

PEARSON

Broadview®
www.broadview.com.cn

Design
Patterns
In Java™ (2nd Edition)

Java 设计模式

(第2版)



John Metsker
William C. Wake 著

张逸 史磊 译



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

更多高清PDF电子书，请加入360云盘文件共享群：电子书联盟，

邀请链接：<http://qun.yunpan.360.cn/12104384> ,
邀请码：7920

QQ群：315971553

Java 设计模式

(第2版)

Design Patterns In JavaTM
(2nd Edition)

Steven John Metsker 著
William C. Wake 著
张逸 史磊 译

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书通过一个完整的 Java 项目对经典著作 *Design Patterns* 一书介绍的 23 种设计模式进行了深入分析与讲解，实践性强，却又不失对模式本质的探讨。本书创造性地将这些模式分为 5 大类别，以充分展现各个模式的重要特征，并结合 UML 类图与对应的 Java 程序，便于读者更好地理解。全书给出了大量的练习，作为对读者的挑战，以启发思考，督促读者通过实践练习的方式来掌握设计模式。同时，作者又给出了这些练习的参考答案，使读者可以印证比较，找出自己的不足，提高设计技能。

本书适合各个层次的 Java 开发人员与设计人员阅读，也可以作为学习 Java 与设计模式的参考读物或教材。

Authorized translation from the English language edition, entitled DESIGN PATTERNS IN JAVA, 2E, 9780321333025 by METSKER, STEVEN JOHN; WAKE, WILLIAM C., published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright© 2006 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 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright©2012

本书简体中文版专有出版权由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号 图字：01-2011-4224

图书在版编目 (CIP) 数据

Java 设计模式：第 2 版 / (美) 梅特尔斯克 (Metsker,S.J.), (美) 维克 (Wake,W.J.) 著；张逸，史磊译。

北京：电子工业出版社，2012.9

书名原文：Design Patterns in Java: 2nd Edition

ISBN 978-7-121-17826-9

I. ①J… II. ①梅… ②维… ③张… ④史… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字 (2012) 第 179321 号

策划编辑：张春雨 符隆美

责任编辑：刘 膺

印 刷：北京丰源印刷厂

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：24.5 字数：549 千字

印 次：2012 年 9 月第 1 次印刷

定 价：75.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。



译者序

如今，介绍和讲解设计模式的书籍可谓汗牛充栋。无论是定义、解读、延伸还是扩展，都是基于面向对象的设计原则，用了放大镜对着 GoF (*Design Patterns* 一书) 提出的 23 种设计模式，如科学解剖一般，剖析每一道脉络，观察每一片纹理，细微至纤毫毕现，真可以说是道尽个中妙处；许多精妙阐述，又如黄钟大吕，发聩振聋，醍醐灌顶。

是否设计模式的精妙之处，业已为这些著作所穷尽？然，又未必尽然！以模式而论，若只局限在这 23 种模式的范围内，几乎每种模式的变化，都可以被悉心推演出来；每种模式的结构，也已被阐述得淋漓尽致。然而，若论及设计，则如大道苍穹，实则是不可穷尽的。基本上，设计的复杂程度已不亚于一个纷繁的世界，而软件，就是我们要构造的这个世界。

因此，再出现一本讲解设计模式的书，就不足为怪了。那么，它值得你去阅读吗？

讨论一本书是否值得阅读，应基于书本身的价值去判断，判断的标准则依据读者的目标而定。从读者而非译者的角度看待本书，个人认为，它确乎是有价值的。这些价值主要体现在三个方面。

GoF 对于 23 种设计模式的分类已经深入人心，即众所周知的创建型模式、结构型模式与行为型模式。这一分类浅显易懂，明白无误地表达了模式的意图与适用场景。但是，这一分类仍有不足之处。例如创建型模式除了关注对象的创建之外，还需处理好对象之间的结构；又如桥接模式对于抽象与实现的解耦，在一定程度上又体现了对行为的抽象；再比如行为型模式中的迭代器模式，其实还涵盖了创建迭代器的职责。本书对于设计模式的分类不落窠臼，根据作者

对于设计模式的思考，别出心裁地给出了自己的一种分类，即分为接口型模式、职责型模式、构造型模式、操作型模式与扩展型模式。如果仔细阅读和思考这些模式，你会发现这 5 个分类很好地抓住了相关模式的设计本质。譬如，扩展型模式关注的是代码功能的扩展，因而很自然地就可以把装饰器模式与访问者模式归入这一类。

彰显本书价值的第二个方面在于贯穿本书始终的习题练习，作者将其称之为“挑战”。确实如此，这些挑战仿佛是作者故意为读者设定的“陷阱”、“障碍”，是登堂入室所必须跨过的门槛。最关键的一点是，通过这些“挑战”，就从单方面的灌输知识，变成了一定程度的双向互动。作者就像课堂上的老师，提出问题引人思考；读者就是学生，面对老师“咄咄逼人”的提问，必须打起十二分的精神，分析问题，寻找问题的答案。最后，循循善诱的老师给出了自己的解决方案。学生可以与之对比，以便发现自己在设计上还存在的问题。因此，本书不适合那些惫懒的读者，不适合那些喜欢被动接收知识输入，不善于思考，不善于总结的程序员。

真正让本书获得赞誉的还是本书给出的案例，不过，也很有可能因此收获负面的批评。本书的案例是一个虚拟的真实项目，Oozinoz 公司纯属子虚乌有，完全是由作者杜撰出来的一家虚拟公司。但这个案例又如此真实，既牵涉复杂的领域逻辑，又面对客户提出的种种需求变化，与我们工作中需要开发的项目何其相似！可能面临的批评是，为了学习设计模式，可能读者还需要成为一名焰火专家。然而，我谨以最谦卑的态度恳求诸位，在满怀怨气、恶毒诅咒作者（也可能包括躺着中枪的译者）之前，先想想我们平时开发的软件，是否存在相似复杂度的领域需求呢？让我们再仔细想想，倘若作者给出一个纯粹编造出来的玩具项目，贴近生活，浅显易懂，学习起来势如破竹，一路通关，是否真的意味着你已经明白如何在真实项目中运用设计模式？窃以为，学习尤其是技术学习，并不都是舒舒服服寓教于乐，躺着、玩着以及笑着也能学好设计模式。你以为的懂，以为的悟，其实还是一种虚妄。你抓住的是水中央的月影，一旦遇到真实案例，就好似石头打破水面的宁静，一切都会破碎。

本书的原版事实上获得了业界的广泛赞誉，同时也是 John Vlissides 主编的“软件模式”丛书之一。John Vlissides 就是著名的 GoF 其中的一位，可惜他已在多年前离开人世。本书作者是 John Vlissides 的生前好友，本书内容曾经得到过他的建议。从本书的内容来看，部分 Java 案例显得有些过时，不过，就设计而言，拥有悠久的历史，有时候意味着它可能成为经典。不错，与经典的 GoF 相比，本书无疑要失色许多。GoF 的光芒在于它的开创性。只要是讲解设计模式，没有哪一本书的光芒可以盖过 GoF 的著作。它就像是一颗恒星，其他有关设计模式的书籍，是围绕着它公转的一颗颗行星，都是借着恒星的光芒反射出属于自己的光亮。然而，从光芒的热

度与亮度来讲，也许行星才是当前的你最适合的。

阅读本书的读者，除了需要具备一些面向对象与设计模式的基础知识外，还需要有足够的耐心，并保存一份渴望与热情。耐心可以帮助你坚持细读与精读，持之以恒地深入理解本书的案例分析，努力面对作者给出的挑战。而这种耐心则需要提高技术能力的渴望，探求技术奥秘的热情来时刻保鲜。

本书的翻译由我的同事史磊与我共同完成，并最后由我完成审校工作。在翻译本书时，我还参考了由龚波、赵彩琳、陈蓓翻译的前一个版本，在此向他（她）们表示衷心的感谢。因为工作繁忙的缘故，本书的翻译工作一直断断续续持续了近一年的时间，如今交稿，既有卸下重任的轻松畅快，却又因为自己的惫懒使得翻译工作进展缓慢而深感愧疚。这里需要感谢本书编辑符隆美女士给予我的耐心与支持。

在写作这篇译者序时，同事史磊已经远赴 ThoughtWorks 南非工作，而我则从北京回到了 ThoughtWorks 成都。非常怀念我们在北京一起工作的日子。我们曾经在同一个项目结对编程，本书的翻译也可以说是结对完成，算是一次愉快的翻译体验。鉴于本人能力水平有限，翻译或有疏漏或错误，还请读者不吝赐教，并通过我的博客 www.agiledon.com 与我联系。

张逸

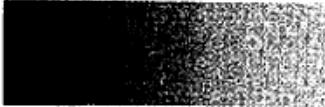
ThoughtWorks 高级咨询师

To Alison, Emma-Kate, and Sarah-Jane

—Steve

To May Lyn, Tyler, and Fiona

—Bill



序

设计模式是面向对象设计中常见问题的类级与方法级的解决方案。本书面向那些希望更上一层楼的初、中级 Java 程序员，以及那些不具备设计模式知识的高级 Java 程序员。

本书采用工具书的方式进行编写。每一章针对一个特定的模式。为了更好地阐释模式，每章还包含了大量的练习，要求读者回答问题，或者编写代码解决问题。

我们强烈地建议读者在阅读本书时，不要对自我挑战中的问题视而不见。通过完成这些挑战，你将获益匪浅，即使你一周只能阅读一至两章的内容。

修订版

本书是对 *Design Patterns Java WorkbookTM* 和 *Design Patterns in C#* 两本书的集成与修订。本书综合了前者面向 Java 语言的特点，又保留了后者独立陈述的写作方式。如果你业已阅读并理解这两本书，可以不必再阅读本书。

编码规范

本书代码可以在线获得。若欲了解获取代码的方法，请参阅本书附录 C。

本书采用了与 Sun 编码规范一致的通用风格。如有可能，会省略一些能够省略的括号，以保证本书排版上的一致性。为了适应版面，有的变量名比我们真正使用的要短。为了避免源代码管理的麻烦，我们直接在文件名后面添加数字后缀，以表示多个版本的文件（例如 `ShowBallistics2`）。

致谢

写书是一项挑战。在写书期间，许多审稿人为本书提出了非常有价值的建议：Daryl Richter、Adewale Oshineye、Steven M. Luplow、Tom Kubit、Rex Jaeschke、Jim Fox 和 David E. DeLano。他们每个人的建议都使得本书的质量得以改善。早期版本的读者以及审稿人也做出了同样的贡献。

还要感谢 Addison-Wesley 的编辑团队，尤其是 Chris Guzikowski、Jessica D'Amico 和 Tyrrell Albaugh。其他编辑包括 Mary O'Brien 和 John Wait 也为我们提供了诸多帮助。

感谢已经辞世的 John Vlissides 对本书以及其他书籍的鼓励与建议。John 是软件模式丛书的编辑，经典著作 *Design Patterns* 的合著者，我的朋友，灵感迭出的天才。

除了主要参考了 *Design Patterns* 一书外，我们还从其他诸多著作中获益，详见本书参考书目。其中，*Unified Modeling Language User Guide*（由 Booch、Rumbaugh 和 Jacobson 在 1999 年编写）提供了对 UML 清晰的阐释，*Java™ in a Nutshell*（由 Flanagan 在 2005 年编写）对 Java 语言简明扼要的介绍，让我受益颇丰。*The Chemistry of Fireworks*（由 Russell 在 2000 年编写）则是我获取焰火知识的主要来源。

最后，我们要感谢出版社所有员工的辛勤劳动与努力奉献，正是你们的工作使得本书得以付梓出版。

Steve Metsker (Steve.Metsker@acm.org)

Bill Wake (William.Wake@acm.org)



目 录

| | |
|-----------------------|----------|
| 序 | xv |
| 第 1 章 绪论 | 1 |
| 为何需要模式 | 1 |
| 为何需要设计模式 | 2 |
| 为何选择 Java | 3 |
| UML | 3 |
| 挑战 | 4 |
| 本书的组织 | 4 |
| 欢迎来到 Oozinoz 公司 | 6 |
| 小结 | 6 |

第 1 部分 接口型模式

| | |
|----------------------------|----------|
| 第 2 章 接口型模式介绍 | 8 |
| 接口与抽象类 | 8 |
| 接口与职责 | 10 |
| 小结 | 11 |
| 超越普通接口 | 12 |

| | |
|--------------------------------|----|
| 第 3 章 适配器 (Adapter) 模式 | 13 |
| 接口适配 | 13 |
| 类与对象适配器 | 17 |
| JTable 对数据的适配 | 20 |
| 识别适配器 | 24 |
| 小结 | 25 |
| 第 4 章 外观 (Facade) 模式 | 27 |
| 外观类、工具类和示例类 | 27 |
| 重构到外观模式 | 29 |
| 小结 | 38 |
| 第 5 章 合成 (Composite) 模式 | 39 |
| 常规组合 | 39 |
| 合成模式中的递归行为 | 40 |
| 组合、树与环 | 42 |
| 含有环的合成模式 | 47 |
| 环的影响 | 50 |
| 小结 | 51 |
| 第 6 章 桥接 (Bridge) 模式 | 52 |
| 常规抽象：桥接模式的一种方法 | 52 |
| 从抽象到桥接模式 | 54 |
| 使用桥接模式的驱动器 | 57 |
| 数据库驱动 | 57 |
| 小结 | 59 |

第 2 部分 职责型模式

| | |
|----------------------|----|
| 第 7 章 职责型模式介绍 | 62 |
| 常规的职责型模式 | 62 |
| 根据可见性控制职责 | 64 |

| | |
|--|-----|
| 小结 | 65 |
| 超越普通职责 | 65 |
| 第 8 章 单例 (Singleton) 模式 | 67 |
| 单例模式机制 | 67 |
| 单例和线程 | 68 |
| 识别单例 | 70 |
| 小结 | 71 |
| 第 9 章 观察者 (Observer) 模式 | 72 |
| 经典范例：GUI 中的观察者模式 | 72 |
| 模型/视图/控制器 | 76 |
| 维护 Observable 对象 | 82 |
| 小结 | 84 |
| 第 10 章 调停者 (Mediator) 模式 | 85 |
| 经典范例：GUI 调停者 (Mediator) | 85 |
| 关系一致性中的调停者模式 | 89 |
| 小结 | 96 |
| 第 11 章 代理 (Proxy) 模式 | 97 |
| 经典范例：图像代理 | 97 |
| 重新思考图片代理 | 102 |
| 远程代理 | 104 |
| 动态代理 | 109 |
| 小结 | 114 |
| 第 12 章 职责链 (Chain of Responsibility) 模式 | 115 |
| 现实中的职责链模式 | 115 |
| 重构为职责链模式 | 117 |
| 固定职责链 | 119 |
| 没有组合结构的职责链模式 | 121 |
| 小结 | 121 |

| | |
|---------------------------------|-----|
| 第 13 章 享元 (Flyweight) 模式 | 122 |
| 不变性 | 122 |
| 抽取享元中不可变的部分 | 123 |
| 共享享元 | 125 |
| 小结 | 128 |

第 3 部分 构造型模式

| | |
|--|-----|
| 第 14 章 构造型模式介绍 | 130 |
| 构造函数的挑战 | 130 |
| 小结 | 132 |
| 超出常规的构造函数 | 132 |
| 第 15 章 构建者 (Builder) 模式 | 134 |
| 常规的构建者 | 134 |
| 在约束条件下构建对象 | 137 |
| 可容错的构建者 | 139 |
| 小结 | 140 |
| 第 16 章 工厂方法 (Factory Method) 模式 | 141 |
| 经典范例：迭代器 | 141 |
| 识别工厂方法 | 142 |
| 控制要实例化的类 | 143 |
| 并行层次结构中的工厂方法模式 | 145 |
| 小结 | 147 |
| 第 17 章 抽象工厂 (Abstract Factory) 模式 | 148 |
| 经典范例：图形用户界面工具箱 | 148 |
| 抽象工厂和工厂方法 | 153 |
| 包和抽象工厂 | 157 |
| 小结 | 157 |

| | |
|---------------------------------|-----|
| 第 18 章 原型 (Prototype) 模式 | 158 |
| 作为工厂的原型 | 158 |
| 利用克隆进行原型化 | 159 |
| 小结 | 162 |
| 第 19 章 备忘录 (Memento) 模式 | 163 |
| 经典范例：使用备忘录模式执行撤销操作 | 163 |
| 备忘录的持久性 | 170 |
| 跨会话的持久性备忘录 | 170 |
| 小结 | 174 |

第 4 部分 操作型模式

| | |
|---|-----|
| 第 20 章 操作型模式介绍 | 176 |
| 操作和方法 | 176 |
| 签名 | 177 |
| 异常 | 178 |
| 算法和多态 | 179 |
| 小结 | 180 |
| 超越常规的操作 | 181 |
| 第 21 章 模板方法 (Template Method) 模式 | 182 |
| 经典范例：排序 | 182 |
| 完成一个算法 | 186 |
| 模板方法钩子 | 188 |
| 重构为模板方法模式 | 189 |
| 小结 | 191 |
| 第 22 章 状态 (State) 模式 | 193 |
| 对状态进行建模 | 193 |
| 重构为状态模式 | 197 |

| | |
|---|------------|
| 使状态成为常量..... | 201 |
| 小结..... | 203 |
| 第 23 章 策略 (Strategy) 模式..... | 204 |
| 策略建模..... | 204 |
| 重构到策略模式..... | 207 |
| 比较策略模式与状态模式..... | 211 |
| 比较策略模式和模板方法模式..... | 211 |
| 小结..... | 212 |
| 第 24 章 命令 (Command) 模式..... | 213 |
| 经典范例：菜单命令..... | 213 |
| 使用命令模式来提供服务..... | 216 |
| 命令钩子..... | 217 |
| 命令模式与其他模式的关系..... | 219 |
| 小结..... | 220 |
| 第 25 章 解释器 (Interpreter) 模式..... | 221 |
| 一个解释器示例..... | 221 |
| 解释器、语言和解析器..... | 233 |
| 小结..... | 234 |

第 5 部分 扩展型模式

| | |
|----------------------------|------------|
| 第 26 章 扩展型模式介绍..... | 236 |
| 面向对象设计的原则..... | 236 |
| Liskov 替换原则..... | 237 |
| 迪米特法则..... | 238 |
| 消除代码的坏味道..... | 239 |
| 超越常规的扩展..... | 240 |
| 小结..... | 241 |

| | |
|----------------------------------|-----|
| 第 27 章 装饰器 (Decorator) 模式 | 242 |
| 经典范例：流和输出器 | 242 |
| 函数包装器 | 250 |
| 装饰器模式和其他设计模式的关系 | 257 |
| 小结 | 258 |
| 第 28 章 迭代器 (Iterator) 模式 | 259 |
| 普通的迭代 | 259 |
| 线程安全的迭代 | 261 |
| 基于合成结构的迭代 | 267 |
| 小结 | 277 |
| 第 29 章 访问者 (Visitor) 模式 | 278 |
| 访问者模式机制 | 278 |
| 常规的访问者模式 | 280 |
| Visitor 环 | 286 |
| 访问者模式的危机 | 290 |
| 小结 | 292 |
| 附录 A 指南 | 293 |
| 附录 B 答案 | 297 |
| 附录 C Oozinoz 源代码 | 366 |
| 附录 D UML 概览 | 369 |
| 参考文献 | 375 |

第1章

绪论

本书涵盖了 Erich Gamma、Richard Helm、Ralph Johnson 与 John Vlissides 在 1995 年编写的经典著作 *Design Patterns* 相同的技术，并提供了 Java 范例。书中还包括了许多“挑战”（练习）——这些练习能够帮助我们提高在软件开发中运用设计模式的能力。

本书面向那些了解 Java 语言的开发人员，以及希望提高设计能力的设计人员。

为何需要模式

模式是做事的方法，是实现目标，研磨技术的方法。这种对高效技术不懈追求的思想，广泛见于诸多领域，例如制作精美的佳肴、生产绚烂的烟花、开发出色的软件及其他技艺。对于任何一种迈向成熟的全新技艺，身处这个行当的人们都需要寻找通用而有效的方法，以达到他们的目标，并解决不同场景的问题。实践技艺的团体通常会发明一些行话，用来讨论他们的技艺。一些被公认的行话反映了某种模式，或者建立了实现准确目标的技术。随着技艺的增长，以及这些行话的发展，一些行话的创作者开始扮演重要的角色。他们对技艺模式进行编档，为这些行话设定标准，然后发布这些有效的技艺。

Christopher Alexander 是最早对技艺的最佳实践进行模式编档的一位作者。他的著作与建筑有关，而非软件。在 *A Pattern Language: Towns, Buildings Construction* (由 Alexander、Ishikouwa

和 Silverstein 在 1977 年编写)一书中, Alexander 提供了成功构建建筑与城市的多种模式。他的著作对软件社区影响深远,部分原因在于他提出了设计意图的思想。

我们或许会认为建筑模式的意图就是“设计建筑”之类的玩意儿。其实不然。Alexander 阐明了建筑模式的意图,就是为那些渴望拥有建筑与城市的人们服务,诗意地栖居并带给他们感动。Alexander 的著作揭示了模式作为一种卓越的方法,能够捕捉与传播技艺的智慧之光。他认为,正确感知并记录一门技艺的意图至关重要,这是一种挑战,它难以捉摸,却又至关重要,充满着令人沉思的哲学意味。

模式方法引起了软件社区的共鸣,诸多记录软件开发模式的著作如雨后春笋。这些书中记录了软件进程、软件分析、高层架构与类级设计的最佳实践。每年都会有新的模式书籍不断涌现。如果你要挑选一本模式书籍来阅读,必须精挑细选,认真阅读图书,审慎地做出决策,才能选到最能帮助你的那一本。

为何需要设计模式

设计模式是一种模式,在面向对象的语言中,它运用类与它们的方法来达到目标。开发人员通常会在学习编程语言,并在编写一段时间的代码之后才会考虑设计。或许我们已经注意到,某些人的代码简洁而实用,那么他们究竟是怎样才能让代码拥有如此的简单之美?设计模式提升了代码的水准,通常会使用更少的类完成目标。模式是思想的体现,而非具体的实现。

有些开发人员已经发现如何利用面向对象的语言进行高效编程的奥秘。倘若你希望成为 Java 编程高手,就应该学习设计模式,尤其是本书提到的与 *Design Patterns* 一书相同的模式。

Design Patterns 一书描述了 23 种设计模式。此后,其他许多书籍也提出了自己的设计模式,已知的设计模式至少有 100 种。Gamma、Helm、Johnson 与 Vlissides 在 *Design Patterns* 一书中提出的 23 种模式或许不是已知模式中最有效的设计模式,但这些模式至少能够名列前沿。*Design Patterns* 一书的作者们精心挑选并记录的这些模式,都值得仔细研读,它们可以作为从其他来源学习模式的基础与起点。

GoF

或许，你已经注意到“设计模式”这一主题与 *Design Patterns* 一书书名之间的混淆不清。因为这个主题与书名如此相似，许多演讲者与作者为了区分两者的不同之处，而以“四巨头（Gang of Four）”或者“GoF”，也就是该书的作者人数来指代该书。在印刷品中，二者的概念可以通过大小写以及正斜体加以区分，因此，本书并未使用 GoF 这个术语。

为何选择 Java

本书提供了 Java 实例。Java 是由 Sun^{译注1}公司开发的一门面向对象的语言，它提供的库与相关工具组成了整套产品，用于开发与管理具有多层架构的面向对象系统。

选择 Java 的一个重要原因在于它是一门集大成的语言，博采众多早期语言之长。这种包容性使得 Java 语言变得越来越流行，也确保了未来的语言将基于 Java 进行演变，而不会从根本上背离它。因此，不管未来会有什么语言取代 Java，你在 Java 上投入的心血都不会付之东流。

Design Patterns 一书中的模式可以运用到 Java 语言上，因为它与 Smalltalk、C++ 及 C#一样，遵循了类/实例的范式。相比于 Prolog 或 Self 等语言，Java 更接近 Smalltalk 和 C++。尽管其他编程范式仍然颇具竞争力，但类/实例范式在应用计算方面却更加实用。本书使用 Java 语言，正是考虑到它的流行以及未来发展的广阔前景。

UML

本书给出的挑战（习题）要求提供 Java 代码的解决方案。不过，多数挑战还是要求读者绘制类、包与其他相关元素的关系图。可以根据自己的习惯选择你喜欢的图示方式，但本书则使用了统一建模语言（UML）。即使你熟悉 UML，仍有必要选择一本参考书。有两本好书可供选择：*The Unified Modeling Language User Guide*（由 Booch、Rumbaugh 和 Jacobson 在 1999 年

译注1：Sun 公司如今已成过眼云烟，被 Oracle 收购。

编写) 和 *UML Distilled* (由 Fowler 和 Scott 在 2003 年编写)。“附录 D: UML 概览”提供了阅读本书所需要了解的 UML 必备知识。

挑战

所谓“实践出真知”，即使你博览群书，如果不去实践，你所了解的内容依旧似是而非。这是因为，如果不将书中的知识付诸于实践，就无法体会到知识的细微之处，更不会去考虑其他可能的替代方案。只有理论联系实践，才能在面临挑战时，能够信心十足地运用设计模式。

依据经验进行学习的问题在于它可能损害你了解到的知识。如果对你拥有的技能没有足够的信心，就无法将模式运用到实际开发中；反过来，又需要你通过充分地运用模式来收获信心。这可真是左右为难！如何解决？应对之道就是通过案例去实践，即使犯下错误也是一种收获，而不会有任何损失。

本书的每一章开篇都是一段简短的介绍，随后给出一系列挑战以供读者解决。在得到自己的解决方案后，可以与本书附录 B 给出的答案相比较。本书给出的解决方案或许与你的解答不尽相同，但却可帮助你从另外一个视角来思考问题。

或许本书的某些挑战对你而言如天堑一般难以跨越，但如果你能够通过查阅其他书籍，或与同事合作，编写实现代码来检查方案的得失，那就太棒了！这绝非虚掷光阴，因为你付出的努力可以帮助你学习如何运用设计模式。

书后提供的解决方案会让你滋生不劳而获的想法。倘若在你阅读完挑战的内容后，迫不及待地去查阅书末的答案，你将一无所获。如果不首先创建自己的解决方案，那么对你而言，书中提供的解决方案就没有任何价值。

本书的组织

目前，存在许多对模式进行组织与分类的方法。我们可以根据结构的相似性进行组织，也可以遵循 *Design Patterns* 一书中的顺序。但是，任何模式的核心要素还在于它的意图，这才是

运用模式的潜在价值。本书根据 *Design Patterns* 一书中模式的意图对 23 种设计模式进行了分类。

既然决定通过意图对模式进行分类，那么应该如何判别这些意图呢？我们认为，设计模式的意图在于用更为简便的方式表达需求，而这些却是 Java 提供的常规机制所无法满足的。例如，Java 为定义一个接口提供了丰富的支持，以便类能够实现接口。然而，倘若我们拥有一个实现了“错误”接口的类，却又需要满足客户的需求，就需要运用适配器模式。适配器模式的设计意图是内置于 Java 语言中的接口所无法满足的。

本书按照意图将设计模式分为以下 5 类：

1. 接口型模式
2. 职责型模式
3. 构造型模式
4. 操作型模式
5. 扩展型模式

这 5 类设计模式分别对应书中的 5 个部分。每一部分的第 1 章分别对 Java 内建特征及其存在的问题进行了讨论。例如，第 1 部分一开始介绍了普通的 Java 接口。这一章讨论了 Java 接口的结构，尤其对接口和抽象类进行了对比。第 1 部分的其他章节则分别介绍了接口型模式，它们的主要意图均与接口的定义有关，提供了一套便于客户对服务提供者进行调用的方法。这些模式解决了单靠 Java 接口无法解决的问题。

根据意图对模式进行分类并不意味着每种模式仅仅支持一种设计意图。倘若模式支持的意图超过一种，而前面已有完整的章节介绍了它，则随后的介绍就一笔带过。表 1.1 给出了本书对设计模式的分类方法。

表 1.1 根据意图对模式的分类

| 意 图 | 模 式 |
|-------|----------------------------------|
| 接口型模式 | 适配器模式、外观模式、合成模式、桥接模式 |
| 职责型模式 | 单例模式、观察者模式、调停者模式、代理模式、职责链模式、享元模式 |
| 构造型模式 | 构建者模式、工厂方法模式、抽象工厂模式、原型模式、备忘录模式 |
| 操作型模式 | 模板方法模式、状态模式、策略模式、命令模式、解释器模式 |
| 扩展型模式 | 装饰器模式、迭代器模式、访问者模式 |

我们希望读者能够对表 1.1 提出疑问。你是否同意单例模式与职责有关，而非构建？合成模式属于接口型模式吗？对设计模式的分类多少带有主观色彩。但是，我们希望读者能够认识到，分析设计模式背后隐藏的意图，思考如何在实际开发中运用这些模式，都将对你的设计能力大有裨益。

欢迎来到 Oozinoz 公司

本书给出的挑战习题均来自于 Oozinoz 公司的焰火生产系统。我们虚构了这家制造和销售焰火的公司，它还将参与焰火表演（*oozinoz* 名字的灵感缘于 Oozinoz 展览的读音）。读者可以从 www.oozinoz.com 网站获得源代码。若要获取更多构建与测试源代码的信息，请参见本书的附录 C。

小结

模式是集体智慧的结晶，它提供了标准的术语，为富有经验的参与者提供了统一命名的概念。经典书籍 *Design Patterns* 中的模式是类级别的模式，理所当然值得我们认真学习。本书阐释了与 *Design Patterns* 一书中相同的模式，但使用了 Java 及其库作为示例与挑战。通过本书挑战进行的实战练习，可以学到如何识别与运用设计模式这一软件社区积累的智慧结晶。



第 1 部分

接口型模式

第2章

接口型模式介绍

抽象地讲，类的接口是类允许其他类对象访问的方法与字段集。接口通常代表一种承诺，即方法需要实现接口方法名表示的操作，遵循代码注释、测试和其他文档说明。类的实现就是位于方法体中的代码。

Java 将接口概念提升为独立的结构，体现了接口（对象必须遵循的承诺）与实现（对象如何履行承诺）的分离。Java 接口允许多个类提供相同的功能，也允许一个类同时实现多个接口。

许多设计模式都使用了 Java 内建的这一特性。例如，运用适配器（Adapter）模式，通过使用一个接口类型来适配类的接口，从而满足客户的需要。若要用好 Java 基本的内建特性，就要从接口开始，确保自己掌握了 Java 特性的工作原理。

接口与抽象类

最初的 *Design Patterns* 一书总是提到抽象类的使用，而对于接口的使用只字不提。这是因为编写该书实例的 C++ 与 Smalltalk 语言根本就没有接口结构。由于 Java 接口与抽象类非常相似，这一点并不影响 Java 开发人员对该书的理解。

挑战 2.1

写出在 Java 中抽象类和接口的三点区别。

答案参见第 297 页

如果不能使用接口，完全可以像 C++ 那样使用抽象类。然而，作为一种独立的结构，接口在 n 层应用程序开发过程中的地位举足轻重。

让我们考虑一个火箭仿真类所必须实现的接口定义。工程师设计了许多不同的火箭，包括固体燃料火箭和液体燃料火箭，这两种火箭有着完全不同的弹道学原理。不管火箭如何组成，对火箭的仿真都必须提供预期推力与质量等数据。下面是 Oozinoz 定义的火箭仿真接口：

```
package com.oozinoz.simulation;

public interface RocketSim {
    abstract double getMass();
    public double getThrust();
    void setSimTime(double t);
}
```

挑战 2.2

下面的描述哪些是正确的？

- A. 尽管只有 `getMass()` 方法被显式声明为抽象，但 `RocketSim` 接口的三个方法都是抽象方法。
- B. 尽管只有 `getThrust()` 方法被显式声明为公开，但 `RocketSim` 接口的三个方法都是公开方法。
- C. 接口虽然被声明为“公有接口”，但即使省略 `public` 关键字，接口仍然是公有的。
- D. 可以创建另外一个接口，例如 `RocketSimSolid`，去扩展 `RocketSim` 接口。
- E. 每个接口必须至少包含一个方法。

- F. 接口可以声明实例字段，实现该接口的类也必须声明该字段。
- G. 虽然不能实例化接口，但接口仍然可以定义构造函数，并要求实现类必须提供具有相同签名的构造函数。

答案参见第 297 页

接口与职责

Java 接口的优势在于它限制了对象之间的协作，这种约束其实提供了更大的自由。即使实现接口的类的实现发生了巨大变化，接口的客户端仍然可以不受影响。

开发人员在创建 `Rocketsim` 接口的实现类时，有责任编写 `getMass()` 和 `getThrust()` 方法，返回火箭的性能指标。换言之，开发人员必须遵循这些方法制订的契约。

有时候，接口指定的方法并不要求必须为调用者提供服务。在某些情况下，实现类可以忽略这些调用，为实现的方法提供一个空的方法体。

挑战 2.3

请列举一个接口，它包含的方法并不要求实现该接口的类必须返回值，或者代表调用者执行若干操作。

答案参见第 298 页

如果创建的接口指定了一系列用于通知的方法，则可以考虑提供桩（stub），即提供空实现的接口实现类。开发者通过实现桩的子类，重写那些对应用程序具有重要意义的接口方法。`java.awt.event` 中的 `windowAdapter` 就是这样一个例子，如图 2.1 所示。（若要进一步了解 UML，请参见附录 D）。`windowAdapter` 类实现了 `windowListener` 接口的所有方法，但这些实现都是空的，方法不含任何语句。

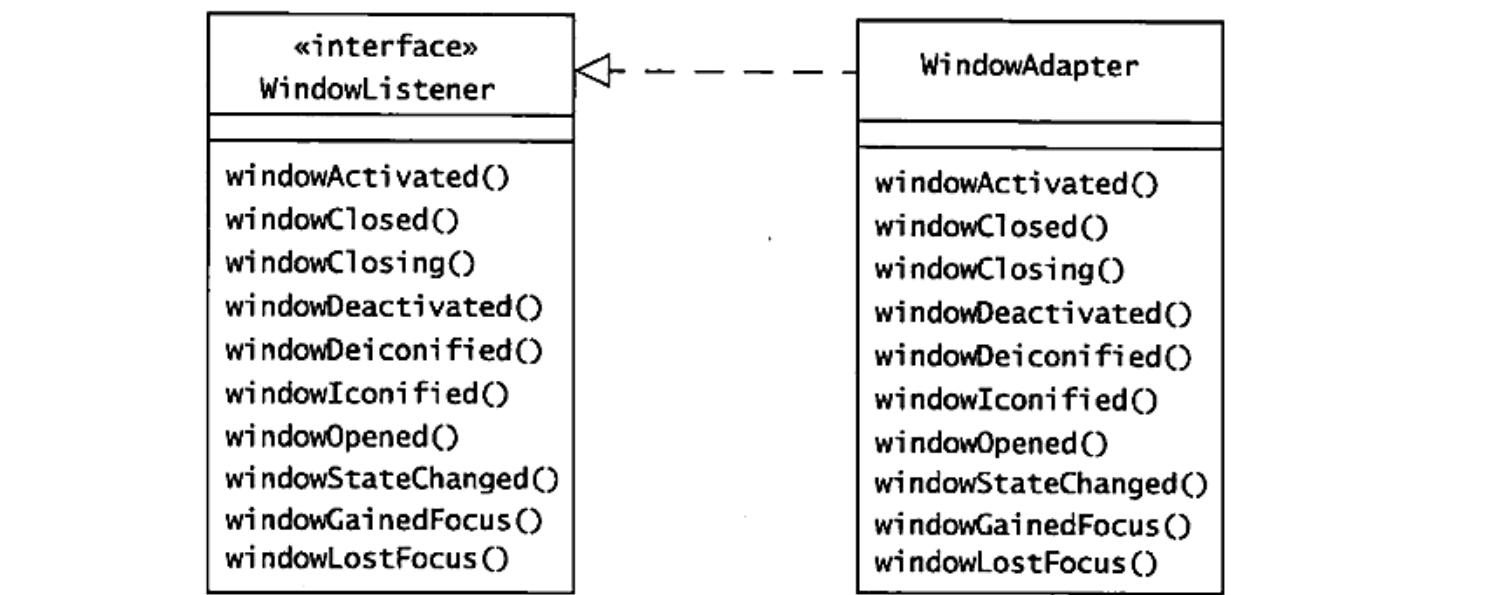


图 2.1 `WindowAdapter` 类注册监听器，可以方便地监听 window 事件，从而使我们忽略自己并不关心的事件

除了声明方法外，接口也可以声明常量。在下面的例子中，`ClassificationConstants` 声明了两个常量^{*注1}，实现该接口的类可以访问它们。

```

public interface ClassificationConstants {
    static final int CONSUMER = 1;
    static final int DISPLAY = 2;
}
  
```

接口与抽象类还有一点关键区别：虽然类只能声明扩展一个类，却可以声明实现多个接口。

小结

接口的威力在于它描述了在类协作中它所期望与不期望的行为。接口与抽象类很相似，定义行为却并不提供实现。

熟练掌握接口的概念与细节，有助于我们更好地运用 Java 接口。这种强有力的数据结构是许多健壮设计与设计模式的核心。

译注1：在 JDK 1.5 引入 enum 类型后，通常不建议在接口中声明常量，而是尽量将具有分组类别的常量定义为 enum 类型。

超越普通接口

如果能恰如其分地运用 Java 接口，就可以简化并完善我们的设计。有时候，接口的设计会超出我们对普通接口的定义与使用。表 2.1 展示了不同场景下应使用的模式。

表 2.1 不同场景下应使用的模式

| 如果你期望 | 可运用模式 |
|---------------------------|-------|
| • 适配类的接口以匹配客户端期待的接口 | 适配器模式 |
| • 为一组类提供一个简单接口 | 外观模式 |
| • 为单个对象与复合对象提供统一的接口 | 合成模式 |
| • 解除抽象与实现之间的耦合，使得二者能够独立演化 | 桥接模式 |

每个设计模式都旨在解决不同场景的问题。面向接口的模式适用于需要对一个类或一组类的方法进行定义或重定义的场景。例如，某个类实现了我们所需要的服务，但它的方法名称却与客户端的期望不符，这就需要运用适配器模式。

第3章

适配器（Adapter）模式

客户端（Client）就是需要调用我们代码的对象。通常，在代码已经存在的情况下编写客户端代码，开发人员可以采取模拟客户端的方式调用我们提供的接口对象。然而，客户端代码也可能与你的代码单独进行开发。例如，设计的火箭仿真程序会使用你所提供的火箭信息，但是对于火箭应该拥有怎样的行为，仿真器也会拥有自己的定义。在这样的情况下，会发现现有的类虽然提供了客户端需要的服务，却被定义为不同的方法名。这时，我们就需要运用适配器（Adapter）模式。

适配器模式的意图在于，使用不同接口的类所提供的服务为客户端提供它所期望的接口。

接口适配

当我们需要适配现有代码时，可能会发现客户端开发人员已经事先考虑到这种情形。开发人员为客户端使用的服务提供了接口，如图 3.1 所示。`RequiredInterface` 接口声明了 `Client` 类所要调用的 `requiredMethod()` 方法。在 `ExistingClass` 类中，则定义了 `usefulMethod()` 方法，它是 `Client` 类需要的实现。若要对 `ExistingClass` 类进行适配，满足客户端对象的需要，就可以编写一个继承自 `ExistingClass`，并同时实现 `RequiredInterface` 接口的类，通过重写 `requiredMethod()` 方法将客户端的请求委派给 `usefulMethod()` 方法。

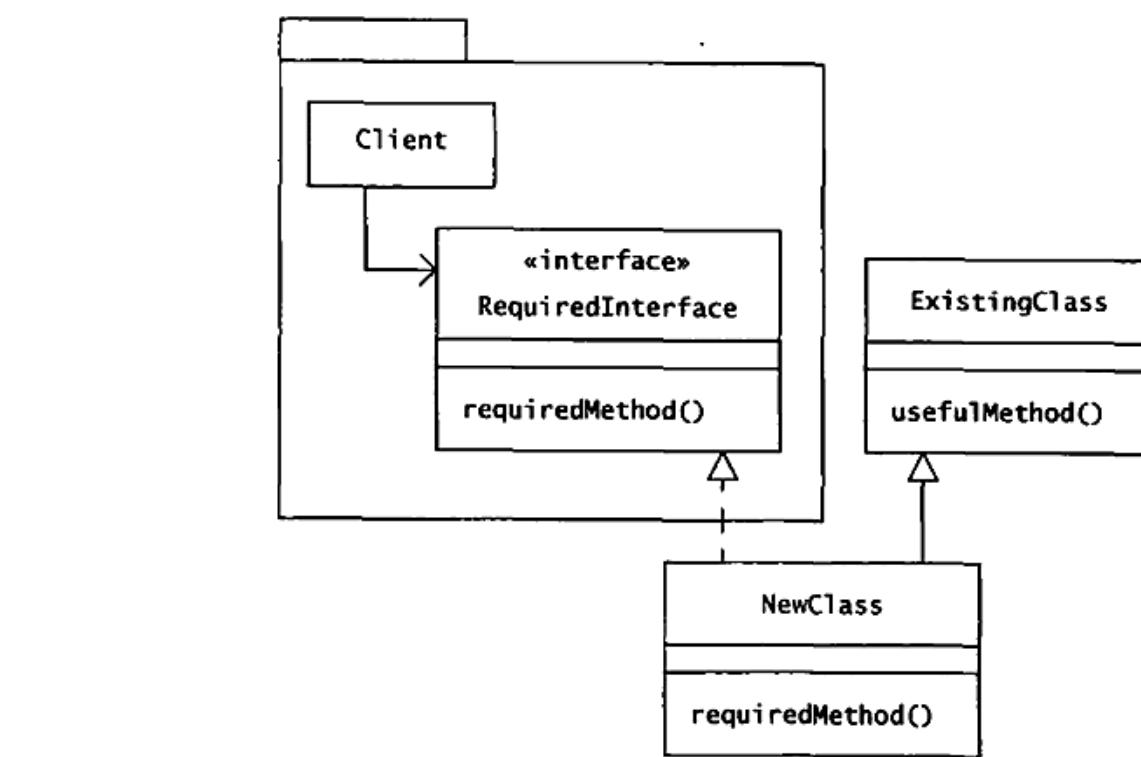


图 3.1 当客户端代码的开发人员在考虑如何满足客户端需求时，可以考虑通过适配现有代码来实现接口

图 3.1 中的 NewClass 类就是适配器模式的一个例子。该类的实例同时也是 RequiredInterface 的实例。换言之，NewClass 类满足了客户端的需求。

为了更具体地说明，假定我们为 Oozinoz 公司开发仿真火箭的飞行与实时控制程序。在仿真功能包中，包含了一个事件仿真器，它能够探测到多个火箭启动时产生的影响。这是一个指定了火箭行为的接口，图 3.2 展示了该功能包。

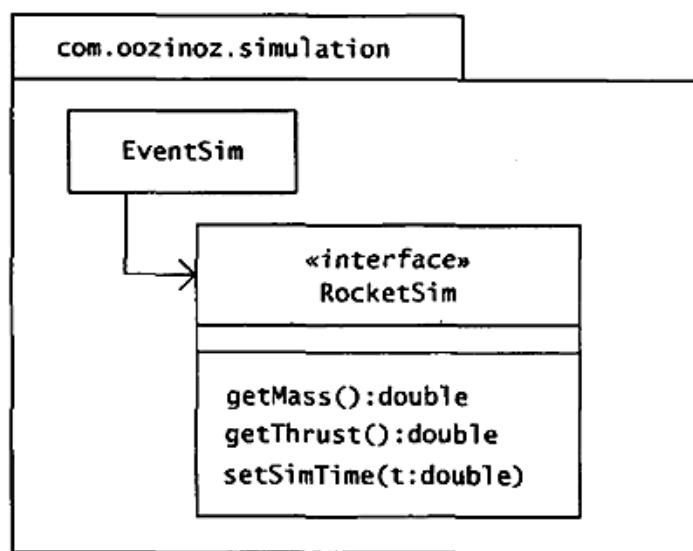


图 3.2 仿真功能包清晰地定义了火箭仿真飞行的需求

假设在 Oozinoz 公司，需要将 `PhysicalRocket` 类放到仿真功能中。该类提供的方法类似仿真器需要的功能行为。此时，就可以运用适配器模式，创建 `PhysicalRocket` 的子类，并同时实现 `RocketSim` 接口。图 3.3 展示了这一设计的部分内容。

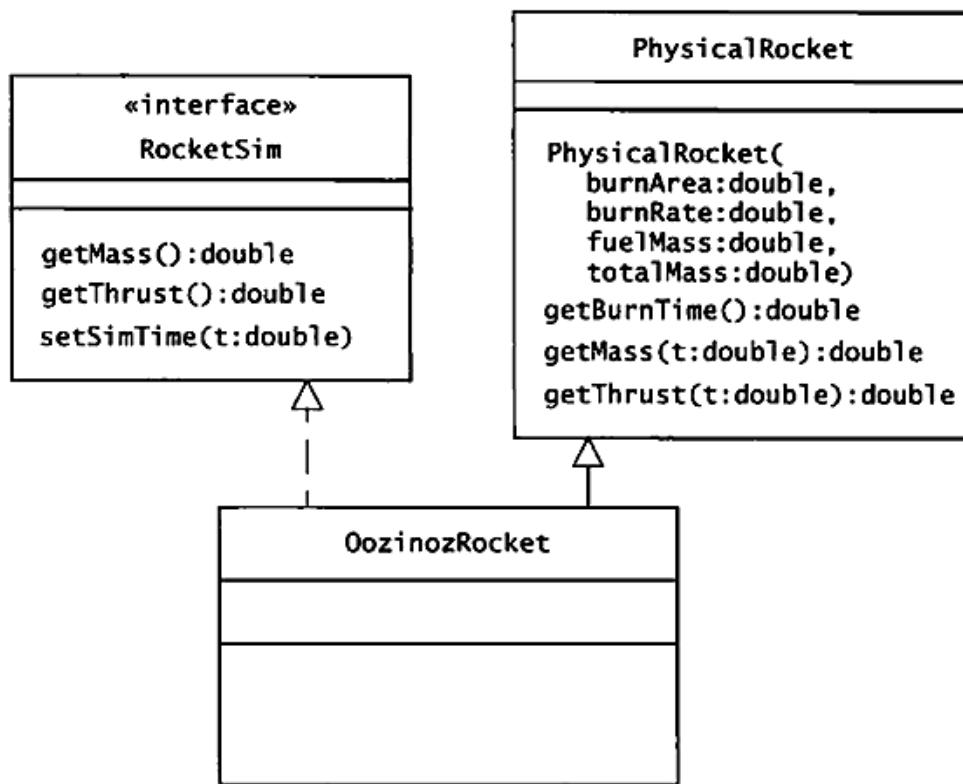


图 3.3 本图展示了设计完成后，类的设计对 Rocket 类进行适配，以满足 RocketSim 接口的需要

`PhysicalRocket` 类拥有仿真器需要的信息，但它的方法并不完全匹配 `RocketSim` 接口中声明的仿真功能。主要的差异在于仿真器保留了一个内部时钟，它不时地会调用 `setSimTime()` 方法更新仿真对象。若要适配 `PhysicalRocket` 类以满足仿真器的需求，`OozinozRocket` 对象可以维持一个实例变量，用来传递 `PhysicalRocket` 类需要的方法。

挑战 3.1

完成图 3.3 中的类图，展现 `OozinozRocket` 类的设计，它将让 `PhysicalRocket` 对象作为 `RocketSim` 对象参与到仿真行为中。假定你无法修改 `RocketSim` 与 `PhysicalRocket` 类。

答案参见第 298 页

`PhysicalRocket` 类的代码稍显复杂，因为它包含了 Oozinoz 用来模拟火箭行为的物理逻

辑。然而，这正是我们希望重用的部分。`OozinozRocket` 适配器类只是简单地将调用转为使用超类拥有的方法。这个新的子类的代码如下所示：

```
package com.oozinoz.firework;
import com.oozinoz.simulation.*;

public class OozinozRocket
    extends PhysicalRocket implements RocketSim {
    private double time;

    public OozinozRocket(
        double burnArea, double burnRate,
        double fuelMass, double totalMass) {
        super(burnArea, burnRate, fuelMass, totalMass);
    }

    public double getMass() {
        // 挑战!
    }

    public double getThrust() {
        // 挑战!
    }

    public void setSimTime(double time) {
        this.time = time;
    }
}
```

挑战 3.2

完成包括 `getMass()` 与 `getThrust()` 方法的 `OozinozRocket` 类的代码。

答案参见第 299 页

当客户端在接口中定义了它所期待的行为时，就可以运用适配器模式，提供一个实现该接口的类，并同时令其成为现有类的接口。倘若没有定义客户端期待的接口，也可以运用适配器模式，但必须使用“对象适配器”。

类与对象适配器

图 3.1 与图 3.3 的设计属于类的适配器，通过子类进行适配。在类的适配器中，新的适配类实现了需要的接口，并继承自现有的类。当你需要适配的一组方法并非被定义在接口中时，这种方式就不奏效了。此时就可以创建一个对象适配器，它使用了委派而非继承。图 3.4 展现了这样的设计（可以对比之前的类图）。

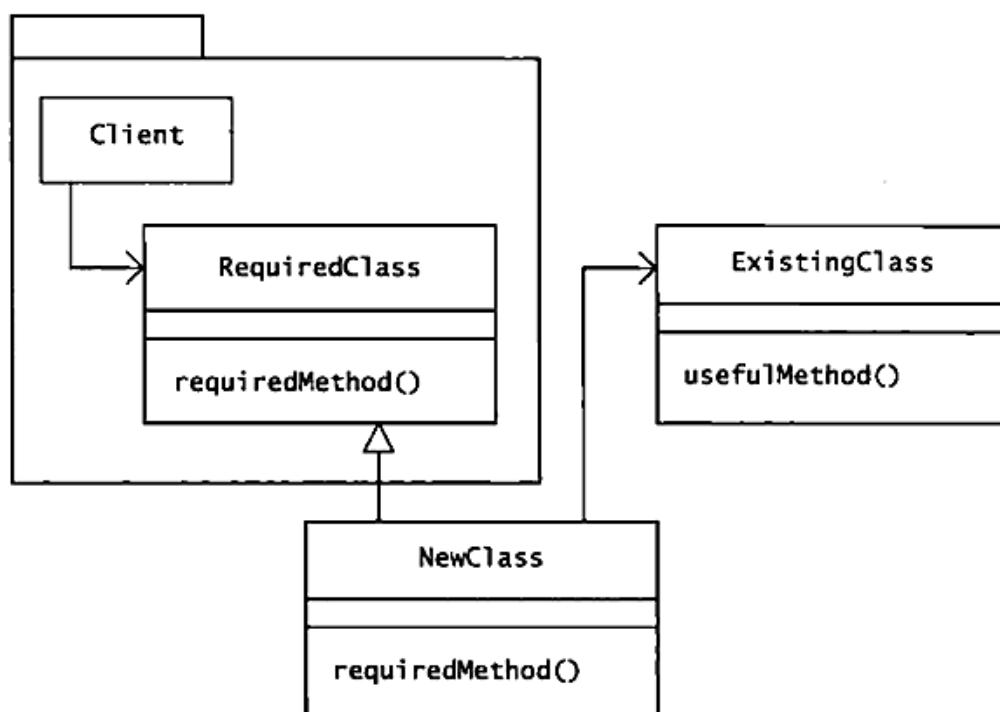


图 3.4 通过继承你所需要的类，可以创建一个对象适配器，利用现有类的实例对象，满足所需方法

图 3.4 中的 **NewClass** 类是适配器的一个例子。该类的实例同时也是 **RequiredClass** 类的实例。换言之，**NewClass** 类满足了客户端的需要。**NewClass** 类通过使用 **ExistingClass** 实例对象，可以将 **ExistingClass** 类适配为符合客户端的需要。

