重构手册

Refactoring Workbook

[美] William C. Wake 编著
黄湘情 张 波 译

科学出版社

图字：01-2010-7769 号

内 容 简 介

本书采用实例手册的方式组织全书内容，帮助读者了解最重要的重构技术并将其应用于代码之中。作者精心组织了一系列问题，通过解决这些问题，让读者不仅在深层次上了解重构，而且会获得自己的一些心得体会。即使你的工作并非重构，本书也有助于你思考如何创建优质的代码。

本书面向有 Java 开发经验的程序员，但 C#和 C++程序员如果对 Java 有一定了解，也可以从本书获得较多受益。

著作权声明

Authorized translation from the English language edition, entitled Refactoring Workbook, 1E, 0321109295 by William C. Wake, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2004 by Pearson Education Inc.

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

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and SCIENCE PRESS Ltd. Copyright © 2010.

本书中文简体字版由培生教育出版公司授权科学出版社有限责任公司合作出版，未经出版者书面许可，不得以任何形式复制或抄袭本书的任何部分。

本书封面贴有 Pearson Education (培生教育出版集团) 激光防伪标签。无标签者不得销售。

图书在版编目（CIP）数据

重构手册/（美）威克（Wake, W.C.）著；黄湘情，张波译．—北京：科学出版社，2011.4

ISBN 978-7-03-030499-5

Ⅰ．①重… Ⅱ．①威… ②黄… ③张… Ⅲ．①代码—程序设计 Ⅳ．①TP311.11

中国版本图书馆 CIP 数据核字（2011）第 039386 号

责任编辑：何 武 刘秀青 / 责任校对：刘雪连
责任印刷：新世纪书局 / 封面设计：彭琳君

科学出版社 出版
北京东黄城根北街 16 号
邮政编码：100717
http://www.sciencep.com

中国科学出版集团新世纪书局策划
三河市少明装订厂印刷
中国科学出版集团新世纪书局发行 各地新华书店经销

*

2011 年 6 月 第 一 版 开本：16 开
2011 年 6 月第一次印刷 印张：13
印数：1—2 500 字数：240 000

定价：42.00 元

（如有印装质量问题，我社负责调换）

前 言 Preface

什么是重构

重构是改善现有代码设计的一门艺术。重构不仅提供了识别问题代码的方法，而且给出了改善此类代码的秘方。

本书目标

本书是一本实战手册，你可以利用本书：

- 尝试找出最重要的“异味（smell）”（即问题）。
- 应用最重要的重构技术。
- 充分考虑如何编写优质代码。
- 尽享重构带来的乐趣。

具体来讲，本书是一本参考指南，提供了以下内容：

- 便于使用的“异味查找器”（smell finder）。
- 用一种标准格式来描述异味。
- 通过附录列出了支持重构的Java工具。
- 通过附录介绍了关键重构技术。

本书面向的读者

重构是一种面向代码的技术，因而这本书是特别为致力于编写和维护代码的程序员所著。

学生们也能够从重构中获益，不过我认为，只有在参与了一个中等规模或更大规模的程序开发，或者作为开发小组中的成员经过一番实战之后，才能体会重构的价值（无论是低年级学生、高年级学生还是研究生，都是如此）。

所需背景知识

如果有Martin Fowler等人所著的《重构：改善既有代码的设计》一书（以下简称《重构》），或者能访问www.Kfactoring.com网站，则可以参考其重构目录，这将对你大有裨益（你可以同时阅读本书和《重构》一书）。在《重构》中，Martin和他的合作伙伴们对多种重构技术进行了循序渐进的讲解，而

前 言 Preface

在本书中，我并不打算做简单的重复。此外，Martin等人还提供了精心设计的实例，并附上了大量精辟的论述和背景材料。有人可能会决定先阅读本书，这也未尝不可，但我还是建议你先看看《重构》。

本书中的示例均使用Java编写，这并不是因为Java是最易于完成重构的语言，而是因为这种语言相当流行，而且好的Java开发环境可以提供自动化重构支持。如果你是对Java有一定了解的C#或C++程序员，那么本书中的大多数问题对你来说也不难理解。不过，在本书后面部分，需要读者动手修改、测试和运行更大规模的程序，这对于使用非Java语言进行开发的程序员来说可能会有一定的困难。

Gamma等人所著的《设计模式》一书将模式描述为“重构的目标”。如果你对此比较熟悉，将有助于你学习本书，因为我会反复提到《设计模式》中所谈到的多种模式。如果你不了解《设计模式》，那么我再向你推荐一本书，即Steve Metsker编写的*Design Patterns Java Workbook*。

如何使用本书

相对来说，实际解决问题的难度要比仅找出解决方案要更大。本书附录提供了部分问题的答案，但是如果你能在查阅答案之前，先尝试自己解决问题的话，将会有更大收获。如果你确实自行解决了这些问题，甚至对我给出的某些答案持有异议，这对你我来说都不失为一大乐事。只是查看答案并一味地接受，真是没有多大乐趣可言。

如果你能够与别人合作（或者参与一个小组）将会更有意思，不过我发现这一愿望往往并不能实现。

后面（更长）的示例需要在一台计算机上完成。在不同计算机环境下阅读程序时，查找问题并确定解决问题的方式可能与本书有所不同。

联系作者

我的邮箱是William.Wake@acm.org。

我还有一个网站：www.xp123.com，这个网站重点讨论极限编程（Extreme Programming，XP），而重构是其中不可或缺的重要组成部分。在这个网站上，重构（以及本书）有专门的介绍页面：www.xp123.com/rwb。

我很想知道你做这些练习时的体会，以及你对重构的一般认识，所以请尽可能对我畅所欲言，给我来信。

致谢

写书过程中面临的一个难题是，很难一一谢过这么多曾帮助过我的人。我可能无法记住每一个伸出过援助之手的人的名字，而且必须承认的是，如果我能够采纳所有人的建议，这本书可能会比现在更加出色。

对于本书中的各种设计方案和练习，许多人都给出了建议，有Philippe Antras，Ron Crocker，Sven Gorts，Harris Kirk，Tom Kubit，Paul Michali（他还提供了一个示例），Edmund Schweppe，Steve Wake，Robert Wenner，等等，恕不在此一一列出。特别是Sven Gorts和Tom Kubit为我提供了非常详尽的反馈意见和建议，Ann Anderson，Ken Auer和Don Wells审阅了本书书稿。

我在Gene Codes Corporation（效力于Howard Cash）公司的程序员朋友们总是力图给我提供示例素材（本书中并没有他们编写的代码，不过在我考虑要表述哪些内容时，他们都曾启发过我，而且每个人都至少对一些示例提出了建议）。在此要感谢Lucy Hadden，Jonathan Hoyle，Anna Khizhnyak，Tom Kubit，Greg Poth和Dave Relyea。

这本书显然还要归功于Martin Fowler和Kent Beck先前所做的工作。他们的鼓励对我来说同样很重要。这本书的编写方式受到了Steve Metsker所著的*Design Patterns Java Workbook*一书的启发，与Steve Metsker的交谈也使我获益匪浅。

对于Addison-Wesley/Prentice Hall公司，我要向Mike Hendrickson，Ross Venables，Anne Garcia和Michelle Vincenti致以感谢，特别要感谢我的责任编辑Paul Petrali，以及BooksCraft的Don MacLaren和Ruth Frick的文字润色工作。这本书能够从一份书稿变成一本真正的书倾注了太多人的心血，可能还有些人我并不知道名字，对于他们的辛勤工作，我在此表示最诚挚的谢意。

前 言 Preface

Pearson公司允许我采用了如下一些书中的信息：

- Beck，*EXTREME PROGRAMMING EXPLAINED:EMBRACING CHANGE*，第57页，©2000 Kent Beck版权所有。经Pearson Education 出版公司许可，由Pearson Addison Wesley出版。
- Fowler，*REFACTORING: IMPROVING THE DESIGN OF EXISTING CODE*，封二（重构列表）；封三（异味和常用重构手法），©1999 Addison Wesley Longman公司版权所有。经Pearson Education 出版公司许可，由Pearson Addison Wesley出版。
- Gamma/Helm/Johnson/Vlissides，*DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE*，第viii和ix页（目录中的模式列表）。©1995 Addison Wesley出版公司版权所有。经Pearson Education出版公司许可，由Pearson Addison Wesley出版。
- Wake，*EXTREME PROGRAMMING EXPLORED*，第24～25页，©2002 Pearson Education公司版权所有。经Pearson Education出版公司许可，由Pearson Addison Wesley出版。

最后，我要感谢我至爱的家人，你们不仅容忍我埋头写作，还不断地给予我鼓励、支持和爱，这是我最大的精神动力。

目 录

第1章 本书导读 …… 1

1.1 概述 …… 1
1.2 第1部分：类中的异味 …… 2
1.3 第2部分：类之间的异味 …… 2
1.4 第3部分：待重构的程序 …… 2
1.5 关于练习 …… 3

第1部分 类中的异味

第2章 重构周期 …… 5

2.1 什么是重构 …… 5
2.2 异味就是问题 …… 6
2.3 重构周期 …… 7
2.4 怎样算完成 …… 8
2.5 重构内部 …… 10
2.6 实战练习 …… 13
2.7 小结 …… 13

第3章 可度量的异味 …… 14

3.1 所涉及的异味 …… 14
3.2 注释 …… 15
3.3 过长的方法 …… 17
3.4 过大的类 …… 22
3.5 过长的参数表 …… 26
3.6 实战练习 …… 28
3.7 小结 …… 29

补充点1 异味和重构 …… 30

第4章 命名 …… 36

4.1 所涉及的异味 …… 37
4.2 名字（包括匈牙利记法）中包含类型 …… 37
4.3 表达力差的名字 …… 38
4.4 不一致的名字 …… 40

第5章 不必要的复杂性 …… 42

5.1 所涉及的异味 …… 42

目录

5.2 死代码……42
5.3 过分一般性……43
补充点2 逆处理……46
第6章 重复……48
6.1 所涉及的异味……49
6.2 魔法数……49
6.3 重复性代码……50
6.4 具有不同接口的相似类……51
6.5 实战练习……52
第7章 条件逻辑……58
7.1 所涉及的异味……58
7.2 Null检查……58
7.3 复杂的布尔表达式……60
7.4 特殊用例……61
7.5 模拟继承（Switch语句）……62
补充点3 设计模式……65

第2部分 类之间的异味

第8章 数据……68
8.1 所涉及的异味……68
8.2 基本类型困扰……68
8.3 数据类……72
8.4 数据泥团……76
8.5 临时字段……77
第9章 继承……78
9.1 所涉及的异味……78
9.2 拒收的遗赠……78
9.3 不当的紧密性（子类形式）……80
9.4 懒惰类……81
第10章 职责……83
10.1 所涉及的异味……83
10.2 依恋情结……83

10.3 不当的紧密性（一般形式）……85
10.4 消息链……86
10.5 中间人……87
10.6 实战练习……88
第11章 相关改变……91
11.1 所涉及的异味……91
11.2 发散式改变……91
11.3 霰弹式修改……94
11.4 并行继承体系……96
11.5 组合爆炸……97
第12章 库类……99
12.1 所涉及的异味……99
12.2 不完备的库类……99
12.3 实战练习……100
补充点4 重构构成形式……103

第3部分 待重构的程序

第13章 一个数据库例子……105
13.1 Course.java……106
13.2 Offering.java……108
13.3 Schedule.java……110
13.4 Report.java……114
13.5 TestSchedule.java……115
13.6 TestReport.java……118
第14章 一个简单的游戏……122
开发环节……128
第15章 名录……130
15.1 引言……130
15.2 第1种做法：Catalog.itemsMatching(query)……131
15.3 第2种做法： Query.matchesIn(catalog)……133
15.4 第3种做法：Process(catalog.data，
query.data)……134

目录

15.5 小结……135
第16章 计划游戏模拟器……136
16.1 第1部分：原始代码……137
代码……137
实战练习……145
16.2 第2部分：重新分配特性……146
5个为什么……148
16.3 去除重复、选择问题和一些模糊性……149
16.4 第3部分：进一步改进代码……152
第17章 下一步何去何从……155
17.1 参考书……155
17.2 警告……155
将重构应用到实践中……155
将测试应用到实践中……155
求助他人……156
17.3 必做练习……156
每周清除一个异味……156
重新重构……156
只是重构……156
吸气/呼气（Inhale/Exhale）……156
反向重构/误构（Defactoring/Malfactoring）……156
重构套路……157
17.4 网站资源……157

第4部分 附录

附录A 所选问题的答案……159
附录B Java重构工具……188
附录C 重构逆处理……189
附录D 主要重构技术……191
参考文献……194

程序员必备的一项基本技能是，能够找出问题代码并改善它，使程序在软件的整个生命周期正常运行。

重构是一门艺术，通过安全地改善现有代码设计来提供高效可靠的系统，使纷繁杂乱归于平稳有序，并最大限度地避免异常！

多数人认为，学习重构的最大障碍是正确找出代码中的“异味（smell）”，即可能存在的问题。本书不是简单地展示这些异味是什么，而是引导读者深入理解这些异味。本书引入了一系列精心组织的问题，通过解决这些问题，读者不仅会在更深层次上了解重构，还会获得自己的心得体会。

Wake采用实战训练的方法组织全书内容，你将通过这种方法学习最重要的重构技术并将其应用于代码。即使你当前所做的工作并非重构，本书也能激发你思考如何创建优质的代码。

本书是一本方便的参考手册，你可以从本书找到：

- 方便快捷的“异味查找器”
- 描述各种异味的标准格式
- 介绍关键重构技术的附录
- 支持重构的Java工具列表

本书针对有Java开发经验的程序员，不过，对Java有基本了解的C#或C++程序员也能顺利阅读本书并学习书中的示例。本书可以作为Martin Fowler的名著《重构：改善既有代码的设计》之配套读物。

作者简介：

William C. Wake 是一位资深的独立软件顾问，也是一名教员和培训师。他致力于程序设计研究逾20年，曾任职于Capital One Financial、MCI/WorldCom和VTLS等世界知名大公司。编著图书 *Extreme Programming Explored*，由Addison-Wesley Professional出版。

出版服务信息

www.ncpress.com.cn

策　　划：中国科学出版集团新世纪书局
责任编辑：何　武　刘秀青
封面设计：彭琳君
技术支持：book@ncpress.com.cn
在线服务：www.ncpress.com.cn
直销电话：010 - 64869353

上架建议 计算机/程序设计/软件工程

ISBN 978-7-03-030499-5

定价：42.00元

第 1 章

本书导读

1.1 概述

本书分为 3 大部分。第 1 部分关注的是类中出现的异味（smell，即问题）。第 2 部分强调类之间的异味。第 3 部分则提供了一些大程序，用于在不同领域实践重构。这些部分之间不时会穿插一些简要说明，或称其为“补充点”，它们是对《重构：改善既有代码的设计》（Martin Fowler 等著，后面简称为 Fowler 的《重构》）一书中重构名录分析的简要介绍，或者是对《设计模式》（Erich Gamma 等所著。同样，后面简称为 Gamma 的《设计模式》）一书中模式的分析说明。

在前两部分中，相关章节主要由异味（暗示潜在问题的一些信号）和实战练习组成。对于异味的描述，我使用了一种标准格式，如下：

异味（Smell）——异味的名字。
症状（Symptoms）——有助于找出问题的线索。
原因（Causes）——说明问题会如何发生。
采取的措施（What to do）——可能的重构。
收益（Payoff）——代码将在哪些方面有所改善。
不适用的情况（Contraindications）——不适合进行重构的情况。

通过这种格式，能更好地将这些有关异味的知识加以应用。即使在完成实战练习后，你也能够参考这些内容，并从中获益。

这里的实战练习形式各异，有些要求你分析代码，有些要求对某种状况

进行评估，还有一些则需要你修改代码。对于基于代码的练习，其中的代码均可在 www.xpl23.com/rwb 网站上找到。

本书中的练习难度各异。对于难度大的练习，我在题目之后标有“有难度”字样，而且你会发现，这些练习的答案也很灵活，可能存在多种答案。

附录 A 给出了大部分练习的解决方案（或有助于找出解决方案的思路）。对于后面的（基于代码的）练习，我不打算提供答案，因为它们是让你修改程序。

1.2 第 1 部分：类中的异味

第 2 章将简要介绍重构周期（refactoring cycle）。第 3 章讨论可度量的异味，它们可以通过简单的长度来度量。在第 4 章中，我们将了解到名字在代码简单性和可理解性方面的意义。第 5 章将探讨代码多余问题。

第 6 章的主题是重复，从许多方面来说，这是需要特别关注的一种核心异味。其他一些异味均可看做是重复的特例。

第 1 部分在第 7 章结束。第 7 章将讨论条件逻辑，即如何使条件和循环语句使用的表达式更清晰明了。

1.3 第 2 部分：类之间的异味

类中的数据有时表示的是丢失的对象，第 8 章将会讨论这个问题。

第 9 章将讨论如何权衡超类和子类的责任。第 10 章将进一步讨论这个问题，且会研究其他类之间的责任平衡问题。当确定如何最佳地连接对象时，有时必须对第 10 章讨论的异味进行权衡。

有时在进行某些修改时，最能体现出重复性问题，我们将在第 11 章介绍这一点。第 12 章作为第 2 部分的结束，讨论了使用库类时存在的一些问题。

1.4 第 3 部分：待重构的程序

本书最后一部分给出了一些有待重构的程序。

- 第 13 章是一个简单的选课系统，它使用了一个数据库。将代码和数据库一起重构是一个正在兴起的研究领域。在该程序的代码中，你可以看到存在大量重复问题需要修正。
- 第 14 章将介绍一个简单的游戏程序。小程序中也不乏需要做决策的地方，这就为大量异味带来了可乘之机。本章还将涉及测试驱动开发。
- 第 15 章所考虑的是权衡责任时面对的一些挑战。我们将使用重构来研究三种访问在线名录的方法。即使只用 3～4 个类，也能得到多种彼此迥异的解决方案。
- 第 16 章涉及一个图形用户界面（Graphical User Interface，GUI）。这个例子展示了一个常见的问题：你希望完成单元测试，以便能安全地进行重构，但是你又必须先进行重构，以使代码可测试。

最后在第 17 章提供了日后你可以在自己的工作中应用的一些练习。另外，这一章还提供了一些参考资源。

1.5 关于练习

要完成这些练习，有一种方法很快捷，即阅读问题，再直接查看答案，因为答案看上去可行，所以你点头称是，如此而已。但这会使你被我的想法所左右。

相应地，还有一种稍微复杂的做法，但却是一种更好地完成练习的方式，即阅读问题，再解决问题，然后才看答案。而且这样不会让你的思维被别人左右。

需要特别指出的是，对于要求你修改的代码，只有亲自动手实践才能有更多的收获。重构是一种需要反复实践的技能。

祝你好运！

第 1 部分

类中的异味

第2章 重构周期
第3章 可度量的异味
补充点1 异味和重构
第4章 命名
第5章 不必要的复杂性
补充点2 逆处理
第6章 重复
第7章 条件逻辑
补充点3 设计模式

第 2 章

重构周期

本章我们将从两个方面来介绍重构：

1. 识别与修正问题的周期。
2. 在各种重构中所采用的处理类型。

2.1 什么是重构

重构是安全地改善现有代码设计的一门艺术，包括如下含义：

- *重构并不包括对系统的所有修改*。如果修改是对设计的改善，或是增加新功能，则均不能被认为是重构。在创建新代码的过程中，尽管重构可以作为其中的一个环节，但它并不承担增加新特性的任务。例如，极限编程（Extreme Programming 或 XP）（详见 Kent Beck 所著的 *Extreme Programming Explained: Embrace Change*）使用了测试驱动开发，这包括先编写一个测试，然后编写新代码来引入新的特性，最后再通过重构来改善设计。
- *重构并非从头开始重新编写*。尽管有时重新开始更合适，但是重构可以改变平衡点，从而有可能改善代码，而无需承担重新编写的风险。Sven Gorts 与我私下交谈时曾指出：重构能够保护现有代码中所含的信息（知识）不“流失”。
- *重构并不仅仅是某种改善代码的结构更改*。区别重构（refactoring）和更改结构（restructuring）的关键在于，重构力图达到一种安全的（safe）转换。即使是需要修改大量代码的大型（big）重构，也可以将它分解

为较小规模的安全重构（在最佳情况下，如果重构定义得当，还可以做到自动化重构）。如果所做的修改使得代码在超出一个工作期时就无法正常工作或运行（即不能通过测试），那么就不能认为这种修改是重构。

- *重构会改变预先设计（up-front design）和紧急设计（emergent design）之间的平衡点*。预先设计是在实现之前完成的设计，紧急设计是在实现之中进行的设计。如何对预先设计和紧急设计加以权衡，关键在于对于代码中可能存在的问题，我们能够做出何等程度的预测（anticipate）或估计（assess）。另外，以下两种做法哪种更简单也对权衡有着影响：即是先进行设计，再将设计“翻译”为代码，还是先编写代码，再对代码加以改善。重构可以降低紧急设计方法的开销和风险。（改善的程度究竟如何，人们对此可能会有不同的看法，但是重构确实会对此有所改善，相信这一点所有人都认同。）
- *重构规模可大可小*。许多重构规模都很小。一般说来，小型重构应用于较“完美”的情况，此时很少需要进行大型重构。即使应用大型重构，对所采用的方法也存在要求，即不能在进行重构的很长时间内（如长达 6 个月）没有任何新特性的增加，而是应当在进行重构的同时保证系统在任何时候都能正常运行。

2.2 异味就是问题

异味（smell），特别是代码异味（code smell），是一些指出代码中潜在问题的警示信号。并非所有异味都表示确实存在的问题，但是对于大多数异味，都有必要进行查看并做出相应决策。

有些人不喜欢用“异味”一词，而更倾向于把它们称作潜在问题（potential problem）或缺陷（flaw），不过，我认为“异味”这种说法很贴切。想象一下，如果冰箱里面有些东西发霉变质，而你正打开这个冰箱，那将是一种什么情况。有些东西的味道可能已经很重了（说明这些东西彻底坏了），显然我们知道该如何处理它。还有一些味道可能并不大，尚不能确定问题是出在原来剩下的青豆上，还是烧鸡有问题。冰箱里可能还有一些东西也已经坏了，但是并没有发出任何臭味。代码异味与此可谓异曲同工：有些很明显，有些

则并不明显，还有一些可能会掩盖其他问题。因此在修正某些问题时，可能会出乎意料地暴露出另外一些问题。

异味所描述的往往是局部性问题。如果人们能够很轻松地找出整个系统中的所有问题，那该多好！可惜，在这方面我们并不擅长，我们通常只会考虑当前所查看的部分，而局部异味正好发挥作用。

2.3 重构周期

重构有以下基本模式：

重构周期（Refactoring Cycle）

从一个正在运行的程序开始。

若存在异味，则：

- 挑出最坏的味道。
- 选择一种可以解决该异味的重构。
- 应用此重构。

在每次重构时，我们都力图选择适当的重构来改善代码。由于这些步骤并不会改变程序的可察行为（observable behavior），所以程序仍能正常运行。因此，重构周期可以改善代码，而同时保持行为不受影响。整个过程中最具难度的当属如何找出异味，而本书之所以用大量篇幅来强调这一主题，也正是因为如此。

采用这种重构的方法一定能够得到一个问题的理想设计吗？遗憾的是，答案是否定的，因为仅仅通过查看局部特性并不能保证获得最大程度的全局认识。但是，当代码变得尽可能简洁后，就可以较容易地得到调整解决方案的设计思路了。

重构就像是过河。要过河，一种方法是通过助跑猛地一跳，然后闭着眼睛祈祷能一跃而过。而重构的方法则像在河中找一些石头，然后一次跳到一个石头上，分步过河。

开始重构时，最好从比较简单的问题开始（例如，分解大型例程，或者为了清晰起见进行重命名）。你会发现，这样能使你更轻松地发现并修正余下的问题。

2.4 怎样算完成

一种方法是寻求最简单的设计。Kent Beck 确定了简单设计的 4 条规则，如果代码违背了这些规则（这些规则存在不同的优先级），就需要对相应的问题加以解决。

简单设计

1. 运行所有测试。
2. 不存在重复的逻辑。要特别注意诸如并行类继承体系结构（parallel class hierarchies）之类的隐藏重复。
3. 要向程序员清楚地说明对其很重要的每一个意图。
4. 尽量少使用类和方法。

——Beck，*Extreme Programming Explained* 第 57 页

对于这些规则，用一个缩写来表示就是 OAOO，即一次且仅一次（once and only once）。代码必须将某些内容表述一次，以便通过测试，并反映出程序员的理解和意图；而且要求只表述一次，也就是说，不应当存在任何重复。

如果一直都不重视代码的简洁性，则要想把一团乱麻似的代码整理清晰谈何容易！但为了力求完美而花费数月来清理原有代码，又很少有小组能够做到这一点。所以我们必须学会在开发进程中建立质量更高的代码，而且每次尝试一个新领域时都能够获得一些经验和教训。

极限编程（XP）和重构

极限编程是一种敏捷（agile）软件开发方法，其特征可以概括为一种特殊的规划风格、一个开发小组全程关注，以及通过测试和重构进行程序设计。重构由来已久，有一群人（包括 Kent Beck、Ward Cunningham、Martin Fowler 和 Ralph Johnson）一直都在致力于面向对象开发、模式、重构等领域及其他相关思想的研究。

采用 XP 思想进行程序设计主要围绕三点展开：设计简单、测试优先的编程和重构。测试优先编程（test-first programming）可以描述为“编写一个测试，看看它成功与否，再编写代码，查看测试是否通过。”将这些结合起来即构成了目前所谓的测试驱动开发（test-driven development）。有

关此技术的深入介绍，请参见 Dave Astels 所著的 *Test-Driven Development: A Practical Guide* 和 Kent Beck 所著的 *Test-Driven Development: By Example*。

我非常推崇测试驱动方法（因而我也很沉迷于 XP），但是重构本身对此却并没有要求。不过，用这种方法创建的代码往往错误较少，而且很少需要一般编码所需的大型重构。特别是对于本书中后半部分所涉及的较大规模的例子，如果是使用测试驱动开发方法完成的，则异味不仅更小，而且也更少。

重构的环境

一个文本编辑程序就可以完成重构。不过，如果有一个支持的环境，可以更容易、更安全地实现重构。

小组或合作人员：有关代码的重大决策，如果能有多人参与讨论，将会很有帮助。相对于单枪匹马，团队合作往往能够催生出更好的想法。不同的人有不同的经验，而且对系统的不同部分也会有不同的体会。

测试：即使重构设计本身是安全的，你也很有可能在真正应用时犯错误。建立一个测试套件（test suite），在重构之前和之后运行它，这样有助于确保所改变的是代码的设计而非其效果。

如果没有测试呢？那就添加吧，至少要在受重构影响的地方增加相关测试。有时这可能比较难——也许在不改动设计的情况下很难有效地进行测试，但是如果未经测试就改变设计又会很不安全。如果某些领域很难测试，则说明设计中存在其他一些问题。

“如果要进行重构，最基本的前提条件就是具备可靠的测试。”

——Martin Fowler，《重构》第 89 页

测试框架：JUnit 测试框架（www.junit.org）已经成为一个标准，后面的例子会用到这个测试框架。测试类扩展了 TestCase 类，其中包含一些名称以 test 开头的方法。此框架提供了一些断言（assert）方法，用来验证代码的行为。它还提供了一个测试运行工具（test runner），可以用它运行一组测试。

CRC 插卡或 UML 框图：重构并不是完全颠覆设计。有时可以不进行重构，而是采用一种“类-责任-协作者（Class-Responsibility-Collaborator，

CRC）”的插卡方式。画 UML（Unified Modeling Language，统一建模语言）框图也是可选的方法。

配置管理/版本控制：范围包括从撤销（Undo）到完备的配置管理系统等。如果你在进行重构时犯了一个错误，那么往往希望能够退回到前一个已知的最佳点（known good Point）。另外，你可能会应用一个重构，但是不能保证结果会有所改进。此时最好是先进行尝试，然后再确定是否保留结果。

完备的集成开发环境（Integrated Development Environment，IDE）：早在多年以前，一些简单但又很强大的语言（如 Smalltalk 和 Lisp）就已经在其环境中提供了对重构的支持，Java 也增加了这种支持。另外，在 C#中对重构的支持也开始有所体现。用户现在还是要先确定应用何种重构，不过利用现成的工具可以使出错几率大大降低。

2.5 重构内部

重视安全转换（safe transformation）是重构的一个重要方面。下面将从头开始完成一个简单的重构。在此过程中，我们将得出一些基本原则，这些原则有助于更好地理解重构的工作原理。

考虑封装字段（Encapsulate Field）这个重构。其目的是让对象的客户使用方法来访问字段，而不是直接访问字段：

```
public String name;
→
private String name;
public String getName() {return name;}
public void setName(String newName) {name = newName;}
```

Martin Fowler 在他的重构名录中指出：

1. 为字段创建获取方法（getter）和设置方法（setter）。
2. 找出对字段的所有引用。将访问替换为调用获取方法，把修改字段替换为调用设置方法。
3. 在修改各个引用后进行编译和测试。
4. 将字段声明为私有。
5. 编译并测试。

重构是一个循序渐进的过程，每一步所迈出的“步伐”比人们预期的要小。大多数重构应用都力图在 1 分钟到 1 小时内完成，平均时间约为 5—10 分钟。因此，如果重构需要几分钟完成，则说明步骤分得非常细了。

重构是采用每次迈出一小步的方式进行的。

初始代码如下：

```
public class Person {public String name;}
```

测试客户端如下所示：

```
Person person;
person.name = "Bob Smith";
assertEquals("Bob Smith", person.name);
```

第 1 步：创建获取方法和设置方法。

```
public class Person {
   public String name;
   public String getName() {return name;}
   public void setName(String newName) {name = newName;}
}
```

注意，客户端没有变化：还不存在调用这些新方法的代码。

第 2 步：查找所有客户端；将引用替换为调用。一次完成一个替换。

要找到这些引用，一种方法是临时性地将该字段设为私有。如果采用这种做法，一定要在修改客户端之前先将它改回公共作用域，以免打断任何客户端。

在许多重构中，编译器会指出是否一切正常。例如，假设你正在使用隐藏方法（Hide Method）将一个公共方法设为私有。当修改可见性关键字并进行重新编译时，你将得到一个警告，告知有些客户仍在使用该方法。由于这是一个编译时出现的警告而不是一个运行时错误，因此重构时就能够有所防范。

充分利用编译器提供的信息。

将赋值

```
person.name = "Bob Smith";
```

变为

```
person.setName("Bob Smith");
```

第 3 步：编译并测试。

重构的最终目标是通过重构得到一个有所改善的系统，不过在许多重构过程中都存在基本点（base）、安全点（safe point）或中断点（interruption point）。可以把基本点想象成棒球赛或打点游戏中的底线；这往往并不是最终目标，但是一旦你身处底线，至少说明你不会被击中。由于在重构中会涉及两种截然不同的方法，所以系统可能还不像最后那样清晰，但是我们可以停下来，运行测试，并确保迄今为止一切正常。

大多数重构都存在内置的基本点。

系统处于一种不安全的状态时，我们会屏住呼吸；当测试运行正确时，我们会长舒一口气。这种一张一弛的感觉很不错。如果在接近胜利之前总是觉得自己心神不定，而且在最终结束之前才发现一切都错了，那是比较糟糕的。相对来讲，我们更倾向于张弛有道。

大型重构也采用了这种基本点的思想。而且在进行大规模重构时，基本点的作用更加明显。也许你需要花费数月来消除某些决策遗留的影响，那么在这个过程中必须存在一些安全点。

第 2 步：替换其他引用。

将如下引用：

```
assertEquals("Bob Smith", person.name);
```

变为：

```
assertEquals("Bob Smith", person.getName());
```

第 3 步：编译并测试。

如果确实需要，我们可以在完成编译和测试后就停止，这也是安全的。不过，从某种意义上说，与最初的代码相比，这样得到的代码更糟糕：它使用了两种不一致的方法来访问字段。不过，这也向更好方法迈进了一步。

第 4 步：修改完所有客户端后将字段变为私有。

```
public class Person {
   private String name;
   public String getName() {return name;}
   public void setName(String newName) {name = newName;}
}
```

第 5 步：最后一次编译并测试。

我们再次回到了安全状态，而且字段完全被封装了。

2.6 实战练习

习题1　小步前行。

挑选一种重构技术，从中找到一处能应用小步设计但也适合应用较大幅的步骤设计的地方。

有关解决方案请见附录 A。

习题2　简单设计。

A. 验证 Beck 对简单设计所提出的规则。

B. 这些规则为什么存在不同的优先级？你能否找出一个例子，其中清晰的表述要比避免重复更重要？

有关解决方案请见附录 A。

2.7 小结

无论是使用现有重构还是开发新的重构技术，都应当以一种适当的方式进行，即能够使系统从一个正常状态稳步前进到下一个正常状态。在进行重构时，我们倾向于步骤较小但更安全（small steps but safer）的方法，而不是快速但往往不安全（fast but not always safe）的方法。请牢记重构周期。

第 3 章

可度量的异味

这一章中的异味基本相似。要检测出这些异味很容易，这些异味都具有客观性（只要确定了计量方法和最大承受限度，即可确定相应的异味）。这些异味都比较讨厌。

此外，这些异味还相当常见。

你可以把这些异味想象成由某种软件度量方法（metric）捕获的问题。代码为什么不像预期的那样“优秀”呢？对此，每种度量方法都试图找出导致这种情况的不同方面。有些度量方法考虑代码的长度变化，另一些则度量方法或对象之间的关联，还有一些测量离最佳状态还有多远。

大部分度量方法都与长度相关，所以我打算先从规模（size）开始讨论，这些异味通常称为过大的类（Large Class）或过长的方法（Long Method）。如果有很容易计算的度量方法，我将把它用作指示器，指示某些代码段需要被进一步查看和分析。

度量方法只是指示器（indicator），即它只表示异味可能存在，而不是说异味一定存在。如果不解决整体的复杂性问题，就很容易落入“造数”（making）的泥潭。因此，不要仅仅为了得到一个更好的数字而进行重构，应确保重构确实能够改善代码。

3.1 所涉及的异味

- 注释（Comments）。
- 过长的方法（Long Method）。
- 过大的类（Large Class）。

- 过长的参数表（Long Parameter List）。

3.2 注释

症状

- 代码中出现注释符号（//或/*）。（有些 IDE 通过为不同类型的注释代码设置不同的颜色来进行有效的区分。）

原因

出现注释最可能的原因是作者认为某些内容没有说清楚，需要增加注释。有些注释特别有用：

- 指出为什么要以这种特定的方式（或者为什么不以某种方式）完成某个工作。
- 引用了并非显而易见的算法（此时比较简单的算法不适用）。

其他注释还可以在代码自身中体现。例如，一个例程的目标不仅可以通过注释表达出来，还可以通过该例程的名字来体现。

采取的措施

- 如果一条注释是对一个代码块的解释，一般可以使用抽取方法（Extract Method）将这个代码块取出，然后将它置于一个单独的方法中。这个新方法的名称往往就存在于注释中。
- 如果一条注释解释了一个方法所做的工作（比方法名提供的信息更详细），则可以使用重命名方法（Rename Method），根据该注释来指定新的方法名。
- 如果一条注释解释的是前提条件，则可以考虑使用引入断言（Introduce Assertion），将注释替换为代码。

收益

可以增强表述能力，有可能暴露出重复性。

不适用的情况

不要将尽职尽责的注释一并删除。

习题 3 注释。

考虑下面这段代码（可在 www.xp123.com/rwb 在线获得）。

Matcher.java

```
public class Matcher {
    public Matcher() {}
    public boolean match(int[] expected, int[] actual,
        int clipLimit, int delta)
    {

        // 找出“过大”的值
        for (int i = 0; i < actual.length; i++)
            if (actual[i] > clipLimit)
                actual[i] = clipLimit;

        // 查看长度差异
       if (actual.length != expected.length)
            return false;

        // 检查 expected +/- delta 范围内的每一项
        for (int i = 0; i < actual.length; i++)
            if (Math.abs(expected[i] - actual[i]) > delta)
                return false;

        return true;
    }
}
```

MatcherTest.java

```
import junit.framework.TestCase;

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

    public void testMatch() {
        Matcher matcher = new Matcher();

        int[] expected = new int[] {10, 50, 30, 98};
        int clipLimit = 100;
        int delta = 5;

        int[] actual = new int[] {12, 55, 25, 110};
```

```
        assertTrue(matcher.match(expected, actual,
                                 clipLimit, delta));

        actual = new int[] {10, 60, 30, 98};
        assertTrue(!matcher.match(expected, actual,
                                  clipLimit, delta));
        actual = new int[] {10, 50, 30};
        assertTrue(!matcher.match(expected, actual,
                                  clipLimit, delta));
        }
}
```

