胡说！”Elmo叫道，“我实在是想做这个。”

“耐心点，急性子（Grassopper），”Joe说道，“只有等蜜蜂离开蜂箱后，享受蜂蜜时才不会蛰肿嘴唇（欲速则不达）。”

Elmo显得很困惑。

每个人都显得很困惑。

“那么……”Jay继续说道，“我觉得我们也可以去掉……”

于是，任务列表渐渐地变少。失去任务的开发人员在剩余的任务中又签订了一个。

商谈的过程不是没有痛苦的。其中有几次，Jay显示出了沮丧和急躁。有一次，当局势特别紧张时，Elmo自愿要求“超时工作来弥补时间的不足。”当你正打算纠正他时，还好，这时Joe看着他说，“一旦你走上了错误的道路，它就会永远控制你的命运。”

最后，终于确定下来了一个Joe可接受的迭代。这不是Joe想要的。事实上，它比Joe想要的要少的多。但是这是团队觉得他们可以在接下来的3周中能完成的东西。并且，毕竟仍然在迭代中完成了Joe想要的最重要的事情。

“那么，Jay，”当会谈接近尾声时你说，“你何时能够提供验收测试呢？”

Jay叹了口气。这是事情的另一个方面。对于

阶段已经结束了。惟一允许做的事情是设计。现在回去设计吧。”

于是，你和你的团队尽最大努力去拼凑设计，不知道是否正确地分析了需求。当然，实际上，这也没有什么大问题，因为需求文档仍然每周都在剧烈地变动着，并且市场部仍然拒绝和你们见面。

设计是一场噩梦。你的上司最近错读了一本书：完成期限（The Finish Line），其中，作者 Mark DeThomaso 轻率地建议设计文档的详细程度应该达到代码级。

“如果我们要达到这个详细程度，”你问道，“那么为什么我们不直接去编写代码呢？”

“因为那样的话，你当然就不是在设计了。而设计阶段惟一允许做的事情就是设计。”

“此外，”他继续说，“我们刚刚购买了一个 Dandelion 的公司范围内使用的许可证！这个工具支持‘往返警报（round-the-horn）工程’！”你只要把所有的设计图传递给它，它就会为我们自动生成代码！同时，它还会保持设计图和代码的同步。”

你的上司把一个色彩明亮、用塑料薄膜包装的盒子交给你，里面装着销售版 Dandelion。你麻木地接过它，步履蹒跚地回到你的小卧室。12 小时后，你终于把该工具安装到服务器上，安装的过程经历了 8 次崩溃、一次磁盘重新格式化并玩了 8 轮 shots of 151 游戏。你想了一下你的团队参加 Dandelion 培训要浪费的那一周。接着，你露出笑容并想到，“不过在这里度过的任何一周都会是愉快的一周。”

一个接一个的设计图被你的团队创建出来。Dandelion 使这些图的绘制变得非常困难。其中会遇到大量的深层嵌套的对话框，并且必须正确填写这些对话框上的一些滑稽可笑的文本域以及检查框。接着，就会碰到在包之间移动类的问题……

起初，这些图都是来自用例的。但是由于需求的频繁变动，用例很快都变得毫无意义。

关于是应该使用 VISITOR 设计模式还是 DECORATOR 设计模式的争论爆发起来。一个开发人员拒绝使用任何形式的 VISITOR 模式，声称它不是真正的面向对象概念。另外一个拒绝使用多重继开发团队实现每个素材，Jay 必须提供一组验收测试来证明它们可以使用。并且团队远在迭代结束前就需要这些验收测试，因为它们会明确地指出 Jay 和开发人员对系统行为认识上的差异。

“今天我会提供给你一些测试脚本的例子，”Jay 许诺道，“此后的每一天，我都会增加一些。到迭代的中期，你就会拥有完整的测试集。”

~ ~ ~

迭代在周一早晨开始了，我们开了一个急速的 CRC 会议。到上午 10 点左右时，所有的开发人员都已经组合成对，并在快速地编码。

“现在，年轻的学徒，”Joe 对 Elmo 说，“你应该学习一下测试优先设计的技术！”

“哇，这听起来相当不错，”Elmo 回答道，“你是如何做的？”

Joe 微笑了一下。显然，此时他已经很想学习了。“年轻人，现在代码做了些什么呢？”

“嘿？”Elmo 回答道，“它根本什么都没有做，还没有代码呢。”

“好，考虑一下我们的任务。你能想起一些代码应该做的事情吗？”

“当然可以。”Elmo 带着年轻人的自信说道，“首先，它应该连接到数据库。”

“那么，要连接数据库，必需的东西是什么呢？”

“你说话真是古怪，”Elmo 笑着说，“我认为我们必须要从某个注册表（registry）得到数据库对象，并调用其 Connect()方法。”

“哈。敏锐的年轻奇才。你正确地觉察到了我们需要一个对象，在该对象中我们可以缓存（cacheth）数据库对象。”

“‘cacheth’是一个真实的单词吗？”

“在我说出它时，它是的！那么，我们可以编写哪些我们认为数据库注册表应该通过的测试呢？”

Elmo 叹了口气。他知道他必须得合作下去。“我们应该能够创建一个数据库对象并用 Store()方法把

承，因为它会带来麻烦。

评审会议很快就变成有关面向对象的意义、分析和设计的定义以及何时使用聚合和关联的争论。

在设计周期的中间，市场人员宣称他们重新考虑了系统的中心内容。他们彻底重新组织了一份新的需求文档。他们去掉了一些主要的特性范围，取而代之的是一些他们从客户调查中预见的更合适的特性范围。

你告诉你的上司，这些变更意味着你需要对系统的大部分内容进行重新分析和重新设计。但是他却说，“分析阶段已经结束。惟一允许做的事情是设计。现在回去设计吧。”

你建议创建一个简单的原型展示给市场人员，或者甚至是一些潜在的客户看一下，可能会好一些。但是你的上司却说，“分析阶段已经结束。惟一允许做的事情是设计。现在回去设计吧。”

拼凑、拼凑、拼凑、拼凑。你设法创建了某种也许会真实反映新需求文档的设计文档。但是，需求的彻底更改并没有导致它们停止变动。事实上，如果说有的话，就是需求文档的疯狂变动只是在频度和幅度方面有所增加。你在它们的包围中艰难的前进着。