一个更为具体的例子是仿真程序包，它直接与 **Skyrocket** 类协作，而没有指定接口去定义仿真系统需要的行为。图 3.5 展示了该类的设计。

Skyrocket 类使用了火箭的基本模型。例如，类假设火箭要在燃料烧尽之后才会坠毁。假设你希望添加一些更为复杂的物理模型，而该模型由 **Oozinoz** 系统中的 **PhysicalRocket** 类使用。为了适配 **PhysicalRocket** 类以满足仿真系统，需要创建 **oozinozSkyrocket** 类作为对象适配器，它继承自 **Skyrocket**，同时使用了 **PhysicalRocket** 对象，如图 3.6 所示。

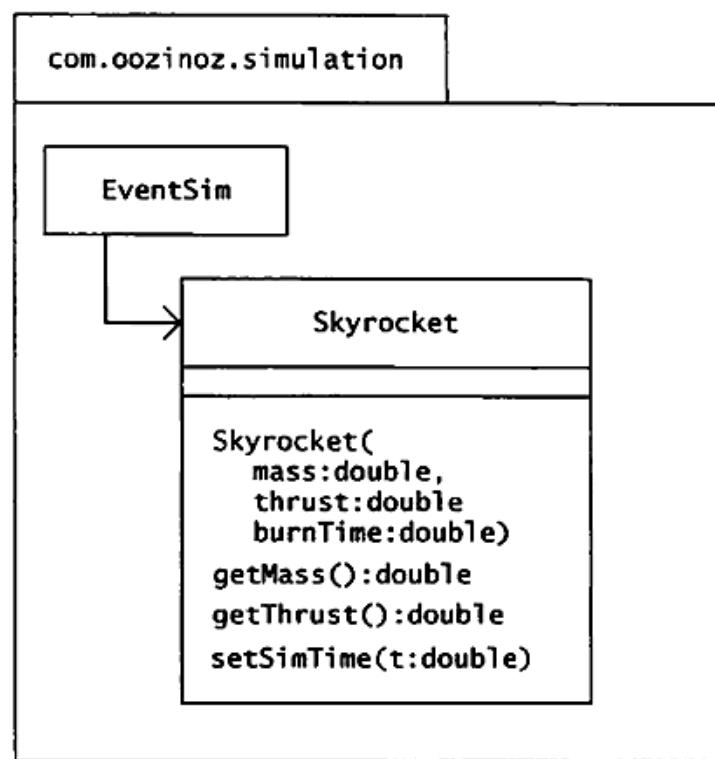


图 3.5 在这个替代设计中，com.oozinoz.simulation 包并没有指定对火箭进行建模所需的接口

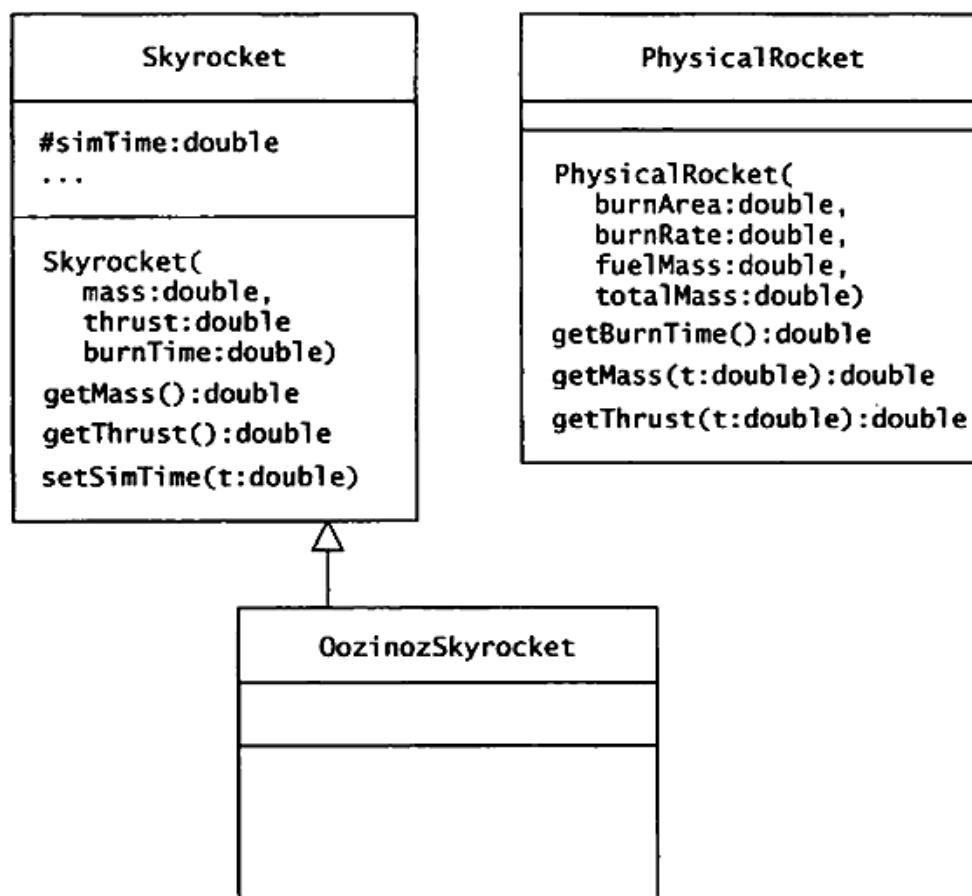


图 3.6 完成该类图使其能够体现对象适配器的设计，将现有的类转换，以满足拥有 Skyrocket 对象的客户端需求

作为一个对象适配器，`OozinozSkyrocket` 类继承自 `Skyrocket`，而非 `PhysicalRocket`。当仿真程序客户端需要 `Skyrocket` 对象时，可以令 `OozinozSkyrocket` 对象代替它。通过让 `simTime` 变量成为受保护的，`Skyrocket` 类就能够支持它的子类化。

挑战 3.3

完成图 3.6 所示的类图，使得 `OozinozRocket` 对象支持 `Skyrocket` 对象。

答案参见第 300 页

`OozinozSkyrocket` 类的代码如下所示：

```
package com.oozinoz.firework;
import com.oozinoz.simulation.*;

public class OozinozSkyrocket extends Skyrocket {
    private PhysicalRocket rocket;
    public OozinozSkyrocket(PhysicalRocket r) {
        super(
            r.getMass(0),
            r.getThrust(0),
            r.getBurnTime());
        rocket = r;
    }

    public double getMass() {
        return rocket.getMass(simTime);
    }

    public double getThrust() {
        return rocket.getThrust(simTime);
    }
}
```

`OozinozSkyrocket` 类可以为需要 `Skyrocket` 对象的仿真程序包提供 `OozinozSkyrocket` 类型的对象。总体而言，对象适配器在一定程度上解决了这一问题，即将对象适配为没有明确定义的接口。

与实现 `RocketSim` 接口相比，运用了对象适配器的 `Skyrocket` 类存在更大的风险。但我

们却不应该吹毛求疵，因为它仅仅是将方法标记为 `final`，使得我们不能防止子类去重写它们。

挑战 3.4

分析为何 `OozinozSkyrocket` 类使用的对象适配器设计要比类的适配器方式更加脆弱。

答案参见第 301 页

JTable 对数据的适配

在表中显示数据时，通常会运用对象适配器。Swing 提供了 `JTable` 控件用以显示表。显然，该控件的设计者并不知道你所要显示的数据。它并没有硬将数据接口塞到控件中，而是定义了 `TableModel` 接口。`JTable` 的实现使用了该接口。然后，你可以提供一个适配器，将数据转换为 `TableModel`，如图 3.7 所示。

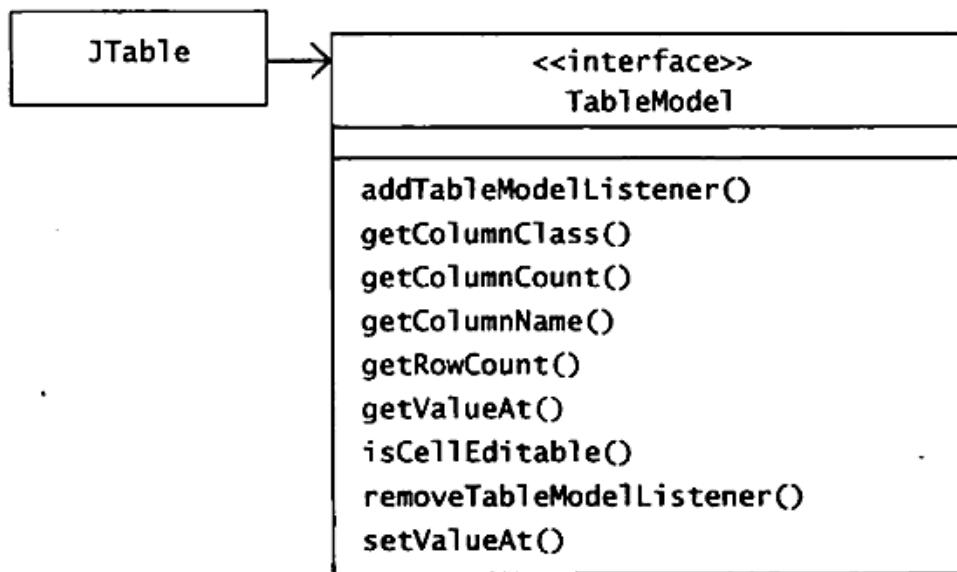


图 3.7 Swing 组件中的 `JTable` 类可以将实现了 `TableModel` 的数据显示到图形界面的表中

`TableModel` 定义了许多给出默认实现的方法。幸运的是，JDK 提供了抽象类的机制，它可以为 `TableModel` 中与特定领域逻辑有关的方法提供默认实现。图 3.8 展现了该类的设计。

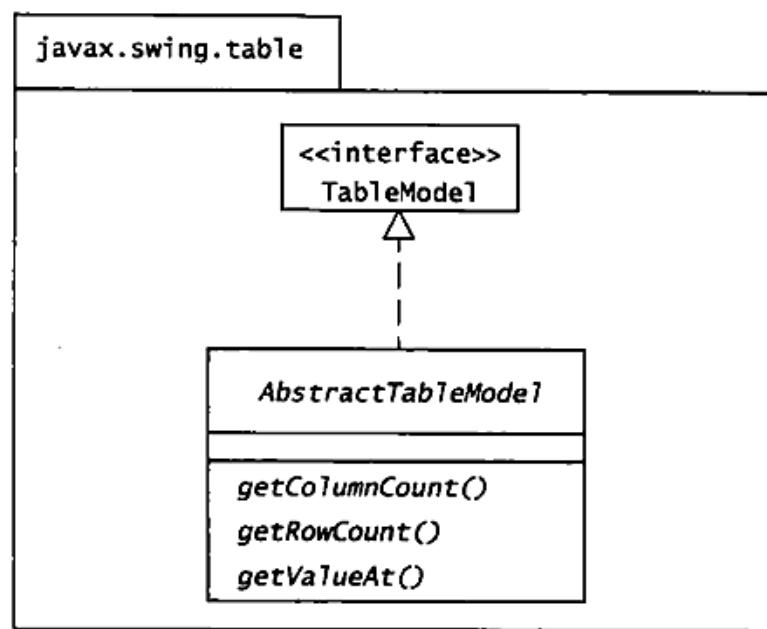


图 3.8 AbstractTableModel 类提供了定义在 TableModel 接口中大多数方法的实现

假设需要使用 Swing 的用户界面在表中显示几个火箭。如图 3.9 所示，可以创建 RocketTableModel 类将这组火箭适配为 TableModel 所期待的接口。

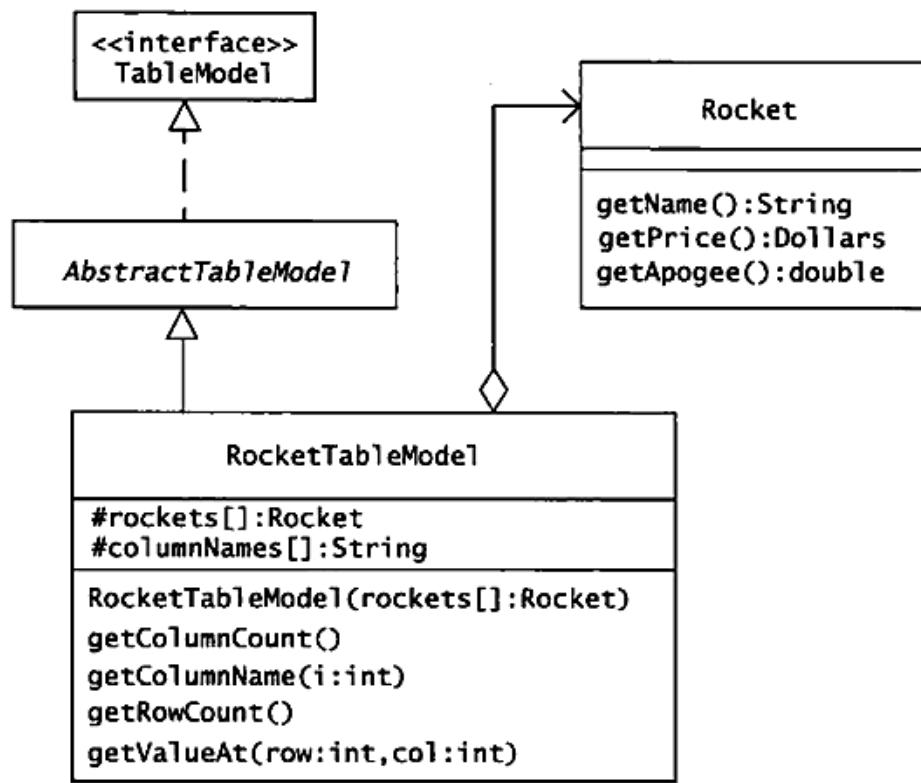


图 3.9 RocketTableModel 类将 TableModel 接口适配为 Oozinoz 领域对象 Rocket 类

RocketTableModel 类必须继承自 AbstractTableModel，因为后者是类而不是接口。无

论何时，只要我们需要使用的抽象类对要适配的接口提供支持，就必须使用对象适配器方式^{译注1}。不能使用类适配器的第二个原因是 `RocketTableModel` 并非 `Rocket` 的子类型。当适配类必须从多个对象处获得相关信息时，通常就应该使用对象的适配器。

注意区分：类的适配器继承自现有的类，同时实现目标接口；对象适配器继承自目标类，同时引用现有的类。

一旦创建了 `RocketTableModel` 类，就很容易在 Swing 的 `JTable` 对象中显示相关信息，如图 3.10 所示。

The screenshot shows a standard Java Swing application window titled "Rockets". Inside the window is a `JTable` component displaying two rows of data. The table has three columns with headers: "Name", "Price", and "Apogee". The first row contains the values "Shooter", "\$3.95", and "50.0". The second row contains the values "Orbit", "\$29.03", and "5000.0".

图 3.10 填充了火箭数据信息的 `JTable` 实例

```
package app.adapter;
import javax.swing.table.*;
import com.oozinoz.firework.Rocket;

public class RocketTableModel extends AbstractTableModel {
    protected Rocket[] rockets;
    protected String[] columnNames =
        new String[] { "Name", "Price", "Apogee" };

    public RocketTableModel(Rocket[] rockets) {
        this.rockets = rockets;
    }

    public int getColumnCount() {
        // 挑战!
    }
}
```

译注1：这里的假设前提是适配器对象必须要使用抽象类，因为抽象类提供了适配器对象需要的部分实现，而该抽象类又实现了客户端期待的接口，这就要求适配器对象必须继承抽象类。而同时，适配器对象还需要重用第三方对象（即目标对象）。重用的方式只能是继承或组合方式。由于 Java 是单继承语言，在已经继承了抽象类的情况下，无法再使用类的继承方式，因此只能将目标对象（这里即 `Rocket` 对象）以组合的方式传给适配器对象。

```
public String getColumnName(int i) {  
    // 挑战!  
}  
  
public int getRowCount() {  
    // 挑战!  
}  
  
public Object getValueAt(int row, int col) {  
    // 挑战!  
}  
}
```

挑战 3.5

完成 `RocketTableModel` 方法中的代码，使其能够将 `Rocket` 对象数组适配为 `TableModel`。

答案参见第 301 页

要实现图 3.10 所示的显示结果，可以创建一组 `rocket` 对象，并将其放到数组中，然后再根据该数组创建 `RocketTableModel` 实例，并使用 Swing 的类显示表的信息。`ShowRocketTable` 类给出了这一例子的实现：

```
package app.adapter;  
  
import java.awt.Component;  
import java.awt.Font;  
  
import javax.swing.*;  
  
import com.oozinoz.firework.Rocket;  
import com.oozinoz.utility.Dollars;  
  
public class ShowRocketTable {  
    public static void main(String[] args) {  
        setFonts();  
        JTable table = new JTable(getRocketTable());  
        table.setRowHeight(36);  
    }  
}
```

```
JScrollPane pane = new JScrollPane(table);
pane.setPreferredSize(
    new java.awt.Dimension(300, 100));
display(pane, "Rockets");
}

public static void display(Component c, String title) {
    JFrame frame = new JFrame(title);
    frame.getContentPane().add(c);
    frame.setDefaultCloseOperation(
        JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
}
private static RocketTableModel getRocketTable() {
    Rocket r1 = new Rocket(
        "Shooter", 1.0, new Dollars(3.95), 50.0, 4.5);
    Rocket r2 = new Rocket(
        "Orbit", 2.0, new Dollars(29.03), 5000, 3.2);
    return new RocketTableModel(new Rocket[] { r1, r2 });
}
private static void setFonts() {
    Font font = new Font("Dialog", Font.PLAIN, 18);
    UIManager.put("Table.font", font);
    UIManager.put("TableHeader.font", font);
}
}
```

只需要不到 20 行代码，`ShowRocketTable` 就实现了在图形化用户界面框架中生成表组件的功能。如果不使用适配器模式，可能需要上千行代码。`JTable` 类几乎可以处理显示表数据的各种功能，但它却无法事先知道你所要显示的数据。若要提供它所需要的数据，就应该求助于适配器模式。为了使用 `JTable`，可以实现 `JTable` 所期待的 `TableModel` 接口，提供希望显示的数据。

识别适配器

在第 2 章，我们已经分析了 `WindowAdapter` 类的价值所在。图 3.11 所示的 `MouseAdapter`

类，则是另一个范例，它可以为接口所需的方法提供桩实现。

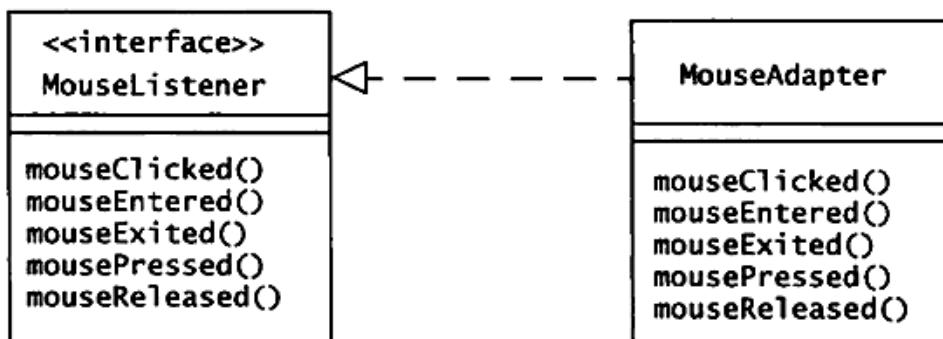


图 3.11 `MouseAdapter` 类可以为 `MouseListener` 接口提供需要的桩实现

挑战 3.6

在使用 `MouseAdapter` 类时，你是否运用了适配器模式？请阐释其原理（如果没有运用，请给出理由）。

答案参见第 302 页

小结

适配器模式使我们可以重用一个现有的类，以满足客户端的需要。当客户端通过接口表达其需求时，通常可以创建一个实现了该接口的新类，同时使该类继承自现有类。这种方式即类的适配器，它能够将客户端的调用转换为对现有类方法的调用。

当客户端没有指定它所需要的接口时，你就可以使用适配器模式。可能需要创建一个新的客户端子类，它将使用现有类的实例。这种方式通过创建一个对象适配器，将客户端的调用指向现有类的实例。如果我们不需要（或许不能）重写客户端可能调用的方法时，这种方式可能存在一定的危险性。

Swing 中的 `JTable` 组件是运用适配器模式的一个绝佳范例。通过定义 `TableModel` 接口，`JTable` 组件将客户端需要的表信息存储到自身对象中。通过编写一个适配器对象，轻易就可以让一个领域对象满足表数据的需求，例如 `Rocket` 类。

若要使用 `JTable`，需要定义一个对象适配器，可以将调用委派给现有类的实例对象。对于 `JTable`，有两个原因不能使用类的适配器。首先，需要创建一个表的适配器作为 `AbstractTableModel` 的子类，这样就无法再继承自现有类了。其次，`JTable` 需要一组对象，而对象适配器更适合需要将多个对象的信息进行适配的情形。

在设计自己的系统时，需要充分考虑功能的强大与灵活性。这样你和其他程序员都可以从适配器模式架构的使用中获益。

第4章

外观（Facade）模式

面向对象编程的最大优势，在于它能够防止应用程序成为混乱纠缠的小块程序。理想状态下，面向对象系统应该是将其他可复用的工具类中的行为组织在一起的最小的类。工具包或子系统的开发人员通常会将设计良好的类组织为包，而不是提供应用程序。Java 类库中的包通常可以当做工具包（toolkits），以保证我们能够随意地开发各种特定领域的应用程序。

伴随工具包复用性而来的问题是：面向对象子系统中类的多样性可能导致选择太多，以至于希望使用该工具包的开发人员变得茫然失措。诸如 Eclipse 这样的集成开发环境（IDE），可以将开发人员从包的错综复杂中解脱出来，然而，IDE 有时却会产生大量开发人员不需要的代码。

简化工具包的另一途径是使用外观（facade）模式。只需少量代码，就能提供典型的、无修饰用法的类库中的类。一个外观就是一个类，它包含的功能介于工具包与完整的应用程序之间，为工具包或子系统的类提供了简单的用法。

外观模式的意图是为子系统提供一个接口，便于它的使用。

外观类、工具类和示例类

外观类可能全是静态方法，在 UML 中，这样的类被称为 utility（工具）。稍后，我们将介

绍一个全为静态方法的 UI（用户界面）类，尽管这样做可能会影响到将来子类对这些方法的重写。

示例类（Demo）则用于展示如何使用类或子系统。以本例而言，示例类提供了和外观类相同的价值。

挑战 4.1

写出示例类和外观类的两点区别。

答案参见第 302 页

`javax.swing` 包包含了 `JOptionPane`，以便弹出一个标准对话框。如下代码会反复显示对话框，直到用户单击了 Yes 按钮，如图 4.1 所示。

```
package app.facade;

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

public class ShowOptionPane {
    public static void main(String[] args) {
        Font font = new Font("Dialog", Font.PLAIN, 18);
        UIManager.put("Button.font", font);
        UIManager.put("Label.font", font);

        int option;
        do {
            option = JOptionPane.showConfirmDialog(
                null,
                "Had enough?",
                "A Stubborn Dialog",
                JOptionPane.YES_NO_OPTION);
        } while (option == JOptionPane.NO_OPTION);
    }
}
```

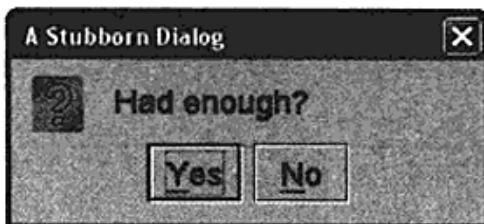


图 4.1 JOptionPane 类使显示这样的对话框变得很容易

挑战 4.2

JOptionPane 类让对话框的显示变得简单，那么，该类是一个外观类、工具类还是示例类？给出答案，并阐述你的理由。

答案参见第 303 页

挑战 4.3

Java 类库中很少有外观类，为什么？

答案参见第 303 页

重构到外观模式

外观模式常见于一些常规的应用程序开发。如果根据关注点将代码分解为不同的类，就可以提取一个类，它的主要职责是为子系统提供简便的访问方式，从而完成对系统的重构。考虑 Oozinoz 公司早期的一个例子，当时并没有 GUI 开发的规范。假设你读到程序员写的一段程序，用于展示一颗哑弹的飞行路径，如图 4.2 所示。

设计的焰火弹会在高空爆炸，发出绚烂的光芒。但是偶尔，某个焰火弹却不会爆炸（即哑弹），而是缓慢坠落到地面。与火箭不同，焰火弹是没有动力的。倘若忽略风和空气的阻力，一颗哑弹的飞行路径就是一条简单的抛物线。图 4.3 展示了执行 `ShowFlight.main()` 后的一个窗口截图。

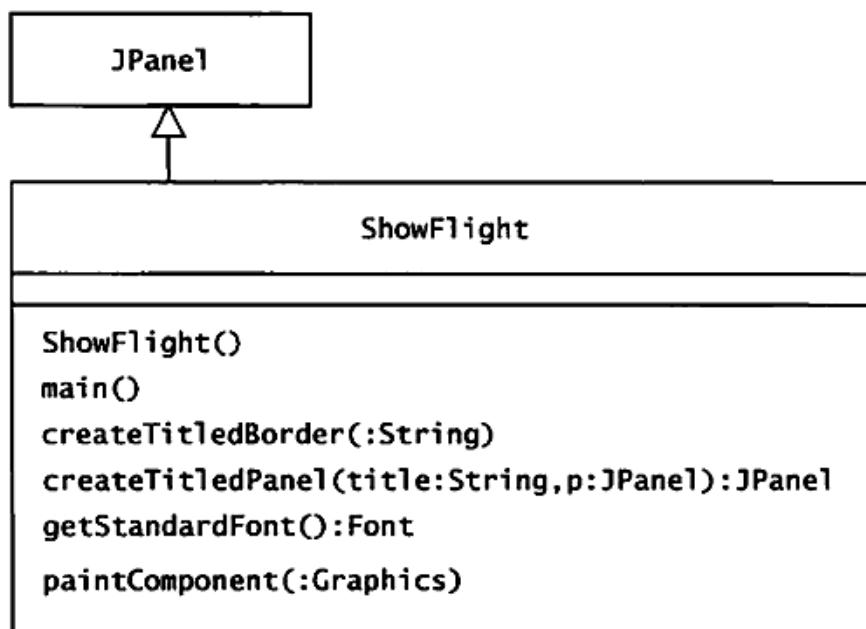


图 4.2 ShowFlight 类显示了哑弹的飞行路径

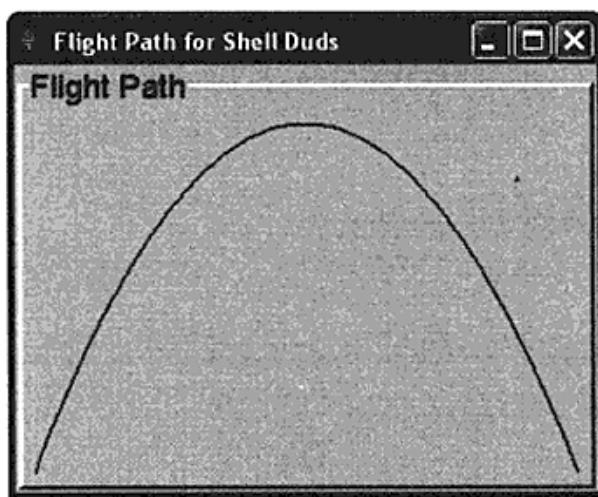


图 4.3 ShowFlight 应用展示了颗哑弹将在何处落地

ShowFlight 类存在一个问题：它混杂了三个功能。它的首要功能是为飞行路径提供一个面板，第二个功能是作为一个完整的应用程序，需要将飞行路径的面板显示在一个具有标题的边框内，最后一个功能是计算飞行路径的抛物线。ShowFlight 类定义了 `paintComponent()` 方法执行计算，代码如下。

```
protected void paintComponent(Graphics g) {
    super.paintComponent(g); // 绘制背景
    int nPoint = 101;
    double w = getWidth() - 1;
    double h = getHeight() - 1;
```

```
int[] x = new int[nPoint];
int[] y = new int[nPoint];
for (int i = 0; i < nPoint; i++) {
    // t 从 0 到 1
    double t = ((double) i) / (nPoint - 1);
    // x 从 0 到 w
    x[i] = (int) (t * w);
    // 当 t=0 和 1 时, y 为 h; 当 t=0.5 时, y 为 0
    y[i] = (int) (4 * h * (t - .5) * (t - .5));
}
g.drawPolyline(x, y, nPoint);
}
```

如需了解代码中如何建立哑弹轨迹坐标的 *x* 和 *y* 值, 请参考下页关于“参数方程”的介绍。

这里无须类的构造函数。它使用静态工具方法去包装面板的标题, 并定义了标准字体。

```
public static TitledBorder createTitledBorder(String title){
    TitledBorder tb = BorderFactory.createTitledBorder(
        BorderFactory.createBevelBorder(BevelBorder.RAISED),
        title,
        TitledBorder.LEFT,
        TitledBorder.TOP);
    tb.setTitleColor(Color.black);
    tb.setTitleFont(getStandardFont());
    return tb;
}

public static JPanel createTitledPanel(
    String title, JPanel in) {
    JPanel out = new JPanel();
    out.add(in);
    out.setBorder(createTitledBorder(title));
    return out;
}

public static Font getStandardFont() {
    return new Font("Dialog", Font.PLAIN, 18);
}
```

注意, `createTitledPanel()`方法把提供的控件放进一个斜边框里, 以提供一个小的间隙

(padding)，保证飞行曲线不碰到容器的边缘。Main()方法也为包含了应用程序控件的表单对象增加了间隙。

```
public static void main(String[] args) {  
    ShowFlight flight = new ShowFlight();  
    flight.setPreferredSize(new Dimension(300, 200));  
    JPanel panel = createTitledPanel("Flight Path", flight);  
  
    JFrame frame = new JFrame("Flight Path for Shell Duds");  
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    frame.getContentPane().add(panel);  
  
    frame.pack();  
    frame.setVisible(true);  
}
```

图 4.3 展示了该程序的执行过程。

参数方程

当你需要画一条曲线时，很难把 y 值表示为 x 值的函数。参数方程可以让你把 x 和 y 定义成其他参数的函数。特别的，你可以将曲线的绘制时间表示为 $0 \sim 1$ 之间的参数 t ，并且可以将 x 和 y 定义成参数 t 的函数。

例如，你想要哑弹的飞行抛物线穿过 Graphics 对象，假设该对象的宽度是 w ，那么关于 x 的参数方程可以简单表示为：

$$x = w * t$$

注意， t 的变化范围是 $0 \sim 1$ ， x 的变化范围是 $0 \sim w$ 。

抛物线的 y 值随着 t 的平方值变化而变化，方向越往下， y 值就会随之而递增。对于抛物线，当 $t=0.5$ 时， y 应该为 0。

因此我们能够得到如下方程：

$$y = k * (t - .5) * (t - .5)$$

这里， k 代表一个指定的常量。该方程规定了当 $t=0.5$ 时， $y=0$ ，并且当 $t=0$ 或 $t=1$ 时， $y=h$ ，也就是显示区域的高度。经过一些代数运算，可以推导出如下完整的关于 y 的方程：

$$y = 4 * h * (t - .5) * (t - .5)$$

图 4.3 显示了方程式绘出的抛物线。

参数方程的另一个优点是，我们可以用它绘制出一个给定 x 值多个 y 值的曲线。比如绘制一个圆。半径为 1 的圆的方程为：

$$x^2 + y^2 = r^2$$

或者

$$y = \pm \sqrt{r^2 - x^2}$$

如果每个 x 值对应两个 y 值，情况会更加复杂。要正确地调整 Graphics 对象的高和宽，并进行绘制，则较为困难。因此，可以利用极坐标来简化绘制圆的功能。

$$\begin{aligned}x &= r * \cos(\theta) \\y &= r * \sin(\theta)\end{aligned}$$

这两个参数方程将 x 和 y 表示成新参数 θ 的函数。 θ 表示绘制圆的过程中扫过的弧度，范围是 $0 \sim 2\pi$ 。可以设置一个圆的半径，使它能塞在高度为 h ，宽度为 w 的图形对象中。下面是一些在 Graphics 对象中绘制圆的参数方程：

$$\begin{aligned}\theta &= 2 * \pi * t \\r &= \min(w, h)/2 \\x &= w/2 + r * \cos(\theta) \\y &= h/2 - r * \sin(\theta)\end{aligned}$$

将这些方程转换为代码后，就可以产生如图 4.4 所示的圆（可以在 oozinoz.com 获得生成这一显示效果的 ShowCircle 应用程序代码）。

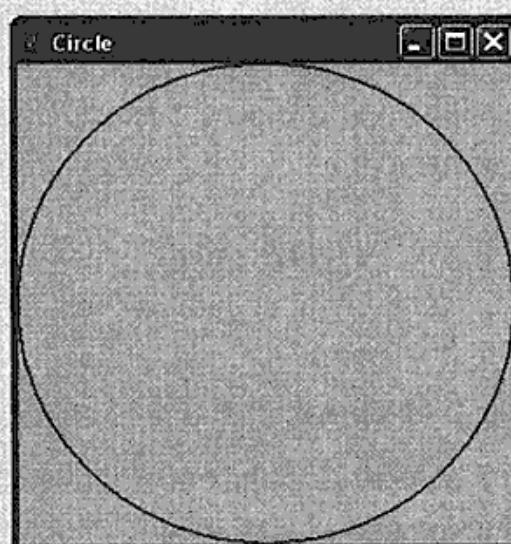


图 4.4 当一个 x 值对应多个 y 值时，参数方程可以简化对曲线的建模

下面绘制圆的代码是对数学公式的直接翻译。由于像素在水平方向和垂直方向的范围分别是 0 到 width-1 和 0 到 height-1，因此我们在代码中相应减小了 Graphics 对象的高和宽。

```
package app.facade;

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

import com.oozinoz.ui.SwingFacade;

public class ShowCircle extends JPanel {
    public static void main(String[] args) {
        ShowCircle sc = new ShowCircle();
        sc.setPreferredSize(new Dimension(300, 300));
        SwingFacade.launch(sc, "Circle");
    }
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        int nPoint = 101;
        double w = getWidth() - 1;
        double h = getHeight() - 1;
        double r = Math.min(w, h) / 2.0;
        int[] x = new int[nPoint];
        int[] y = new int[nPoint];
        for (int i = 0; i < nPoint; i++) {
            double t = ((double) i) / (nPoint - 1);
            double theta = Math.PI * 2.0 * t;
            x[i] = (int) (w / 2 + r * Math.cos(theta));
            y[i] = (int) (h / 2 - r * Math.sin(theta));
        }
        g.drawPolyline(x, y, nPoint);
    }
}
```

根据 t 定义 x 和 y 函数，可以分别计算 x 和 y 的值。这比把 y 定义成 x 的函数要简单，也便于将 x 和 y 映射到 Graphics 对象坐标中。另外，当 y 是 x 的非单值函数时，参数方程也可以简化曲线的绘制。

`ShowFlight` 类可以工作，然而根据关注点分离的原则，我们应将其重构为多个单独的类，以提高可维护性以及可重用性。假设你正在进行设计评审，并且决定做出如下改变：

- 引入一个 `Function` 类，它定义了 `f()` 方法，接收一个 `double` 值（时间值），返回一个 `double` 值（函数的值）。
- 将 `ShowFlight` 类移入 `PlotPanel` 类，改为使用 `Function` 对象来获取 `x` 和 `y` 的值。定义 `PlotPanel` 构造函数接收两个 `Function` 实例和绘图所需的点数。
- 将 `createTitledPanel()` 方法移到已存在的 UI 工具类中，来实现一个像当前 `ShowFlight` 类那样带有标题的面板。

挑战 4.4

完成图 4.5 所示的类图，用以展示将 `ShowFlight` 重构为三个类的代码：`Function` 类、实现两个参数功能的 `PlotPanel` 类和一个 UI 外观类。在你的重新设计中，让类 `ShowFlight2` 为获取 `y` 值创建一个 `Function`，并让 `main()` 方法启动程序。

答案参见第 303 页

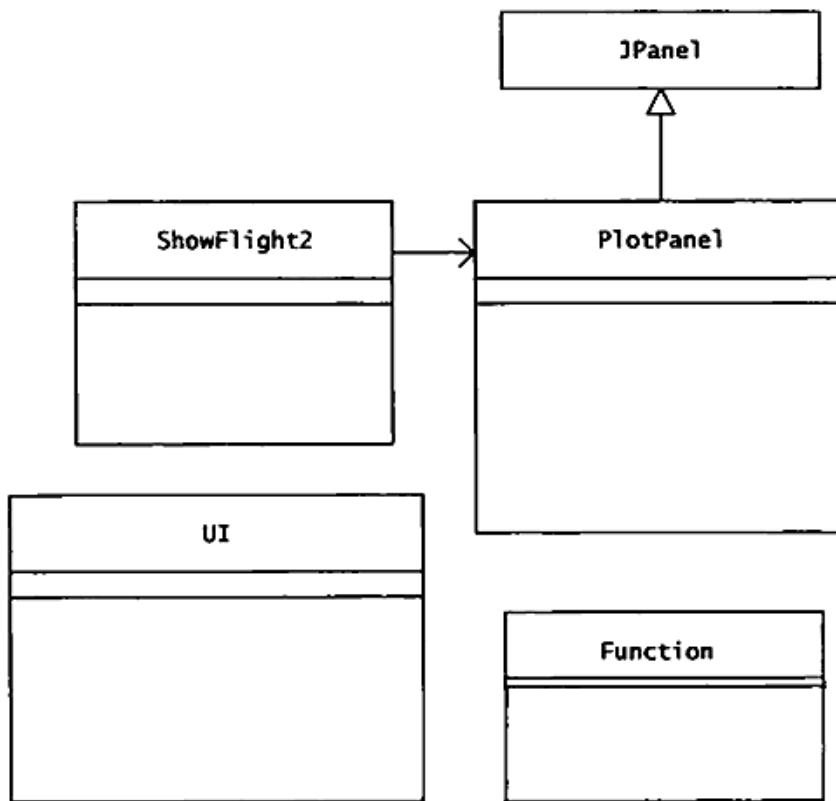


图 4.5 被重构为多个类的飞行轨迹应用程序，每个类都有各自的职责

重构之后, Function 类定义了参数方程。倘若要创建一个包含 Function 类与其他类型的 com.oozinoz.function 包, 则 Function.java 的核心代码可能是:

```
public abstract double f(double t);
```

经过重构, PlotPanel 类只拥有了一个职责: 显示一对参数方程, 代码如下:

```
package com.oozinoz.ui;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;
import com.oozinoz.function.Function;

public class PlotPanel extends JPanel {
    private int points;
    private int[] xPoints;
    private int[] yPoints;

    private Function xFunction;
    private Function yFunction;

    public PlotPanel(
        int nPoint, Function xFunc, Function yFunc) {
        points = nPoint;
        xPoints = new int[points];
        yPoints = new int[points];
        xFunction = xFunc;
        yFunction = yFunc;
        setBackground(Color.WHITE);
    }

    protected void paintComponent(Graphics graphics) {
        double w = getWidth() - 1;
        double h = getHeight() - 1;

        for (int i = 0; i < points; i++) {
            double t = ((double) i) / (points - 1);
            xPoints[i] = (int) (xFunction.f(t) * w);
        }
    }
}
```

```
        yPoints[i] = (int) (h * (1 - yFunction.f(t)));
    }

    graphics.drawPolyline(xPoints, yPoints, points);
}
}
```

注意，`PlotPanel` 类目前是 `com.oozinoz.ui` 包的一部分，和 `UI` 类处于同一位置。对 `ShowFlight` 类进行重构后，`UI` 类也拥有了 `createTitledBorder()` 和 `createTitledBorderBorder()` 方法。`UI` 类逐步演化为外观类，使之更容易使用 Java 控件。

使用这些控件的应用程序可能是一个很小的类，该类唯一的职责就是对这些控件进行布局并显示它们。例如，类 `ShowFlight2` 的代码如下：

```
package app.facade;

import java.awt.Dimension;
import javax.swing.JFrame;
import com.oozinoz.function.Function;
import com.oozinoz.function.T;
import com.oozinoz.ui.PlotPanel;
import com.oozinoz.ui.UI;

public class ShowFlight2 {
    public static void main(String[] args) {
        PlotPanel p = new PlotPanel(
            101,
            new T(),
            new ShowFlight2().new YFunction());
        p.setPreferredSize(new Dimension(300, 200));

        JFrame frame = new JFrame(
            "Flight Path for Shell Duds");
        frame.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(
            UI.NORMAL.createTitledBorder("Flight Path", p));

        frame.pack();
    }
}
```

```
frame.setVisible(true);
}

private class YFunction extends Function {
    public YFunction() {
        super(new Function[] {} );
    }

    public double f(double t) {
        // 当 t 等于 0 或 1 时, y 为 0; 当 t 等于 0.5 时, y 为 1
        return 4 * t * (1 - t);
    }
}
}
```

`ShowFlight2` 类为哑弹的飞行路径提供了 `YFunction` 类。`main()` 方法用来布局和显示用户界面。这个类的运行结果和最初的 `ShowFlight` 类相同。但是，现在你拥有了一个可重用的外观类，它可以简化 Java 应用程序中图形用户界面的创建。

小结

整体而言，应该对子系统中的类进行重构，直到每个类都有一个明确的目的。这可以使你的代码更容易维护，但也可能让使用该子系统的用户变得无所适从。为了让调用这些代码的开发人员使用更方便，可以为子系统提供示例程序或者外观类。通常，示例程序可以独立运行，却无法重用，仅用于演示使用子系统的方法。外观类则是可配置、可重用的类，提供了高层次的接口，使得子系统的使用更加方便。

合成（Composite）模式

合成模式的英文为 Composite，是一组对象的组合，这些对象可以是容器对象，表现为组的概念；另外一些对象则代表了单对象，或称为叶子对象。在对组合进行建模时，必须注意两个重要的概念。第一个概念是组对象允许包含单对象，也可以再包含其他的组对象（常见的错误是将组对象设计为只允许包含叶子对象）。第二个概念则是要为组合对象和单对象定义共同的行为。结合这两个概念，就可以为组对象与单对象定义统一的类型，并将该组对象建模为包含同等类型对象的集合。

合成模式的意图是为了保证客户端调用单对象与组合对象的一致性。

常规组合

图 5.1 展示了一个经典的组合结构。Leaf 类和 Composite 类都实现自一个抽象的 Component 通用接口，同时，Composite 对象又包含了其他的 Composite 和 Leaf 对象的集合。

注意，图 5.1 中的 Component 类是一个抽象类，它未包含任何一个具体方法，因而可以将其定义为接口，让 Leaf 类和 Composite 类去实现它。

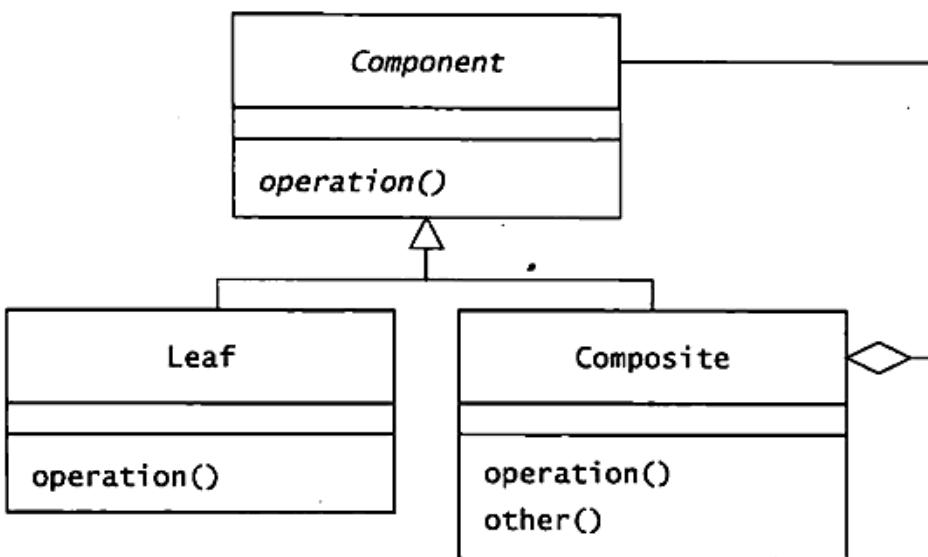


图 5.1 合成模式的关键在于组合对象可以包含其他组合对象（不仅仅是叶子对象），且 `Composite` 和 `Leaf` 节点共享了一个共同的接口

挑战 5.1

在图 5.1 中，为何 `Composite` 类维持了包含 `Component` 对象的集合，而不是仅包含叶子对象？

答案参见第 305 页

合成模式中的递归行为

在对生产焰火的机器进行处理时，Oozinoz 的工程师们观察到组合的特征。工厂是由车间组成的，每个车间有一条或多条生产线。一条生产线上有很多机器，机器之间相互协作，以保证生产进度。Oozinoz 的开发人员设计了图 5.2 所示的类图，将这些工厂、车间、生产线看做是“机器”的组合来进行建模。

如图 5.2 所示，单台机器与机器的组合都定义了 `getMachineCount()` 方法，它会返回给定组件中机器的数量。

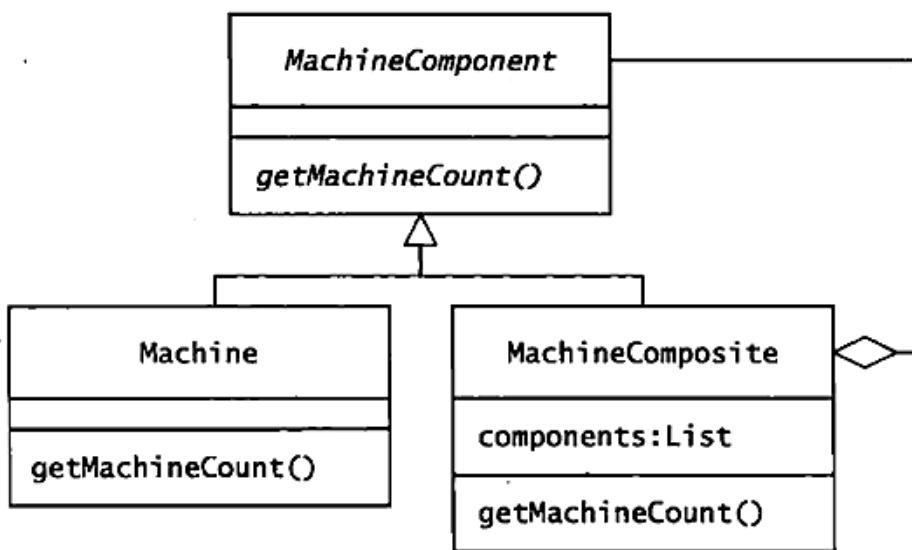


图 5.2 对于单台机器与组合机器而言, `getMachineCount()`方法都是它们需要的

挑战 5.2

写出 `Machine` 与 `MachineComposite` 中 `getMachineCount()` 方法的实现代码。

答案参见第 305 页

假设我们考虑给 `MachineComponent` 增加如下方法:

| 方 法 | 行 为 |
|-------------------------------|-----------------------|
| <code>isCompletelyUp()</code> | 判断组件中的所有机器是否都处于“up”状态 |
| <code>stopAll()</code> | 停止组件中所有机器的运作 |
| <code>getOwners()</code> | 返回负责管理机器的一组工程师 |
| <code>getMaterial()</code> | 返回机器组件中所有正在处理的原料 |

在 `MachineComponent` 中, 每个方法的定义和操作都是递归的。例如, 某个组合中机器的数量是该组合中所有组件包含的机器数量总和。

挑战 5.3

为 `MachineComponent` 声明的每个方法, 写出表 5.1 中 `MachineComposite` 中的递归定义与 `Machine` 中的非递归定义。

表 5.1 为方法定义

| 方 法 | 类 | 定 义 |
|-------------------|------------------|---------------|
| getMachineCount() | MachineComposite | 返回组合中每个组件的数量和 |
| | Machine | 返回 1 |
| isCompletelyUp() | MachineComposite | ？ ？ |
| | Machine | ？ ？ |
| stopAll() | MachineComposite | ？ ？ |
| | Machine | ？ ？ |
| getOwners() | MachineComposite | ？ ？ |
| | Machine | ？ ？ |
| getMaterial() | MachineComposite | ？ ？ |
| | Machine | ？ ？ |

答案参见第 306 页

组合、树与环

在合成结构中，如果一个节点拥有对其他节点的引用，则该节点就是一棵树。然而，这个定义太宽泛了。我们可以引用图论中的一些理论为对象建模，以示其精确性。倘若将对象看做节点，将对象间的引用看做边，就可以将对象模型绘制为图形结构。

考虑一个鉴定或分析化学药品的对象模型。`Assay` 类拥有一个 `Batch` 类型的 `batch` 属性，`Batch` 类则拥有一个 `Chemical` 类型的 `chemical` 属性。假设存在一个特殊的 `Assay` 对象 `a`，它的 `batch` 属性引用 `Batch` 对象 `b`，`b` 的 `chemical` 属性引用 `Chemical` 对象 `c`。图 5.3 展示了该对象模型的两种可供选择的对象图。(若要了解更多关于如何使用 UML 描述对象模型的信息，请参考附录 D。)

`a` 引用 `b`，`b` 引用 `c`，这一系列的引用导致出现了一条从 `a` 到 `c` 的路径。环 (cycle) 指的是路径中包含了出现两次的节点。倘若 `Chemical` 对象 `c` 反向引用 `Assay` 对象 `a` 的话，将会在对象模型中出现一个环。

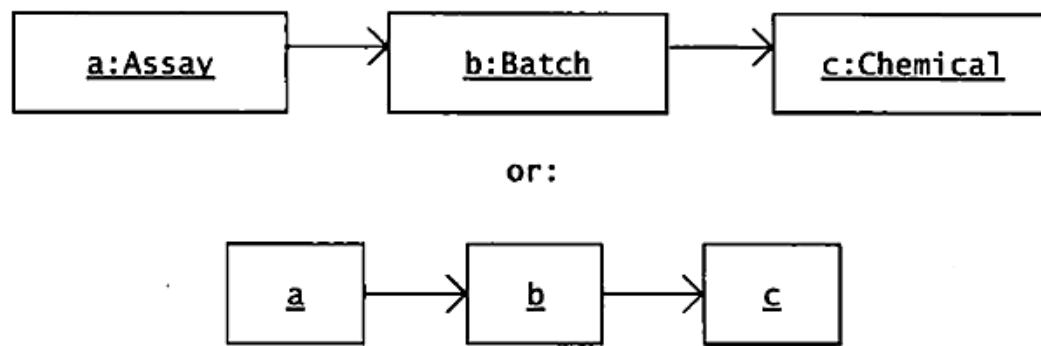


图 5.3 两幅 UML 图展示了描述相同信息的两种不同体现：对象 a 引用对象 b，对象 b 引用对象 c

对象模型是有向图，每个对象引用都会有一个方向。图论中的术语“树”通常指的是无向图，然而，只要满足下列条件，一个有向图也可以被称为树：

- 存在一个没有任何引用指向自身的根节点。
- 其他每个节点都只有一个引用该节点的父节点。

为何要考虑树中图的概念呢？因为合成模式特别适合这种结构形式。（如你所见，可以使用有向无环图甚至有环图实现合成模式，但是这需要额外的工作，必须加倍小心。）

图 5.3 中的对象模型描述的是一棵简单树。对于更大的对象模型，很难去鉴别模型是否为树。图 5.4 展示了 plant 的工厂对象模型，plant 是 MachineComposite 类的对象。这个 plant 包含一个含有三台机器的车间：mixer、press 和 assembler。plant 对象的机器列表组件包含对 mixer 的直接引用。