A. 使用抽取方法，令 match()中的注释成为多余的。

B. 有关代码的所有重要信息仅使用代码就可表达清楚吗？是否有必要使用注释？

C. 找出你最近编写的一些代码，最好是加了注释的代码。看看能不能让代码更直接地反映出你的意图，从而消除这些注释？

有关解决方案请见附录 A。

3.3 过长的方法

症状

- 存在大量的代码行。（只要一看到超过 5～10 行代码的方法，我就会立即对其置疑。）

原因

我把这看做是 Columbo 并发症（Columbo syndrome）。Columbo 是一个侦探，其口头禅是“再多一点点”。对于方法也是类似的，作者在编写方法时总是想不到将整个流程适当分解，或者借助某些辅助类，而只是一味地再写一点。写代码比读代码容易，所以总是会出现过大的代码块。

采取的措施

- 使用抽取方法将方法分解为更小的部分。找出表述有意义代码块的注释或空格。要抽取的方法应当在语义上有具体的含义，而不是千篇一

律地每隔几行代码就引入一个函数（即将相应代码行改为调用这些代码行的函数）。

- 在开始分解方法之前，可能还需要找到其他一些有帮助的重构技术，用于清理直线式代码（即大量代码都放在一行上）、条件式和变量使用。

收益

可以增强表述能力，有可能暴露出重复性问题，通常有助于建立新的类和抽象。

讨论

- 人们有时会担心方法调用的增多会不会影响到性能。在大多数情况下，这都是杞人忧天。只要令代码尽可能清晰，就无需担心性能问题，此时你会更加明了怎样更改系统结构和算法会显著提高性能。
- 不要拿度量方法当做游戏。使用抽取方法是为了更深入地利用它。

不适用的情况

要想更好地表达某种想法，可能需要一个稍长一些的方法。（与所有其他异味类似，长度只是一个警告信号，并不表示一定出现了问题。）

习题 4　过长的方法。

请考虑以下代码（可在 www.xp123.com/rwb 在线获得）。

Machine.java

```
public class Machine {
    String name;
    String location;
    String bin;

    public Machine(String name, String location) {
    this.name = name;
        this.location = location;
    }

    public String take() {
        String result = bin;
        bin = null;
        return result;
    }
```

```
    public String bin() {
        return bin;
    }

    public void put(String bin) {
        this.bin = bin;
    }

    public String name() {return name;}
}
```

Robot.java

```
public class Robot {
    Machine location;
    String bin;

    public Robot() {}

    public Machine location() {return location;}
    public void moveTo(Machine location) {this.location = location;}

    public void pick() {this.bin = location.take();}
    public String bin() {return bin;}

    public void release() {
        location.put(bin);
        bin = null;
    }
}
```

RobotTest.java

```
import junit.framework.*;

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

  public void testRobot() {
        Machine sorter = new Machine("Sorter", "left");
        sorter.put("chips");
        Machine oven = new Machine("Oven", "middle");
        Robot robot = new Robot();

        assertEquals("chips", sorter.bin());
        assertNull(oven.bin());
        assertNull(robot.location());
        assertNull(robot.bin());

        robot.moveTo(sorter);
```

```
        robot.pick();
        robot.moveTo(oven);
        robot.release();

        assertNull(robot.bin());
        assertEquals(oven, robot.location());
        assertNull(sorter.bin());
        assertEuals("chips", oven.bin());
    }
}
```

Report.java

```
import java.util.*;
import java.io.*;

public class Report {
    public static void report(Writer out, List machines, Robot robot)
            throws IOException
    {
        out.write("FACTORY REPORT\n");

        Iterator line = machines.iterator();
        while (line.hasNext()) {
            Machine machine = (Machine) line.next();
            out.write("Machine " + machine.name());

            if (machine.bin() != null)
                out.write(" bin=" + machine.bin());
            out.write("\n");
        }
        out.write("\n");

        out.write("Robot");
        if (robot.location() != null)
             out.write(" location=" + robot.location().name());

        if (robot.bin() != null)
            out.write(" bin=" + robot.bin());

        out.write("\n");

        out.write("========\n");
    }
}
```

ReportTest.java

```
import junit.framework.TestCase;
```

```
import java.util.ArrayList;
import java.io.PrintStream;
import java.io.StringWriter;
import java.io.IOException;

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

    public void testReport() throws IOException {
        ArrayList line = new ArrayList();
        line.add(new Machine("mixer", "left"));

        Machine extruder = new Machine("extruder", "center");
        extruder.put("paste");
        line.add(extruder);

        Machine oven = new Machine("oven", "right");
        oven.put("chips");
        line.add(oven);

        Robot robot = new Robot();
        robot.moveTo(extruder);
        robot.pick();

        StringWriter out = new StringWriter();
        Report.report(out, line, robot);

        String expected =
            "FACTORY REPORT\n" +
            "Machine mixer\nMachine extruder\n" +
            "Machine oven bin=chips\n\n" +
            "Robot location=extruder bin=paste\n" +
            "========\n";

        assertEquals(expected, out.toString());
    }
}
```

A. 在 Report.java 中圈出 4 个代码块，说明在对此代码进行重构的过程中，你可能会抽取哪些函数。

B. 将 Report()方法重写为 4 条语句，就好像你已经对各个代码块应用了抽取方法一样。

C. 抽取一个单行方法是否有意义？

D. 过长的方法很容易识别出来，不过这在实际代码中仍然相当常见，这是为什么呢？

有关解决方案请见附录 A。

3.4 过大的类

症状

- 存在大量实例变量。
- 存在大量方法。
- 存在大量代码行。

原因

过大的类往往是一次增大一点。作者不断地向类增加新的功能，直到类最终变得过于庞大。有时这是因为作者没有充分了解组成整个类的各个部分。不管怎样，这样一个类往往说明了过多的功能被糅合在了一起。

采取的措施

一般来说，需要对类进行分解。如果类存在过长的方法，就应当先解决该异味。分解类有以下三种最常用的方法：

- 抽取类（Extract Class），如果能确定一个新类能承担此类的部分职责的话。
- 抽取子类（Extract Subclass），如果可以划分类和新子类之间的功能的话。
- 抽取接口（Extract Interface），如果可以确定客户所用的特性子集的话。

有时，类太大是因为它是一个 GUI 类，它表示的不仅是显示组件，还包括模型。在这种情况下，可以使用复制受察数据（Duplicate Observed Data）来帮助抽取域类（Domain Class）。

收益

可以增强表述能力，有可能暴露重复性问题。

习题 5　过大的类。

请考虑取自 Java 库的以下声明：

```
public class JTable extends JComponent
  implements Accessible, CellEditorListener,
     ListSelectionListener, Scrollable,
     TableColumnModelListener, TableModelListener
{
// 常量
public static final int AUTO_RESIZE_ALL_COLUMNS
public static final int AUTO_RESIZE_LAST_COLUMN
public static final int AUTO_RESIZE_NEXT_COLUMN
public static final int AUTO_RESIZE_OFF
public static final int AUTO_RESIZE_SUBSEQUENT_COLUMNS

// 构造函数
public JTable()
public JTable(TableModel, TableColumnModel)
public JTable(TableModel, TableColumnModel, ListSelectionModel)
public JTable(int, int)
public JTable(Object[][], Object[][])
public JTable(java.util.Vector, java.util.Vector)

// 方法
public void addColumn(TableColumn column)
public void addColumnSelectionInterval(int start, int finish)
public void addNotify()
public void addRowSelectionInterval(int start, int finish)
public void clearSelection()
public void columnAdded(TableColumnModelEvent event)
public void columnAtPoint(Point p)
public void columnMarginChanged(ChangeEvent event)
public void columnMoved(TableColumnModelEvent event)
public void columnRemoved(TableColumnModelEvent event)
public void columnSelectionChanged(ListSelectionEvent event)
public void convertColumnIndexToModel(int viewColumn)
public void convertColumnIndexToView(int modelColumn)
public void createDefaultColumnsFromModel()
public boolean editCellAt(int row, int column)
public boolean editCellAt(int row, int column, EventObject event)
public void editingCanceled(ChangeEvent event)
public void editingStopped(ChangeEvent event)
public AccessibleContext getAccessibleContext()
public boolean getAutoCreateColumnsFromModel()
public int getAutoResizeMode()
public TableCellEditor getCellEditor()
public TableCellEditor getCellEditor(int row, int column)
public Rectangle getCellRect(int row, int column,
                                    boolean includeSpacing)
public boolean getCellSelectionEnabled()
public TableColumn getColumn(Object object)
public Class getColumnClass(int column)
```

```
public int getColumnCount()
public TableColumnModel getColumnModel()
public String getColumnName(int column)
public Boolean getColumnSelectionAllowed()
public TableCellEditor getDefaultEditor(Class class)
public TableCellRenderer getDefaultRenderer(Class class)
public int getEditingColumn()
public int getEditingRow()
public Component getEditorComponent()
public Color getGridColor()
public Dimension getIntercellSpacing()
public TableModel getModel()

public Dimension getPreferredScrollableViewportSize()
public int getRowCount()
public int getRowHeight()
public int getRowMargin()
public Boolean getRowSelectionAllowed()

public int getScrollableBlockIncrement(
Rectangle visible, int orientation, int direction)
public Boolean getScrollableTracksViewportHeight()
public Boolean getScrollableTracksViewportWidth()
public int getScrollableUnitIncrement(
            Rectangle visible, int orientation, int direction)
public int getSelectedColumn()
public int getSelectedColumnCount()
public int[] getSelectedColumns()
public int getSelectedRow()
public int getSelectedRowCount()
public int[] getSelectedRows()
public Color getSelectionBackground()
public Color getSelectionForeground()
public ListSelectionModel getSelectionModel()
public Boolean getShowHorizontalLines()
public Boolean getShowVerticalLines()
public JTableHeader getTableHeader()
public String getToolTipText(MouseEvent event)
public TableUI getUI()
public String getUIClassID()
public Object getValueAt(int row, int column)
public Boolean isCellEditable(int row, int column)
public Boolean isCellSelected(int row, int column)
public Boolean isColumnSelected(int column)
public Boolean isEditing()
public boolean isManagingFocus()
public Boolean isRowSelected(int row)
public void moveColumn(int column, int newColumn)
public Component prepareEditor(TableCellEditor editor,
```

```
                                        int row, int column)
public Component prepareRenderer(TableCellRenderer renderer,
                                        int row, int column)
public void removeColumn(TableColumn column)
public void removeColumnSelectionInterval(int column1,
                                        int column2)
public void removeEditor()
public void removeRowSelectionInterval(int row1, int row2)
public void reshape(int x, int y, int width, int height)
public int rowAtPoint(Point point)
public void selectAll()
public void setAutoCreateColumnsFromModel(Boolean doAutoCreate)
public void setAutoResizeModel(int mode)
public void setCellEditor(TableCellEditor editor)

public void setCellSelectionEnabled(Boolean maySelect)
public void setColumnModel(TableColumnModel model)
public void setColumnSelectionAllowed(Boolean maySelect)
public void setColumnSelectionInterval(int column1, int column2)
public void setDefaultEditor(Class class, TableCellEditor editor)
public void setDefaultRenderer(Class class,
                                        TableCellRenderer renderer)
public void setEditingColumn(int column)
public void setEditingRow(int row)
public void setGridColor(Color color)
public void setIntercellSpacing(Dimention dim)
public void setModel(TableModel model)
public void setPreferredScrollableViewportSize(Dimension dim)
public void setRowHeight(int height)
public void setRowMargin(int margin)
public void setRowSelectionAllowed(Boolean maySelect)
public void setSelectionBackground(Color background)
public void setSelectionForeground(Color foreground)
public void setSelectionMode(int mode)
public void setSelectionModel(ListSelectionModel model)
public void setShowGrid(Boolean showing)
public void setShowHorizontalLines(Boolean b)
public void setShowVerticalLines(Boolean b)
public void setTableHeader(JTableHeader header)
public void setUI(TableUI ui)
public void setValueAt(Object value, int row, int column)
public void sizeColumnsToFit(int resizingColumn)
public void tableChanged(TableModelEvent event)
public void updateUI()
public void valueChanged(ListSelectionEvent)

//加 22 个受保护的变量

//加 10 个受保护的方法
```

```
//加从 JComponent 继承来的 97 个方法（和字段等）

//大约加 35 个从 Container 继承而来的方法

//大约加 85 个从 Component 继承而来的方法

//加从 Object 继承来的 11 个方法
}
```

A. 为什么这个类有如此多的方法？

B. 分析上面列出的所有方法，将它们划分到 5—10 个主要的功能领域。

C. 库的编写者可以采用哪些方法来去除上面的某些方法？

D. 在 Java 中，Object 有 11 个方法，而在 Smalltalk 中 Object 有 100 多个方法。为什么会存在这种差别？请与 Smalltalk 开发人员交流，弄清这是否是一种异味，以及相应的原因。

有关解决方案请见附录 A。

3.5 过长的参数表

症状

- 方法有 1 个或 2 个以上的参数。

原因

可能是为了尽量减少对象之间的耦合。这样做不是由被调用对象来了解类之间的关系，而是让调用者来确定所有一切，如此一来，方法所关注的就是调用者的信息要求它做些什么。

也有可能是程序员对例程进行了通用化，即通过创建一个通用算法并利用大量控制（control）参数来处理多种不同的情况。

采取的措施

- 如果参数值可以由另一个已知的对象得到，则可以将参数替换为方法（Replace Parameter with Method）。
- 如果参数来自一个对象，则可以保持对象完整（Preserve Whole Object）。

- 如果数据不是来自一个逻辑对象，则仍需要通过引入参数对象（Introduce Parameter Object）将其分组。

收益

可以增强表述能力，有可能暴露重复性问题。通常可以缩小规模。

不适用的情况

- 有时你希望两个类之间不要出现依赖性。例如，调用者存在依赖性，但是你并不希望传播这种依赖性。要确保对代码的修改不影响这种平衡。
- 有时参数并不存在有意义的分组，即无法将它们组织在一起。

说明

如前所述，异味并不一定表示有问题，在此便可体现这一点。你可能会“闻”到这样一种异味，即过长的参数表，但是对于当前情况而言这可能是合适的。

习题 6　过长的参数表。

请考虑 Java 库中声明的以下方法：

取自java.swing.CellRendererPane：

```
public void paintComponent(Graphics gr, Component renderer,
  Container parent, int x, int y, int width, int height,
  Boolean shouldValidate)
```

取自java.awt.Graphics:

```
public Boolean drawImage(Image image,
   int x1Dest, int y1Dest, int x2Dest, int y2Dest,
   int x1Source, int y1Source, int x2Source, int y2Source,
   Color color, ImageObserver obs)
```

取自java.swing.DefaultBoundedRangeModel：

```
public void setRangeProperties(
   int newValue, int newExtent,
   int newMin, int newMax,
   boolean isAdjusting)
```

取自java.swing.JOptionPane：

```
public static int showConfirmDialog(Component parent, Object
message, String title, int optionType, int messageType, Icon icon)
```

A. 对于以上各个声明，是否可以将某些参数合理地分组到一个新的对象中？

B. 为什么这些签名有这么多的参数？

C. 回顾 JTable 声明（本章前面的习题 5），能否在其中找出可分组的参数？

有关解决方案请见附录 A。

3.6 实战练习

习题 7　异味和重构。

请考虑以下异味：

A. 注释类。
B. 过大的类。
C. 过长的方法。
D. 过长的参数表。

对于以下所列的各种重构，指出它们对解决哪些异味有帮助（写出相应的表示字母）：

——复制受察数据
——抽取类
——抽取接口
——抽取方法
——抽取子类
——引入断言
——引入参数对象
——保持对象完整
——重命名方法
——将参数替换为方法

有关解决方案请见附录 A。

习题 8　触发器。

请考虑本章所述的异味（注释、过大的类、过长的方法和过长的参数表）。

A. 你认为其中哪一种最常见？哪一种是你最常创建的？

B. 为了避免孩子吮手指，有些父母会在孩子的手指上抹些有怪味道或者很辣的东西。这就相当于一种触发器，可以提醒孩子不要吮手指。你能否提供一些触发器，以便在某种异味初露端倪时就发现它？

有关解决方案请见附录 A。

3.7 小结

本章所述的异味是最容易发现的，但并不一定是最容易修复的。

另外还有一些度量方法可以用在软件中，其中许多都只是简单地缩短代码长度。当代码变得过大时，要特别注意。

重构和异味之间并不存在着一对一的关系，后面我们还会遇到以上某些重构技术。例如，抽取方法是一种可以修复许多问题的“多功能”工具。

最后请注意，异味只是表示可能存在问题，并不表示一定存在问题。有时你可能会发现一些误报（false positive），即某种做法看似异味，但实际上可能比其他做法更胜一筹。不过，大多数代码都确实存在着大量异味，你必须马不停蹄地予以解决。

补充点 I

异味和重构

表 I.1 列出了 Martin Fowler 所著的《重构》一书中介绍的异味（以及常用于修复它们的重构技术），表 I.2 列出了各种重构技术。

补充点 I1.1

检录。对于各种重构技术，每当一个异味用到一项重构技术时，就为该项技术设置一个检录标记。

补充点 I1.2

修复大多数异味的重构技术。哪些重构技术可以修复大多数异味？

有关解决方案请见附录 A。

补充点 I1.3

没有提到的重构技术。任何异味都未提及的重构技术有哪些？为什么没有涉及这些重构技术？

有关解决方案请见附录 A。

补充点 I1.4

其他异味。你是否能根据这个列表想到其他可能需要关注的异味？

有关解决方案请见附录 A。

表 I.1　异味及其相关重构技术（摘自 Fowler 的《重构》封底）

异味	常用重构技术
接口不同的相似类（Alternative Classes with Different Interfaces）	重命名方法（Rename Method）、搬移方法（Move Method）
注释（Comments）	抽取方法（Extract Method）、引入断言（Introduce Assertion）
数据类（Data Class）	搬移方法（Move Method）、封装字段（Encapsulate Field）、封装集合（Encapsulate Collection）
数据泥团（Data Clump）	抽取类（Extract Class）、引入参数对象（Introduce Parameter Object）、保持对象完整（Preserve Whole Object）
发散式改变（Divergent Change）	抽取类（Extract Class）
重复性代码（Duplicated Code）	抽取方法（Extract Method）、抽取类（Extract Class）、上移方法（Pull Up Method）、构建模板方法（Form Template Method）
依恋情结（Feature Envy）	搬移方法（Move Method）、搬移字段（Move Field）、抽取方法（Extract Method）
不当的紧密性（Inappropriate Intimacy）	搬移方法（Move Method）、搬移字段（Move Field）、将双向关联改为单向（Change Bidirectional Association to Unidirectional）、将继承替换为委托（Replace Inheritance with Delegation）、隐藏委托（Hide Delegate）
不完备的库类（Incomplete Library Class）	引入外来方法（Introduce Foreign Method）、引入本地扩展（Introduce Local Extension）

（续表）

异味	常用重构技术
过大的类（Large Class）	抽取类（Extract Class）、抽取子类（Extract Subclass）、抽取接口（Extract Interface）、将数据值替换为对象（Replace Data Value with Object）
懒惰类（Lazy Class）	内联类（Inline Class）、折叠继承体系（Collapse Hierarchy）
过长的方法（Long Method）	抽取方法（Extract Method）、将临时变量替换为查询（Replace Temp with Query）、将方法替换为方法对象（Replace Method with Method Object）、分解条件式（Decompose Conditional）
过长的参数表（Long Parameter List）	将参数替换为方法（Replace Parameter with Method）、引入参数对象（Introduce Parameter Object）、保持对象完整（Preserve Whole Object）
消息链（Message Chains）	隐藏委托（Hide Delegate）
中间人（Middle Man）	移除中间人（Remove Middle Man）、内联方法（Inline Method）、将委托替换为继承（Replace Delegation with Inheritance）
并行继承体系（Parallel Inheritance Hierarchies）	搬移方法（Move Method）、搬移字段（Move Field）
基本类型困扰（Primitive Obsession）	将数据值替换为对象（Replace Data Value with Object）、抽取类（Extract Class）、引入参数对象（Introduce Parameter Object）、将数组替换为对象（Replace Array with Object）、将类型码替换为类（Replace Type Code with Class）、将类型码替换为子类（Replace Type Code with Subclasses）、将类型码替换为状态/策略（Replace Type Code with State/Strategy）
拒收的遗赠（Refused Bequest）	将继承替换为委托（Replace Inheritance with Delegation）

（续表）

异味	常用重构技术
霰弹式修改（Shotgun Surgery）	搬移方法（Move Method）、搬移字段（Move Field）、内联类（Inline Class）
过分一般性（Speculative Generality）	折叠继承体系（Collapse Hierarchy）、内联类（Inline Class）、移除参数（Remove Parameter）、重命名方法（Rename Method）
Switch 语句（Switch Statements）	将条件式替换为多态（Replace Conditional with Polymorphism）、将类型码替换为子类（Replace Type Code with Subclasses）、将类型码替换为状态/策略（Replace Type Code with State/Strategy）、将参数替换为显式方法（Replace Parameter with Explicit Methods）、引入 Null 对象（Introduce Null Object）
临时字段（Temporary Field）	抽取类（Extract Class）、引入 Null 对象（Introduce Null Object）

表 I.2　重构技术名录（摘自 Fowler 的《重构》封二）

Add Parameter（增加参数）	Pull Up Constructor Body（上移构造函数体）
Change Bidirectional Association to Unidirectional（将双向关联改为单向）	Pull Up Field（上移字段）
Change Reference to Value（将引用改为值）	Pull Up Method（上移方法）
Change Unidirectional Association to Bidirectional（将单向关联改为双向）	Push Down Field（下移字段）
Change Value to Reference（将值改为引用）	Push Down Method（下移方法）
Collapse Hierarchy（折叠继承体系）	Remove Assignment to Parameters（移除对参数的赋值）
Consolidate Conditional Expression（合并条件表达式）	Remove Control Flag（移除控制标记）
Consolidate Duplicate Conditional Expression（合并重复的条件表达式）	Remove Middle Man（移除中间人）
Convert Procedural Design to Objects（将过程式设计转换为对象）	Remove Parameter（移除参数）
Decompose Conditional（分解条件式）	Remove Setting Method（移除设置方法）
Duplicate Observed Data（复制受察数据）	Rename Method（重命名方法）

（续表）

Encapsulate Collection（封装集合）	Replace Array with Object（将数组替换为对象）
Encapsulate Downcast（封装向下强制）	Replace Conditional with Polymorphism（将条件式替换为多态）
Encapsulate Field（封装字段）	Replace Constructor with Factory Method（将构造函数替换为工厂方法）
Extract Class（抽取类）	Replace Data Value with Object（将数据值替换为对象）
Extract Hierarchy（抽取继承体系）	Replace Delegation with Inheritance（将委托替换为继承）
Extract Interface（抽取接口）	Replace Error Code with Exception（将错误码替换为异常）
Extract Method（抽取方法）	Replace Exception with Test（将异常替换为测试）
Extract Subclass（抽取子类）	Replace Inheritance with Delegation（将继承替换为委托）
Extract Superclass（抽取超类）	Replace Magic Number with Symbolic Constant（将魔法数替换为符号常量）
Form Template Method（构建模板方法）	Replace Method with Method Object（将方法替换为方法对象）
Hide Delegate（隐藏委托）	Replace Nested Conditional with Guard Clause（将嵌套条件式替换为卫哨子句）
Hide Method（隐藏方法）	Replace Parameter with Explicit Method（将参数替换为显式方法）
Inline Class（内联类）	Replace Parameter with Method（将参数替换为方法）
Inline Method（内联方法）	Replace Record with Data Class（将记录替换为数据类）
Inline Temp（内联临时变量）	Replace Subclass with Fields（将子类替换为字段）
Introduce Assertion（引入断言）	Replace Temp with Query（将临时变量替换为查询）
Introduce Explaining Variable（引入解释变量）	Replace Type Code with Class（将类型码替换为类）

（续表）

Introduce Foreign Method（引入外来方法）	Replace Type Code with State/Strategy（将类型码替换为状态/策略）
Introduce Local Extension（引入本地扩展）	Replace Type Code with Subclasses（将类型码替换为子类）
Introduce Null Object（引入 Null 对象）	Self Encapsulate Field（自封装字段）
Introduce Parameter Object（引入参数对象）	Separate Domain from Presentation（将域与表示分离）
Move Field（搬移字段）	Split Temporary Variable（分解临时变量）
Move Method（搬移方法）	Substitute Algorithm（替代算法）
Parameterize Method（令方法参数化）	Tease Apart Inheritance（梳理并分解继承体系）
Preserve Whole Object（保持对象完整）	

第 4 章

命名

如何创建一个好的声明模型是软件开发中的一大挑战。为此，人们使用了多种工具来提供帮助，包括：

- 项目字典（project dictionariy）。
- 域术语、词汇及语言。
- XP 风格的表述方法。

如何命名是相当重要的。好的名字具有以下功能：

- 为相应域的讨论提供了一系列词汇。
- 传达意图。
- 能够大致预测系统将如何工作。
- 能在一个名字系统中相互支持。

选择好的名字是一项艰巨的任务，不过对此的努力不会白费。Ward Cunningham 介绍了一种使用辞典来得到合适名字的方法（请参见 http://c2.com/cgi/wiki?System-OfNames）。

有些开发小组拥有自己的编码标准和命名标准，由此来确定如何选择名字。我一般采用以下原则：

- 对处理方法（manipulator）使用动词，对访问方法（accessor）则使用名词和/或形容词。
- 对类似的事物使用相同的词，不同的事物则用不同的词。
- 倾向于使用独词名（即由一个单词构成的名字）。
- 最看重名字的表述能力。

在你发现还有更好的名字时大可改名，这非常简单，特别是在借助有关工具时更是轻松。在修改代码时，很有必要花一些精力来改进其中的命名。

4.1 所涉及的异味

- 名字（包括匈牙利记法）中嵌有类型（type Embedded In Name）。
- 表达力差的名字（Uncommunicative Name）。
- 不一致的名字（Inconsistent Names）。

4.2 名字（包括匈牙利记法）中包含类型

症状

- 名字采用复合词，即由一个词再加上参数的类型组成。例如，方法 addCourse(Course c)即属此类。
- 名字采用匈牙利记法，其中对象的类型被编码到名字中。例如，用 iCount 作为一个整型成员变量。
- 变量名反映的是其类型，而不是其用途或作用。

为何会出现这种情况

在具有表述能力的名字中可以加入类型。例如，相对于 schedule.add(course)，有人可能认为 schedule.addCourse(course)可读性更强。（尽管我不这么认为，但确实有些人持这种观点。）

包含类型的名字反映出一种重复性：参数和名字都提到了同一个类型。嵌有类型的名字可能会在以后带来一些不必要的麻烦。例如，假设我们为 course 引入一个父类来同时涵盖各门课程和一系列课程。这样，对于引用 addCourse()的所有位置，它们的名字就不合适了。这样的话，我们要么对每个调用进行改名，要么就无奈地忍受糟糕名字所带来的困扰。最后，如果仅以操作来命名，可以更容易地发现重复点和找出新的抽象。

匈牙利记法往往被用作编码标准的一部分。在基于指针的语言（如 C）中，如果你明白**ppc 实际上是一个字符的话会很有用，但是在面向对象的语言中，这种记法会使名字与其类型过度耦合。

有些程序员或开发小组采用以下约定：用一个前缀来指示某个变量究竟是一个成员变量（_count 或 m_count）还是一个常量（ALL_UPPER_CASE）。同样地，在对某个变量（不管是局部变量还是成员变量，或者其他变量）进行修改时，这也会导致出现问题。难道我们完全不需要知道一个变量属于哪个类吗？并非如此，有时确实需要对此有所了解。另外，如果你不能说明各个变量的所属类别，这可能意味着相应的类过于庞大。（关于这些约定是否有问题的讨论并没有一个统一的结论，即便在 Fowler 的《重构》中，其代码也用了_member 等形式的变量。）

采取的措施

- 使用重命名方法（Rename Method），或者重命名字段和常量，使名字能够表述其意图，而不必与类型绑定得过于紧密。

收益

可以增强表述能力，可能更容易发现重复性。

不适用的情况

你可能希望某个类对两个彼此相关的不同类型执行同一种操作，不过这种情况相当少见。例如，我们可能有一个 Graph（图）类，其中包括 addPoint()方法和 addLink()方法。如果针对这两种方法的抽象行为相同，那么重载方法名（add()）会更合适。

有时你使用的编码标准会采用一些排版约定来区别不同的变量类。这时你就必须遵守这些约定，即优先考虑代码的可读性，不要太看重无类型（untyped）名字的灵活性。

4.3 表达力差的名字

症状

名字未能充分表述意图。

- 单字符或双字符名。
- 无元音的名字。
- 带有编号的变量（例如 panel、pane2 等）。

- 奇怪的缩写。
- 容易产生误导的名字。

原因

最初在实现某个对象时，必须以某种方式进行命名，你会给出当时你认为最佳的名字，然后继续操作。但是后来，你发现其实可以选择一个更好的名字。

采取的措施

- 重命名方法（字段、常量等）以得到一个更好的名字。

收益

可以增强表述能力。

不适用的情况

有些小组习惯用 i/j/k 表示循环索引，或者用 c 表示字符。如果作用域相当小，这不会有什么大问题。类似地，有时我们也会发现，带有编号的变量的表达能力似乎更好。

习题 9　命名。

分析以下名字，看看哪些内嵌有类型，哪些表达力差，哪些则是好名字。

```
___ addItem(item)
___ doIt()
___ getNodesArrayList()
___ getData()
___ makeIt()
___ multiplyIntInt(int1, int2)
___ processItem()
___ sort()
___ spin()
```

有关解决方案请见附录 A。

4.4 不一致的名字

症状

- 对于同一个对象，在某处用了一个名字，而在别处又用了另外一个名字。例如，对于同一个基本特性，可能用了 add()、store()、put()和 place()。

原因

不同的人会在不同时刻创建类（在增加新类之前，应当先对现有类进行分析，但人们往往会忽略这一点）。有时，你会发现这是故意而为，目的是区分不同的名字（但这样会产生误导）。

采取的措施

选择最佳的名字并使用重命名方法（字段或常量等），为同一对象提供相同的名字。做到这一点后，你就会发现，相比从前，有些类越发显得相似。查找是否存在重复性异味并予以消除。

收益

可以增强表述能力，还可能暴露重复性问题。

说明

Eiffel 语言使用了一个常用词库来命名它的库特性。受这种技术的启发，你可以在现有库名字中查找你使用的词汇。

习题 10　评价命名。

你会使用下面哪些名字？

A. 清空窗口（屏幕窗口）。

clear()
wash()
erase()
deleteAll()

B. 对于栈。

add()
insert()
push()
addToFront()

C. 对于编辑器（删除所选文本）。

cut()
delete()
clear()
erase()

D. 作为一个文件比较程序的一部分。

line1.compare(line2)
line1.equals(line2)
line1.identicalTo(line2)
line1.matches(line2)

有关解决方案请见附录 A。

习题 11　XmlEditor。

有一个类 XmlEditor，现在要为它引入一个父类。

A. 如何命名它？
B. 如果要引入该父类将实现的一个接口，使用什么名字好？

有关解决方案请见附录 A。

第 5 章

不必要的复杂性

“它对于你实属多余。”——Ron Jeffries

就解决当前问题而言，有时代码会存在不必要的复杂性。这个问题可以归为两个原因：

- 有时代码过于复杂是因为历史原因[例如，可能存在代码腐化（code rot）的情况，这是原有做法所遗留的产物]，但现在可以免去这种复杂性了。
- 产生复杂性的另一个原因是设计过于通用化（或具有过分一般性）。这通常是因为编写者考虑到了将来的需求，也可能是因为他过早地进行了性能优化。

一定要将这些问题根除。你会发现，消除这些问题会使你获得更深入的认识，而且会使代码更加简化。

5.1 所涉及的异味

- 死代码（Dead Code）。
- 过分一般性（Speculative Generality）。

5.2 死代码

症状

- 变量、参数、字段、代码段、方法或类未在任何地方使用（可能除了测试以外）。

原因

- 需求发生改变，或者采用了新的方法，但没有进行充分的清理。
- 过于复杂的逻辑会产生一些根本不会发生的条件组合。在简化条件时，即可发现这一点。

采取的措施

删除未用到的代码及所有相关测试。

收益

降低规模。可以增强表述能力，代码更简单。

不适用的情况

如果应用程序是一个框架，你可能会提供一些内容来支持客户的需求，严格来说，框架本身并不需要这些内容。例如，一个类可能有一个空的 hook 方法供（尚不存在的）子类来调用。

说明

如果不借助工具，你将很难检测到这种异味。一旦你有所怀疑，就应该立即采取行动，比如对 new ClassName 进行全局搜索。

5.3 过分一般性

症状

- 存在未用到的类、方法、字段、参数，等等。它们可能没有客户，或者仅有测试作为客户。
- 就当前要实现的需求而言，代码过于复杂。

原因

人们在编写代码时，总是希望它在将来会更有用，但往往事与愿违。人们在试图充分挖掘代码的需求时，往往会为了一般性或完备性而增加一些东西，但最后这些新增内容却从未用到。有时，一些以前用过的代码可能会由

于出现了新的或改进的处理方法而被束之高阁。（过分一般性也可能是故意创建的死代码。）

采取的措施

- 对于一个不必要的类：
  - 如果此类的父类或子类更适合完成操作，则通过折叠继承体系（Collapse Hierarchy）将其合并。
  - 否则，通过内联类（Inline Class）将其行为合并到它的调用者中。
- 对于不必要的方法，使用内联方法（Inline Method）或移除方法（Remove Method）。
- 对于不必要的字段，确保没有对它进行任何引用后将其删除。
- 对于不必要的参数，使用移除参数（Remove Parameter）。

收益

降低规模。可以增强表述能力，代码更简单。

不适用的情况

- 如果应用程序是一个框架，则可能需要提供一些内容来支持客户的需求，严格地说，框架本身并不需要这些内容。例如，一个类可能会有一个空的 hook 方法供（尚不存在的）子类来调用。
- 有些内容只有测试方法会使用，但是它们被作为测试探测点（test probe points）提供，以便测试有关类的特权信息。不过请注意，这可能意味着你忽略了某个抽象，而你原本能够基于此抽象独立完成测试。

习题 12　着眼今天，还是放眼未来。

请分析以下观点：

XP 认为过分一般性是一种异味，而且你也不需要这种过分的一般性。也就是说，应当让代码适应目前的需求，而不要好高骛远，试图考虑如何满足未来的需求。（因此，XP 开发小组更倾向于基于应用程序来开发框架，而不是先构建一个框架，再用它来创建一个应用程序。）

还有一种方法是基于灵活性进行设计，或者面向一般性来设计。也就是说，应当基于所有可能的需求使类尽可能完备。

哪些因素有助于确定以上哪种方法更好？

如果你认为“今天只做今天事”（即仅针对当前需求进行设计）更好，理由是什么？

如果你认为“未雨绸缪”（即今天的设计要考虑到未来的需求）更好，理由又是什么？

有关解决方案请见附录 A。

补充点 2

逆处理

进行重构时，我们总是尽一切可能改变代码。有时，对于目前而言还不错的修改，将来再看时却觉得改得不尽如人意，此时我们通常会取消原先的重构，即进行所谓的“逆处理”（reverse）。

补充点 I2.1

表 I.3 提供了一组重构技术。请在每种重构技术的右边写出能够抵消其作用的相应重构技术，即逆处理。

有关解决方案请见附录 C。

表 I.3　重构技术及其逆处理

重构技术	逆处理
Add Parameter（增加参数）	
Change Bidirectional Association to Unidirectional（将双向关联改为单向）	
Change Reference to Value（将引用改为值）	
Change Unidirectional Association to Bidirectional（将单向关联改为双向）	
Change Value to Reference（将值改为引用）	
Collapse Hierarchy（折叠继承体系）	
Extract Class（抽取类）	
Extract Method（抽取方法）	
Extract Subclass（抽取子类）	
Hide Delegate（隐藏委托）	