6月15日，Dandelion的数据库遭到了破坏。显然，破坏是逐步形成的。数据库中的小错误在几个月内累积成越来越大的错误。最后，CASE工具完全停止工作了。当然，逐步形成的破坏在所有的备份中都有出现。

给Dandelion的技术支持人员打了几天的电话，都没有得到任何答复。最后，你收到了一封来自Dandelion的简短的电子邮件，通知你这是一个已知的问题，解决办法就是购买新的版本（他们承诺新版本在下季度的某个时候可以使用），然后手工重新输入所有的图。

*　*　*

接着，7月1日，另一个奇迹发生了！你完成了设计！

这次，你没有去见你的上司并抱怨什么，相反它传递给注册表。然后，我们应该能够使用Get()方法把它从注册表中取出来并证实它就是上一个对象。”

“哦，说得好，我的年轻捣蛋鬼！”

“嗨！”

“那么，现在，我们来编写一个测试函数来检验你所说的情形。”

“但是，我们不应该先编写数据库对象和注册表对象吗？”

“啊，你还有许多东西需要学习，没有耐心的年轻人。先编写测试。”

“但是这甚至无法编译！”

“你肯定呢？如果可以编译怎么办呢？”

“嗯……”

“先编写测试，Elmo。相信我。”

于是，Joe、Elmo以及所有其他的开发人员都开始编写他们的任务，每次一个测试用例。他们工作的房间中充满了结对人员之间交谈的嗡嗡声。嗡嗡声不时被高呼声打断，这些高呼呼声是某一对人员完成了一个任务或者通过了一个困难的测试用例时所发出的。

在开发的进行过程中，开发人员每1到2天就更换结对伙伴。每个开发人员都会了解所有其他人做的东西，因此关于代码的知识就广泛地在整个团队中传播。

每当一对人员完成某个重要的东西，不管是一个完整的任务或者仅仅是任务的一个重要部分，他们都会把完成的东西和系统的其余部分集成起来。这样，代码基每天都在增长，并且集成的难度被减至最小。

开发人员每天都和Jay进行交流。每当他们对系统的功能或者验收测试用例的解释有疑问时，都会去找Jay。

Jay很好的履行了他的诺言，平稳持续地给团队提供验收测试脚本。团队用心地理解这些脚本，

你在写字台中间的抽屉中放入了一些伏特加酒。

* * *

他们举行了一个宴会来庆祝设计阶段的按时完成，以及通过了 CMM3 级认证。这次，你发现 BB 的讲话非常的煽情，因此你只好躲到休息室去。

在你工作的地方遍布着一些新的标语和牌匾。上面显示着鹰和登山者的图案，并且写着关于团队协作以及授权方面的内容。上面增加了一些方格线后，辨认起来好多了。这让你想起你需要在你的文件柜中腾出点地方来放白兰地。

你和你的团队开始编码。但是你很快就发现设计在一些重要的方面存在不足。实际上，它在所有重要的方面都有缺乏。你在一个会议室中召集了设计会议，试图解决一些严重级别高一些的问题。但是你的上司在会议室中抓住你并解散了会议，说，“设计阶段已经结束。惟一允许做的事情是编码。现在回去编码吧。”

Dandelion 生成的代码实在是丑陋。你和你的团队终究还是误用了关联和聚合。为了改正这些错误，必须得编辑所有生成的代码。编辑这种代码异常的困难，因为它上面被添加了一些具有特殊语法的丑陋注释块，Dandelion 要使用这些注释来保持图和代码之间的同步。如果你不小心更改了某个注释，那么重新生成的图就会不正确。结果表明，“往返警报（round-the-horn）工程”还需要非常多的工作要做。

你越是想保持代码和 Dandelion 兼容，Dandelion 产生的错误就越多。最后，你放弃了这种做法，并决定手工地使图保持最新。一秒钟后，你发现使图保持最新根本没有意义。此外，以谁的时间为准呢？

你的上司雇佣了一个顾问来构建一个计算所编写的代码行数的工具。他把一张很大的坐标纸贴在墙上，在顶部标出了数字 1 000 000。每天他都会延长红线来显示增加了多少行代码。

贴出坐标纸 3 天后，你的上司在大厅里拦住你。“那张图增长的不够快。我们要在 10 月 1 日时完成 100 万行代码。”

“我们还不确信该产品会需要 100 万行代码，”从而对 Jay 期望系统做的东西有了更好的理解。

到第 2 周初时，所完成的功能已经足以演示给 Jay 看。Jay 热切地观看着，演示通过了一个接一个的测试用例。

“这真是太棒了，”当演示最后结束时，Jay 说道，“但是这看起来好像不到 1/3 的任务。你们的速度比预期的慢吗？”

你皱起眉头。你本来想等待一个合适的时机把这告诉 Jay，但是现在他却提前提出了这个问题。

“是的，很遗憾，我们比期望的要慢一些。我们使用的新应用服务器配置起来很费劲。而且，还得常常重新启动它，每次即使我们对它的配置做了最微小的更改，都必须得重新启动它。”

Jay 用怀疑的眼光看着你。上周一商谈中的紧张状态还没有完全消散。他说，“那么，这对我们的进度意味着什么呢？我们不能再落后进度了，绝对不能。Russ 会很生气的！他会惩罚我们所有人，并为我们增加一些新人手。”

你一直看着 Jay。这样的消息是没有办法以令人愉快的方法说出来的。于是你完全不加思索的说道，“看，如果事情还像这样进行下去，那么到下周五时，我们将不能完成所有的东西！现在我们是有可能找出一条可以快一些的方法的。但是，坦白地说，我不会依赖于它的。你应该考虑一下从迭代中去掉 1 个或者 2 个任务，而又不破坏给 Russ 的演示。无论如何，我们都会在周五进行演示的，并且我认为你不会想让我们来挑选去掉哪些任务。”

“啊，看在上帝的面上！”当 Jay 摇着头大步离开时，几乎无法抑制住喊出最后一句话。

不止一次，你对自己说，“从来没有人敢向我保证项目管理会是容易的。”你非常肯定这也不会是最后一次。

~ ~ ~

实际的情况要比你期望的稍好一点。事实上，团队确实从迭代中去掉一个任务，但是 Jay 做了明智地选择，所以给 Russ 的演示很顺利。

Russ 对进度没有太深的印象，但是他也没有感

你急着说。

“我们必须在 10 月 1 日时完成 100 万行代码，”你的上司重复着。他的发尖再一次增长了，并且他在它们上面使用的希腊式配方营造出一种权威和能力的氛围。“你确信你们的注释块足够大吗？”