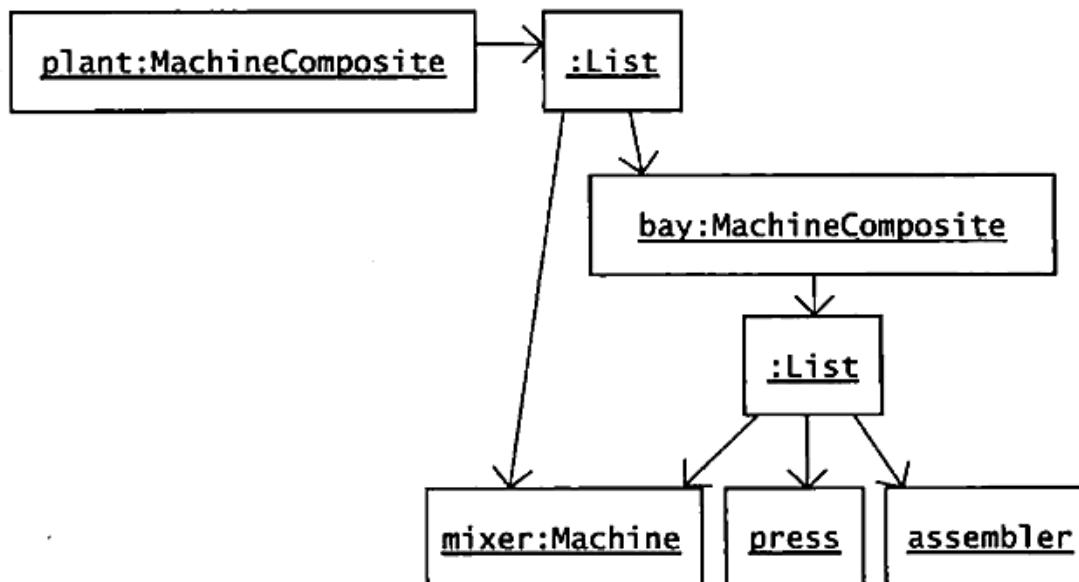


图 5.4 既不是环也不是树的对象模型图

图 5.4 中的对象图不包括环，但由于有两个对象同时引用了 `mixer` 对象，因此它也不是树。如果删除或者忽视 `plant` 对象和它相应的 `list`, `bay` 对象就成为树的根了。

假设合成模式是树形结构，而系统又允许使用不为树形结构的 `Composite`，则合成模式中的方法就存在问题。挑战 5.2 要求定义一个 `getMachineCount()` 操作，`Machine` 类对此方法的实现无疑是正确的：

```
public int getMachineCount() {  
    return 1;  
}
```

`MachineComposite` 类同样正确地实现了 `getMachineCount()` 方法，返回组合对象中每个组件的数量总和：

```
public int getMachineCount() {  
    int count = 0;  
    Iterator i = components.iterator();  
    while (i.hasNext()) {  
        MachineComponent mc = (MachineComponent) i.next();  
        count += mc.getMachineCount();  
    }  
    return count;  
}
```

只要 `MachineComponent` 对象是树形结构，这些方法都是正确的。然而，这种假设并非总是成立，特别是当用户可以编辑这一组合模型时，可能会突然变成非树形结构。考虑如下的 `Oozinoz` 实例。

`Oozinoz` 的焰火工程师使用图形界面应用程序来记录和更新工厂中的机器组合对象模型。某天，他们报告称工厂中现有机器的数量有误。我们可以在 `OozinozFactory` 类中定义 `plant()` 方法，重新创建对象模型。

```
public static MachineComposite plant() {  
    MachineComposite plant = new MachineComposite(100);  
    MachineComposite bay = new MachineComposite(101);  
    Machine mixer = new Mixer(102);  
    Machine press = new StarPress(103);  
    Machine assembler = new ShellAssembler(104);  
    bay.add(mixer);
```

```
bay.add(press);
bay.add(assembler);
plant.add(mixer);
plant.add(bay);
return plant;
}
```

这正是图 5.4 中创建 `plant` 对象的代码实现。

挑战 5.4

下面的程序将会输出什么？

```
package app.composite;
import com.oozinoz.machine.*;

public class ShowPlant {
    public static void main(String[] args) {
        MachineComponent c = OozinozFactory.plant();
        System.out.println(
            "Number of machines: " + c.getMachineCount());
    }
}
```

答案参见第 306 页

Oozinoz 的工程师们通常会使用 GUI 应用程序为工厂中的机器进行对象建模，它可以在第二次添加节点时，检查该节点是否已经在组件树中存在。一个简单的实现是维护一个已存在节点的集合。然而，你可能无法控制一个组合对象的构成。因此需要编写一个 `isTree()` 方法去检查组合对象是否为树形结构。

倘若在遍历对象模型之后，没有哪个节点被访问两次，我们就认为这个对象模型是一棵树。可以在抽象类 `MachineComponent` 中实现 `isTree()` 方法，以便它可以维护访问过的节点列表。`MachineComponent` 类可以将带参数的 `isTree(set:Set)` 方法作为抽象方法。图 5.5 展示了 `isTree()` 方法的实现方式。

在 `MachineComponent` 类的实现中，`isTree()` 方法调用了自身的抽象方法

`isTree(s:Set)`, 代码如下:

```
public boolean eanisTree() {
    return isTree(new HashSet());
}

protected abstract boolean isTree(Set s);
```

该方法使用了 Java 类库中的 `Set` 类。

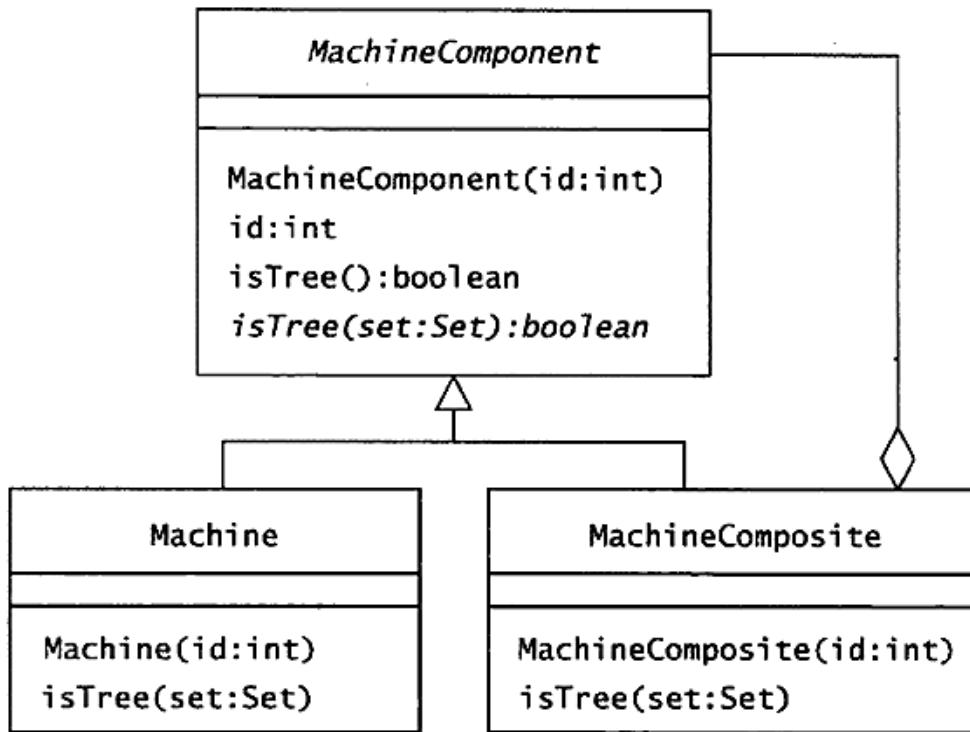


图 5.5 `isTree()`方法可以检测该组合模型是否为树形结构

`Machine` 和 `MachineComposite` 类都必须实现 `isTree(s:Set)` 方法, `Machine` 类的 `isTree()` 实现非常简单, 反映了单个机器始终是树形结构:

```
protected boolean isTree(Set visited) {
    visited.add(this);
    return true;
}
```

`MachineComposite` 实现的 `isTree()` 必须把调用它的对象加到 `visited` 列表中, 并且迭代调用组合对象中各个组件的 `isTree()` 方法。如果任何一个被访问过的组件返回 `false`, 或者任何一个组件本身并不是树形结构, 该方法就会返回 `false`, 否则返回 `true`。

挑战 5.5

写出 `MachineComposite.isTree(Set visited)` 的代码。

答案参见第 307 页

只需多加注意，防止 `isTree()` 返回 `false`，就可以保证对象模型是一个树状模型。另一方面，也可能需要允许组合对象不是树形结构，尤其是当正在建模的问题领域包含环的时候。

含有环的合成模式

挑战 5.4 中提到的非树形合成模式是一个失误，原因在于用户将 `machine` 同时当做 `plant` 和 `bay` 的一部分。若是物理对象，可能并不允许它被包含在多个对象中。然而，问题域可以包括多个非物理元素，此时环状包含关系是有意义的。这种场景在对工作流建模时经常发生。

考虑图 5.6 描述的礼花弹的构造过程。我们点燃位于底部提供动力的黑火药，形成一股推动力，把焰火弹发射出去。当焰火弹升空后，它的二级引线点燃，最终燃至内核，然后内核燃烧并且爆炸，形成焰火的效果。

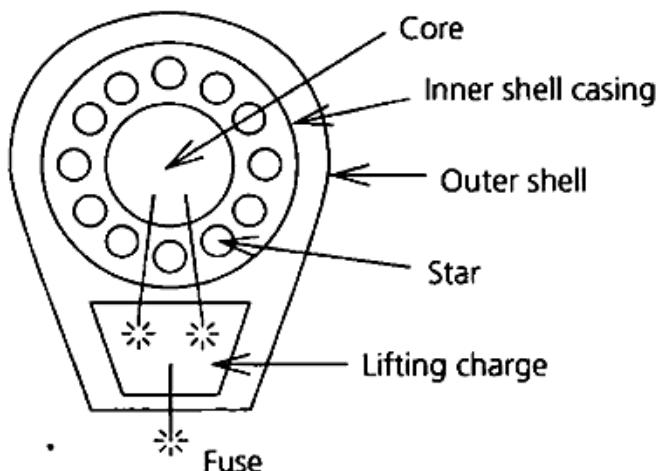


图 5.6 一个焰火弹使用两份火药：一份用于开始升空时提供动力，另一份用于在焰火弹到达顶点时爆炸并产生焰火效果

制作焰火弹的工艺流程是：制作内层的焰火弹，进行检测，然后确定它是否需要返工或者完成组装。

内层焰火弹的制作流程为：操作员使用焰火弹装配器把火药放入一个半球状的盒子里，插入黑火药核，然后在核的顶部放入更多的火药，再把它密封在其他半球状的盒子中。

检察员验证内层焰火弹是否安全，质量是否符合标准。如果不是，操作员就需要拆解内层焰火弹，重新制作。如果内层焰火弹通过检查，操作员将使用引线将内层焰火弹与提供上升动力的部分联系起来。最后，操作员手工给焰火弹包上外壳完成组装。

和之前的机器实现合成模式一样，Oozinoz 工程师用一个 GUI 程序来描述组合的过程。图 5.7 给出了支持建模过程的类图结构。

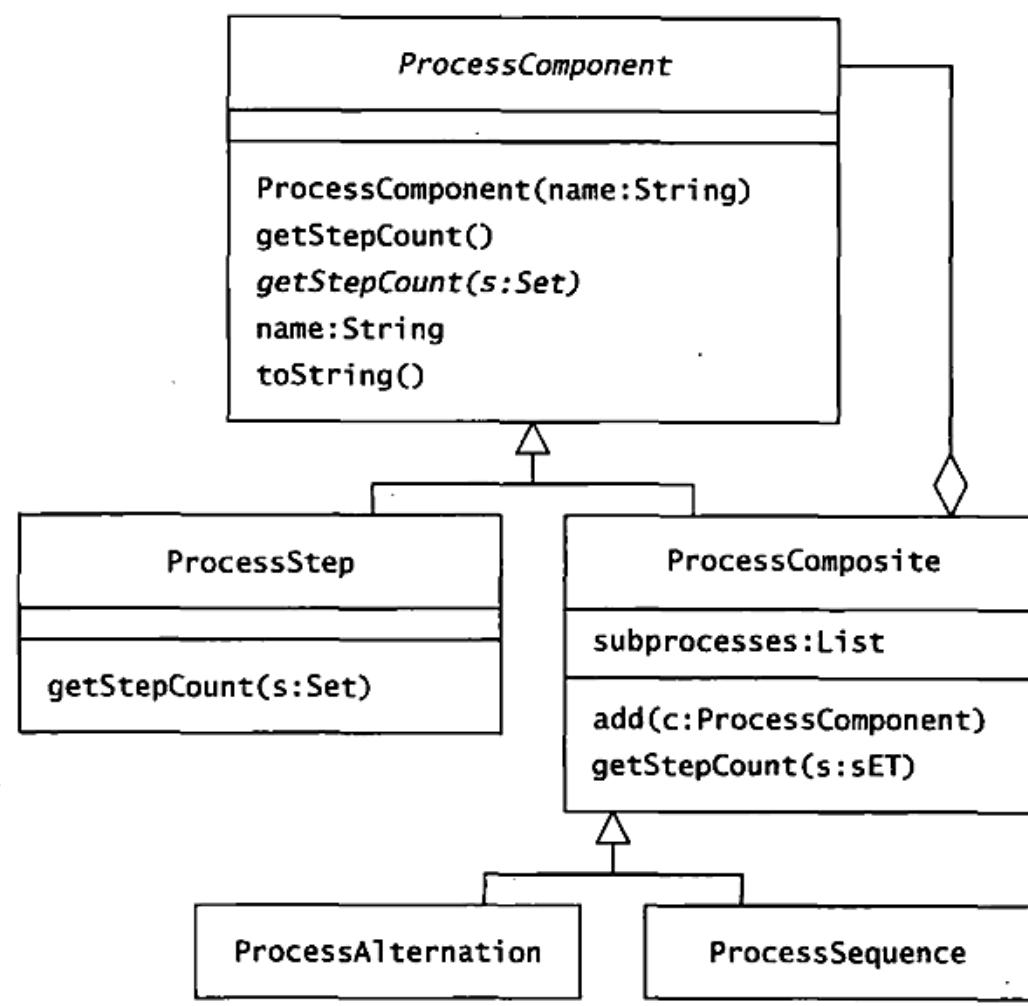


图 5.7 制作焰火弹的流程，包括一些可相互替代的或者有顺序的步骤

图 5.8 展示了代表制作焰火弹工艺流程的对象。`make` 流程按顺序包含 `buildInnershell` 步骤、`inspect` 步骤以及 `reworkOrFinish` 子步骤。`reworkOrFinish` 子步骤含有两个可相互替换的路径。`make` 流程的最后可能是 `disassemble` 步骤或者仅仅是 `finish` 步骤。

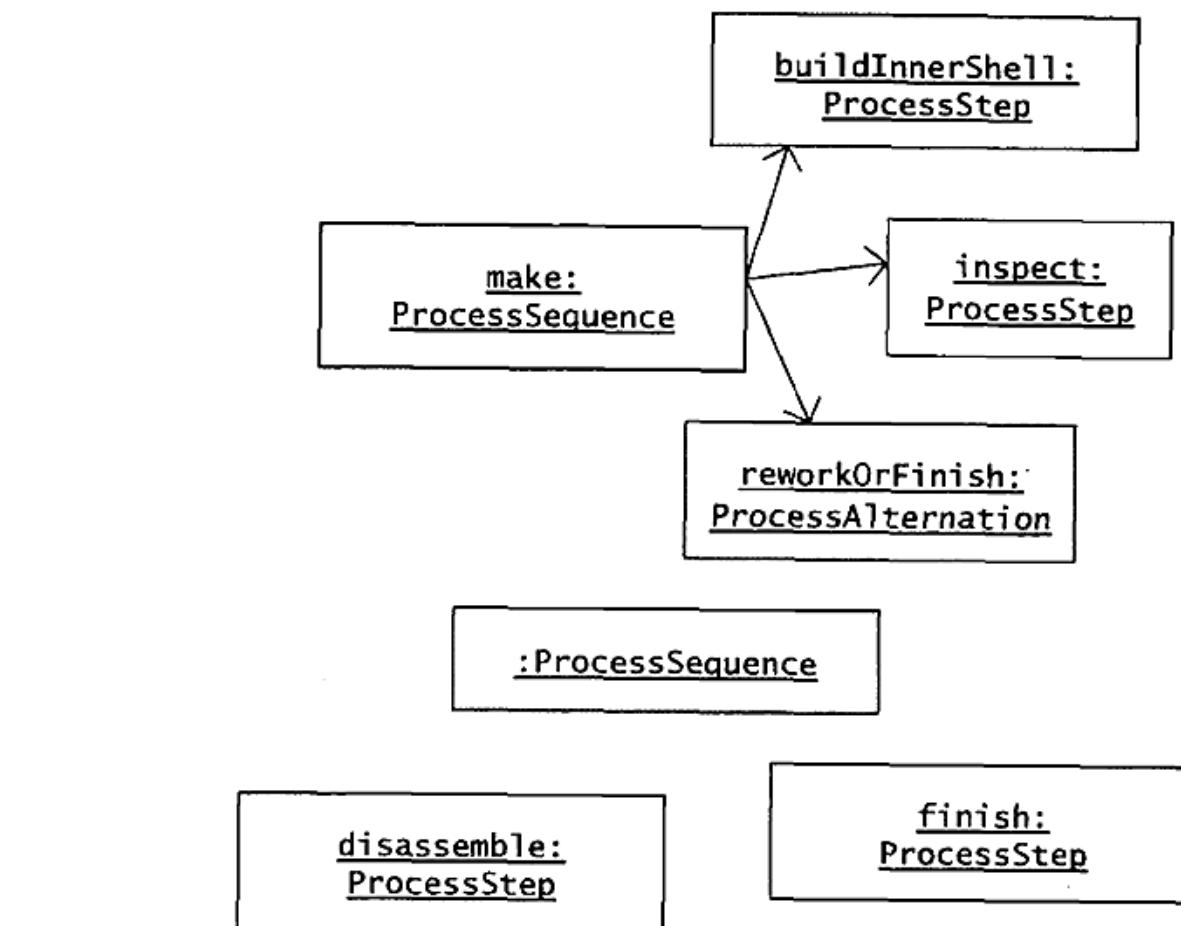


图 5.8 当你完成这幅图后，将会展示出 Oozinoz 在制作焰火弹流程时的对象模型

挑战 5.6

图 5.8 展示了焰火弹组装流程模型中的对象。一幅完整的对象图应该包含各个对象间的相互引用关系。例如，该图已经包含了 `make` 对象包含的引用关系。请补充图中缺失的引用关系。

答案参见第 307 页

位于 `ProcessComponent` 层级的 `getStepCount()` 操作负责统计工艺流程中单个处理步骤的数量。注意，这个数量并非流程的深度，而是图中叶子节点处理步骤的数量。`getStepCount()` 方法必须确保每个步骤的统计只有一次，如果流程中包含了环，应避免出现死循环。`ProcessComponent` 类实现了 `getStepCount()` 方法，以便通过该方法获得访问过的节点集合。

```

public int getStepCount() {
    return getStepCount(new HashSet());
}
  
```

```
public abstract int getStepCount(Set visited);
```

ProcessComposite 类在实现 `getStepCount()` 方法时应避免再次访问之前已经访问过的节点：

```
public int getStepCount(Set visited) {  
    visited.add(getName());  
    int count = 0;  
    for (int i = 0; i < subprocesses.size(); i++) {  
        ProcessComponent pc =  
            (ProcessComponent) subprocesses.get(i);  
        if (!visited.contains(pc.getName()))  
            count += pc.getStepCount(visited);  
    }  
    return count;  
}
```

ProcessStep 类中的 `getStepCount()` 方法的实现很简单：

```
public int getStepCount(Set visited) {  
    visited.add(name);  
    return 1;  
}
```

`com.oozinoz.process` 包提供了一个 `ShellProcess` 类。该类定义了图 5.8 描述的返回 `make` 对象的 `make()` 方法。`com.oozinoz.testing` 包有一个 `ProcessTest` 类，该类提供各类流程图的自动化测试。例如，`ProcessTest` 类包含一个方法，该方法测试 `getStepCount()` 操作能否正确返回具有环状结构的 `make` 流程中的步骤数。

```
public void testShell() {  
    assertEquals(4, ShellProcess.make().getStepCount());  
}
```

这些测试在 JUnit 测试框架下运行并通过。若要了解更多关于 JUnit 的信息，请访问 www.junit.org。

环的影响

许多组合对象上的操作，例如统计叶子节点的数量，在组合对象不是树形结构时仍有意义。

通常，非树形与树形结构组合对象的唯一区别在于，你必须小心谨慎地避免重复操作相同的节点。然而，如果组合对象包含了环，则某些操作是没有意义的。例如，我们不能通过算法决定在 Oozinoz 中制作焰火弹所需要的最大步骤数。在任何一个包含环的组合对象中，依赖路径长度的操作都是无意义的。因此，尽管我们可以讨论树的高度——从根到叶子的最长距离，但在一个有环图中却不存在。

倘若允许合成模式不是树形结构，带来的另一个后果是不能假设每个节点有且仅有一个父节点，而是可能具有多个父节点。例如，图 5.8 的流程模型可能有多个组合对象引用 `inspect` 步骤，则 `inspect` 对象就将存在多个父节点。一个节点拥有多个父节点本身没有任何问题，但在建模和编码时，必须考虑这个问题。

小结

合成模式包含两个相互关联的重要概念。一个概念是一个组合对象可以包含单对象或者其他组合对象。与之相关的另一个概念就是组合对象与单对象共享同一个接口。将这些概念应用于对象建模上，就可以在组合对象与单对象上创建抽象类或 Java 接口，以此定义公共行为。

在对组合对象建模时，通常需要给组合节点引入递归定义。倘若存在递归定义，编写代码时，需要注意防止死循环。为避免这一问题，可以确保组合对象都是树形结构。此外，虽然允许环出现在合成模式中，但必须修改算法避免可能出现的死循环。

第6章

桥接（Bridge）模式

桥接模式的英文为 Bridge，桥接模式关注抽象的设计。抽象是指包含了一组抽象方法的类，这些抽象方法可能包含多个实现。

实现抽象的一般做法是创建类的层次结构，该层次的顶部是一个包含抽象方法的抽象类；该类的每个子类都提供这些抽象方法的不同实现。但是，当需要对该层次进行子类化时，这一做法就存在不足了。

你可以创建一个桥，然后把这些抽象方法移到接口中。这样，抽象就将依赖于接口的实现。

桥接模式的意图是将抽象与抽象方法的实现相互分离来实现解耦，以便二者可以相互独立地变化。

常规抽象：桥接模式的一种方法

从某种意义上说，如果认为每个类都是实体模型的近似化、理想化或者简单化，那么几乎每个类都是一个抽象。但在讨论桥接模式时，我们用“抽象”这个词专指依赖一组抽象操作的类。

假设 Oozinoz 公司有一些机器控制类，这些类控制物理机器来生产焰火弹。它们操作机器的方法各不相同。你可能需要创建一些抽象方法，使得所有机器达到同样的效果。图 6.1 展示

了 com.oozinoz.controller 包中当前的控制类。

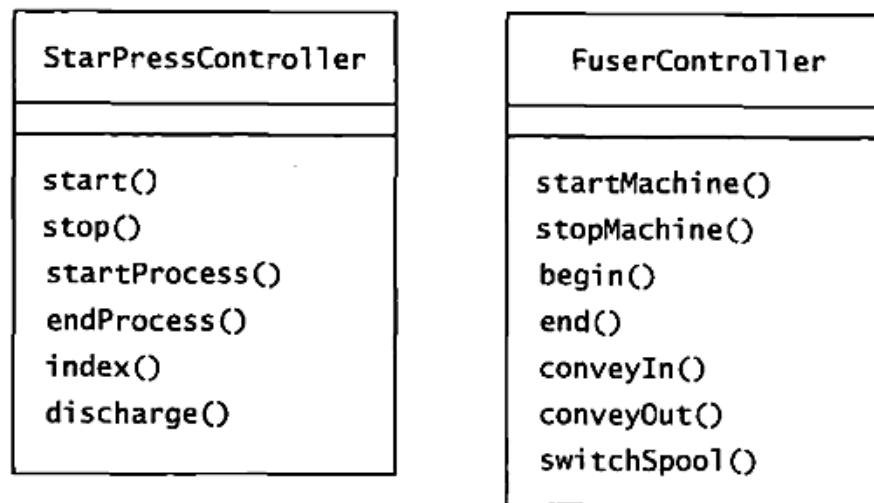


图 6.1 两个类展示了相似的方法，你可以将它们抽象到操作机器的通用模型中

在图 6.1 中，尽管 `StarPressController` 类将这两个方法命名为 `start()` 和 `stop()`，而 `FuserController` 类则将它们命名为 `startMachine()` 和 `stopMachine()`，但它们表达的意义都是启动和停止机器。两个控制类都包含了处理箱子的操作：将箱子移到加工区的方法 (`index()` 和 `conveyIn()`)，开始处理和结束处理的方法，以及移除箱子 (`discharge()` 和 `conveyout()`) 的方法。`FuserController` 类还包含一个用于切换使用备用引线线圈的方法。

现在假设你要创建一个 `shutdown()` 方法，在两台机器上执行相同的步骤，以确保有序地关机。为了简化 `shutdown()` 方法的编写，需要为通用方法的命名标准化，例如 `startMachine()`、`stopMachine()`、`startProcess()`、`stopProcess()`、`conveyIn()` 和 `conveyOut()`。但是，我们不能改变控制类，因为其中的某些方法是由机器自身提供的。

挑战 6.1

请阐述如何运用一个设计模式，通过一个公共接口来控制不同的机器。

答案参见第 308 页

图 6.2 展示了一个含有子类的 `MachineManager` 抽象类，它可以转发机器控制指令，适配它们到 `FuserController` 和 `StarPressController` 支持的方法上。

如果机器的控制器包含一些只针对特定机器类型的操作，则没有问题。例如，`FuserManager` 类包含一个 `switchSpool()` 方法（没有在图 6.2 中展现出来），该方法会转发到

FuserController 对象的 switchSpool()方法中。

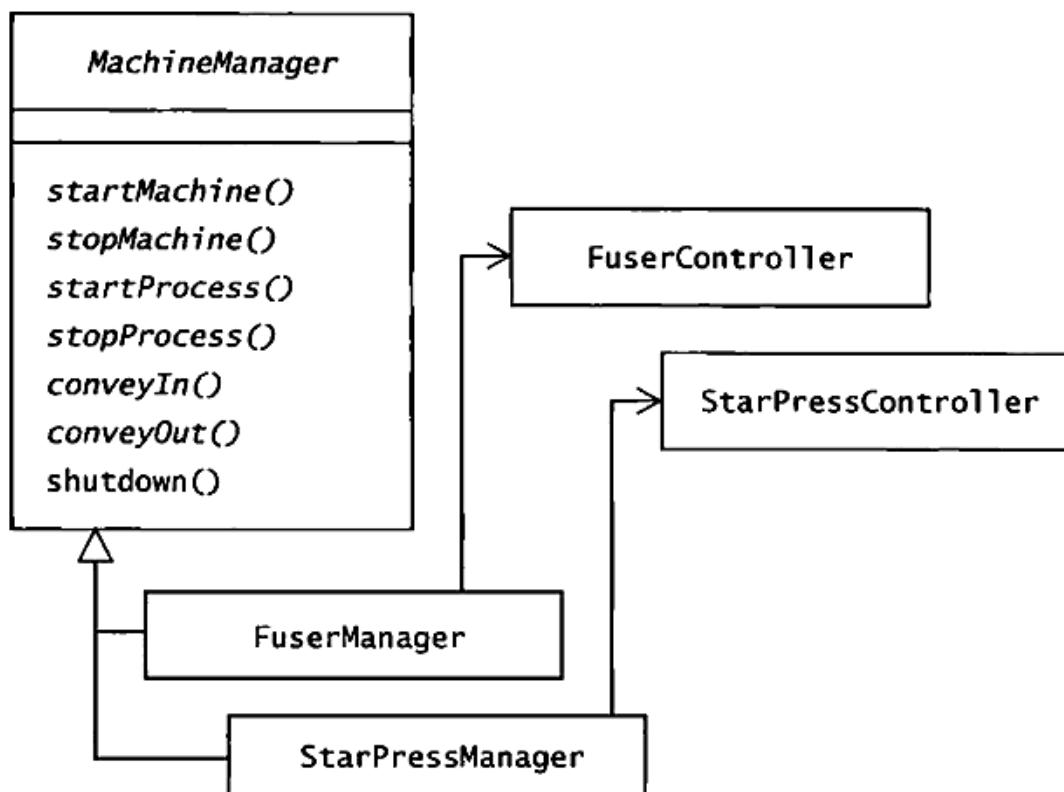


图 6.2 FuserManager 和 StarPressManager 类将调用传递给 FuserController 和 StarPressController 对象中相应的方法，以此实现 MachineManager 中的抽象方法

挑战 6.2

请写出 MachineManager 类的 shutdown()方法，该方法会停止处理过程。取出正在处理中的箱子，并且停止机器。

答案参见第 308 页

虽然 MachineManager 类的 shutdown()方法是具体方法，而非抽象方法，然而，我们仍可以说它是抽象的，因为关闭设备的步骤已经被泛化。

从抽象到桥接模式

根据不同设备的排列定义 MachineManager 的层次结构，则每个机器类型都需要

`MachineManager` 类的不同子类。如果需要增加新的排列方式，则类的层次结构会发生什么变化呢？例如，假设直接在机器上工作，每当机器完成一个步骤后都会提供一个反馈。与之对应，就需要创建 `MachineManager` 类的握手子类，允许设置我们与机器交互的参数，比如设置一个超时时间。然而，我们仍然需要不同的 `MachineManager` 对象管理火药球的压入与引线。如果不修改 `MachineManager` 类的层次结构，新的类层次结构就如图 6.3 所示。

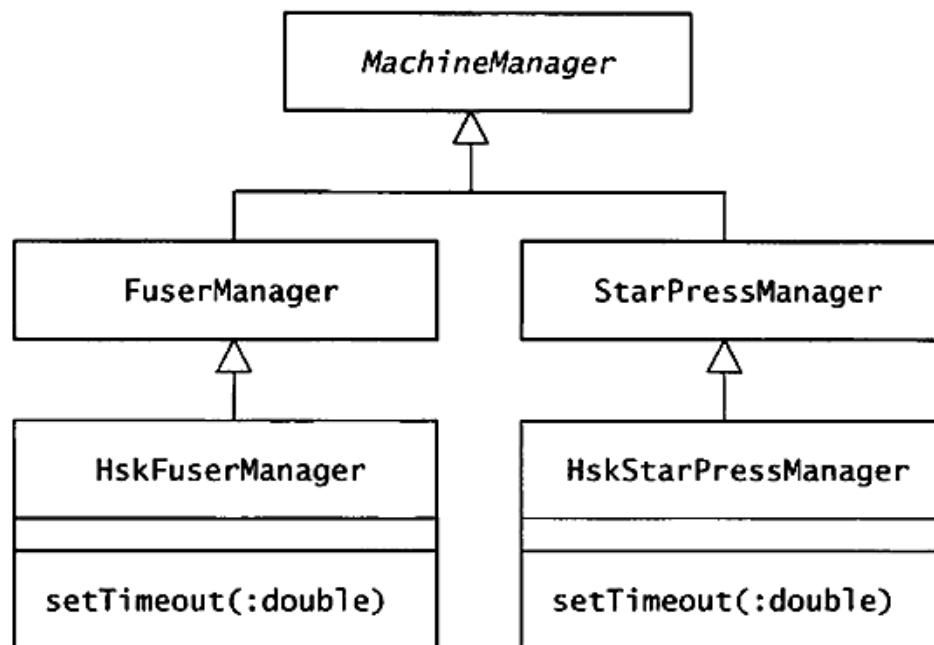


图 6.3 握手(Hsk)子类增加了一个用于表示等待实际机器反馈需要多长时间的参数

图 6.3 中的类层次结构有两种不同的排列：一种依据机器类型，另一种依据是否支持握手协议^{译注1}。这种对偶原则(dual principle)会带来诸多问题。特别如 `setTimeout()` 这样的方法可能会重复出现在两个地方，却不能将该方法提升到超类中，因为它的超类并不支持握手协议。

支持握手协议的类通常无法共享代码，因为并不存在支持握手协议的超类。随着我们向该层次结构中添加更多的类，问题会变得更加糟糕。假如，最终需要 5 台机器的控制器，从而要求修改 `setTimeout()` 方法，就必须同时更改 5 个地方。

此时，就是桥接模式粉墨登场的时候了。通过将 `MachineManager` 的抽象方法分离到单独的类层次中，可以完成抽象方法的实现与抽象的解耦。`MachineManager` 类依然是抽象类，调

译注1：即常说的一个对象，承担了两个及以上具有不同变化方向的职责。它事实上违背了单一职责原则，即一个对象只能有一个引起它变化的原因。此时，我们常常需要分离职责，并对分离出去的职责进行抽象，再以组合的方式与原有对象进行协作。

用其方法则取决于我们是要控制火药球的压入还是引线。

将抽象从抽象方法的实现中分离出来，使得两个类层次结构能够独立变化。在不影响 `MachineManager` 类层次结构的情况下，我们能够添加新机器；也可以在不改变机器控制器的情况下，对 `MachineManager` 的类层次结构进行扩展。图 6.4 展示了我们所期望的分离效果。

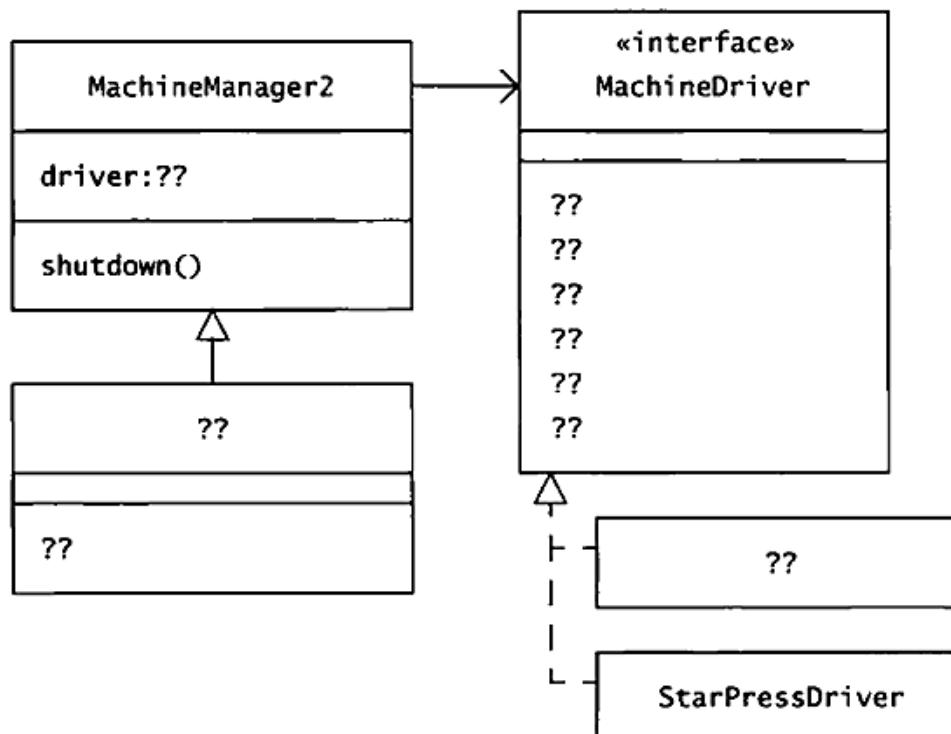


图 6.4 完成此图后，将展现 `MachineManager` 抽象类与其抽象方法的实现相分离

新的设计目标就是将 `MachineManager` 类层次结构与其抽象方法的实现分离。

挑战 6.3

图 6.4 描述了将 `MachineManager` 类层次结构重构为桥接模式，请填写缺失的部分。

答案参见第 308 页

注意在图 6.4 中，尽管 `MachineManager2` 类是抽象类，但是它却是具体功能的体现。`MachineManager2` 类所依赖的抽象方法现在被放到 `MachineDriver` 接口中^{#*2}。该接口的名称

^{#*2}译注2：这一说法似有不妥。直观上看，`MachineManager2` 仍然体现了抽象层面的接口，相反，相关的 `Driver` 类则代表了实现，而 `MachineDriver` 则是对实现的抽象。

就代表它成为了驱动器 (driver)，用以将 `MachineManager` 的请求转到指定的机器上。驱动器 (driver) 对象能够根据定义好的接口操作计算机系统或外部设备。驱动器是最为常见的桥接模式的实践。

使用桥接模式的驱动器

驱动器是抽象的。应用程序的执行结果取决于当前执行的是哪个驱动器。每个驱动器都是适配器 (Adapter) 模式的一个实例。它们通过调用具有不同接口的类提供的服务，向客户提供期望的接口。使用驱动器的整体设计就是一个桥接模式的实例。这样的设计将应用程序的开发从驱动器的开发中分离出来，驱动器实现了应用程序所依赖的抽象方法。

基于驱动器的设计迫使你为被驱动出来的机器或系统设计出一个通用、抽象的模型。这样做的好处是，抽象端的代码可以被应用到任何可能被执行的驱动器中。为驱动器定义一系列通用方法可能会引发一些问题，例如某个驱动实体支持的行为可能会消失。重新来看图 6.1，`fuser` 控制器包含了 `switchSpool()` 方法。再看修改后的设计图 6.4，这一方法到哪里去了？答案是我们将它抽象出去了。可以在新的 `FuserDriver` 类中定义 `switchSpool()` 方法。然而，这可能导致抽象端的代码必须检查当前驱动器对象是否为 `FuserDriver` 的实例。

为了避免丢失 `switchSpool()` 方法，我们必须让每个驱动器都实现此方法，并让某些驱动程序简单地忽略该调用^{译注3}。当选择一个驱动器支持的方法的抽象模型时，常常会面临这样的抉择。我们既可以包含一些驱动程序不支持的方法，也可以将这些方法排除在外。后者可能缩小驱动程序抽象的范围，或者强制要求抽象包含一些特殊情况下的代码。

数据库驱动

数据库访问对驱动程序的使用，我们已经司空见惯。在 Java 程序中，数据库连接依赖于 JDBC，*JDBC™ API Tutorial and Reference (2/e)*（由 White 等人在 1999 年编写）很好地解释了

译注3：只有如此才能让所有驱动器保持统一的抽象。

如何运用 JDBC 的资源。简而言之，JDBC 是执行结构化 SQL 语句的应用程序接口 (API)。该接口的实现类就是 JDBC 驱动程序，依赖于该接口的应用程序是抽象的，它可以工作在任何提供了 JDBC 驱动的数据库中。JDBC 架构将抽象与实现相互分离，因而两者可以独立地变化：这是一个非常好的桥接模式案例。

要使用 JDBC 驱动程序，首先需要加载它，连接数据库，然后创建一个 `Statement` 对象：

```
Class.forName(driverName);
Connection c = DriverManager.getConnection(url, user, pwd);
Statement stmt = c.createStatement();
```

有关 `DriverManager` 类如何工作的内容不在我们讨论的范围之列。这里，只需知道 `stmt` 是一个 `Statement` 对象，它能够执行 SQL 查询并返回结果集。

```
ResultSet result = stmt.executeQuery(
    "SELECT name, apogee FROM firework");
while (result.next()) {
    String name = result.getString("name");
    int apogee = result.getInt("apogee");
    System.out.println(name + ", " + apogee);
}
```

挑战 6.4

图 6.5 是一个 UML 顺序图，它展现了一个典型的 JDBC 应用程序中的消息流。
请在图中填充缺失的类型名称以及消息名称。

答案参见第 309 页

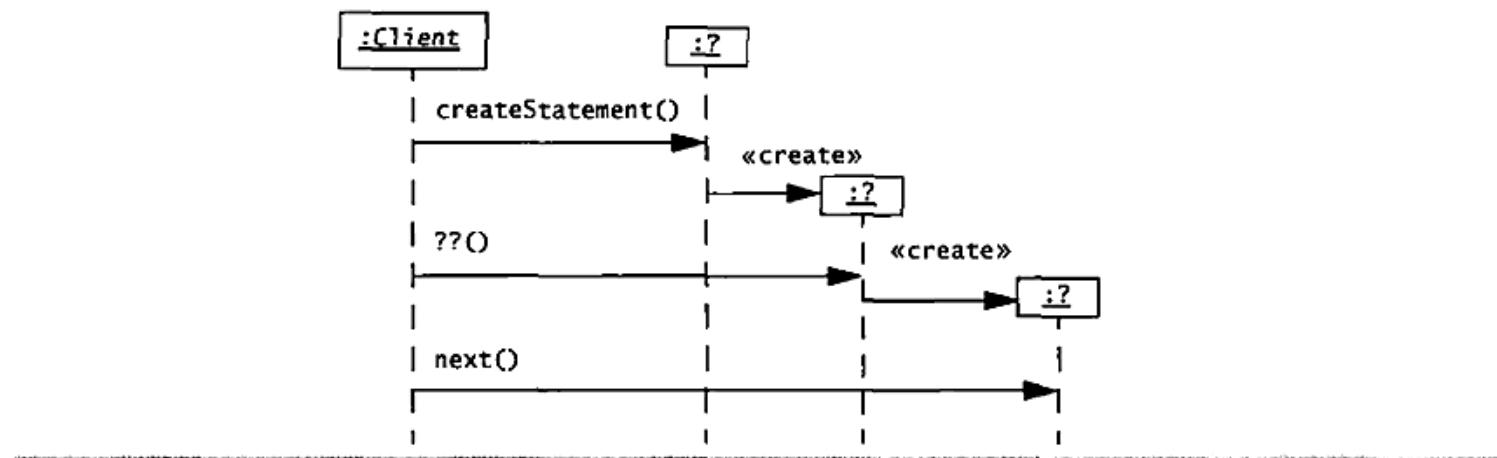


图 6.5 该图展现了 JDBC 应用程序中最为典型的消息流

挑战 6.5

假设在 Oozinoz 公司，当前仅有 SQL Server 数据库。请讨论是否应该为数据库做一套通用或专用的数据访问器或适配器。

答案参见第 309 页

JDBC 架构清晰地区分了驱动程序员与应用程序员两种角色。我们可以为抽象的超类建立驱动程序子类，每个子类负责驱动不同的子系统。在这种情形下，一旦需要更好的灵活性，就可以使用桥接模式。

小结

抽象是指依赖于抽象方法的类。最简单的抽象例子是一个抽象的类层次结构，在该结构下，超类的具体方法依赖于其他抽象方法。如果想将最初的类层次结构分解为另一种类层次结构，就必须将这些抽象方法移到另一个类层次结构中。此时，就应该运用桥接模式，完成抽象方法实现与抽象的分离。

驱动程序是最为常见的桥接模式范例，例如数据库驱动。数据库驱动程序很好地体现了在运用桥接模式时，需要对结构进行权衡的本质。一个驱动程序可能会调用一些实现者不支持的方法。另一方面，驱动程序可能会忽略一些用于特定数据库的方法。这将要求你写一些针对实现而非抽象的代码。抽象是否优于具体，或许并无定论，但有意识地去做出这些决定却是极为重要的。



第2部分

职责型模式

第7章

职责型模式介绍

对象的职责好似 Oozinoz 公司呼叫中心所代表的职责。当要从 Oozinoz 公司购买焰火弹时，你需要联系公司代表或者代理机构。他/她需要完成一些既定任务，通常，会将这些任务委托给其他系统或者其他去完成。有时，这些代表会将这些任务请求委托给唯一的中心机构，该机构会协调各种请求，或者将任务请求抛给职责链（chain of responsibility）去完成。

就像呼叫中心代表一样，普通的对象也需要一些独立操作的信息和方法。然而，有时你却需要将对象从一般的独立性操作中分离出来，以便集中职责。很多设计模式都能满足这一需求。有的模式则通过引入对象来封装这些请求，并将该对象从依赖于它的其他对象中分离出来。面向职责的模式提供了用于集中、加强以及限制普通对象责任的技术。

常规的职责型模式

你或许对那种设计良好的类抱有强烈的意识：属性与职责应该统一在一起，不过这种理解你却只可意会不可言传。

挑战 7.1

图 7.1 所示的类结构图中至少包含 10 处职责分配的问题。请尽可能地圈出你所发现的问题，并写出其中 4 处错误的原因。

答案参见第 310 页

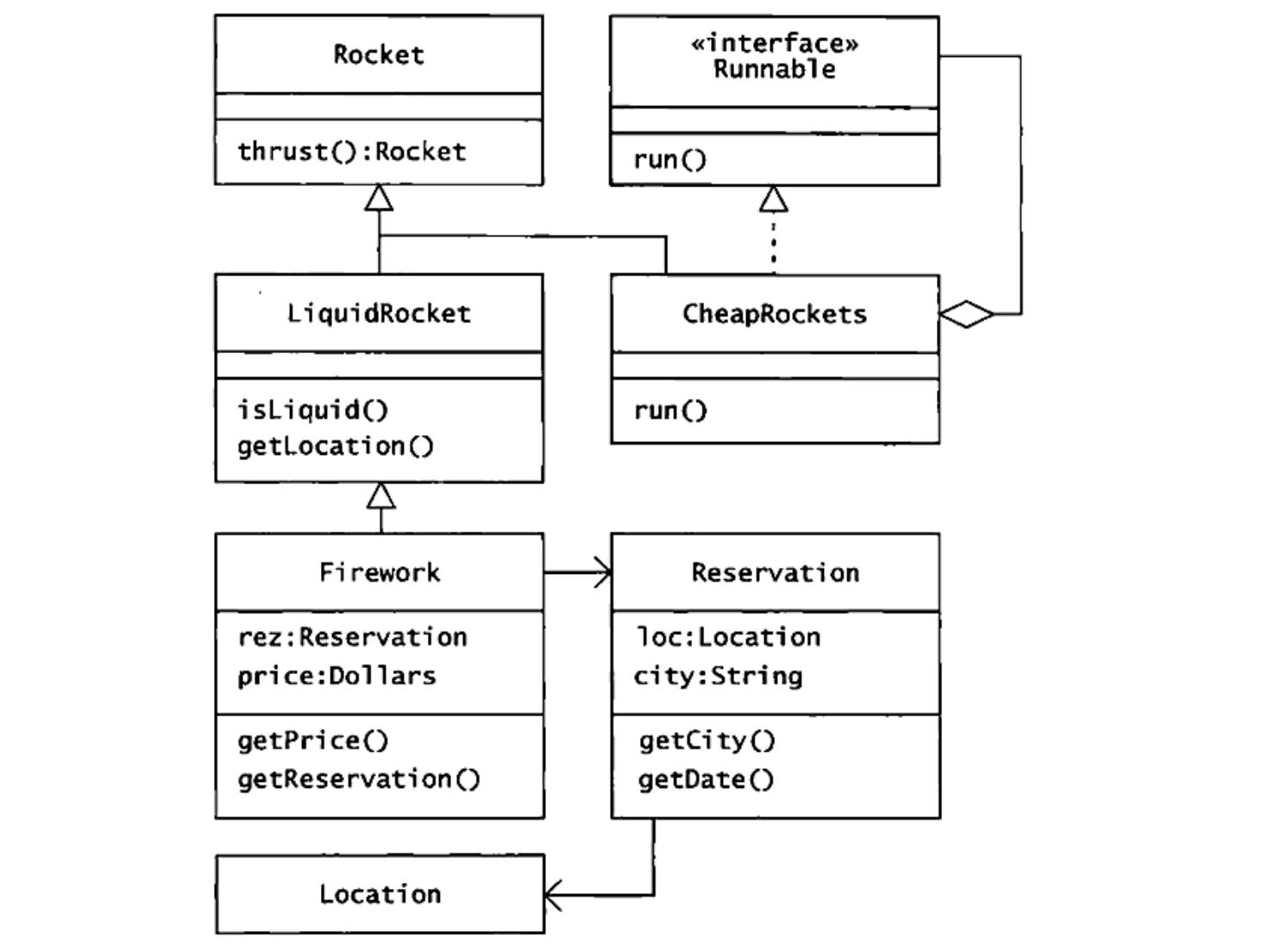


图 7.1 这幅图有什么错误

图 7.1 中所有奇怪的地方，都可能会引发你对如何正确进行对象建模的思考。当你开始定义像“类”一样的术语时，这种思考非常好。术语的价值随着人们交流的增加而增加，但如果只是为了定义术语而去定义术语，并会引起冲突，就没有必要定义术语。基于这一观念，让我们迎接下面这个困难的挑战。

挑战 7.2

请定义一个高效且易于使用的类的特性。

答案参见第 310 页

一个易于使用的类的特征在于，它的方法名是有意义的，并能准确地表述方法要做的事情。然而通常所见的是，所要调用的方法，其名称并没有准确表达它内部实现的信息。

挑战 7.3

请举出一个方法名未能准确表示内部实现的例子，并给出充分的理由。

答案参见第 311 页

在面向对象的系统中，职责的合理分配所建立的原则，似乎促进了计算机科学更进一步的成熟。在一个系统中，如果每个类和方法都清晰定义了它的职责，并能正确使用它们，这个系统就是迄今为止我们所能见到最为健壮的系统。

根据可见性控制职责

我们习惯去说类和方法承担着各种各样的职责。事实上，这通常意味着你有责任提供健壮的设计，并让代码承担合适的功能。幸运的是，Java 语言分担了一部分责任。我们可以限制类、字段和方法的可见性，从而去限制其他开发人员对你开发的代码的调用。可见性向读者展示了该如何暴露类的部分内容。表 7.1 给出了访问修饰符的非正式定义。

表 7.1 访问修饰符的非正式定义

| 访 问 | 非正式定义 |
|------------------------|----------------------|
| <code>public</code> | 访问不受限制 |
| <code>(none)</code> | 只允许包内的访问 |
| <code>protected</code> | 只允许所在包的类，或者继承该类的子类访问 |
| <code>private</code> | 只允许包含它的类访问 |

在实践过程中，会有一些微妙的问题出现，这就需要我们更多地思考这些修饰符的正式定义。一个需要思考的问题就是，可见性是否影响了类或者对象。

挑战 7.4

一个对象是否能引用该类其他实例的私有成员？特别的，如下代码能通过编译吗？

```
public class Firework {  
    private double weight = 0;  
    // ...  
    private double compare(Firework f) {  
        return weight - f.weight;  
    }  
}
```