（续表）

重构技术	逆处理
Inline Class（内联类）	
Inline Method（内联方法）	
Inline Temp（内联临时变量）	
Introduce Explaining Variable（引入解释变量）	
Move Field（搬移字段）	
Move Method（搬移方法）	
Parameterize Method（令方法参数化）	
Pull Up Field（上移字段）	
Pull Up Method（上移方法）	
Push Down Field（下移字段）	
Push Down Method（下移方法）	
Remove Middle Man（移除中间人）	
Remove Parameter（移除参数）	
Replace Delegation with Inheritance（将委托替换为继承）	
Replace Inheritance with Delegation（将继承替换为委托）	
Replace Parameter with Explicit Method（将参数替换为显式方法）	
Substitute Algorithm（替代算法）	

第 6 章

重复

“你可以再说一遍！”

30 多年来，重复一直被认为是扰乱程序员的一大祸害。重复会导致下面这些问题：

- 需要维护更多的代码（无论是从理论上讲，还是从实际角度看，代码越多负担越重）。
- 变化的部分隐藏在不变的部分之中（这是一个认知问题，这样很难看到重要的内容）。
- 代码变化通常会隐藏更深层次的相似性。这样，要了解所有相似代码之间更进一步的解决方案会变得非常困难。
- 很有可能出现这样的情况：某处的一个 bug 被修正了，而位于其他位置的相同 bug 却没有被发现。如果你发现同一事物存在两种变体，那么很难确定哪一种变化才是正确的模式，而且也不知道是否有充足的理由进行这种变化。

David Parnas 引入了信息隐藏（information hiding）的概念：一个好的模块应当有自己的“秘密”。通过确保模块保守其秘密，往往能够减少重复。（请参见 On the criteria to be used in decomposing systems into modules，*Communications of ACM*，15[2]，1972）。

重复是一个根本性问题。其他许多异味所指示的问题都可以算是重复的一些特例。重复并不总是显而易见的，但一定要加以处理，这一点至关重要。务必让代码将各种思想表述“一次且仅一次”。

6.1 所涉及的异味

- 魔法数（Magic Number）。
- 重复性代码（Duplicated Code）。
- 具有不同接口的相似类。

6.2 魔法数

症状

- 代码体中出现了常量（除空字符串以外，还包括 0、1，可能还有 2，以及其他一些基本情况）。

原因

有人可能需要某个值，于是就将这个值置于代码中。就其本身而言，这也许并不算很糟糕，但是通常还有另外一些值是由该值派生而来的或者依赖它。例如，假设有一个定义为 banana 的字符串和一个长度变量 6。如果修改了此字符串，则需要调整该长度变量，但这一点并不明显，往往会被忽视。

这是重复的一种简单形式，而且识别起来非常容易。

采取的措施

- 对于特殊值，将魔法数替换为符号常量（Replace Magic Number with Symbolic Constant）。
- 如果值是字符串（例如，对话框的文本），则可以将它放在某个具有映射功能的函数中。（这就向着国际化迈进了一步，同时还会减少重复。）

收益

减少重复，增强表述能力。

不适用的情况

如果测试只使用它需要的值，那么它的可读性会更强。但即便这样，如果涉及派生值，你仍然需要借助符号常量。

如果使用求助某种映射解决方案，那么关键问题就是让代码与映射保持一致。

6.3 重复性代码

症状

- 简单形式（easy form）：两段代码看上去几乎相同。
- 复杂形式（hard form）：两段代码具有几乎相同的作用（无论在哪个概念层次上）。

原因

由于程序员们通常会独立地开发系统的不同部分，因此会出现一些重复，而他们并没有意识到创建了几乎相同的代码。有时，人们已经认识到了存在重复，但却懒得去消除这些重复。还有一些情况是，重复有可能被其他异味掩盖，一旦修正了这些异味，重复也就暴露出来了。

最坏的情况是程序员故意复制代码，这也可能是最普遍的情况。他们会找到一些差不多正确的代码，然后将它们稍加改动后复制并粘贴到新的位置。

采取的措施

- 如果一个方法内部或者同一个类的两个不同方法中出现重复，则使用抽取方法（Extract Method）将公共部分取出放在一个单独的方法中。
- 如果两个兄弟类中出现重复，则使用抽取方法创建一个单独的例程，然后使用上移字段（Pull Up Field）和/或上移方法（Pull Up Method）合并公共部分。之后便可以使用构建模块方法在父类中创建一个公共算法，使子类中不再出现重复。
- 如果两个无关的类中存在重复，则可以通过抽取类（Extract Class）将公共部分抽取到一个新的类中，或者确定这个异味是否是依恋情结（Feature Envy）（参见第 10 章），倘若如此，公共代码实际上只属于其中某一个类。

- 在以上情况中，你可能会发现某两处在字面上并不相同，但它们却有着相同的作用。如果是这样，可以采用替代算法（Substitute Algorithm）确保只涉及一个副本。

收益

可以减少重复，降低规模。还可以得到更好的抽象和更灵活的代码。

不适用的情况

在极少的情况下，你可能会发现有了重复似乎表述得更清晰，因此仍保留这些重复不动。另外，有些重复可能纯属巧合，在这种情况下，将重复的两处硬生生合并到一起只会使读者摸不着头脑。

6.4 具有不同接口的相似类

症状

- 两个类似乎在执行相同的工作，但却使用了不同的方法名。

原因

人们会创建类似的代码来处理类似的情况，却没有意识到还存在与之类似的其他代码。

采取的措施

协调各个类，使它们一致，从而去除其中某个类。

1. 采用重命名方法使方法名类似。
2. 使用搬移方法（Move Method）、增加参数（Add Parameter）和令方法参数化（Parameterize Method）来使协议（即方法签名和实现途径）类似。
3. 如果两个类只是相似而并非相同，在对它们进行合理协调后，可抽取超类（Extract Superclass）。
4. 尽量删除多余的类。

收益

减少重复。还可能会降低规模，另外有可能增强表述能力（更加明确地表述使用哪种途径）。

不适用的情况

有时两个类都无法改动（例如，如果二者在不同的库中）。每个库对于同一个概念有它自己的解释，将它们统一起来可能并没有合适的做法。

6.5 实战练习

习题 13　两个库（有难度）。

集成两个来自不同源的模块，每个模块有各自的日志记录方法。

系统A：

```
package com.fubar.log;
   public final class Log {
      public int INFO=1, WARN=2, ERROR=3, FATAL=4;
      public static void setLog(File f) {...}
      public static void log(int level, String msg) {...}
   }
```

代码中出现了大量对 Log.log 的调用。

系统B：

```
package com.bar.logger;
   public class Logger {
      public void informational(String msg) {...}
      public void informational(String msg, Exception e) {...}
      public void warning(String msg) {...}
      public void warning(String msg, Exception e) {...}
      public void fatal(String msg) {...}
      public void fatal(String msg, Exception e) {...}
   }

   public class LogFacility {
      public Logger makeLogger(String id) {...}
      public static void setOutput(OutputStream out) {...}
   }
```

可能需要记录的对象拥有一个 Logger 对象。

你的远期目标是升级到 JDK 1.4 中新的标准日志记录功能（logging facility），但是目前环境尚不支持。

A. 要使这些类实现目标，总体方案是什么？（一定要考虑到 JDK 1.4 的有关内容。）

B. 为每个日志记录工具（logger）创建一个简单的测试，然后用最简单的方法来实现相应的日志记录工具。

C. 协调类，从而去除其中某个类。（不必担心 JDK 1.4 将来的情况。）

有关解决方案请见附录 A。

习题 14　属性。

（可在 www.xpl23.com/rwb 在线获得）

```
public void getTimes(Properties props)
throws Exception {
  String valueString;
  int value;

  valueString = props.getProperty("interval");
  if (valueString == null) {
    throw new MissingPropertiesException("monitor interval");
  }
  value = Integer.parseInt(valueString);

  if (value <= 0) {
    throw new MissingPropertiesException("monitor interval > 0");
  }
  checkInterval = value;

  valueString = props.getProperty("duration");
  if (valueString == null) {
    throw new MissingPropertiesException("duration");
  }

  value = Integer.parseInt(valueString);
  if (value <= 0) {
    throw new MissingPropertiesException("duration > 0");
  }
  if ((value % checkInterval) != 0) {
    throw new MissingPropertiesException("duration % checkInterval");
  }
  monitorTime = value;

  valueString = props.getProperty("departure");
```

```
    if (valueString == null) {
      throw new MissingPropertiesException("departure offset");
    }
    value = Integer.parseInt(valueString);
    if (value <= 0) {
      throw new MissingPropertiesException("departure > 0");
    }
    if ((value % checkInterval) != 0) {
      throw new MissingPropertiesException("departure
                                                % checkInterval");
    }
    departureOffset = value;
  }
```

A. 你将如何处理此重复？请注意，测试 checkInterval 时并没有涉及%。

有关解决方案请见附录 A。

习题 15　模板示例。

最早见于 *Extreme Programming Explored*（可在 www.xp123.com/rwb 在线获得）。

```
try {    String template = new String(sourceTemplate);
    // 代替%CODE%
    int templateSplitBegin = template.indexOf("%CODE%");
    int templateSplitEnd = templateSplitBegin + 6;
    String templatePartOne = new String(
        template.substring(0, templateSplitBegin));
    String templatePartTwo = new String(
        template.substring(templateSplitEnd,
                                          template.length()));
    code = new String(reqId);
    template = new String(templatePartOne + code + templatePartTwo);


    // 代替%ALTCODE%
    templateSplitBegin = template.indexOf("%ALTCODE%");
    templateSplitEnd = templateSplitBegin + 9;
    templatePartOne = new String(
       template.substring(0, templateSplitBegin));
    templatePartTwo = new String(
       template.substring(templateSplitEnd, template.length()));
    altcode = code.substring(0,5) + "-" + code.substring(5,8);
    out.print(templatePartOne + altcode + templatePartTwo);
} catch (Exception e) {
    System.out.println("Error in substitute()");
}
```

A. 你发现了什么重复？

B. 怎么做才能去除这些重复？

C. 有一段代码存在重复，即形式为 new String（一些不同的字符串）的结构。此代码的作用是什么？它与类 String 所用的 intern()方法有什么关系？本例中是否可以使用类 String 的 intern()方法？

有关解决方案请见附录 A。

习题 16　复制受察数据（有难度）。

重构技术“复制受察数据（Duplicate Observed Data）”的原理如下：

如果某个显示部件（widget）类型的类中存在域数据，则将此域数据移到一个新的域类中并设置一个观察者，以便在数据改变时通知显示部件。

这样，数据在某一处，我们不仅复制了它，而且还在两个对象之间添加了同步需求。

A. 为什么这种重复被认为是可以接受的（甚至是必要的）？[提示：回答时请考虑到观察者（Observer）或模型-视图-控制器（Model-View-Controller）模式。]

B. 此方法对性能有什么影响？

有关解决方案请见附录 A。

习题 17　Java 库。

A. Java 库中有多处存在重复，请举几个例子。这些重复有的是低层次的，有的是中等层次的，还有的是高层次的。

B. 为什么会存在这些重复？这些重复有意义吗？

有关解决方案请见附录 A。

习题 18　点。

假设有以下两个类：

```
public class Bird {
   // ...
   public void move(Point vector) {
      x += vector.x % maxX;
      y += vector.y % maxY;
   }
```

```
}

public class Button {
   // ...
   public void setPosition(Point p) {
      x = p.x;
      while (x >= maxX) x -= maxX;
      while (x < 0) x += maxX;

      y = p.y;
      while (y >= maxY) y -= maxY;
      while (y < 0) y += maxY;
   }

}
```

A. 这段代码中存在什么重复？

B. 你会怎么做来去除这两个类中的重复？

C. 有时，两个重复代码版本很相似，只是其中一个已经修复了某个 bug，而另一个还没有。重构对这种情况有何帮助？

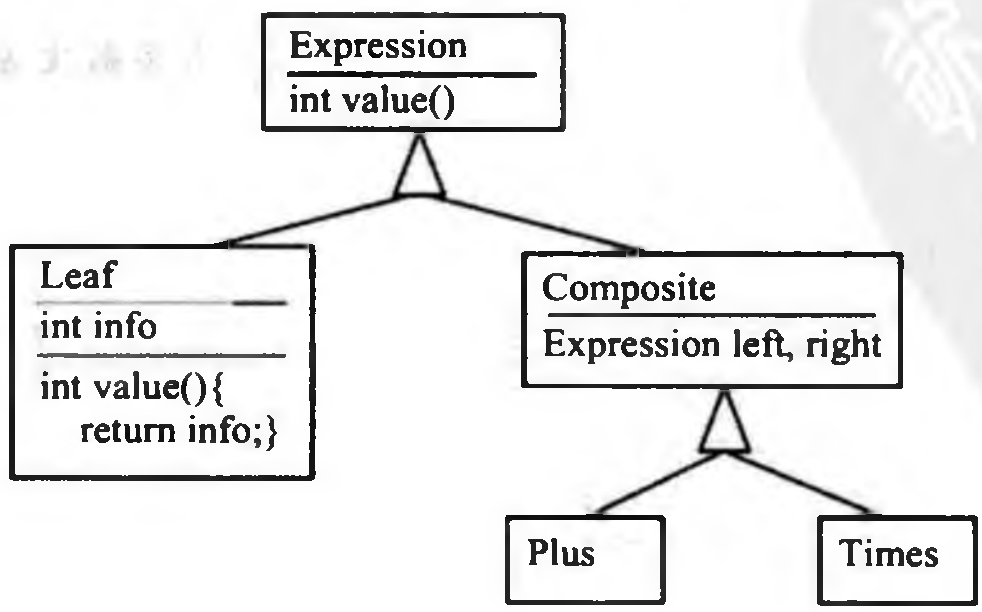
有关解决方案请见附录 A。

习题 19 表达式。

假设有一个树形构造的表达式。3 + 4*5 所对应的树形如下所示：

```
  +
/   \
3    *
   /   \
   4   5
```

类继承体系可能如下所示：



Plus.value()的代码如下：

```
public int value() {
   int v1 = left.value();
   int v2 = right.value();
   return v1 + v2;
}
```

Times. value()的代码如下：

```
public int value() {
   int v1 = left.value();
   int v2 = right.value();
   return v1 * v2;
}
```

A. 你会采取哪些措施来减少重复？

有关解决方案请见附录 A。

第 7 章

条件逻辑

“不怕做不到，就怕想不到。”

面向对象程序设计关注的是对象及对象之间的关系，这是自然，不过对象内部的代码同样很重要。你可以阅读一些经典的书籍，从中获得关于如何编写优质而简洁的代码的启迪。如 Jon Bentley 的 *Programming Pearls*（《编程珠玑》)或 Brian Kernighan 和 P.J.Pike 合著的 *Elements of Programming Style*。

条件逻辑通常是此类代码中最具技巧性的部分，原因有如下几点。

- 因为必须考虑代码中的多条路径，所以分析起来相当困难。
- 很有可能试图增加特殊用例处理，而不是开发面向一般情况的用例。
- 条件逻辑有时能勉强代替面向对象机制。

7.1 所涉及的异味

- Null 检查（Null Check）。
- 复杂的布尔表达式。
- 特殊用例（Special Case）。
- 模拟继承（Switch 语句）。

7.2 Null 检查

症状

类似下面的代码反复出现。

```
If (xxx == null) …
```

原因

有人认为可以用 Null 作为默认值，这样便免去了初始化某些字段或创建某些对象的麻烦，或者不考虑那些意料之外的情况。引入 Null 检查是为了“绕开”空指针（Null pointer）bug，但这并没有从根本上解决问题。

采取的措施

- 如果有合适的默认值，还是采用该默认值吧。
- 否则引入 Null 对象（Introduce Null Object），创建一个显式使用的默认对象。

收益

减少重复，减少逻辑错误和异常。

不适用的情况

- 如果只有一处有 Null 检查（例如，在一个工厂方法中），那么创建一个单独的 Null 对象（Null Object）就得不偿失了。
- Null 对象提供的方法必须定义安全的行为。它们通常相当于标识（identity）对象（如同 0 与加法的关系那样）。如果你没有对每个方法定义安全的行为，就可能无法使用某个 Null 对象。
- 避免出现以下情形，即在不同上下文中，Null 有两种或两种以上的含义。（可以用不同的 Null 对象对此提供支持。）

习题 20　Null 对象。

请考虑习题 4 中的代码（参见第 3 章）。

A. 有些 Null 检查是对 Null 字符串的检查。还有一种做法是使用空字符串，这种方法有什么缺点（注意考虑测试及这里未出现的所有其他客户类）？

B. 对于这个问题，还有什么解决方法？

C. 抽取一个 Bin 类并引入一个 Null 对象。

有关解决方案请见附录 A。

7.3 复杂的布尔表达式

症状

代码中存在复杂的条件，其中涉及 and、or 和 not。

原因

代码可能从一开始就很复杂，也可能在开发过程中被加入了额外的条件。

采取的措施

应用 DeMorgan 法则：

```
! (a && b)  => (!a) || (!b)
```

和

```
!(a || b)  =>  (!a) && (!b)
```

你会发现，如果将某些变量改名来反映其“反面”含义（即取反后的含义）的话，它们会具有更好的表达能力。

- 引入解释变量（Introduce Explaining Variable），使每个子句意思更清晰。
- 使用卫哨子名来剔除（peel off）某些条件，这样余下的子句就会变得更简单。
- 采用分解条件式（Decompose Conditional）将各个部分置于它自己的方法中。

收益

可以增强表达能力。

不适用的情况

你可能会找到其他方法来简化表达式，或者你发现采用以上重构技术重写后，表达式的表达能力并没有多大改善。

习题 21　条件表达式。

考虑以下代码段：

```
if (!((score > 700) ||
   ((income >= 40000) && (income <= 100000)
  && authorized && (score > 500)) ||
   (income > 100000)))
   reject();
else
   accept();
```

A. 应用 DeMorgan 法则尽可能对此简化。

B. 基于原来的形式，通过引入解释变量重写此条件。

C. 基于原来的形式，反转 if 和 else 子句，然后将原来的子句分解为多个 if 子句（需要在 3 个不同的位置调用 accept()）。

D. 通过抽取一个方法来计算条件，从而合并条件表达式。

E. 哪种方法最简单？哪种方法最清晰？能否结合使用这些技术？

F. 请用表格的形式描述条件。行和列要以 3 个变量为基础：其中一个变量表示 3 个 Score 范围，另一个变量对应不同 income 范围，还有一个变量则对应 authorized（授权）标志。表中单元格的值不是“接受（accept）”就是“拒绝（reject）”。

有关解决方案请见附录 A。

7.4　特殊用例

症状

- 复杂的 if 语句。
- 在工作（特别是在比较常量和枚举变量时）之前检查某些特定的值。

原因

有人认为需要一种特殊用例。

采取的措施

- 如果条件式取代了多态，则将条件式替换为多态（Replace Conditional with Polymorphism）。

- 如果 if 和 then 子句非常相似，可以重写它们，使同样的代码段无论对于哪种情况都能生成正确的结果，然后便可以去除条件式了。

收益

可以增强表达能力。还可能会暴露重复性问题。

不适用的情况

- 在递归定义中，常常有一个基本用例来结束递归，不要将它们一并删除。
- 有时，使用 if 子句可能是最简单的实现方法。

7.5 模拟继承（Switch 语句）

症状

- 代码使用了一条 switch 语句（特别是在设置类型字段时）。
- 代码某一行存在多条 if 语句（特别是在对同一个值进行比较时）。
- 代码使用了 instanceof（或其等价形式）来确定处理的类型是什么。

原因

出现这种异味往往是因为你懒得引入新的类。第一次需要某种条件行为时，你可能会使用一条 if 或 switch 语句，而不是一个新类。此时还不会出现什么大问题，因为条件只出现一次。假设需要另一个基于同一类型码的条件，你可能会引入第二条 switch 语句，而不是修正这个缺乏多态性的问题。

有时缺乏多态性并不能通过一条明显的 switch 语句发现，而是暗藏在一系列 if 语句中，不过根本问题是一样的。

采取的措施

不要模拟继承，而应使用程序设计语言中内置的机制。

- 如果针对相同条件的一条 switch 语句在多处出现，通常使用一个类型码（type code），然后将它替换为对象中内置的多态。为此，需要采用一系列重构技术。

  1. 抽取方法。抽出每个分支的代码。
  2. 搬移方法。将相关代码搬移至适当的类。

3. 将类型码替换为子类（Replace Type Code with Subclass）或将类型码替换为状态/策略（Replace Type Code with State/Strategy）。建立继承体系结构。
4. 将条件式替换为多态（Replace Conditional with Polymorphism）。去除条件式。

- 如果条件式出现在一个单独的类中，可以通过将参数替换为显式方法（Replace Parameter with Explicit Methods）或引入 Null 对象来取代条件逻辑。

收益

可以增强表述能力，可能会暴露重复性问题。

不适用的情况

有时，采用 switch 语句是表述逻辑的最简单方法。如果代码处理的是一个简单的问题，而且仅用在一处，那么你可能觉得没有必要去创建一个单独的类。在与系统中的非面向对象部分进行交互时，这种情况特别常见。Michael Feathers 曾指出："如果 switch 语句能把数据转换成对象，那真的就什么都不缺了。"

单独的一条 switch 语句有时会用在一个工厂（Factory）或抽象工厂（Abstract Factory）中（更多相关信息请参见 Gamma 的《设计模式》一书），这样一个地方就确定了如何配置整个工厂。有时可以将它替换为 Class.forName()（即动态类加载），但是由于安全性、表述能力或性能等问题，这种做法并不受欢迎。

有时，一条 switch 语句会用在多个相关的位置来控制一个状态机。要确定这样做是否更有意义，还是采用状态模式（State pattern）（请参见 Gamma 的《设计模式》）更合适。

习题 22　switch 语句。

请考虑以下代码：

```
public void printIt(Operation op) {
    String out = "?";
    switch (op.type) {
    case '+': out = "push"; break;
    case '-': out = "pop"; break;
    case '@': out = "top"; break;
    default: out = "unknown";
```

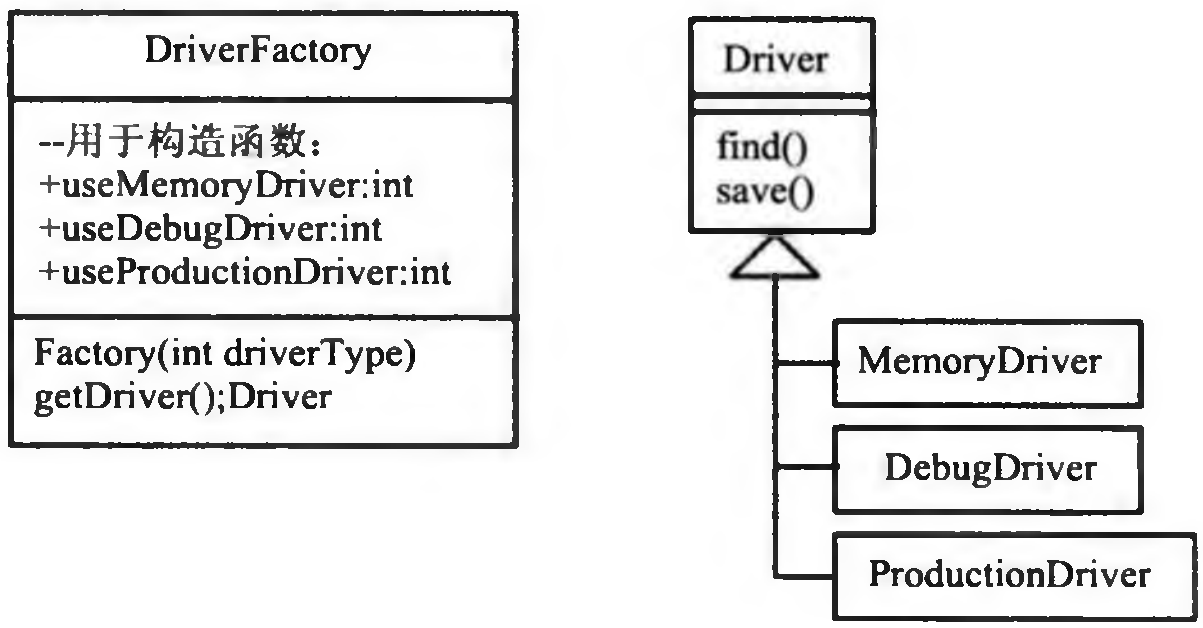
```
    }
    System.out.println("operation=" + out);
}

public void doIt(Operation op, Stack s, Object item) {
    switch (op.type) {
    case '+': s.push(item); break;
    case '-': s.pop(); break;
    }
}
```

A. 需要采取什么措施？

有关解决方案请见附录 A。

习题 23　工厂方法（有难度）。

考虑以下类结构：



A. 根据所示设计为此工厂编写代码。注意：要向构造函数传递以上 3 个常量之一，通过这来确定 getDriver()返回的是哪种类型的驱动器（Driver）。

B. 代码中可能存在关于 switch 语句（即使是用 if 来实现）的异味，这种条件逻辑合适吗？

C. 有的工厂使用动态类加载，即通过 Class.forName()根据需要加载相应的类。请修改你的工厂，让它接受一个字符串参数（即驱动器类的名字）并动态地加载实例。你的代码将不再显式地提及类型。这会有哪些好处？（提示：注意考虑对构建系统的影响。）

D. 这种修改有哪些缺点？

有关解决方案请见附录 A。

补充点3

设计模式

补充点 13.1 模式

以下所列的是 Gamma 的《设计模式》中所描述的一组设计模式。如果要发展其中某些模式，你会采用哪些重构技术？

创建型模式（Creational Patterns）：

——抽象工厂模式（Abstract Factory）
——生成器模式（Builder）
——工厂方法模式（Factory Method）
——原型模式（Prototype）
——单例模式（Singleton）

结构型模式（Structural Patterns）：

——适配器模式（Adapter）
——桥接模式（Bridge）
——组合模式（Composite）
——装饰器模式（Decorator）
——外观模式（Facade）
——享元模式（Flyweight）
——代理模式（Proxy）

行为型模式（Behavior Patterns）：

——职责链模式（Chain of Responsibility）

——命令模式（Command）
——解释器模式（Interpreter）
——迭代器模式（Iterator）
——中介者模式（Mediator）
——备忘录模式（Memento）
——观察者模式（Observer）
——状态模式（State）
——策略模式（Strategy）
——模板方法模式（Template Method）
——访问者模式（Visitor）

有关解决方案请见附录 A。

第 2 部分

类之间的异味

第8章 数据

第9章 继承

第10章 职责

第11章 相关改变

第12章 库类

补充点4 重构构成形式

第 8 章

数据

数据（Data）可以定义为简单的事实，和与它们相关的信息区别开。“数据”的概念有些模糊，究竟是数据处理（data processing）还是数据结构（data structure），对此存在一个概念往复。

显然，数据往往是考虑问题的出发点。例如，我们知道每个人都有自己的名和姓，因此可以利用这一信息来创建一个 Person 类。但是对象（Object）则要同时考虑数据（data）和行为（behavior），如果通过行为来组织对象，那么代码会更健壮。

面向数据的对象为异味提供了可乘之机。本章所介绍的异味一般意味着缺失类，或者类的构造不当。如果数据代表一个很好的聚类（clustering），那么我们通常能够找出属于该类的行为。

8.1 所涉及的异味

- 基本类型困扰（Primitive Obsession）。
- 数据类（Data Class）。
- 数据泥团（Data Clump）。
- 临时字段（Temporary Field）。

8.2 基本类型困扰

症状

查看是否存在以下情况：

- 使用了基本类型或接近基本类型的类型（int，float，String，等等）。
- 存在表示小整数的常量和枚举。
- 存在表示字段名的字符串常量。

原因

造成基本类型使用过度有多种原因：

- 缺失类（Missing class）：因为几乎所有数据都会在某处以基本类型出现，所以很容易以一个基本类型作为起点，而错过了引入一个新类的机会。
- 模拟类型（Simulated type）：有些基本类型被用作类型码，实际上就是在模拟对象。一个类最初可能只有一种行为，随着类的发展，可能需要选择一个布尔值来指出使用两种行为中的哪一种；接着可能会有一个由小值组成的枚举，需要指出在多种行为中选择使用哪个。不过，此时很可能隐藏了一个实际需求，即需要多个类。
- 模拟字段访问函数（Simulated field accessor）：有时基本类型会被作为一个索引来访问数组中的伪字段（Pseudo field）。在哈希表（Hash Table）或映射（Map）中，字符串通常就是这样使用的。

采取的措施

对于缺失对象（Missing Object）：

- 请参见本章的“数据泥团（Data Clump）”一节，解决数据泥团问题后通常可以封装基本类型。
- 通过将数据值替换为对象，建立首类（first-class）数据值。

对于模拟类型，一个整型类型码对应一个类：

- 对于该类型码，如果行为没有条件，则它更像是一个枚举，因此可以将类型码替换为类。
- 如果此类型码是不可变的，而且该类还没有子类，则将类型码替换为子类。
- 如果类型码改变，或者类已经有子类了，就将类型码替换为状态/策略（Replace Type Code with State/Strategy）。在 Fowler 的《重构》一书中，

在描述将条件式替换为多态（Replace Conditional with Polymorphism）时，提供了此变换的一个例子。

对于模拟字段访问函数：

- 如果基本类型专门用于处理某些数组元素，就将数组替换为对象。

收益

可以增强表述能力。可能会暴露重复性问题，但能够提高灵活性。通常能表明还需要使用其他重构技术。

不适用的情况

- 在本应使用对象的位置使用了基本类型（相应地也缺失了对象），这种情况相当常见，因此我迟疑再三，不知是否有必要提供不引入对象的理由。有时你可能会因为依赖性或性能问题而不去处理这种异味。
- 有时可以将字段的名字用作标记，从而使用映射而不是固定字段的对象。这样可以减少一个简单对象结构的耦合度，不过会稍微影响性能、类型安全性和简洁性。

说明

- 当 ArrayList（或其他一些通用结构）被滥用时，会出现这类紧密相关问题。

习题 24　可替代表示。

对以下各种类型，提供两至三种可替代表示。

金额：

位置（在一个列表中）：

范围：

社会保险号（身份证号为“123-45-6789”）：

电话号码：

街区地址（“123 E.Main Street”）：

ZIP code（邮政编码）：

有关解决方案请见附录 A。

习题 25　讨论。

假设有一个商业应用程序，用户在其中输入一个邮政编码（以及其他一些内容），并将其保存在一个关系数据库中。有人认为："没有必要费力地将它转换为一个对象。在真正编写时，肯定还要将它重新转换为一个基本类型。"既然要做两次转换，那为什么还是有必要创建对象呢？

有关解决方案请见附录 A。

习题 26　迭代器。

迭代器（Iterator）或枚举器（Enumerator）如何减少基本类型困扰？

有关解决方案请见附录 A。

习题 27　编辑器。

考虑这样一个编辑器接口：

```
public class Editor {
   public void insert(String text) {...}
   public String fetch(int numberOfCharactersToFetch) {...}
   public void moveTo(int position) {...}
   public int position() {...}
   //还有很多，在此忽略
}
```

存在以下调用序列：

```
   editor.insert("ba(nana)");
   int firstParendPosition = 2;
   editor.moveTo(firstParendPosition);
   assertEquals("(", editor.fetch(1));

   editor.moveTo(1);
   editor.insert("x");   // Now: bxa(nana)
   editor.moveTo(firstParendPosition);
   assertEquals(___, editor.fetch(1));
```

A. 如果提供上面的接口，你希望用什么字符串来填入___？

B. 对于变量名（firstParendPosition），你希望改用什么样的字符串？它的作用是什么？

C. 问题的关键在于 int 被作为一个位置索引使用，请给出另外一种方法。

D. 将你的解决方案与备忘录（Memento）设计模式（参见 Gamma 的《设计模式》一书）进行比较。

有关解决方案请见附录 A。

8.3 数据类

症状

- 类仅由公共数据成员组成，或者只包含简单的获取方法和设置方法。这样客户就要依赖此类的可变性和表示形式了。

原因

类通常是这样形成的：你意识到某些数据是某个独立对象的一部分，因此将它们抽取出来形成一个类。但是对象所考虑的是行为（behavior）的共性，而这些对象尚未充分开发，它们还没有多少行为。

采取的措施

1. 使用封装字段（Encapsulate Field）防止直接访问字段（仅允许通过获取方法和设置方法进行访问）。

2. 对方法尽可能采用移除设置方法（Remove Setting Methods）。

3. 使用封装集合（Encapsulate Collection）防止直接访问任何集合类型的字段。

4. 查看对象的各个客户。你会发现，尽管类已经为客户提供了访问和处理字段的功能，但客户仍会访问字段并处理结果（这往往会导致重复，因为多个调用者可能会对数据执行相同的操作）。可以对客户使用抽取方法，取出与类相关的代码，然后采用搬移方法将其置于类中。

5. 在完成上述工作后，你可能会发现类中还有多个相似的方法。使用重命名方法、抽取方法、增加参数或移除参数等重构技术来协调签名并消除重复。

6. 对字段的大多数访问都不再需要了，因为搬移的方法涵盖了实际使用。此时便可以使用隐藏方法（Hide Method）来消除对获取方法和设置方法的访问。（你可能还是认为应对其进行私有访问，并让所有内部访问均通过这些获取方法和设置方法来完成。）

收益

可以增强表述能力，可能会暴露重复性问题（因为通常会发现客户在以类似的方式处理字段）。

不适用的情况

有时封装字段会牺牲一定的性能。例如一个坐标为 x 和 y 的点。接口（可能）不会改变，而人们往往会处理多个点。因此，在 Java 中，Point 类的字段是公共的（这样可以降低针对每次访问的过程调用开销）。

有些持久性机制依赖于反射（reflection）或获取/设置方法来查看字段，以确定应当加载或存储什么数据。对于这些类，可能无法消除它们的数据类本性。如果坚持这样做，我会试着将这些类看做备忘录（Mementos，请参见 Gamma 的《设计模式》一书）。有时我会使用另一个类来作为这些仅基于持久性的类上的一层。这个新类可以充分利用上述所有修改，而且它会隐藏底层的类。

习题 28　库类。

请比较以下库类，它们有什么共同点？

```
java.awt.Event
java.awt.GridBagConstraints
java.awt.Insets
java.awt.Point
java.awt.Polygon
java.awt.Rectangle
```

有关解决方案请见附录 A。

习题 29　Color 和 Date。

类 java.awt.Color 和 java.util.Date 是封装类的例子，在此，对成员的访问只能通过方法完成。

A. 请为各个类提出至少两种内部表示。

B. 不能直接访问成员对于增强类的不可变性有什么作用？

有关解决方案请见附录 A。

习题 30　适当的名字（有难度）。

A. 清理下面这个 Data 类。（可在 www.xp123.com/rwb 在线获得）

Person.java

```
public class Person {
    public String last;
    public String first;
    public String middle;

    public Person(String last, String first, String middle) {
        this.last = last;
        this.first = first;
        this.middle = middle;
    }
}
```

PersonClient.java

```
// 为方便起见，将客户放在一个文件中；
// 可以把它们想象成单独客户类中的非测试方法。

import junit.framework.TestCase;

import java.io.*

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

    public void client1(Writer out, Person person)
        throws IOException {
        out.write(person.first);
        out.write(" ");
        if (person.middle != null) {
            out.write(person.middle);
            out.write(" ");
        }
        out.write(person.last);
    }

    public String client2(Person person) {
        String result = person.last + ", " + person.first;
        if (person.middle != null)
            result += " " + person.middle;
        return result;
    }
```

```
    public void client3(Writer out, Person person)
        throws IOException {
        out.write(person.last);
        out.write(", ");
        out.write(person.first);
        if (person.middle != null) {
            out.write(" ");
            out.write(person.middle);
        }
    }

    public String client4(Person person) {
        return person.last + ", " +
              person.first +
              ((person.middle == null) ? "" : " " + person.middle);
    }

    public void testClients() throws IOException {
        Person bobSmith = new Person("Smith", "Bob", null);
        Person jennyJJones = new Person("Jones", "Jenny", "J");
        StringWriter out = new StringWriter();
        client1(out, bobSmith);
        assertEquals("Bob Smith", out.toString());

        out = new StringWriter();
        client1(out, jennyJJones);
        assertEquals("Jenny J Jones", out.toString());

        assertEquals("Smith, Bob", client2(bobSmith));
        assertEquals("Jones, Jenny J", client2(jennyJJones));

        out = new StringWriter();
        client3(out, bobSmith);
        assertEquals("Smith, Bob", out.toString());

        out = new StringWriter();
        client3(out, jennyJJones);
        assertEquals("Jones, Jenny J", out.toString());

        assertEquals("Smith, Bob", client4(bobSmith));
        assertEquals("Jones, Jenny J", client4(jennyJJones));
    }
}
```