接着，他立刻闪现出了管理方面的洞察力，说，“我知道了！，任何一行代码都不能超过 20 个字符。任何超过 20 个字符的代码行必须得分成两行或者更多的行——越多越好。现有的所有代码都必须按这个标准改写。这会使我们的代码行增加！”

你决定不告诉他这需要 2 个计划外的人月。你决定根本不告诉他任何事情。你觉得静脉注射酒精是惟一的办法。你做了适当的安排。

拼凑、拼凑、拼凑还是拼凑。你和你的团队疯狂地编码。到 8 月 1 日，你的上司皱着眉看着墙上的坐标纸，制定出了强制性的每周要工作 50 小时。

拼凑、拼凑、拼凑还是拼凑。到 9 月 1 日，坐标图显示代码有 120 万行，你的上司让你写一个报告描述一下你们为何超出代码预算 20%。他制定了强制性的周六加班，并要求项目代码减少到 100 万行代码。你们开始着手对代码进行重新合并。

拼凑、拼凑、拼凑还是拼凑。脾气变得暴躁；人员一个一个地辞职；QA 把大量的故障报告发给你。客户在要求安装产品以及用户手册；销售人员要求给特殊的客户进行一些预先的演示；需求文档仍然在变动；市场人员在抱怨产品根本不是他们所要的，卖酒的店铺也不再卖给你酒了。必须要交出一些东西了。9 月 15 日，BB 召开了一次会议。

当他走进会议室时，他的发尖散发着朦胧的雾气。当他说话时，他精心修饰过的低音致使你的胸口要翻转过来。“QA 的管理人员告诉我，这个项目只实现了不到 50% 的必需的特性。他还告诉我系统总是会崩溃，产生错误的结果，并且非常慢。他还抱怨他无法跟上连续的每日发布，每次发布都比上一次出现更多的错误！”

他停顿了几秒，明显想镇定一下。“QA 的管理人员估计，像这样开发下去，要到 12 月份，我们才能够发售产品！”

到沮丧。他只是说，“相当好。但是记住，我们必须要能够在 7 月份的展览会上进行演示，如果以这样的速度的话，看起来你们不能完成所有的要展示的东西。”

Jay 的态度在迭代完成后有了很大的改善，他回答 Russ 说，“Russ，这个团队工作的很努力，也很好。到 7 月时，我确信我们会有一些最重要的东西去演示。它不是所有的东西，并且其中的一些可能没用真正实现，但是我们会有一些东西的。”

虽然刚刚结束的迭代很费劲，但是它却校准了你们的开发速度。接下来的迭代好了许多。这并不是因为你的团队完成了比上一次更多的任务，而只是因为不必在迭代的中期去掉任何的任务或者素材。

到第 4 次迭代开始时，一个自然的开发节奏被建立起来了。Jay、你以及开发团队都可以准确地知道彼此期望的是什么。虽然团队工作的很艰苦，但是开发速度却是可持续的。你确信团队能够保持这个速度一年或者更长的时间。

在进度方面几乎没有出现什么问题；但是在需求方面却非如此。Jay 和 Russ 常常检查逐渐增长的系统并对现有的功能提出一些建议和更改。但是任何一方都知道这些更改是花费时间的并且必须要被列入计划。因此，更改没有导致对任何人期望的违背。

在 3 月份，给董事会做了一个该系统的较大型的演示。系统功能非常的有限，还不足以拿到展示会上去演示，但是进展却非常稳定，给董事会留下了相当深刻的印象。

第 2 次发布甚至比第 1 次还要顺利。现在，团队已经找到了一个可以自动执行 Jay 的验收测试脚本的方法。他们同样也对系统进行重构，直到确实可以容易地向其中增加新特性以及更改原有的特性。

到 6 月底完成了第 2 次发布，并被拿到展览会上。系统的功能要比 Jay 和 Rus 原本想要的少一些，但是它确实演示了系统最重要的特性。虽然展览会上的客户注意到了某些功能没有实现，但是在总体上却给他们留下了深刻的印象。你、Russ 以及 Jay

事实上，你认为更可能在明年 3 月，但是你什么也没有说。

“12 月！”BB 吼叫着，面带着嘲笑，每个人都低下头就好像他正用一只突击步枪对准自己一样。“12 月是绝对不行的。团队领导们，我希望明天上午在我的办公桌上见到新的估算。因此，我要求每周工作 65 小时直到这个项目完成。最好能在 11 月 1 日完成。”

当离开会议室时，就听他嘀咕道，“授权——呸！”

* * *

你的上司秃顶了；他的发尖被安放在 BB 的墙上。荧光灯照在他的头顶所反射的光很快使你眼花。

“你这儿有喝的东西吗？”他问到。刚刚喝完你最后一瓶 Boone's Farm，你又从书架上取下一瓶 Thunderbird 并倒入他的咖啡杯中。“怎样做才能完成这个项目？”他问到。

“我们需要冻结需求，分析它们，设计它们，然后实现它们。”你麻木地回答。

“到 11 月 1 日？”你的上司怀疑地大叫到 。“不！赶快回去编写这该死的东西。”他抓着他那光秃秃的脑袋气冲冲的走了。

几天后，你发现你的上司被调到公司研究部门。销售量大幅地增长。客户一知道他们的订单无法按时完成，就立即要取消他们的订单。根据市场情况，又对该产品是否符合公司的总体目标进行了评估，等等，等等。信函乱飞，人员被免职，政策改变，总的来说，事态变得相当严峻。

最后，到 3 月份。经过了大量的 65 小时工作周后。一个非常不可靠的版本完成了。实地使用时，错误的出现率非常高，技术支持人员对于发怒的客户的抱怨和要求束手无策。所有人都不高兴。

4 月，BB 决定通过购买的方式来解决问题，他购买了由 Rupert 工业公司开发的产品的使用授权并重新销售。客户的怒火被平息了，市场人员沾沾自喜，而你被解雇了。

在从展览会上返回时都面带笑容。你们都仿佛觉得这个项目是一个胜利者。

事实上，许多个月以后，Rufus 公司和你们进行了联系。他们曾经为了他们的内部业务开发过一个类似的系统。经历过一个死亡的项目后，他们取消了这个系统的开发，并和你们商谈有关在他们的环境中使用你们的技术的授权许可事宜。