答案参见第 311 页

修饰符的可见性能帮助我们通过限制提供给其他开发者的服务，来限制你的职责。例如，如果不希望开发者操作类的字段，就可以把该字段设置为 `private`。另一方面，如果想给将来的开发者提供更多弹性，则可以把字段设置为 `protected`，尽管这样会有子类与父类耦合过紧的风险。我们可以做出刻意为之的决策，甚至建立一套策略，既能够支持未来的可扩展性，又能通过限制访问来约束当前的职责范围。

小结

作为一名 Java 开发者，你有责任去创建拥有一组逻辑相关的属性与行为的类。设计良好的类是一门艺术，它们所拥有的基本特征更需要我们去总结。对于自己编写的类，有责任确保方法名是方法实现的准确表达。可以通过合理运用访问修饰符去限制相应的职责，但同时还需要权衡代码的安全性和灵活性。

超越普通职责

无论一个类如何限制它的成员，面向对象开发通常都会将职责分散到各个独立的对象中。换句话说，面向对象开发促进了封装，封装是指对象基于自己的数据进行操作。

职责分离是一种规范的做法，但一些设计模式却反对这种规范，并且将职责转移到中间对象或者中心对象。例如，单例模式将职责集中到一个单独的对象中，并提供对该对象的全局访

问。识别单例和其他模式意图的一种办法，就是将它们作为普通职责分离原则的反例。表 7.2 列出了不同场景下适用的模式。

表 7.2 不同场景下适用的模式

| 如果你的意图是 | 适用的模式 |
|--------------------------------|-------|
| 将责任集中到某个类的单个实例中 | 单例模式 |
| 将对象从依赖于它的对象中解耦 | 观察者模式 |
| 将职责集中在某个类，该类可以监督其他对象的交互 | 调停者模式 |
| 让一个对象扮演其他对象的行为 | 代理模式 |
| 允许将请求传递给职责链的其他对象，直到这个请求被某个对象处理 | 职责链模式 |
| 将共享的、细粒度的对象职责集中管理 | 享元模式 |

每种设计模式的意图都是为了解决特定场景下的问题。如果需要违背通常的原则，尽可能早地分离职责，那么就是面向职责模式粉墨登场的时候了。

第8章

单例（Singleton）模式

通常，对象通过在自身属性上执行任务来承担自己的职责，除了需要维护自身的一致性外，无须承担其他任何职责。然而，仍有一些对象承担了更多的职责，例如对真实世界的实体进行建模、协调工作或者对整个系统的状态进行建模。当系统的其他对象都依赖于特殊对象所承担的职责时，我们需要通过某种方式找到这个承担责任的对象。例如，可能需要找到一个代表指定机器的对象，或者是从数据库获取数据来创建自身的客户对象，又或者是初始化系统内存用以恢复的对象。

在某些场景，你需要找到一个承担责任的对象，并且这个对象是它所属类的唯一实例。例如，焰火工厂可能需要唯一的一个 `Factory` 对象。此时，可以使用单例模式，单例模式的英文为 `Singleton`。

单例模式的意图是为了确保一个类有且仅有一个实例，并为它提供一个全局访问点。

单例模式机制

单例模式的机制比其意图更加容易记忆。解释如何保证一个类有且仅有一个实例，要比解释为何需要这种限制要简单得多。正如 *Design Patterns* 一书就把单例模式归类为“创建型”模式。当然，理解模式的方式是没有限制的，只要能帮助你记忆、识别和运用这些模式。对于单例模式，它的意图说明了该单例对象承担了其他对象所要依赖的职责。

创建一个担当着独一无二角色的对象，有多种方式。但是，不管你如何创建一个单例对象，

都必须确保其他开发人员不能创建该单例对象的新的实例。

挑战 8.1

怎样才能阻止其他开发人员不能创建类的新实例？

答案参见第 312 页

设计一个单例类时，需要确定何时实例化该类的单例对象。一种做法是创建这个类的实例，并将它作为该类的静态成员变量。例如，`SystemStartup` 类可能包括这一行：

```
private static Factory factory = new Factory();
```

这个类通过一个公共的 `getFactory()` 静态方法获得该类的唯一实例。

如果不希望提前创建单例实例，还可以在第一次需要该实例时，延迟初始化它。例如，`SystemStartup` 类可能采用如下方式获取单个实例：

```
public static Factory getFactory() {  
    if (factory == null)  
        factory = new Factory();  
    // ...  
    return factory;  
}
```

挑战 8.2

为什么要决定延迟初始化一个单例实例，而不是在声明时就初始化？

答案参见第 312 页

无论哪种场景，单例模式都建议提供一个公共的静态方法去访问单例对象。如果该方法创建了一个对象，它就要保证只有一个实例可以被创建。

单例和线程

如果想在一个多线程环境下延迟初始化一个单例模型，必须小心谨慎地避免多个线程同时

初始化该单例对象。在多线程环境下，无法保证在其他线程开始执行该方法时，当前线程已经完整地执行完该方法。这可能出现两个线程同时初始化一个单例对象的情况。假设第一个线程发现该单例对象为 null，紧接着第二个线程运行，也会发现该单例对象为 null。然后两个线程都会对该单例对象进行初始化。为了避免这种竞争，需要用锁机制去协调不同线程对同一方法的执行。

Java 支持多线程开发。它可以为每个对象提供一个锁，用于表示对象是否已经被线程所占用。为确保仅有一个线程初始化单例对象，可以通过对适当的对象进行加锁来同步初始化。其他需要互斥访问单例对象的方法，也可以通过相同的锁机制进行同步。若要了解在并发模式下如何进行面向对象编程，可以参考 *Concurrent Programming in Java™* 一书。书中建议使用属于当前类的锁进行同步，就像如下代码：

```
package com.oozinoz.businessCore;
import java.util.*;

public class Factory {
    private static Factory factory;
    private static Object classLock = Factory.class;

    private long wipMoves;

    private Factory() {
        wipMoves = 0;
    }
    public static Factory getFactory() {
        synchronized (classLock) {
            if (factory == null)
                factory = new Factory();

            return factory;
        }
    }

    public void recordWipMove() {
        // 挑战!
    }
}
```

`getFactory()` 方法保证：当一个线程开始初始化单例的实例时，如果另一个线程也开始相同的操作，则第二个线程就会等待，直到获得 `classLock` 对象的锁。当它获得锁后，就会发现

该单例不再为 null (因为类仅可能有一个实例，所以可以使用单个静态锁)。

wipMoves 变量记录了半成品 (WIP) 的进展状况。每次把箱子放入一个新机器时，引起移动或记录移动步骤的子系统必须调用工厂单例类的 recordWipMove() 方法。

挑战 8.3

写出 Factory 类 recordWipMove() 方法的代码。

答案参见第 312 页

识别单例

这种独一无二的对象并不罕见。事实上，应用程序中的很多对象都承担了唯一的职责，既然如此，为何要创建拥有相同职责的两个对象？同样的，几乎每个类都拥有独一无二的角色，又何必为相同的类重复开发两次？然而，允许类只能拥有一个实例的单例类却极为罕见。事实上，一个对象或一个类是唯一的，并不意味着就是单例模式。考虑图 8.1 所示的类。

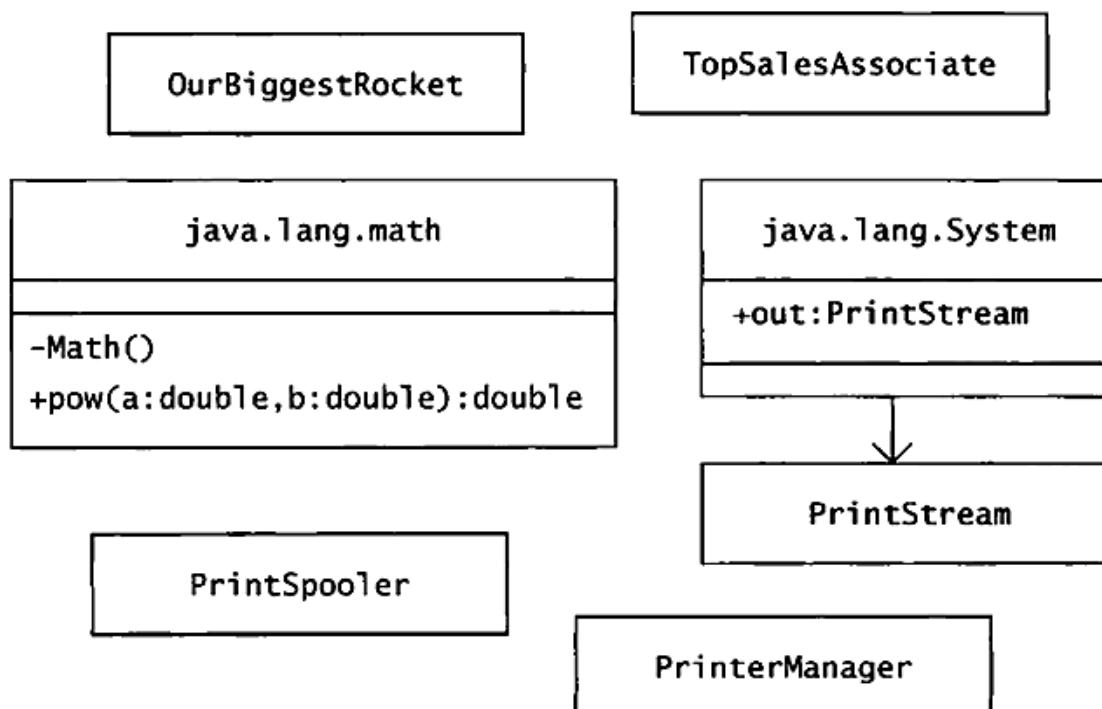


图 8.1 哪些类适用于单例模式

挑战 8.4

对于图 8.1 中的每个类，说说它们是否适用于单例类，为什么？

答案参见第 313 页

单例模式或许是最负盛名的一个模式了，但它很容易被误用，因此不要轻易使用。不要让单例模式成为创建全局变量的一种花哨方法。单例模式常常会引入一些耦合。应该减少运用单例模式的类的数量：最好的方式是，类只需知道与它协作的对象，却不必了解创建它所需要的限制。请注意：倘若需要为测试提供子类或不同的版本，由于单例只能拥有一个实例，因此它可能不是最佳选择。

小结

单例模式保证了类仅有一个实例，并为其提供了一个全局访问点。通过延迟初始化（仅在第一次使用它时才初始化），一个单例对象是达到此目的的通用做法。在多线程环境下，必须小心管理线程间的协作，因为它们访问单例对象方法与数据的时间，可能只有毫厘之差。

对象具有唯一性，并不意味着使用了单例模式。单例模式通过隐藏构造函数，提供对象创建的唯一入口点，从而将类的职责集中在类的单个实例中。

第9章

观察者（Observer）模式

客户端通常通过调用它所感兴趣的对象的方法来获取信息。然而，一旦感兴趣的对象发生改变，就会出现问题：客户端怎样才能知道它所依赖的对象信息发生了改变呢？

你可能遇到过这样的设计：当客户端感兴趣的对象某个方面发生了改变时，需要创建一个对象负责通知客户端。问题是，由于客户端自己知道它所感兴趣的对象的某些方面，因此这个对象自身却不应该承担更新客户端的职责。一个解决方案是当对象发生改变时通知客户端，让客户端自己去查询对象的新状态。

观察者模式的意图是在多个对象之间定义一对多的依赖关系，当一个对象的状态发生改变时，会通知依赖于它的对象，并根据新状态做出相应的反应。

经典范例：GUI 中的观察者模式

在观察者模式中，当某个对象发生改变时，会通知关注它的对象。该模式最常见的例子是用户图形界面。无论用户是在单击按钮，还是调整滑动条，程序中的很多对象都可能会对这些变化产生反应。Java 的设计者认为，用户需要了解 GUI 组件发生的变化，因而在 Swing 中广泛地运用了观察者模式。Swing 将客户端称为“侦听器（Listener）”，并且可以为组件的改变事件注册多个“侦听器”。

考虑图 9.1 中展示的 Oozinoz 公司一个典型的 GUI 应用程序。该应用程序可以让焰火引擎实验根据不同的参数进行可视化展示，从而确定火箭推力和燃料表面燃烧率的关系。

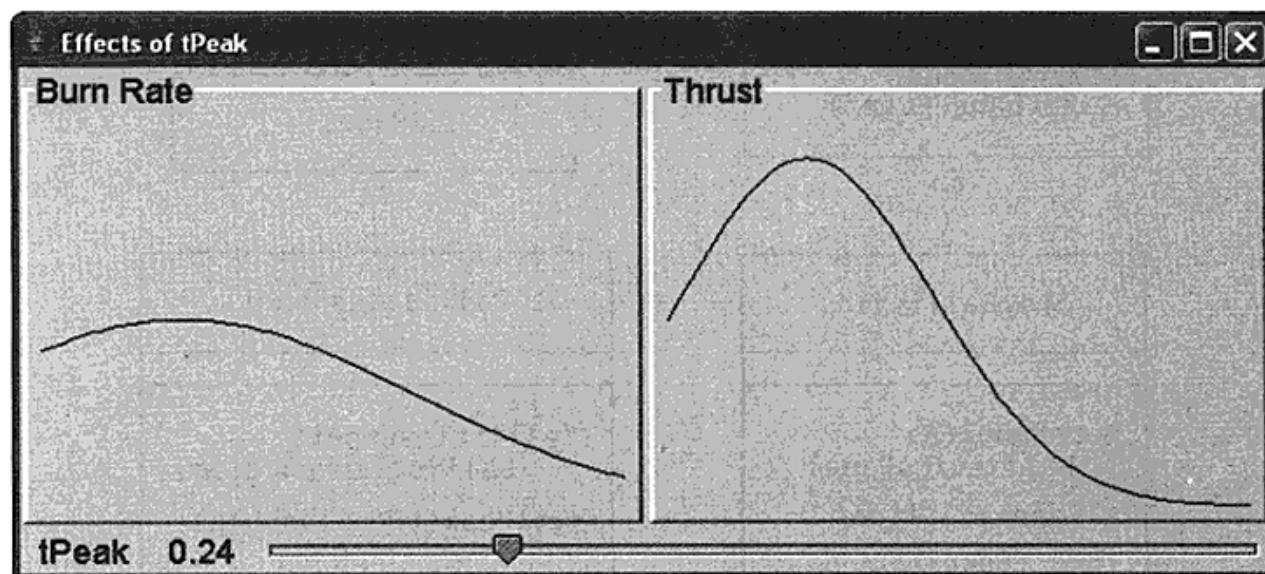


图 9.1 当用户调整滑动条的 t_{peak} 变量时，该曲线可以实时反映出变化

当固体火箭发动机点火时，燃料中暴露在空气中的部分将会燃烧，产生推动力。从点火到最大燃烧率的过程中，燃烧区间从点火区域扩大到整个燃料表面。最大燃烧率的峰值发生在 t_{peak} 时刻。随着燃料的燃烧，表面区域不断减少，直到燃料消耗完毕。这个弹道应用程序约定了时间标准，0 代表点火时刻，1 代表燃烧结束，因此 t_{peak} 是一个在 0~1 之间的数字。

Oozinoz 使用如下等式来计算燃烧率和推力：

$$rate = 25^{-(t-t_{peak})^2}$$

$$thrust = 1.7 \cdot \left(\frac{rate}{0.6} \right)^{1/0.3}$$

图 9.1 中的应用程序展示了 t_{peak} 是如何影响燃烧率和火箭推动力的。当用户移动滑动条时， t_{peak} 的值发生改变，曲线就呈现出新的形状。图 9.2 展示了该程序的一些主要类。

`ShowBallistics` 类和 `BallisticsPanel` 类都属于 `app.observer.ballistics` 包。`BallisticsFunction` 接口属于 `com.oozinoz.ballistics` 包，该接口定义了燃烧率和推力曲线。这个包也包含了 `Ballistics` 工具类，该类实现了 `BallisticsFunction` 接口。

当弹道应用程序初始化滑动条时，应用程序将自己注册成监听器，以便接收滑动条事件。当滑动条滑动时，应用程序负责更新展示曲线的容器，并且更新用于显示 t_{peak} 的标签。

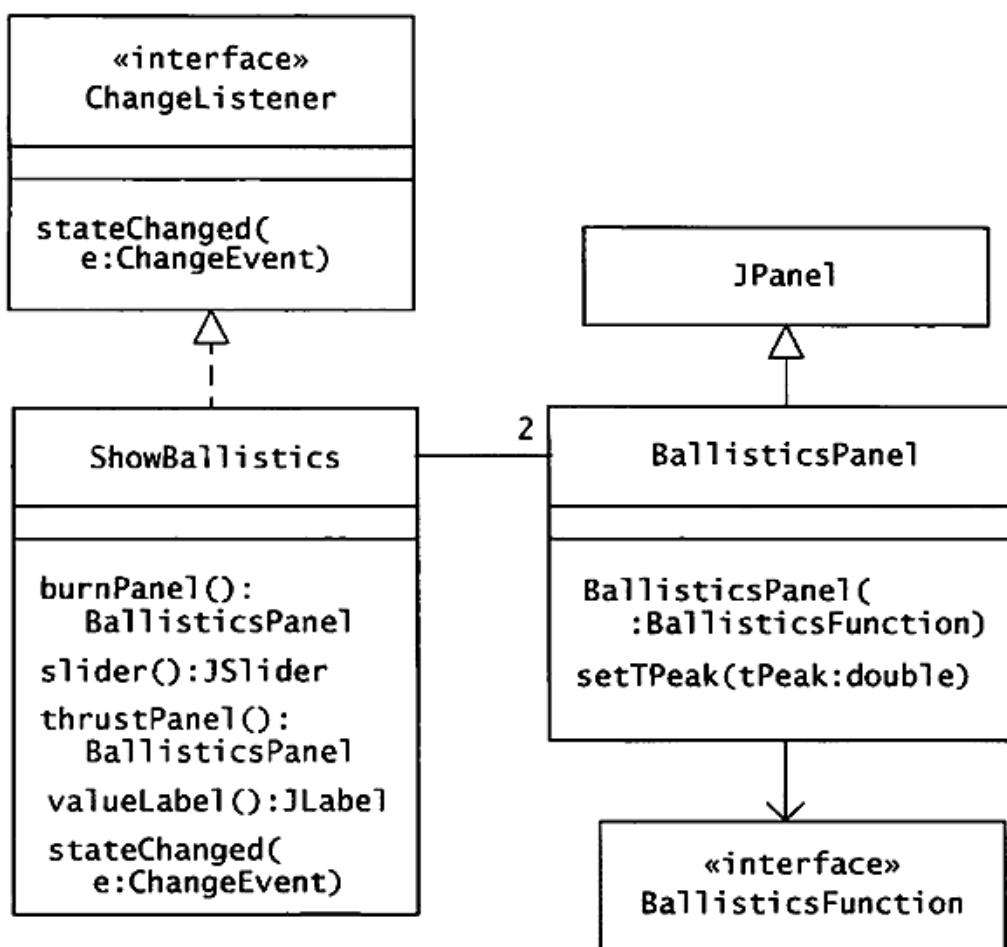


图 9.2 Ballistics 应用程序注册自身以便于接收滑动条事件

挑战 9.1

完成 `ShowBallistics` 的 `slider()` 和 `stateChanged()` 方法，以便弹道容器和 t_{peak} 标签可以反映出滑动条的值。

```

public JSlider slider() {
    if (slider == null) {
        slider = new JSlider();
        sliderMax = slider.getMaximum();
        sliderMin = slider.getMinimum();
        slider.addChangeListener( ?? );
        slider.setValue(slider.getMinimum());
    }
    return slider;
}

public void stateChanged(ChangeEvent e) {
}
  
```

```
doubleval = slider.getValue();
doubletp = (val - sliderMin) / (sliderMax - sliderMin);
burnPanel().??(??);
thrustPanel().??(??);
valueLabel().??(??);
}
```

答案参见第 313 页

`ShowBallistics` 类更新燃烧率与推力容器，以及依赖于滑动条值的 t_{peak} 标签。这种做法很常见，方案也还算不错。但需要注意的是，它完全曲解了观察者模式的意图。Swing 使用观察者模式的目的是为了让滑动条无须关心有哪些客户端对它有兴趣。但是 `ShowBallistics` 应用又将我们带回到之前极力避免的问题：一个独立对象（即应用程序）需要知道要更新哪些对象，并且负责进行相关的调用，而不是让每个独立的对象注册它们自身。

为了创建粒度更细的观察者模式，我们可以对代码进行少量修改，让每个感兴趣的对象注册它们自己，以便可以接收滑动条的改变事件。

挑战 9.2

请设计一个新的类图，可以让感兴趣的对象注册自身以便接收滑动条事件。
不要忘记用于显示滑动条值的标签。

答案参见第 314 页

根据该设计，可以将 `addChangeListener()` 方法的调用从 `slider()` 方法中移出来，放到所依赖组件的构造函数中。

```
public BallisticsPanel2(
    BallisticsFunction func,
    JSlider slider) {
    this.func = func;
    this.slider = slider;
    slider.addChangeListener(this);
}
```

当滑动条滑动时，`BallisticsPanel2` 对象会被通知。标签会重新计算 t_{peak} 值并重新绘制

自己。

```
public void stateChanged(ChangeEvent e) {  
    double val = slider.getValue();  
    double max = slider.getMaximum();  
    double min = slider.getMinimum();  
    tPeak = (val - min) / (max - min);  
    repaint();  
}
```

重构之后出现了一个新问题。这个设计调整了职责的分配，以便让感兴趣的对象向滑动条注册自身，并对滑动条的滑动事件做出自己的反应。这样的职责分配没有问题，但是，现在每个侦听滑动条的组件都需要重新计算 `tPeak` 的值。特别是如果使用挑战 9.2 中提供的 `BallisticsLabel2` 类的解决方案，它的 `stateChanged()` 方法几乎和这里的 `statechanged()` 方法一样。为了消除重复代码，需要从当前设计中提取出一个潜在的领域对象来再次重构。

可以通过引入一个包含关键峰值的 `Tpeak` 类来简化系统。我们让应用程序侦听滑动条，并更新 `Tpeak` 对象，然后让其他所有感兴趣的对象去侦听这个对象。这就是模型/视图/控制器（MVC）模式。查阅 Buschmann 等人的著作 *Pattern Oriented Software Architecture*，其中对 MVC 有更深入的讨论^{译注1}。

模型/视图/控制器

随着应用程序和系统的增长，需要将类和包的职责划分得足够细，以便于维护。模型/视图/控制器是指将对象（模型）从显示它的 GUI 元素中（视图/控制器）分离出来。Java 通过使用侦听器来支持这种分离，但是正如上一节提到的，并不是所有使用侦听器的设计都是 MVC 模式。

`ShowBallistics` 应用的早期版本将 GUI 程序与弹道学知识组合在一起。可以重构这段代码，按照 MVC 的思想去分解应用程序的职责。在这次重构过程中，修正过的 `ShowBallistics` 类应该将视图和控制器保留在 GUI 元素中。

译注1：Buschmann 等人的著作一共分为 5 卷，在架构模式的大前提下，每卷都有一个自己的主题。严格意义上讲，MVC 模式起源于 20 世纪 80 年代的 Smalltalk。现今，MVC 模式已经得到了相当普及的运用，诸多 MVC 框架如 Struts、Ruby On Rails、ASP.NET MVC 已经非常成熟。

MVC 的创建者期望将组件的外观(视图)和行为(控制器)分离。在实际应用中，组件的外观与用户交互的支持是紧耦合的，Swing 的典型应用没有将视图从控制器中分离出来。(如果深入 Swing 内部实现，可能会发现这种分离现象。) MVC 的价值在于：它将模型从应用程序中提了出来，形成了自己的领域。

ShowBallistics 应用程序的模型就是 tPeak 值。为了将其重构为 MVC，需要引入一个 Tpeak 类，该类拥有一个峰值，并且可以允许感兴趣的监听器去监听事件的变化。如下所示：

```
package app.observer.ballistics3;
import java.util.Observable;

public class Tpeak extends Observable {
    protected double value;
    public Tpeak(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }

    public void setValue(double value) {
        this.value = value;
        setChanged();
        notifyObservers();
    }
}
```

如果评审 Oozinoz 公司的这段代码，会发现一个关键点：代码与火箭引擎燃料燃烧率到达峰值的时间几乎没有关系。事实上，这段代码看起来更像是一个通用的工具类：该类可以控制值，且当该值发生改变时，会去通知监听器。我们可以对这段代码进行重构，使其变得更为通用，不过，在重构前先来看看使用修正过的 Tpeak 类的设计，同样很有价值。

现在我们的设计让应用程序能够侦听滑动条，其余的则可以侦听 Tpeak 对象。当滑动条移动时，应用程序为 Tpeak 对象设置了一个新值。面板和文本框侦听 Tpeak 对象，当对象值改变时会更新它们自身。BurnRate 和 Thrust 类使用 Tpeak 对象来计算它们自己的功能，但是它们不需要通过注册来侦听事件。

挑战 9.3

创建一个类图，该图可以展示应用程序和滑动条的依赖关系，以及文本框和测绘容器与 Tpeak 对象的依赖关系。

答案参见第 315 页

该设计允许将滑动条的值同时转换为峰值。应用程序负责更新唯一的 Tpeak 对象，所有侦听（滑动条）值改变的 GUI 对象都可以查询 Tpeak 对象的新值。

然而，除了维持该值外，Tpeak 类再没有其他的价值了。因此我们希望分解出一个维护值的类。另外，像峰值这样的观察值，并不是独立的值，而是领域对象的一个属性。例如峰值是火箭引擎的一个属性。可以通过分离这些类来改善我们的设计，如使用维护值的类让 GUI 对象观察领域对象。

当你从领域对象或者业务对象中分解 GUI 对象时，可以创建一个代码层。这一层代码是拥有相似职责的一组类，通常会被放进一个单独的 Java 包中。像 GUI 层这样的高层，通常仅依赖于同层或者底层代码。层与层之间通常具有清晰定义的接口，就像 GUI 和它所代表的业务层。你要重新组织 ShowBallistics 代码的职责，使之构成系统的一层。就像图 9.3 所示。

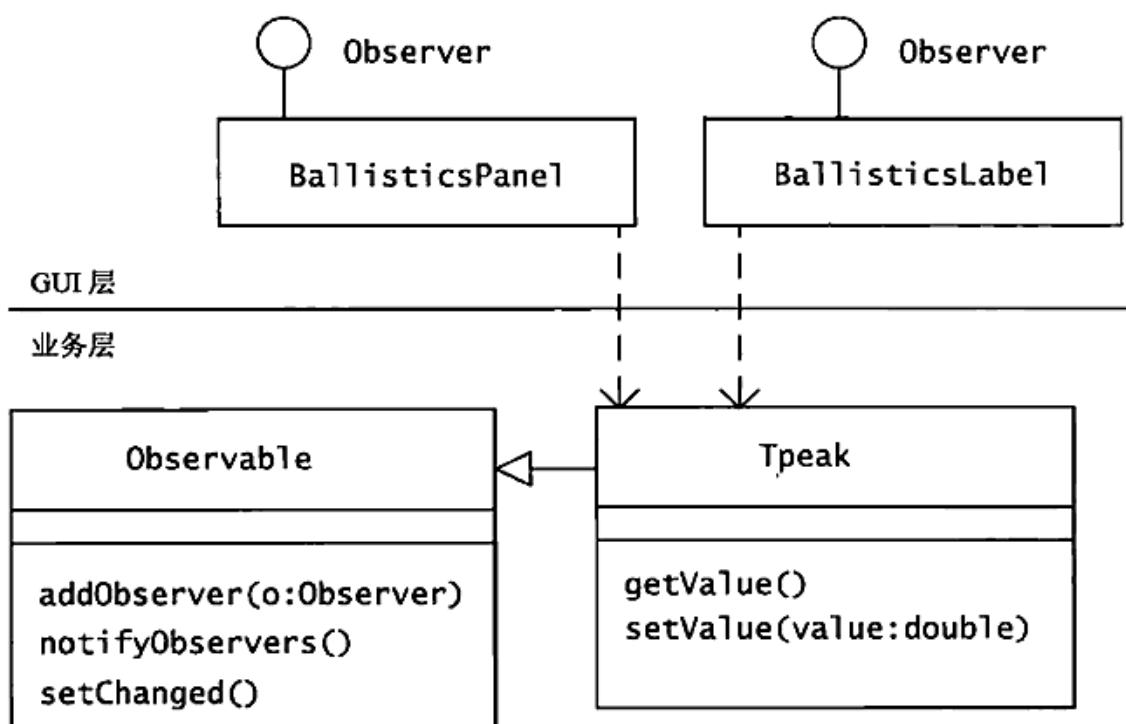


图 9.3 通过创建一个可观察的 Tpeak 类，可以将业务逻辑层从 GUI 层分离出来

图 9.3 的设计创建了一个 `Tpeak` 类对 t_{peak} 值进行建模，该值是应用程序显示弹道公式的一个关键值。`BallisticsPanel` 和 `BallisticsLabel` 类依赖于 `Tpeak`，为了避免让 `Tpeak` 对象负责去更新 GUI 元素，该设计应用了观察者模式，让感兴趣的对象可以注册到 `Tpeak` 类，以便在该类发生改变时得到通知。Java 类库中的 `java.util` 包中提供的 `observable` 类和 `observer` 接口，就是为了支持该设计。`Tpeak` 类继承自 `observable` 类，它的数值发生改变时会通知其观察者。

```
public void setValue(double value) {
    this.value = value;
    setChanged();
    notifyObservers();
}
```

注意，你需要调用 `setChanged()` 方法，以便使继承自 `observable` 类的 `notifyObservers()` 方法可以将改变广播出去。

`notifyObservers()` 方法调用每个已经注册的观察者的 `update()` 方法。如图 9.4 所示，`observer` 接口的实现者都必须实现 `update()` 方法。

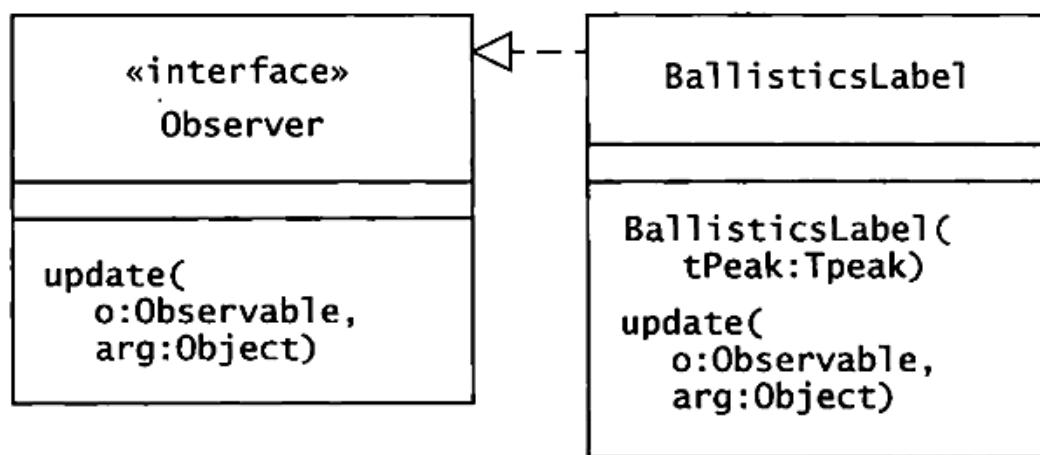


图 9.4 `BallisticsLabel` 类是一个观察者：它可以在 `Observable` 对象里注册它感兴趣的东西，当 `Observable` 对象发生改变时，`BallisticsLabel` 对象的 `update()` 就会被调用

`BallisticsLabel` 对象无须保持对它所观察的 `Tpeak` 对象的引用。当然，在 `BallisticsLabel` 的构造函数中，该对象可以被注册成 `Tpeak` 对象的监听器。该标签类的 `update()` 方法将会接收一个参数类型为 `observable` 的 `Tpeak` 对象，并且可以将参数转化为 `Tpeak` 类型，得到新值，然后更新标签的文本以及重绘曲线。

挑战 9.4

写出 `BallisticsLabel.java` 的完整代码。

答案参见第 315 页

弹道应用程序的新设计将业务对象从展现它的 GUI 元素中分离出来。这一设计包含两个关键步骤：

1. 观者者的实现类必须注册它们感兴趣的对像，并且自身能做出相应的变更，这通常包括重绘它们自己。
2. 当被观察者的实现类的值发生改变时，必须记得通知观察者。

在弹道应用程序中，这两个步骤几乎涵盖了需要跨层调用的代码。同时，还需要准备一个 `Tpeak` 对象，当应用程序的滑动条滑动时，该对象的值能随之变动。这可以通过准备一个 `ChangeListener` 的匿名类来实现。

挑战 9.5

假设 `tPeak` 是 `Tpeak` 的一个实例，并且是 `ShowBallistics3` 类的一个属性。请补充完成 `ShowBallistics3.slider()` 的代码，以便滑动条的改变能更新到 `tPeak`：

```
public JSlider slider() {
    if (slider == null) {
        slider = new JSlider();
        sliderMax = slider.getMaximum();
        sliderMin = slider.getMinimum();
        slider.addChangeListener
        (
            new ChangeListener()
            {
                // 挑战!
            }
        );
        slider.setValue(slider.getMinimum());
    }
    return slider;
}
```

```

    }
    return slider;
}

```

答案参见第 316 页

当你运用 MVC 时，事件的流向可能看起来有些迂回。弹道应用程序中滑动条的移动会驱动 `ChangeListener` 去更新 `Tpeak` 对象。反过来，`Tpeak` 对象的改变会通知应用程序的标签和面板容器对象，然后让这些对象重绘它们自己。变化的传播从 GUI 层到业务层，最后又回到 GUI 层。

挑战 9.6

请填写图 9.5 所示的消息。

答案参见第 316 页

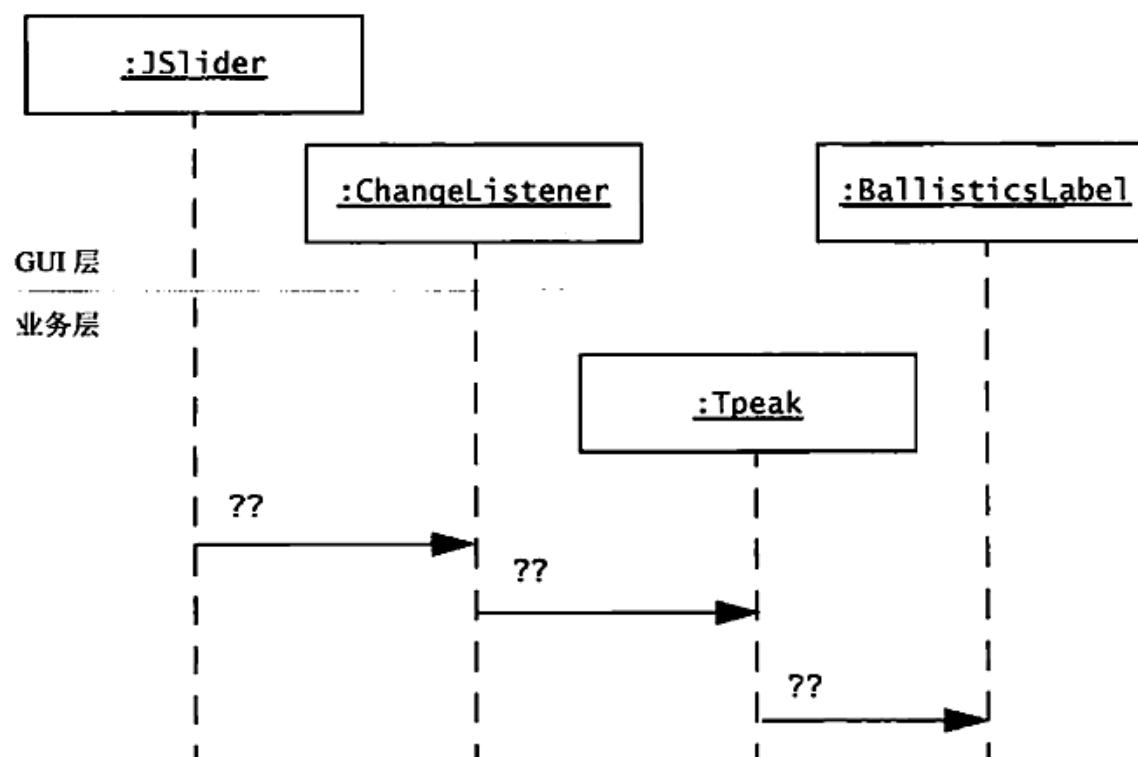


图 9.5 MVC 使得调用从 GUI 层到业务层，又从业务层回到 GUI 层

分层设计的好处在于使用接口能够分离不同的层。代码的分层就是职责的分层，这就使得代码更加容易维护。例如，在这个弹道应用程序中，你可以新增一个类似手持设备的 GUI，而不需要改变业务对象层的类。在业务对象层，你可以新增一个可以更新 `Tpeak` 对象的事件源，

而不需要更改 GUI。观察者模式提供的机制会自动更新 GUI 层的对象。

分层设计还提供了让不同层运行在不同计算机上的可能性，这样的系统称之为 *n* 层系统。一个 *n* 层软件设计可以大量减少运行在用户桌面上的程序的代码量。它还能使你在更改系统的业务类后无须升级安装用户机器上的软件。这极大地简化了部署。然而，在计算机之间传送消息是有代价的，你需要慎重实施 *n* 层系统的部署。例如，滚动条的来回滑动事件频繁在用户端和服务器之间发生，会消耗一定时间，这是用户难以接受的。此时，就必须让滑动事件发生在用户机器上，然后在达到新峰值时提交，从而达到分离用户行为的目的。

简而言之，观察者模式支持 MVC，该模式鼓励分层设计软件，这给软件开发和部署带来了很多实用的好处。

维护 Observable 对象

有时我们可能无法创建一个观察者类，该类继承自 `observable` 类的子类。特别是当该类已经是其他类（不是 Java 内置的 `Object` 类）的子类时，我们可以提供一个拥有 `Observable` 对象的类，该类可以将关键的方法调用转发给 `observable` 对象。`java.awt` 包中的 `Component` 类就使用了这种方法，但是它用了一个 `PropertyChangeSupport` 对象代替了 `observable` 对象。

`PropertyChangeSupport` 类和 `observable` 类非常相似，但前者属于 `java.beans` 包。JavaBeans API 支持创建可复用的组件，并在 GUI 组件中得到了广泛使用。当然，你可以在任何地方使用它。`Component` 类使用 `PropertyChangeSupport` 对象让感兴趣的观察者将自己注册到其中，以便在标签、面板容器或其他 GUI 组件发生改变时得到通知。图 9.6 展示了 `java.awt` 包中的 `Component` 类和 `PropertyChangeSupport` 类的关系。

`PropertyChangeSupport` 类描述了一个在使用观察者模式时需要解决的问题：被观察类需要向观察者提供多少改变的细节？该类使用了“推”模式，用模型给出了改变的细节（在 `PropertyChangeSupport` 中，通知给出了由旧值到新值的属性变化）；另一个方式是“拉”模式，它建立的模型会告诉观察者有改变发生，但是观察者需要查询模型来获取改变的具体信息。两种方式各有千秋。“推”模式需要更多的工作量，并且会将观察者与被观察者绑定起来，但是它提供了较好的性能。

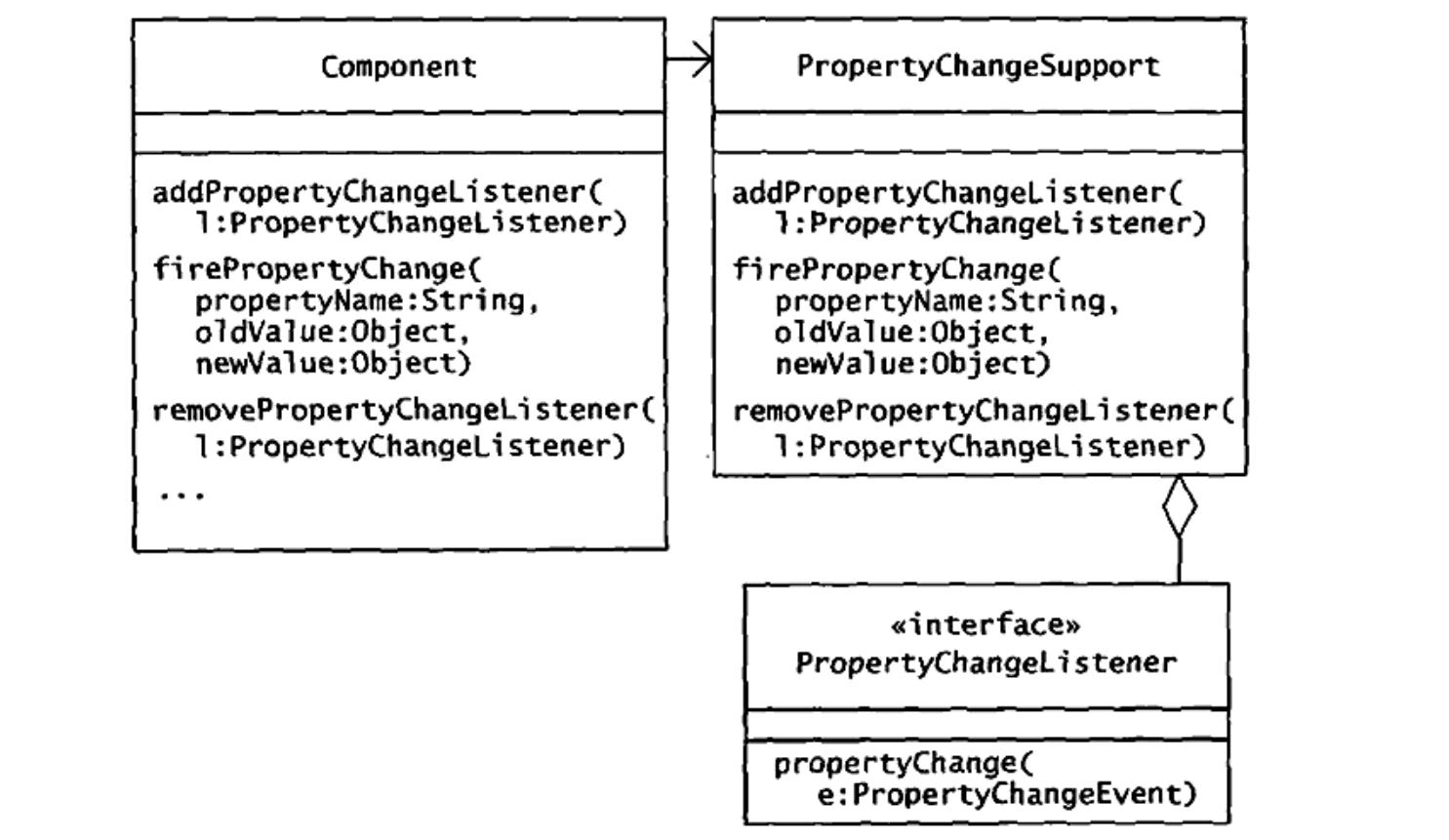


图 9.6 一个 Component 对象维护一个 PropertyChangeSupport 对象，
一个 PropertyChangeSupport 对象维护一组监听器集合

Component 类复制了 **PropertyChangeSupport** 类的部分接口。**Component** 类的每一个方法都会将消息调用转发给 **PropertyChangeSupport** 类的实例。

挑战 9.7

完成图 9.7 中的类图，该类图展示了 Tpeak 使用 **PropertyChangeSupport** 对象来管理监听器。

答案参见第 316 页

使用 **Observer** 也好，使用 **PropertyChangeSupport** 或其他类也罢，构建观察者模式的要点是要在对象间建立起一个一对多的关系。当一个对象状态发生改变时，所有依赖它的对象都会被通知，并做出相应的更新。这有助于缩小职责范围，使得维护观察者和被观察对象变得轻而易举。

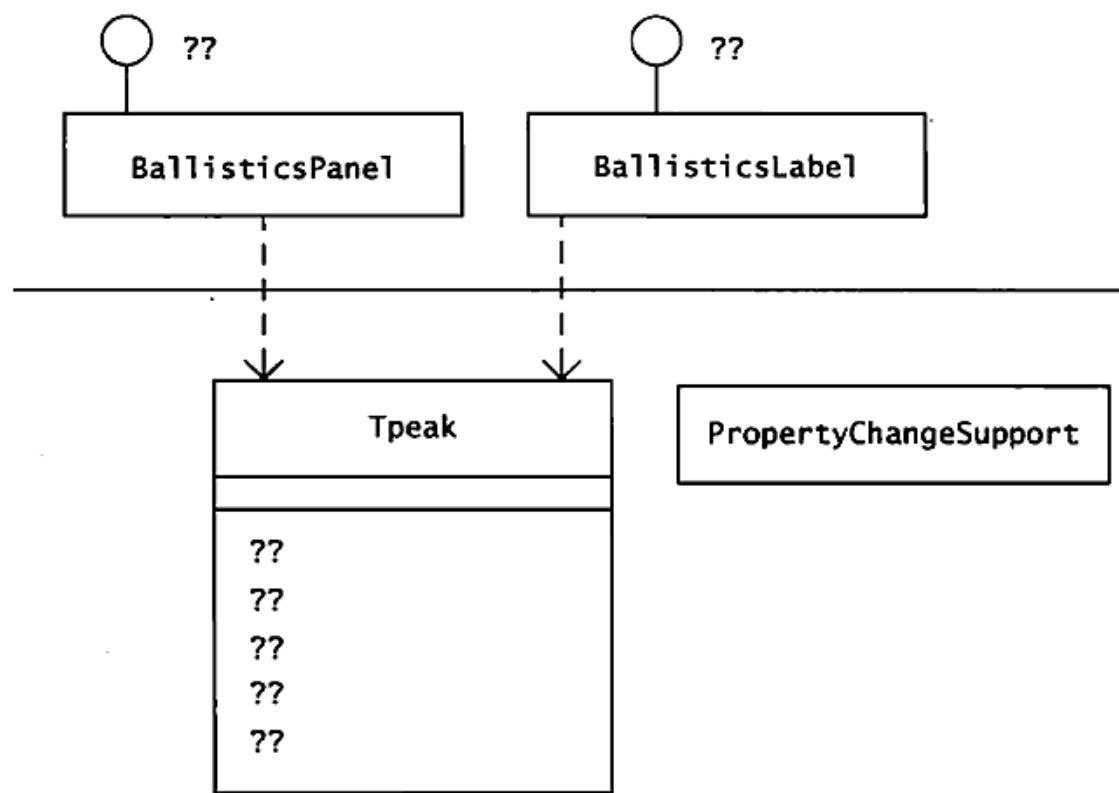


图 9.7 Tpeak 业务对象可以将影响侦听者的调用委托给 PropertyChangeSupport 对象

小结

观察者模式经常出现在 GUI 应用程序中，它是 Java GUI 类库的基础模式。有了这些组件，当需要简单地将事件通知给相关对象时，就无须再改变或者继承自组件类了。对规模较小的应用程序而言，通常的做法是注册一个单独的对象，它将负责接收 GUI 中的所有事件。这一做法本身没有问题，但你必须知道它违背了观察者模式职责分离的意图。对于大型的 GUI 程序，建议使用 MVC 模式：让每一个相关对象都注册自己的事件，而不是让一个中间对象负责注册所有的事件。MVC 还使你能够创建相互独立的松耦合的层级系统，系统的每一层都可以运行在不同的机器上。

第 10 章

调停者（Mediator）模式

面向对象开发要求尽可能恰当地分配职责，要求对象能够独立地完成自己的任务。例如观察者模式，就是通过最小化对象与对象之间的职责交互，从而支持职责的合理分配。单例模式是将职责集中在某个对象中以便其他对象的访问与重用。与单例模式相似，调停者模式也是集中职责，但它是针对一组特殊的对象，而不是系统中全部的对象。

当对象间的交互趋向复杂，而每个对象都需要知道其他对象的情况时，提供一个集中的控制权是很有用的。当相关对象的交互逻辑独立于对象的其他行为时，职责的集中同样有用。

调停者模式的意图是定义一个对象，封装一组对象的交互，从而降低对象间的耦合度，避免了对象间的显式引用，并且可以独立地改变对象的行为。

经典范例：GUI 调停者（Mediator）

在开发 GUI 应用时，可能经常遇到调停者模式。应用程序常常会变得越来越庞大，类聚集了太多的代码，这些代码完全可以重构到多个类中。如第 4 章外观模式中的 `ShowFlight` 类，最初扮演了 3 个角色，在重构之前，它是一个显示面板，一个完整的 GUI 应用，一个轨迹计算器。重构完成后，飞行轨迹面板的应用变得很简单，仅仅包含少量代码行。然而，对于大型应用，即使这些代码仅仅包含创建部件与安排部件的交互逻辑，在经过重构后，仍然很复杂。思

考图 10.1 所示的应用。

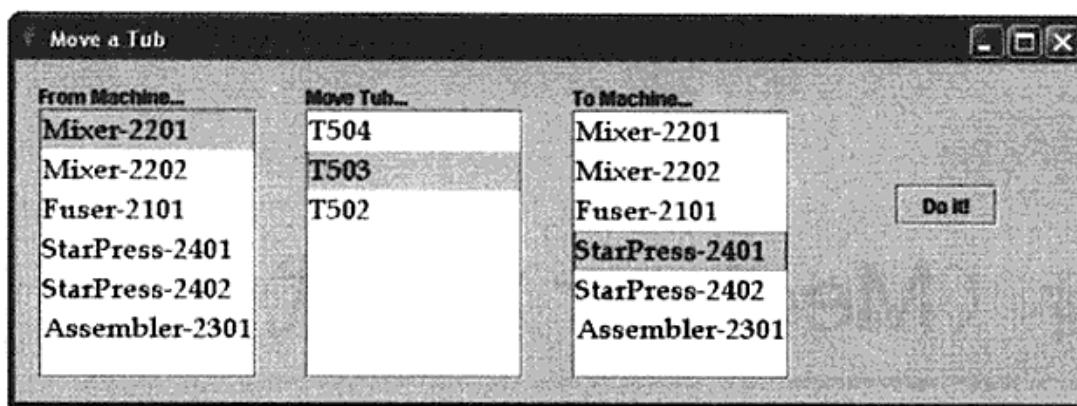


图 10.1 该应用程序可以使用户手工更新药品位置

Oozinoz 公司使用橡胶箱存放化学药品。机器会读取箱子上的条形码，以便记录箱子在工厂中的位置。有时，人工干预是必要的，特别是由人而非机器人去移动箱子时。图 10.1 展示了一个全新的、开发了部分功能的应用程序，可以使用户指定箱子在机器上的位置。

在包 `app.mediator.moveATub` 的 `MoveATub` 应用程序中，用户在左侧箱子列表菜单中选择其中一台机器，继而选择箱子，再选中目标机器，最后单击 `Do it!` 按钮，系统就会更新箱子的位置。图 10.2 展示了程序中的部分类。