B. 现在有一个新的需求，要求支持单名（比如说 Cher 或 Madonna），或者允许有多个姓（Oacar de los Santos）。比较在对前一部分应用重构技术之前和之后做这种改变的难易程度。

8.4 数据泥团

症状

- 两到三个同样的项频繁地一同出现在类和参数表中。
- 代码声明了某些字段及处理这些字段的方法，然后又声明了更多的字段或更多的方法，且会一直持续下去（也就是说，在类中同时存在多组字段和方法）。
- 各组字段名以类似的子字符串开头或结尾。

原因

这些项通常是另外某个实体的一部分，但到目前为止，还没有人意识到缺失类。或者人们已经知道缺失了类，但是认为这个类太小或者不重要，不必独立存在。

（识别这些类通常是简化系统的一个主要步骤，而且常常能帮助你更容易地泛化类）。

采取的措施

- 如果项是类中的字段，则使用抽取类将其抽取到一个新的类中。
- 如果值同时出现在方法签名中，则使用引入参数对象（Introduce Parameter Object）来抽取新的对象。
- 查看由新对象传递这些项的调用，看看它们是否能改用保持对象完整（Preserve Whole Object）。
- 查看这些项的使用。通常可以利用搬移方法等重构技术将这些使用移至新的对象中（与解决异味数据类的做法类似）。

收益

可以增强表述能力。还可能暴露重复性问题，通常会降低规模。

不适用的情况

有时，传递一个完整的对象会引入你不需要的依赖性（因为较低层次的

类会对整个新对象而不是其值公开），因此可以进行部分传递来避免这种依赖性。

在一些极少见的情况下，可通过传递部分对象而不是对象本身来解决可测量的性能问题。注意，这是对象模型为性能所做的一个让步，且有必要为这种代码增加注释。

8.5 临时字段

症状

- 字段仅在某些时候进行设置，而在其余时间为 Null（或不使用）。

原因

当对象中的某一部分存在一个算法，而该算法要通过字段而非参数来传递信息时，就会发生这种情况。这些字段仅在算法处于活动状态时才有效或得到使用。字段有时用有时不用，说明可能缺失了一个对象，其生命周期与保存它的对象的生命周期有所不同。

采取的措施

- 抽取类，移出字段及所有相关代码。

收益

增强了表述能力并提高了清晰性。可能会减少重复，特别是在其他位置可以使用新类时。

不适用的情况

如果新对象不是表示一个有用的抽象，那么就没必要费力创建这样一个对象了。

第 9 章

继承

类与其子类之间的关系开始时往往很简单，但是随着时间的推移会变得越来越复杂。相比一个单独的类，子类通常会更紧密地依赖它的父类，但有时会太过于紧密。

9.1 所涉及的异味

- 拒收的遗赠（Refused Bequest）。
- 不当的紧密性（子类形式）[Inappropriate Intimacy (Subclass Form)]。
- 懒惰类（Lazy Class）。

另请参见其他一些与继承有关的异味：

- 模拟继承（Switch 语句），第 7 章。
- 并行继承体系（Parallel Inheritance Hierarchies），第 11 章。
- 组合爆炸（Combinatorial Explosion），第 11 章。

9.2 拒收的遗赠

症状

- 一个类继承自其父类，但是抛出了一个异常而不是支持一个方法，这是坦率的拒绝（honest refusal）。
- 一个类继承自其父类，但是在调用这个类时，某个继承的方法不能正常工作，这是隐式的拒绝（implicit refusal）。

- 客户试图通过子类的句柄而不是父类的句柄来访问类。
- 继承没有什么实际意义，子类并不是父类的一个例子。

原因

某个类之所以继承自另一个类，可能只是为了实现方便，而不是真的想用这个类来取代其父类。也可能是因为无法确定是否应该让子类拒绝使用父类的某些特性，以此避免所有特性组合可能造成的类型爆炸。

采取的措施

- 如果不会导致混淆，可以不去管它。
- 如果找不到共享某个类关系的理由，就将继承替换为委托（Replace Inheritance with Delegation）。
- 如果父—子关系确实有意义，则可以通过抽取子类（Extract Subclass）、下移字段（Push Down Field）和下移方法（Push Down Method）来创建一个新的子类，见图 9.1。让这个类拥有非拒绝行为，并将父类的客户修改为这个新类的客户，这样父类就不必再提及此特性了。你可以从原来的类及其父类中去除这些被拒绝的方法。

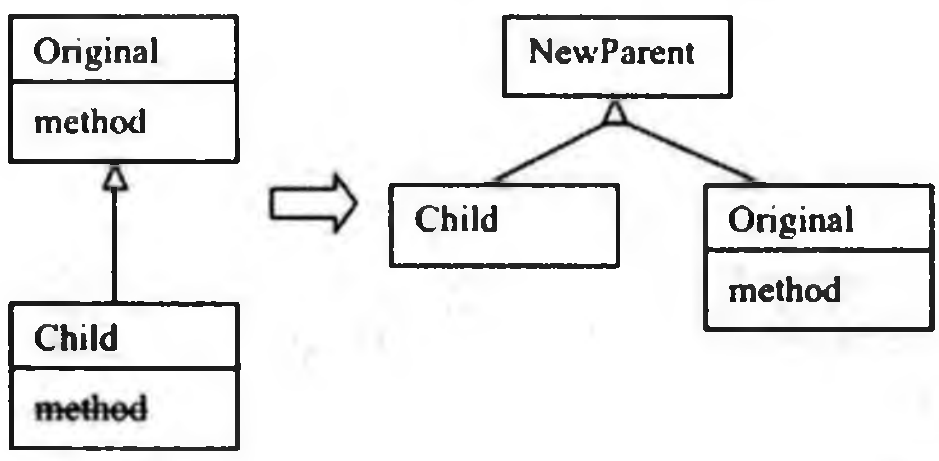


图 9.1　重组继承体系

收益

可增强表述能力，能改善可测试性（因为无需担心类继承体系中各处的不同功能）。

不适用的情况

有时拒收的遗赠可用于避免新类型的爆炸（通过拒绝特定的方法，就不需要为各种拒绝组合创建一种类型继承体系了）。

说明

有一条被称为 Liskov 替换原则（Liskov Substitution Principle，LSP）的指导原则，它指出：一个类的所有子类都应当能被看做该类的例子。而拒绝父类的方法则违背了这一原则。Eiffel 语言在使用前置条件和后置条件时就体现了这个原则：方法可以降低需求，但它必须保证与其父方法相当。请参见 Robert Martin 所著的 *Agile Software Development: Principles, Patterns, and Practices*，了解关于 LSP 的更多信息。

习题31 集合类。

Java 中的集合类并没有为只读集合提供一个单独的继承体系，它只是提供了一个“包装器（wrapper）”，用于拒绝对其底层集合做出的修改。

A. 此时可以应用哪种方法来发现这一点？

B. 为什么使用了只读集合的类不用捕获或声明异常呢？

C. 你认为库设计者所采用的这种方法可行吗？你认为还可以采用另外哪些方法？

有关解决方案请见附录 A。

习题32 拒收遗赠。

请在你所访问的代码中找出一个拒收遗赠的例子。

9.3 不当的紧密性（子类形式）

症状

- 一个类访问了它父类的内部（本应是私有的）部分。[各类之间也有一种类似的不当紧密性形式，请参见第 10 章中的“不当的紧密性（一般形式）”一节]。

原因

相对于两个“陌生”类来说，父类与子类之间当然会亲密一些。不过有时它们太过亲密了（即过度耦合）。

采取的措施

- 如果子类以一种非受控方式访问父类的字段，则使用自封装字段（Self Encapsulate Field）。
- 如果父类可以定义一个子类能插入其中的通用算法，则使用构建模板方法（Form Template Method）。
- 如果父类和子类需要进一步解耦合，则采用将继承替换为委托（Replace Inheritance with Delegation）方法。

收益

可以减少重复。通常能够增强表述能力，还可能会降低规模。

不适用的情况

尚未发现。

9.4 懒惰类

症状

- 类没有做什么工作，似乎是它的父类、子类或者调用者完成了所有相关工作，而且这个类中没有足够的行为，往往会让人们质疑是否有留它的必要。

原因

这通常是因为在重构过程中，类的所有职责都移到了其他位置。

采取的措施

- 如果一个类的父类或子类更适合用来完成该类的行为，则通过折叠继承体系（Collapse Hierarchy）将该类与其父类或子类合并。
- 否则，通过内联类（Inline Class）将其行为合并到它的调用者中。

收益

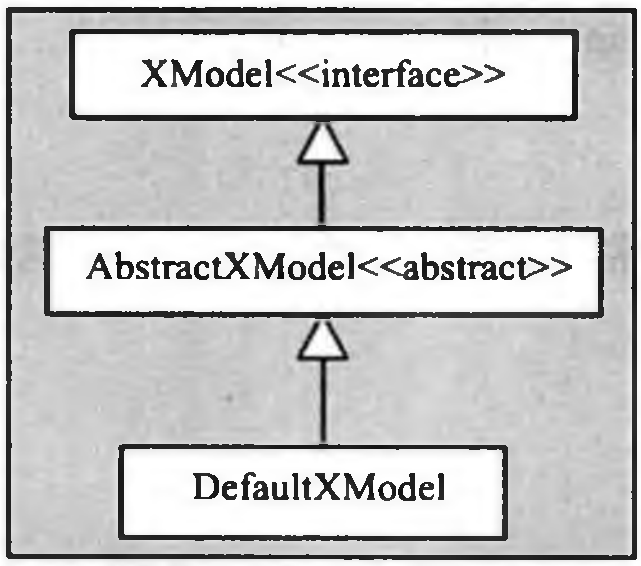
可降低规模。能增强表述能力，代码更简单。

不适用的情况

- 有时使用懒惰类是为了传达意图，你必须在表述能力和简单性之间进行权衡。

习题33　Swing库（有难度）。

Swing 库有如下模式：



AbstractXModel 是此接口的唯一实现，DefaultXModel 是 AbstractXModel 的唯一子类（抽象模型一般有内置的通知处理，不过它可能还支持其他功能）。

A. 让模型作为一个接口与 AbstractXModel 类分离有什么好处？

B. 为什么这不是一个懒惰类示例？

C. 你的对象结构可以采用这种 3 个类的结构吗？

有关解决方案请见附录 A。

第 10 章

职责

要正确平衡对象之间的职责很难。重构的优点之一正是在于它使得我们可以以一种安全的方式尝试各种不同的想法，也就是说，我们可以大胆地改变想法并将它付诸实践。

有一些工具可以帮助我们确定对象应当如何协同工作，即设计模式和 CRC 插卡（请参见 www.c2.com/cgi/wiki?CrcCards）。

重构通常是可逆的，而且通常需要在两个好的方面进行权衡。关于这一点，消息链（Message Chains）对中间人（Middle Man）是一个非常好的例子。有时可能存在一种方法能同时改善两种异味，但大多数情况下都需要在不同异味之间进行权衡。

10.1 所涉及的异味

- 依恋情结（Feature Envy）。
- 不当的紧密性（一般形式）[Inappropriate Intimacy (General Form)]。
- 消息链（Message Chains）。
- 中间人（Middle Man）。

10.2 依恋情结

症状

- 一个方法似乎过于强调处理其他类的数据，而不是它自己的数据。（你

可能会因为重复性注意到这一点，即多个客户在做相同的操作；或者一个对象在一行中出现了好几次。）

原因

对于当前的数据类和原来的数据类而言，客户之间出现这种情况相当常见，任何类及其客户之间都有可能出现这种现象。

采取的措施

- 使用搬移方法将动作放在适当的类中（可能必须先使用抽取方法分离出位置不当的部分）。

收益

可减少重复。通常会增强表述能力（因为代码已经移到了预期的位置），还可能会暴露更多需要重构的地方。

不适用的情况

有时某些行为被故意放在“错误”的类中。例如，一些设计模式[如策略模式（Strategy）或访问者模式（Visitor）]就将行为置于一个单独的类中，使行为可以被单独更改。如果采用搬移方法将此行为置回原处，就会把本应单独修改的内容混在一起。

说明

区分依恋情结和不当的紧密性（本章后面将介绍）有时并不简单。Industrial Logic 的 Joshua Kerievsky 指出：“依恋情结是指，一个类自身所做甚少，需要借助大量其他类才能完成自己的工作。不当的紧密性则是指，一个类为了访问某些本不该访问的内容，过于深入到其他类中。”

习题34　依恋情结。

回顾一下第 3 章的习题 4。在 Report.report()中，注意输出信息是如何通过查看 Machine 或 Robot 的字段得到的。请修正这两个依恋情结示例。

有关解决方案请见附录 A。

10.3 不当的紧密性（一般形式）

症状

- 一个类访问了另一个类的内部部分（而这一部分本应是私有的）。[子类及其父类之间也有这样一种类似的不当紧密性形式，请参见第 9 章中的“不当的紧密性（子类形式）”一节。]

原因

两个类之间有时可能会稍有关联。等你意识到存在问题时，这两个类已经过于耦合了。它们之间可能缺失一个调解类。相比父类和子类之间，如果彼此无关的类之间出现这个问题，那么情况会更严重。

采取的措施

- 如果两个独立的类彼此“纠缠”，则使用搬移方法和搬移字段将适当的部分放在适当的类中。
- 如果纠缠的部分看上去是一个被遗漏的类，则使用抽取类和隐藏委托（Hide Delegate）引入这个新类。
- 如果类相互指向对方，则使用将双向引用改为单向（Change Bidirectional Reference to Unidirectional）方法使之成为一种单向依赖关系。
- 如果子类与父类过于耦合[如第 9 章中“不当的紧密性（子类形式）”一节所述]，有以下几种情况。
  - 如果子类以一种非受控方式访问父类的字段，则使用自封装字段（Self Encapsulate Field）。
  - 如果父类可以定义子类能够插入其中的通用算法，则使用构建模板方法（Form Template Method）。
  - 如果父类和子类需要进一步解耦合，则使用将继承替换为委托（Replace Inheritance with Delegation）方法。

收益

可以减少重复。通常能够增强表述能力，还可能会降低规模。

不适用的情况

尚未发现。

10.4 消息链

症状

你可能看过如下形式的调用：

```
a.b().c().d()
```

（这可能会直接发生，也可能体现在中间结果中。）

原因

一个对象必须与其他对象协作才能完成工作，这是自然，问题在于这不仅会使对象相互耦合，还会使获得这些对象的路径存在耦合。这种耦合与面向对象程序设计所恪守的两个座右铭正好背道而驰，即“尽管吩咐，不要问（Tell, Don't Ask）”和 Demeter 法则（Law of Demeter）。“尽管吩咐，不要问”是指，不要寻找对象并处理它，而应吩咐对象完成你所要求的操作。Demeter 法则则可以更清晰地表述为：方法不应与“陌生人”说话，也就是说，它应当只与其自身、其参数、它自己的字段或者它创建的对象有信息传递。（Andrew Hunt 和 David Thomas 合著的 *The Pragmatic Programmer* 一书对这两条规则提供了更详细的描述。）

采取的措施

- 如果处理实际针对的是目标对象（即消息链最末端的对象），则使用抽取方法和搬移方法将处理置于该对象中。
- 使用隐藏委托使方法仅依赖于一个对象（因此，不要采用 a.b().c().d() 的形式，而应将 d()方法置于 a 对象中。这可能还需要为 b()和 c()对象增加一个 d()方法）。

收益

可以减少重复或者暴露重复性问题。

不适用的情况

这是一种折衷的重构技术。如果过分应用隐藏委托，那么对象都去忙着委托，就会没有一个真正做实事的。有时，调用一个小消息链可能更简单，而且也不太会造成混淆。

10.5 中间人

症状

- 类的大多数方法都是在调用另一个对象的同一个（或类似的）方法：

```
f() {delegate.f();}
```

（如果一个类的任务主要是将工作委托出去，这样的类就称为中间人）。

原因

首先，这可能是因为应用了隐藏委托来解决消息链引起的。可能在此之后其他一些特性已经被移除了，剩下的主要就是委托方法了。

采取的措施

- 一般来说，可以通过让客户直接调用委托来移除中间人（Remove Middle Man）。
- 如果委托被中间人所有，或者是不可变的，而且中间人还有增加的行为，那么此中间人可以看做委托的一个示例，此时可以将委托替换为继承（Replace Delegation with Inheritance）。

收益

可降低规模，还可能增强表述能力。

不适用的情况

- 有些模式[如代理模式（Proxy）和装饰器模式（Decorator）]会有意地创建委托。如果中间人确实有充分存在的理由，则不要将其移除。
- 要在中间人和消息链之间权衡。

- 委托提供了一种外观（facade），使得调用者可以不用了解消息和结构的细节。移除中间人则可能会使客户获得一些本不应知道的信息。

10.6 实战练习

习题35　遍历列表。

请考虑第 3 章中习题 4 所用的代码。

A. 遍历机器列表为什么是一种依恋情结或不当的紧密性？

B. 请通过抽取一个新类来解决这个问题。要确保新类有一个 report()方法。

习题36　中间人（有难度）。

请考虑以下类：

（此代码已被大大简化，队列中原本不存储字符串）

Queue.java

```
import java.util.ArrayList;

public class Queue {
    ArrayList delegate = new ArrayList();
    public Queue() {}
    public void addRear(String s) {delegate.add(s);}
    public int getSize() {return delegate.size();}

    public String removeFront() {
        String result = delegate.get(0).toString();
        delegate.remove(0);
        return result;
  }
}
```

QueueTest.java（可选）

```
  public void testQ() {
    Queue q = new Queue();
    q.addRear("E1");
    q.addRear("E2");
    assertEquals("E1", q.removeFront());
    assertEquals("E2", q.removeFront());
    assertEquals(0, q.getSize());
}
```

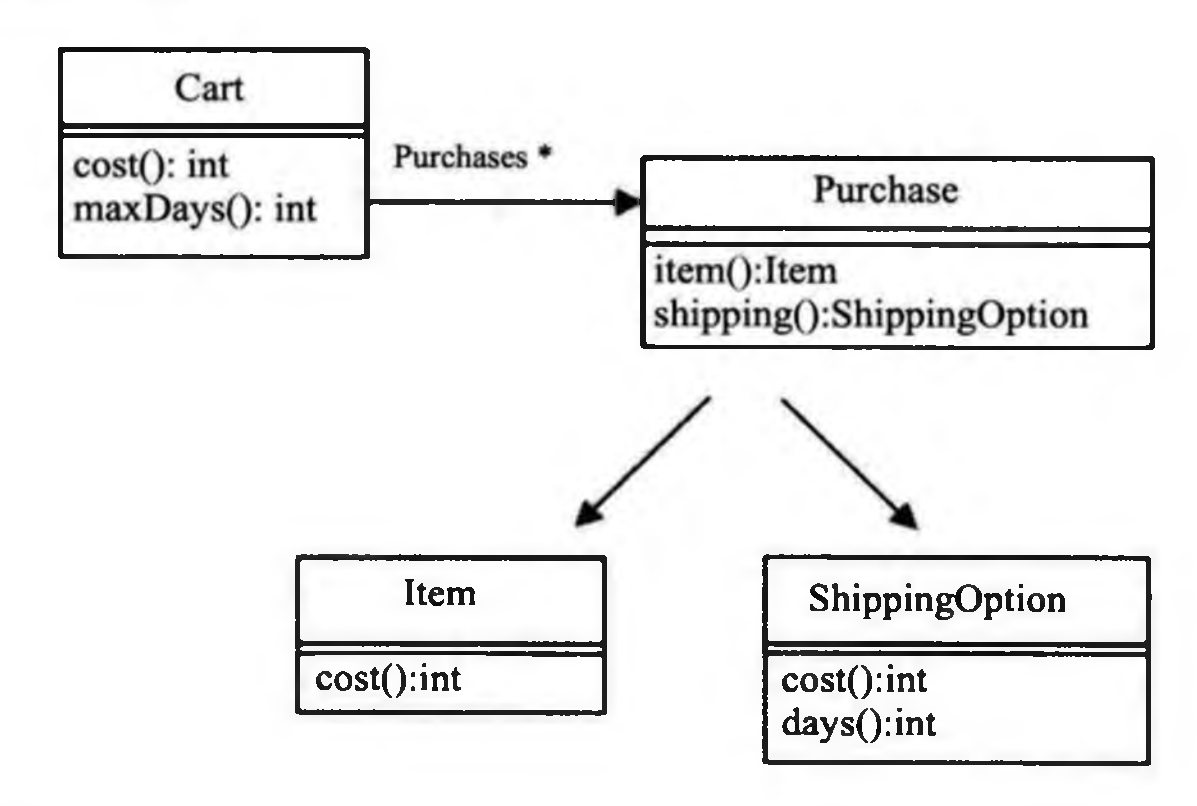
A. 移除中间人，使得 Queue 不再是 ArrayList 的一个中间人。这是一个改进吗？

B. 通过隐藏委托将中间人放回原处。

有关解决方案请见附录 A。

习题37　Cart购物车（有难度）。

请考虑以下类：



Cart.cost()如下：

```
public int cost() {
   int total = 0;
   for (int i=0; i < purchases.size(); i++) {
      Purchase p = (Purchase) purchases.elementAt(i);
      total += p.item().cost + p.shipping().cost;
   }
   return total;
}
```

A. 请写出所示的类（及测试）。maxDays()方法要计算出 Purchase 中所有 ShippingOption 的最大天数。

B. 应用隐藏委托，使 Cart 只直接访问 Purchase。

C. 隐藏委托使中间人（Purchase）类有一个更宽的接口（扩大接口），也就是说，它提供更多的方法。但是应用此重构技术也可以使接口变窄（缩小）。请解释这种明显的矛盾情况。

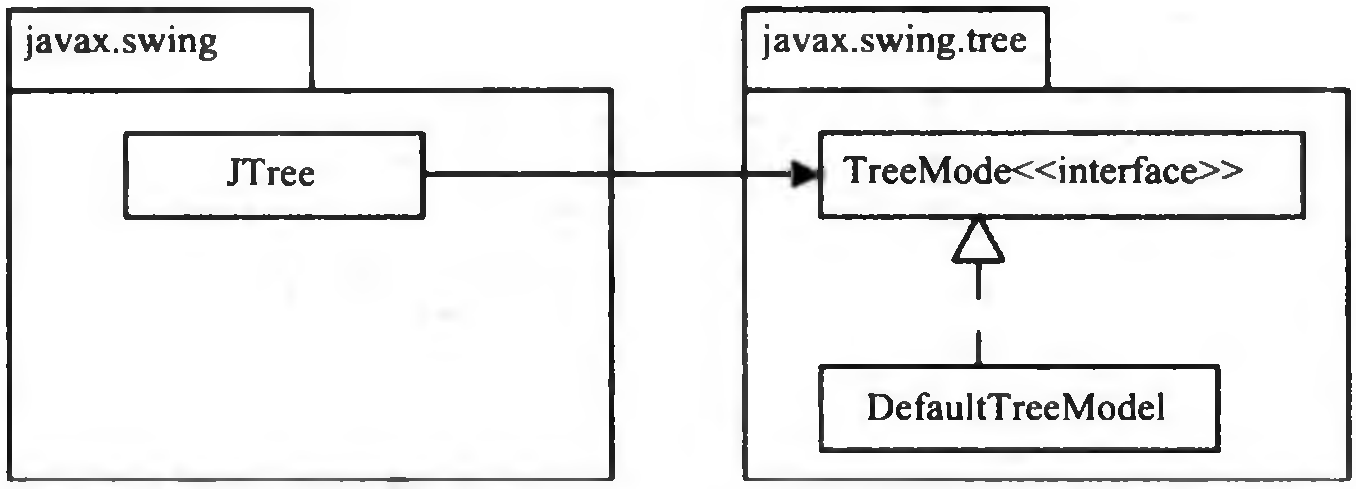
D. 基于以上原因缩小 Purchase 接口。

E. 注意本例中使用了基本类型 int 来表示金额。在委托更改之前或之后引入一个 Money 类是否会更容易？

有关解决方案请见附录 A。

习题38　Swing中的Tree（有难度）。

在 Swing 中，JTree 位于 javax.swing 包中，TreeModel 和 DefaultTreeModel 位于 javax.swing.tree 包中（表类也有类似的组织）。



A. 请说明 TreeModel 接口怎样打破 JTree 和 DefaultTreeModel 之间的依赖性。

B. 请解释如何在包级别仍保存依赖性。

C. 假设每个包对应一个库（这有点过于简单）。请说出一种可以打破包依赖性的方法，从而使 JTree 可以被放在一个独立于 DefaultTreeModel 的库中。

D. 如今的系统往往有非常大的磁盘空间，那为什么还要考虑库依赖性呢？

有关解决方案请见附录 A。

第 11 章

相关改变

只有在试图修改代码时，有些问题才会变得明显（我们讨论过的大多数异味都可以通过静态查看代码检测出来）。

理想情况下，一个修改应该只影响代码中的一处。如果不是这样，则预示着代码中存在重复。

处理这些异味通常还有一个意外收获，即往往会使代码更容易测试。

11.1 所涉及的异味

- 发散式改变（Divergent Change）。
- 霰弹式修改（Shotgun Surgery）。
- 并行继承体系（Parallel Inheritance Hierarchies）。
- 组合爆炸（Combinatorial Explosion）。

11.2 发散式改变

症状

- 尽管每次原因各异，但发现自己所修改的是同一个类（与之相反的是“霰弹式修改”，这是下一个将要介绍的异味）。

原因

类在发展过程中会承担越来越多的职责，但没有人注意到这会涉及两种

截然不同的决策（还记得我们在第 6 章所介绍的 Parnas 名言吗，即模块应当有且仅有一个秘密）。

采取的措施

- 如果类既要找到对象，又要对其做一些处理，则让调用者查找对象并将该对象传入，或者让类返回调用者使用的值。
- 采用抽取类（Extract Class）为不同的决策抽取不同的类。
- 如果几个类共享相同类型的决策，则可以考虑合并这些新类[例如，通过抽取超类（Extract Superclass）或抽取子类（Extract Subclass）]。至少，这些类可以构成一个层（例如，一个持久存储层）。

收益

可增强表述能力（更好地传达意图），还可以提高健壮性以备将来修改。

不适用的情况

尚未发现。

习题 39　CsvWriter。

考虑下面这段编写逗号分隔值（Comma-Separated Value，CSV）文件的代码。（可在 http://www.xp123.com/rwb 在线获得）

CsvWriter.java

```
public class CsvWriter {
    public CsvWriter() {}

    public void write(String[][] lines) {
        for (int i = 0; i < lines.length; i++)
            writeLine(lines[i]);
    }

    private void writeLine(String[] fields) {
        if (fields.length == 0)
            System.out.println();
        else {
            writeField(fields[0]);

            for (int i = 1; i < fields.length; i++) {
                System.out.print(",");
```

```
            writeField(fields[i]);
        }
        System.out.println();
    }
}

private void writeField(String field) {
    if (field.indexOf(',') != -1 || field.indexOf('\"') != -1)
        writeQuoted(field);
    else
        System.out.print(field);
}

private void writeQuoted(String field) {
    System.out.print('\"');
    for (int i = 0; i < field.length(); i++) {
        char c = field.charAt(i);
        if (c == '\"')
            System.out.print("\"\"");
        else
            System.out.print(c);
    }
    System.out.print('\"');
}
}
```

CsvWriterTest.java（一个手动测试）

```
import junit.framework.TestCase;

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

public void testWriter() {
        CsvWriter writer = new CsvWriter();
        String[] [] lines = new String[] [] {
  new String[] {},
  new String[] {"only one field"},
  new String[] {"two", "fields"},
  new String[] {"", "contents", "several words included"},
  new String[] {",", "embedded , commas, included",
                                                "trailing comma,"},
  new String[] {"\"", "embedded \" quotes",
      "multiple \"\"\" quotes\"\""},
  new String[] {"mixed commas, and \"quotes\"", "simple field"}
};
```

```
        // Expected:
        //                    -- (empty line)
        // only one field
        // two,fields
        // ,contents,several words included
        // ",","embedded , commas, included","trailing comma,"
        // """","embedded "" quotes","multiple """""" quotes"""""
        // "mixed commas, and ""quotes""",simple field

        writer.write(lines);
    }
}
```

A. 这段代码为什么是发散式改变的一个例子？（它体现的是什么决策？）

B. 修改此代码，将它写入到一个作为参数传入的流中。

C. 基于原始代码修改函数，返回一个对应于函数所写内容的字符串值。

D. 哪个版本看起来更好，为什么？哪一个版本更易于测试？

有关解决方案请见附录 A。

习题 40 CheckingAccount。

请考虑以下类片段：

```
public class CheckingAccount {
   public int balance() {...}
   public void add(Transaction transaction) {...}
}

public class Transaction {
   public int cost() {...}
}
```

A. 这些类体现了什么决策？

B. 请抽取一个简单的 Money 类。

有关解决方案请见附录 A。

11.3 霰弹式修改

症状

- 仅做一个简单的修改就要更改多个类。

原因

将多个类分离是代码的一大职责。你可能缺失一个通晓全部职责的类（而大量修改本应通过这个类来完成）。另外，也有可能是因为你过度去除发散式改变而导致了这个异味。

采取的措施

- 找出一个负责这些修改的类。这可能是一个现有类，也可能需要你通过应用抽取类来创建一个新类。
- 使用搬移字段（Move Field）和搬移方法（Move Method）将功能置于所选类中。让未选中的类足够简单后，便可以使用内联类（Inline Class）将该类去除了。

收益

可以减少重复，增强表达能力，并能改进可维护性（将来的修改将更为本地化）。

不适用的情况

尚未发现。

习题 41　霰弹式修改。

请在你所访问的代码中找一些存在这个问题的例子。问题很可能出现在以下方面：

- 配置信息。
- 日志记录。
- 持久性存储。
- 完成一件普通的工作有时需要调用一个对象两次，而且这种两步式处理会在多处用到。

11.4 并行继承体系

症状

- 在一个继承体系中建立了一个新子类后，却发现还需要在另一个继承体系中创建一个相关的子类。
- 可能发现两个继承体系中的子类有着相同的前缀（命名可以反映出协调继承体系的需求）。

如前所述，这是霰弹式修改的一个特例。

原因

继承体系可能在以一种并行的方式发展着，即同时需要一个类及它的一对子类（这两个子类分别位于不同的继承体系中）。与往常一样，刚开始时也许没什么问题，但是引入了两对或多对子类之后，再进行修改时就会因为情况过于复杂而无法进行了。（这样两个类一般体现了同一决策的不同方面。）

采取的措施

- 使用搬移字段和搬移方法来重新分配特性，以便能去除某个继承体系。

收益

可以减少重复。可能会增强表述能力，也可能会降低规模。

不适用的情况

尚未发现。（在改善特别复杂的情况时，可能会伴随出现这种异味。）

习题 42　复制受察数据。

复制受察数据（Duplicate Observed Data）将类一分为二：一部分是模型（model），另一部分是视图（view）。例如，可以将 Card 分离为 CardModel 和 CardView。让模型类构成一个继承体系是很自然的做法（它们有类似的通知需求），而让视图构成一个继承体系（它们都用于显示）也是很自然的。这听上去像是一个并行继承体系。这有什么问题吗？

有关解决方案请见附录 A。

11.5 组合爆炸

这与并行继承体系相关，但是所有内容都折叠到了一个继承体系中。

症状

- 你本来希望只引入一个单一的新类，但是最后却发现必须在继承体系的不同位置引入多个类。
- 你发现继承体系的各个层使用了一组常见的词（例如，一层增加了样式信息，下一层增加了可变性）。

原因

原本应当是独立的决策却通过一个继承体系实现了。

采取的措施

- 如果问题不严重，可以采用将继承替换为委托（Replace Inheritance with Delegation）。通过为各种变化提供同样的接口，可以创建一个装饰器设计模式示例。
- 如果情况已经变得相当复杂了，则有必要考虑进行大型重构了，可能还需要梳理并分解继承体系（Tease Apart Inheritance）。有关详细内容请参见 Fowler 的《重构》。

收益

可以减少重复，降低规模。

不适用的情况

尚未发现。

习题 43　文档。

请考虑以下继承体系：

```
Document
   AsciiDocument
      ZippedAsciiDocument
      RawAsciiDocument
```

```
        BriefAsciiDocument
HtmlDocument
   RawHtmlDocument
   ZippedHtmlDocument
MarcDocument
   BriefMarcDocument
   FullMarcDocument
```

A. 增加一个所有文档类型都支持的新压缩类型会有什么影响？

B. 重新组织这个继承体系，使它首先基于压缩（或什么也不基于），然后是简要/详尽选择，之后才是文档类型。这是一种改进吗？

C. 请描述另外一种使用装饰器模式的方法。

第 12 章

库类

目前的应用程序都会用到库类。

库类有时像一把双刃剑，会将我们置于两难的境地。我们希望库有所不同，但我们又不希望更改它。即使有可能改变一个库，也会承受风险：它会影响其他客户，而这意味着我们必须针对库的将来版本重做这样的修改。

12.1 所涉及的异味

- 不完备的库类（Incomplete Library Class）。

12.2 不完备的库类

症状

- 你正在使用一个库类，希望该类具有某种功能，但它没有。如果这是一个正常的类，你可以修改它；但由于它是库的一部分，因此可能无法修改，或者你也不愿意去更改它。

原因

库类的作者未能满足你的要求（或者鉴于其他原因降低了对此需求的支持力度）。

采取的措施

- 弄清类或库的所有者是否会考虑增加你所需要的支持。

- 如果仅仅是一两个方法，可以对库类的客户应用引入外来方法（Introduce Foreign Method）（这仍存在依恋情结，对此我们也无可奈何）。
- 如果有多个方法需要增加，则应引入本地扩展（Introduce Local Extension）（为该库类建立子类来创建一个新的伪库类），然后使用这个新的扩展类继续操作。
- 可能需要引入一个层来覆盖这个库。