情况确实在不断变好！

附录 D　源代码就是设计

至今，我仍能记起当我顿悟并最终产生下面文章时所在的地方。那是 1986 年的夏天，我在加利福尼亚中国湖海军武器中心担任临时顾问。在这期间，我有幸参加了一个关于 Ada 的研讨会。讨论当中，有一位听众提出了一个具有代表性的问题，“软件开发者是工程师吗？”我不记得当时的回答，但是我却记得当时并没有真正解答这个问题。于是，我就退出讨论，开始思考我会怎样回答这样一个问题。现在，我无法肯定当时我为什么会记起几乎 10 年前曾经在 Datamation 杂志上阅读过的一篇论文，不过促使我记起的应该是后续讨论中的某些东西。这篇论文阐述了工程师为什么必须是好的作家（我记得该论文谈论就是这个问题——好久没有看了），但是我从该论文中得到的关键一点是：作者认为工程过程的最终结果是文档。换句话说，工程师生产的是文档，不是实物。其他人根据这些文档去制造实物。于是，我就在困惑中提出了一个问题，“除了软件项目正常产生的所有文档以外，还有可以被认为是真正的工程文档的东西吗？”我给出的回答是，“是的，有这样的文档存在，并且只有一份——源代码。”

把源代码看作是一份工程文档——设计——完全颠覆了我对自己所选择的职业的看法。它改变了我看待一切事情的方式。此外，我对它思考的越多，我就越觉得它阐明了软件项目常常遇到的众多问题。更确切地说，我觉得大多数人不理解这个不同的看法，或者有意拒绝它这样一个事实，就足以说明很多问题。几年后，我终于有机会把我的观点公开发表。C++ Journal 中的一篇有关软件设计的论文促使我给编辑写了一封关于这个主题的信。经过几封书信交换后，编辑 Livleen Singh 同意把我关于这个主题的想法发表为一篇论文。下面就是这篇文章。

——Jack Reecves, December,22,2001

什么是软件设计

©Jack W.Reeves, 1992

面向对象技术，特别是 C++，似乎给软件界带来了不小的震动。出现了大量的论文和书籍去描述如何应用这项新技术。总的来说，那些关于面向对象技术是否只是一个骗局的问题已经被那些关于如何付出最小的努力即可获得收益的问题所替代。面向对象技术出现已经有一段时间了，但是这种爆炸式的流行却似乎有点不寻常。人们为何会突然关注它呢？对于这个问题，人们给出了各种各样的解释。事实上，很可能就没有单一的原因。也许，把多种因素的结合起来才能最终取得突破，并且这项工作正在进展之中。尽管如此，在软件革命的这个最新阶段中，C++本身看起来似乎成为了一个主要因素。同样，对于这个问题，很可能也存在很多种理由，不过我想从一个稍微不同的视角给出一个答案：C++之所以变得流行，是因为它使软件设计变得更容易的同时，也使编程变得更容易。

虽然这个解释好像有点奇特，但是它却是深思熟虑的结果。在这篇论文中，我就是想要关注一下编程和软件设计之间的关系。近 10 年来，我一直觉得整个软件行业都没有觉察到做一个软件设计和什么是真正的软件设计之间的一个微妙的不同点。只要看到了这一点，我认为我们就可以从 C++

增长的流行趋势中，学到关于如何才能成为更好的软件工程师的意义深远的知识。这个知识就是，编程不是构建软件，而是设计软件。

几年前，我参见了一个讨论会，其中讨论到软件开发是否是一门工程学科的问题。虽然我不记得了讨论结果，但是我却记得它是如何促使我认识到：软件业已经做出了一些错误的和硬件工程的比较，而忽视了一些绝对正确的对比。其实，我认为我们不是软件工程师，因为我们没有认识到什么才是真正的软件设计。现在，我对这一点更是确信无疑。

任何工程活动的最终目标都是某些类型的文档。当设计工作完成时，设计文档就被转交给制造团队。该团队是一个和设计团队完全不同的群体，并且其技能也和设计团队完全不同。如果设计文档正确地描绘了一个完整的设计，那么制造团队就可以着手构建产品。事实上，他们可以着手构建该产品的许多实物，完全无需设计者的任何进一步的介入。在按照我的理解方式审查了软件开发的生命周期后，我得出一个结论：实际上满足工程设计标准的惟一软件文档，就是源代码清单。

对于这个观点，人们进行了很多的争论，无论是赞成的还是反对的都足以写成无数的论文。本文假定最终的源代码就是真正的软件设计，然后仔细研究了该假定带来的一些结果。我可能无法证明这个观点是正确的，但是我希望证明：它确实解释了软件行业中一些已经观察到的事实，包括C++的流行。

在把代码看作是软件设计所带来的结果中，有一个结果完全盖过了所有其他的结果。它非常重要并且非常明显，也正因为如此，对于大多数软件机构来说，它完全是一个盲点。这个结果就是：软件的构建是廉价的。它根本就不具有昂贵的资格；它非常的廉价，几乎就是免费的。如果源代码是软件设计，那么实际的软件构建就是由编译器和连接器完成的。我们常常把编译和连接一个完整的软件系统的过程称为“进行一次构建”。在软件构建设备上所进行的主要投资是很少的——实际需要的只有一台计算机、一个编辑器、一个编译器以及一个连接器。一旦具有了一个构建环境，那么实际的软件构建只需花费少许的时间。编译50 000行的C++程序也许会花费很长的时间，但是构建一个具有和50 000行C++程序同样设计复杂性的硬件系统要花费多长的时间呢？

把源代码看作是软件设计的另外一个结果是，软件设计相对易于创作，至少在机械意义上如此。通常，编写（也就是设计）一个具有代表性的软件模块（50至100行代码）只需花费几天的时间（对它进行完全的调试是另外一个议题，稍后会对它进行更多的讨论）。我很想问一下，是否还有任何其他的学科可以在如此短的时间内，产生出和软件具有同样复杂性的设计来，不过，首先我们必须要弄清出如何来度量和比较复杂性。然而，有一点是明显的，那就是软件设计可以极为迅速地变得非常庞大。