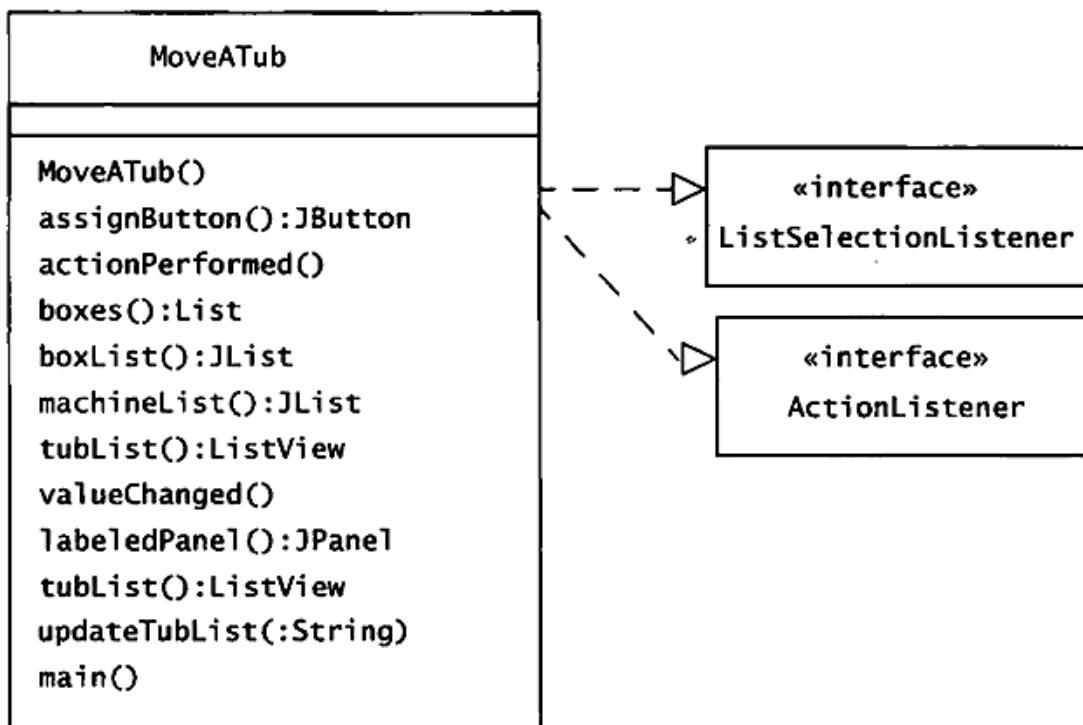


图 10.2 `MoveATub` 类混合了创建组件、事件处理以及模拟数据库的方法

该程序的开发人员最初使用向导进行开发，并且已经进行了重构。MoveATub 类的一半方法都包含了拥有 GUI 组件的变量，这些变量是可以延迟初始化的。`assignButton()`方法就是一个典型的例子：

```
private JButton assignButton() {
    if (assignButton == null) {
        assignButton = new JButton("Do it!");
        assignButton.setEnabled(false);
        assignButton.addActionListener(this);
    }
    return assignButton;
}
```

程序员已经清除了由向导自动生成的那些指定按钮位置和大小的硬编码。然而，现在的问题是，MoveATub 在类中具有的职责不够单一。

大多数静态方法都提供了包含箱子名称与机器名称的模拟数据库。开发人员最终放弃了仅仅使用名称的工作方式，并用 Tub 对象和 Machine 对象来升级程序。程序中剩下的方法主要包含处理事件的逻辑。例如，`valueChanged()`方法用来管理分配按钮是否可用：

```
public void valueChanged(ListSelectionEvent e) {
    // ...
    assignButton().setEnabled(
        ! tubList().isSelectionEmpty()
        && ! machineList().isSelectionEmpty());
}
```

我们可能会考虑将 `valueChanged()` 方法与其他事件处理方法移到一个单独的协调类中。首先注意到，调停者模式已经应用到这个类中，即组件间不能直接更新对方。例如，无论是机器组件还是列表组件，都不能直接更新分配按钮。然而，MoveATub 程序注册了列表的选择事件，根据列表中的选项是否被选中来更新按钮。在移动箱子的应用程序中，MoveATub 对象扮演了调停者，接收事件并分配相应的动作。

Java 类库的机制促使你使用调停者，尽管 JCL 并不要求应用程序必须是自己的调停者。为了避免将组件创建方法、事件处理方法与模拟数据库方法全都混合到一个类中，可以按照职责的不同，将它们写到不同的类。

挑战 10.1

完成图 10.3 中重构 MoveATub 后的类图，引入一个独立的模拟数据库类与一个调停者类来接收 MoveATub GUI 的事件。

答案参见第 318 页

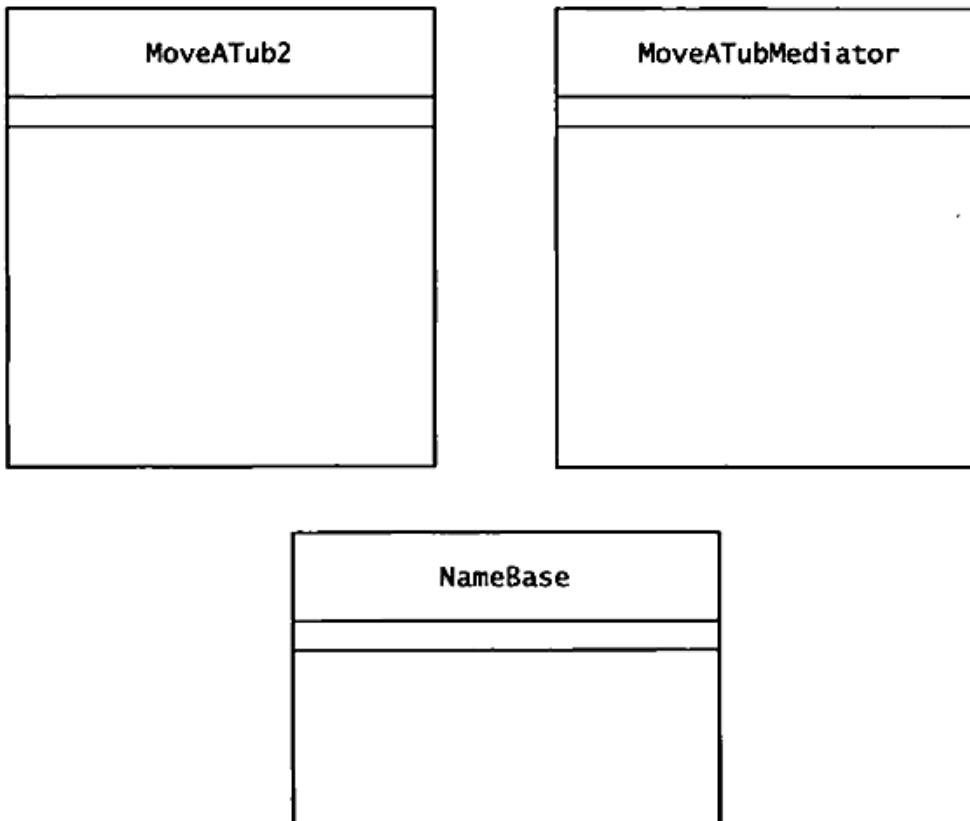


图 10.3 分离程序中的组件创建、事件处理以及模拟数据库的模块

通过重构，调停者变成一个独立的类，使得我们可以独立地开发并处理它。运行 MoveATub 程序后，组件将事件传递给 MoveATubMediator 对象。调停者可能会调用非 GUI 对象：例如，当分配完成后更新数据库。调停者对象可能也会回调 GUI 的组件：例如，当分配完成后禁用该按钮。

挑战 10.2

当用户单击 Do it! 按钮后，绘出后面发生的事件图。标出你认为最重要的对象，并且标出在这些对象间传递的消息。

答案参见第 319 页

GUI 组件对调停者模式的运用可谓水到渠成，当事件发生时，它会通知调停者而不是直接更新其他组件。GUI 应用可能是最常见的使用调停者模式的程序了。但在其他场景下，可能也需要引入调停者模式。

无论何时，只要对象之间存在复杂的交互行为，就可以将这些交互职责集中到这些对象之外的一个调停者对象中。这就形成了松耦合——减少了对象间的直接交互。在一个独立类中，管理对象间的交互还能简化与标准化对象间的交互规则。比如，当需要管理关系的一致性时，可以使用调停者模式。

关系一致性中的调停者模式

面向对象的方法总是让人很容易地将现实世界中的对象与 Java 对象映射起来。然而，用 Java 对象来建模现实世界至少存在两个基本的缺陷。首先，现实世界中的对象可以随时间的变化而变化，而 Java 却没有这种内建的机制。例如，赋值语句清除了语句左边对象之前的值，却没有记住该值，但是人却可以。其次，在现实世界中，关系和对象一样重要，但是包括 Java 语言在内的面向对象语言中，关系却没有任何的支持。例如，没有内建机制支持这样的事实：如果压缩机 2402 在 1 号车间，则 1 号车间一定包含压缩机 2402。事实上，这种关系很可能出错，这促使我们引入调停者模式。

考虑 Oozinoz 公司的药品箱。箱子总是被分配给某台特定机器。可以用表格来对这种关系进行建模，如表 10.1 所示。

表 10.1 在表格中记录关系信息以保持关系一致性

| 箱 子 | 机 器 |
|------|---------------------|
| T305 | StarPress-2402 |
| T308 | StarPress-2402 |
| T377 | ShellAssembler-2301 |
| T379 | ShellAssembler-2301 |
| T389 | ShellAssembler-2301 |
| T001 | Fuser-2101 |
| T002 | Fuser-2101 |

表 10.1 展示了箱子与机器间的关系——两者间是如何相互作用的。从数学的角度来说，关系是所有有序对象序列对的子集。因此，箱子与机器间存在一种关系，反过来，机器与箱子间也存在一种关系。保持表中箱子列的名称唯一，可以保证一个箱子不会同时分配给两台机器。

下面给出了对象模型中关系一致性的严格定义。

关系一致性

在对象模型中，如果在对象 a 指向对象 b 的同时，对象 b 也指向对象 a ，则称对象 a 与对象 b 关系一致。

在更严格的定义中，假定有 Alpha 和 Beta 两个类， A 代表 Alpha 类的一组对象集合， B 代表 Beta 类的一组对象集合，让 a 和 b 分别代表 A 和 B 的成员，让有序对 (a, b) 代表 $a \in A$ ，同时 $b \in B$ 。这种引用既可以是直接引用，也可以是集合引用，比如对象 a 拥有一个集合对象，而该集合包含了 b 。

$A \times B$ 的笛卡儿积是所有 $a \in A$ 和 $b \in B$ 的有序对 (a, b) 的集合。集合 A 和 B 可以产生两个笛卡儿积 $A \times B$ 和 $B \times A$ 。 A 和 B 的对象模型关系是存在于 $A \times B$ 对象模型的子集。让 AB 代表这个子集， BA 代表该模型中 $B \times A$ 的子集。

任意二元关系 $R \subseteq A \times B$ 都有一个逆关系 $R^{-1} \subseteq B \times A$ ，定义如下：

$$(b, a) \in R^{-1} \text{ 当且仅当 } (a, b) \in R$$

如果对象模型是一致的，则 AB 的逆关系提供一组 B 实例到 A 实例的引用。换句话说，当且仅当 BA 与 AB 互逆时，Alpha 类与 Beta 类才是关系一致的。

当在表格中记录箱子和机器之间的关系信息时，可以限制每一个箱子仅在箱子列中出现一次，以此保证箱子与机器一一对应。在关系型数据库中，一种做法是用箱子列作为表的主键。在现实中存在这样的模型，当且仅当 $(a, b) \in R, (b, a) \in R^{-1}$ 时，一个箱子就不会同时出现在两个机器中。

对象模型无法保证关系一致性如关系模型一般简单。考虑 MoveATub 应用程序，开发者会逐渐修改设计，不再使用名称，而是使用 Tub 对象与 Machine 对象来做设计。当现实世界中箱子在机器附近时，代表箱子的对象将会引用代表机器的对象。每一个机器对象都会有一组箱子对象集合，该集合表示在机器附近的箱子。图 10.4 展示了一个典型的对象模型。

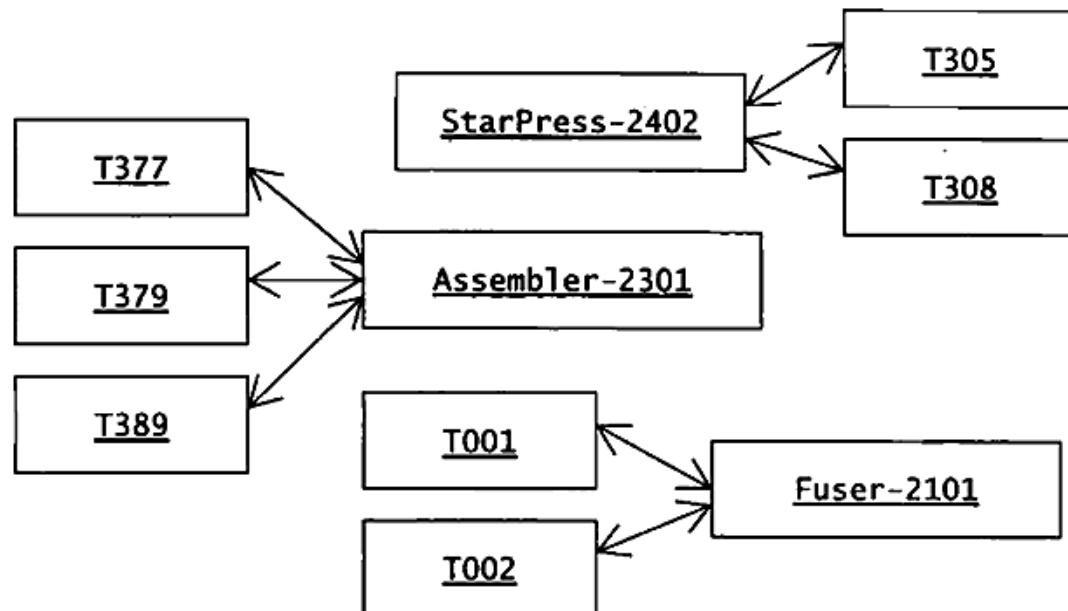


图 10.4 对象模型中分布的关系信息

图 10.4 中的箭头表示箱子知道所属机器，机器也知道包含了哪些箱子。箱子与机器的关系信息现在分布在多个对象中，而不是集中在一个表里。箱子与机器关系的分布化，增加了管理的难度。这时就需要考虑调停者模式。

思考 Oozinoz 公司的一个问题：一名开发人员开始对一台新机器建模，该机器包含了读取箱子条形码的功能。通过扫描箱子的 ID，开发人员通过如下代码将箱子位置 t 的信息通知给机器 m 。

```
// 在 tub 设置机器，而在机器中添加 tub
t.setMachine(m);
m.addTub(t);
```

挑战 10.3

假设对象的初始状态如图 10.4 所示，对象 t 代表箱子 T308，对象 m 代表机器 Fuser-2101。完成图 10.5 中的对象图，以展示执行更新箱子位置代码的效果。看看出现了什么问题？

答案参见第 319 页

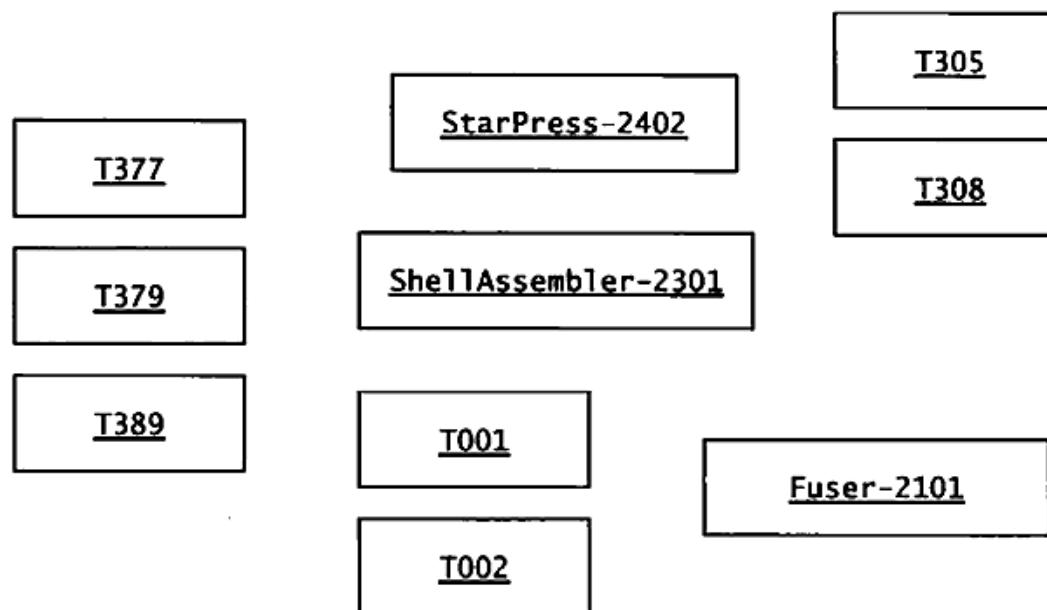


图 10.5 完成此图后，在更新箱子位置时，将看到出现的问题

管理关系一致性最简单的做法是：将关系信息拉回到到单个表中，并通过调停者对象进行管理，而不让机器与箱子相互知道对方。所有对象只需要维护一个到调停者对象的引用。这个“表”可以是 Map 类（`java.util` 包中）的一个实例。图 10.6 展示了类图中的调停者对象。

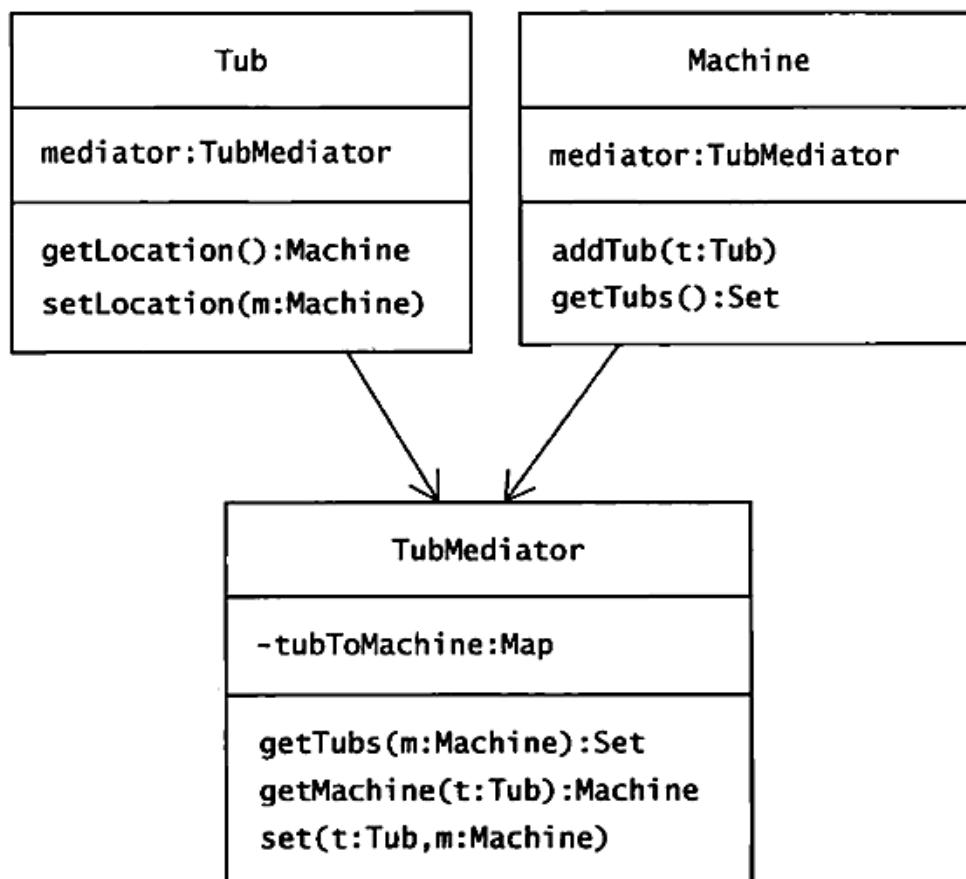


图 10.6 箱子和机器依赖于一个调停者，通过该调停者来控制箱子与机器的关系

Tub 类中包含一个位置信息，允许记录箱子离哪台机器比较近。代码保证了一个箱子在同一时间只能在一个地方，并且使用 TubMediator 对象来管理箱子与机器的关系：

```
package com.oozinoz.machine;
public class Tub {
    private String id;
    private TubMediator mediator = null;

    public Tub(String id, TubMediator mediator) {
        this.id = id;
        this.mediator = mediator;
    }

    public Machine getLocation() {
        return mediator.getMachine(this);
    }

    public void setLocation(Machine value) {
        mediator.set(this, value);
    }

    public String toString() {
        return id;
    }

    public int hashCode() {
        // ...
    }

    public boolean equals(Object obj) {
        // ...
    }
}
```

Tub 类的 setLocation() 方法使用一个调停者来更新箱子的位置，并委托其维护与调停者之间的关系一致性。Tub 类实现了 hashCode() 与 equals() 方法，以便在散列表中适当存储 Tub

对象。代码细节如下所示：

```
public int hashCode() {
    return id.hashCode();
}
public boolean equals(Object obj) {
    if (obj == this) return true;

    if (obj.getClass() != Tub.class)
        return false;
    Tub that = (Tub) obj;
    return id.equals(that.id);
}
```

`TubMediator` 类使用 `Map` 对象来存储箱子与机器之间的关系。通过将关系保存在单个表中，调停者可以保证对象模型永远不会有两台机器同时处理一个箱子的情况。

```
public class TubMediator {
    protected Map tubToMachine = new HashMap();

    public Machine getMachine(Tub t) {
        // 挑战!
    }

    public Set getTubs(Machine m) {
        Set set = new HashSet();
        Iterator i = tubToMachine.entrySet().iterator();
        while (i.hasNext()) {
            Map.Entry e = (Map.Entry) i.next();
            if (e.getValue().equals(m))
                set.add(e.getKey());
        }
        return set;
    }

    public void set(Tub t, Machine m) {
        // 挑战!
    }
}
```

挑战 10.4

写出 `TubMediator` 类中 `getMachine()` 和 `set()` 方法的代码。

答案参见第 320 页

如果没有引入调停者类，而是通过在 `Tub` 类与 `Machine` 类中增加逻辑，你可能无法保证两台机器不会同时处理一个箱子。不过，对于化学箱子以及机器而言，这种关系一致性的逻辑几乎无效，也容易出错。特别是容易导致箱子被移到新机器中，更新了箱子与新机器，却忘记更新箱子在原机器中的位置。将关系管理的代码逻辑转移到调停者，由这样一个独立的类来封装对象间的交互逻辑。

将关系管理的代码逻辑写到一个调停者中，让一个独立的类来封装对象间的交互逻辑。对调停者而言，很容易保证在更改 `Tub` 对象的位置时，自动将箱子从原机器上移除。下面来自 `TubTest.java` 的 JUnit 测试代码体现了这种行为：

```
public void testLocationChange() {
    TubMediator mediator = new TubMediator();
    Tub t = new Tub("T403", mediator);
    Machine m1 = new Fuser(1001, mediator);
    Machine m2 = new Fuser(1002, mediator);

    t.setLocation(m1);
    assertTrue(m1.getTubs().contains(t));
    assertTrue(!m2.getTubs().contains(t));

    t.setLocation(m2);
    assertFalse(m1.getTubs().contains(t));
    assertTrue(m2.getTubs().contains(t));
}
```

如果存在一个不涉及关系型数据库的对象模型，则可以使用调停者来维持模型的关系一致性。将关系管理逻辑转移给这些调停者，让它们专门维护模型的关系一致性。

挑战 10.5

与其他模式一样，调停者模式也将逻辑从一个类转移到另一个新类中。请列出另外两种模式，它们都是通过重构将现有类或者继承关系中的行为移出的。

答案参见第 321 页

小结

调停者模式提供了松耦合的对象关系，避免了关联对象的显式引用。该模式经常应用在 GUI 程序开发中，使用它可以让你不用关心对象间复杂的更新关系。Java 架构会指导你使用该模式，鼓励你定义一些对象来监听 GUI 事件。如果你用 Java 开发用户界面，很可能需要使用调停者模式。

尽管在创建 GUI 程序时，Java 会要求使用调停者模式，但它并不要求将这个调停者移出应用类。不过，这种做法其实可以简化代码。调停者类可以集中处理 GUI 组件的交互，应用程序类则可以集中处理组件的构建。

还有其他一些例子可以引入调停者对象。例如，可能需要一个调停者来集中管理对象模型的关系一致性，甚至可以在任何需要封装对象交互的地方使用调停者模式。

第 11 章

代理（Proxy）模式

普通对象可以通过公共接口完成自己所需完成的工作。然而，有些对象却由于某些原因无法履行自己日常的职责。例如有的对象加载时间过长，有的对象运行在其他的计算机上，或者需要拦截发送到对象的消息等。对于这些场景，我们可以引入代理对象，通过它承担客户端需要的职责，并将相应的请求转发给底层的目标对象。

代理模式的意图是通过提供一个代理（Proxy）或者占位符来控制对该对象的访问。

经典范例：图像代理

代理对象通常拥有一个几乎和实际对象相同的接口。它常常会控制访问，并将请求合理地转发给底层的真实对象。代理模式的一个经典范例是如何避免将较大的图像加载到内存中。假设应用程序中的图像存在于页面或者容器中，并且在初始状态时并未显示。为避免在使用图像前将图像都加载进内存，就需要为这些图像创建一些代理，以便在真正需要使用图像时，才执行加载的操作。本节提供了一个图像代理的范例。但需要注意的是，使用代理模式的设计有时非常脆弱，因为它依赖于将方法调用转发给底层对象。这种转发机制可能会创建难以维护的脆弱设计。

假设 Oozinoz 公司的工程师在使用图像代理时，出于性能方面的考虑，在需要加载大图像

时，显示一个小的临时图片。工程师有一个原型版本，如图 11.1 所示。该程序的代码在 `app.proxy` 包的 `ShowProxy` 类中，支持该程序的底层代码在 `com.oozinoz.imaging` 包中。

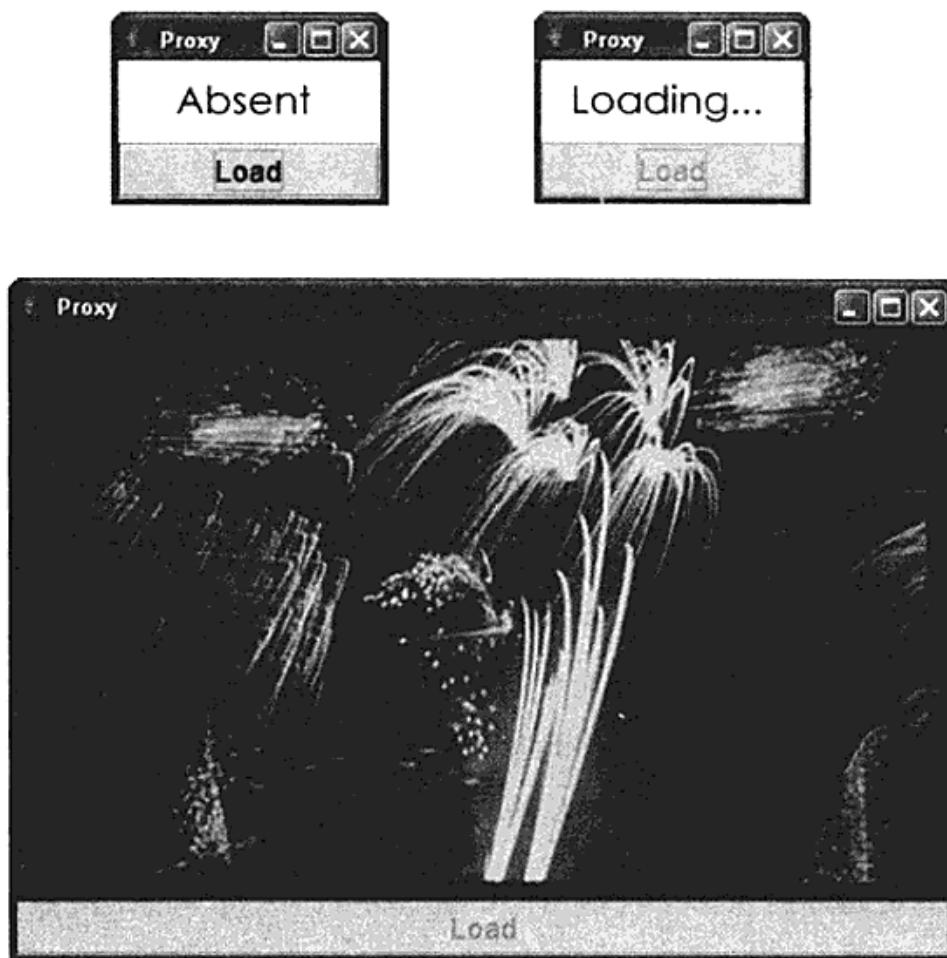


图 11.1 三张截图分别展示了应用程序中大图像在展示前、展示中，以及展示后的样子

用户界面包含三张图片：一张显示图片尚未加载，一张显示真实的图片正在加载，最后一张显示真实的图片。当应用程序启动时显示“Absent”图片，该图片需要提前用绘图工具做好。当用户单击 Load 按钮时，图片立刻就变成了预制好的“Loading...”。目标图像加载完毕后，显示该图片。

有一种简单的方法可以显示 JPEG 图片，创建一个 `ImageIcon` 类，将 JPEG 文件作为参数传递给该类，再用 `JLabel` 来显示图像：

```
ImageIcon icon = new ImageIcon("images/fest.jpg");
JLabel label = new JLabel(icon);
```

在创建应用程序时，需要给 `JLabel` 传递一个代理对象，该对象将转发绘图请求给：(1) “absent” 图像，(2) “loading” 图像，(3) 请求的图像。图 11.2 所示的顺序图大致给出了调用

的消息流。

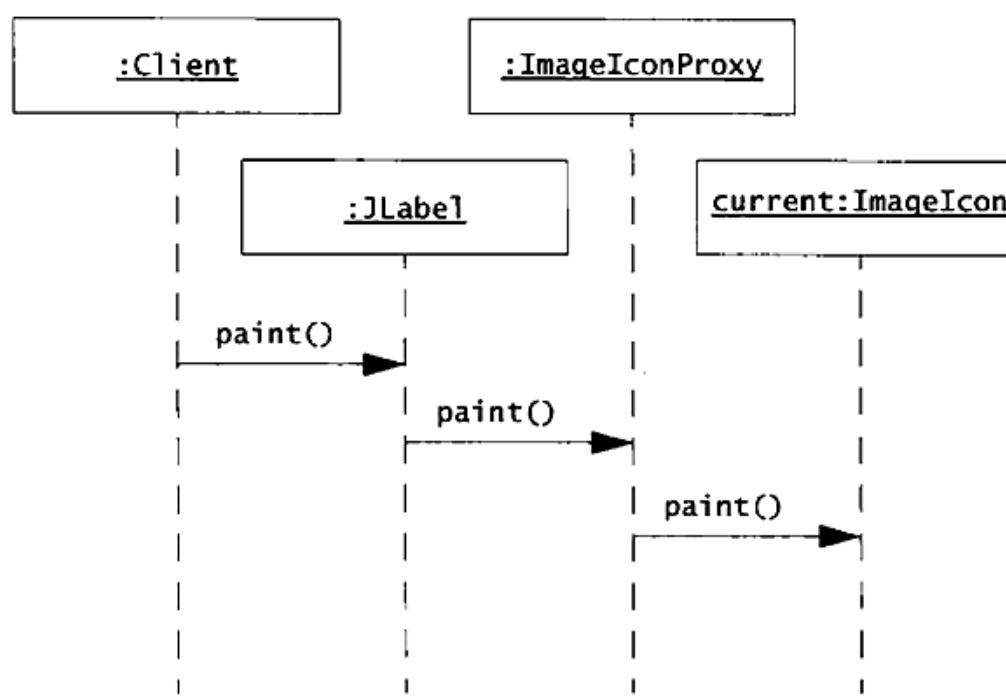


图 11.2 ImageIconProxy 对象将 paint()方法的请求转发给当前的 ImageIcon 对象

当用户单击 Load 按钮时，代码中的 ImageIconProxy 对象会将当前图片切换成 Loading... 图片，此时，代理对象也会加载目标图片。当目标图片加载完毕后，ImageIconProxy 对象会将当前图片切换为目标图片。

如图 11.3 所示，可以通过创建 ImageIcon 类的子类获得一个代理类。ImageIconProxy 类定义了两个静态变量，分别包含 Absent 和 Loading... 两张图片：

```
static final ImageIcon ABSENT = new ImageIcon(
    ClassLoader.getSystemResource("images/absent.jpg"));

static final ImageIcon LOADING = new ImageIcon(
    ClassLoader.getSystemResource("images/loading.jpg"));
```

ImageIconProxy 的构造函数需要图片文件的名称作为参数来加载图片。当调用 ImageIconProxy 对象的 load() 方法时，会将当前图片切换到 LOADING 图片，并且新开启一个线程来加载目标图片。使用独立线程的目的是当图片加载时，不让应用程序一直等待而造成假死状态。load() 方法接收一个 JFrame 对象，当目标图片加载完毕后，会回调 run() 方法。以下是 ImageIconProxy.java 文件较为完整的代码：

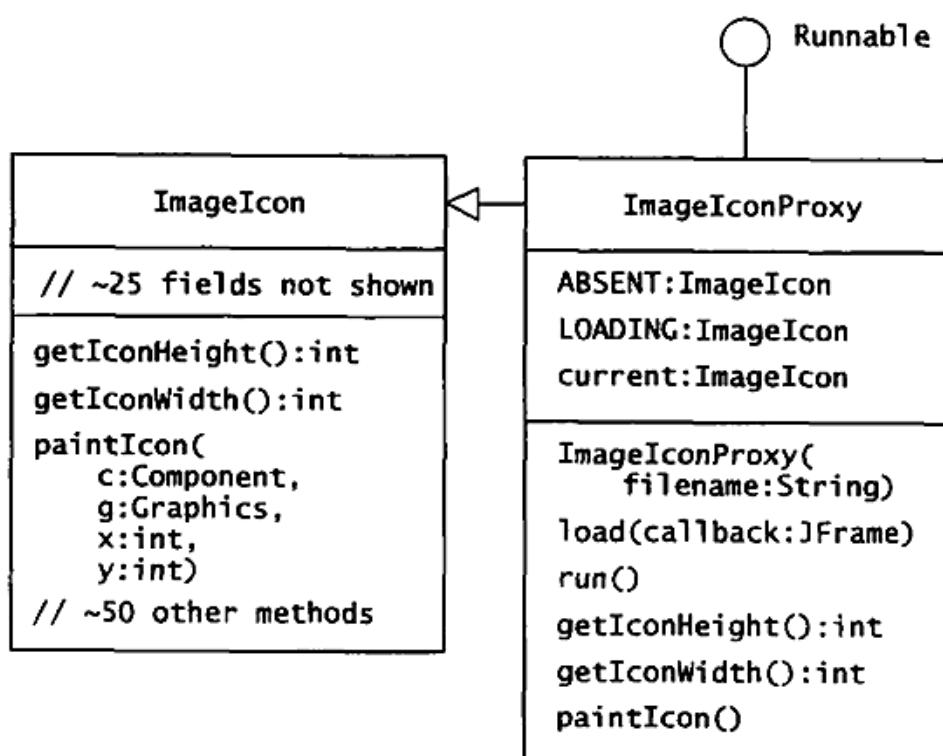


图 11.3 由于 ImageIconProxy 对象是一种 ImageIcon 对象，因此它可以替代 ImageIcon 对象

```

package com.oozinoz.imaging;
import java.awt.*;
import javax.swing.*;

public class ImageIconProxy
    extends ImageIcon implements Runnable {
    static final ImageIcon ABSENT = new ImageIcon(
        ClassLoader.getSystemResource("images/absent.jpg"));
    static final ImageIcon LOADING = new ImageIcon(
        ClassLoader.getSystemResource("images/loading.jpg"));
    ImageIcon current = ABSENT;
    protected String filename;
    protected JFrame callbackFrame;

    public ImageIconProxy(String filename) {
        super(ABSENT.getImage());
        this.filename = filename;
    }

    public void load(JFrame callbackFrame) {
        this.callbackFrame = callbackFrame;
        current = LOADING;
    }
}

```

```
callbackFrame.repaint();
new Thread(this).start();
}

public void run() {
    current = new ImageIcon(
        ClassLoader.getSystemResource(filename));
    callbackFrame.pack();
}

public int getIconHeight() { /* 挑战! */ }

public int getIconwidth() { /*挑战! */ }

public synchronized void paintIcon(
    Component c, Graphics g, int x, int y) {
    // 挑战!
}
}
```

挑战 11.1

`ImageIconProxy` 对象可以将对显示三个图片的调用转发给当前图片。请写出 `ImageIconProxy` 类的 `getIconHeight()`、`getIconwidth()` 和 `paintIcon()` 方法的实现代码。

答案参见第 321 页

假设你拿到了这个演示程序的代码，在创建真正的应用程序，而不仅仅是包含一个加载功能之前，需要进行设计评审，这样才能让设计中的问题显露出来。

挑战 11.2

`ImageIconProxy` 类并非一个设计良好的可重用组件。请指出设计中存在的两个问题。

答案参见第 322 页

当你在评审别人的设计时，既要理解别人的设计，又要持有自己的看法。开发者在运用特定设计模式时，常常会对模式的运用是否正确产生分歧。在该例子中，很明显使用了代理模式，但并不能说该模式的运用恰到好处。事实上，对于该例子有更好的设计。在使用代理模式时，使用必须得当。因为转发请求可能会造成一些问题，而这些问题在其他设计中可能没有。在阅读下一节时，你需要重新思考代理模式是否运用合理。

重新思考图片代理

对此，你可能会问设计模式是否有用。之前费尽心思地实现了一个模式，现在却需要将该模式去掉。事实上，这种现象在实际开发过程中十分常见。代码的作者在其他评审人员的帮助下，会重新思考，以改进原有设计。在实际生产过程中，设计模式可以帮助我们设计应用程序，也便于对我们的设计进行讨论。在 Oozinoz 公司的 `ImageIconProxy` 例子中，尽管不用模式会更简单，但模式也起到了自己的作用。

`ImageIcon` 类操作 `Image` 对象。比起将绘图请求转发给独立的 `ImageIcon` 对象，用 `ImageIcon` 对象来包装操作 `Image` 对象会更简单。图 11.4 展示了 `com.oozinoz.imaging` 包中的 `LoadingImageIcon` 类，该类除了构造函数外，仅有 `load()` 和 `run()` 两个方法。

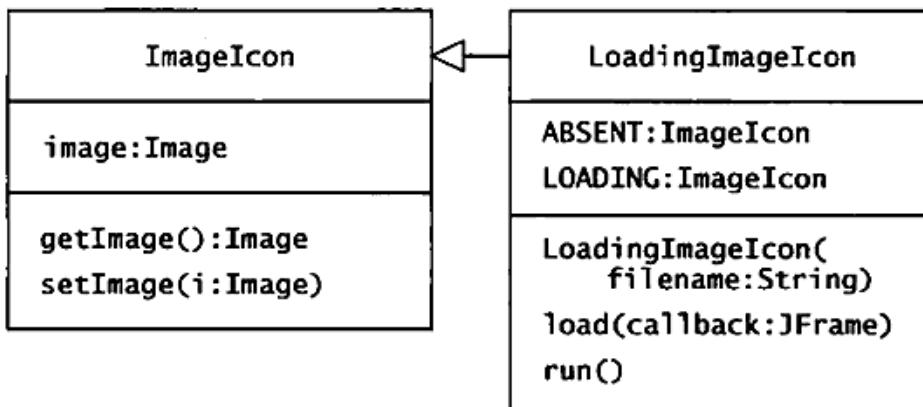


图 11.4 LoadingImageIcon 类的职责是切换 Image 对象

改进后的 `load()` 方法依然接收一个 `JFrame` 对象作为参数，用于在指定图片加载后进行回调。当 `load()` 方法执行时，会单独启动一个线程，用 `LOADING` 图片作为参数来调用 `setImage()` 方法，重绘图片显示窗体。`run()` 方法执行在一个单独的线程中，它会根据构造函数中的文件名创建一个新的 `ImageIcon` 对象，调用 `setImage()` 方法将该对象传入进去。最后重新绘制该窗口。

>LoadingImageIcon.java 的主要代码如下所示：

```
package com.oozinoz.imaging;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
public class LoadingImageIcon
    extends ImageIcon implements Runnable {
    static final ImageIcon ABSENT = new ImageIcon(
        ClassLoader.getSystemResource("images/absent.jpg"));
    static final ImageIcon LOADING = new ImageIcon(
        ClassLoader.getSystemResource("images/loading.jpg"));
    protected String filename;
    protected JFrame callbackFrame;

    public LoadingImageIcon(String filename) {
        super(ABSENT.getImage());
        this.filename = filename;
    }

    public void load(JFrame callbackFrame) {
        // 挑战!
    }

    public void run() {
        // 挑战!
    }
}
```

挑战 11.3

将 LoadingImageIcon 类中 load() 与 run() 方法的代码补齐。

答案参见第 322 页

修正后的代码降低了与 ImageIcon 之间的耦合，依赖于 getImage() 与 setImage() 方法，而不是方法转发机制。事实上，根本不存在这种转发机制：LoadingImageIcon 类虽然结构上不是一个代理，但本质上却是。

依赖于转发的代理模式会造成维护上的负担。当底层对象改变时，Oozinoz 公司将不得不更新代理。为了避免这个负担，通常应考虑代理模式的替代方案。但是，代理模式依然可能是

正确的选择，尤其是当需要获取的消息对象在另一台机器上执行时。

远程代理

如果需要调用的对象方法运行在其他机器上，不能直接调用该方法，这就需要另辟蹊径。可以在远程机器上打开一个套接字（Socket），并设计一些协议来与远程对象之间进行消息传输。理想情况下，这种方案可以使你像在本地一样传输消息。可以调用代理对象的方法，将对真实对象的调用转发给远程机器。事实上，这种方案业已实现，例如在 CORBA（公共对象请求代理体系结构）、ASP.NET（.NET 下的动态页面）以及 Java 远程方法调用（RMI）上。

RMI 使客户类可以很容易地获取代理对象，并且转发给另一台电脑的目标对象。企业版 JavaBeans（EJB）规范是一个非常重要的行业标准。RMI 属于其中一部分，因此，很有必要了解它。无论行业标准如何改进，我们可以预见，代理模式始终会应用在分布式计算中。而 RMI 为这种模式的实践提供了一个很好的范例。

在实践 RMI 前，你需要阅读几篇相关的文章，例如，*Java™ Enterprise in a Nutshell*（由 Flanagan 等人在 2002 年编写）。下面的例子并不会介绍 RMI 的用法，而是通过 RMI 应用程序展现代理模式及其价值所在。RMI 和 EJB 带来了许多新的设计理念。当然，简单地将所有对象都变成远程对象，是不能得到一个合理的系统的。我们无法深入讨论这些内容，只能浮光掠影地看看为何 RMI 是运用代理模式的好例子。

倘若你决定开始 RMI 的实践：用 Java 代码使一个对象方法可以在另一台机器上被调用。首先需要为远程访问类创建一个接口。作为一个实验性项目，假设你创建了一个独立于 Oozinoz 公司的 Rocket 接口：

```
package com.oozinoz.remote;
import java.rmi.*;
public interface Rocket extends Remote {
    void boost(double factor) throws RemoteException;
    double getApogee() throws RemoteException;
    double getPrice() throws RemoteException;
}
```

Rocket 接口继承自 Remote，该接口中的所有方法都声明会抛出 RemoteException 异常。这种做法的原因超出了本书的讨论范围，但是任何一本关于 RMI 的书都会涉及这一点。作为一台服务器，你的远程接口应该继承自 UnicastRemoteObject 类，如图 11.5 所示。

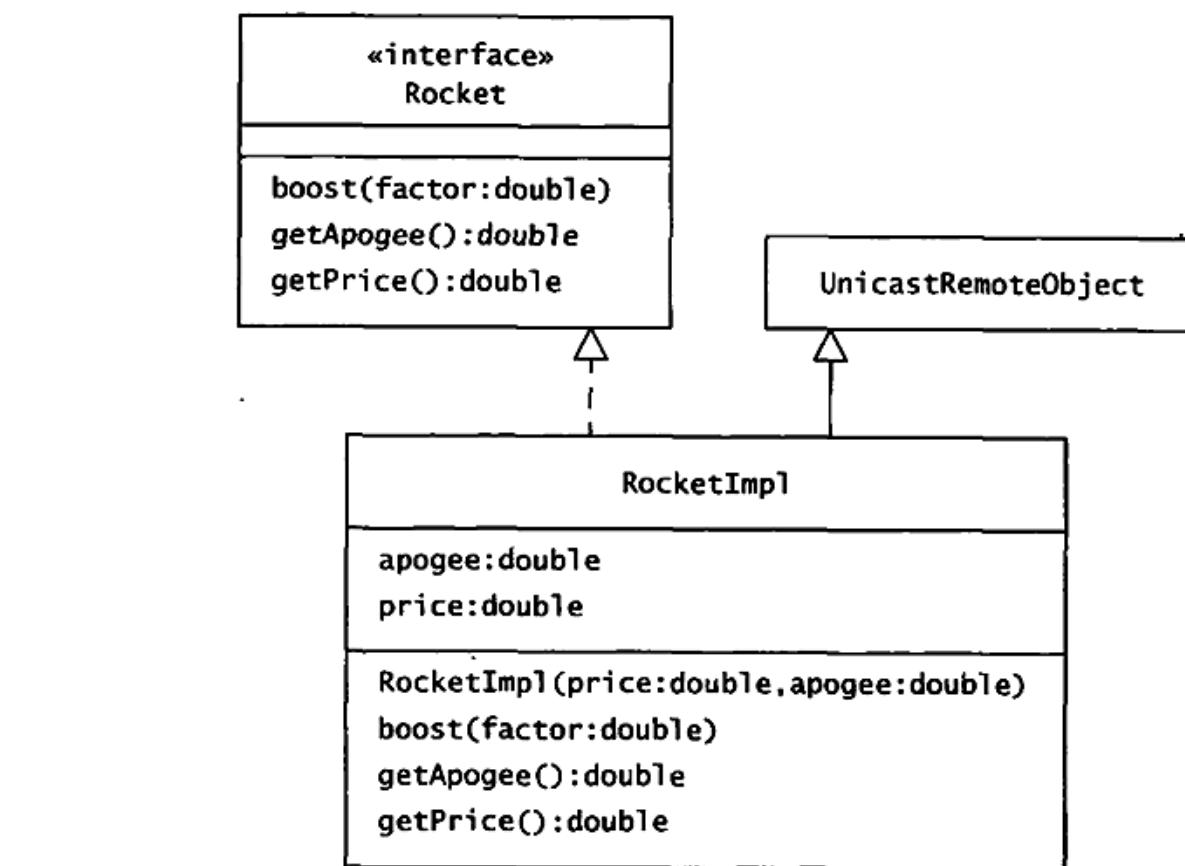


图 11.5 要使用 RMI，首先需要定义一个在计算机间传递消息的接口，然后创建 `UnicastRemoteObject` 类的子类来实现它

我们希望 `RocketImpl` 对象运行在服务器上，客户端可以通过代理访问服务器端的 `RocketImpl` 对象。`RocketImpl` 类的代码很简单。

```
package com.oozinoz.remote;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class RocketImpl
    extends UnicastRemoteObject
    implements Rocket {
    protected double price;
    protected double apogee;

    public RocketImpl(double price, double apogee)
```

```
throws RemoteException {  
    this.price = price;  
    this.apogee = apogee;  
}  
  
public void boost(double factor) {  
    apogee *= factor;  
}  
  
public double getApogee() {  
    return apogee;  
}  
  
public double getPrice() {  
    return price;  
}  
}
```

`RocketImpl` 的实例可以运行在一台机器上，运行在其他机器上的 Java 程序可以访问它。因此，客户端需要 `RocketImpl` 对象的代理。它必须实现 `Rocket` 接口，并且包含远程对象通信的额外特性。RMI 的一个重要好处是能自动创建这个代理类。为了产生代理类，需要把 `RocketImpl.java` 文件和 `Rocket.java` 接口文件替换到 RMI 注册目录下：

```
c:\rmi>dir /b com\oozinoz\remote  
RegisterRocket.class  
RegisterRocket.java  
Rocket.class  
Rocket.java  
RocketImpl.class  
RocketImpl.java  
ShowRocketClient.class  
ShowRocketClient.java
```

为了创建用于简化远程通信的 `RocketImpl` 存根，需要运行 JDK 提供的 RMI 编译器。

```
c:\rmi>rmic com.oozinoz.remote.RocketImpl
```

注意，`rmic` 的执行需要一个类名作为参数，而不是文件名。JDK 的早期版本将客户端的代码和服务器端的代码存放在不同的文件中。从版本 1.2 起，RMI 编译器将客户端与服务器端的

代码都创建在一个桩文件中。rmic 命令会创建一个 RocketImpl_Stub 类：

```
c:\rmi>dir /b com\oozinoz\remote  
RegisterRocket.class  
RegisterRocket.java  
Rocket.class  
Rocket.java  
RocketImpl.class  
RocketImpl.java  
RocketImpl_Stub.class  
ShowRocketClient.class  
ShowRocketClient.java
```

为了运行该对象，必须使用运行在服务器上的 RMI 注册程序注册该对象。rmiregistry 可执行程序是 JDK 的一部分。当运行该注册程序时，需要指定该注册程序侦听的端口号：

```
c:\rmi> rmiregistry 5000
```

在服务器上运行该注册程序后，可以创建并注册一个 RocketImpl 对象：

```
package com.oozinoz.remote;  
import java.rmi.*;  
public class RegisterRocket {  
    public static void main(String[] args) {  
        try {  
            // 挑战！  
            Naming.rebind(  
                "rmi://localhost:5000/Biggie", biggie);  
            System.out.println("Registered biggie");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

编译并运行该代码，程序会显示出注册的确认信息。

```
Registered biggie
```

你需要替换 RegisterRocket 类的“//挑战！”代码，该类会创建 biggie 对象，对火箭进行建模。main()方法中的其余代码注册了这个对象。关于 Naming 类机制的描述超过了本章的

讨论范围。当然，你必须拥有足够的信息来创建 `biggie` 对象。

挑战 11.4

用声明和实例化 `biggie` 对象的代码来替换“//挑战！”的内容。假定 `biggie` 建模火箭的价格是\$29.95，最高 820 米。

答案参见第 323 页

运行 `RegisterRocket` 程序，可以在服务器上创建一个 `RocketImpl` 对象 `biggie`。运行在另一台机器上的客户端只要有机权限访问 `Rocket` 接口和 `RocketImpl_Stub` 类，就可以访问 `biggie` 对象。如果只有一台机器，仍然可以测试 RMI，可以通过 `localhost`，而不是另一台主机访问服务器。