收益

可以减少重复（即可以重用库代码，而不是完全从头开始实现它）。

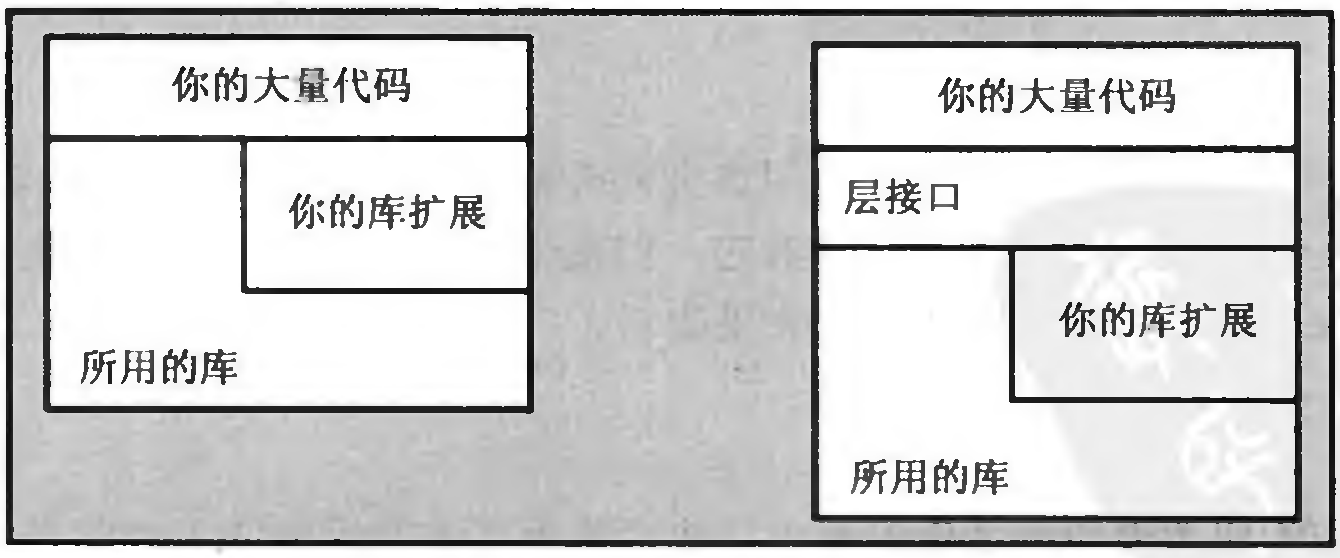
不适用的情况

如果有多个项目，每个项目都使用不兼容的方式来扩展一个类，那么在改变库时就会导致额外的工作。

12.3 实战练习

习题 44 层（有难度）。

处理库的一种方法是将它放在某个层的下面，这样就可以把大量代码独立出来，而不会直接依赖于其他一些库。请考虑以下两种做法：



A. 将上图重新绘制成一个显示依赖关系的 UML 包图。

B. 在这些情况下，你编写的大量代码如何依赖于（或不依赖于）库代码，请予以说明。

C. 这种加层的做法是否具有以下特性：

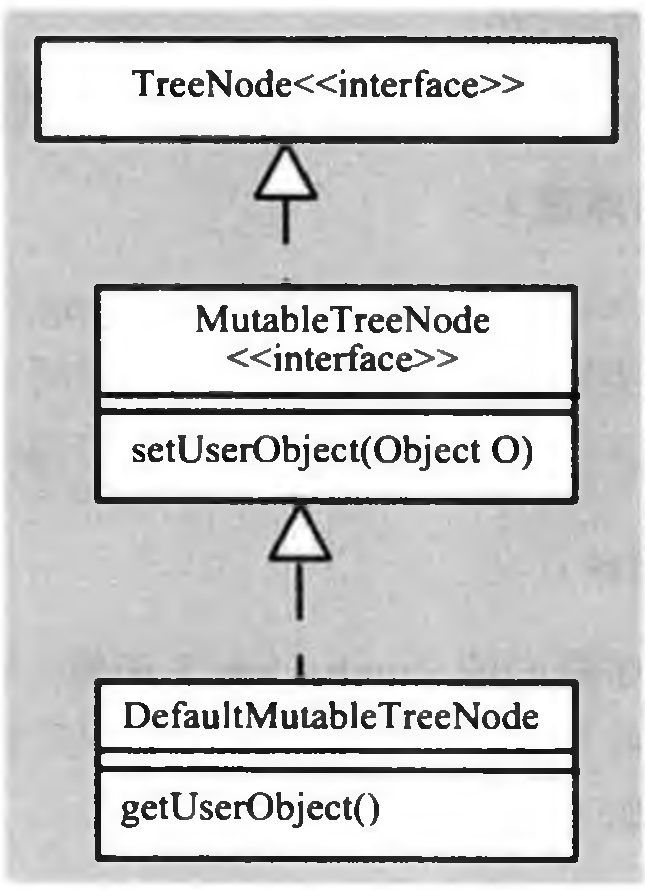
- 概念完整性？
- 可移植性？
- 性能改进？
- 便于测试？

D. 哪些机制可以增强这种加层（也就是说，可以避免有人将第二种方法变成第一种方法）？

有关解决方案请见附录 A。

习题 45　树。

Swing 树库（请见 javax.swing.tree.*）有一个很有意思的缺陷。这个库有一种思想，即 DefaultTreeModel（类）基于 MutableTreeNode（接口）对象建立。尽管存在一个 DefaultMutableTreeNode 类，但如果你已经有了类似节点的对象，可能并不打算用这样一个 DefaultMutableTreeNode 类。（你可能更倾向于实现 MutableTreeNode 接口，而不是将你的信息复制到一个 DefaultMutableTreeNode 中。）



这个库的缺陷就在于 MutableTreeNode 接口中有一个缺口：它支持方法 setUserObject()，但是却没有方法 getUserObject()。

假设你希望有此方法（getUserObject()），你会怎么做？

A. 是采用引入外来方法合适，还是引入本地扩展更合适？为什么？

有关解决方案请见附录 A。

习题 46　字符串。

A. Java 中的 String 类被声明为 final，这对引入本地扩展有什么影响？

B. 为什么 Java 的发明者会将 String 设为 final？

有关解决方案请见附录 A。

习题 47　过滤器（有难度）。

java.util 中的 Collections 类定义了许多实用方法，包括一个为集合生成枚举的方法。这个枚举器（enumerator）提供了一个包含所有元素的列表，但没有提供过滤它们的方法。

A. 请编写一个类来包装 Enumerator，然后扩展它，使它能返回符合一定条件的对象（包括测试）。

B. 这个类有哪些父类或接口（如果有的话）？

C. 构造函数有哪些参数（如果有的话）？

D. 这个类要定义哪些方法？

有关解决方案请见附录 A。

习题 48　图表（有难度）。

引入外来方法、引入本地扩展、加层（layering）和包装（wrapping）之间有什么区别，请画出类图或其他 UML 图进行说明。

有关解决方案请见附录 A。

习题 49　缺失的函数。

请考虑作为应用程序核心的 Zumbacker Z 函数。[实际上，在应用领域中这个函数相当常用，因此你可能会奇怪 Java 的数学（Math）库中怎么没有这个函数。] 此函数定义如下：

```
Z(x) = abs[ cos(x) + sin(x) - exp(x) ]
```

java.lang.Math 是一个不完备的库，你将如何解决这个问题？

有关解决方案请见附录 A。

补充点 4

重构构成形式

一些重构技术形式为“动词+名词”。

请考虑以下动词：抽取（Extract）、内联（Inline）；搬移（Move）；重命名（Rename）；上移（Pull Up）、下移（Push Down）；隐藏（Hide）、暴露（Expose）。

再请考虑以下名词：字段（Field）、方法（Method）；类（Class）、接口（Interface）、子类（Subclass）、超类（Superclass）；继承体系（Hierarchy）。

补充点 14.1 动词和名词

填写下表，用横线（—）表示没有意义的项，用加号（+）表示 Fowler 的《重构》名录中列出的重构技术，用星号（*）表示有意义但未列在重构名录中的重构技术。

	抽取	内联	搬移	重命名	上移	下移	隐藏	暴露
字段								
方法								
类								
接口								
子类								
超类								
继承体系								

有关解决方案请见附录 A。

第 3 部分

待重构的程序

第 13 章　一个数据库例子
第 14 章　一个简单的游戏
第 15 章　名录
第 16 章　计划游戏模拟器
第 17 章　下一步何去何从

第 13 章

一个数据库例子

R eggie 是一个供小规模学校管理选课的系统。

开发人员使用一个数据库来维护有关学生和班级的信息。此数据库的第一个版本很简单（如图 13.1 所示）。

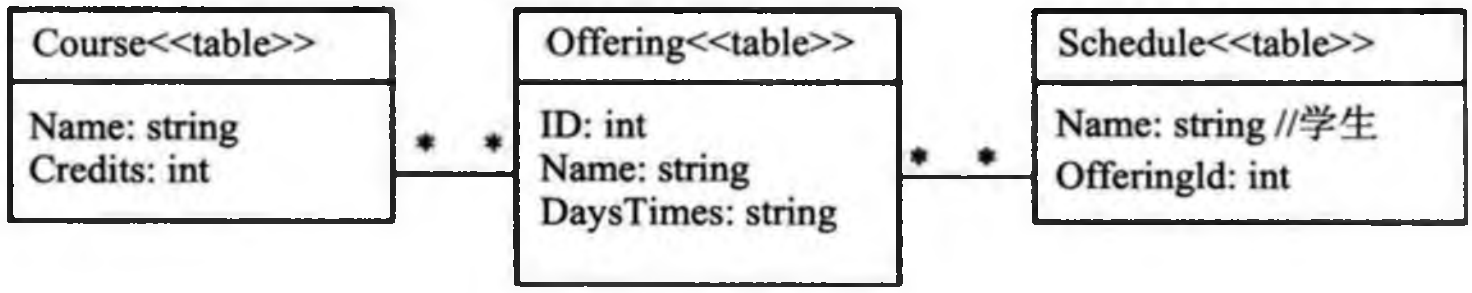


图 13.1　学校选课数据库（第一版）

课程（Course）就是一个班级（class）讲授的科目。此时我们只使用了简单的名字 Econ101 对应“经济学导论（Introduction to Economics）”。之后，我们会将它扩展为包含一个全名、描述及课程的其他一些相关信息。

授课（Offering）是一种授课模式，根据某一课程计划进行教授。授课时间（DaysTimes）是一个包含日期和时间的字符串（用逗号分隔，如 M10, T11, F10）。同样地，在开发过程中也会为这些对象增加更多信息。

排课（Schedule）是一个学生所选的一组特定课程。排课需遵守以下规则：

- 各门课程加起来至少要有 12 个学分。
- 最多有 18 个学分，除非特别允许。
- 课程在时间安排上不存在冲突。
- 不存在重复课程。

这里我们不通过数据库来确保满足这些规则，而是通过检查排课代码来实现。

习题50 数据异味。

重构技术处理的主要是代码异味，不过也存在着数据异味。数据库的一致性能反映出好的数据设计应包括哪些要素。

A. 你发现这个数据库结构中存在哪些潜在问题？

B. 怎样修改数据库可以解决这些问题？（不过现在不要修改）。

有关解决方案请见附录 A。

在下面的代码中有三个类，它们分别对应于上面三个表。开始时它们都很小，可能是因为没有太关注数据，但是随着数据库代码的增加，这些类将变得非常大。有关 Java 数据库程序设计，请参见 Java 教程中的 JDBC 数据库访问指导（java.sun.com/docs/books/tutorial/jdbc）或 JDBC 简易教程（developer.java.sun.com/developer/onlineTraining/Database/JDBCShortCourse）。这段代码使用了 JDBC-ODBC 驱动程序来与文本文件数据库进行交互。

13.1 Course.java

可在 www.xp123.com/rwb 在线获得。

```
import java.sql.*;

public class Course {
    private String name;
    private int credits;
    static String url = "jdbc:odbc:Reggie";

    static { try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            } catch (Exception ignored) {} }

    public static Course create(String name, int credits)
        throws Exception
    {
      Connection conn = null;

      try {
          conn = DriverManager.getConnection(url, "", "");
```

```
        Statement statement = conn.createStatement();
        statement.executeUpdate(
            "DELETE FROM course WHERE name = '" + name + "';");
        statement.executeUpdate(
            "INSERT INTO course VALUES ('" + name
            + "', '" + credits + "');");
        return new Course(name, credits);
    } finally {
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public static Course find(String name) {
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();
        ResultSet result = statement.executeQuery(
            "SELECT * FROM course WHERE Name = '" + name + "';");
        if (!result.next()) return null;

        int credits = result.getInt("Credits");
        return new Course(name, credits);
    } catch (Exception ex) {
        return null;
    } finally {
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public void update() throws Exception {
    Connection conn = null;

    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();

        statement.executeUpdate(
            "DELETE FROM COURSE WHERE name = '" + name + "';");
        statement.executeUpdate(
            "INSERT INTO course VALUES('" +
            name + "','" + credits + "');");
    } finally {
        try { conn.close(); } catch (Exception ignored) {}
    }
}

Course(String name, int credits) {
    this.name = name;
```

```
        this.credits = credits;
    }

    public int getCredits() {
        return credits;
    }

    public String getName() {
        return name;
    }
}
```

13.2 Offering.java

```
import java.sql.*;

public class Offering {
    private int id;
    private Course course;
    private String daysTimes;

    static String url = "jdbc:odbc:Reggie";

    static { try {
            Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
           }catch (Exception ignored) {} }

    public static Offering create(Course course,
                                  String daysTimesCsv)
        throws Exception
    {
        Connection conn = null;

        try {
            conn = DriverManager.getConnection(url, "", "");
            Statement statement = conn.createStatement();

            ResultSet result = statement.executeQuery(
                "SELECT MAX(ID) FROM offering;");
            result.next();
            int newId = 1 + result.getInt(1);

            statement.executeUpdate("INSERT INTO offering
              VALUES ('"+ newId + "','" + course.getName()
                      + "','" + daysTimesCsv + "');");
            return new Offering(newId, course, daysTimesCsv);
        } finally {
```

```
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public static Offering find(int id) {
    Connection conn = null;

    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();
        ResultSet result = statement.executeQuery(
           "SELECT * FROM offering WHERE ID =" + id + ";");
        if (result.next() == false)
            return null;

        String courseName = result.getString("Course");
        Course course = Course.find(courseName);
        String dateTime = result.getString("DateTime");
        conn.close();

        return new Offering(id, course, dateTime);
    } catch (Exception ex) {
        try { conn.close(); } catch (Exception ignored) {}
        return null;
    }
}

public void update() throws Exception {
    Connection conn = null;

    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();

        statement.executeUpdate(
           "DELETE FROM Offering WHERE ID=" + id + ";");
        statement.executeUpdate(
               "INSERT INTO Offering VALUES('" + id + "','" +
                course.getName() + "','" + daysTimes + "');");
    } finally {
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public Offering(int id, Course course, String daysTimesCsv) {
    this.id = id;
    this.course = course;
    this.daysTimes = daysTimesCsv;
}
```

```
    public int getId() {
        return id;
    }

    public Course getCourse() {
        return course;
    }

    public String getDaysTimes() {
        return daysTimes;
    }

    public String toString() {
        return "Offering " + getId() + ": "
                   + getCourse() + " meeting " + getDaysTimes();
    }
}
```

13.3 Schedule.java

```
import java.util.*;
import java.sql.*;

public class Schedule {
    String name;
    int credits = 0;
    static final int minCredits = 12;
    static final int maxCredits = 18;
    boolean overloadAuthorized = false;
    ArrayList schedule = new ArrayList();

    static String url = "jdbc:odbc:Reggie";
    static { try {
            Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch (Exception ignored) {} }

    public static void deleteAll() throws Exception {
        Connection conn = null;

        try {
            conn = DriverManager.getConnection(url, "", "");
            Statement statement = conn.createStatement();

            statement.executeUpdate("DELETE * FROM schedule;");
        } finally {
```

```
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public static Schedule create(String name) throws Exception {
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();

        statement.executeUpdate(
            "DELETE FROM schedule WHERE name = '" + name + "';");
        return new Schedule(name);
    } finally {
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public static Schedule find(String name) {
    Connection conn = null;

    try {
        conn = DriverManager.getConnection(url, "", "");
        Statement statement = conn.createStatement();
        ResultSet result = statement.executeQuery(
            "SELECT * FROM schedule WHERE Name= '" + name + "';");

        Schedule schedule = new Schedule(name);

        while (result.next()) {
            int offeringId = result.getInt("OfferingId");
            Offering offering = Offering.find(offeringId);
            schedule.add(offering);
        }

        return schedule;
    } catch (Exception ex) {
        return null;
    } finally {
        try { conn.close(); } catch (Exception ignored) {}
    }
}

public static Collection all() throws Exception {
    ArrayList result = new ArrayList();
    Connection conn = null;

    try {
        conn = DriverManager.getConnection(url, "", "");
```

```
            Statement statement = conn.createStatement();
            ResultSet results = statement.executeQuery(
                "SELECT DISTINCT Name FROM schedule;");

            while (results.next())
              result.add(Schedule.find(results.getString("Name")));
        } finally {
            try { conn.close(); } catch (Exception ignored) {}
        }

        return result;
    }

    public void update() throws Exception {
        Connection conn = null;

        try {
            conn = DriverManager.getConnection(url, "", "");
            Statement statement = conn.createStatement();

            statement.executeUpdate(
                "DELETE FROM schedule WHERE name = '" + name + "';");

            for (int i = 0; i < schedule.size(); i++) {
                Offering offering = (Offering) schedule.get(i);
                statement.executeUpdate(
                    "INSERT INTO schedule VALUES('" + name + "','"
                    + offering.getId() + "');");
            }
        } finally {
            try { conn.close(); } catch (Exception ignored) {}
        }
    }

    public Schedule(String name) {
        this.name = name;
    }

    public void add(Offering offering) {
        credits += offering.getCourse().getCredits();
        schedule.add(offering);
    }

    public void authorizeOverload(boolean authorized) {
        overloadAuthorized = authorized;
    }

    public List analysis() {
        ArrayList result = new ArrayList();
```

```
        if (credits < minCredits)
            result.add("Too few credits");

        if (credits > maxCredits && !overloadAuthorized)
            result.add("Too many credits");

        checkDuplicateCourses(result);

        checkOverlap(result);
        return result;
    }

    public void checkDuplicateCourses(ArrayList analysis) {
        HashSet courses = new HashSet();
        for (int i = 0; i < schedule.size(); i++) {
            Course course = ((Offering) schedule.get(i)).
                                                getCourse();
            if (courses.contains(course))
              analysis.add("Same course twice - " +
                                            course.getName());
            courses.add(course);
        }
    }

    public void checkOverlap(ArrayList analysis) {
      HashSet times = new HashSet();

      for (Iterator iterator = schedule.iterator();
          iterator.hasNext();)
      {
          Offering offering = (Offering) iterator.next();
          String daysTimes = offering.getDaysTimes();
          StringTokenizer tokens = new StringTokenizer
                                                (daysTimes, ",");
          while (tokens.hasMoreTokens()) {
              String dayTime = tokens.nextToken();
              if (times.contains(dayTime))
                  analysis.add("Course overlap - " + dayTime);
              times.add(dayTime);
          }
      }
    }

    public String toString() {
        return "Schedule " + name + ": " + schedule;
    }
}
```

13.4 Report.java

```
import java.util.*;

public class Report {
    public Report() {
    }

    Hashtable offeringToName = new Hashtable();

    public void populateMap() throws Exception {
        Collection schedules = Schedule.all();
        for (Iterator eachSchedule = schedules.iterator();
            eachSchedule.hasNext();)
        {
            Schedule schedule = (Schedule) eachSchedule.next();

            for (Iterator each = schedule.schedule.iterator();
                each.hasNext(); )
            {
                Offering offering = (Offering) each.next();
                populateMapFor(schedule, offering);
            }
        }
    }

    private void populateMapFor(Schedule schedule,
                                                Offering offering) {
        ArrayList list = (ArrayList) offeringToName.get(
            new Integer(offering.getId()));
        if (list == null) {
            list = new ArrayList();
            offeringToName.put(new Integer(offering.getId()), list);
        }
        list.add(schedule.name);
    }

    public void writeOffering(
        StringBuffer buffer, ArrayList list, Offering offering)
    {
        buffer.append(offering.getCourse().getName() + " "
                  + offering.getDaysTimes() + "\n");

        for (Iterator iterator = list.iterator();
                                            iterator.hasNext();)
        {
```

```
            String s = (String) iterator.next();
            buffer.append("\t" + s + "\n");
        }
    }

    public void write(StringBuffer buffer) throws Exception {
        populateMap();

        Enumeration enumeration = offeringToName.keys();
        while (enumeration.hasMoreElements()) {
            Integer offeringId =
                        (Integer) enumeration.nextElement();
            ArrayList list =
                        (ArrayList) offeringToName.get(offeringId);
            writeOffering(buffer, list,
                Offering.find(offeringId.intValue()));
        }

        buffer.append("Number of scheduled offerings: ");
        buffer.append(offeringToName.size());
        buffer.append("\n");
    }
}
```

13.5 TestSchedule.java

```
import junit.framework.TestCase;
import java.util.List;
import java.util.Collection;

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

    public void testMinCredits() {
        Schedule schedule = new Schedule("name");
        Collection analysis = schedule.analysis();
        assertEquals(1, analysis.size());
        assertTrue(analysis.contains("Too few credits"));
    }

    public void testJustEnoughCredits() {
        Course cs110 = new Course("CS110", 11);
        Offering mwf10 = new Offering(1, cs110, "M10,W10,F10");
        Schedule schedule = new Schedule("name");
        schedule.add(mwf10);
```

```
        List analysis = schedule.analysis();
        assertEquals(1, analysis.size());
        assertTrue(analysis.contains("Too few credits"));

        schedule = new Schedule("name");
        Course cs101 = new Course("CS101", 12);
        Offering th11 = new Offering(1, cs101, "T11,H11");
        schedule.add(th11);
        analysis = schedule.analysis();
        assertEquals(0, analysis.size());
    }

    public void testMaxCredits() {
        Course cs110 = new Course("CS110", 20);
        Offering mwf10 = new Offering(1, cs110, "M10,W10,F10");
        Schedule schedule = new Schedule("name");
        schedule.add(mwf10);
        List analysis = schedule.analysis();
        assertEquals(1, analysis.size());
        assertTrue(analysis.contains("Too many credits"));

        schedule.authorizeOverload(true);
        analysis = schedule.analysis();
        assertEquals(0, analysis.size());
    }

    public void testJustBelowMax() {
        Course cs110 = new Course("CS110", 19);
        Offering mwf10 = new Offering(1, cs110, "M10,W10,F10");
        Schedule schedule = new Schedule("name");
        schedule.add(mwf10);
        List analysis = schedule.analysis();
        assertEquals(1, analysis.size());
        assertTrue(analysis.contains("Too many credits"));

        schedule = new Schedule("name");
        Course cs101 = new Course("CS101", 18);
        Offering th11 = new Offering(1, cs101, "T11,H11");
        schedule.add(th11);
        analysis = schedule.analysis();
        assertEquals(0, analysis.size());
    }

    public void testDupCourses() {
        Course cs110 = new Course("CS110", 6);
        Offering mwf10 = new Offering(1, cs110, "M10,W10,F10");
        Offering th11 = new Offering(1, cs110, "T11,H11");
        Schedule schedule = new Schedule("name");
        schedule.add(mwf10);
```

```
        schedule.add(th11);
        List analysis = schedule.analysis();
        assertEquals(1, analysis.size());
        assertTrue(analysis.contains("Same course twice - CS110"));
    }

    public void testOverlap() {
        Schedule schedule = new Schedule("name");

        Course cs110 = new Course("CS110", 6);
        Offering mwf10 = new Offering(1, cs110, "M10,W10,F10");
        schedule.add(mwf10);

        Course cs101 = new Course("CS101", 6);
        Offering mixed = new Offering(1, cs101, "M10,W11,F11");
        schedule.add(mixed);

        List analysis = schedule.analysis();
        assertEquals(1, analysis.size());
        assertTrue(analysis.contains("Course overlap - M10"));

        Course cs102 = new Course("CS102", 1);
        Offering mixed2 = new Offering(1, cs102, "M9,W10,F11");
        schedule.add(mixed2);

        analysis = schedule.analysis();
        assertEquals(3, analysis.size());
        assertTrue(analysis.contains("Course overlap - M10"));
        assertTrue(analysis.contains("Course overlap - W10"));
        assertTrue(analysis.contains("Course overlap - F11"));
    }
    public void testCourseCreate() throws Exception {
        Course c = Course.create("CS202", 1);

        Course c2 = Course.find("CS202");
        assertEquals("CS202", c2.getName());

        Course c3 = Course.find("Nonexistent");
        assertNull(c3);
    }

    public void testOfferingCreate() throws Exception {
        Course c = Course.create("CS202", 2);
        Offering offering = Offering.create(c, "M10");
        assertNotNull(offering);
    }

    public void testPersistentSchedule() throws Exception {
        Schedule s = Schedule.create("Bob");
        assertNotNull(s);
    }

    public void testScheduleUpdate() throws Exception {
        Course cs101 = Course.create("CS101", 3);
```

```
        cs101.update();
        Offering off1 = Offering.create(cs101, "M10");
        off1.update();
        Offering off2 = Offering.create(cs101, "T9");
        off2.update();

        Schedule s = Schedule.create("Bob");
        s.add(off1);
        s.add(off2);
        s.update();

        Schedule s2 = Schedule.create("Alice");
        s2.add(off1);
        s2.update();

        Schedule s3 = Schedule.find("Bob");
        assertEquals(2, s3.schedule.size());

        Schedule s4 = Schedule.find("Alice");
        assertEquals(1, s4.schedule.size());
    }
}
```

13.6 TestReport.java

```
import junit.framework.TestCase;
import java.util.List;
import java.util.Collection;

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

    public void testEmptyReport() throws Exception {
        Schedule.deleteAll();
        Report report = new Report();

        StringBuffer buffer = new StringBuffer();

        report.write(buffer);

        assertEquals(
            "Number of scheduled offerings: 0\n",
            buffer.toString());
    }

    public void testReport() throws Exception {
        Schedule.deleteAll();

        Course cs101 = Course.create("CS101", 3);
        cs101.update();
        Offering off1 = Offering.create(cs101, "M10");
        off1.update();
```

```
        Offering off2 = Offering.create(cs101, "T9");
        off2.update();

        Schedule s = Schedule.create("Bob");
        s.add(off1);
        s.add(off2);
        s.update();

        Schedule s2 = Schedule.create("Alice");
        s2.add(off1);
        s2.update();

        Report report = new Report();

        StringBuffer buffer = new StringBuffer();

        report.write(buffer);
        String result = buffer.toString();
        String valid1 = "CS101 M10\n\tAlice\n\tBob\n" +
                        "CS101 T9\n\tBob\n" +
                        "Number of scheduled offerings: 2\n";

        String valid2 = "CS101 T9\n\tBob\n" +
                      "CS101 M10\n\tAlice\n\tBob\n" +
                      "Number of scheduled offerings: 2\n";
        assertTrue(result.equals(valid1) ||
                                    result.equals(valid2));
    }
}
```

习题 51　重复（有难度）。

A. 找出这些类中的异味，特别是重复（以及区别！）。

B. 持久性访问的模式是什么？（即跟踪对象从创建到释放的整个生命期。）

有关解决方案请见附录 A。

习题 52　应用（有难度）。

A. 与数据库相关的例程都非常相似，请应用重构技术减少这种重复性。

B. 创建一个类来提供大量数据库支持。这个类是以上对象的一个父类还是一个单独的类？

有关解决方案请见附录 A。

习题 53　数据库层（有难度）。

在刚实现的重构中，其中一项功能就是将代码向数据库层迁移。

A. 要创建一个内存版的数据库类还需要做哪些工作？

B. 你为什么这么做？

C. 你希望这个新层暴露多少 JDBC？

有关解决方案请见附录 A。

有关更多创建数据库层的信息，请参见 J2EE[支持 Enterprise Java Beans（EJB）自动映射的工具]、WebGain TopLink（www.webgain.com/products/toplink/）或有关 Scott Ambler 的描述（www.ambysoft.com/mappingObjects.html）。从这些参考资料中可以发现，即完成一个层的完整开发绝不是一件轻而易举的事。

习题 54 查找（有难度）。

fmd()例程为对象创建一个新的实例，不管它之前是否已经存在。

A. 令 find()缓存其对象，如果可能则返回一个之前找到的对象。

B. 这是一个重构还是一个开发？你增加了相关测试来验证这个新行为了吗？（建议你最好这样做）。

C. Report 可用于存储授课科目的主键（key）。利用改进的 find()机制简化此报告的代码。

有关解决方案请见附录 A。

习题 55 多个打开查询。

有些数据库存在一个限制，即一个连接不能同时打开（open）多个查询（在 Schedule.fmd()中可以看到这样一个例子：在构建一个 Schedule 时，它要查询 Offering.find()，而 Offering.find()也要查询数据库）。

A. 要强化这种限制，相应的修改会如何影响代码？哪些位置会受影响？

B. 此代码用本地（native）方法实例化了所加载的各行的所有相关对象。对此还有什么做法？

有关解决方案请见附录 A。

习题 56 计数器。

Offering 中的计数器（counter）通过一种两步式（two-step）过程进行处理：首先取得所有行的计数器的当前最大值，然后创建一个新行，其 ID 比当

前最大值多 1。这种策略在什么情况下不适用？对此你有什么解决方法？

有关解决方案请见附录 A。

习题 57　Report。

Report 创建了一个关于授课及选学该课的学生的列表。

A. 编写一个结构化查询语言（SQL）查询（如果你了解 SQL 的话），以生成此报告中的信息。

B. 请权衡这两种方法。

习题 58　数据库重构（有难度）。

上面介绍了一些可能的数据库更改，如引入 ID（而不是使用字符串主键）、抽取一个新表，等等。不能仅在数据库中做这些修改，由于我们的代码依赖于数据库结构，因此还需要对代码进行相应调整。

A. 描述重命名列（Rename Column）用作重构技术时的情况。指出在数据库或代码中应做什么修改、何时修改及何时运行测试，等等。

B. 描述分离表（Split Table）用作重构技术时的情况（所谓分离表就是将一个表分为两个相关的表）。务必考虑到所有数据迁移。

C. 是否存在一些中间步骤会暂时影响数据库的结构，却能使整个重构更安全？（考虑抽取方法在删除原有方法之前是如何复制并调整新方法的，或者考虑一些重构技术是如何一次只修改一个内容，并在每次修改之后立即运行测试的。）

有关解决方案请见附录 A。

习题 59　域类独立性。

请说出几种可以让域类（domain class）更加独立于这些类的数据库访问部分的方法。

有关解决方案请见附录 A。

第 14 章

一个简单的游戏

本例涉及重构技术和测试驱动设计。

假设要开发一个系统玩井字游戏：在方格内放上不同的标识。在井字游戏中，有 3×3 个方格，如果你能将标识放在同一行上的三个方格中即取胜。在 Hasbro 所提出的 4 连子游戏（Connect Four）中则有一个矩形格，要求你将 4 个格置于一行，但是各列必须由下至上填充。我们先来看一个简化版的井字游戏，寻求解决一般情况的方法。

以下是一些测试代码和第一个版本的代码（可在 www.xp123.com/rwb 在线得到）。

```
import junit.framework.*;

public class GameTest extends TestCase {

  public GameTest(String s) {super(s);}

  public void testDefaultMove() {
    Game game = new Game("XOXOX-OXO");
    assertEquals(5, game.move('X'));

    game = new Game("XOXOXOOX-");
    assertEquals(8, game.move('O'));

    game = new Game("---------");
    assertEquals(0, game.move('X'));

    game = new Game("XXXXXXXXX");
    assertEquals(-1, game.move('X'));
  }

  public void testFindWinningMove() {
    Game game = new Game("XO-XX-OOX");
```

```
        assertEquals(5, game.move('X'));
    }
    public void testWinConditions() {
        Game game = new Game("---XXX---");
        assertEquals('X', game.winner());
    }
}

public class Game {
    public StringBuffer board;

    public Game(String s) {board = new StringBuffer(s);}

    public Game(StringBuffer s, int position, char player) {
        board = new StringBuffer();
        board.append(s);
        board.setCharAt(position, player);
    }

    public int move(char player) {
        for (int i = 0; i < 9; i++) {
            if (board.charAt(i) == '-') {
                Game game = play(i, player);
                if (game.winner() == player)
                    return i;
            }
        }

        for (int i = 0; i < 9; i++) {
            if (board.charAt(i) == '-')
                return i;
        }
        return -1;
    }

    public Game play(int i, char player) {
        return new Game(this.board, i, player);
    }

    public char winner() {
        if (board.charAt(0) != '-'
                && board.charAt(0) == board.charAt(1)
                && board.charAt(1) == board.charAt(2))
            return board.charAt(0);
        if (board.charAt(3) != '-'
                && board.charAt(3) == board.charAt(4)
                && board.charAt(4) == board.charAt(5))
            return board.charAt(3);
        if (board.charAt(6) != '-'
```

```
            && board.charAt(6) == board.charAt(7)
            && board.charAt(7) == board.charAt(8))
        return board.charAt(6);
    return '-';
  }
}
```

注意 winner()例程被简化了：只要让三个标识位于水平上的同一行即可取胜。注意，游戏规则也很简单：尽可能赢，否则可以将标识置于任何位置。不过后面我们会使用一些更复杂的策略。

习题 60 异味。

遍历此代码，找出其中的异味。

有关解决方案请见附录 A。

习题 61 简单的修改。

要明确对代码做什么修改往往并不容易。下面先来修正一些简单的问题。一次改正一个。

- 变量 i 表述不够清晰，将它改为 move。
- 值-1 是一个标志值，创建一个常量 NoMove 来表示此标志。
- winner()函数存在大量重复，消除这些重复。
- 检查方格字符是否为'-'，这实际上是检查这个方格中是否已经有标识了。抽取一个方法来完成这个工作，并为它设置合适的名称。

本例使用了两个“for”循环：一个用于找出一个成功走法（move），另一个用于找出一个默认走法。对此，一种做法是将两个循环抽取至一个方法。后面还要增加更多的策略，在此过程中，我们会看到每个策略都有它自己的方法。还有一种做法就是将两个循环合并，在一步中通过可能的走法进行处理。我们将采用后面一种做法。

为此，首要工作可能是为一个临时变量分配一个值，而不是立即返回它。可以如下修改第二个循环：

```
defaultMove = NoMove;
for (int i = 0; i < 9; i++) {
  if (board[i] == '-')
    if (defaultMove == NoMove)
      defaultMove = i;
}
```

习题 62　合并循环。

合并循环（Fuse Loops）表示将两个循环合二为一。请做小步调整，即每次仅做少量修改，以保证修改过程的安全性。何时能够安全地合并两个循环呢？（合并循环是编译器使用的一个标准术语，未列在 Fowler 的《重构》名录中。）

有关解决方案请见附录 A。

这是最安全的方法。之所以使用这种方法是因为我们不希望重构更改代码行为。为了与原先的代码等价，还需要使用一个卫哨子句来确保没有对 defaultMove 赋值。不过我们可以先做一个开发假设：我们并不关心默认走法是什么，因此可以删除 defaultMove == NoMove 测试。在找到一个可靠走法时没有必要停下来：也就是说，大可尝试每一种可能的走法，我们的走法很可能比默认走法更胜一筹。因此可以删除提前退出的中止（break）测试。再次运行测试，确保没有更改任何重要内容。（你可能需要调整测试。这说明了什么？）

现在只有一个循环了，但是确定返回什么的条件仍有些复杂：

```
if (winningMove != NoMove)
   return winningMove;
if (defaultMove != NoMove)
   return defaultMove;
return NoMove;
```

习题 63　结果。

如何进行简化？

习题 64　下一步。

下一步要使用哪些重构技术？

此时还存在着大量魔法数（magic number）。winner()例程中就充斥着许多魔法数，而且 bestMove()的主循环中还有一个“9”。

习题 65　常量。

“9”是什么？命名某些常量并重写它们。

命名对我来说是件棘手的事，最终我选定了 rows 和 columns（我也在尝试使用其他不同的命名约定。有些风格可能会采用 ROWS 和 COLUMNS 这样的形式）。

习题 66 winner()中的重复。

下面是新的 winner ()例程。

```
public char winner() {
   if (!canPlay(0)
       && board.charAt(0) == board.charAt(1)
       && board.charAt(1) == board.charAt(2))
       return board.charAt(0);
       if (!canPlay(3)
           && board.charAt(3) == board.charAt(4)
           && board.charAt(4) == board.charAt(5))
           return board.charAt(3);
           if (!canPlay(6)
               && board.charAt(6) == board.charAt(7)
               && board.charAt(7) == board.charAt(8))
               return board.charAt(6);
           return '-';
}
```

请消除此例程中的重复。

假设你对行（row）使用了一个循环，并抽取了某种方法来确定是否存在一个成功的组合。

为了估计这种重构，再来做一个开发假设。在我们看来，这里可以进行两种修改。首先，我们玩的并不是井字游戏，因为这里只认为水平的三连子才算获胜。扩展此例程，让垂直和对角线上有三子相连也算赢。在采用以上重构技术后，做这样的更改是否容易？

这里还存在另一个隐藏的常量：即作为取胜条件的行数（我们曾提到过 4 连子游戏，这是我们最终所支持的形式的一个变种）。假设改为 5×5 方格，并认为一行中有 4 子相连即获胜。在代码中设置此策略是否容易？目前还不需要你增加这个特性，仅提出来供你思考。（我的解决方法也许不算最好：我定义了一个 winningCombination()例程，它取三个位置作为参数；另外我还创建了一个数组，它显式调用成功放置标识的方格。这种做法对于井字游戏可行，但是却不适用于一行有 n 子相连的情况。）

到目前为止，我们应用的大部分重构技术都取得了明显的成效。现在我希望通过合并重构和新的实现来扩展并改进我的程序，但是我还不确定下一步怎么做最好。

我把这当做虚拟程序设计（subjunctive programming）。“虚拟”意指所

谈及的话题是一些可能的情况（“如果我是一个富翁……”）。我认为可以尝试一些想法，看看会得到什么，不明确的话也无所谓。

进行虚拟程序设计基于两点：首先是合作者，有合作者的协助才能尽情尝试。另外一点是源代码控制系统，有了它才能把不希望做的修改改回去。

主导方向是确定一些更高超的策略，而不是“尽可能赢，否则将标识置于任何位置”。我的想法是创建一个 Move 对象，让它来评价下一步走法是好是坏。

习题 67　迭代器。

上面运行的是基于整数的循环，这些整数代表可能的走法。下面改为使用一个针对走法的迭代器。Fowler 的《重构》名录中没有列出这个重构技术，所以请通过小步调整来实现。

1. 先把 int 转换为 Integer，以便能访问对象域。
2. 创建一个迭代器，其中 next()方法返回一个 Integer。从 bestMoveFor () 方法中删除所有对 int 的引用。
3. 引入一个 Move 类。