假设软件设计相对易于创作，并且在本质上构建起来也没有什么代价，一个不令人吃惊的发现是，软件设计往往是难以置信的庞大和复杂。这看起来似乎很明显，但是问题的重要性却常常被忽视。学校中的项目通常具有数千行的代码。具有10 000行代码（设计）的软件产品被它们的设计者丢弃的情况也是有的。我们早就不再关注于简单的软件。典型的商业软件的设计都是由数十万行代码组成的。许多软件设计达到了上百万行代码。另外，软件设计几乎总是在不断地演化。虽然当前的设计可能只有几千行代码，但是在产品的生命期中，实际上可能要编写许多倍的代码。

尽管确实存在一些硬件设计，它们看起来似乎和软件设计一样复杂，但是请注意两个有关现代硬件的事实。第一，复杂的硬件工程成果未必总是没有错误的，在这一点上，它不存在像软件那样让我们相信的评判标准。多数的微处理器在发售时都具有一些逻辑错误：桥梁坍塌，大坝破裂，飞机失事以及数以千计的汽车和其他消费品被召回——所有的这些我们都记忆犹新，所有的这些都是

设计错误的结果。第二，复杂的硬件设计具有与之对应的复杂、昂贵的构建阶段。结果，制造这种系统所需的能力限制了真正能够生产复杂硬件设计公司的数目。对于软件来说，没有这种限制。目前，已经有数以百计的软件机构和数以千计的非常复杂的软件系统存在，并且数量以及复杂性每天都在增长。这意味着软件行业不可能通过仿效硬件开发者找到针对自身问题的解决办法。倘若一定要说出有什么相同之处的话，那就是，当 CAD 和 CAM 可以做到帮助硬件设计者创建越来越复杂的设计时，硬件工程才会变得和软件开发越来越像。

设计软件是一种管理复杂性的活动。复杂性存在于软件设计本身之中，存在于公司的软件机构之中，也存在于整个软件行业之中。软件设计和系统设计非常相似。它可以跨越多种技术并且常常涉及多个学科分支。软件的规格说明往往不固定、经常快速变化，这种变化常常在正进行软件设计时发生。同样，软件开发团队也往往不固定，常常在设计过程的中间发生变化。在许多方面，软件都要比硬件更像复杂的社会或者有机系统。所有这些都使得软件设计成为了一个困难的并且易出错的过程。虽然所有这些都不是创造性的想法，但是在软件工程革命开始将近 30 年后的今天，和其他工程行业相比，软件开发看起来仍然像是一种未受过训练（undisciplined）的技艺。

一般的看法认为，当真正的工程师完成了一个设计，不管该设计有多么复杂，他们都非常确信该设计是可以工作的。他们也非常确信该设计可以使用公认的技术构建出来。为了做到这一点，硬件工程师花费了大量的时间去验证和改进他们的设计。例如，请考虑一个桥梁设计。在这样一个设计实际构建之前，工程师会进行结构分析——他们建立计算机模型并进行仿真，他们建立比例模型并在风洞中或者用其他一些方法进行测试。简而言之，在建造前，设计者会使用他们能够想到的一切方法来证实设计是正确的。对于一架新型客机的设计来说，情况甚至更加严重：必须要构建出和原物同尺寸的原型，并且必须要进行飞行测试来验证设计中的种种预计。

对于大多数人来说，软件中明显不存在和硬件设计同样严格的工程。然而，如果我们把源代码看做是设计，那么就会发现软件工程师实际上对他们的设计做了大量的验证和改进。软件工程师不把这称为工程，而称它为测试和调试。大多数人不把测试和调试看作是真正的“工程”——在软件行业中肯定没有被看作是。造成这种看法的原因，更多的是因为软件行业拒绝把代码看作设计，而不是任何实际的工程差别。事实上，试验模型、原型以及电路试验板已经成为其他工程学科公认的组成部分。软件设计者之所以不具有或者没有使用更多的正规方法来验证他们的设计，是因为软件构建周期的简单经济规律。

第一个启示：仅仅构建设计并测试它比做任何其他事情要廉价一些，也简单一些。我们不关心做了多少次构建——这些构建在时间方面的代价几乎为零，并且如果我们丢弃了构建，那么它所使用的资源完全可以重新利用。请注意，测试并非仅仅是让当前的设计正确，它也是改进设计的过程的一部分。复杂系统的硬件工程师常常建立模型（或者，至少他们把设计用计算机图形直观地表现出来）。这就使得他们获得了对于设计的一种“感觉”，而仅仅去检查设计是不可能获得这种感觉的。对于软件设计来说，构建这样一个模型既不可能也无必要。我们仅仅构建产品本身。即使正规的软件验证可以和编译器一样自动进行，我们还是会去进行构建/测试循环。因此，正规的验证对于软件行业来说从来没有太多的实际意义。

这就是现今软件开发过程的现实。数量不断增长的人和机构正在创建着更加复杂的软件设计。这些设计会被先用某些编程语言编写出来，然后通过构建/测试循环进行验证和改进。过程易于出错，并且不是特别的严格。相当多的软件开发人员并不想相信这就是过程的运作方式，也正因为这一点，使问题变得更加复杂。

当前大多数的软件过程都试图把软件设计的不同阶段分离到不同的类别中。必须要在顶层的设计完成并且冻结后，才能开始编码。测试和调试只对清除建造错误是必要的。程序员处在中间位置，他们是软件行业的建造工人。许多人认为，如果我们可以让程序员不再进行“随意的编码（hacking）”并且按照交给他们的设计去进行构建（还要在过程中，犯更少的错误），那么软件开发就可以变得成熟，从而成为一门真正的工程学科。但是，只要过程忽视了工程和经济学事实，这就不可能发生。

例如，任何一个现代行业都无法忍受在其制造过程中出现超过 100%的返工率。如果一个建造工人常常不能在第一次就构建正确，那么不久他就会失业。但是在软件业中，即使最小的一块代码，在测试和调试期间，也很可能会被修正或者完全重写。在一个创造性的过程中（比如：设计），我们认可这种改进不是制造过程的一部分。没有人会期望工程师第一次就创建出完美的设计。即使她做到了，仍然必须让它经受改进过程，目的就是为了证明它是完美的。

即使我们从日本的管理方法中没有学到任何东西，我们也应该知道由于在过程中犯错误而去责备工人是无益于提高生产率的。我们不应该不断地强迫软件开发去符合不正确的过程模型，相反，我们需要去改进过程，使之有助于而不是阻碍产生更好的软件。这就是“软件工程”的石蕊测试。工程是关于你如何实施过程的，而不是关于是否需要一个 CAD 系统来产生最终的设计文档。