```
package com.oozinoz.remote;

import java.rmi.*;
public class ShowRocketClient {
    public static void main(String[] args) {
        try {
            Object obj = Naming.lookup(
                "rmi://localhost:5000/Biggie");
            Rocket biggie = (Rocket) obj;
            System.out.println(
                "Apogee is " + biggie.getApogee());
        } catch (Exception e) {
            System.out.println(
                "Exception while looking up a rocket:");
            e.printStackTrace();
        }
    }
}
```

程序运行时，会通过已经注册的“`Biggie`”名字来查找对象。该对象代表着 `RocketImpl`，而 `obj` 对象的 `lookup()` 方法将返回 `RocketImpl_Stub` 类的实例。`RocketImpl_Stub` 类实现了 `Rocket` 接口，因此可以将 `obj` 对象转换为 `Rocket` 接口的实例。`RocketImpl_Stub` 类继承自 `RemoteStub` 类，并且可以让该对象与服务器进行通信。

运行 `ShowRocketClient` 程序时，它将打印出“Biggie”火箭的高度。

`Apogee is 820.0`

通过代理，对 `getApogee()` 的调用会转发给运行在服务器上的 `Rocket` 接口的实现。

挑战 11.5

图 11.6 展示了 `getApogee()` 调用被转发的过程。最右边的对象用加粗标识，表明它和 `ShowRocketClient` 程序不在一起运行。请填上图中缺失的类名。

答案参见第 323 页

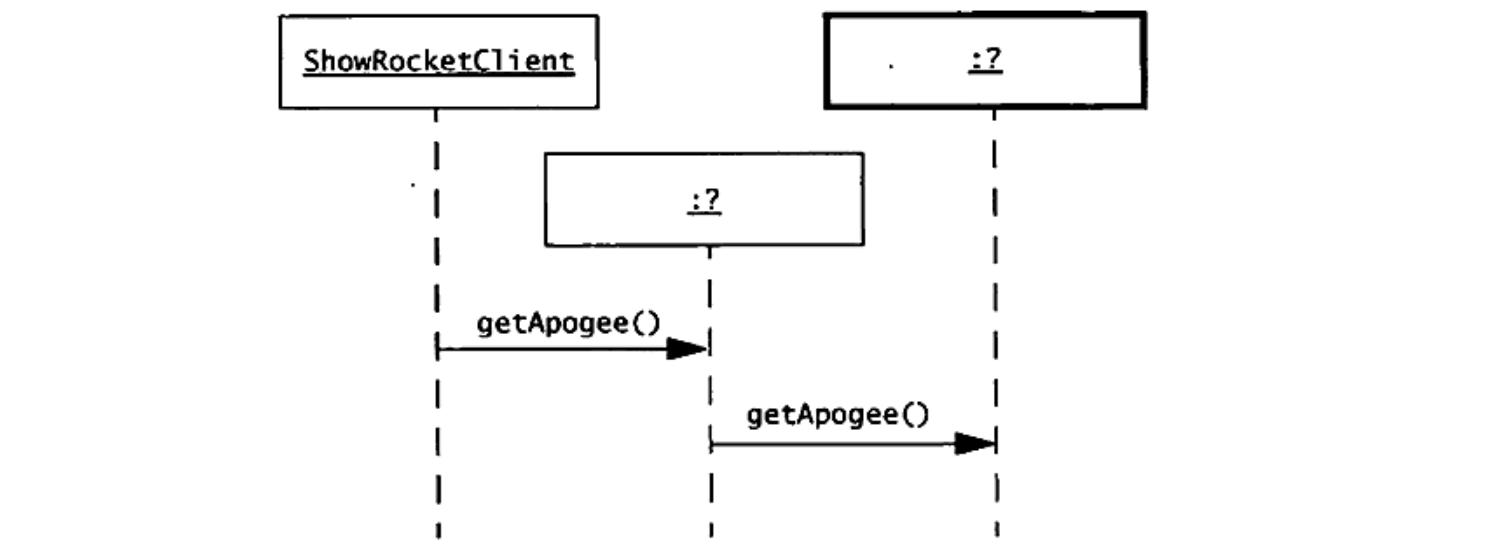


图 11.6 当你完成这幅图后，该图会展示基于 RMI 分布应用的消息流

使用 RMI 的好处在于，让客户端像与本地对象通信那样与远程对象通信。你为该对象定义了一个接口，并且让客户端和服务端都共用这个接口。RMI 提供了客户端与服务器端的通信机制，并将它们相互隔离。两端相互协作，提供了进程间的无缝通信。

动态代理

Oozinoz 公司的工程师们偶尔会遇到性能问题，但他们却并不希望就此而对代码进行大的改动。

Java 的动态代理特性可以轻易实现这一点。它允许我们使用代理对象来包装其他对象。你可以使用代理对象来拦截对被包装对象的请求，然后用代理再转发给这些被包装对象。你还可以在拦截调用的前后增加自己的代码。对动态代理施加限制可以防止包装任意对象。在正常条件下，动态代理使你能完整地控制被包装对象的操作。

动态代理依赖于对象类所实现的接口。接口中定义什么调用，代理就能拦截什么调用。如果你想拦截一个接口实现类的方法，需要使用动态代理去包装这个类的实例。

为创建动态代理，必须列出所要拦截的接口列表。幸运的是，可以通过询问所要包装的对象，获得这一列表，代码如下：

```
Class[] classes = obj.getClass().getInterfaces();
```

这行代码可以获得所要拦截的方法实现的所有接口。为了建立动态代理，还需要考虑两个因素：类加载器与当代理拦截调用时需要执行的行为类。对于接口列表，可以使用需要包装的相关对象来获取一个合适的类加载器：

```
ClassLoader loader = obj.getClass().getClassLoader();
```

最后一个需要的元素是代理对象自身。这个对象的类型必须实现 `java.lang.reflect` 包中的 `InvocationHandler` 接口。该接口声明了如下操作：

```
public Object invoke(Object proxy, Method m, Object[] args)  
    throws Throwable;
```

在对动态代理进行包装时，对包装对象的调用会转发给你所提供的 `invoke()` 方法。`invoke()` 方法会继续将方法调用转发给被包装对象。可以通过如下代码转发调用：

```
result = m.invoke(obj, args);
```

这行代码通过反射将目标调用转发给被包装对象。动态代理的美妙之处在于可以在转发调用之前或之后执行任何行为。

假设某个方法需要执行较长的时间，你可能希望记录一个报警日志，就可以用如下代码创建一个 `ImpatientProxy` 类：

```
package app.proxy.dynamic;  
  
import java.lang.reflect.*;
```

```
public class ImpatientProxy implements InvocationHandler {  
    private Object obj;  
  
    private ImpatientProxy(Object obj) {  
        this.obj = obj;  
    }  
  
    public Object invoke(  
        Object proxy, Method m, Object[] args)  
        throws Throwable {  
        Object result;  
        long t1 = System.currentTimeMillis();  
        result = m.invoke(obj, args);  
        long t2 = System.currentTimeMillis();  
        if (t2 - t1 > 10) {  
            System.out.println(  
                "> It takes " + (t2 - t1)  
                + " millis to invoke " + m.getName()  
                + "() with");  
            for (int i = 0; i < args.length; i++)  
                System.out.println(  
                    "> arg[" + i + "]: " + args[i]);  
        }  
        return result;  
    }  
}
```

它实现了 `invoke()` 方法，便于检查方法调用所需要的完整时间。如果执行时间过长，`ImpatientProxy` 类会打印出警告信息。

若要使用 `ImpatientProxy` 对象，就要用到 `java.lang.reflect` 包中的 `Proxy` 类。`Proxy` 类需要一组接口和一个类加载器，以及 `ImpatientProxy` 类的实例。为了简化动态代理的创建，可以为 `ImpatientProxy` 类增加如下方法：

```
public static Object newInstance(Object obj) {  
    ClassLoader loader = obj.getClass().getClassLoader();  
    Class[] classes = obj.getClass().getInterfaces();  
    return Proxy.newProxyInstance(  
        loader, classes, new ImpatientProxy(obj));  
}
```

这个静态方法为我们创建了一个动态代理。为了包装这个对象，`newInstance()`方法会获取对象的接口列表以及类加载器。该方法实例化了 `ImpatientProxy` 类，传递要包装的对象。所有这些准备的对象都会传递给 `Proxy` 类的 `newProxyInstance()` 方法。

返回的对象会实现被包装对象的所有接口。我们可以将其转换成它所实现的任意一种类型。

假设你正在维护一组对象，其中一些对象的某些方法运行得比较慢。为了找到这些运行缓慢的方法，可以包装一个 `ImpatientProxy` 对象。下面的代码展示了这个例子：

```
package app.proxy.dynamic;

import java.util.HashSet;
import java.util.Set;
import com.oozinoz.firework.Firecracker;
import com.oozinoz.firework.Sparkler;
import com.oozinoz.utility.Dollars;

public class ShowDynamicProxy {
    public static void main(String[] args) {
        Set s = new HashSet();
        s = (Set)ImpatientProxy.newInstance(s);
        s.add(new Sparkler(
            "Mr. Twinkle", new Dollars(0.05)));
        s.add(new BadApple("Lemon"));
        s.add(new Firecracker(
            "Mr. Boomy", new Dollars(0.25)));
        System.out.println(
            "The set contains " + s.size() + " things.");
    }
}
```

这段代码创建了一个 `Set` 对象来维护一些项目，然后代码调用了 `ImpatientProxy` 对象的 `newInstance()` 方法，通过强制转换的方式对集合进行包装。这样做的结果是使 `s` 对象的行为看起来像一个集合，唯一的例外是当 `ImpatientProxy` 对象的方法执行时间过长时，导致结果不能正常返回。例如，当程序调用集合的 `add()` 方法时，`ImpatientProxy` 对象会拦截该调用，并将该调用转发给实际集合对象，然后记录每个调用的耗时。

运行 `ShowDynamicProxy` 程序会产生如下的输出：

```
> It takes 1204 millis to invoke add() with
>     arg[0]: Lemon
The set contains 3 things.
```

`ImpatientProxy` 能够帮助我们找出集合中有哪些对象执行时间过长。目前是 `BadApple` 类的“Lemon”实例。`BadApple` 类的代码如下：

```
package app.proxy.dynamic;

public class BadApple {
    public String name;

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

    public boolean equals(Object o) {
        if (!(o instanceof BadApple))
            return false;
        BadApple f = (BadApple) o;
        return name.equals(f.name);
    }

    public int hashCode() {
        try {
            Thread.sleep(1200);
        } catch (InterruptedException ignored) {
        }
        return name.hashCode();
    }
    public String toString() {
        return name;
    }
}
```

`ShowDynamicProxy` 代码使用 `ImpatientProxy` 对象去监视集合的调用，而集合和 `ImpatientProxy` 对象间没有任何连接。一旦你写了一个动态代理类，只要对象实现了你想拦截的方法所属的接口，就可以使用该代理类去包装该对象。

在方法执行前后对调用进行拦截，并创建自己行为的思想属于面向切面编程（Aspect

Oriented Programming, AOP)。在 AOP 中，切面就是所谓的“建议”(advice，你要插入的代码)和“切入点”(point cut，对插入代码执行点的定义)的组合。AOP 的知识足够写一本书了，但你可以使用动态代理来尝试在各种对象间复用行为。

Java 中的动态代理可以让你使用代理来包装对象，还可以拦截对象方法的调用，以便在调用方法前后增加自己的行为。这种给任意对象增加可复用行为的方式，与面向切面编程十分相似。

小结

代理模式的实现为对象建立了一个占位符，用来管理对目标对象的访问。代理对象可以隔离目标对象的状态迁移，例如图片加载的时间变化。然而，运用代理模式会使得代理对象与被代理对象造成紧耦合。Java 中的动态代理提供了一种增加可复用功能的机制。倘若某个对象实现了你想要拦截的接口方法，就可以使用动态代理去包装这个对象，增加自己的逻辑或者替换被包装对象的代码。

第 12 章

职责链（Chain of Responsibility）模式

面向对象的开发者往往力求对象之间保持松散耦合，确保对象各自的责任具体并能最小化。这样的设计可以使得系统更加容易修改，同时降低产生缺陷的风险。从某种程度上讲，Java 语言有利于做出解耦的设计。客户端通常只能访问对象可见的接口，而不了解其实现细节。同时，客户端只需知道哪个对象具有它所需要调用的方法即可。当我们将若干对象按照某种层次结构进行组织时，客户端可能事先并不了解应该使用哪个类。这些对象要么执行该方法，要么将请求传递给下一个对象。

职责链模式的目的是通过给予多个对象处理请求的机会，以解除请求的发送者与接收者之间的耦合。

现实中的职责链模式

当一个人负责某项任务时，可以选择自己做或是让别人做，这就是现实中的职责链模式。以 Oozinoz 公司的情况为例，工程师会负责维护制造焰火的机器。

正如第 5 章中所介绍的那样，Oozinoz 公司把机器、生产线、车间、工厂模型看做“机器组件”。这种方法可以简化并递归实现某些操作，比如关闭某车间中所有的机器，或是简化工厂中工程师职责的模型。在 Oozinoz 公司，通常是特定的工程师负责某些特定的机器组件，尽管这种职责分配可能会分为不同的层次。

举例来说，一个复杂的机器，比如一个星形的冲床，可能由一个具体的工程师直接负责。然而，一个简单的机器可能没有直接负责的工程师，此时，就会由负责机器所在的生产线或车间的工程师来管理。

当客户对象查询负责机器的工程师时，希望能避免查询多个对象。因此，可以使用职责链模式，为每个机器组件分配责任人对象。图 12.1 说明了这种设计思路。

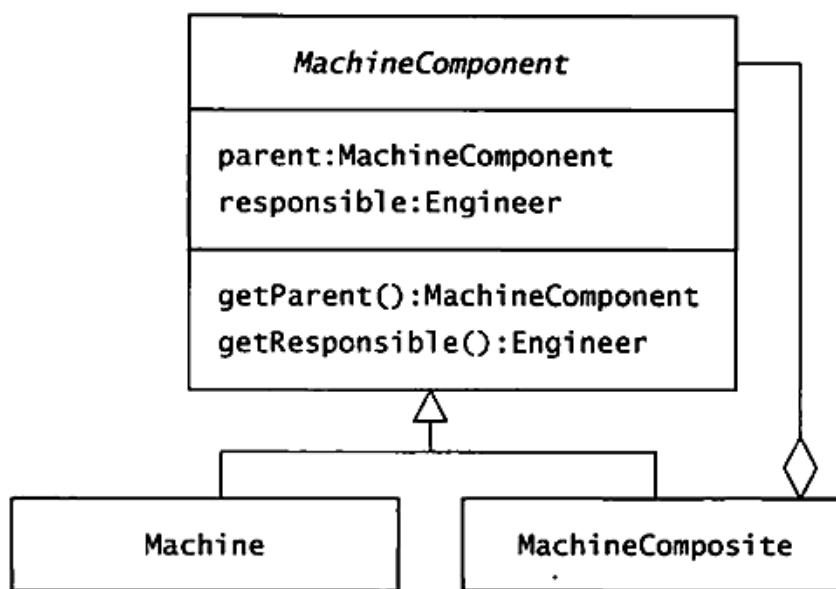


图 12.1 每个 Machine 或 MachineComposite 对象都有父对象 (parent) 和责任人对象 (responsible)，它们均继承自 MachineComponent 类

图 12.1 的设计思路允许每个机器组件跟踪负责它的工程师，但这并非强制性的要求。如果某台机器没有分配直接负责的工程师，可以把查询负责人的请求发送给父节点。一般而言，机器的父节点是生产线，生产线的父节点是车间，车间的父节点是工厂。在 Oozinoz 公司中，这个职责链的某处总有一位负责人。

这种设计思路的优点是机器组件的客户（代码）无须了解负责的工程师是如何分配的，客户（代码）可以查询任何机器组件的负责人。机器组件把客户（代码）与职责的分配标准相互独立。当然，在另一方面，这种设计思路也存在不合理的地方。

挑战 12.1

指出图 12.1 所示的设计思路中的两个缺陷。

答案参见第 323 页

职责链模式可以帮助我们简化客户端代码，尤其当客户端代码不清楚对象组中哪个对象负责处理查询请求时。如果事先没有建立职责链模式，也许要借助其他方法来简化之前复杂的设计。

重构为职责链模式

如果发现客户代码在发出调用请求前对调用做了判断，就应该通过代码重构来改善代码设计。若要应用职责链模式，必须事先明确这一组类对象是否支持该模式。例如，Oozinoz 中的机器组件有时提供对负责人的引用。把期望的操作添加到这组类对象的每个类中，并用链策略(Chaining Strategy)来实现该操作，以满足这一请求。

考虑 Oozinoz 代码库中对工具和工具车的建模。工具不属于 `MachineComponent` 类层次，但是在某些地方与机器类似。特别的，工具始终分配给工具车，并且工具车会包含一位负责的工程师。假设某可视化程序可以显示特定车间的所有工具和机器，并且提供弹出式信息以显示特定项的责任人。图 12.2 显示了在查询指定设备的责任人时所涉及的类。

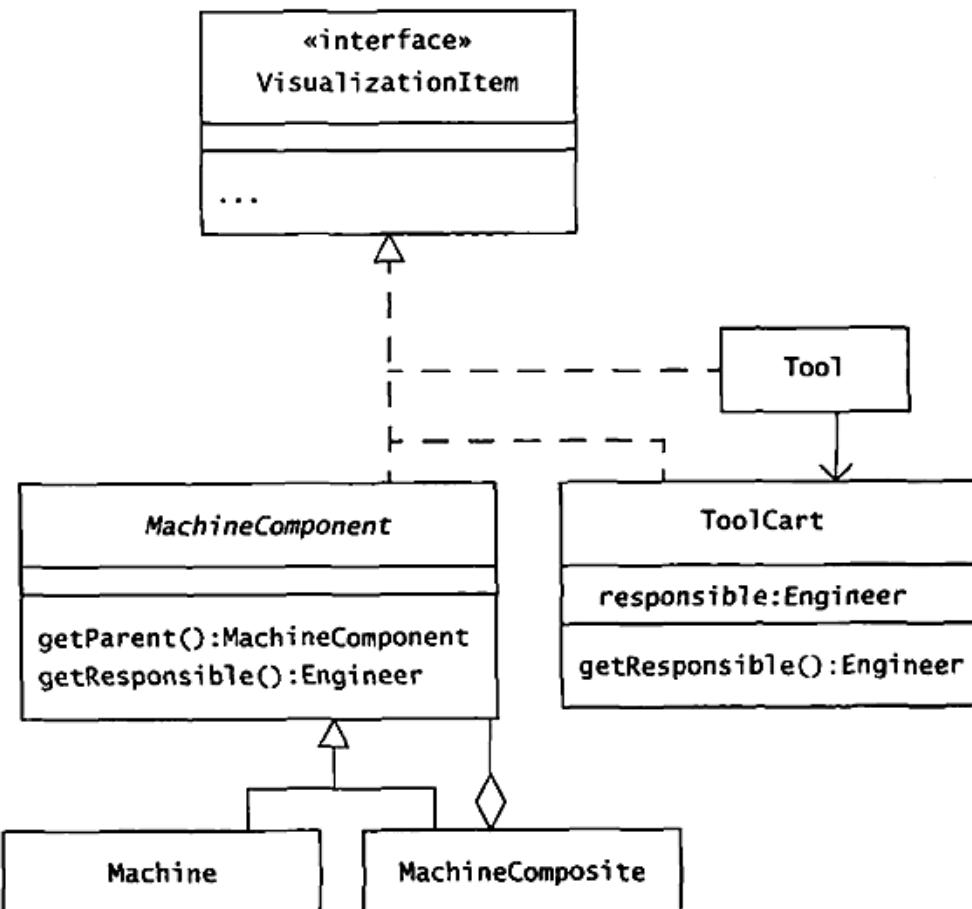


图 12.2 一个模拟环境下包含的设备项，包括各种机器、机器组合、工具以及工具车

`VisualizationItem` 接口具体指定了类需要的部分行为，以便于实现可视化，同时没有定义 `getResponsible()` 方法。事实上，并不是 `VisualizationItem` 的所有项都知道具体的负责人。当可视化程序需要决定哪个工程师对特定设备负责时，取决于所选的那个设备项。机器、机器组和工具车都有 `getResponsible()` 方法，而工具没有。为确定工具的责任工程师，程序代码必须查看这个工具属于哪个工具车，然后查看谁负责这个工具车。为确定指定设备项的负责人，应用程序的菜单代码使用了一些 `if` 语句和类型判断。这些特性表明重构可能有助于改善代码，具体代码如下：

```
package com.oozinoz.machine;

public class AmbitiousMenu {
    public Engineer getResponsible(VisualizationItem item) {
        if (item instanceof Tool) {
            Tool t = (Tool) item;
            return t.getToolCart().getResponsible();
        }
        if (item instanceof ToolCart) {
            ToolCart tc = (ToolCart) item;
            return tc.getResponsible();
        }
        if (item instanceof MachineComponent) {
            MachineComponent c = (MachineComponent) item;
            if (c.getResponsible() != null)
                return c.getResponsible();
            if (c.getParent() != null)
                return c.getParent().getResponsible();
        }
        return null;
    }
}
```

职责链模式的意图在于减轻调用者的压力，使它们无须了解哪个对象可以处理调用请求。在本例中，菜单是个调用者，它要求找到对应的负责人。根据当前设计，调用者必须了解哪个设备项具有 `getResponsible()` 方法。借助于职责链模式为所有模拟项提供责任人，我们可以借此来简化代码。这样可以将了解哪些对象知道其负责人的相关职责从菜单代码中转移到模拟项。

挑战 12.2

重新绘制类图 12.2，将 `getResponsible()`方法移入 `VisualizationItem` 接口，并将该方法加入到 `Tool` 类。

答案参见第 324 页

现在，菜单代码变得更加简单，可以直接通过调用 `VisualizationItem` 对象的方法来查找负责的工程师，具体如下：

```
package com.oozinoz.machine;

public class AmbitiousMenu2 {
    public Engineer getResponsible(VisualizationItem item) {
        return item.getResponsible();
    }
}
```

每个设备项的 `getResponsible()`方法也会变得更加容易实现。

挑战 12.3

为下面的类分别写出对应的 `getResponsible()`方法。

- A. `MachineComponent`
- B. `Tool`
- C. `ToolCart`

答案参见第 325 页

固定职责链

在为 `MachineComponent` 类编写 `getResponsible()`方法时，必须考虑到它的父对象可能为空。一种解决方法是，让每个 `MachineComponent` 对象都有一个非空父对象。这样做可以让

我们的对象模型更加紧凑。为实现这一目的，可以为 `MachineComponent` 类的构造函数添加一个参数来提供父对象（如果提供的父对象为空，甚至可以抛出一个异常，以便知道异常被捕获的位置）。此外，还要考虑到位于根部的对象——该对象没有父对象。一种合理的选择是创建一个 `MachineRoot` 类，并让该类继承 `MachineComposite` 类（而不是 `MachineComponent`）。为保证每个 `MachineComponent` 对象都对应一名负责人，可以这样做：

- `MachineRoot` 类的构造函数需要一个 `Engineer` 对象。
- `MachineComponent` 类的构造函数需要一个类型为 `MachineComponent` 的父对象。
- 只有 `MachineRoot` 使用 `null` 作为其父对象的值。

挑战 12.4

请写出图 12.3 中各个类的构造函数，从而保证每个 `MachineComponent` 对象都对应一名负责的工程师。

答案参见第 325 页

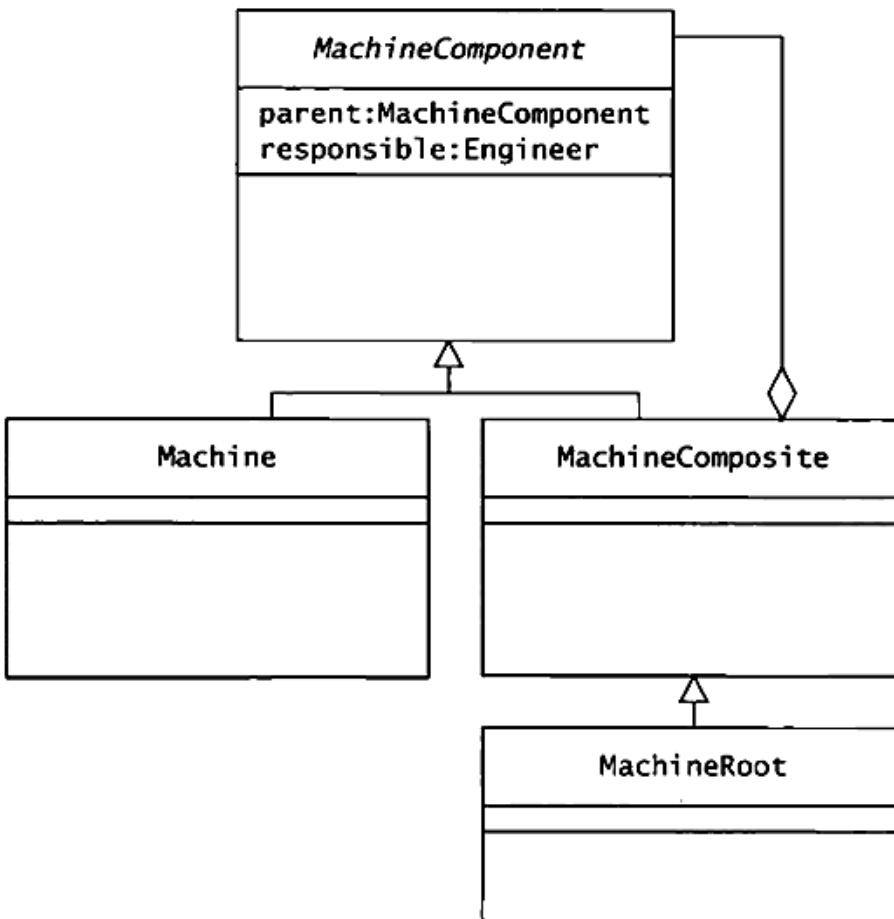


图 12.3 构造函数如何保证每个 `MachineComponent` 对象都对应一名负责的工程师

通过固定职责链，我们的对象模型变得更加健壮，代码也更为简洁。现在可以实现 `MachineComponent` 类的 `getResponsible()` 方法了。

```
public Engineer getResponsible() {  
    if (responsible != null)  
        return responsible;  
    return parent.getResponsible();  
}
```

没有组合结构的职责链模式

职责链模式需要一种策略来确定处理请求的对象的查询顺序。采用哪种查询策略取决于建模领域的背景知识。倘若对象模型存在某些类型的组合，如 Oozinoz 公司的机器类层次结构，这一情形就极为常见。不过，职责链模式也可以用于不带组合结构的对象模型。

挑战 12.5

举例说明职责链模式可以存在于哪些没有组合结构的对象链模型中。

答案参见第 326 页

小结

在运用职责链模式时，客户端不必事先知道对象集合中哪个对象可提供自己需要的服务。当客户端发出调用请求后，该请求会沿着职责链转发请求，直到找到提供该服务的对象为止。这就可以降低客户端与提供服务的对象之间的耦合度。

如果某个对象链能够应用一系列不同的策略来解决某个问题，如解析用户的输入，这时也可以应用职责链模式。该模式更常见于组合结构，它具有一个包容的层次结构，为对象链提供了一种自然的查询顺序。简化对象链和客户端的代码是职责链模式的一个主要优点。

第13章

享元（Flyweight）模式

享元模式在客户对象间提供共享对象，并且为共享对象创建职责，以便普通对象不需要考虑共享对象创建的问题。通常情况下，任何时候都只能有一个客户对象引用该共享对象。当某个客户对象改变该共享对象的状态时，该共享对象不需要通知其他客户对象，然而有时候可能需要让多个客户对象同时共享访问某个对象。

存在这么一种需要在多个客户对象间共享对象的场景：当你需要管理成千上万个小对象时，例如在线电子书中的字符对象。在这种场景下，出于性能的考虑，需要在不同的客户对象间安全地传输这些细粒度的对象。例如：某本书需要 A 对象，当有很多 A 对象时，就需要采用某种方法对其进行建模。

享元模式的意图是通过共享来有效地支持大量细粒度的对象。

不变性

享元模式让多个客户对象间共享访问限定数量的对象。为了实现这个目的，你必须要考虑到当某个对象改变了该共享对象的状态时，该状态变化会影响到每个访问它的对象。当多个客户对象要共享访问某个对象时，若要保证对象间不会相互影响，一种最简单而又常用的做法是，限制客户对象调用任何可能改变共享对象的方法。你可以通过将对象设置为不变（immutable）类型

来达到这一目的。然而，这样做会导致对象在创建后无法改变。Java中最常见的不可变对象是 `String` 类。当创建了一个 `String` 对象后，无论你还是其他客户对象都无法改变该对象的字符。

挑战 13.1

给出一个理由，表明 Java 语言的设计者为何将 `String` 对象设置为不可变？
如果你认为这不是一种明智的做法，也请给出理由。

答案参见第 327 页

若存在大量相似对象，而你可能需要共享访问它们，但它们可能并非一成不变。在这种情况下，使用享元模式的第一步是将对象中不可变的部分抽取出来，共享它们。

抽取享元中不可变的部分

对于 Oozinoz 公司，化学药品的使用程度就像文档中的字符一样普遍。工厂中的采购部门、工程部门、制造部门、安全部门都会关心成千上万的化学药品的流动情况。成批的化学药品都被建模到 `Substance` 类的实例中，如图 13.1 所示。

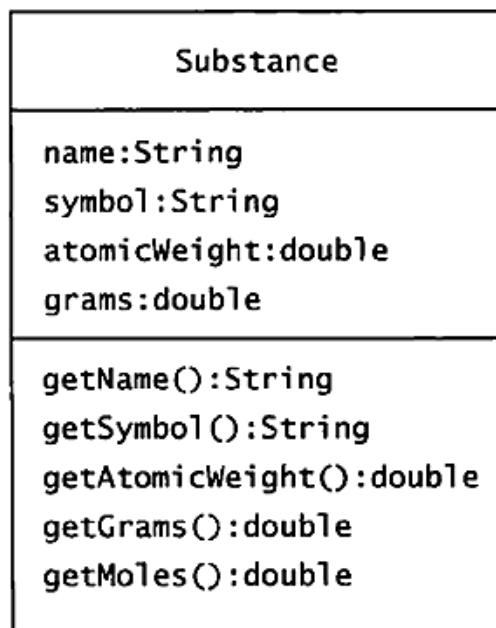


图 13.1 化学药品被建模成 `Substance` 类的实例

Substance 类有很多方法可以访问它的属性，还有 `getMoles()` 方法用来返回其摩尔数——即化学成分的分子数量。一个 **Substance** 对象代表特定的摩尔数。Oozinoz 公司使用 **Mixture** 类来建模化学药品的组成成分。例如，图 13.2 展示了黑火药的对象图。

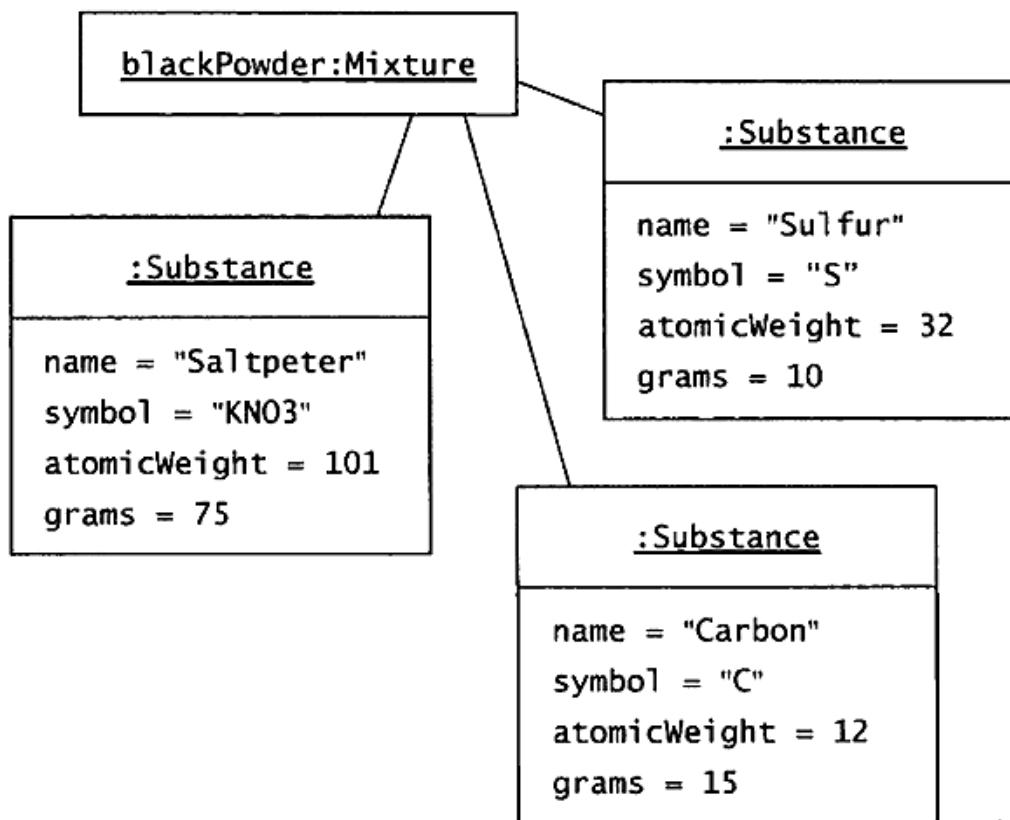


图 13.2 黑火药包含硝酸钾、硫黄以及碳粉

假设 Oozinoz 公司的化学药品越来越多，因而决定使用享元模式来减少程序中 **Substance** 对象的数量。为了将 **Substance** 对象变成享元模式，首先需要做的事情就是将类中不变的部分与变化的部分分离出来。假设你要重构 **Substance** 类，可以将不变的部分抽取到 **Chemical** 类中。

挑战 13.2

完成图 13.3 中的类图，给出重构后的 **Substance2** 类，以及一个新的、不可变的 **Chemical** 类。

答案参见第 327 页

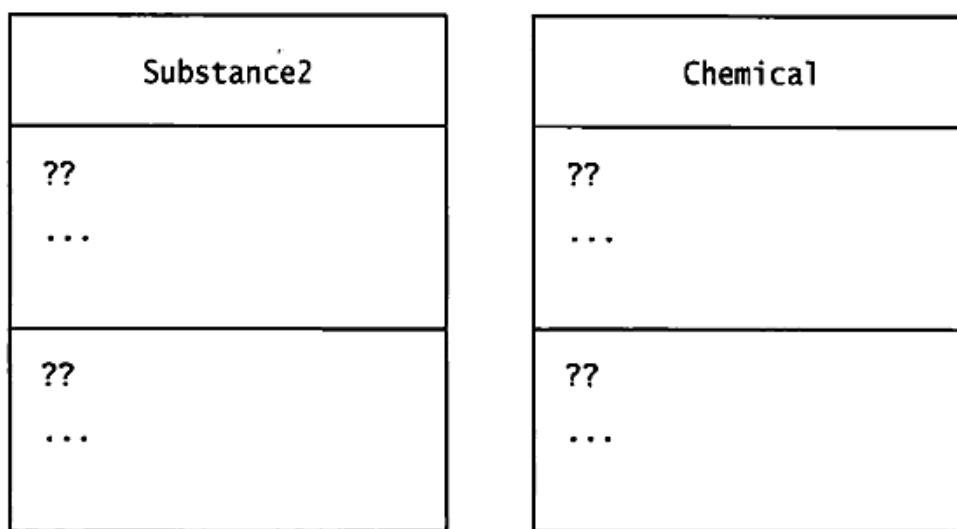


图 13.3 完成该图，将 Substance2 中不可变的部分抽取到 Chemical 类中

共享享元

抽出对象中不变的部分仅仅只完成了享元模式的一半。另一半包括创建享元工厂、实例化享元，以及让客户对象共享享元对象。我们还需要确保客户对象应使用享元工厂创建享元对象，而不是自己创建。

为了创建化学药品的享元对象，需要定义一个 `ChemicalFactory` 类作为享元工厂。该类包含一个静态方法，负责接收一个名称，返回对应的化学药品。你可以将化学药品存在一个哈希表中，在初始化工厂的时候创建它们。图 13.4 展示了 `ChemicalFactory` 类的设计。

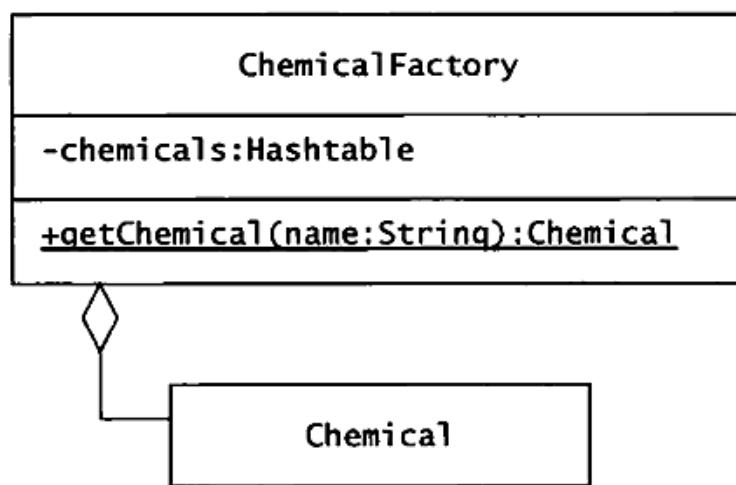


图 13.4 ChemicalFactory 类是一个返回 Chemical 对象的享元工厂

`ChemicalFactory` 类的代码使用静态构造函数将 `Chemical` 对象存储在哈希表中。

```
package com.oozinoz.chemical;
import java.util.*;

public class ChemicalFactory {
    private static Map chemicals = new HashMap();

    static {
        chemicals.put(
            "carbon", new Chemical("Carbon", "C", 12));
        chemicals.put(
            "sulfur", new Chemical("Sulfur", "S", 32));
        chemicals.put(
            "saltpeter", new Chemical("Saltpeter", "KN03", 101));
        //...
    }

    public static Chemical getChemical(String name) {
        return (Chemical) chemicals.get(name.toLowerCase());
    }
}
```

创建完化学药品工厂后，需要执行一些步骤确保让开发人员使用工厂创建享元对象，而不是直接创建 `Chemical` 对象。一种简单的方式是借助 `Chemical` 类的可见性。

挑战 13.3

如何使用 `Chemical` 类的访问修饰符来限制其他开发人员初始化 `Chemical` 对象？

答案参见第 328 页

访问修饰符还不能完全控制享元对象的实例化，需要确保 `ChemicalFactory` 是唯一一个能创建 `Chemical` 实例的类。为了达到这一级别的控制，需要把 `Chemical` 类定义为 `ChemicalFactory` 的内部类（参见 `com.oozinoz.chemical2` 包）。

要访问内嵌的类型，客户对象必须使用如下代码：

```
ChemicalFactory.Chemical c =
    ChemicalFactory.getChemical("saltpeter");
```

可以通过将 `Chemical` 定义为接口，内嵌类定义为该接口的实现，从而简化对内嵌类型的使用。`Chemical` 接口可以定义如下三个访问方法：

```
package com.oozinoz.chemical2;
public interface Chemical {
    String getName();
    String getSymbol();
    double getAtomicWeight();
}
```

客户对象永远不会引用该内部类，因此，可以把它定义成私有类型，以确保只有 `ChemicalFactory2` 类才能访问它。

挑战 13.4

完成如下 `ChemicalFactory2.java` 的代码：

```
package com.oozinoz.chemical2;
import java.util.*;

public class ChemicalFactory2 {
    private static Map chemicals = new HashMap();

    /* 挑战！ */ implements Chemical {
        private String name;
        private String symbol;
        private double atomicWeight;

        ChemicalImpl(
            String name,
            String symbol,
            double atomicWeight) {
            this.name = name;
            this.symbol = symbol;
            this.atomicWeight = atomicWeight;
        }

        public String getName() {
            return name;
        }

        public String getSymbol() {
            return symbol;
        }

        public double getAtomicWeight() {
            return atomicWeight;
        }
    }
}
```

```
    }

    public double getAtomicweight() {
        return atomicweight;
    }

    public String toString() {
        return name + "(" + symbol + ")" + [
            atomicweight + "]";
    }
}

/* 挑战! */
chemicals.put("carbon",
    factory.new ChemicalImpl("Carbon", "C", 12));
chemicals.put("sulfur",
    factory.new ChemicalImpl("Sulfur", "S", 32));
chemicals.put("saltpeter",
    factory.new ChemicalImpl(
        "Saltpeter", "KN03", 101));
//...
}

public static Chemical getChemical(String name) {
    return /* 挑战! */
}
```

答案参见第 328 页

小结

享元模式可以使你共享地访问那些大量出现的细粒度对象，例如字符、化学药品以及边界等。享元对象必须是不可变的，可以将那些需要共享访问，并且不变的部分提取出来。为了确保你的享元对象能够被共享，需要提供并强制客户对象使用享元工厂来查找享元对象。访问修饰符对其他开发者进行了一定的限制，但是内部类的使用使限制更进一步，完全限制了该类仅能由其外部容器访问。在确保客户对象正确地使用享元工厂后，你就可以提供对大量细粒度对象的安全共享访问了。



第3部分

构造型模式

第 14 章

构造型模式介绍

若要创建一个 Java 类，通常会为它提供多个构造函数^{译注1}。构造函数是有用的，尽管只有客户类知道该使用哪个构造函数以及传递什么参数来创建类。很多设计模式都可以解决常规构造函数无法处理的情况。在检查 Java 语言在构造函数设计上存在的不足之前，先来回顾一下常规的构造函数。

构造函数的挑战

构造函数是一些特殊的方法，在诸如访问修饰符、重载以及参数列表语法等诸多方面，构造函数和常规方法没有什么不同。然而，在语义与语法规则方面，构造函数的行为与使用受到更多的约束。

挑战 14.1

列出使用 Java 构造函数的 4 种约束。

答案参见第 330 页

译注1：若要延续 C++ 的传统，Constructor 应该被翻译为构造器，事实上，微软的 C# 官方文档中，也采用了这一术语。不过在 Java 语言中，似乎翻译为“构造函数”的更为普遍。因此，本书选择用“构造函数”来表示 Constructor。

在一些场景下，Java 提供默认的构造函数与行为。首先，如果某个类没有声明构造函数，Java 会提供一个默认的构造函数。默认构造函数是一个公共的构造函数，没有任何参数，方法体也没有执行语句。

第二个默认行为是，如果构造函数的声明没有显式调用 `this()` 或 `super()` 方法，Java 会自动调用不带参数的 `super()` 方法。这可能会导致出现一些意外的结果，就像如下的代码编译错误一样：

```
package app.construction;
public class Fuse {
    private String name;
    // public Fuse(String name) { this.name = name; }
}
```

与

```
package app.construction;
public class QuickFuse extends Fuse {}
```

只有当你删除了“//”注释标记后，这段代码才能正常编译。

挑战 14.2

请解释为何注释掉 `Fuse` 类的构造函数后，会发生编译错误？

答案参见第 330 页

初始化对象的常见方法是调用 `new` 操作符。当然，也可以使用反射。反射提供了能将类型与类型成员像对象一样操作的能力。即使不常使用反射，也可以提供依靠反射实现的程序逻辑，如下所示：

```
package app.construction;
import java.awt.Point;
import java.lang.reflect.Constructor;
public class ShowReflection {
    public static void main(String args[]) {
        Constructor[] cc = Point.class.getConstructors();
        Constructor cons = null;
        for (int i = 0; i < cc.length; i++)
```

```
    if (cc[i].getParameterTypes().length == 2)
        cons = cc[i];
    try {
        Object obj = cons.newInstance(
            new Object[] { new Integer(3), new Integer(4) });
        System.out.println(obj);
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
    }
}
```

挑战 14.3

`ShowReflection` 程序输出什么？

答案参见第 331 页

当使用其他手段很难或无法达到想要的结果时，反射能达成你的目标。若要了解使用反射的更多内容，请参见 *Java™ in a Nutshell*（由 Flanagan 在 2005 年编写）。

小结

一般情况下，你需要为自己开发的类提供构造函数使其能够被初始化。这些构造函数可能相互调用协作，并且类中的每个构造函数最终都会去调用超类的构造函数。调用构造函数的常规做法是使用 `new` 操作符，但也可以使用反射来初始化和使用对象。

超出常规的构造函数

在设计一个新类时，Java 构造函数的这些特性可以给你提供许多选择。然而，只有在类的用户知道该如何初始化类，以及传递所需参数时，构造函数才是有效的。例如，选择怎样的用

户界面，取决于程序运行在哪种显示器上，是手持设备，还是更大的显示器。或许，开发者知道使用哪一个类进行初始化，但他并不知道初始化所需的全部参数或格式。例如，开发者可能需要根据对象的静态方式或文本方式去构建对象。此时，常规的 Java 构造函数很难解决问题，你需要引入新的设计模式。

表 14.1 描述了各种模式的意图。

表 14.1 各种模式的意图

| 如果你的意图是 | 使用的模式 |
|----------------------------|--------|
| • 在请求创建对象前，逐步收集创建对象需要的信息 | 构建者模式 |
| • 决定推迟实例化类对象 | 工厂方法模式 |
| • 创建一族具有某些共同特征的对象 | 抽象工厂模式 |
| • 根据现有对象创建一个新的对象 | 原型模式 |
| • 通过包含了对象内部状态的静态版本重新构建一个对象 | 备忘录模式 |

每种设计模式的意图都是为了解决某种场景下的问题。构造型模式的设计就是为了让客户类不通过类构造函数来创建对象。例如，当你需要逐步获取对象的初始值时，则可能需要使用构建者模式。

构建者 (Builder) 模式

当你创建一个对象时，并不一定拥有创建该对象的全部信息。倘若需要逐步获得创建对象的信息，更方便的做法就是分步骤构建对象。这种情况通常发生在构建解释器和用户界面上。或者，当类的构造函数过于复杂，却对类的重点功能没有太大影响时，就可能需要让类变得更小一些。

构建者模式的意图是将类的构建逻辑转移到类的实例化外部。

常规的构建者

一个使用构建者模式的常见场景是，希望对象的数据嵌套在一段文本中。随着数据解析过程的发生，你需要在查询出数据的同时进行存储。无论解析器是基于 XML 还是通过手工方式，在一开始时，可能没有足够的数据来创建完整的目标对象。引入构建者模式，就可以提供一个中间对象来存储数据，直到准备好所有数据后，再根据中间对象的数据来构造目标对象^{译注1}。

假设 Oozinoz 公司除了制造焰火弹外，偶尔还提供焰火表演。旅行社会发送预约表演的请

译注1：在现在的设计方案中，构建者模式越来越倾向于以领域特定的方式，运用类似 Fluent Interface 模式，流畅而自然地表达构建的过程，并赋予客户端灵活组装的权利。例如要创建一个 `SqlStatement`，就可以通过如下代码来表示：`new SqlStatement("select * from Order").where().orderBy()`。这里的 `where()` 与 `orderBy()` 方法相当于后面提到的 `build()` 方法，实现了 `SqlStatement` 的组装过程。

求，它的邮件如下所示：

```
Date, November 5, Headcount, 250, City, Springfield,  
DollarsPerHead, 9.95, HasSite, False
```

或许，你会猜测这是一个出现在 XML 之前的协议，但是截至目前，它已经足够用了。

这个请求告诉我们潜在客户对焰火表演期望的时间和地点，以及能保证的最少人数及每位客人的服务费。在这个例子中，客户希望举办一场能容纳 250 人的焰火表演，并且会为每位客人支付 9.95 美元，或者一次性为我们的服务支付 2487.5 美元。旅行社同时还表示没有为演出提供地点。

现在的任务是将这个文本请求转换成它所代表的 `Reservation` 对象。可以创建一个空的 `Reservation` 对象，并且把它的参数设置成解析器转换的结果。这可能存在一个问题，转换后的 `Reservation` 对象不一定表示一个合法的请求。例如，可能在读完请求文本后才发现它丢失了日期。

可以使用一个 `ReservationBuilder` 类，来保证 `Reservation` 对象总是能代表合法的请求。`ReservationBuilder` 对象可以保存解析器解析出来的预订请求属性，在解析完毕后再去构建 `Reservation` 对象，并验证它的合法性。图 15.1 展示了该设计的类图。

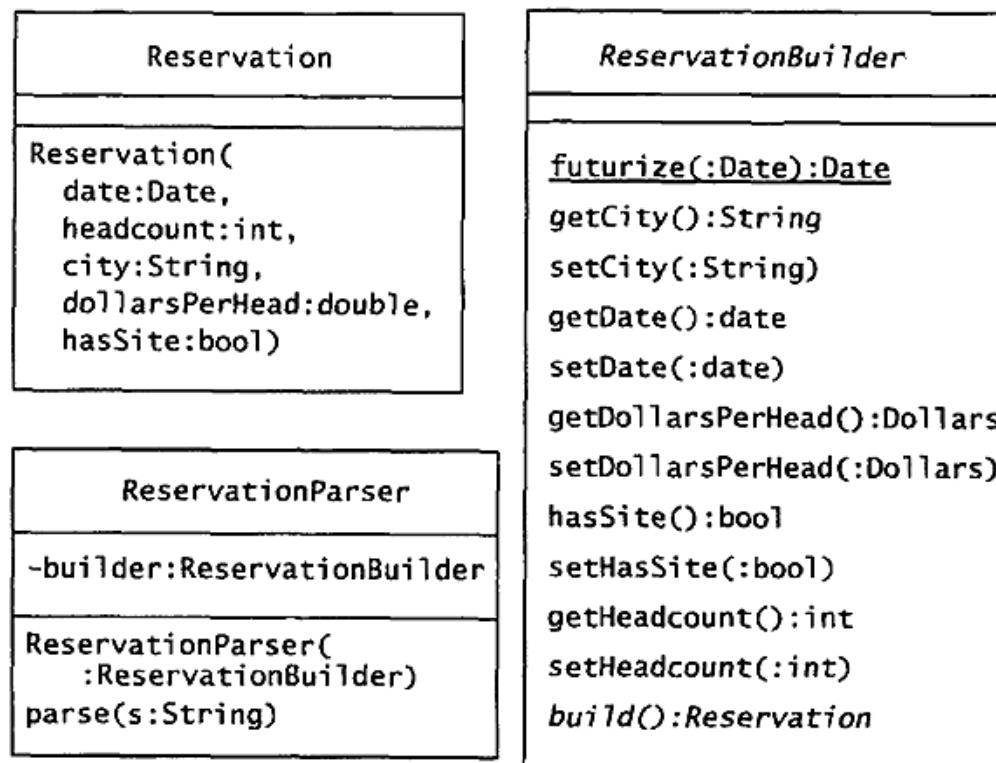


图 15.1 构建类 `ReservationBuilder` 将类的构建逻辑从业务类 `Reservation` 中移了出来，并且可以随着解析器的解析，逐步接受初始化参数来构建 `Reservation` 对象

`ReservationBuilder` 是一个抽象类，拥有一个 `build()` 方法。我们将会根据创建 `Reservation` 对象的方式不同，来创建 `ReservationBuilder` 类的子类。`ReservationParser` 类的构造函数接收一个构建者传递的信息。`parse()` 方法将信息从预订字符串中取出来，并传递给构建者。如下所示：