现在主循环如下所示：

```
for (Iterator iter = new MoveIterator(); iter.hasNext(); ) {
  Move move = (Move)iter.next();
  if (!canPlay(move)) continue;
```

习题 68　仅允许合法的走法

我们的迭代器会发送所有走法，不管它是否合法。将 canPlay 测试移至迭代器中，让迭代器只发送合法的走法。

这样将只循环可能的走法，选择其中最好的走法。在此遵循一个简单的规则：赢了最好，其他走法也可以接受。但是赢总是少数情况，因此应选出一个比较好的中间走法（有些走法优于其他走法）。可以为每种走法打分：即这种走法有多好？作为一种具体处理方法，我们认为赢就是 100 分，而其他走法均为 0 分（也可以认为对手得分为-100 时我方赢，不过目前我们不考虑这种情况）。

注意，现在已经脱离了重构领域，而是对程序进行语义上的调整。这也是开发的一贯方式。因为重构能使问题变得更清晰，因而可以发现更好的解决方法。

开发环节

习题 69 得分。

修改程序来计算走法的得分并返回分数最高的走法。

注意如何将分数与特定走法关联，这可能是 Move 对象的一部分。这样做可以避免在“主”循环中跟踪整数得分。

习题 70 比较走法。

将此得分添加到 Move 类中，再增加某种 max 方法，或者增加比较走法的操作符。

这样做是不是更好一些呢？我认为是这样，所以暂时做了这种修改。但是我并不会“一条道走到黑”。就目前而言，它有所帮助，但如果将来它有所妨碍，我会毫不迟疑地删除它。

此程序会计算每一种可能的走法并做出响应。对于井字游戏来说这是可行的，而且将它用于 Hasbro 的 4 连子游戏也许仍可行，但是对于象棋或 Go 之类的游戏，这显然不适用。最终我们必须开发一种新的策略。

一种处理方法是限制搜索深度。比如说建立了一个深度切断值（即达到这个值后停止搜索），比该值更深的搜索都将返回“不知道”。我们将传递一个附加的参数来表示当前深度。

习题 71 深度。

使用增加参数（Add Parameter）增加一个深度参数，同时适当地维护它的值。设置深度参数后，增加一个早期检测来返回何时搜索过深。此时会返回什么走法呢？

习题 72 缓存。

可以认为性能优化调整是对性能的重构（refactoring for performance），它试图让程序完成同样的工作，只是速度更快。如果将此程序看做探索可能走法的一棵游戏树，则可以看到通过不同路径的棋盘（状态）。你能否对这些走法进行缓存，从而识别出已经分析过的棋盘（状态）呢？

习题 73　平衡。

以上对象是否获得了适当的平衡？是否还缺失某些对象？应该由 Game 计算得分，还是由 Move 完成？请分别进行尝试并查看结果。其中一些决策使缓存过程更容易还是更困难？

习题 74　新特性。

增加一些新特性，要求以测试优先的方式完成，同时确保伴随有重构。

A. 对手得分为-100 表示赢。

B. 扩展至 m×n 井字游戏。

C. 要求每一步把标识（棋子）放在一列中最下面的空位上。

习题 75　最小-最大算法（Min-Max）。

A. 再增加一个特性：即使用最小-最大算法（Min-Max），有关这种算法在任何一本人工智能课本中都有描述。这里不是说“如果不赢，怎么走分数都一样”，而是“我要选择最佳的走法，假设对手此时的走法对我最不利”。对手也采用同样的规则。如何在代码中反映这一点？有没有什么使用技巧？

B. 这种方法还有一个扩展，称为 α-β 剪枝（alpha-beta pruning），即通过建立切断值来避免搜索部分树。请参考某本人工智能（AI）方面的书，了解是否可以实现这种方法。这是一种重构还是一种新开发，抑或是别的什么做法？

习题 76　完成了吗？

这是一个更改应用程序结构的试验，我们还可以采取其他一些途径。具体来讲，我认为类之间的平衡可以有不同的设计。另外，最初的测试针对的是一个 3×3 井字游戏，如果从 1×1 开始，逐步扩展至 m×n，让 3×3 是一个特例，那将会很有趣。

是从头开始，还是基于当前的基础加以扩展，哪种做法更好呢？

第 15 章

名录

“这里谁是负责人？”

这是一个中等规模的例子，涉及重构和开发。本章的目标是展示如何使用重构技术来探索设计决策并分析其结果。

15.1 引言

假设一个商店在出售各种商品。通过简单设计，我们认为存在以下需要重点关注的类：Item（待售商品）、Catalog（所有商品清单）和 Query（查找某一组特定商品）。

Catalog 和 Query 必须协作才能完成一个搜索，但是这种协作如何进行有待商榷。假设还存在第 4 个类，即 Interrogator，它有一个方法 evaluate(catalog, query)，此方法返回一个商品列表，由这个方法封装如何处理问题的决策。

习题 77　计算。

evaluate()的实现核心有以下可选项：

(a) catalog. itemsMatching(query)

(b) query.matchesIn(catalog)

(c) process(catalog.data, query.data)（也就是说，此模块使用 Catalog 和 query 中的信息来决定做何处理）

评价一下各个选项，如果有其他想法也可以提出来。

有关解决方案请见附录 A。

15.2 第 1 种做法：Catalog.itemsMatching(query)

下面分析第一个选项（a）。作为第一次尝试，我们将从一个 Catalog 出发，并用字符串来表示查询和商品。下面是一个测试：

```
Catalog catalog;
public void setUp() {
   catalog = new Catalog();
   catalog.add("Hammer - 10 lb");
   catalog.add("shirt - XL - blue");
   catalog.add("shirt - L - green");
   catalog.add("Halloween candle - orange");
   catalog.add("Halloween candy - gum");
}

public void testSimpleQuery() {
   List result = catalog.itemsMatching("shirt");
   AssertEquals(2, list.size());
}
```

习题 78　Catalog（名录）。

编写一个简单的 Catalog 类，其中包括以上两个方法。

习题 79　Query（查询）。

将当前查询字符串抽取到一个 Query 类中。

Catalog 类可能有这样一行代码：

```
if (item.indexOf(query) != -1)
```

将其修改为：

```
if (item.indexOf(query.toString()) != -1)
```

这是最基本的修改。

习题 80　权衡异味。

在清除少许基本类型困扰（Primitive Obsession）的过程中会增加一些依恋情结或不当的紧密性，请解释为什么会这样。

有关解决方案请见附录 A。

习题 81 搬移至 Query。

对 item.contains (query.toString())应用抽取方法，然后使用搬移方法将匹配行为置于查询中（最终将调用 query.matches (item)）。

我们希望将查询扩展为支持更复杂的情况。通过“Halloween OR shirt”应找出多个商品。在本题中不必考虑解析；相反，假设我们的查询是一个组合，形式如图 15.1 所示。将图中最上面的节点称为 OrQuery，其他节点称为 StringQuery。这两类节点都是 Query 的子类。

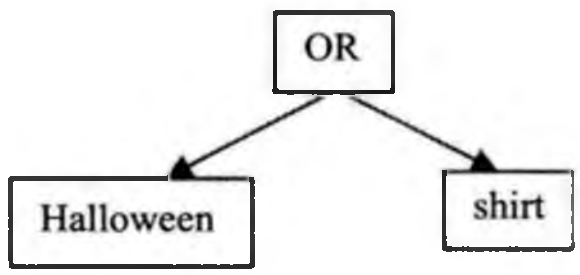


图 15.1 组合形式的查询

习题 82 StringQuery 和 Query。

抽取超类（或子类，这取决于你如何看待），使 StringQuery 变成 Query 的一个子类。

有时你会听到这样一种说法，即重构技术就是为新代码留出空间。上面就是这样的一个例子：我们在为一个 OrQuery 子类留出空间。也可以通过另一种方式得到这个结果，即引入 OrQuery，再消除其中的重复。

习题 83 OrQuery。

创建一个能通过以下测试的 OrQuery 类：

```
Query query = new OrQuery (
   new StringQuery("shirt"),
   new StringQuery("Halloween"));
list = catalog.itemsMatching(query);
assertEquals(4, list.size());
```

可以认为上面这种方法是策略（Strategy）模式的一个例子。名录是上下文，不同的查询实现了不同的匹配算法（可能还需要检测构造查询的组合模式）。

习题 84 性能（有难度）。

假设通过记录表发现查询太慢，而且我们每天都要成批地使用这些查询

成千上万次，而其他工作则每天仅做一次。另外我们还发现，任何给定的词都不太可能出现在小范围商品清单中。

对于提高性能，你有哪些好的建议？

有关解决方案请见附录 A。

性　能

我们希望系统性能好、响应快，等等，因此有时会担心代码的性能。这是可以理解的。有些重构技术会引入更多的对象、更多的方法调用，诸如此类，因此可能会使系统变慢。

其他一些有利因素则包括：

- 大多数未经调整的代码只有几处影响性能的热点，可以以后再调整它们。
- 简洁的设计为深入理解提供了基础，从而可以完成大幅改进。
- 简洁的设计更易于进行优化调整。

如果担心性能，可以采用下面这些方法：

- 设计一个周全的实验，查看通过某种途径是否能提升性能。（例如，我们可能有线性搜索集合，但我们知道，如果将它变成我们需要的，可以通过索引它来得到更高的性能。）
- 查找表明有更快方法的证据。
- 分散式尝试——抽取出基本算法，然后单独运行它。

发现一种可以加速的方法后，就算大功告成了。不必急于实现它，等到真正需要时再说。

15.3　第 2 种做法：Query.matchesIn(catalog)

研究另一种解决方法时，我们仍从同一个 setup()例程入手，但是采用一个不同的测试：

```
Query query = new Query("shirt");
List list = query.matchesIn(catalog);
AssertEquals(2, list.size());
```

习题 85 不同的测试。

为此测试实现 Catalog 和 Query（可能需要循环商品并在内部某处调用 query.matches(item)）。

习题 86 不同的测试。

将 StringQuery 重构为 Query 的一个子类（与前同）。

习题 87 再看 OrQuery。

实现 OrQuery。应当对它的各个子查询使用 matchesIn(catalog)。

你可能会发现 List 并不是最方便的类型，可能更愿意使用一个 Set 方法（以便让 OrQuery 不产生重复）。因此你可以将此 List 类型修改为 Set，但也可以用它作为警告：查询的结果可能是它自己的对象，也可能是另一个名录。

习题 88 商品（Item）和集合（Set）（有难度）。

在某种程度上，这比前面的实现更复杂。我认为原因在于先前的实现所比较的是单独的商品，而这个实现则是按集合处理。

A. 如果采用这种机制，名录中可以有哪些类型的查询？

B. 在哪里可以优化性能？

C. 如果将这个版本看做解释器（Interpreter）模式的一个实现，请给出相关说明。

有关解决方案请见附录 A。

第一种方法优化向名录提供了查询的信息（筛选的内容），第二种方法优化则向查询提供了名录的信息（商品集合）。

15.4 第 3 种做法：Process(catalog.data, query.data)

在第 1 种做法中，我们将查询传递给名录。在第 2 种做法中，则将名录传递给查询。这两种方法都使类相互耦合：一个类必须知道另一个类的信息。第 3 种方法则是让 Interrogator. evaluate()从名录获得某些信息，然后将此信息

与通过查询获得的信息结合起来，这样只有 Interrogator.evaluate()了解这两个类。

习题 89　FilterEnumerator（有难度）。

A. 让 Catalog 提供一个商品的枚举器，让 Query 确定一个商品是否可接受（这两个类都了解商品，但这两个类相互之间却互不知晓）。创建一个 FilterEnumerator 类：这是一个枚举器，它以另一个枚举器和一个查询作为参数，只返回与查询匹配的那些商品。Interrogator 将建立对象并使用这个新的枚举器。

B. 这是哪种模式的一个例子？

C. 这会对性能有什么影响？

有关解决方案请见附录 A。

习题 90　其他方法。

是否有其他方法能使 Query 和 Catalog 相互了解？

15.5　小结

我们首先从两个需要相互协作的类开始，了解了实现其核心交互的三种方法。对于哪一种方法最好，并没有定论。需要说明的是，它们最终会导致得到不同的系统。平衡各对象之间的职责也是一门艺术。

第 16 章

计划游戏模拟器

我们有一个 GUI（Graphic User Interface，图形用户界面）程序需要清理。这是 XP 计划游戏（planning game）的一个简单模拟，如图 16.1 所示。客户（Customer）可以创建新卡、分解现有卡、删除原卡、对计划进行简单分析。程序员（Programmer）可以对卡做成本估计和更新速度估计。另外，任何人都可以在卡上键入内容，或者通过拖动卡的边框来移动它。另请注意，屏幕底部有一个摘要信息，每做一个动作后该信息都会得到更新。

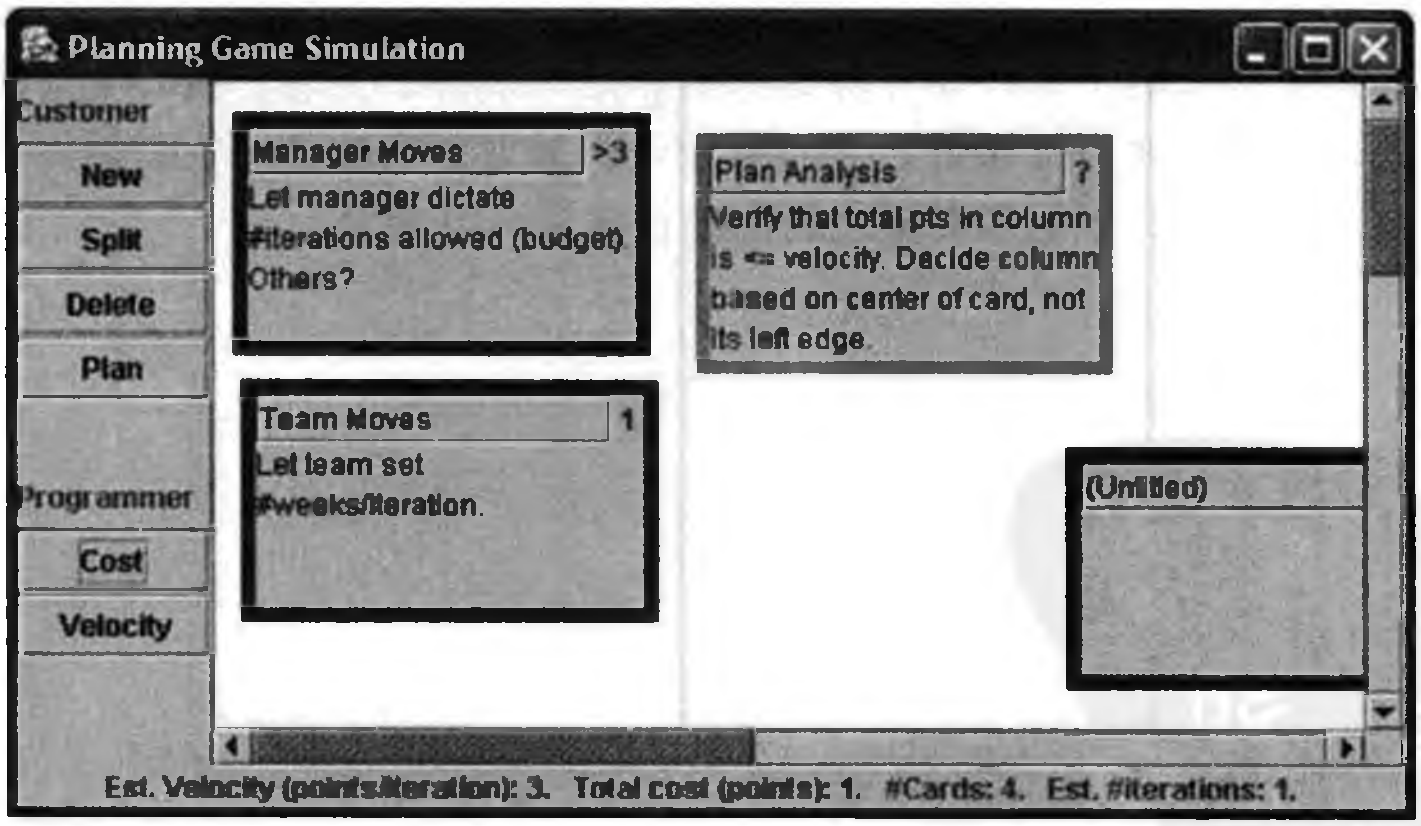


图 16.1　模拟 XP 计划游戏

这个程序的代码基于 3 个类：Table（整个应用，包括按钮）、Background（卡所在环境）和 Card（表示一个索引卡）。

16.1 第 1 部分：原始代码

代码

可以从 www.xp123.com/rwb 在线获得。

Table.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.JTextComponent;

public class Table extends JFrame {
    public static void main(String[] args)
    {
        new Table();
    }

    private JComponent body;
    private Card selection;
    private JLabel summary;
    int velocity = 1;
    int budget = 4;
    JTabbedPane tabs;

    public Table()
    {
        super("Planning Game Simulation");
        JComponent buttons = makeButtons();
        this.getContentPane().add(buttons, "West");

        body = new Background(1000, 1000, 190);
        body.addContainerListener(new ContainerListener() {
            //监听容器变化
            //以了解何时更新选择高光

            public void componentAdded(ContainerEvent e)
            {
                resetSelection();
            }

            public void componentRemoved(ContainerEvent e)
            {
                resetSelection();
```

```
            }

            private void resetSelection()
            {
                if (selection != null)
                    selection.setBorder(
                     BorderFactory.createLineBorder(Color.blue, 6));

                if (body.getComponentCount() == 0) {
                    selection = null;
                } else {
                    selection = (Card) body.getComponent(0);
                    selection.setBorder(
                     BorderFactory.createLineBorder(Color.red, 6));
                }
            }
        });
        JScrollPane scroll = new JScrollPane(body);
        scroll.setPreferredSize(new Dimension(100, 100));
        this.getContentPane().add(scroll, "Center");

        summary = new JLabel("", SwingConstants.CENTER);
        summary.setText(summary());
        this.getContentPane().add(summary, "South");

        this.pack();
        this.setSize(800, 600);
        this.show();
    }

    private JComponent makeButtons()
    {
        JPanel panel = new JPanel(new GridLayout(0, 1));

        panel.add(new JLabel("Customer"));
        makeCustomerButtons(panel);

        panel.add(new JLabel(" "));

        panel.add(new JLabel("Programmer"));
        makeProgrammerButtons(panel);

        JPanel outer = new JPanel(new BorderLayout());
        outer.add(panel, "North");
        outer.add(new JLabel(""), "Center");
        return outer;
    }

    private void makeCustomerButtons(JPanel panel)
```

```
{
    JButton button;
    button = new JButton("New");
    panel.add(button);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            Card card = new Card();
            body.add(card, 0);
            selection = card;
            updateCost();
            repaint();
        }
    });

    button = new JButton("Split");
    panel.add(button);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            if (selection == null) return;
            Card card = new Card(selection);
            body.add(card, 0);
            selection = card;
            updateCost();
            body.repaint();
        }
    });

    button = new JButton("Delete");
    panel.add(button);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            if (body.getComponentCount() == 0) return;

            body.remove(0);
            selection = null;
            if (body.getComponentCount() != 0)
                selection = (Card) body.getComponent(0);
            updateCost();
            body.repaint();
        }
    });


    button = new JButton("Plan");
    panel.add(button);
    button.addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e)
            {
                StringBuffer report = new StringBuffer();
                //检查需要估计或分解的索引卡
        for (int i = 0; i < body.getComponentCount(); i++) {
                    Card card = (Card) body.getComponent(i);
                    if (card.needsEstimate())
                        report.append(
                        "Needs estimate: " + card.title() + "\n");
                    else if (card.needsSplit())
                        report.append(
                        "Needs to be split: " + card.title() + "\n");
                }

                if (report.length() == 0)
                    JOptionPane.showMessageDialog(
                        body,
                        "Plan OK; no cards need estimates or splitting",
                        "Issues in plan",
                        JOptionPane.OK_OPTION);
                else
                    JOptionPane.showMessageDialog(
                      body, report.toString(),
                      "Issues in plan", JOptionPane.OK_OPTION);
            }
        });
    }

    private void makeProgrammerButtons(JPanel panel)
    {
        JButton button;

        button = new JButton("Cost");
        panel.add(button);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                if (selection == null) return;
                selection.rotateCost();
                updateCost();
            }
        });

        button = new JButton("Velocity");
        panel.add(button);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                velocity++;
```

```
            if (velocity >= 10) velocity = 1;
            updateCost();

        }
    });
}

private void updateCost()
{
    summary.setText(summary());
}

private String summary()
{
    StringBuffer result = new StringBuffer();
    result.append(
       "Est. Velocity (points/iteration): " + velocity + ".   ");
    result.append("Total cost (points): " + cost() + ".   ");
    result.append("#Cards: " + body.getComponentCount() + ".   ");
    result.append(
       "Est. #iterations: "
       + (cost() + velocity - 1)/ velocity + ".   ");
    return result.toString();
}

    //一组索引卡的总成本
    private int cost()
    {
        int total = 0;
        for (int i = 0; i < body.getComponentCount(); i++) {
            Card card = (Card) body.getComponent(i);
            total += card.cost();
        }
        return total;
    }
}
```

Background.java

```
import java.awt.*;
import javax.swing.*;

public class Background extends JLayeredPane {
    int lineDistance;

    public Background(int width, int height, int lineDistance)
    {
        super();
        setPreferredSize(new Dimension(width, height));
```

```
        this.lineDistance = lineDistance;

        setOpaque(true);
        setBackground(Color.yellow);
        setForeground(Color.orange);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int height = getHeight();
        for (int i = 0; i < this.getWidth(); i+= lineDistance)
            g.drawLine(i, 0, i, height);
    }
}
```

Card.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class Card extends JPanel {
    static int cards = 0;

    final static int scale = 34;            //索引卡大小比例因子
    final static int height = 3 * scale;    // 3×5
    final static int width = 5 * scale;

    final static int rollover = 12;         //新索引卡放在不同的行
    final static int offset = scale;        // 新索引卡置于多远处

    JTextField title;
    JTextArea body;
    JLabel costLabel;
    int cost;

    public Card()
    {
        this("(Untitled)", "");
    }

    public Card(Card from)
    {
        this(from.title.getText(), from.body.getText());
    }

    private Card(String titleText, String bodyText)
    {
        super();
```

```
        setBorder(BorderFactory.createLineBorder(Color.blue, 6));
        cards++;

        setLayout(new BorderLayout());
        JPanel top = new JPanel(new BorderLayout());

        title = new JTextField(titleText);
        title.setBackground(Color.pink);
        title.addFocusListener(new FocusAdapter() {
            public void focusGained(FocusEvent e)
            {
                moveToFront();
            }
        });

        top.add(title, "Center");
        costLabel = new JLabel(" ? ");
        cost = 0;
        top.add(costLabel, "East");
        add(top, "North");

        body = new JTextArea(bodyText);
        body.setLineWrap(true);
        body.setWrapStyleWord(true);
        body.setBackground(Color.pink);
        body.addFocusListener(new FocusAdapter() {
            public void focusGained(FocusEvent e)
            {
                moveToFront();
            }
        });

        add(body, "Center");

        setLocation(
            (cards % rollover) * offset,
            (cards % rollover) * offset);
        setSize(width, height);

        //打开鼠标事件以便检测到被拖动
        enableEvents(
            AWTEvent.MOUSE_EVENT_MASK
            | AWTEvent.MOUSE_MOTION_EVENT_MASK);
    }

    private void moveToFront()
    {
        JLayeredPane background = (JLayeredPane) this.getParent();
        background.moveToFront(this);
```

```
    }

    private int lastx, lasty;

    public void processMouseEvent(MouseEvent e)
    {
        if (e.getID() == MouseEvent.MOUSE_PRESSED) {
            lastx = e.getX();
            lasty = e.getY();
            moveToFront();
        } else
            super.processMouseEvent(e);
    }

    public void processMouseMotionEvent(MouseEvent e)
    {
        if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
            Point here = getLocation();
            setLocation((int) (here.getX() + e.getX() - lastx),
                        (int) (here.getY() + e.getY() - lasty));
        } else
            super.processMouseMotionEvent(e);
    }

    public void rotateCost()
    {
        String label = costLabel.getText();
        if (label.equals(" ? ")) {
            costLabel.setText(" 1 ");
            cost = 1;
        } else if (label.equals(" 1 ")) {
            costLabel.setText(" 2 ");
            cost = 2;
        } else if (label.equals(" 2 ")) {
            costLabel.setText(" 3 ");
            cost = 3;
        } else if (label.equals(" 3 ")) {
            costLabel.setText(" >3 ");
            cost = 0;
        } else if (label.equals(" >3 ")) {
            costLabel.setText(" ? ");
            cost = 0;
        } else {  // shouldn't happen
            costLabel.setText(" ? ");
            cost = 0;
        }
    }

    public int cost()
    {
```

```
        return cost;
    }

    public String title()
    {
        return title.getText();
    }

    public boolean needsSplit()
    {
        return costLabel.getText().equals(" >3 ");
    }

    public boolean needsEstimate()
    {
        return costLabel.getText().equals(" ? ");
    }
}
```

实战练习

习题 91　异味（有难度）。

A. 你在这段代码中发现了哪些异味？可以参考 Fowler 的《重构》名录清单，同时你可能还会发现一些别的问题。

B. 你将使用什么策略（重构的顺序）来改进这段代码？

有关解决方案请见附录 A。

习题 92　测试（有难度）。

测试怎样做？此代码的作者声称："这基本上都是 GUI，所以无法进行测试。"找出两到三种可以测试此代码的策略，并对当前结构稍加调整。

有关解决方案请见附录 A。

习题 93　编写测试。

为每个类编写至少一个测试，要求使用你能找到的最简单策略。

有关解决方案请见附录 A。

面对这样一个比较大的重构任务时，我总是喜欢先从容易的地方入手，即首先修正不好的名字（Poor Name）或过长的方法（Long Method）之类的异味。当代码中存在许多可以改进的地方时，在试图做大幅改进之前，先对代码自身加以处理会很有帮助。

习题 94 死代码。

Table 类中有未使用的代码（其形式为未引用的变量），请去除这些死代码。

有关解决方案请见附录 A。

习题 95 将 Table 改为 PlanningGame。

类名 Table 似乎是一个顶层类的名字，这不是很合适（到底是什么表呢？）。以 Main 作为类名甚至都强一点。虽然 Main 无法表述太多信息，但起码它也不会产生误导（miscommunicate）。下面以 PlanningGame 作为这个类的名字，请适当地重命名该类。

习题 96 匿名内部类。

PlanningGame 类在许多地方使用了一个匿名内部类来设置事件监听者（匿名内部类的例程在代码中间定义，可以通过查看某处的“});”加以识别）。

A. 通过把各个匿名内部类抽取到它所属的内部类，将匿名内部类转换为内部类（Convert Anonymous Inner Class to Inner Class）。

B. 将它们作为独立的类是否更合适？

有关解决方案请见附录 A。

习题 97 魔法数。

这里到处都是魔法数（magic numbers）[其中一些称为魔法色（magic colors）可能更准确]。将各个魔法数抽取至类顶部作为最终静态变量（final static variable）使用（把重点放在数据和颜色上，以后再处理字符串）。

有关解决方案请见附录 A。

16.2 第 2 部分：重新分配特性

到目前为止，我们基于 3 个类建立了一个计划游戏模拟器，这 3 个类是 PlanningGame（管理控件和背景）、Background（游戏界面）和 Card（对应索引卡）。我们发现了大量需要改进的地方并创建了几个简单测试，同时完成了一些简单的重构。这部分的原始代码可以从 www.xp123.com/rwb/获得。

PlanningGame（先前的 Table）中有多处访问了 Background 内部的字段。这是一种依恋情结：一个类花时间处理本该由其他类处理的数据。解决办法是将数据和方法移至别处。

习题 98　测试按钮。

将特性移至别处之前，希望如果我们采取了不正确的做法，测试会给出相应的警告。到目前为止，测试只是调用了公共的可用方法，这不包括对模拟按钮的点击功能。如果你的测试没有这一功能，请务必增加。（可以逐个提供按钮，也可以找出一种更通用的方法。有了按钮之后，就可以调用 doClick() 来使按钮完成它的工作了。）

有关解决方案请见附录 A。

你会发现，Plan 按钮会带来问题：它希望创建一个对话框，而这对于自动化（程序化）测试来说是一件棘手的事情。我现在暂不去管它，同时在做相应处理时要格外小心。（你有时希望进行重构来创建一个测试，但同时又希望有一个测试来帮助你进行安全的重构，Plan 按钮正是这样一种情况。）

习题 99　将特性移至 Background。

- 将 Plan 按钮的“字符串计算”部分抽取出来并移至 Background。（可能必须进行手动测试，但是在重构之后，就能自动化完成这一测试了。）
- 将 cost ()移至 Background。
- 在 Background 上创建一个 count()方法，此方法查看 getComponent-count()。再调整调用者。
- 在 Background 上创建一个 top()方法，它将返回 getComponent(0)。再调整调用者。（因为所有调用者都希望对返回结果进行某种类型的转换，因此现在我尝试返回类型为 Card 的某种结果。）

后两个修改更多的是为了改善表述能力而不是解耦合。

习题 100　将特性移至 Card。

Card 不知道它自己的选择状态。请修改这一点，即抽取代码来建立一个新方法 setSelected(Boolean)。我认为应该有一个 isSelected()方法，但是没有人查看选择状态！

对于选择（selection）处理我不是完全满意，另外对于 Card 知道它背景这一点，我也认为有待改进，不过我现在不能完全确定如何进行处理。此外，在 Card 的成本（cost）方面及 PlanningGame 的建立按钮方面还存在着大量重复。对于这些重复我已经有了解决思路，所以下面会先处理这个问题。

习题 101 清理成本。

- 去除各个标签周围的空格（显然这些空格符是用于填充的）。
- 抽取一个 Cost 类。调整测试类。

有关解决方案请见附录 A。

如果将 Cost 类抽取出来的话，我们可能搞不清楚在何处调用 Card.needsEstimate()和 Card.needsSplit()。它们均由 Background.planAnalysis()和测试调用。但是 planAnalysis()会要求 Card 做这些检查，那么为什么不让 Card 计算它自己的分析呢？（以后可能还会有其他非 Card 级别的分析，到时就要由 Background 自行处理了。）因此需要通过在 Card 中设置一个 planAnalysis()方法来消除这种依恋情结。这个问题我之前并没有注意到，是重构让我发现了这一点。

5 个为什么

查看 planAnalysis()方法时，注意 JOptionPane.showMessageDialog()的最后一个参数。在看完这个习题之后，我的同事 Edmund Schweppe 指出 OK_OPTION 并不是传递给该方法的合适常量。这个参数意在指出该消息是一个警告还是用于提供信息，抑或是其他某种类型。如果我们与客户交流，会发现这应当是一个警告（如果出现问题的话），或者用于提供信息。在重构中发现 bug 并不罕见。

在处理问题时，通常可以自问 5 次“为什么”来找出问题的根本原因。最明显的症状是：它总是显示停止标记——这确实是用户所希望的吗？往回退一步，可以看到我们使用了错误的常量。但是为什么这种情况没被检测出来呢？原因之一在于我们没有提供针对对话框类型的任何测试（对话框确实很难测试）。还要注意代码的格式：代码中有一行非常长，错误常量位于最右端。在我的编辑器中，这意味着我必须浏览到行末，否则我是看不到这个常量的。（个人总结：调整编程风格，对于长代码行中的参数，让每个参数占一行）。

为什么最初会写错呢？原因之一是我没有及时建立文档，而是依赖于编辑器所建议的常量，并采用了看上去正确的常量。（个人总结：参考一本新的 Java 基类（Java Foundation Classes）手册，不要过分相信自己的好记性）。在本题中，代码是我独立编写的，如果有合作者，他们在我键入代码时应该很容易发现有问题的参数。

为什么会出现这种错误呢？从一定程度上讲，这是库设计者做决策的直接后果：他们对所有选项类型和消息类型仅使用整型常量。库设计者本可以为这些选项创建单独的类（作为一种枚举）；或者针对不同的消息类型使用不重叠的值或不同方法。（个人总结：即使不能对库做任何处理，也要特别留心自己代码中的基本类型困扰）。

你看，即使一个小小的错误也会得到如此多的教训。

16.3　去除重复、选择问题和一些模糊性

目前的情况如何呢？

- Cost——对于这个类我相当满意。当然还有其他做法，不过就目前而言它已经足够好了。
- Card——对于索引卡/滚动问题，我还在尽力寻求解决方法，暂时先不做调整。可以对选择加以改善：当索引卡被点击时是否应当通知它的焦点监听者？这样可以去除它对 Background 的依赖性。
- Background——除了要更改索引卡和选择之外，其他方面我都比较满意。
- PlanningGame——设置时仍然存在重复。我的同事 Ron Crocker 指出，可以让 makeCustomerButtons()和 makeProgrammerButtons()负责放置适当的标签。但是容器和选项的所有处理仍然很麻烦。我们还要在多处调用 updateCost()。

习题 102　按钮创建。

清理 PlanningGame 中的按钮创建。

- 将标签移到 make 例程中。
- 消除构造按钮过程中存在的重复。

有关解决方案请见附录 A。

我一直想弄清为什么选项处理会如此困扰我。我想主要原因在于类的职责：我希望计划游戏屏幕负责保存按钮、背景（Background）和摘要部分。目前，这个计划游戏屏幕还跟踪了选项，因为这正是动作监听者所处理的内容。不过我认为应当让 Background 跟踪所有选项，而且有些动作监听者也应当属于 Background。

这一修改是试探性的，因此要对代码设置检查点（checkpoint），这样便可将不喜欢的结果还原至检查点。（我认为设置检查点就是保存一个版本，以便轻松地恢复，而不必将恢复工作转嫁至整个开发小组。在你所处的环境中，这可能很容易，但也可能并不容易。）

习题 103 将选项处理移至 Background。

- 令 Background 保存选项。
- 将适当的监听者行为转交给 Background（至少从 New、Split 和 Delete 中搬移）。
- 搬移 ContainerListener。

有关解决方案请见附录 A。

对于可能修改成本的按钮，成本的更新通过按下相应按钮来完成。每个命令在修改之后都必须包括一个 updateCost()调用。在当前代码中，这通过将此调用置于各个监听者中来实现。要减少这种重复性，一种做法是从一个常用父类派生出监听者子类，让父类负责完成此调用。

还有一种做法，即引入监听者风格的通知。摘要部分将监听底层 Background 中的变化或速度改变，并进行相应的更新。propertyChangeEvent 可以实现这一功能。在 Background 上，我们通常要监视的两项内容是计数（count）和成本（cost）。在此并不关心具体修改的是哪个属性，我将采用标准 bean 约定，即对于一个属性修改而言，Null 表示可以做任何修改。

这一修改是一种重构，但算不上是大型（large）重构（如果这个程序规模足够大，则是大型重构）。我并不认为以上测试足以覆盖整个程序，因此下面引入几个新的测试来确保 Background 发送了事件。由于这是一个比较大的更改，所以在开始之前要先设置检查点。

习题 104　属性更改。

转换 PlanningGame，让成本更新由 Background 中的属性变化触发，而不是通过显式的调用完成。

有关解决方案请见附录 A。

动作监听者仍存在重复：许多动作监听者在完成工作之前都需要检查是否存在索引卡。由此可以得到一种修改：除非确实需要某一按钮，否则就将它禁用。这是对用户界面的一个修改，而我们的用户很欢迎这种变化。

习题 105　启用按钮。

仅在适当的时候启用按钮（当然要先做测试）。

有关解决方案请见附录 A。

再回过头来看 Card。它仍依赖于 Background。当一个索引卡被点击时，它会查看背景，并通知背景将该索引卡移到最前面。这是一种不必要的依赖性，因为 Background 很了解 Card，而 Card 也很了解 Background。

习题 106　Card 和 Background（有难度）。

去除 Card 对 Background 的依赖性。

- 令 Card 在被点击时发出一个属性通知（如果需要，可以令它发出不同的通知）。
- 令 Background 监听这种新通知。
- 从 Background 中去除有关 ContainerListener 的内容（因为现在由 Background 来控制 moveToFront()的调用，它已经知道内容何时会更改）。