关于软件开发有一个压倒性的问题，那就是一切都是设计过程的一部分。编码是设计，测试和调试是设计的一部分，并且我们通常认为的设计仍然是设计的一部分。虽然软件构建起来很廉价，但是设计起来却是难以置信的昂贵。软件非常的复杂，具有众多不同方面的设计内容以及它们所导致的设计考虑。问题在于，所有不同方面的内容是相互关连的（就像硬件工程中的一样）。我们希望顶层设计者可以忽视模块算法设计的细节。同样，我们希望程序员在设计模块内部算法时不必考虑顶层设计问题。糟糕的是，一个设计层面中的问题侵入到了其他层面之中。对于整个软件系统的成功来说，为一个特定模块选择算法可能和任何一个更高层次的设计问题同样重要。在软件设计的不同方面内容中，不存在重要性的等级。最低层模块中的一个不正确设计可能和最高层中的错误一样致命。软件设计必须在所有的方面都是完整和正确的，否则，构建于该设计基础之上的所有软件都会是错误的。

为了管理复杂性，软件被分层设计。当程序员在考虑一个模块的详细设计时，可能还有数以百计的其他模块以及数以千计的细节，他不可能同时顾及。例如，在软件设计中，有一些重要方面的内容不是完全属于数据结构和算法的范畴。在理想情况下，程序员不应该在设计代码时还得去考虑设计的这些其他方面的内容。

但是，设计并不是以这种方式工作的，并且原因也开始变得明朗。软件设计只有在其被编写和测试后才算完成。测试是设计验证和改进过程的基础部分。高层结构的设计不是完整的软件设计；它只是细节设计的一个结构框架。在严格地验证高层设计方面，我们的能力是非常有限的。详细设计最终会对高层设计造成的影响至少和其他的因素一样多（或者应该允许这种影响）。对设计的各个方面进行改进，是一个应该贯穿整个设计周期的过程。如果设计的任何一个方面内容被冻结在改进过程之外，那么对于最终设计将会是糟糕的或者甚至无法工作这一点，就不会觉得奇怪了。

如果高层的软件设计可以成为一个更加严格的工程过程，那该有多好呀，但是软件系统的真实情况不是严格的。软件非常的复杂，它依赖于太多的其他东西。或许，某些硬件没有按照设计者认为的那样工作，或者一个库例程具有一个文档中没有说明的限制。每一个软件项目迟早都会遇到这些种类的问题。这些种类的问题会在测试期间被发现（如果我们的测试工作做得好的话），之所以如此是因为没有办法在早期就发现它们。当它们被发现时，就迫使对设计进行更改。如果我们幸运，

那么对设计的更改是局部的。时常，更改会波及到整个软件设计中的一些重要部分（莫菲定律）。当受到影响的设计的一部分由于某种原因不能更改时，那么为了能够适应影响，设计的其他部分就必须得遭到破坏。这通常导致的结果就是管理者所认为的“随意编码”，但是这就是软件开发的现实。

例如，在我最近工作的一个项目中，发现了模块 A 的内部结构和另一个模块 B 之间的一个时序依赖关系。糟糕的是，模块 A 的内部结构隐藏在一个抽象体的后面，而该抽象体不允许以任何方法把对模块 B 的调用合入到它的正确调用序列中。当问题被发现时，当然已经错过了更改 A 的抽象体的时机。正如所料，所发生的就是把一个日益增长的复杂的“修正”集应用到 A 的内部设计上。在我们还没有安装完版本 1 时，就普遍感觉到设计正在衰退。每一个新的修正很可能都会破坏一些老的修正。这是一个正规的软件开发项目。最后，我和我的同事决定对设计进行更改，但是为了得到管理层的同意，我们不得不自愿无偿加班。

在任何一般规模的软件项目中，肯定会出现像这样的问题，尽管人们使用了各种方法来防止它的出现，但是仍然会忽视一些重要的细节。这就是工艺和工程之间的区别。如果经验可以把我们引向正确的方向，这就是工艺。如果经验只会把我们带入未知的领域，然后我们必须使用一开始所使用的方法并通过一个受控的改进过程把它变得更好，这就是工程。

我们来看一下只是作为其中很小一点的内容，所有的程序员都知道，在编码之后而不是之前编写软件设计文档会产生更加准确的文档。现在，原因是显而易见的。用代码来表现的最终设计是惟一一个在构建/测试循环期间被改进的东西。在这个循环期间，初始设计保持不变的可能性和模块的数量以及项目中程序员的数量成反比。它很快就会变得毫无价值。

在软件工程中，我们非常需要在各个层次都优秀的设计。我们特别需要优秀的顶层设计。初期的设计越好，详细设计就会越容易。设计者应该使用任何可以提供帮助的东西。结构图表、Booch 图、状态表、PDL 等等——如果它能够提供帮助，就去使用它。但是，我们必须记住，这些工具和表示法都不是软件设计。最后，我们必须创建真正的软件设计，并且是使用某种编程语言完成的。因此，当我们得出设计时，我们不应该害怕对它们进行编码。在必要时，我们必须应该乐于去改进它们。

至今，还没有任何设计表示法可以同时适用于顶层设计和详细设计。设计最终会表现为以某种编程语言编写的代码。这意味着在详细设计可以开始前，顶层设计表示法必须被转换成目标编程语言。这个转换步骤耗费时间并且会引入错误。程序员常常是对需求进行回顾并且重新进行顶层设计，然后根据它们的实际去进行编码，而不是从一个可能没有和所选择的编程语言完全映射的表示法进行转换。这同样也是软件开发的部分现实情况。

也许，如果让设计者本人来编写初始代码，而不是后来让其他人去转换语言无关的设计，就会更好一些。我们所需要的是一个适用于各个层次设计的统一表示法。换句话说，我们需要一种编程语言，它同样也适用于捕获高层的设计概念。C++正好可以满足这个要求。C++是一门适用于真实项目的编程语言，同时它也是一个非常具有表达力的软件设计语言。C++允许我们直接表达关于设计组件的高层信息。这样，就可以更容易地进行设计，并且以后可以更容易地改进设计。由于它具有更强大的类型检查机制，所以也有助于检测到设计中的错误。这就产生了一个更加健壮的设计，实际上也是一个更好的工程化设计。