```
public void parse(String s) throws ParseException {
    String[] tokens = s.split(",");
    for (int i = 0; i < tokens.length; i += 2) {
        String type = tokens[i];
        String val = tokens[i + 1];

        if ("date".compareToIgnoreCase(type) == 0) {
            Calendar now = Calendar.getInstance();
            DateFormat formatter = DateFormat.getDateInstance();
            Date d = formatter.parse(
                val + ", " + now.get(Calendar.YEAR));
            builder.setDate(ReservationBuilder.futurize(d));
        } else if ("headcount".compareToIgnoreCase(type) == 0)
            builder.setHeadcount(Integer.parseInt(val));
        else if ("City".compareToIgnoreCase(type) == 0)
            builder.setCity(val.trim());
        else if ("DollarsPerHead".compareToIgnoreCase(type) == 0)
            builder.setDollarsPerHead(
                new Dollars(Double.parseDouble(val)));
        else if ("HasSite".compareToIgnoreCase(type) == 0)
            builder.setHasSite(val.equalsIgnoreCase("true"));
    }
}
```

`Parse()` 代码使用 `String.split()` 方法来对输入字符串进行词法分析。代码中期望的预订字符串是由逗号进行分隔的，每一部分都包含了类型和值信息。`String.compareToIgnoreCase()` 方法忽略待比较字符的大小写。当解析器读到“`date`”一词时，它会将随后出现的值存起来。`futurize()` 方法将年份移到日期的前面，来确保类似“`November 5`”的日期会被解析为 11 月 5 日。在阅读代码的过程中，可能会注意到解析器需要多次重新遍历预订字符串。

挑战 15.1

`split(s)` 调用使用正则表达式，将用逗号分隔的预订信息分隔成若干独立的字符串。请为这个正则表达式提供一个改进的方案，或者一个全新的方法，来使解析器更好地识别预订信息。

答案参见第 331 页

在约束条件下构建对象

你需要确保非法的 `Reservation` 对象永远不会被创建出来。针对这个例子，假设每个预订信息都必须有一个非空的日期和城市。假设 Oozinoz 公司的业务规则规定在人数低于 25 人，或者总费用少于 495.95 美元时，将不进行演出。我们需要将这些限制记录到数据库中。但是就目前而言，可以使用 Java 代码写出这些常量。如下所示：

```
public abstract class ReservationBuilder {  
    public static final int MINHEAD = 25;  
    public static final Dollars MINTOTAL = new Dollars(495.95);  
    // ...  
}
```

为了避免在请求不合法时创建 `Reservation` 实例，可能会在 `Reservation` 类的构造函数中进行业务逻辑检查和异常处理。但是 `Reservation` 对象一旦被创建，这些逻辑相对于 `Reservation` 本身的业务对象来讲就没用了。此时，引入一个构建者，并将 `Reservation` 的构建逻辑移植到构建者对象，将会使 `Reservation` 类本身变得简单，使其更加关注自身的业务逻辑。引入构建者还能使 `Reservation` 对象对传入的不同参数做出不同的反应。最终，将 `Reservation` 的构建逻辑移植到 `ReservationBuilder` 子类中，并在解析器解析预订字符串的过程中逐步创建 `Reservation` 实例。图 15.2 展示了 `ReservationBuilder` 的两个子类对不同的错误参数进行了不同的错误处理。

图 15.2 给出了构建者模式的一大优势：通过将 `Reservation` 类的构建逻辑抽取出来，就可以将这部分逻辑看做一个完整的任务，甚至可以为该逻辑创建一个独立的层级关系。不同的

构建逻辑对预订业务几乎没有影响。例如，在图 15.2 中，构建者之间的区别在于是否抛出 `BuilderException` 异常。使用构建者的代码如下所示。

```
package app.builder;
import com.oozinoz.reservation.*;

public class ShowUnforgiving {
    public static void main(String[] args) {
        String sample =
            "Date, November 5, Headcount, 250, "
            + "City, Springfield, DollarsPerHead, 9.95, "
            + "HasSite, False";
        ReservationBuilder builder = new UnforgivingBuilder();
        try {
            new ReservationParser(builder).parse(sample);
            Reservation res = builder.build();
            System.out.println("Unforgiving builder: " + res);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

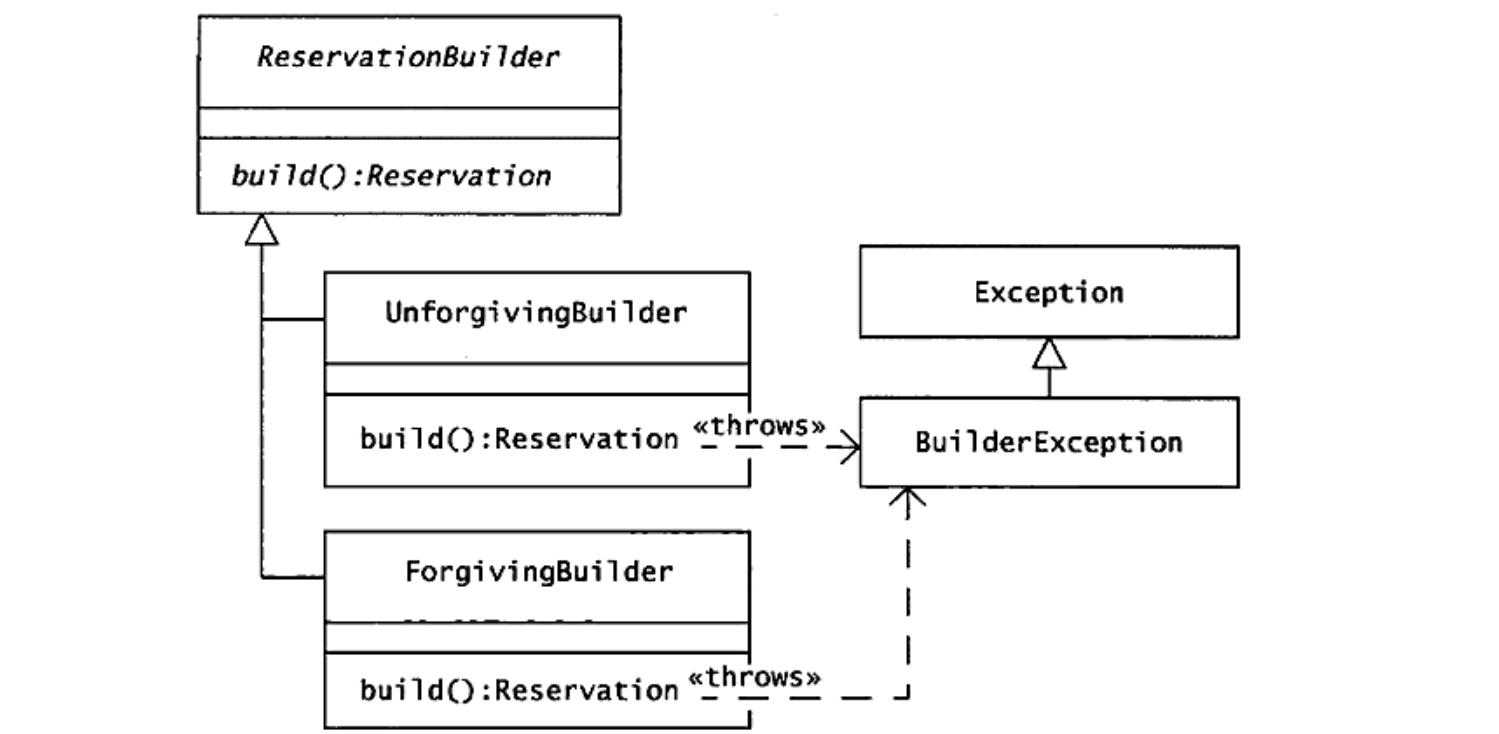


图 15.2 在面对不同的错误参数时，构建者对象会进行不同的错误处理

运行该程序将输出这个 `Reservation` 对象：

```
Date: Nov 5, 2001, Headcount: 250, City: Springfield,  
Dollars/Head: 9.95, Has Site: false
```

当给定一个预订字符串时，代码会初始化一个构建者和一个解析器，并会让解析器去解析字符串。当解析器读到预订的属性值时，会使用构建者的 `set` 方法将该值传递给构建者。

转换结束后，代码会要求构建者构建一个合法的预订。当出现异常时，这个例子会简单地打印出异常信息。而在实际应用中，我们可能需要采取一些行动来处理异常。

挑战 15.2

当日期或时间为空时，或者总人数太少，再或者演出费太低时，`UnforgivingBuilder` 类的 `build()` 方法都将抛出异常。请根据这些约束条件写出 `build()` 方法的实现。

答案参见第 332 页

可容错的构建者

`UnforgivingBuilder` 类会拒绝任何信息不完整的请求。对请求中缺失的信息做出合理的变通方案，可能是一个更好的业务规则。

当预订请求缺少总人数信息时，Oozinoz 公司的分析员会要求为该请求提供一个最少人数。类似的，当缺少单人的金额值时，构建者应该为该属性设置一个合理的值。这些需求都很简单，但是这种设计却需要一些技巧。例如当预订字符串包含了单人的金额值，却未指定总人数时，构建者应该具有什么样的行为呢？

挑战 15.3

为 `ForgivingBuilder.build()` 方法编写一个规则，重点是构建者如何处理缺失的人数或者单人的金额值。

答案参见第 333 页

挑战 15.4

请写出 `ForgivingBuilder` 类的 `build()` 方法的实现。

答案参见第 333 页

`ForgivingBuilder` 与 `UnforgivingBuilder` 类保证你创建的 `Reservation` 对象都是合法的。这个设计还有一个好处，当构建一个预订对象时，如果出错，该设计可以让你有足够的弹性来处理错误。

小结

构建者模式将复杂对象的构建逻辑从对象本身抽离了出来，这样能够简化复杂的对象。构建者关注目标类的构建过程，目标类关注合法实例的业务本身。这样做好处是在实例化目标类前，确保得到的是一个有效的对象，并且不会让构建逻辑出现在目标类本身。构建者构建类对象的过程通常是分步骤的，这使得该模式通常被应用于解析文本以创建对象的场景。

第 16 章

工厂方法（Factory Method）模式

在创建类时，通常可以同时定义多个构造函数，然后让它们创建类的实例。然而有时候，客户代码虽然需要某个对象，却并不关心或者不需要关心这个对象究竟是由哪个类创建而来的。

工厂方法模式的意图是定义一个用于创建对象的接口，并控制返回哪个类的实例。

经典范例：迭代器

迭代器（Iterator）模式提供了用于顺序访问容器中的各个元素的方法（参见本书第 28 章）。但是，迭代器实例却经常需要运用工厂方法模式来创建。Java JDK 1.2 引进了一个容器类接口，其中包含了一个 `iterator()` 方法，所有容器类均实现了该方法，它使得调用者并不知道 `iterator` 对象是由哪个类实例化的。

使用 `iterator()` 方法可以创建一个对象，顺序地返回容器内的所有元素。例如，如下代码创建了一个列表，并打印了列表中包含的内容：

```
package app.factoryMethod;
import java.util.*;

public class ShowIterator {
    public static void main(String[] args) {
```

```
List list = Arrays.asList(  
    new String[] {  
        "fountain", "rocket", "sparkler"});  
  
Iterator iter = list.iterator();  
while (iter.hasNext())  
    System.out.println(iter.next());  
}  
}
```

挑战 16.1

这段代码中的 `Iterator` 对象的实际类型是什么？

答案参见第 334 页

工厂方法模式使得客户代码无须关心使用哪个类的实例。

识别工厂方法

你可能认为任何可以创建或返回一个新对象的方法都是“工厂方法”。然而，在面向对象编程中，方法返回新对象是非常普遍的，但是并非每个这样的方法都应用了工厂方法模式。

挑战 16.2

说出 Java 类库中两个可以返回新对象的常用方法。

答案参见第 335 页

事实上，并非每个能创建并返回一个新对象的方法，都是工厂方法模式的实例。工厂方法模式不仅要求有一个能够创建新对象的方法，还要让客户代码无须了解具体实例化的类。工厂方法模式通常包含了若干类，这些类实现了相同的操作，返回了相同的抽象类型，然而这些操作的内部，实际上却实例化了不同的类，并且，这些类都实现了上述抽象类型。当客户代码请求一个新对象时，这个新对象该由哪个类实例化，取决于工厂对象接收创建请求时的行为。

挑战 16.3

类 `javax.swing.BorderFactory` 看似运用了工厂方法模式。请说明工厂方法模式的意图和 `BorderFactory` 类的意图有何不同。

答案参见第 335 页

控制要实例化的类

通常，客户代码使用类的某个构造函数来实例化类。但是有时候，客户代码并不知道使用哪个类去创建它所需要的对象。比如，在迭代器模式中，客户代码需要的迭代器的类取决于客户代码想要遍历的容器的类型。这种应用很常见。

假定 Oozinoz 公司允许顾客通过借贷形式购买焰火产品。在早期的信贷授权系统中，你开发一个名为 `CreditCheckOnline` 的类，用于在线核对系统是否允许为客户提供信用额度。

开始开发时，你意识到信贷代理机构有时并不在线。此时，项目分析者认为你需要为呼叫中心的代理建立一个对话框（传递信贷请求），通过设置一些问题使其自动做出信贷决定。因此，你创建了 `CreditCheckOffline` 类，使它能够根据某些规定被使用。最初，你的类设计如图 16.1 所示。其中，`creditLimit()` 方法接收客户的身份证号码，返回此客户的信贷限额。

有了图 16.1 中设计的类，不管信贷代理机构是否在线，你都可以为客户提供信贷限额信息。现在的问题是，这些类的使用者需要知道究竟实例化了哪个类。然而，事实上却只有你知道信贷代理机构是否在线。

在这种场景下，需要把创建对象的工作转交给一个接口，来完成实例化的控制。一种解决方案是让这两个类同时实现一个标准接口，并创建一个工厂方法来返回该接口的某个实例。具体做法如下：

- 创建一个名为 `CreditCheck` 的接口，使其包含 `creditLimit()` 方法。

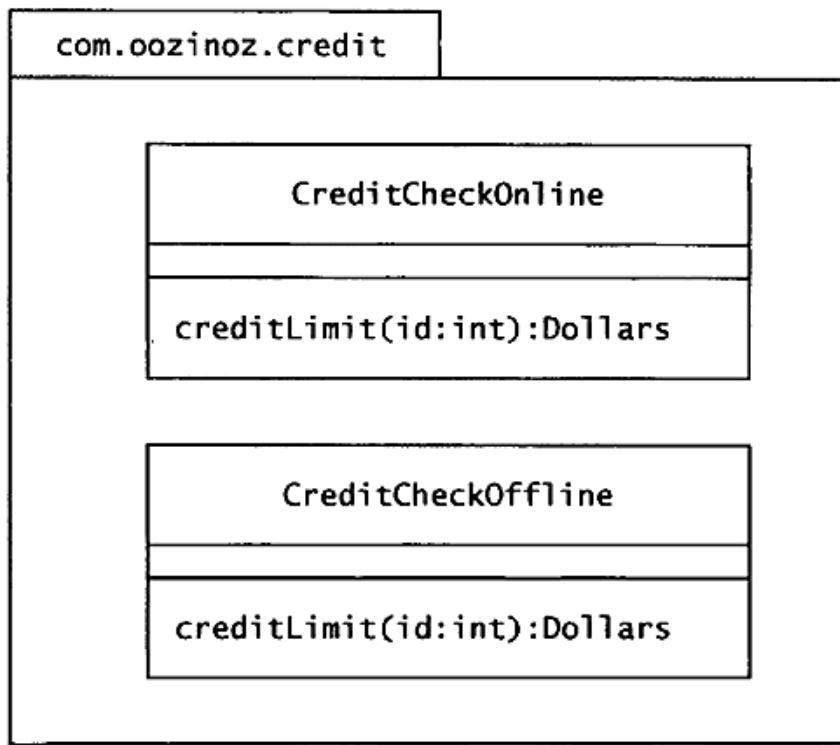


图 16.1 你可以判断信贷代理机构是否在线，从而决定实例化图中的哪一个类，以此核对客户的信贷信息

- 使所有信贷核对类声明实现 `CreditCheck` 接口。
- 创建一个名为 `CreditCheckFactory` 的类，此类包含一个名为 `createCreditCheck()` 的方法，并返回一个类型为 `CreditCheck` 的对象。

在实现 `createCreditCheck()` 方法时，通过掌握的信贷代理机构是否在线的信息，你可以决定究竟实例化哪一个类。

挑战 16.4

为这个新设计设计一个类图，使其不仅可以创建信贷核对对象，还可以确定由哪个类去实例化对象。

答案参见第 335 页

通过应用工厂方法模式，使用此服务的用户可以通过调用 `createCreditCheck()` 方法获得一个信贷核对的对象，这样不管信贷代理机构是否在线都可以满足顾客的服务请求。

挑战 16.5

假定 `CreditCheckFactory` 类有一个名叫 `isAgencyUp()` 的方法，它确定信贷代理机构是否在线，请写出 `createCreditCheck()` 的代码。

答案参见第 336 页

并行层次结构中的工厂方法模式

在使用并行层次结构对问题域进行建模时，常常会使用工厂方法模式。一个并行层次结构是一对类层次结构。其中，一个类在一个层次，与其相关的类在另一层次。当将现有类层次结构的部分行为移出该层次后，并行层次结构就会显露出来。

第 5 章的合成模式介绍了焰火弹的制造，Oozinoz 公司就是使用图 16.2 所示类图的机器生产这些焰火弹的。

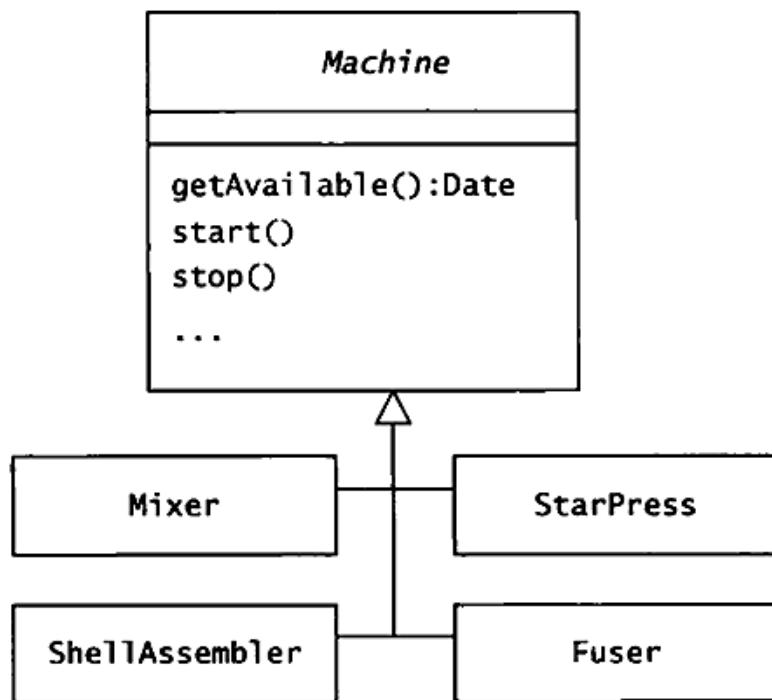


图 16.2 Machine 层次结构包含了控制物理机器和规划的逻辑

要生产一个焰火弹，需要混合相关的化学原料，然后将这些化学原料放入一个星形的压力

机中，这样混合过的原料就被挤压成了一个个星形的混合物。接着，使用一个外壳装备器，把这些星形的混合物装到一个具有黑色粉末内核的壳内，并且放置在推进药的上方。最后，再使用一根导火线插入到焰火弹中，一旦导火线被点燃，推进药和内核都会被引燃。

假定你想用 `getAvailable()` 方法来预测机器完成加工的时间，从而决定是否可以执行接下来的工作。该方法需要某些私有方法的支持，总体而言，就是为每个机器类添加一定数量的逻辑。但是这些规划逻辑不应该添加到 `Machine` 的类层次结构中，更好的方式是独立出 `MachinePlanner` 层次结构。对于大多数机器类型而言，需要一个独立的计划类；但是，其他的工作总是需要用到混合器和导火线。此时，就可以使用 `BasicPlanner` 类。

挑战 16.6

完成图 16.3 所示的 Machine/MachinePlanner 并行层次结构图。

答案参见第 336 页

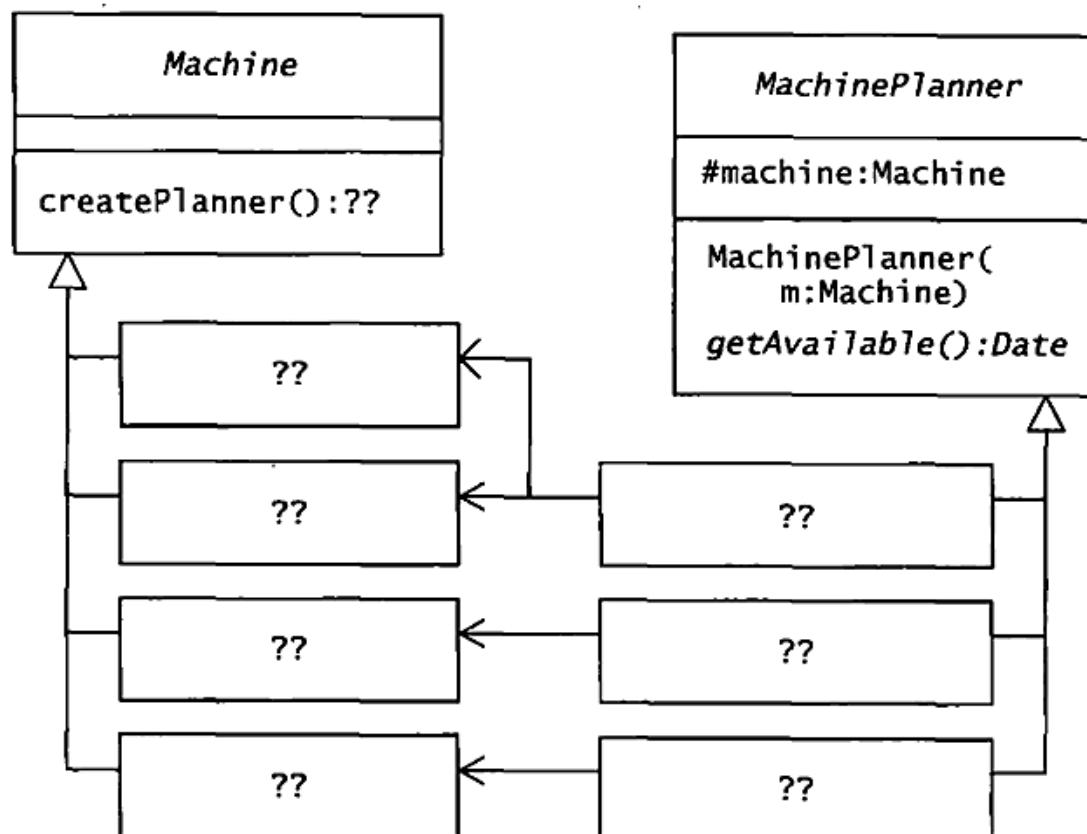


图 16.3 通过将规划逻辑移植到并行层次结构中，可以缩减 Machine 类的层次结构

挑战 16.7

为 `Machine` 类写一个 `createPlanner()` 方法，使其返回一个 `BasicPlanner` 对象，并为 `StarPress` 类编写 `createPlanner()` 方法。

答案参见第 337 页

小结

工厂方法模式的意图是让服务的提供者确定实例化哪个类，而不是客户代码。这种模式在 Java 类库中极为常见，比如 `Collection` 接口的 `iterator()` 方法。

当你不想让客户代码决定实例化哪个类时，常常可以运用工厂方法模式。此外，工厂方法模式还可用于当客户代码不知道它需要创建哪个对象类的时候，例如客户代码无法获知信贷机构是否在线的情况。另外，在并行类层次结构中使用该模式可以避免类的规模过于庞大。工厂方法模式可以根据一个类层次中的子类，确定另一个相关层次中哪一个类被实例化，从而建立对应的并行层次结构。

抽象工厂（Abstract Factory）模式

在创建对象时，有时会指定具体的类去实例化一个对象。可以使用工厂方法模式来定义一个外部方法以决定实例化哪个类。但有时候，控制实例化哪个类的因素可能与很多类息息相关。

抽象工厂模式又名工具箱，其意图是允许创建一族相关或相互依赖的对象。

经典范例：图形用户界面工具箱

GUI 工具箱是抽象工厂模式的一个典型范例。一个 GUI 工具箱就是一个对象，也是一个抽象工厂，它为用户提供各种 GUI 控件，并为控件提供一些基本的用户界面。它可以决定一些控件，例如按钮、文本区域等的实现方式，还可以设置具体的界面风格，比如背景颜色、外观以及控件家族中一些 GUI 设计的其他方面。当然，你可以为整个系统建立一个统一的界面，但是，也可以使用不同的界面风格反映应用程序版本的改变，体现公司标准图像的变化，甚至一些日常的简单改进。抽象工厂实现界面风格的多样化，使得应用程序更加容易理解和使用。图 17.1 所示的 Oozinoz 公司的 UI 类设计思路是一个很好的例子。

UI 类的子类可以重写用户控件工厂中的任何元素。GUI 对象是 UI 类的一个实例，创建该实例的应用程序可以通过在 UI 类的子类中创建不同的界面风格。例如，Oozinoz 公司使用 visualization 类去帮助工程师布局一个新的设备生产线。该可视化界面如图 17.2 所示。

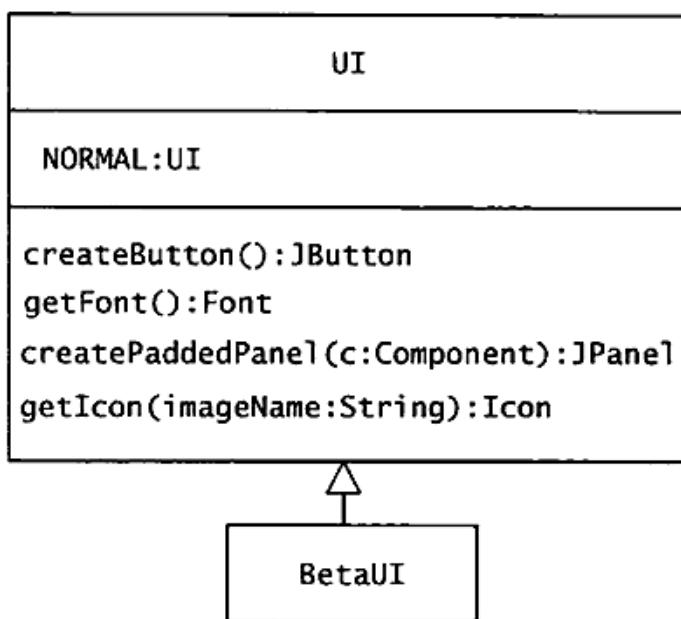


图 17.1 UI 类和 BetaUI 类的实例化对象是创建 GUI 控件族的抽象工厂

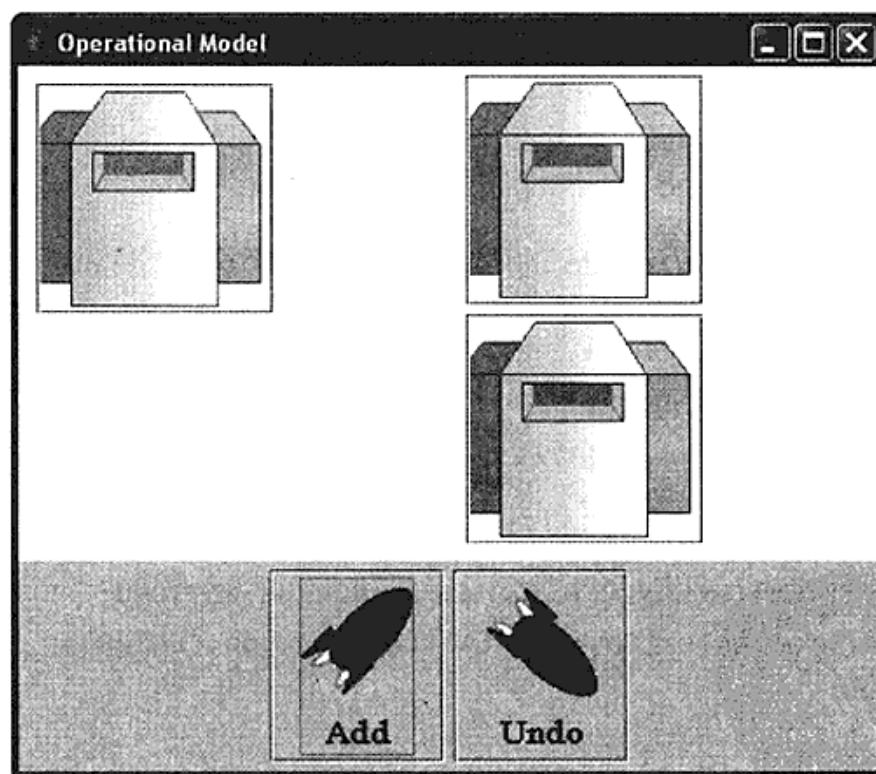


图 17.2 用户可以单击 Add 按钮，在面板的左上区域添加机器，并把机器拖曳至合适位置。
单击 Undo 按钮可撤销添加或拖曳操作

图 17.2 的可视化界面告诉我们用户可以在工厂区域任意添加或拖曳机器（参见 `app.abstractFactory` 目录中的 `ShowVisualization` 程序）。`Visualization` 类在构造函数内接收一个 `UI` 对象来获取按钮。图 17.3 显示了 `visualization` 类。

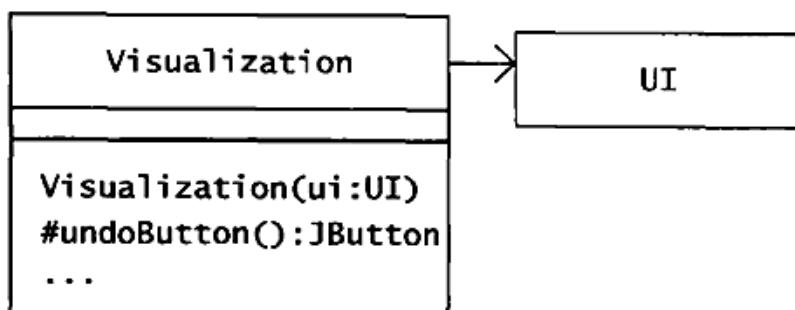


图 17.3 **Visualization** 类使用一个 **UI** 工厂对象创建 GUI

Visualization 类使用 **UI** 对象创建它的 GUI。下面是 **undoButton()** 方法的代码示例。

```
protected JButton undoButton() {
    if (undoButton == null) {
        undoButton = ui.createButtonCancel();
        undoButton.setText("Undo");
        undoButton.setEnabled(false);
        undoButton.addActionListener(mediator, undoAction());
    }
    return undoButton;
}
```

这段代码用于创建一个名称为 Undo 的取消按钮。UI 类会决定按钮上的图片与大小。代码如下所示：

```
public JButton createButton() {
    JButton button = new JButton();
    button.setSize(128, 128);
    button.setFont(getFont());
    button.setVerticalTextPosition(AbstractButton.BOTTOM);
    button.setHorizontalTextPosition(AbstractButton.CENTER);
    return button;
}

public JButton createButtonOk() {
    JButton button = createButton();
    button.setIcon(getIcon("images/rocket-large.gif"));
    button.setText("Ok!");
    return button;
}
```

```
public JButton createButtonCancel() {  
    JButton button = createButton();  
    button.setIcon(getIcon("images/rocket-large-down.gif"));  
    button.setText("Cancel!");  
    return button;  
}
```

为了给可视化提供一个与众不同的界面风格，我们创建了一个子类，它重写 UI 工厂类中的一些元素，并且可以传递该 GUI 工厂的实例到 `visualization` 类的构造函数中。

假定我们发行了一个增添了新特性的 `visualization` 类的新版本，同时，相关代码正在进行 beta 测试。此时，我们希望改变一些用户接口，例如把字体改成斜体并且用 `cherry-large.gif` 和 `cherry-large-down.gif` 两张图片来替换火箭图片。`BetaUI` 类代码的主要实现如下：

```
public class BetaUI extends UI {  
    public BetaUI () {  
        Font oldFont = getFont();  
        font = new Font(  
            oldFont.getName(),  
            oldFont.getStyle() | Font.ITALIC,  
            oldFont.getSize());  
    }  
  
    public JButton createSuttonOk() {  
        // 挑战!  
    }  
  
    public JButton createButtonCancel() {  
        // 挑战!  
    }  
}
```

挑战 17.1

完成 `BetaUI` 类的代码。

答案参见第 338 页

运行如下代码可以获得新的界面效果：

```
package app.abstractFactory;
```

```
// ...
public class ShowBetaVisualization {
    public static void main(String[] args) {
        JPanel panel = new Visualization(new BetaUI());
        SwingFacade.launch(panel, "Operational Model");
    }
}
```

`Visualization` 类的程序运行界面如图 17.4 所示。`UI` 和 `BetaUI` 类的实例为 GUI 应用程序提供了不同风格的 GUI 控件。尽管这是抽象工厂模式的一个有效应用，但有时候这样的设计还是显得不够健壮。特别是 `BetaUI` 类将依赖这种重写创建方法并访问受保护的实例变量的能力，尤其是 `UI` 类中的 `font` 变量。

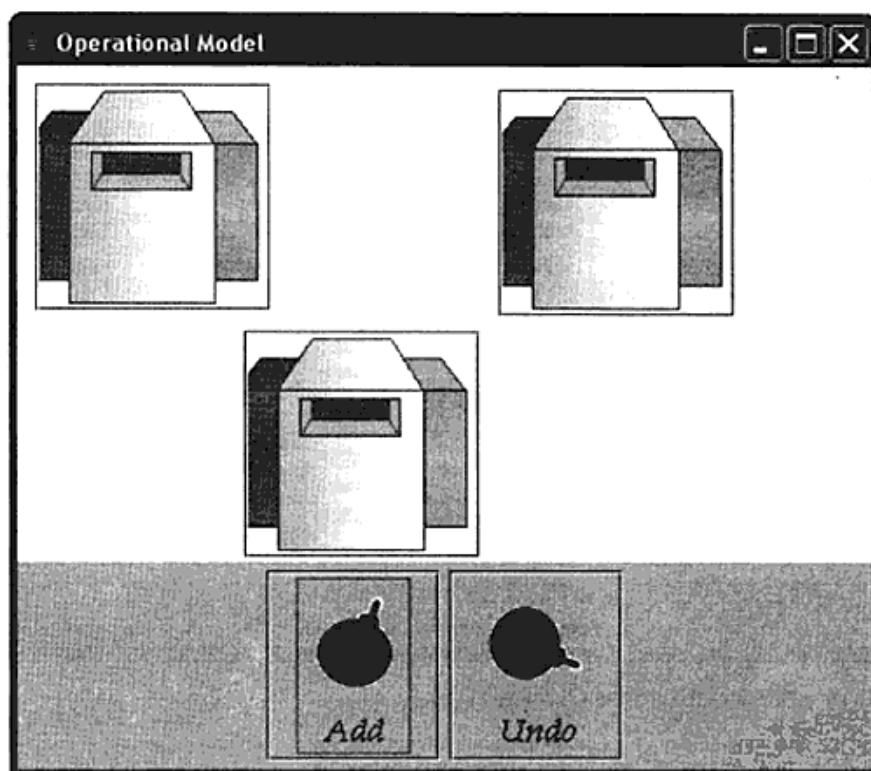


图 17.4 在不修改 `Visualization` 类的任何代码的前提下，`BetaUI` 类呈现了新的界面风格

挑战 17.2

对设计进行修改，使得 GUI 控件工厂仍然允许支持多种扩展，但需要减少子类对 UI 类方法修饰符的依赖。

答案参见第 338 页

抽象工厂模式使得客户端代码在需要新对象的时候，无须关心该对象是由哪个类实例化的。就这点而言，抽象工厂就像是工厂方法的集合。在某些情况下，工厂方法模式的设计可能会迈向抽象工厂模式的设计。

抽象工厂和工厂方法

在第 16 章中，介绍了实现 `CreditCheck` 接口的两个类。当客户端代码调用 `CreditCheckFactory` 类的 `createCreditCheck()` 方法时，该工厂会实例化其中的一个类。工厂具体实例化哪一个类，取决于信贷机构是否已经建立和营业。这个设计使得其他开发者无须关心信贷机构的状态。图 17.5 是 `CreditCheckFactory` 类和 `CreditCheck` 接口的当前实现。

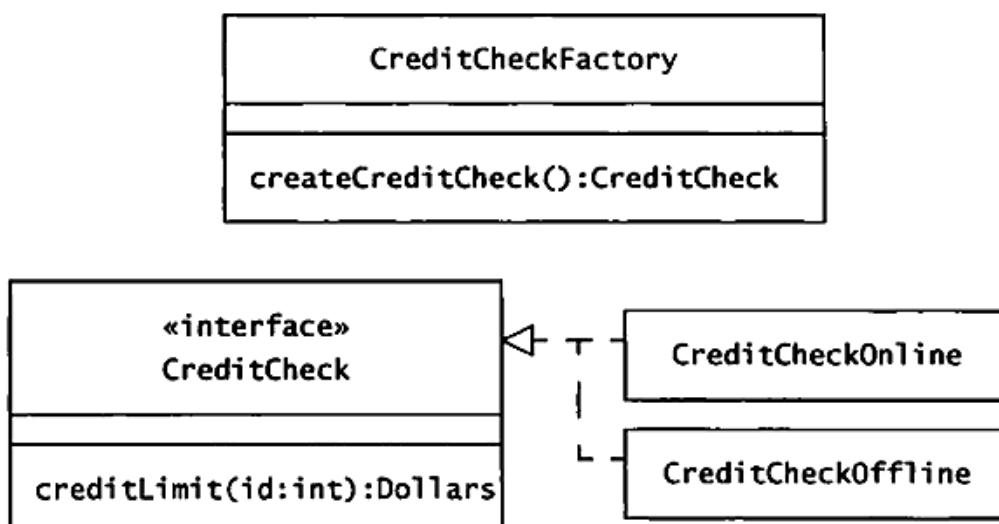


图 17.5 在一个工厂方法模式的设计中，客户端代码无须知道信贷账单是由哪个类实例化的

通常情况下，`CreditCheckFactory` 类为信贷机构提供客户的信贷信息。除此之外，`credit` 包还包括可以帮客户查询交易额和账单记录的类。图 17.6 是 `com.oozinoz.credit` 包的设计。

现在，假定需求分析者告诉你 Oozinoz 公司打算在加拿大开展客户的服务业务。为了在加拿大做业务，不得不使用一个不同的信贷机构和数据源去处理发货交易和账单记录的相关信息。

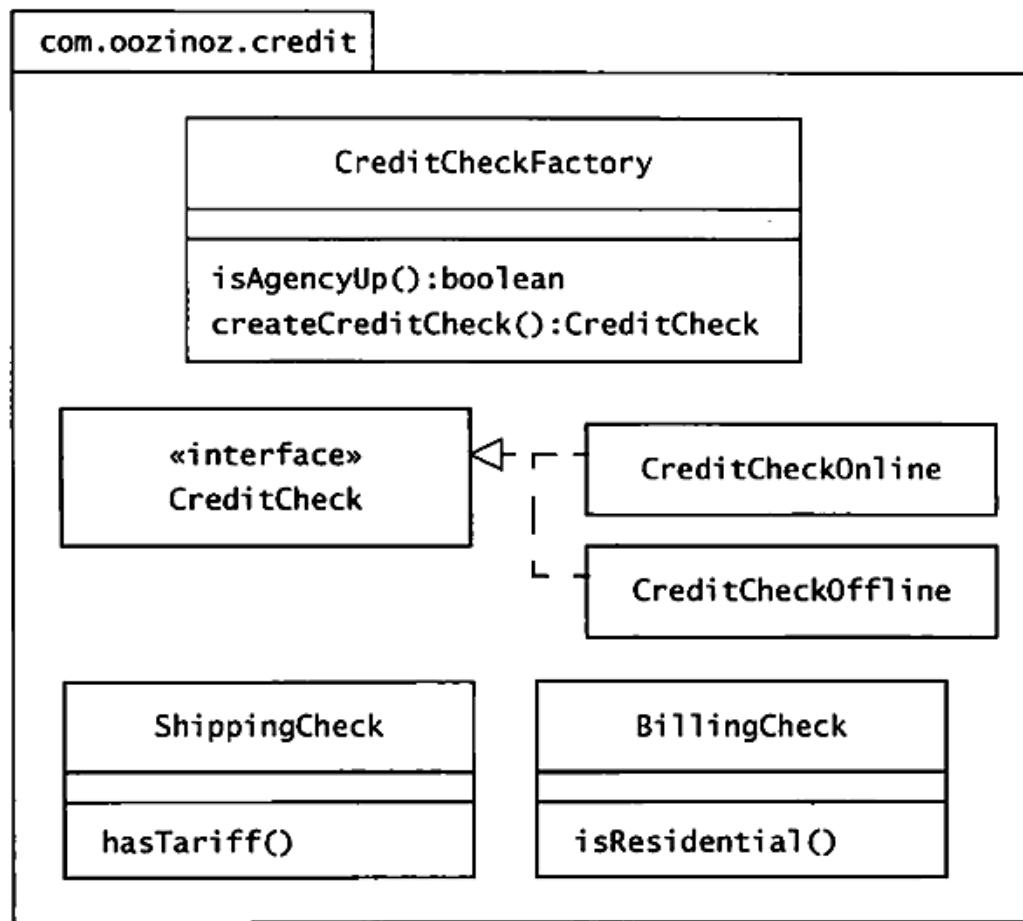


图 17.6 com.oozinoz.credit 包中的类负责核对顾客的信贷信息、发货地址和账单地址

当顾客发出查询请求时，呼叫中心应用程序需要一系列的对象去执行不同的信息核对。使用哪些对象取决于查询呼叫的所在地是加拿大还是美国。你可以运用抽象工厂模式去创建这些对象组。

业务扩展到加拿大势必要使得关于信贷查询的类数量倍增。假定你已经决定将这些类分别放置在三个不同的包中。credit 包将包含三个“check”接口和一个抽象工厂类。这个类有三个创建方法，为查询信贷信息、交易额和账单信息提供合适的对象。同时，如果你不介意呼叫的所在地使用 CreditCheckoffline 类进行离线查询，也可以将该类放到这个包中。图 17.7 是加入 com.oozinoz.credit 包中的一些新内容。

为了在 credit 包中使用具体类实现接口，可以引入两个新的包：com.oozinoz.credit.ca 和 com.oozinoz.credit.us。每个包都包含了一个工厂类和实现了 credit 包中每个接口的若干类。

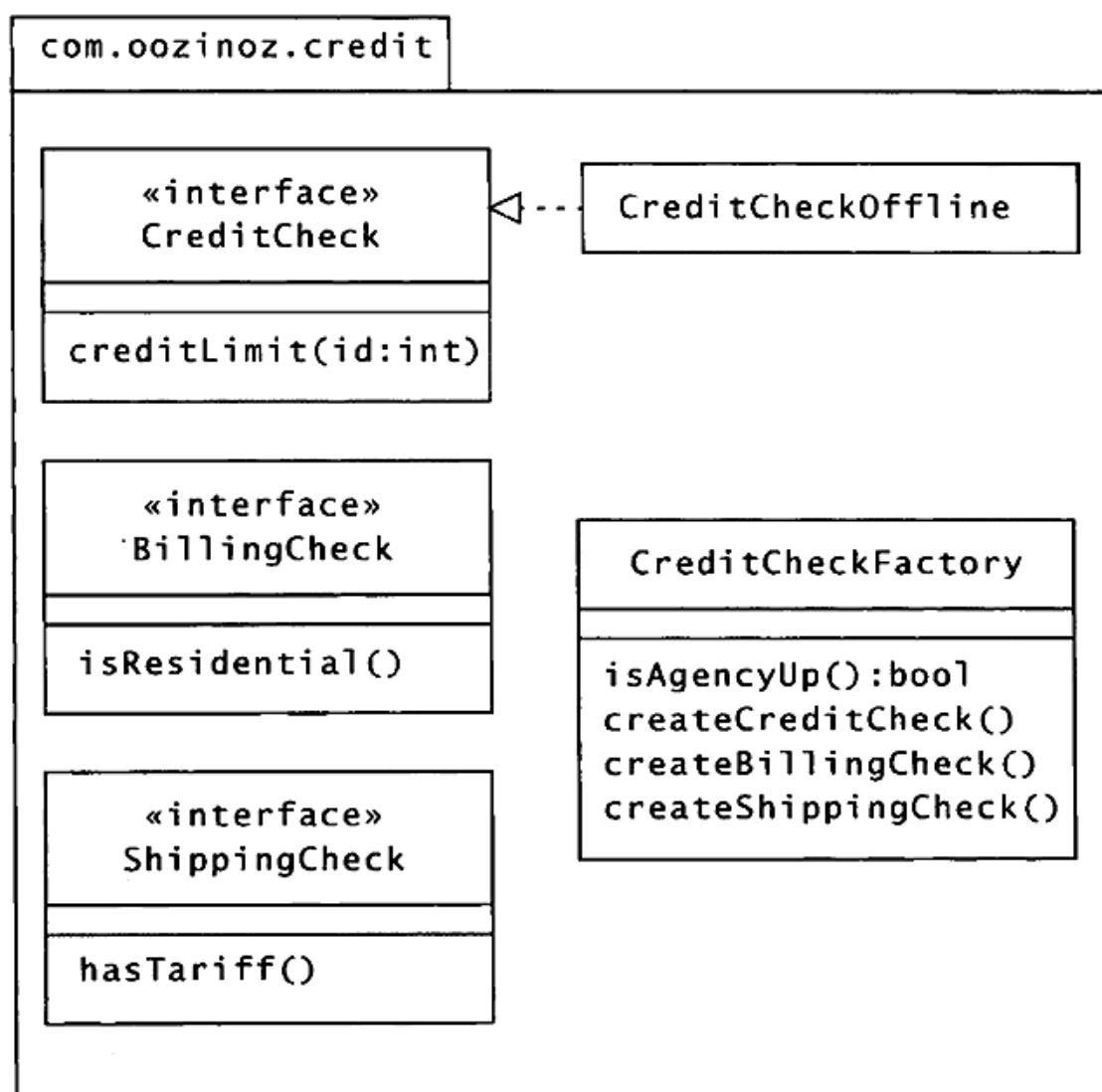


图 17.7 修订过的 `com.oozinoz.credit` 包所包含的主要接口和抽象工厂类

挑战 17.3

图 17.8 显示了 `com.oozinoz.credit.ca` 包中的类以及它们与 `credit` 包中类和接口的关系。请完成该图。

答案参见第 339 页

加拿大和美国信贷查询的具体工厂类非常简单。它们返回加拿大或美国版本的“check”接口，当本地信贷机构处于离线时，情况将变得特殊。此时，所有的具体工厂将返回 `CreditCheckOffline` 对象。正如前一章中所述，`CreditCheckFactory` 类中也有一个 `isAgencyUp()` 方法来告知信贷机构是否已经可以访问。

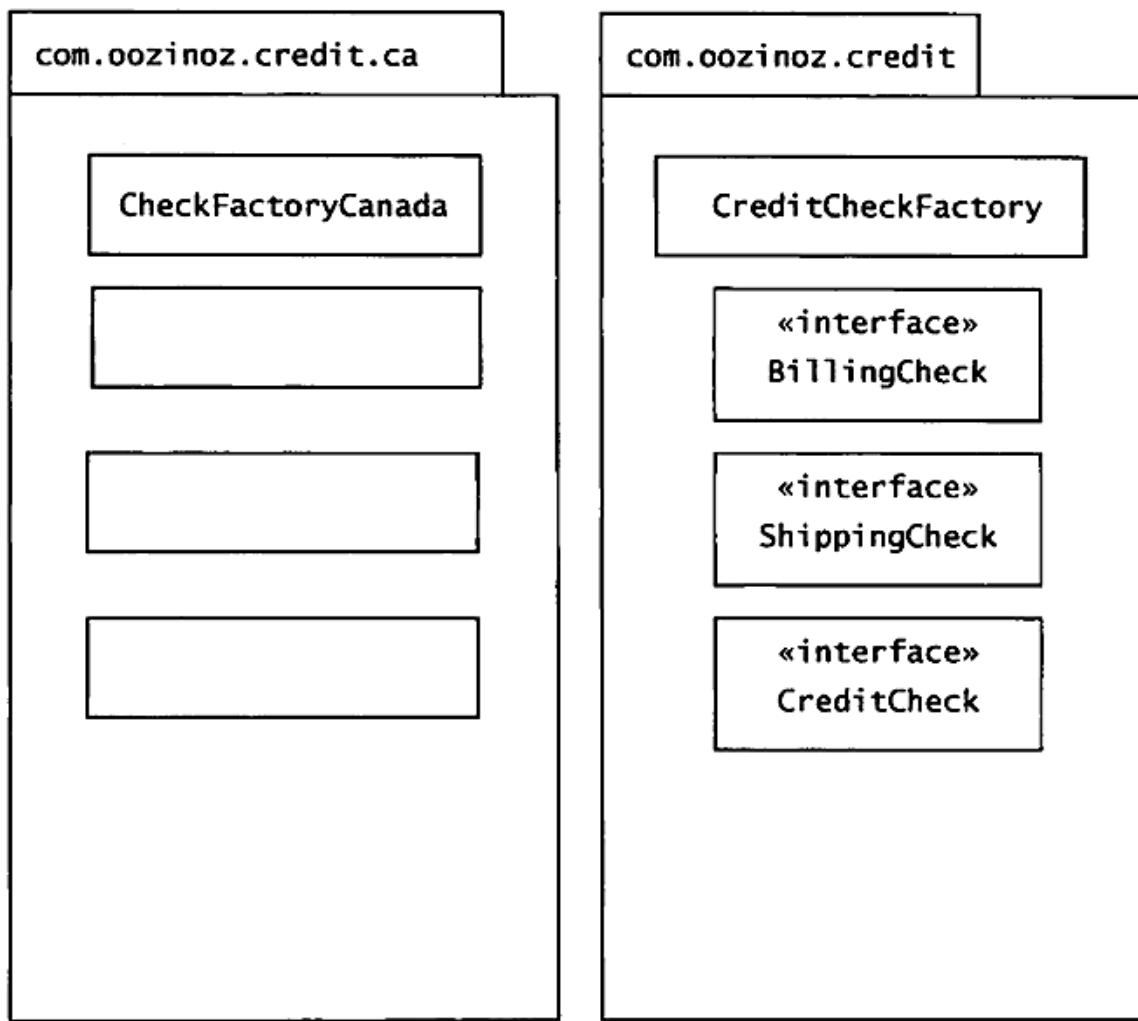


图 17.8 该图展示了 com.oozinoz.credit.ca 包中的类以及它们与 com.oozinoz.credit 包中的类的关系

挑战 17.4

完成 `CheckFactoryCanada.java` 的代码。

```
package com.oozinoz.credit.ca;

import com.oozinoz.credit.*;

public class CheckFactoryCanada extends CreditCheckFactory{
    // 挑战!
}
```

答案参见第 340 页

此时，可以应用抽象工厂模式来创建完成不同顾客信息核对的对象系列。`CreditCheckFactory` 抽象类的实例可以是 `CheckFactoryCanada` 类或 `CheckFactoryUS` 类。这些对象都是抽象工厂，可以创建账单、交易量，同时信贷查询对象完全匹配这个工厂对象所代表的国家。

包和抽象工厂

粗略地讲，一个包通常包含一系列的类，一个抽象工厂可以创建一系列的对象。在之前的例子中，使用两个不相关的包实现了加拿大和美国的抽象工厂，还用另一个包来提供工厂所产生的包的公共接口。