有关解决方案请见附录 A。

习题 107　清理。

再检查一遍所有测试和代码，尽可能完成所有简单的清理工作。

有关解决方案请见附录 A。

下面将现在的新代码与最初的代码进行比较。表述能力是不是增强了呢？现在的代码是不是更简单？它的结构是不是更好了？

16.4 第 3 部分：进一步改进代码

第 3 部分给出了一些进一步改进代码的想法，你可以独立完成相关练习。

习题 108 残留的异味。

代码中还存在哪些异味？

我发现，速度更新处理与其他更新处理不同。在 Card 中仍有滚动（rollover）计算，这是我不希望看到的。另外还存在一些重复，我仍然认为有必要建立一个单独的模型层。在某些方面，当前设计已经接近最佳了，设计中不让模型和视图分离也是可行的，但是我们还需要更进一步。（现在的问题是，此代码仍未表述出我们所知道的一些情况：此程序的一些部分受到用户界面的影响，但另外一些部分却不会受影响。这正是一个有关发散式改变的例子。）

如果你还不熟悉模型/视图（model/view）或模型/视图/控制器（model/view/controller）思想，可能需要了解相关基础知识：请参见 Gamma 等人所著的《设计模式》中的观察者（Observer）模式、Java 库中的 TableModel（或其他模型）或 HotDraw 框架（查看 c2.com/cgi?CrcCards 和 www.c2.com/doc/crc/draw.html 处的描述）。

我们可以从创建一个 Card 模型开始。要马上确定它是应包含标题的模型，还是应使用 Java 在 TextField 中提供的标题？如果希望 Card 有它自己的模型，则一定要小心，注意不要造成通知循环（即 Card 通知 TextField 有所变化，而 TextField 又会告诉 Card，这样 Card 又会通知 TextField，如此反复）。如果我们使用 Java 提供的模型，那么这种连接就不会那么复杂了。

习题 109 单独的模型。

A. 在进行这些修改之前确保已经保存了原有版本。

B. 应用复制受察数据，将 Card 类分解为一个 Card 和一个 CardView。（注意，这不是一个轻松的重构。这个练习也许会耗费你大量的时间。）

C. 对 Background 应用复制受察数据。

- 可以充分利用 DefaultListModel，它在通知中设置。
- 可以获得非常详细的关于列表更改的通知，但是在收到更改通知后，作为起步，我们只是重建整个列表。

- 注意，Card 的位置和选择状态都存储在视图中（这是正确的位置吗？）。如果你尝试重新生成视图，请务必跟踪这一信息。

D. 对 Plan 应用复制受察数据。

E. 将程序划分至两个单独的包：一个存储所有模型，另一个存储视图。让视图类依赖模型类，但是反之不然（可能需要为模型和视图创建单独的测试类）。

F. 这里涉及大量的工作。这是一个改进吗？做这些工作会简化将来的哪些修改？（如果这不是改进，请恢复原来的版本。）

有关解决方案请见附录 A。

习题 110　一种优化（有难度）。

最简单的通知形式如下：

```
public void setTitle(String title) {
 this.title = title;
 notify();
}
```

下面是一种更复杂的实现：

```
public void setTitle(String title) {
 if (this.title.equals(title)) return;
 this.title = title;
 notify();
}
```

A. 这在什么情况下是一种性能优化？

B. 它另外还有哪些用途？

有关解决方案请见附录 A。

习题 111　新特性。

请考虑还可以增加哪些新特性。对于这些新特性而言，实现的健壮性如何？

- 新索引卡并非沿对角线放置，而是从上到下、自左向右放置。
- 我们希望按钮担当管理者或开发小组的角色。
- 所有颜色和大小都要成为首选项。
- 计划分析应当能够检测出一次迭代中有太多工作（即负荷太重）的情况。

- 通过某种持久存储机制来保存并恢复工作。
- 允许多个地方的人同时处理计划。

你还会增加哪些特性？

习题 112　测试驱动开发。

从头开始重新实现此代码，此时使用测试优先的方法。在开发新版本时不要参考老版本。你会得到什么？

此练习的代码本来不是用测试优先的方法编写的。测试驱动开发人员的经验表明，开发过程中往往会出乎意料地得到一种不同的设计。你的情况如何？得到的代码更好吗？能设置更好的测试吗？原先的设计对你影响多大？

习题 113　测试驱动开发带来的教训。

假设你的测试驱动代码有别于你之前使用的代码，你能否重构原来的代码，使它与新代码一致？你是否需要一些未在任何书中提及的重构技术？哪些异味能够“指导”你，使你能按这一方向自然地进行重构？这会给你一些关于重构或测试驱动开发方面的启发吗？

第 17 章

下一步何去何从

重构是一种技能，这是这本书一直倡导的主旨，而这项技能要通过实践获得。尽可能寻找机会着手实践吧，并请充分利用这门技能。

17.1 参考书

参考文献中列出的所有书都值得一看。不过，如果你手边还没有 Martin Fowler 的《重构》，快去找一本。《重构》名录中所列的重构技术几乎有一半会在本书的练习中用到。工具对于重构机制的支持，表现越来越好，不过，也还需很长时间才能有效涵盖 Fowler 名录中所列重构技术的方方面面。

17.2 警告

将重构应用到实践中

知道如何重构并没有什么，除非真正应用它。尽可能使代码简洁清晰。对于 XP 开发小组而言，这是他们每天生活的一部分。但是，即使设计方法再好，也要求程序员能够很好地实现设计。

将测试应用到实践中

有一个很老的谚语（这样的谚语还有很多）：“既然不会有问题，何必费力去修补。”（很多时候，最后一个小小的修改就导致一个意外的 bug）。在程序设计中，如果信奉这条谚语，那么代码会变得越来越糟糕。

重构则试图通过两个原则打破这个规则：一方面是要安全地重构，另一方面是应用测试来验证正确地完成了转换。不要小看测试。

求助他人

在编写代码时可参考他人的意见，不论通过哪种方式，例如合作编程、请求别人完成设计/代码审阅，或是简单地请教身边的人。编写这本书让我真正领悟到了一点：几乎所有代码都有改进空间（而且有时我们还要充分利用网络来寻求帮助！）。

17.3 必做练习

每周清除一个异味

挑选一个异味，尽可能找出所有出现这种异味的地方并进行清除。每周寻找一个新的异味。

重新重构

挑选一段大小适中的代码（可以是你自己编写的，也可以是本书所提供的某个更大规模的例子）。每天都从最初版本开始，尽量在 10 分钟内完成重构。

你每天的感受相同吗？是否有更深刻的认识？

只是重构

挑选或开发一个项目。花 10 分钟进行重构（每天都在前一天的基础上开始重构）。

吸气/呼气（Inhale/Exhale）

找出表露某种异味的代码，应用一种重构技术加以解决。然后应用另一种重构技术，它与前一种重构技术的作用正好相反。重复此工作两次。由此你会更深入地体会到代码中的问题是怎样引入的，以及如何去除这些问题。

反向重构/误构（Defactoring/Malfactoring）

反向重构（Defactoring）和误构（Malfactoring）是我起的两个名字，表

示不好的重构，即这些重构会让现有代码设计更糟。取一段代码，重构它，使它出现尽可能多的异味（这比看上去要困难得多。）

除了提供重构的实践机会，这样做还能帮助你认识到，在开发过程中，什么时候你会无意识地进行一些错误的重构。

在“玩”够之后，务必恢复原来的代码。

重构套路

套路（kata）是指必须每天反复练习的一些招数，不仅适用于实践，也有助于理解重构艺术的真谛。（传统的套路可能是面对 4 个对手如何反击。）为重构开发一种套路：即应用一系列既定重构的程序。选择一系列经常看到或用到的异味和重构技术，对我来说，这可能包括基本类型困扰、某些过长的方法、某些重复的受察数据（复制受察数据），以及一些需要重新权衡的职责。

这为你提供了磨炼编辑技巧的机会，并让你更深入地认识当前所在的环境。另外，你还可以借此机会实践寻找异味和进行重构。

17.4 网站资源

以下网站是一些很有用的资源：

- groups.yahoo.com/group/refactoring。讨论重构的一个新闻组。
- groups.yahoo.com/group/extremeprogramming。讨论 XP 的一个新闻组。在此有大量与重构无关的讨论，但重构也是 XP 中最关键的实践之一。
- www.refactoring.com。Martin Fowler 为《重构》开设的网站。他在此提供了这本书的大多数名录及其他人的一些贡献。
- www.industriallogic.com。Joshua Kerievsky 的企业网站。他还有一本书：*Refactoring to Patterns*。这里还提供了特别有意思的文章和游戏。
- www.xpl23.com/rwb。总的说来，我的网站关注的是 XP，而 rwb 则是专门为这本书设置的。你可以在此找到本书中较大例子的源代码和与重构相关的文章，另外它还提供了一些链接，由此可以访问别人提出的一些重构实战练习。

第 4 部分

附录

附录 A 所选问题的答案
附录 B Java 重构工具
附录 C 重构逆处理
附录 D 主要重构技术

附录 A

所选问题的答案

第 2 章　重构周期

习题 1　小步前行

大多数重构都反映了这种思想（即使在重构中也要保证安全性）。请考虑我们给出的封装字段（Encapsulate Field）例子。即使是这样一个简单的程序，也编译了三次。或者查看抽取方法（Extract Method）。这里并不是简单地剪切和移动原代码，Fowler 的重构名录建议复制此代码，然后调整它再删除原版本。

习题 2　简单设计

A.

1. 通过所有测试。通过测试确保有效性。

2. 表述能力。显然，这一点对于将来阅读代码的读者（也包括我们自己）来说意义重大。

3. 没有重复。重复性代码会带来许多麻烦，很容易出现这样的情况，即你在某处进行了修改，但却未在其他位置做相应修改。

4. 最少的类和方法。如果所有对象的功能相当，我们更喜欢少的代码。

B. 对我而言，基本原则是尽量使读者容易理解。为了使代码能被更好地理解，出现一点重复是可以容忍的。

Robert Wenner 在与我私下交谈时曾指出，Java 本地接口（Java Native Interface，JNI）代码用于将 Java 链接至 C，其中各方法之间就存在着重复，

但是各个方法又有足够多的不同，从而避免了大家的质疑。每个方法都必须为它访问的各个函数处理不同的参数和不同的返回值。

出于表述原因，测试代码有时也存在重复。例如，对于一个所需的值，相对于将其赋给一个变量并使用该变量来说，重复这个值可能更容易。这样，在阅读代码时，你就能准确地知道它所查找的是什么，而且不必查看代码来找到相应变量，同时也能确保在此过程中不对它做任何其他修改。

第 3 章 可度量的异味

习题 3 注释

B. 所有关于代码的重要信息是否通过代码即可表达清楚？注释有无存在的必要？代码通常可以很好地表述“怎么做（how）”，但是往往不能很好地说明“为什么这么做（why）”，而且几乎不太可能说清“为什么不这么做（why not）”。

将代码发布给其他人时，有些人会发现添加 JavaDoc 注释很重要，它会提供进一步解释。

习题 4 过长的方法

A.

第 1 个代码块：out.println("FACTORY REPORT\n");

第 2 个代码块：From there until out.print ("Robot");

第 3 个代码块：From out.print ("Robot"); through write("\n");

第 4 个代码块：out.println("=======\n");

你选择的代码块可能与此稍有不同。

B.

```
public static void Report (
  PrintStream out, List machines, Robot robot) {
    reportHeader(out);
    reportMachines(out, machines);
    reportRobot(out, robot);
    reportFooter(out);
}
```

我们不会止步于此，但这是一个很好的开始（还可以移至一个 Report 类，或者在 Machine 类和 Robot 类中设置 report()方法）。

C. 抽取一个单行方法是否有意义？如果表述能力会增强，那么答案就是肯定的。

习题 5　过大的类

A. 它要完成大量工作，有些是继承来的，但它似乎要承担各种各样的职责。

B. 分析列出的所有方法，将它们划分至 5—10 个主要的责任领域。

我划分的领域如下（你的可能有所不同）：

Columns（列），Editing（编辑），Rendering（呈现），Model Selection（模型选择），Appearance（外观），Notification（通知），Other（其他）。

C. 可以通用化方法，这样就不是那么需要对称的行和列函数了。可以抽取出一些辅助类，让它们自行承担自己的职责。其中一些方法似乎只是向表或列模型寻求答案。外观属性可以进行单独的管理，或者通过一个更简单的接口来管理。

D. 我不是 Smalltalk 专家，但是我认为造成这种差别可能有如下原因：

- Smalltalk 的动态类型能更容易地定义通用方法。
- Smalltalk 的发展历史更为久远，而且有更多的可行方法。
- Smalltalk 的解释环境（更开放）允许人们进行更多的试验。

习题 6　过长的参数表

A. x/y/width/height 看上去像一个 Rectangle 对象。

value/extent 则像一个范围（注意，限制参数使用的是 min 和 max，而不是另一个范围。如果有一个单独的 Range 类，那么一切会更清晰）。

x1/y1/x2/y2 则像是另一个 Rectangle。

B. 在某些方面，这反映出一种动机，即力图使类更通用，因此传递它能处理的所有内容。要使图形之类的对象成为无状态的，使用大量参数有助于实现这个目的。

C. 行和列多次同时出现，由此我们可能会想，是否能使用某种定位器（locator）简化这个对象呢。这可能会简化选择，而且还能解决一些难题，如跨两行的单元格等。

习题 7 异味和重构

A. 注释（Comments）
B. 过大的类（Large Class）
C. 过长的方法（Long Method）
D. 过长的参数表（Long Parameter List）

B ——复制受察数据（Duplicate Observed Data）
B ——抽取类（Extract Class）
B ——抽取接口（Extract Interface）
A 或 C ——抽取方法（Extract Method）
B ——抽取子类（Extract Subclass）
A ——引入断言（Introduce Assertion）
D ——引入参数对象（Introduce Parameter Object）
D ——保持对象完整（Preserve Whole Object）
A ——重命名方法（Rename Method）
D ——将参数替换为方法（Replace Parameter with Method）

习题 8 触发器

A. 哪一种你最常看到或最常创建？每个人的答案都会不同。过长的方法和（没什么作用的）注释就是我最常看到的异味。其中，过长的方法可能是我最常犯的错误。

B. 对于这些可度量的异味，可以为自己设定一个阈值，如果超过此阈值，就有必要检查所做的工作了。例如，如果一个方法超过 7 行，我就会反复检查两遍，而且对于方法体中的所有注释我都会质疑。请定义你自己的触发器。

补充点 I 异味和重构

I1.2 修复大多数异味的重构技术

搬移方法（Move Method）、抽取类（Extract Class）、搬移字段（Move Field）和抽取方法（Extract Method）。

I1.3 未提及的重构技术

未提及的重构技术有很多。有些只是代码处理，这并不算严格的异味，重构技术只是提供了一种能在两种合法做法之间进行转换的安全方式。另外

一些则有些特殊（特别是“大型”重构）。还有一些则是应用其他重构技术时的中间步骤，对应其他重构的异味也会触发这种重构。

I1.4　其他异味

每个人列出的结果可能各不相同。我认为还需要关注以下异味：

- 模型和 UI 纠缠（Intertwined Model and UI）——复制受察数据、将域与表示分离（Separate Domain from Presentation）。
- 强制转换（Cast）——封装向下强制转换（Encapsulate Downcast）。
- 不清晰的表述（Unclear Communication）——移除对参数的赋值（Remove Assignment to Parameter）、将错误码替换为异常（Replace Error Code with Exception）、将异常替换为测试（Replace Exception with Test）、将魔法数替换为符号常量（Replace Magic Number with Symbolic Constant）、分解临时变量（Split Temporary Variable）。
- 条件逻辑（Conditional Logic）——合并条件表达式（Consolidate Conditional Expression）、合并重复的条件表达式（Consolidate Duplicate Conditional Expression）、引入 Null 对象（Introduce Null Object）、将错误码替换为异常（Replace Error Code with Exception）、将异常替换为测试（Replace Exception with Test）、将嵌套条件式替换为卫哨子句（Replace Nested Conditional with Guard Clause）、将条件式替换为多态（Replace Conditional with Polymorphism）。

第 4 章　命名

习题 9　命名

- addItem(item) ——内嵌有类型。
- doIt()——可能很好，也可能表达得不够清晰，具体取决于上下文。
- getNodesArrayList()——内嵌有类型。
- getData()——表述能力差（可能）。
- makeIt()——表述能力差（可能）。
- multiplyIntInt()——内嵌有类型。
- processItem()——内嵌有类型或表述能力差。
- sort()——很好。
- spin()——很好（取决于具体域）。

习题 10 评价命名

如果说有哪个领域完全取决于个人喜好，那么非命名莫属。因此如果你的答案与我的不同，不要感到奇怪。

A. clear()或 erase()似乎都不错（具体要看库或其他代码的使用）。deleteAll()则有些过时。wash()如果用于模拟擦洗玻璃可能还好，不过似乎也只能这么用。

B. push()是传统命名。如果集合库所用的是 add，那么 add()也不错。用 insert()则似乎过头了点，因为栈并不会将元素插在中间。addToFront()也很奇怪，队列有队头（front），但栈有的是栈顶（top）而不是什么头（front）。

C. cut()暗示此文本要在某处保存用于粘贴。delete()可能是最好的。clear()和 erase()也行，不过对我来说，它们的意思似乎是要用于整个文档。

D. equals()显然是 Java 用词。如果 equals()出于某种原因不可用，那么 identicalTo()也可以。matches()可以使用，但是有一些歧义，即暗示这可能是一个模式匹配。compare()是最糟糕的一个，其他用词都使我们了解到“如果相等则返回 true”，而 compare()则没有说明会得到什么答案。

习题 11 XmlEditor

A. 我选择的父类是 Editor。

B. 接口可以叫 Editor，XmlEditor 的父类可以叫 AbstractEditor（这里有发展空间，对此我很满意。也有可能类只有一个接口 Editor 和一个实现 XmlEditor）。

第 5 章 不必要的复杂性

习题 12 着眼今天，还是放眼未来

认为“今天只做今天事”（即仅针对当前需求进行设计）更好的理由	认为“未雨绸缪”（即今天的设计要考虑到未来的需求）更好的理由
如果今天只完成今天的设计，则成本较低。 不用考虑某个特定方向的发展需求（因此不必回溯）。 今天不必维护明天的代码。 代码仅完成基本需求时更容易理解。	当需求仍在我们脑海中时，可能更易于充分建立完整的类。 针对未来的需求进行开发，能帮助我们更好地理解当今的需求。

这就像下赌注：一般来说，今天只做今天事，明天再去处理明天的事情，

这样做是不是成本更低呢？或者通用化设计本身是不是就足够好了，提前的“花销”是物有所值的？

最伟大的硬件设计师之一 Gordon Bell 曾说过：“计算机系统中最廉价、速度最快且最可靠的部件尚不存在。”(引自 Bentley *More programming Pearls*，第 62 页)

第 6 章 重复

习题 13 两个库

A. 一种策略是：

- 定义一个新的 Logger，其接口与 JDK 1.4 Logger 兼容。这可以是一个简化的层(layer)接口、一个有兼容接口的类(将来会是 JDK 1.4 Logger 的一个子类)，也可以是新类的一个简单实现。
- 让原来的 Logger 调用新 Logger。
- 修改 Log 和它的调用者，使 Log 变成一个新的 Logger，这样就可以删除 Log 类了。
- 将 Logger 修改成像新的一样，这样也可以删除 Logger 类了。

这个工作完成起来可能相当慢（也就是说，现在要为修改后的新代码使用新的 Logger）。注意，这在概念上会增加我们的负担。也可以使用自动化支持使这个过程更简单。

习题 14 属性

A. 使用抽取方法提取一个查找属性的例程，将其转换为一个整数，并验证它是否为正数（请分步完成：包括第一、第二和第三个版本）。

你可能认为，即使会抛出异常，也可以设置 monitorTime 和 departureoffset。这会减少对临时变量的需求。

最终结果可能如下：

```
checkInterval = getProperty("interval");
monitorTime = getProperty("duration");
departureOffset = getProperty("departure");

if ((monitorTime % checkInterval) != 0)
  throw new MissingPropertiesException(
    "duration % checkInterval");
```

```
if ((departureOffset % checkInterval) != 0)
  throw new MissingPropertiesException(
    "departure % checkInterval");
```

然后便可以抽取一个单独的方法来强制%限制。

习题 15 模板示例

A. 重复

- 这二者几乎完全相同，只不过一个对应%CODE%，另一个对应%ALTCODE%。不过一个写到一个字符串中，而另一个则写到一个输出流中。
- 字符串字面量%CODE%和数字常量 6 反映了同一个问题（魔法数），%ALTCODE%和 9 也是这样。
- 各个部分最后得到的字符串的结构是类似的：都是追加一个前缀、主体和后缀。

B. 去除重复

- 将魔法数替换为符号常量（Replace Magic Number with Symbolic Constant）和函数。
- 抽取方法。
- 查看字符串的统一（即匹配）和输入/输出（I/O），使方法更相似。

C. new String()

- 它的作用是什么？new 创建一个新字符串，内容与原字符串相同。虽然这个新字符串 equels()等同于原字符串，但它们是不同的对象。
- intern()方法将返回字符串的一个唯一实例，所有与之内容相同的其他内部字符串都对应这一个对象（默认情况下，字符串无需共享）。
- 它用在这里合适吗？答案是否定的。因为我们不会比较内部字符串，new String 部分是多余的。

习题 16 复制受察数据

A.

- 重复性通常并没有最初想的那么严重。一般说来，域（domain）对象有它自己的对象表示，而显示部件（widget）最后会保存一个字符串或其他显示表示。

- 用户界面通常是程序中最活跃的部分，而域类（在开发过程中）则一般很少调整。
- 将域信息放在绑定它们的显示部件内。域类应当能够独立地修改它的值，不管值是否显示在屏幕上（请参考观察者模式）。
- 将域和屏幕类混在一起会使域依赖它的表示，这是一个缺陷。最好让二者分开，这样域类就可以采用完全不同的表示了。

B. 性能可能会有所变化。当被置于对象中时，域类会使用显示部件方法来更新其值。这通常会比较慢，因为此时必须考虑到缓存、屏幕更新，等等。

另一方面，同步的开销会变得比较大。有时必须找到某种方法来降低这种通知成本。在某些情况下，将域类和显示部件分离还可以使显示部件不被显示。

习题 17　Java 库

A. 示例。

- AWT 与 Swing：包含两个完整的显示部件。
- Collections 与 Vector/HashTable：存在两个集合库。
- 事件监听者：存在许多密切相关的变种。
- GUI 类中的重复方法。例如，show()和 setVisible()就重复了。
- 其他示例，在此不一一列举了。

B. 为什么？

- 最常见的原因可能是历史遗留问题。人们要依赖 Sun 公开的接口进行开发，因此 Sun 不愿意对这些接口进行修改，这是可以理解的。Sun 不是不断地调整，而是不断地增加内容，即使目标或代码出现重叠也不去管。
- Java 提供了一种过时（建议不使用）标志，可以为某些重复设置这一标志，警告客户这是一种老式解决方法，而目前已经有了更好的解决方法。
- 对于 Java 库这样的大规模库，需要许多人协作，而他们往往不能很好地协调，没有意识到存在着重复的工作。
- Java 的库是公开的（有许多书和文章都对此做了详细介绍），这意味着人们对这些库的挑剔达到了无可复加的地步。换句话说，其他（语言）库其实也存在着同样的重复性，只是不像 Java 库这样受瞩目。

习题 18 点

A. 二者均使用基于 maxX 和 maxY 值调整的点。

B. 采用替换算法（Substitute Algorithm）使两个类按同样的方式计算调整，然后使用抽取类提取一个 WrappingPoint 类。

C. 搜索重复有助于识别这种情况。你可以创建一个测试，找出差代码中存在的 bug。在修复过程中，尽量朝着得到较好代码的方向发展，然后使用处理重复的重构技术来清除这些重复。

习题 19 表达式

A. 可以对返回（return）语句使用抽取方法，将 return v1 + v2 变成 return value(v1, v2);，对*做同样的处理（即也使用抽取方法）。这就使 value()例程变得一样了，从而便可以使用上移方法（Pull Up Method）将它们放到一个 Composite 节点中了。（子类中仍然要有两参数例程。）

第 7 章 条件逻辑

习题 20 Null 对象

A. 空字符串并不是在任何情况下能代表默认值。

B. 为 Bin 抽取一个新的类，这样有可能提供你所需的灵活性。

习题 21 条件表达式

A.

```
if ((score <= 700) &&
   ((income < 40000) || (income > 100000)
     || !authorized || (score <= 500)) &&
   (income  <= 100000))
  reject()
else
  accept();
```

B.

```
boolean hasMidRangeIncome =
  (income >= 40000) && (income <= 100000);
boolean hasHighIncome = (income > 100000);

boolean hasHighScore = (score > 700);
boolean hasMidScore = (score > 500);
```

```
if (! (hasHighScore
     || (hasMidRangeIncome && authorized && hasMidRangeScore)
     || hasHighIncome) )
  reject();
else
  accept();
```

C.

```
if (score > 700)
  accept();
else if ((income >= 40000) && (income <= 100000)
     && authorized && (score > 500))
  accept();
else if (income > 100000)
  accept();
else
  reject();
```

D.

```
boolean acceptable(int income, int score, boolean authorized)
{
  if ((score > 700) || (income > 100000))
    return true;

  if ((income >= 40000) && (income <= 100000)
     && authorized && (score > 500))
    return true;

  return false;
}

if (acceptable(income, score, authorized))
  accept();
else
  reject();
```

F.

	高收入		中等收入		低收入	
	AUTH=Y	AUTH=N	AUTH=Y	AUTH=N	AUTH=Y	AUTH=N
高分	接受	接受	接受	接受	接受	接受
中等分数	接受	接受	接受	拒绝	拒绝	拒绝
低分	接受	接受	拒绝	拒绝	拒绝	拒绝

或者：

	高收入	中等收入	低收入
高分	接受	接受	接受
中等分数	接受	如果授权则接受	拒绝
低分	接受	拒绝	拒绝

习题 22 switch 语句

A. 如果工作只有这些，则不必去除 switch。不过很显然的是，如果为操作设置 print()和 do()方法，则需要消除类型字段。

习题 23 工厂方法

B. 你对此可能有不同的想法，不过在开始使用整数枚举是合适的。

C. 使用动态类加载有哪些优点？

- 代码更简单（没有条件逻辑，仅在一处创建类）。
- 代码中的直接依赖关系更少（不命名实际的驱动类）。
- 所发布的代码更小（不再需要发送调试驱动类——因为没有直接依赖它的代码了）。
- 可以动态加载新的驱动类，而无需重新编译整个系统。

D. 这种修改有哪些缺点？

- 性能可能会稍稍降低。
- 简单的文本扫描将无法发现代码中的所有依赖关系。
- 配置较困难。如果命名不正确或者 CLASSPATH 不合适，系统都有可能无法运行。

补充点 3 设计模式

I3.1 模式

- 抽象工厂模式（Abstract Factory）：将构造函数替换为工厂方法（Replace Constructor with Factory Method）（需要执行好几次）。
- 工厂方法模式（Factory Method）：将构造函数替换为工厂方法（Replace Constructor with Factory Method）。

- 享元模式（Flyweight）：移除设置方法（Remove Setting Method），将值改为引用（Change Value to Reference）。
- 解释器模式（Interpreter）：抽取方法和搬移方法（从 Composite 中）。
- 观察者模式：将域与表示分离出来，复制受察数据。
- 状态模式（State）：将类型码替换为状态/策略（Replace Type Code with State/Strategy）。
- 策略模式（Strategy）：将类型码替换为状态/策略（Replace Type Code with State/Strategy）。
- 模板方法（Template Method）：构建模板方法（Form Template Method）。

Joshua Kerievsky 正在写一本有关这个主题的书，名为 *Refactoring to Patterns*。可能在你看到这些内容时，他的这本书已经出版了。

第 8 章 数据

习题 24 可替代表示

金额[使用费美国货币单位，其中 100 美分=1 美元，美分（便士）是最小货币单位]：

- 美分数（整数）。
- 一对整数，形如 long long。
- 使用小数类型。
- 可以记录便士的小数部分（有时按 1/10 美分来管理）。
- 字符串。

位置（即在列表中的位置）：

- 整数。
- 如果只有一个感兴趣的位置，可以通过两个列表来管理此表（从外部看），一个包含该位置之前的元素，另一个包含该位置之后的元素。
- 该位置上的元素。

范围：

- 第一个和最后一个索引。
- 第一个索引和长度。

社会保险号（身份证号为“123-45-6789”）：

- 字符串。
- 整数。
- 三个整数。

电话号码：

- 字符串。
- 整数。
- 两个数：区号和当地号码。
- 三个数：区号、交换机和最后 4 位数字。

这里只考虑了美国的电话号码。如果增加国际电话号码支持，可能会更复杂。另外，还可能需要支持分机号码。

街区地址（“123 E.Main Street”）：

- 字符串。
- 多个字段。
- 物理坐标。
- 标准地址（标准简称）。
- 一个标准地址列表中的索引。

ZIP Code（邮政编码）：

- 字符串。
- 整数。
- 两个整数（美国邮政编码目前使用 ZIP+4 或 123456789）。
- 一个标准编码列表中的索引。

习题 25 讨论

这取决于屏幕和数据库之间发生了什么，如果这确实是一个要求录入的窗体应用程序，即要从屏幕上的这个字段取值，然后放到数据库中的相应字段中去，那么可以不使用面向对象方法。但是随着增加越来越多有关 ZIP 编码的函数（验证、计算运货距离、确定路由等等），使用面向对象方法会有所帮助。

习题 26　迭代器（Iterator）

在使用迭代器模式时，代码不会太依赖 for 循环方法，因此很少将整数用作位置计数器。

习题 27　编辑器（Editor）

```
assertEquals(___, editor.fetch(1));
```

A. 对于以上提供的接口，你会用什么字符串来填入_____？

“a”

B. 对于变量名（firstParendPosition），你希望改用什么字符串？它的作用是什么？

“(”

有时记录文本的位置会很有用，即使是在最前面插入文本。例如，程序设计编辑器就可以跟踪每个方法声明的位置。

C. 问题的关键在于 int 被用作一个位置索引，请给出另外一种方法。

不要提供死整数，而是提供 Position 对象，但让编译器拥有这些对象。当文本更改时，编辑器就会更新 Position 对象。对象的拥有者并不了解这一点，它们只知道可以得到一个对象，或者将其移回到前一个位置。

D. 将你的解决方案与备忘录（Memento）设计模式进行比较。

备忘录设计模式使用一个不透明（opaque）对象。在这种情况下，编辑器可以知道其中有什么，但是客户肯定是不知道的。客户无法直接处理备忘录模式，必须将它交还给主对象来使用。

习题 28　库类

- 都有公共数据成员。
- 都是 Object 类的子类。
- 大多数是相当稳定的。
- 对于在 Java 中的使用，大多数库类都有很容易理解的含义（仅 GridBagConstraints 除外）。
- 库类可能有很多。

习题 29　Color 和 Date

A. Color 构造函数显示了三种不同的表示：一个由整数组成的 RGB（红/

绿/蓝）三元组、一个存储所有值的整数、一个由浮点数组成的 RGB 三元组。颜色值也可以是一个 HSB（色度/饱和度/亮度）三元组。

Date 可以存储为一组值（年、月、日，等等），也可以存储为一个整数值（自某个事件以来的秒数或毫秒数），还可以存储为文本。

B. 由于客户不直接访问这些字段，所以他们不会隐式地修改该对象实例（即不通过其方法修改）。

习题 30 适当的名字

A. Client1 生成一个字符串，它采用“名在前”格式。Client2、Client3 和 Client4 生成的则是“姓在前”的字符串。针对这两种情况，在 Person 中建立相应的方法。然后可以用封装字段将这些字段设为私有。

B. 合并重复代码后，处理这些修改会变得更容易。

第 9 章 继承

习题 31 集合类

A. 只读集合抛出一个异常，因此这是一种坦率的拒绝。

B. Java 并不需要对所有异常进行显式声明，而且这些是不需要声明的类型。

C. 还有一种方法是让所有集合类都采用以下两种方式：一种针对只读，另一种针对非只读。这就会带来大量近乎重复的类。相对而言，拒收的遗赠反而不那么严重。

习题 33 Swing 库

A. 此接口提供有关对象的基本信息，而且有助于解除客户与具体类的耦合。另外，Java 支持接口有多重继承，但是对于类则仅支持单重继承。分离出接口后，类就可以增加对接口的支持了，而同时它还能保持其原来的继承体系。

B. 尽管此框架只有类的一个实现，但显然框架的用户必将引入其他的类，而抽象类显然可以构成这些类的根。如果没有其他类了，也可以分离出抽象类，使代码更具表述性。

C. 你可能并未采用这种结构，但这种结构能使你的代码焕然一新。

第 10 章　职责

习题 34　依恋情结

为 Machine 和 Robot 提供它们专用的 report()方法。

习题 35　遍历列表

A. 代码依赖于机器作为列表进行维护这一事实。

B. 列表的存在可能是 Machine 或 Line 类的一个秘密。

习题 36　中间人

A. 移除中间人是一个改进吗？我认为并非如此。这样做只会降低代码的清晰性，而且还可能带来其他不利影响。如果为事件处理程序提供一个 ArrayList，那么修改会有更大的空间。采用 Queue 类，我们知道这是一个队列，通过它可以准确知道在哪里处理委托（delegate）。如果没有 Queue 类，就必须查看所有事件处理程序，以确保没有用到非队列操作。

习题 37　Cart（购物车）

B. 为 Purchase 增加 cost()和 days()。

C. Cart 不再需要访问 item()或 shipping()了。因此在为相关对象创建方法时，隐藏委托可以扩大接口，但是由于客户无需导航，我们也可能需要缩小（窄化）接口。

D. 删除 item() 和 shipping()。

E. 根据修改的顺序，结果可能各不相同。

习题 38　Swing 中的 Tree

A. TreeModel 打破了类的依赖关系，因为 JTree 和 DefaultTreeModel 都依赖 TreeModel，但是 JTree 不依赖 DefaultTreeModel。

B. 包依赖性依然存在，因为 JTree 在 javax.swing 包中，它依赖 javax.swing.tree 包中的一个类（可以这样考虑包依赖性：它指示了包中的所有内容要与其所依赖的内容同时提供。在包这一级别，JTree 同时依赖于接口和具体类）。

C. 打破包依赖性有以下两种方法。

- 一种方法是将 TreeModel 移到 JTree 所在的包中，使 javax.swing.tree 依赖于 javax.swing,，但反之不成立。这是一个改进，因为如果实现你

自己的树模型类，将不再需要 javax.swing.tree 包。不过，问题仍然存在。如果想创建一个无需 JTree 显示的树模型，则仍需要在包这一级别有依赖性。

- 另一种方法是将 TreeModel 提取到它自己的包中（不依赖于任何其他内容）。让 javax.swing 和 javax.swing.tree 都依赖此包，但是 javax.swing 和 javax.swing.tree 相互之间不存在任何依赖关系。

类加载不是说一次加载一个包，因此包依赖性是一种颇具技巧的构造。但是我认为考虑这些依赖性很有必要，因为由此可以发现一些异常。在这里本应稳定的内容要依赖于调整的内容，而在正常情况下，应当是调整的内容依赖于稳定的内容（有关依赖性规则的更多内容，请参见 Robert Martin 所著的 *Agile Software Development:Principles, Patterns and practices* 一书）。

D. 如今已不需要担心磁盘空间问题了，但还是需要考虑库依赖性，原因有两点：首先，过分的依赖性可以暴露出我们在哪里处理不当，没有正确理解类所需完成的基本工作。其次，并非所有系统都是有大硬盘的桌面系统。J2ME 就面临着这一挑战，所以它引入了最小量级的库。作为补充练习，可以考虑一下，如果你希望有一个最小库支持 Object，那么需要添加哪些类（请在类级和包级进行分析）。

第 11 章 相关改变

习题 39 CsvWriter

A. 一是要决定在哪里写（where）；二是要决定怎么写（how）。

D. 哪一个更好？可能两者差异并不大，因为这两种方法采用的都是测试驱动思想。对于只生成字符串的版本，测试起来要稍微容易一些。

习题 40 CheckingAccount

A. 一是要确定 CheckingAccount 和 Transaction 之间的关系；二是要确定如何表示金额。

习题 42 复制受察数据

这可能是一个问题，也可能不是。是否分离模型/视图常常要看实际需要，需要单独地修改模型和视图。继承体系开始可能是并行的，但不会总是这样。（一个模型可能有多个视图，而另一个模型可能没有任何视图。）

你可以看到通用模型应运而生（请考虑如何将 TableModel 用作通用模型来处理表之类的事情）。

习题 43　文档

A. 它影响了类继承体系中的每一处。

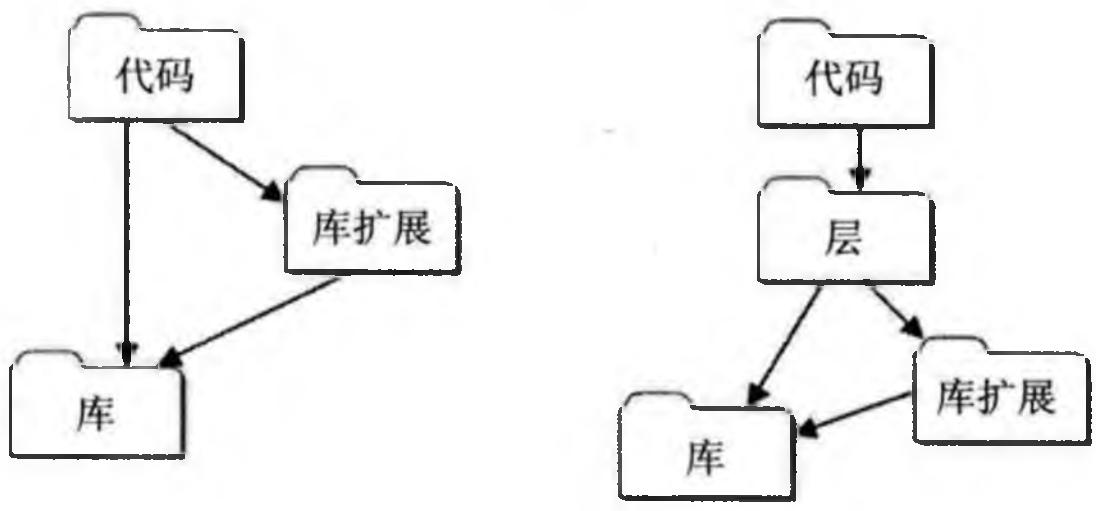
B. 这是否是一个改进取决于它如何使用，在此我们没有足够的信息作出判断。

C. 简要/详尽选择，以及压缩与否，都作为包装（wrapping）类型。

第 12 章　库类

习题 44　层

A.