最后，软件设计必须要用某种编程语言表现出来，然后通过一个构建/测试循环对其进行验证和改进。除此之外的任何其他主张都完全没有用。请考虑一下都有哪些软件开发工具和技术得以流行。结构化编程在它的时代被认为是创造性的技术。Pascal 使之变得流行，从而自己也变得流行。面向对象设计是新的流行技术，而 C++是它的核心。现在，请考虑一下那些没有成效的东西。CASE 工

具，流行吗？是的；通用吗？不是。结构图表怎么样？情况也一样。同样地，还有 Warner-Orr 图、Booch 图、对象图以及你能想起的一切。每一个都有自己的强项，以及惟一的一个根本弱点——它不是真正的软件设计。事实上，惟一一个可以被普遍认可的软件设计表示法是 PDL，而它看起来像什么呢？

这表明，在软件业的共同潜意识中本能地知道，编程技术，特别是实际开发所使用的编程语言的改进和软件行业中任何其他东西相比，具有压倒性的重要性。这还表明，程序员关心的是设计。当出现更加具有表达力的编程语言时，软件开发者就会使用它们。

同样，请考虑一下软件开发过程是如何变化的。从前，我们使用瀑布式过程。现在，我们谈论的是螺旋式开发和快速原型。虽然这种技术常常被认为可以“消除风险”以及“缩短产品的交付时间”，但是它们事实上也只是为了在软件的生命周期中更早地开始编码。这是好事。这使得构建/测试循环可以更早地开始对设计进行验证和改进。这同样也意味着，顶层软件设计者很有可能也会去进行详细设计。

正如上面所表明的，工程更多的是关于如何去实施过程的，而不是关于最终产品看起来像什么。处在软件行业中的我们，已经接近工程师的标准，但是我们需要一些认知上的改变。编程和构建/测试循环是工程软件过程的中心。我们需要以像这样的方式去管理它们。构建/测试循环的经济规律，再加上软件系统几乎可以表现任何东西的事实，就使得我们完全不可能找出一种通用的方法来验证软件设计。我们可以改善这个过程，但是我们不能脱离它。

最后一点：任何工程设计项目的目标是一些文档产品。显然，实际设计的文档是最重要的，但是它们并非惟一要产生的文档。最终，会期望某些人来使用软件。同样，系统很可能也需要后续的修改和增强。这意味着，和硬件项目一样，辅助文档对于软件项目具有同样的重要性。虽然暂时忽略了用户手册、安装指南以及其他一些和设计过程没有直接联系的文档，但是仍然有两个重要的需求需要使用辅助设计文档来解决。

辅助文档的第一个用途是从问题空间中捕获重要的信息，这些信息是不能直接在设计中使用的。软件设计需要创造一些软件概念来对问题空间中的概念进行建模。这个过程需要我们得出一个对问题空间中概念的理解。通常，这个理解中会包含一些最后不会被直接建模到软件空间中的信息，但是这些信息却仍然有助于设计者确定什么是本质概念以及如何最好地对它们建模。这些信息应该被记录在某处，以防以后要去更改模型。

对辅助文档的第二个重要需要是对设计的某些方面的内容进行记录，而这些方面的内容是难以直接从设计本身中提取的。它们既可以是高层方面的内容，也可以是低层方面内容。对于这些方面内容中的许多来说，图形是最好的描述方式。这就使得它们难以作为注释包含在代码中。这并不是说要用图形化的软件设计表示法代替编程语言。这和用一些文本描述来对硬件学科的图形化设计文档进行补充没有什么区别。

决不要忘记，是源代码决定了实际设计的真实样子，而不是辅助文档。在理想情况下，可以使用软件工具对源代码进行后期处理并产生出辅助文档。对于这一点，我们可能期望过高了。次一点的情况是，程序员（或者技术方面的编写者）可以使用一些工具从源代码中提取出一些特定的信息，然后可以把这些信息以其他一些方式文档化。毫无疑问，手工对这种文档保持更新是困难的。这是另外一个支持需要更具表达力的编程语言的理由。同样，这也是一个支持使这种辅助文档保持最小并且尽可能在项目晚期才使之变成正式的理由。同样，我们可以使用一些好的工具；不然的话，我们就得求助于铅笔、纸以及黑板。

总结如下：

- 实际的软件运行于计算机之中。它是存储在某种磁介质中的0和1的序列。它不是使用C++语言（或者其他任何编程语言）编写的程序。
- 程序清单是代表软件设计的文档。实际上把软件设计构建出来的是编译器和连接器。
- 构建实际软件的廉价程度是令人难以置信的，并且它始终随着计算机速度的加快而变得更加廉价。
- 设计实际软件的昂贵程度是令人难以置信的，之所以如此，是因为软件的复杂性是令人难以置信的，并且软件项目的几乎所有步骤都是设计过程的一部分。
- 编程是一种设计活动——好的软件设计过程认可这一点，并且在编码显得有意义时，就会毫不犹豫的去编码。
- 编码要比我们所认为的更频繁地显现出它的意义。通常，在代码中表现设计的过程会揭示出一些疏漏以及额外的设计需要。这发生的越早，设计就会越好。
- 因为软件构建起来非常廉价，所以正规的工程验证方法在实际的软件开发中没有多大用处。仅仅构建设计并测试它要比试图去证明它更简单、更廉价。
- 测试和调试是设计活动——对于软件来说，它们就相当于其他工程学科中的设计验证和改进过程。好的软件设计过程认可这一点，并且不会试图去减少这些步骤。
- 还有一些其他的设计活动——称它们为高层设计、模块设计、结构设计、构架设计或者诸如此类的东西。好的软件设计过程认可这一点，并且慎重地包含这些步骤。
- 所有的设计活动都是相互影响的。好的软件设计过程认可这一点，并且当不同的设计步骤显示出有必要时，它会允许设计改变，有时甚至是根本上的改变，
- 许多不同的软件设计表示法可能是有用的——它们可以作为辅助文档以及工具来帮助简化设计过程。它们不是软件设计。
- 软件开发仍然还是一门工艺，而不是一个工程学科。主要是因为缺乏验证和改善设计的关键过程中所需的严格性。
- 最后，软件开发的真正进步依赖于编程技术的进步，而这又意味着编程语言的进步。C++就是这样的一个进步。它已经取得了爆炸式的流行，因为它是一门直接支持更好的软件设计的主流编程语言。
- C++在正确的方向上迈出了一步，但是还需要更大的进步。

后　记

当我回顾几乎10年前所写的东西时，有几点让我印象深刻。第一点（也是和本书最有关的）是，现今，我甚至比那时更加确信我试图去阐述的要点在本质上的正确性。随后的一些年中，许多流行的软件开发方法增强了其中的许多观点，这支持了我的信念。最明显的（或许也是最不重要的）是面向对象编程语言的流行。现在，除了C++外，出现了许多其他的面向对象编程语言。另外，还

有一些面向对象设计表示法，比如：UML。我关于面向对象语言之所以得到流行是因为它们允许在代码中直接表现出更具表达力的设计的论点，现在看来有点过时了。

重构的概念——重新组织代码基础，使之更加健壮和可重用——同样也和我的关于设计的所有方面的内容都应该是灵活的并且在验证设计时允许改变的论点相似。重构只是提供了一个过程以及一组如何去改善已经被证实具有缺陷的设计的准则。

最后，文中有一个敏捷开发的总的概念。虽然极限编程是这些新方法中最知名的一个，但是它们都具有一个共同点：它们都承认源代码是软件开发工作中的最重要的产品。

另一方面，有一些观点——其中的一些我在论文中略微谈到过——在随后的一些年中，对我来说变得更加重要。第一个是构架，或者顶层设计的重要性。在论文中，我认为构架只是设计的一部分内容，并且在构建/测试循环对设计进行验证的过程中，构架需要保持可变。这在本质上是正确的，但是回想起来，我认为我的想法有点不成熟。虽然构建/测试循环可能揭示出构架中的问题，但是更多的问题是常常由于改变需求而表现出来的。 一般来说，设计软件是困难的，并且新的编程语言，比如：Java 或者 C++，以及图形化的表示法，比如：UML，对于不知道如何有效地使用它的人来说，都没有多大的帮助。此外，一旦一个项目基于一个构架构建了大量的代码，那么对该构架进行基础性的更改，常常相当于丢弃掉该项目并重新开始一个，这就意味着该项目没有出现过。即使项目和机构在根本上接受了重构的概念，但是他们通常仍然不愿意去做一些看起来就像是完全重写的事情。这意味着第一次就把它作对（或者至少是接近对）是重要的，并且项目变得越大，就越要如此。幸运的是，软件设计模式有助于解决这方面问题。

还有其他一些方面的内容，我认为需要更多地强调一下，其中之一就是辅助文档，尤其是构架方面的文档。虽然源代码就是设计，但是试图从源代码中得出构架，可能是一个令人畏惧的体验。在论文中，我希望能够出现一些软件工具来帮助软件开发者自动地维护来自源代码的辅助文档。我几乎已经放弃了这个希望。一个好的面向对象构架通常可以使用几幅图以及少许的十几页文本描述出来。不过，这些图（和文本）必须集中于设计中的关键类和关系。糟糕的是，对于软件设计工具可能会变得足够聪明，以至于可以从源代码的大量细节中提取出这些重要方面的内容这一点，我没有看到任何真正的希望。这意味着还得必须由人来编写和维护这种文档。我仍然认为，在源代码完成后，或者至少是在编写源代码的同时去编文档，要比在编写源代码之前去编写文档更好一些。

最后，我在论文的最后谈到了 C++是编程——并且因此是软件设计——艺术的一个进步，但是还需要更大的进步。就算我完全没有看到语言中出现任何真正的编程进步来挑战 C++的流行，那么在今天，我会认为这一点甚至要比我首次编写它时更加正确。

——Jack Reeves, 2002 年 1 月 1 日

索　引

说明：本索引按英文原书索引顺序列出，并给出释义、其在原书中的页码号及在本书中引用的章节号，以方便读者对照、参考。

英文	释义	原书页码	引用章号
	A		
A metric	抽象性度量	265	20
Abbott,Edwin A.	Edwin A. Abbott（人名）	177	16
Abstract	抽象	317	25
class	抽象类	264,483	20, 附录 A
Server	ABSTRACT SERVER 模式	317,318	25
abstraction	抽象	100,194,233,249, 278	9,18,19,22
metric *A*	抽象性度量值	282	22
underlying	潜在的抽象	201	18
abusing patterns	滥用模式	164	14
acceptance tests	验收测试	14	2
framework	验收测试框架	27	4
activation rectangle	激活矩形	485	附录 A
Active Object	主动对象	151,155,500,502	13, 附录 B
activity diagram	活动图	496,498	附录 B
actors	参与者	469	附录 A
Acyclic Visitor	ACYCLIC VISITOR 模式	107,391	9,28
Acyclic-Dependencies Principle	无环依赖原则	256,260	20
Adapter	ADAPTER 模式	138,317,319,322, 325,347,360,500	12, 25, 26, 27, 附录 B
Adaptive Software Development	自适应软件开发	19	3
ADP	自适应软件开发	19	3
afferent coupling metric;see C_a metric	输入耦合度量；参见 C_a 度量	262	20
aggregation relationship	聚合关系	480	附录 A
Agile Alliance	敏捷联盟	4	1
agile development	敏捷开发	87	7
analysis	分析		
use case	用例分析	194,201	18
analyst	分析师	12	2
anticipation	预测	105	9
architecture	构架	8,29,199,476	1, 4, 18, 附录 A
system	系统构架	278	22
association relationship	关联关系	474,480	附录 A
asynchronization bar	异步棒图	499	附录 B
axis of change	变化轴线	97	8
	B		
batch	批量	193	18
beauty	美	32	5
Beck,Kent	Kent Benk（人名）	vi,4	文前, 1
Beedle,Mike	Mike Beedle（人名）	4	1
Booch diagram	Booch 图	249	19
Booch,Grady	Grady Booch（人名）	444	30
bowling	保龄球	43	6
rules	保龄球规则	83	6
Bridge	BRIDGE 模式	317,322,325,362,365	25, 27
Brittingham,Pete	Pete Brittingham（人名）	448	30
Brooks,Matt	Matt Brooks（人名）	44,55,60,61,64	6
bubble sort	冒泡排序	165	14

英文	释义	原书页码	引用章号
Wirfs-Brock,Rebecca	Rebecca Wirfs-Brock（人名）	123,446	10, 30
Woolf,Bobby	Bobby Woolf（人名）	190	17
workspace,open	开放的工作空间	15	2
Wumpus	游戏中的一种怪兽	24	4
	X		
XML	可扩展标记语言	28	4
XUnit	单元测试系列工具的统称	348	26
	Y		
YAGNI	你将不会需要它	16	2