B. 在第一种情况下，代码直接依赖于库。在第二种情况下，代码直接依赖于层，只间接依赖于库。

C.

- 概念完整性：依具体情况而定——好的层接口可以改进我们思考问题的方式。
- 可移植性：更好，修改将集中在层中。
- 性能改进：存在两种情况。使用层会存在小的开销，但是层也能缓存数据或提高性能。
- 便于测试：在层中测试可能更容易，特别是在层接口窄化时。测试实现也可能更加容易。

D. Java 并未提供增强这一点的语言机制。可以借助外部机制（例如，检查引用加层包的工具）。

习题 45 树

A. 外来方法似乎更为容易。既然我们必须修改代码来实现此接口，那么可以将该方法置于代码中。本地扩展可能是 MutableTreeNode 的一个新的子接口，这似乎更合适，因为树客户可以使用 userObject，而无需关心节点类型。

习题 46 字符串

A. 将一个类声明为 final 就不用为它创建子类了，但这样也不能引入本地扩展（Introduce Local Extension）了。

B. 除了有其他原因，将 String 类设为 final 还与安全性有关。在 Java 中，字符串是不可变的，设置后就不能再改变。有些安全性检查依赖于这一点。子类可能会破坏这种不可变性（同时也可能会提升性能。子字符串操作可以指向字符串中不会更改的部分）。

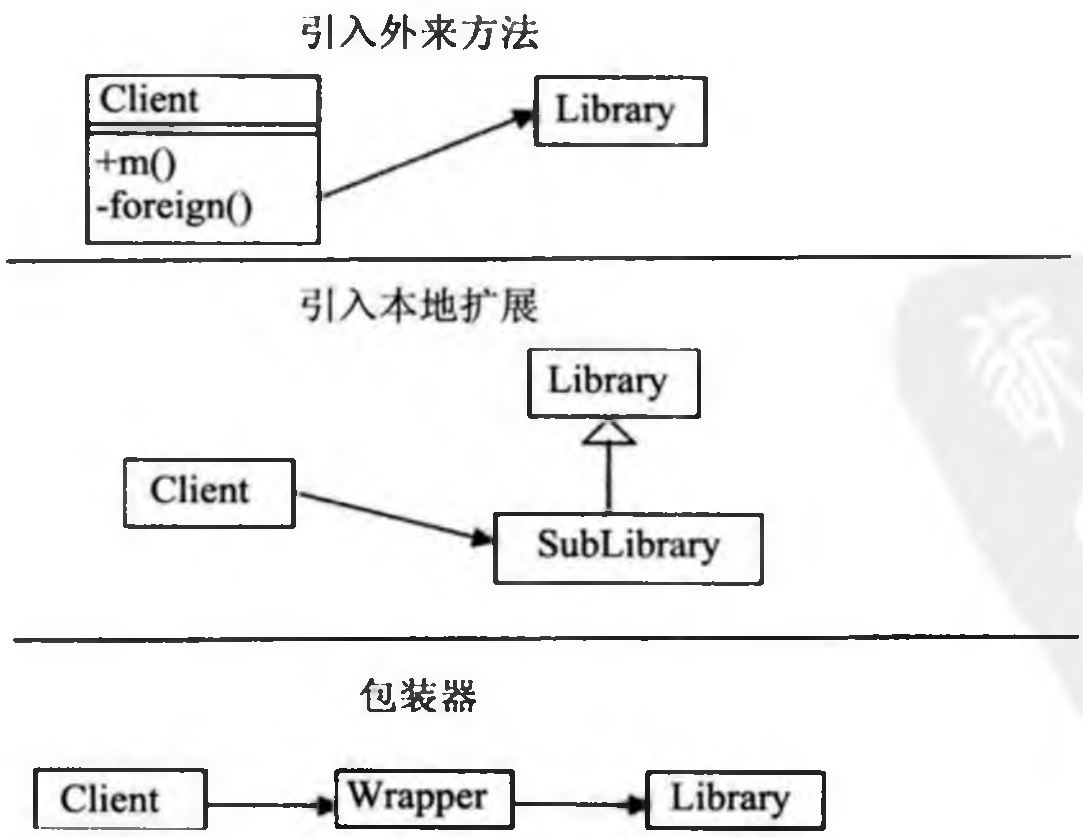
习题 47 过滤器

B. 实现 Enumerator。

C. 使用一个枚举器（待过滤的枚举器）。

D. 定义所需的方法（hasMoreElements() 和 nextElement()）和一个新方法 isvalid()，新方法将用于覆盖子类。

习题 48 图表



习题 49　缺失的函数

为某个类引入一个外来方法可能是最简单的操作了。

补充点 4　重构构成形式

I4.1　动词和名词

填写下表，用横线（—）表示没有意义的项，用加号（+）表示 Fowler 的《重构》名录中列出的重构技术，用星号（*）表示有意义但未列在重构名录中的重构技术。

	抽取	内联	搬移	重命名	上移	下移	隐藏	暴露
字段	—	—	+	*	+	+	*	*
方法	+	+	+	+	+	+	+	*
类	+	+	*	*	—	—	*	*
接口	+	*	*	*	—	—	*	*
子类	+	+	—	—	—	—	—	—
超类	+	+	—	—	—	—	—	—
继承体系	+	*	—	—	—	—	—	—

有些还存在争议，特别是考虑到内部类时。

第 13 章　一个数据库例子

习题 50　数据库异味

A. 潜在问题：

- 没有指示出各个表中的主键是什么。
- ID 作为主键存在不一致的使用(两位同名的 John Smiths 可以选课吗？)
- 除了排课表（Schedule），其他地方不会出现学生信息。
- DaysTimes 是非原子的，它还可以分解为更小的部分。

设计表的一个基本原则是各行依赖于主键——要求是整个主键，而且除了主键之外没有其他决定因素。

B. 如何修复？

- 确定主键。Course：Name（课程名）；Offering：ID（授课 ID）；Schedule：Name（学生名）+OfferingID（授课 ID）。
- 为 Course 和 Student 提供 ID 字段。
- 抽取一个 Student 表。
- 将各段时间放在它自己的行中（也可以放在另一个表中）。

习题 51 重复

A. 重复：

- 每个类都加载数据库驱动程序（以防万一）。
- 每个类都定义一个 create()方法（类似但并不完全相同）。
- 许多 find()和 update()方法是相似的。
- 重复建立了测试类。

测试用例可以更好地表述。（如果在不太清楚的断言上加一个解释字符串会如何？）

发散式改变：类同时依赖于数据库和域。

Course 和 Offering 都包含大量公共访问方法，而它们自己的行为却少之又少。可能缺失一些集合类。

B. create()在内存中创建对象，这样 all()或 fmd()就可以找到一个现有副本。代码需要处理内存中的对象，然后调用 update()来写对象。删除对象则没有一般的方法。

习题 52 应用

B. 有必要创建一个类来表示表。开始时让这个类作为一个超类可能最简单，但从长远来看，这个类很可能会发展成域对象所在继承体系中的单独一部分。

习题 53 数据库层

A. 先做一些清理工作再定义一个单独的持久存储层可能会更容易。

B. 有了层，单独地测试域模型和数据库代码就更容易了。基于内存的数据库可以提高许多测试的运行速度，因为不是所有测试都需要基于磁盘的数据库（速度慢但是安全）。

C. 不应暴露任何 JDBC 内容。

习题 54　查找

B. 这可能要深入到开发中去。

C. 可以将对象而不是主键置于映射中。

习题 55　多个打开查询

A. 这很可能会影响大量代码。

B. 一种方法是只加载相关对象的主键。当客户获取相应字段时，就会基于所存储的主键完成一个查找。还可以缓存引用（这是可选项）。

习题 56　计数器

如果同时有多个进程或线程在访问数据库，就会出现问题。通常的解决方案是引入事务。（有些数据库可能有针对计数器特殊用例的简单解决方案，但一般都需要事务。）

习题 58　数据库重构

参考 Martin Fowler 和 Scott Ambler 在这个领域所做的研究。

习题 59　域类独立性

理想情况下，数据库类依赖于数据模型，反之则不然。

第 14 章　一个简单的游戏

习题 60　异味

- 将棋盘表示为一个 StringBuffer，它可以是一个新类。
- 设置一个 Player 类可能会有好处。
- 在此存在大量魔法数。
- 在此存在多条复杂的 if 语句。
- 存在大量重复，特别要注意获胜计算。
- 参见习题 61。

习题 62　合并循环

如果两个循环的范围相同就最简单了。

很重要的一点是：第二个循环中的第 i 项不依赖于第一个循环中第 i 项之后的所有项。

第 15 章 名录

习题 77 计算

A. 这样写似乎很自然，但是名录必须知道查询的情况。

B. 查询必须知道名录的情况。

C. 名录和查询无需相互了解，但是它们必须暴露各自的一些信息。

习题 80 权衡异味

这是基本类型困扰，因为我们的查询本来只是一个字符串。这也可以被认为是一种依恋情结或不当的紧密性，因为 tostring()方法暴露了查询（它的字符串）的内部详细信息，而且是由其调用者做出决定，而不是由查询来自行决定。

习题 84 性能

由于查找（查询）操作比增加操作多，所以如果能以牺牲增加操作的性能为代价来提高查找的速度，就能够加快整个系统的速度。也就是说可以在最后一次执行增加之后再处理名录，这样就能提高查询速度。

使用这种建立交互的方式，名录并不了解查询要做什么。加快查询的一种办法就是进一步耦合它们。假设查询要告诉名录“下面某个单词必须出现在我可能选择的商品中”，然后我们就可以缓存一个单词映射来代替具体商品，这样需要查找的商品就会少很多。

习题 88 商品和集合

A. 只将 StringQuery 放在名录中，其他均在查询结果之外构造。

B. 我们当然想改为使用 Set 类型。同样地，当执行完增加操作后，这样做就能做好充分准备。可以重新设置一个从词（即 StringQuery 的查询字符串串）到商品集合的映射，这样就可以将简单的查询变成一个查找而不是一次扫描。

C. 解释器（Interpreter）模式——将 matchesIn()看作计算方法，而名录是上下文。此计算可以“冒泡”得出查询结果。

习题 89　FilterEnumerator

B. 这是一个装饰器（Decorator）模式示例（也可以认为它是一个迭代器模式的示例，而且还不限于此）。

C. 最大的问题是枚举器暗示我们必须查看每一项。可以采用其他一些方法来避免这个问题。

第 16 章　计划游戏模拟器

习题 91　异味

A. 异味如下。

过长的方法：

- Table 中有几个方法都太长了，特别是那些嵌有动作代码的方法。很难看出布局何时结束，动作何时开始。
- Card 中也存在着类似的问题。特别是构造函数，它似乎太长了。

过大的类：

- Table 和 Card 的负担都过重（在此没有单独的模型，因此这些类同时担任了模型、视图和控制器的角色）。

重复性代码：

- 建立各个按钮的代码都颇为相似。
- 每个动作似乎都在重复维护选项中的某些工作。
- Table 和 Card 都设置了边框颜色。
- 在 Card 中，设置标题和文本的工作是类似的。
- 成本的字符串名出现了多次。
- 每个成本名两边都有空格（也许标签可以设计为本身就包含空格）。
- 我相信经过一定清理之后还会有一些重复性代码。

发散式改变：

- 不能单独修改 GUI 和动作。没有 GUI 就不能轻松地运行。

霰弹式修改（修改一处涉及多处调整）：

- 将加亮代码（索引卡边框）放在两个不同的对象中。

依恋情结：

- 同样地，加亮代码属于 Card，但是 Table 也需要它。
- 成本信息在 Table 中计算，为此要把 Background 中的一些内容抽取出来。
- 摘要和计划报告（以及许多动作）都要从 Background 中抽取出信息。

数据泥团：

- costLabel 和 cost 变量似乎总是一起出现。
- lastX/lastY、x/y、getx()/getY()都处理成对的值。

基本类型困扰：

- 对 x/y 用 Int。
- 对速度用 Int。
- 对 BorderLayout 用字符串（定义了一些常量）。
- 对成本标签用字符串。

switch 语句：

- rotateCost()方法用了一条 if 语句，这实际上是一条 switch。

懒惰类：

- Background 是一个非常小的类。可以看到，某些建立工作被分派给了调用者。但是我不打算称之为懒惰类。由于 Table 对 Background 组件存在着依恋情结，所以我打算将某些方法移到这里来。

过分一般性/死代码：

- 这里有两个未用的变量：budget 和 tabs。

临时字段：

- （Table 中的）选项就存在一些这样的问题（有时设置了，有时却为 Null）。

不当的紧密性：

- Table 要查看 Background 中的组件列表。

- Card 对它的 Background 有一定的了解（它在 moveToFront()中会用到这些信息）。

还有：

- 对名字需要做一些调整。Table 这个名字尤其糟糕，Updatecost()是在更新一个摘要而不仅仅是成本。
- ContainerListener 似乎不够得力（所有选择也都有此问题）。
- 每个动作会调用 updateCost()来更新摘要部分。（用事件模型来处理是不是更清晰呢？）
- 统计索引卡数和换行数（以便设计初始位置）的方法也不怎么样。同样地，你需要考虑有多少配置应放在组件中，而多少配置应放在外面。
- 这里没有单独的模型。从某个层次上说，这是可以接受的，因为这是为了实现简单。不过我们已经指出了没有单独的模型会对哪些方面造成影响（例如摘要更新）。
- 有多处存在魔法数（颜色和字符串）。
- 可以将按钮放在一个 Box 中，而不是将它放在正使用的面板组合中。

在寻找各种异味时，我们总是会发现一些同样的问题，部分原因在于异味的结果并不是完全独立的，一个问题可能会导致多种不同的异味。

B. 我设想的修复策略如下：

- 先处理简单的问题：修复名字问题、过长的方法、魔法数，等等。
- 将与 Background 有关的内容搬移至该类中。
- 看看可以消除哪些重复。
- 简化选择。这包括在类之间转移特性，还可能需要引入新的类。
- 估计成本和速度是否应该有它们自己的类。
- 查看哪里还有问题，如果是我，我还会分离出模型。

习题 92　测试

从易（影响较小）到难，如下所列：

- 测试类。例如，可以调用 rotateCost()并检验得到的 cost()。
- 开发可以遍历（walk）组件树的方法，找出特定的组件。例如，会找出 New 按钮，调用其 doClick()方法，然后你可能发现检验方法体中有多个 Card。

- 提供类的实例变量，使测试能够更直接地访问组件。
- 更改应用程序的结构，以获得更好的可测试性。例如，实现一个单独的、易于测试的模型，并拥有一个非常简单的瘦 GUI 层。

习题 93 编写测试

使用最简单的方法，即正常创建对象并调用公共方法，测试类会对各个对象做一定的测试。此测试请见本章第 2 部分中的代码。

习题 94 死代码

变量 budget 和 tabs 没有用到。

习题 96 匿名内部类

B. 将内部类改为单独的类，你会发现这样很有用——可以更容易发现重复。

习题 97 魔法数

结果就是本章第 2 部分中的起始代码，此代码可在线获得。

习题 98 测试按钮

将特性移至别处之前，希望如果采取了不正确的做法，测试会给出相应的警告。到目前为止，我的测试只是调用了公共的可用方法，这不包括模拟按钮点击功能。如果你的测试没有这一功能，请务必增加。可以逐个提供按钮，也可以找出一种更通用的方法。有了按钮之后，就可以调用 doClick()来使按钮完成它的处理了。

增加额外的测试。我令按钮可以实现包级访问，从而使测试能够涉及按钮。但我没有对 Plan 按钮的对话框做任何处理。

习题 101 清理成本

让 Cost 有一个可取值列表，其中每个值显示它的名字和成本，同时指出是否需要估计或分解。

习题 102 按钮创建

抽取一个创建各按钮的方法，然后将各个按钮添加到面板上，这非常简单。

习题 103　将选项处理移至 Background 中

对于 PlanningGame 中的选项，我使用了封装字段，以便只能通过设置方法和获取方法来访问选项。然后便可轻松地使用搬移方法移至 Background 了。在移动监听者时，我将卫哨部分留在按钮处，并假设动作被正确调用。对于 ContainListener，我将 Background 用作一个 ContainerListener，而不是维护另一个单独的类（这样做有好也有坏）。最后，我会全部检查一遍，在可以的地方将方法置为私有。

习题 104　属性更改

这要求 PlanningGame 有一个监听者，为此我们要去除 updateCost()调用。

习题 105　启用按钮

我给 makeButton()增加了一个参数，以便通知按钮对索引卡数目是否敏感。

习题 106　Card 和 Background

这比我想象中的难，在边框更改时，就已经有了一些其他的属性更改通知。我用了一个新的属性名，并令 Background 监听这些通知。

习题 107　清理

我重新构造了 resetSelection()处理，减少了创建新索引卡和分解索引卡动作中的一些重复，并做了一些其他改进工作。

习题 109　单独的模型

F. 在我看来这是一个改进，现在可以单独地修改 GUI 和模型了。

习题 110　一种优化

A. 当比较字符串的开销小于通知的开销时，相应代码要更快一些。

B. 这有助于避免一些设置模型的小循环通知监听者（如谁更新了字段，谁通知监听者模型有所更新，等等），以至于造成无限循环。

附录 B

Java重构工具

一些 Java 编程环境增加了对重构的支持。

工具	公司	类型	URL
Borland Together	Borland	提供Java和重构支持的 UML 工具	www.Borland.com
CodeGuide	Omnicore	IDE	www.omnicore.com
Eclipse		IDE	www.eclipse.org
Idea	IntelliJ	IDE	www.intellij.com
JavaRefactor		jEdit 插件	plugins.jedit.org/plugins/?JavaRefactor
JBuider	Borland	IDE	www.Borland.com/jbuilder
JFactor	Instantiations	JBuilder 和 VisualAge 插件	www. instantiations.com/jfactor
JRefactory		Elixir、JBuilder 和 NetBeans 插件	jrefactory.sourceforge.net
RefactorIt	Aqris	JBuilder、JDeveloper、NetBeans 和 Sun ONE Studio 插件	www. refactorit. com
Transmogrify		JBuilder 和 Forte4Java 插件	transmogrify.sourceforge.net
XRefactory	Xref-Tech	Emacs、jEdit 和 XEmacs 插件	www.xref-tech. com

在 www.refactortng.com/tools.html 中给出了一个工具列表，除了上面的工具，此列表还提供了一些面向其他语言的工具。

附录 C

重构逆处理

重构技术	逆处理
Add Parameter（增加参数）	Remove Parameter（移除参数）
Change Bidirectional Association to Unidirectional（将双向关联改为单向）	Change Unidirectional Association to Bidirectional（将单向关联改为双向）
Change Reference to Value（将引用改为值）	Change Value to Reference（将值改为引用）
Change Unidirectional Association to Bidirectional（将单向关联改为双向）	Change Bidirectional Association to Unidirectional（将双向关联改为单向）
Change Value to Reference（将值改为引用）	Change Reference to Value（将引用改为值）
Collapse Hierarchy（折叠继承体系）	Extract Subclass（抽取子类）
Extract Class（抽取类）	Inline Class（内联类）
Extract Method（抽取方法）	Inline Method（内联方法）
Extract Subclass（抽取子类）	Collapse Hierarchy（折叠继承体系）
Hide Delegate（隐藏委托）	Remove Middle Man（移除中间人）
Inline Class（内联类）	Extract Class（抽取类）
Inline Method（内联方法）	Extract Method（抽取方法）
Inline Temp（内联临时变量）	Introduce Explaining Variable（引入解释变量）
Introduce Explaining Variable（引入解释变量）	Inline Temp（内联临时变量）
Move Field（搬移字段）	Move Field（搬移字段）
Move Method（搬移方法）	Move Method（搬移方法）
Parameterize Method（令方法参数化）	Replace Parameter with Explicit Method（将参数替换为显式方法）
Pull Up Field（上移字段）	Push Down Field（下移字段）

（续表）

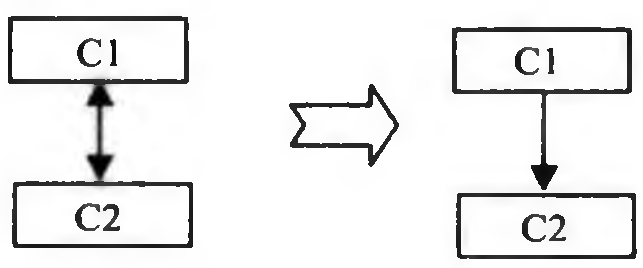
重构技术	逆处理
Pull Up Method（上移方法）	Push Down Method（下移方法）
Push Down Field（下移字段）	Pull Up Field（上移字段）
Push Down Method（下移方法）	Pull Up Method（上移方法）
Remove Middle Man（移除中间人）	Hide Delegate（隐藏委托）
Remove Parameter（移除参数）	Add Parameter（增加参数）
Rename Method（重命名方法）	Rename Method（重命名方法）
Replace Delegation with Inheritance（将委托替换为继承）	Replace Inheritance with Delegation（将继承替换为委托）
Replace Inheritance with Delegation（将继承替换为委托）	Replace Delegation with Inheritance（将委托替换为继承）
Replace Parameter with Explicit Methods（将参数替换为显式方法）	Parameterize Method（令方法参数化）
Substitute Algorithm（替代算法）	Substitute Algorithm（替代算法）

附录 D

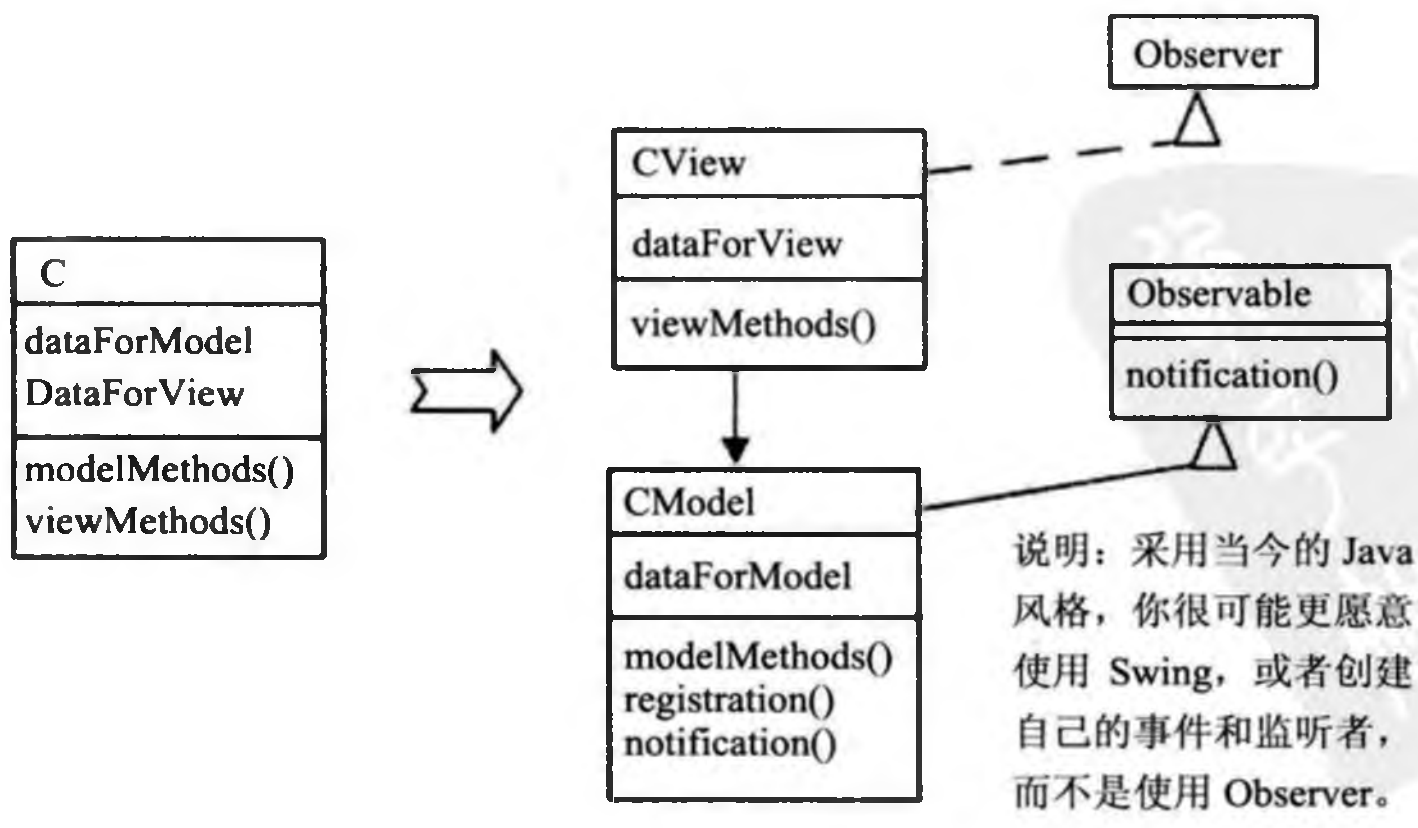
主要重构技术

以下 UML 图是完成某些重要重构之前和之后的表示图。有关完整的转换名录，请参见 Fowler 等人所著的《重构》，从中你会了解到安全应用这些重构技术的步骤。

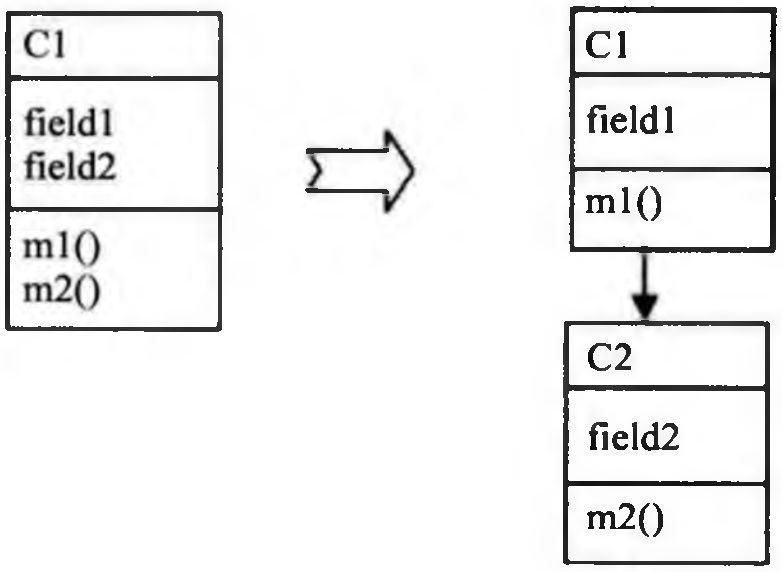
将双向关联改为单向



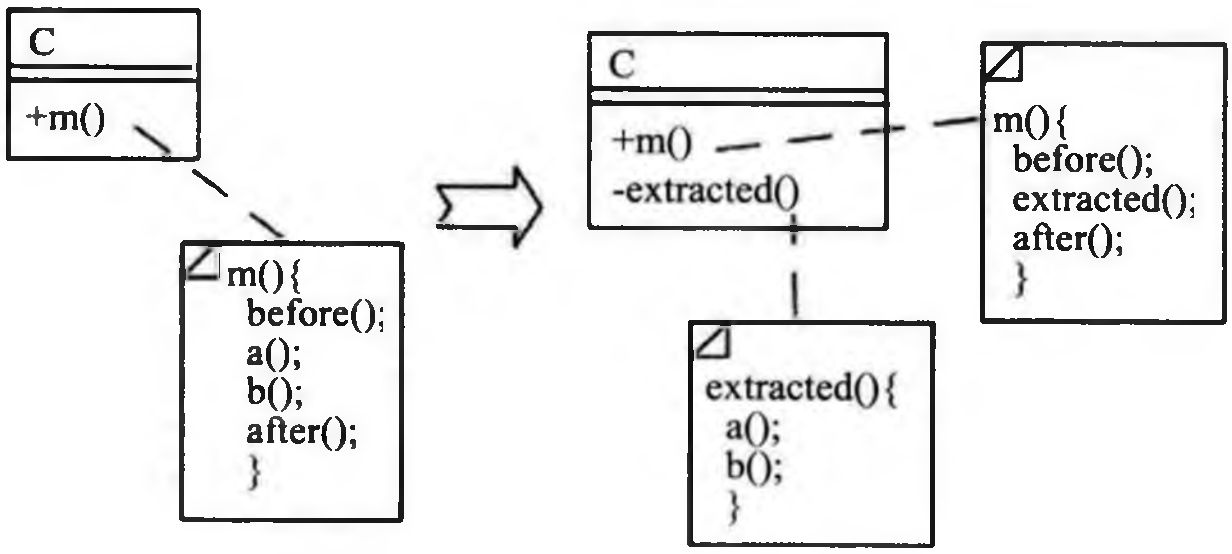
复制受察数据



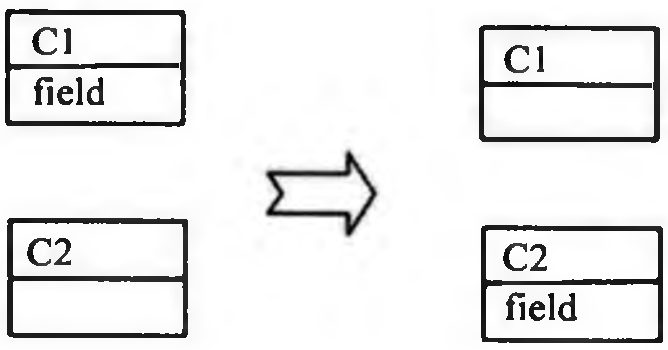
抽取类



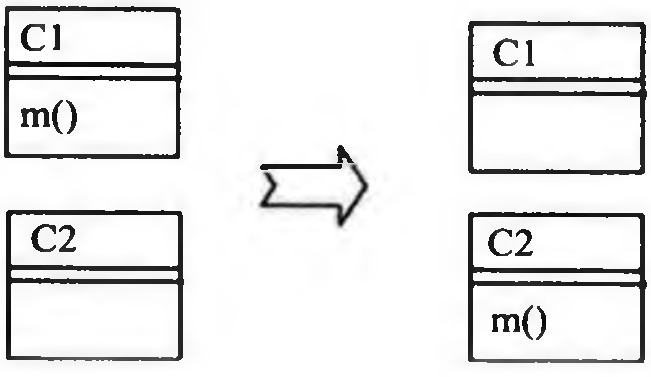
抽取方法



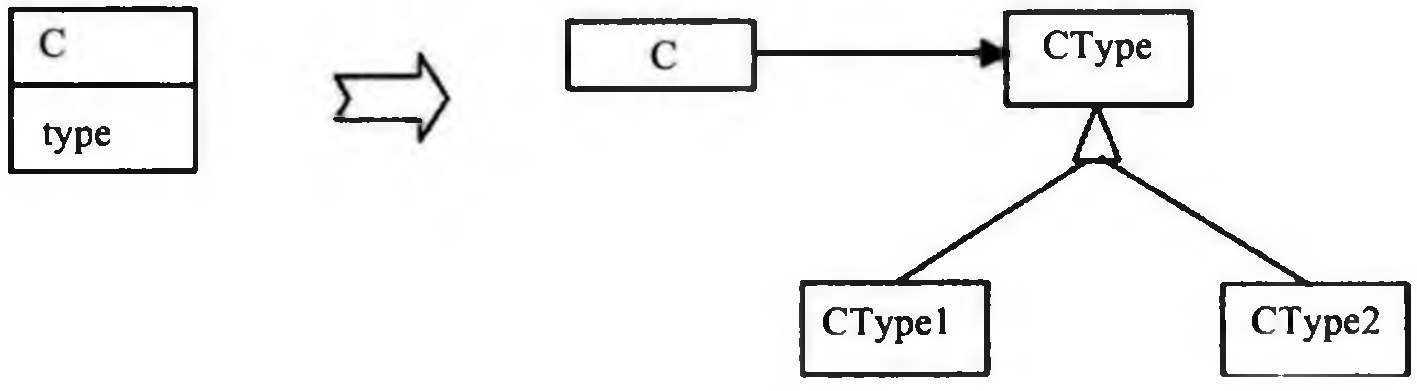
搬移字段



搬移方法



将类型码替换为状态/策略



参 考 文 献

[1] Astels, Dave. Test-Driven Development: A Practical Guide (Upper Saddle River, NJ: Prentice Hall, 2003).

[2] Beck, Kent. Extreme Programming Explained: Embrace Change (Boston: Addison-Wesley, 2000).

[3] ——. Test-Driven Development: By Example (Boston: Addison-Wesley, 2003).

[4] Bentley, Jon. More Programming Pearls (Reading, MA: Addison-Wesley, 1988).

[5] ——. Programming Pearls (Reading, MA: Addison-Wesley, 1986).

[6] ——. Writing Efficient Programs (Upper Saddle River, NJ: Prentice Hall, 1982).

[7] Cunningham, Ward. CRC Cards. Portland Pattern Repository. June 24, 2003. <http://c2.com/cgi/wiki?CrcCards>.

[8] ——. A CRC Description of HotDraw. Portland Pattern Repository. 1994. <http://www.c2.com/doc/crc/draw.html>.

[9] Flanagan, David. Java in a Nutshell, 4/e (Sebastopol, CA: O'Reilly, 2002).

[10] Fowler, Martin, et al. Refactoring: Improving the Design of Existing Code (Boston: Addison-Wesley, 2000).

[11] Fowler, Martin, and Kendall Scott. UML Distilled (Reading, MA: Addison-Wesley, 1997).

[12] Gamma, Erich, et al. Design Patterns (Reading, MA: Addison-Wesley, 1995).

[13] Hunt, Andrew, and David Thomas. The Pragmatic Programmer (Boston: Addison-Wesley, 2000).

[14] Jensen, Bill. Simplicity: The New Competitive Advantage (Cambridge, MA: Perseus Books, 2000).

[15] Kent, William. "A Simple Guide to Five Normal Forms in Relational Theory," Communications of the ACM 26, no. 2 (February 1983), pp. 120–25.

[16] Kernighan, Brian W., and P. J. Plauger. The Elements of Programming Style (New York: The McGraw-Hill Companies, 1988).

[17] Martin, Robert C. Agile Software Development: Principles, Patterns, and Practices (Upper Saddle River, NJ: Prentice Hall, 2002).

[18] Metsker, Steve. Design Patterns Java Workbook (Boston: Addison-Wesley, 2002).

[19] Meyer, Bertrand. Object-Oriented Software Construction (Upper Saddle River, NJ: Prentice Hall, 1988).

[20] Wake, William C. Extreme Programming Explored (Boston: Addison-Wesley, 2002).