挑战 17.5

给出充分理由说明应该将每个工厂及其相关类放在相互独立的包中，或者提出一个更好的解决方案。

答案参见第 340 页

小结

可以使用抽象工厂模式为客户端代码做一些代码准备，比如，创建相互关联或相互依赖的系列对象中的部分对象。这个模式的一个经典应用是界面风格系列——GUI 控件（工具箱）。也有部分主题并不包含在这个对象族中，例如客户的所在国家决定不同的界面风格。如同工厂方法模式，通过运用抽象工厂模式，客户端代码无须知道哪个类被实例化。抽象工厂给客户端提供了工厂类，每个工厂类都可以创建同一主题下的相关对象。

原型（Prototype）模式

设计一个类时，通常会提供构造函数，使得客户端应用程序能够通过它去创建对象。但在某些情形下，可能不允许类的调用者直接调用构造函数。在构造型设计模式中，如构建者模式、工厂方法模式、抽象工厂模式都可以避免客户端直接调用构造函数。这些模式都是通过引入新类，代表客户端代码实例化合适的类对象。原型模式创建对象的方式与这些模式殊途同归。

原型模式的意图是通过复制一个现有的对象来生成新的对象，而不是通过实例化的方式。

作为工厂的原型

假定你在 Oozinoz 系统中使用抽象工厂模式为不同的应用场景提供用户界面。图 18.1 列出了所用到的用户界面工厂。

Oozinoz 系统可以为不同用户在不同环境下提供合适的界面。但是，一旦用户需要更多的用户界面工具包，则为每种应用场合创建一个新的用户界面工厂类的工作，就会变得非常繁重。因此，为了避免用户界面工厂类数量的不断膨胀，Oozinoz 系统的开发者建议采用原型模式：

- 删去 `UIKit` 的子类。
- 让 `UIKit` 类的每个实例成为能够分发控件原型副本的用户界面工厂。
- 把静态方法中创建的新 `UIKit` 对象的代码转移到 `UIKit` 类中。

在这种设计下，`UIKit` 对象将拥有完整的原型实例变量集合：一个按钮对象、一个网格对

象、一个面板对象等。创建新 `UIKit` 对象的代码将会为这些原型控件设置相应的属性去实现期望的界面风格。`create-()`方法将会返回这些原型控件的副本。

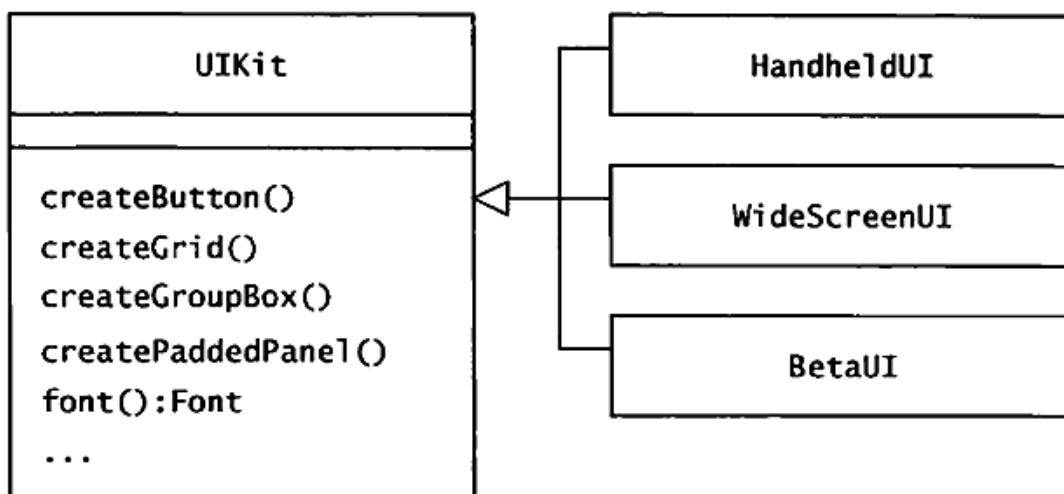


图 18.1 三种抽象工厂(工具包)产生三种不同的用户界面风格

例如，我们可以在 `UI` 类中定义一个静态的 `handheldUI()` 方法。它将实例化 `UIKit` 对象，设置适合手持设备的对象实例变量，同时，返回用做 GUI 工具集的对象。

挑战 18.1

原型模式的使用会减少 Oozinoz 公司系统用来维护多个不同 GUI 工具箱的类数量。请列出这种设计方法的其他两个优点和缺点。

答案参见第 341 页

创建对象的常规方法是调用类的构造函数。原型模式提供了灵活的解决方案，你可以在运行时决定使用哪个对象作为一个模板去创建新的对象。然而，Java 程序开发中的原型方法并不允许对象拥有与父对象不同的方法。因此，可能需要重新考虑原型模式的利弊，甚至需要先试试原型模式是否符合需求。为了使用原型模式，需要掌握 Java 语言的复制对象机制。

利用克隆进行原型化

原型模式的意图是通过复制一个样本来创建新对象。当复制一个对象时，新对象将拥有与原对象相同的属性和方法。当然，这个新对象还可以继承部分甚至全部原对象的数据值。例如，

面板的副本应该具有与原面板相同的属性。

我们需要考虑一个关键问题：复制对象时，复制操作会提供原始对象属性值的副本，还是副本与原始对象共享这些属性值？作为一个应用系统开发者，这些问题往往被忽略甚至回答错误。如果开发者错误地理解复制对象机制，系统缺陷便会应运而生。Java类库中的很多类都提供复制机制，但是，作为一个应用系统开发者，你必须理解复制的工作机制，尤其是当你想使用原型模式时。

挑战 18.2

所有的对象都可以从 `Object` 类继承 `clone()` 方法。如果你不熟悉这个方法，可以从联机帮助或相关文档中了解。然后写下你对这个方法的理解。

答案参见第 341 页

挑战 18.3

假定 `Machine` 类有两个属性：整型的 `ID` 和 `Location` 对象，其中，`Location` 是一个单独的类。

设计一个对象图，说明 `Machine` 对象及它的 `Location` 对象，还有其他对象都是从 `Machine` 对象的 `clone()` 方法创建而来的。

答案参见第 342 页

`clone()` 方法使我们能很容易地往类中添加 `copy()` 方法。比如，你可以使用如下代码创建可克隆面板的类：

```
package com.oozinoz.ui;
import javax.swing.JPanel;

public class OzPanel extends JPanel implements Cloneable {
    // 危险!
    public OzPanel copy() {
        return (OzPanel) this.clone();
    }
}
```

```
// ...  
}
```

上面代码中的 `copy()` 方法使得大家都可以使用克隆功能，并且复制对象为需要的类型。然而，代码存在的问题是，`clone()` 方法会创建某 `JPanel` 对象所有属性的副本，而不管你是否理解这些属性的功能。注意，`JPanel` 类的属性包括它的超类的所有属性。如图 18.2 所示。

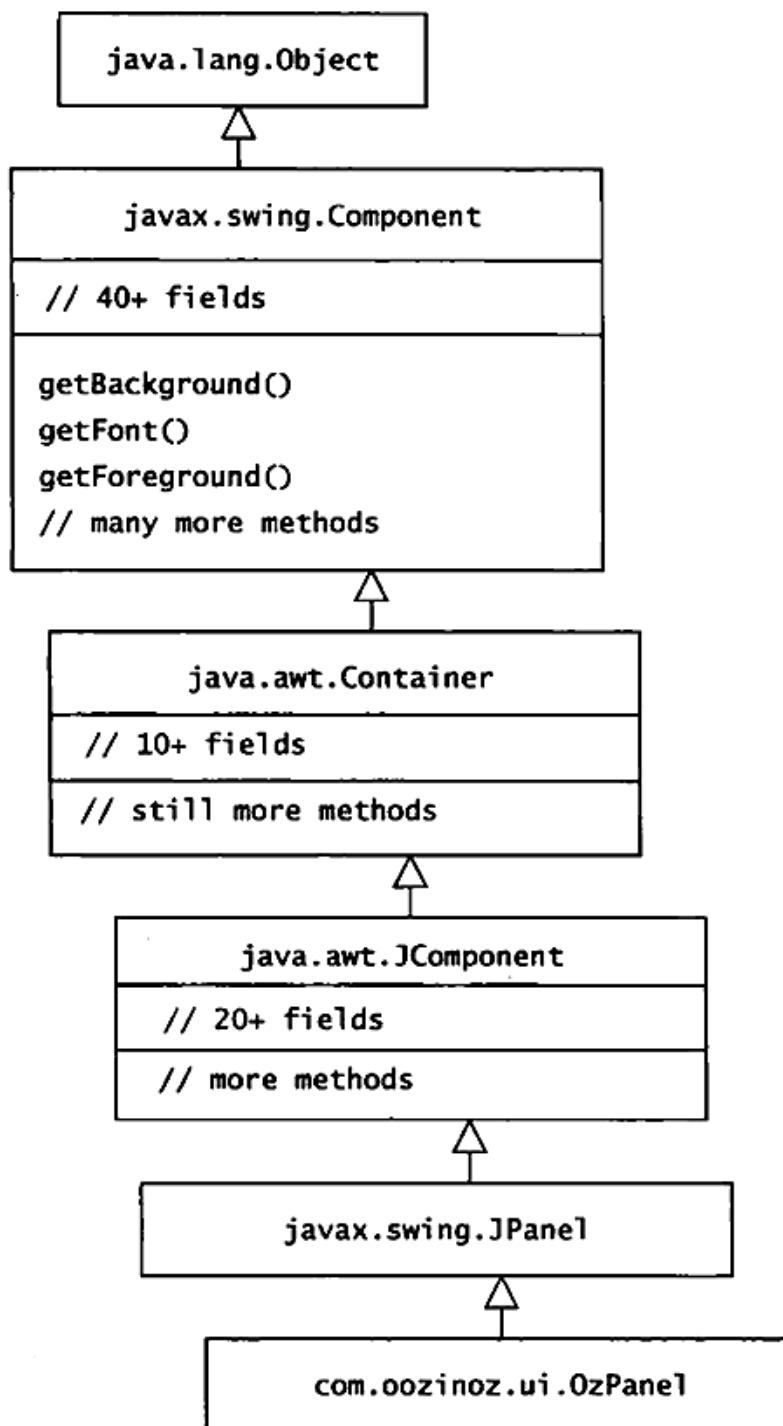


图 18.2 `OzPanel` 类从超类中继承了大量的属性字段和变量

在图 18.2 中，OzPanel 类从 Component 类继承了大量的属性，这些也是使用 GUI 对象时需要复制的属性。

挑战 18.4

不依赖 `clone()` 方法，写一个 `OzPanel.copy2()` 方法复制一个面板。假定需要复制的重要属性是 `background`、`font` 和 `foreground`。

答案参见第 342 页

小结

原型模式使得用户可以通过复制对象样本来创建新对象。与通过调用构造函数创建对象相比，二者的主要区别在于：通过复制创建的新对象一般会包含原始对象的某些状态。你可以充分利用这一特点，尤其是当多个对象的类在属性上仅存在细微差别，方法上却完全相同的时候。在这种情形下，通过为用户提供一个可以复制的对象原型，就可以在运行时动态创建新类。

当需要创建一个副本时，可以借助 `Object.clone()` 方法。但是必须记住，该方法创建的新对象与对象原型的属性字段是完全相同的。所以，通过这种复制方法获得的新对象或许并不是应用系统开发者所期望的副本，还需要执行更进一步的复制工作。如果对象原型的字段太多，那么我们可以选择先实例化一个新对象，然后将对象原型中需要的字段赋给新对象的对应字段。

第 19 章

备忘录（Memento）模式

有时候，你想要创建的对象已经在系统中存在。例如，当用户执行撤销操作，使系统回滚到之前的某一状态，或重新执行之前搁置的工作时，就会出现这种情形。

备忘录模式的意图是为对象状态提供存储和恢复功能。

经典范例：使用备忘录模式执行撤销操作

第 17 章的抽象工厂模式介绍了一个可视化的应用，用户可以通过一个工厂对象使用材料进行操作建模实验。假定撤销按钮的功能尚未实现，我们就可以运用备忘录模式。

事实上，备忘录就是存储状态信息的对象。在可视化应用程序中，我们需要保存的是应用程序的状态。无论何时添加或移动机器，用户都可以通过单击 Undo 按钮取消操作。为了给可视化应用程序添加撤销功能，需要考虑如何在一个备忘录中捕获应用程序的状态。同时，我们还必须决定何时去捕获状态，如何恢复程序到之前捕获到的状态。当应用程序启动后，会出现如图 19.1 所示的界面。

程序最初启动时是一个空白状态。任何时刻只要程序处在这种空白状态下，Undo 按钮都是禁用的。经过执行一些添加或拖曳操作，可视化应用程序变成如图 19.2 所示的样子。

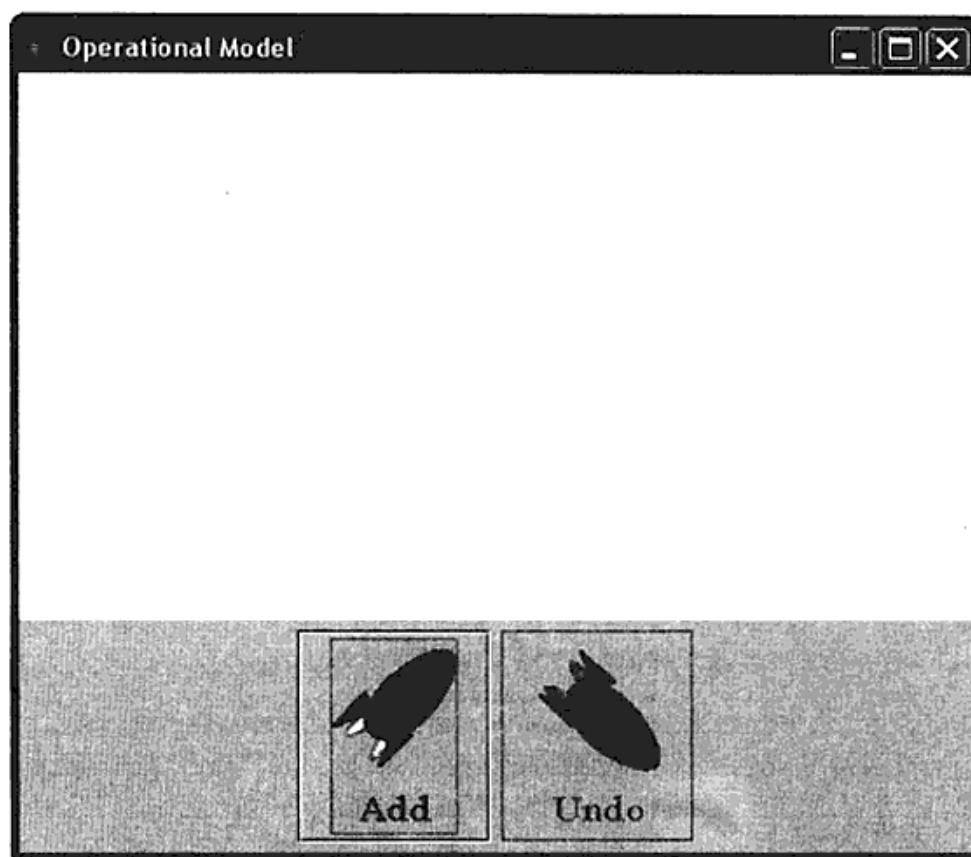


图 19.1 当可视化应用程序启动后，工作面板是空的，Undo 按钮处于禁用状态

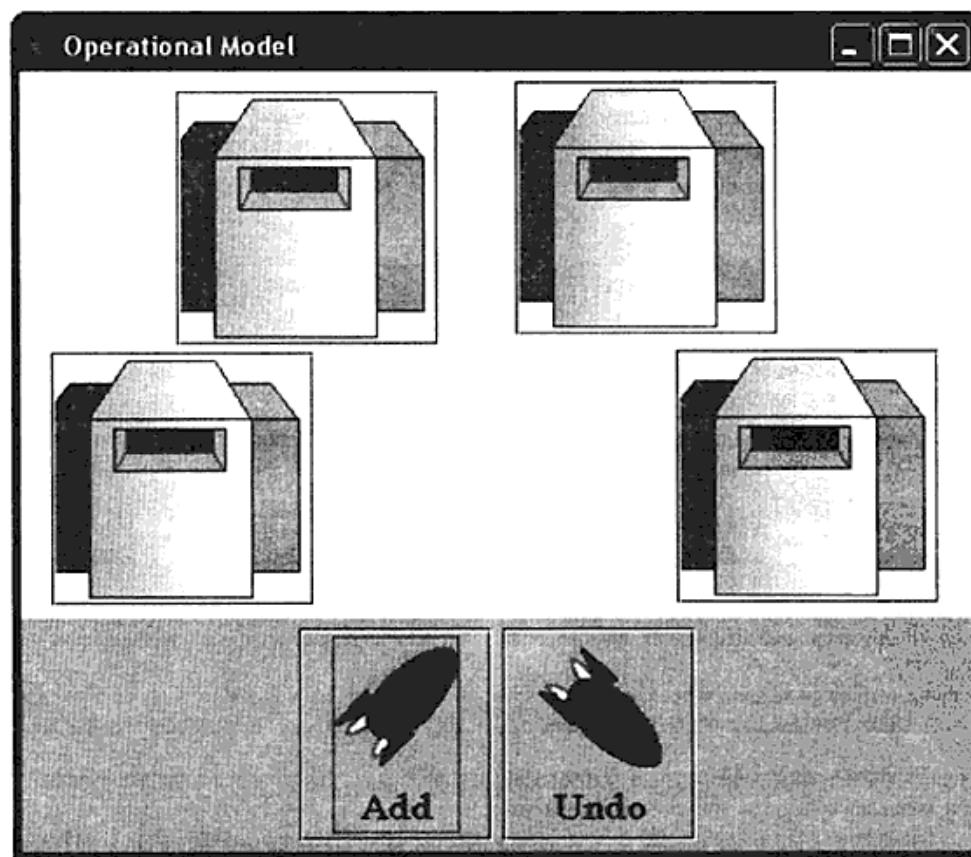


图 19.2 用户可以在可视化应用程序中添加和重新布置机器

在备忘录中，我们需要捕获的状态由用户放置的机器位置信息组成。可以使用栈来保存这些备忘录，每当用户单击一次 Undo 按钮就弹出一个备忘录。

- 每当用户在可视化应用程序中添加或移动机器时，应用程序就为这个仿真工厂创建一个备忘录，并把它放入一个栈中。
- 每当用户单击 Undo 按钮时，应用程序就将栈顶部的备忘录弹出，然后将仿真对象恢复为该备忘录所记录的状态。

当可视化应用程序启动后，程序会向空栈压入一个初始的空备忘录，而这个备忘录永远也不会被弹出，以确保栈顶总有一个有效的备忘录。一旦栈仅包含一个备忘录，Undo 按钮就会被禁用。

可以在单个类中编写这些代码，但是，我们希望能够添加一些可以支持操作建模或其他用户期望用到的功能。如果仍然在一个类中集中这些职责，就会导致最终的应用程序变得非常庞大。因此，使用 MVC 设计是比较明智的选择。图 19.3 说明了如何将工厂建模功能转移到单独的类中。

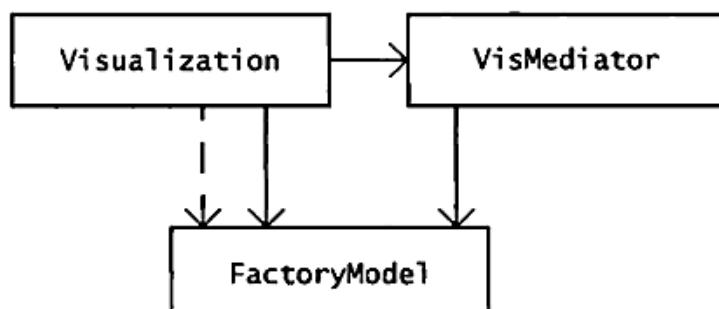


图 19.3 这个设计把应用程序的功能分到单独的类中，包括建模工厂、提供 GUI 元素以及处理用户的单击操作

这样的设计思路首先要求集中开发一个 `FactoryModel` 类，它没有任何 GUI 控件，和 GUI 类也没有任何依赖关系。

`FactoryModel` 类是我们设计的核心。它负责维护机器的当前配置信息以及之前配置的备忘录信息。

每当客户端代码要求工厂添加或移动一个机器时，工厂都会创建一个副本——一个备忘录，用来记录当前机器的位置信息并将其压入备忘录栈中。在本例中，我们并不需要特殊的备忘录类。每个备忘录仅仅是位置的列表：特定时间的机器位置列表。

工厂模型必须提供事件用来支持客户对工厂状态变化的关注。这就使得可视化应用程序 GUI 能够将用户所做的修改通知给模型。假设我们希望工厂允许客户注册添加机器和拖动机器等事件。图 19.4 所示的是 FactoryModel 类的设计说明。

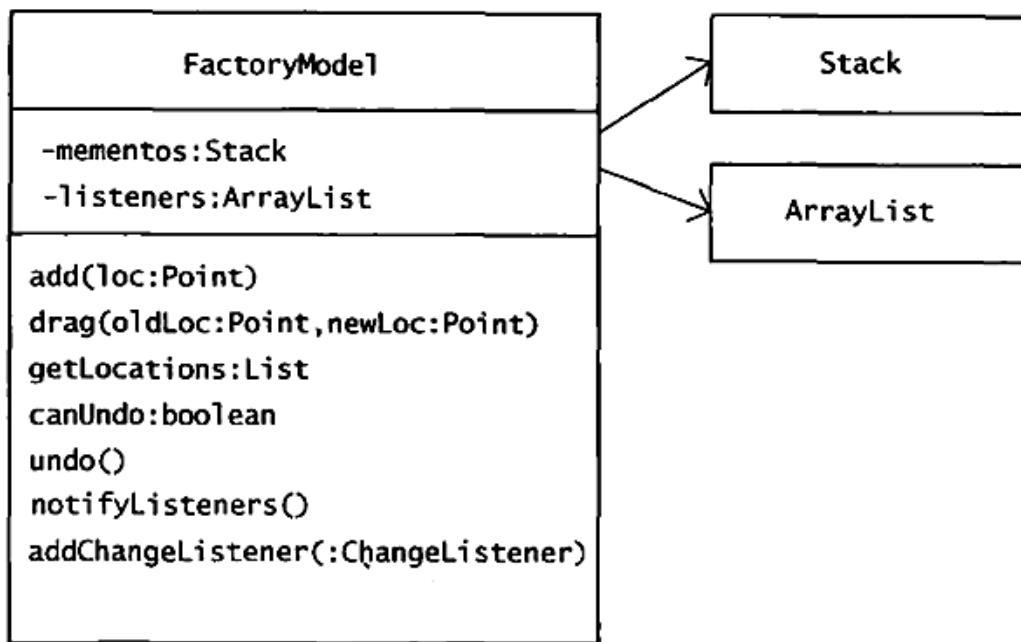


图 19.4 FactoryModel 类维护一个存储工厂配置信息的栈，并允许客户记录工厂的变更信息

图 19.4 的设计思路要求 FactoryModel 类为客户提供注册多个事件的能力。

比如，考虑添加机器事件。任何已注册的 ChangeListener 对象都会接收到这个变更操作。

```

package com.oozinoz.visualization;
// ...
public class FactoryModel {
    private Stack mementos;

    private ArrayList listeners = new ArrayList();

    public FactoryModel() {
        mementos = new Stack();
        mementos.push(new ArrayList());
    }
    //...
}
  
```

一开始，构造函数将工厂的初始配置设置为一个空列表。类中的其他方法维护机器配置备忘录的栈，并在改变发生时触发相应的事件。例如，要给当前配置添加一个机器，客户可以调

用下面的方法：

```
public void add(Point location) {  
    List oldLocs = (List) mementos.peek();  
    List newLocs = new ArrayList(oldLocs);  
    newLocs.add(0, location);  
    mementos.push(newLocs);  
    notifyListeners();  
}
```

这段代码创建了一个机器位置的新列表，并将其压入工厂模型所维护的备忘录栈中。略有不同的是，这段代码确保新机器位于列表的第一位。新机器（位置信息）应该出现在其他机器（信息）的前面，这是因为工厂平面图中的机器位置可能重合。

在接收到工厂模型的事件时，注册了变化通知的客户可以通过完全重绘来更新工厂模型的视图。`getLocations()`方法始终会提供工厂模型的最新配置，代码如下所示：

```
public List getLocations() {  
    return (List) mementos.peek();  
}
```

`FactoryModel` 类的 `undo()` 方法允许客户把机器位置模型恢复为以前的版本。执行这些代码的时候，它也会调用 `notifyListeners()` 方法。

挑战 19.1

实现 `FactoryModel` 类的 `undo()` 方法。

答案参见第 343 页

通过注册为监听者，可提供重绘客户的工厂视图的方法，这样，感兴趣的客户可以提供撤销操作。`Visualization` 类就是这样的一个客户。

图 19.3 所示的 MVC 设计思路使得解释用户动作的任务和维护 GUI 的任务相分离。`visualization` 类创建 GUI 控件，但是不处理传递给中间者的 GUI 事件。`visMediator` 类把 GUI 事件解释为工厂模型中相应的变化。当模型发生改变时，GUI 可能需要相应的更新。`visualization` 类会注册 `FactoryModel` 类所提供的通知事件。注意它们的职责划分：

- 可视化应用程序 (`visualization` 对象) 把工厂事件转换为 GUI 的变化。
- 调停者 (`visMediator` 对象) 把 GUI 事件转换为工厂变化。

图 19.5 说明了这三个类更详细的协作关系。

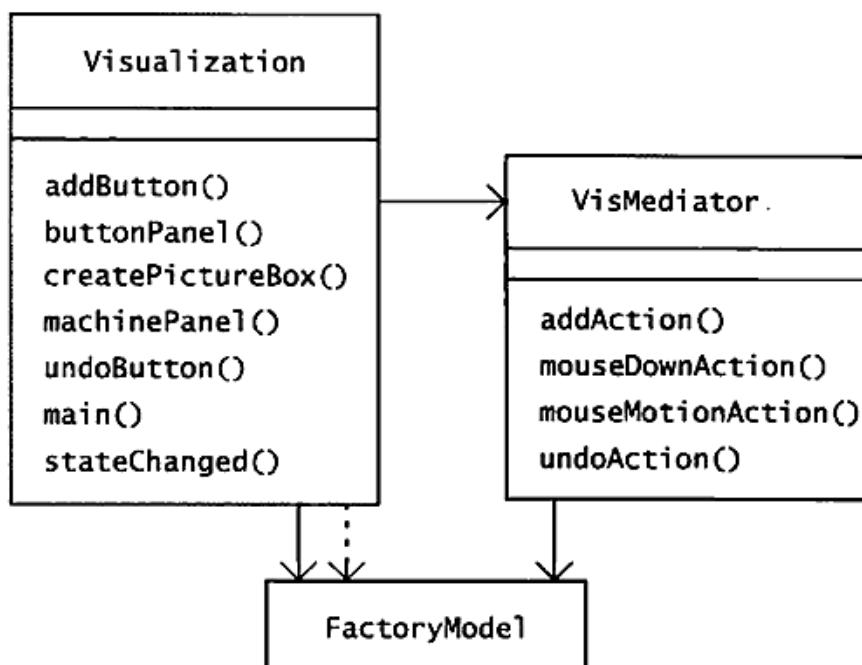


图 19.5 VisMediator 类将 GUI 事件解释成工厂模型的变化, Visualization 类处理工厂事件以更新 GUI

假定在拖曳一张机器照片时，用户碰巧将它放置在一个错误的位置，此时可以单击 Undo 按钮。为了能够处理这个单击任务，`visualization` 类注册了一个调停者对象，用以通知按钮事件。`visualization` 类中的 Undo 按钮的代码如下所示：

```

protected JButton undoButton() {
    if (undoButton == null) {
        undoButton = ui.createButtonCancel();
        undoButton.setText("Undo");
        undoButton.setEnabled(false);
        undoButton.addActionListener(mediator.undoAction());
    }
    return undoButton;
}
  
```

这段代码把处理单击操作的任务交给了调停者。调停者再通知请求变化的工厂模型。使用如下代码将执行撤销请求，以恢复工厂模型的变化：

```
private void undo(ActionEvent e) {
```

```

factoryModel.undo();
}

```

方法中的 factoryModel 变量是 visualization 类创建的 FactoryModel 的一个实例，同时，它会把工厂模型传入 visMediator 类的构造函数。我们已经检查了 FactoryModel 类的 pop() 命令。图 19.6 显示了当用户单击 Undo 按钮后消息的流动过程。

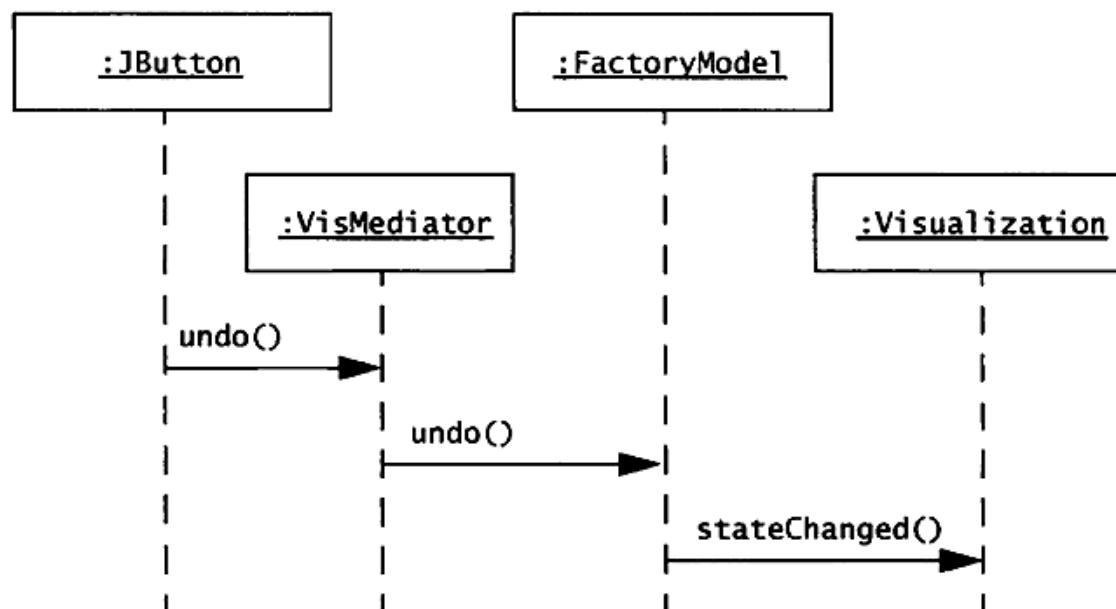


图 19.6 用户单击 Undo 按钮后消息的流动过程

当 FactoryModel 类弹出当前的配置信息时，之前存储为备忘录的配置会显露出来，undo() 方法会通知所有的 ChangeListeners。Visualization 类在其构造函数中注册如下信息：

```

public Visualization(UI ui) {
    super(new BorderLayout());
    this.ui = ui;
    mediator = new VisMediator(factoryModel);
    factoryModel.addChangeListener(this);
    add(machinePanel(), BorderLayout.CENTER);
    add(buttonPanel(), BorderLayout.SOUTH);
}

```

对于工厂模型中每台机器的位置，可视化应用程序都维护一个 Component 对象，该对象是在 createPictureBox() 方法中建立起来的。stateChanged() 方法必须从机器面板上清除所有的图片箱子控件，同时，在工厂模型的当前位置集合中重新添加新的图片箱子。如果工厂栈中只剩下唯一一条备忘录记录，statechanged() 方法必须禁用 Undo 按钮。

挑战 19.2

完成 `Visualization` 类中 `stateChanged()` 方法的代码编写。

答案参见第 343 页

借助备忘录模式，你可以保存和恢复对象的状态。备忘录模式最常见的用法是在应用程序中提供撤销操作。在某些应用程序中，例如工厂可视化程序，你需要存储信息的仓库可能是一个对象。在某些情况下，你可能需要持久地保存备忘录。

备忘录的持久性

备忘录是一个用来存储对象状态的小型数据仓库。通常，你可以用另一个对象、字符串或者文件来创建备忘录。在设计备忘录的时候，应事先预测对象状态可能在备忘录中存储的时间，因为这个时间将影响我们的设计策略。这段时间可能是一瞬间、几小时、几天或者几年。

挑战 19.3

写出你决定将备忘录存储在文件中而不是对象中的两个理由。

答案参见第 344 页

跨会话的持久性备忘录

用户运行一个程序，执行程序中的事务，直到最后退出，整个过程称为一次会话。假设你的用户希望存储一次会话中的一个仿真（对象），然后在另一个会话中恢复该对象。这种能力通常叫做持久性存储。持久性存储能力和备忘录模式的意图是一致的，同时它还是已经实现的撤销操作的自然扩展。

假定由 `Visualization` 类派生出一个 `Visualization2` 类，该类具有一个菜单栏，包含了

File 菜单，其中提供了 Save As... 和 Restore From ... 命令。

```
package com.oozinoz.visualization;

import javax.swing.*;
import com.oozinoz.ui.SwingFacade;
import com.oozinoz.ui.UI;

public class Visualization2 extends Visualization {
    public Visualization2(UI ui) {
        super(ui);
    }

    public JMenuBar menus() {
        JMenuBar menuBar = new JMenuBar();

        JMenu menu = new JMenu("File");
        menuBar.add(menu);

        JMenuItem menuItem = new JMenuItem("Save As...");
        menuItem.addActionListener(mediator.saveAction());
        menu.add(menuItem);

        menuItem = new JMenuItem("Restore From...");
        menuItem.addActionListener(
            mediator.restoreAction());
        menu.add(menuItem);

        return menuBar;
    }

    public static void main(String[] args) {
        Visualization2 panel = new Visualization2(UI.NORMAL);
        JFrame frame = SwingFacade.launch(
            panel, "Operational Model");
        frame.setJMenuBar(panel, menus());
        frame.setVisible(true);
    }
}
```

为了实现上述代码, VisMediator 类还需要添加 `saveAction()` 和 `restoreAction()` 两个方法。当用户选择菜单项时, MenuItem 对象会促使这些动作被调用。运行 `Visualization2` 类的 GUI, 如图 19.7 所示。

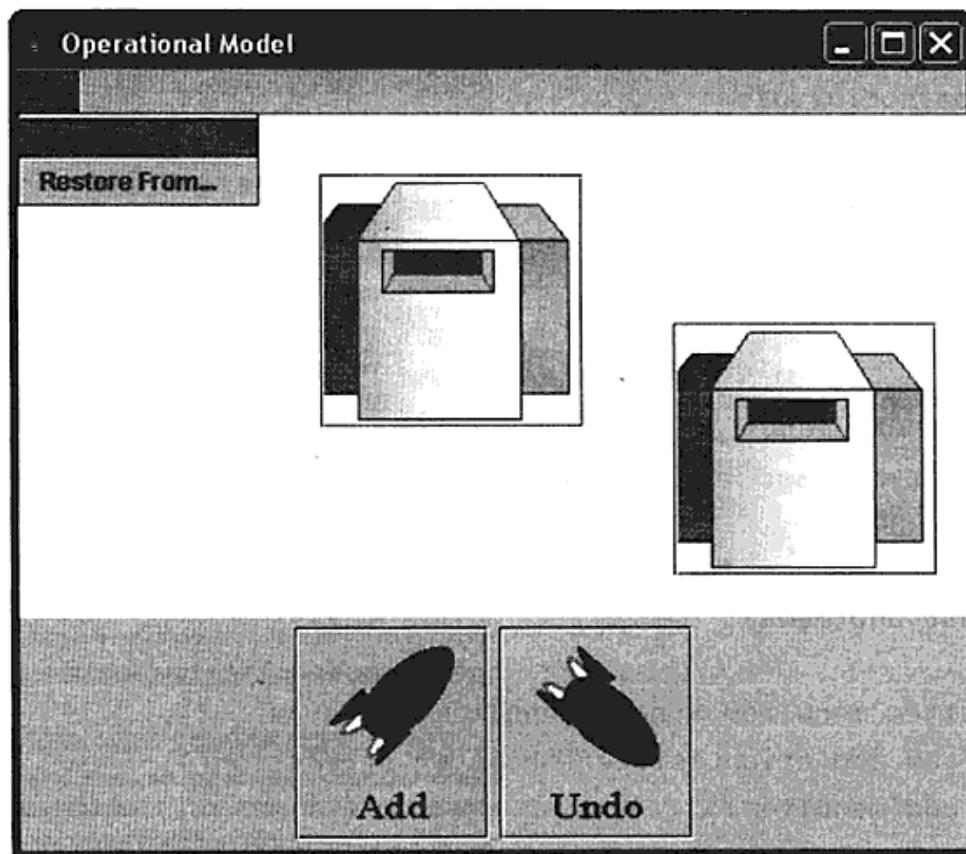


图 19.7 File 菜单的添加使用户可以保存一个备忘录, 以便于应用程序的后期恢复

一个存储对象 (比如工厂模型的配置) 的简易方式是序列化。VisMediator 类的 `saveAction()` 方法的代码如下所示:

```
public ActionListener saveAction() {
    return new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            try {
                VisMediator.this.save((Component)e.getSource());
            } catch (Exception ex) {
                System.out.println(
                    "Failed save: " + ex.getMessage());
            }
        }
    };
}
```

```
public void save(Component source) throws Exception {  
    JFileChooser dialog = new JFileChooser();  
    dialog.showSaveDialog(source);  
  
    if (dialog.getSelectedFile() == null)  
        return;  
  
    FileOutputStream out = null;  
    ObjectOutputStream s = null;  
    try {  
        out = new FileOutputStream(dialog.getSelectedFile());  
        s = new ObjectOutputStream(out);  
        s.writeObject(factoryModel.getLocations());  
    } finally {  
        if (s != null) s.close();  
    }  
}
```

挑战 19.4

请写出 VisMediator 类中 restoreAction() 方法的代码。

答案参见第 344 页

在 *Design Patterns* 一书中，备忘录模式的意图被定义为：在不违反封装的前提下，捕获对象的内部状态，并在该对象之外保存这个状态，以便于对象的状态可以在将来被恢复。

挑战 19.5

在这种情况下，我们使用 Java 序列化机制将备忘录写到一个二进制文件中。假定已经将备忘录信息以 XML 格式（文本格式）保存起来了。请结合自己的理解，简单陈述把备忘录保存为文本格式是否会破坏封装。

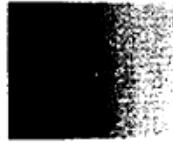
答案参见第 345 页

当开发者希望借助于序列化或者 XML 文件来保存和恢复对象的数据时，你应该认真理解其中的含义。这就涉及设计模式中的重点：通过使用公共词汇，我们才能更好地讨论设计模式

概念及其应用。

小结

借助备忘录模式，可以捕获对象的状态，以便于将来把对象恢复为以前的状态。具体采用哪种方法来存储对象状态，取决于对象状态需保存时间的长短。恢复一个对象，有时候只在敲击几下鼠标和键盘之后，但也可能要经过几天或者几年。在应用程序会话期间保存和恢复对象，最常见的理由是支持撤销操作。在这种情况下，我们可以将对象的状态存储在另一个对象中。为了支持对象跨多个会话的持久性存储，可以使用对象序列化或其他方式来保存备忘录。



第4部分

操作型模式

第 20 章

操作型模式介绍

在编写一个 Java 方法时，你完成的是整个工作中级别高于单行代码的一个基本单元。这些方法牵涉到整体设计、架构以及测试计划。编写方法是面向对象编程的中心环节。但反过来说，尽管方法是软件设计的核心，我们还是难以解释方法到底是什么，方法又是如何工作的。追本溯源，还是因为许多开发人员常常混淆了方法与操作的含义。进一步讲，算法和多态的概念比方法更加抽象，但最终它们还是通过方法来实现。

对术语“算法”、“多态”、“方法”与“操作”具有清晰的认识，有助于你理解多种设计模式。尤其是状态模式、策略模式和解释器模式，都是通过实现多个类中的方法来完成操作的。但是，只有当我们对方法和操作的理解达成共识时，这样的表达才有用。

操作和方法

关于类，有很多相关的术语。我们尤其要注意甄别“操作”和“方法”。UML 定义了操作和方法之间的区别。如下所示：

- 操作是一个服务的规格说明，它可以被类的实例调用。
- 方法是操作的实现。

注意，这意味着操作是方法的一种抽象。

操作表示类做了什么，还表示服务提供的接口。不同的类可能用不同的方法实现同样的操作。例如，很多类用自己的方式实现 `toString()` 操作。每个类都通过方法来实现操作。方法就是（包含）让操作能够工作的代码。

搞清楚方法和操作的定义，有助于理清许多设计模式的结构。设计模式来源于类和方法，因此，能够在诸多设计模式中看到操作的身影，也就不足为奇了。例如在合成模式中，叶子节点与合成节点皆实现了共同的操作。在代理模式中，代理对象和目标对象具有相同的操作，因而，代理对象可以管理对目标对象的访问。

挑战 20.1

使用术语“操作”和“方法”来解释职责链模式是如何实现操作的。

答案参见第 345 页

在 Java 语言中，方法的声明包括方法头和方法体两部分。方法体是一系列的指令，可以通过调用方法的签名来执行。方法头包括方法的返回类型以及方法签名，还可能包含访问修饰符与异常语句。方法头的格式如下：

modifiers return-type signature throws-clause

挑战 20.2

尽可能多地写出 9 种 Java 访问修饰符。

答案参见第 345 页

签名

表面看来，操作的意义和签名相似。两个词都指的是接口的方法。当实现一个方法后，就可以使用方法的签名来调用它。*Java™ Language Specification*（由 Arnold 和 Gosling 在 1998 年

编写)一书的 8.4.2 节中说:

方法的签名包括方法名、传入参数的数量以及类型。

注意，方法的签名不包括返回类型，尽管当一个方法重载了另一个方法的声明时，如果它们的返回类型不同时，会发生编译错误。

挑战 20.3

即使 `Bitmap.clone()` 通常返回 `Bitmap` 类的实例，但它返回的却是 `Object` 类型。请问当其返回类型被声明为 `Bitmap` 时，是否依然能通过编译？

答案参见第 346 页

方法签名代表了客户调用的方法。操作是可以被请求的服务规格。术语“签名”和“操作”的意思很接近，但是它们并不是一个词。这两个术语的区别主要在于它们的不同应用场景。当涉及不同类中的方法拥有相同的接口时，使用术语“操作”。当涉及如何将 Java 方法调用匹配到接收对象的方法上时，使用术语“签名”。签名依赖于方法的名字和参数，但是不依赖于方法的返回类型。

异常

在 *Illness as Metaphor* 一书中，Susan Sontag 提到：“每个人生来就具有两种身份，一谓健康，一谓疾病。”这个隐喻不仅适合于人，也适合于方法：正常情况下，方法会正常返回。但是，方法也可能抛出异常，或者在其内部调用其他方法时抛出异常。当程序正常返回时，会返回上次调用结束的入口点。出现异常时，则应用其他规则。

出现异常时，Java 运行时必须找到 `try/catch` 语句来匹配异常。`try/catch` 语句可能会出现在抛出异常的方法中，也可能出现在调用当前方法的方法中，还可能出现在调用之前方法的方法中。如果在方法调用的堆栈中没有找到 `try/catch` 语句，程序会停止并且崩溃。

任何方法都可以使用 `throw` 语句来抛出异常，例如：

```
throw new Exception("Good Luck!");
```

如果你的应用程序在调用方法时抛出了一个非预期的异常，程序可能会突然终止。为避免这类行为发生，需要从架构层面捕获和处理这类异常。或许，你会觉得声明所有可能的异常是一件麻烦事儿。例如，C#不要求声明异常；C++允许异常的声明，但不要求在编译时进行检查。

挑战 20.4

与 Java 不同，C#不要求方法声明任何它可能抛出的异常。这种方式是否是对 Java 的一种改进？

答案参见第 346 页

算法和多态

算法和多态是编程中非常重要的概念，但我们却很难表达这两个术语的含义。如果你想向某人展示方法，可以直接拿代码来讲解。偶尔，某个方法也可能包含完整的算法，但算法通常都是作用于一些方法的。在 *Introduction to Algorithms*（由 Cormen、Leiserson 和 Rivest 在 1990 年编写）一书中提到：

算法是已经定义好的计算程序，将数据或者数据集作为输入，将产生的数据或者数据集作为输出。

算法是一个过程——一个有输入和输出的指令序列。单个的方法可能是一个算法：接收输入——参数列表——产生并且输出其返回值。然而在面向对象的程序中，很多算法需要多个方法来执行。例如第 5 章合成模式中的 `isTree()` 算法，需要 4 个方法，如图 20.1 所示。

挑战 20.5

图 20.1 中描述了多少种算法、多少种操作以及多少个方法？

答案参见第 346 页

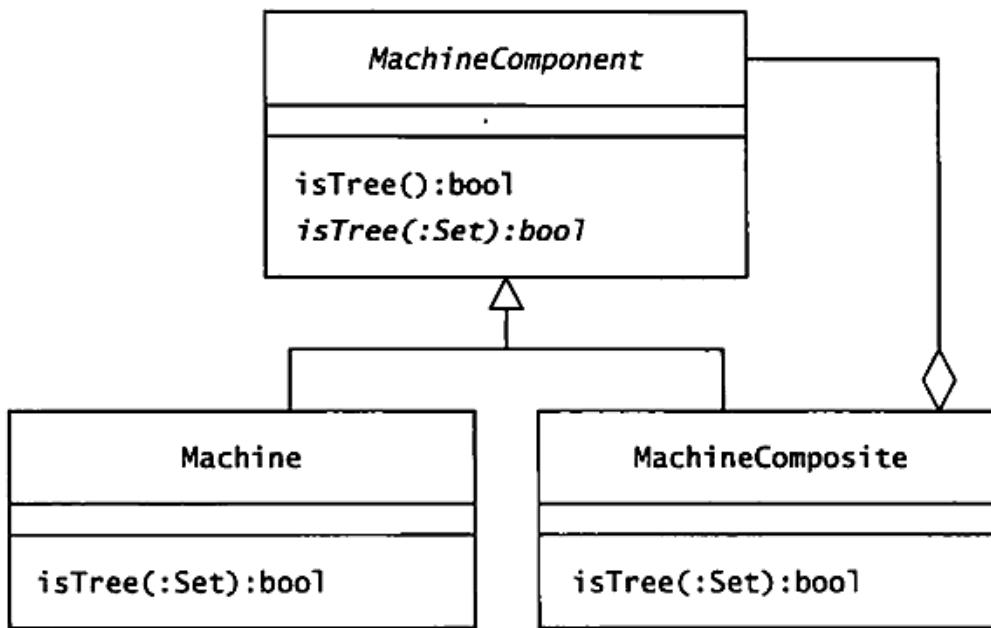


图 20.1 `isTree()`中的 4 个方法相互协作，以判断一个 `MachineComponent` 实例是否是一棵树

算法完成了一些事情，它可能是方法的一部分，也可能调用了很多方法。在面向对象的设计中，需要多个方法参与的算法通常依赖于多态，因为多态机制允许一个操作具有多个不同的实现。多态是一个既依赖于调用的操作，又依赖于调用接收者类型的一种方法调用的原则。例如，当执行到表达式 `m.isTree()` 时，到底哪个方法会被执行呢？答案取决于它的依赖关系。如果 `m` 是 `Machine` 类的实例，Java 会调用 `Machine.isTree()`。如果 `m` 是 `MachineComposite` 类的实例，Java 会调用 `MachineComposite.isTree()`。通俗地讲，多态是指合适的对象调用合适的方法。这在某些情形下，正是设计模式使用多态的直接意图。

小结

尽管混淆操作、方法、签名和算法的做法普遍存在，但是，弄清楚这些术语有助于我们描述一些重要的概念。操作和签名一样，是服务的规格说明。操作意味着许多方法拥有相同的接口。签名意味着方法的查询规则。方法的定义包括它的签名——方法名和参数列表，以及访问修饰符、返回类型与方法体。方法拥有签名和操作的实现。

调用方法的常规做法是直接调用。结束方法调用的常规做法是使其返回。但是在出现未处理的异常时，任何方法都会停止执行。

算法是一个接收输入和产生输出的过程。方法接收输入，产生输出，并且包含了一个过程的方法体。因此有人将方法看做是一个算法。然而，一个算法过程可能包括多个操作和方法，或者属于其他方法的一部分。当你在讨论一个过程会产生一个结果时，最适合用算法来表达。

许多设计模式都会将一个操作分散设计在多个类中。你也可以说这些模式依赖于多态，即调用的方法取决于传入的对象类型。

超越常规的操作

不同类在实现同一个操作时采用不同的方式。换言之，Java 支持多态。这看似简单的想法却被多种设计模式所使用。表 20.1 列出了不同场景适用的模式

表 20.1 不同场景适用的模式

| 如果你的意图是 | 使用的模式 |
|---------------------------------|--------|
| • 在方法中实现算法，推迟对算法步骤的定义使得子类能够重新实现 | 模板方法模式 |
| • 将操作分散，使得每个类都能够表示不同的状态 | 状态模式 |
| • 封装操作，使得实现是可以互相替换的 | 策略模式 |
| • 用对象来封装方法调用 | 命令模式 |
| • 将操作分散，使得每个实现能够运用到不同类型的集合中 | 解释器模式 |

面向操作的模式适合于这类场景：设计中需要多个具有相同签名的方法。例如，模板方法模式允许子类重新实现及调整父类已经定义好的方法。

第21章

模板方法（Template Method）模式

普通方法的方法体定义了一系列的指令。方法常常会调用当前对象或其他对象的其他方法。此时，可以将该普通方法看做“模板”，它给出了计算机执行的指令序列。然而，就模板方法模式而言，它还引入了一种更为特殊的模板。

在编写某个方法时，可能需要先定义出算法的轮廓，因为该算法的实现可能五花八门。在这种情况下，就可以定义一个方法，将它的一些步骤定义为抽象方法，或者是存根方法；又或者将它们定义到一个独立的接口中。这样就产生了一个更为严格的“模板”，它特别指出了算法中的哪些步骤是由其他类提供的。

模板方法模式的意图是在一个方法里实现一个算法，并推迟定义算法中的某些步骤，从而让其他类重新定义它们。

经典范例：排序

排序是一种常用的经典算法。假设一位古代女工设计了一种可以根据箭簇的锋利程度对箭进行排序的算法。假定她将箭从左到右排成一排，逐个进行比较，并将更为锋利的箭放在左侧。设计完该算法后，她又认为或许可以根据箭的飞行距离或者其他属性来重新设计排序算法。

不同排序算法的方式和速度是不相同的，但无论是哪种排序算法，都遵循比较两个元素或者属性的基础步骤。如果你设计了一个排序算法，可以比较任意两个元素的某个属性，就可以

通过该属性对元素的集合进行排序。

排序算法是模板方法模式的经典范例。只需修改一个关键步骤——两个对象的比较，就可以对其他对象的各种集合的各种属性进行算法的重用。

近些年来，排序算法可能是重新实现频率最高的算法之一，实现次数或许已经超过了当今程序员的数量。但是，除非你需要对一个超大的集合排序，否则并不需要实现自己的排序算法。

`Arrays` 类和 `Collections` 类都提供了 `sort()` 方法。这是一个静态方法，使用一个数组作为参数，并且接收一个可选的比较器 `Comparator`。`ArrayList` 类的 `sort()` 方法是一个静态方法，会对 `sort()` 消息的接收者进行排序。换句话说，这些方法共享一个公共策略，该策略依赖于 `Comparable` 和 `Comparator` 接口。如图 21.1 所示。

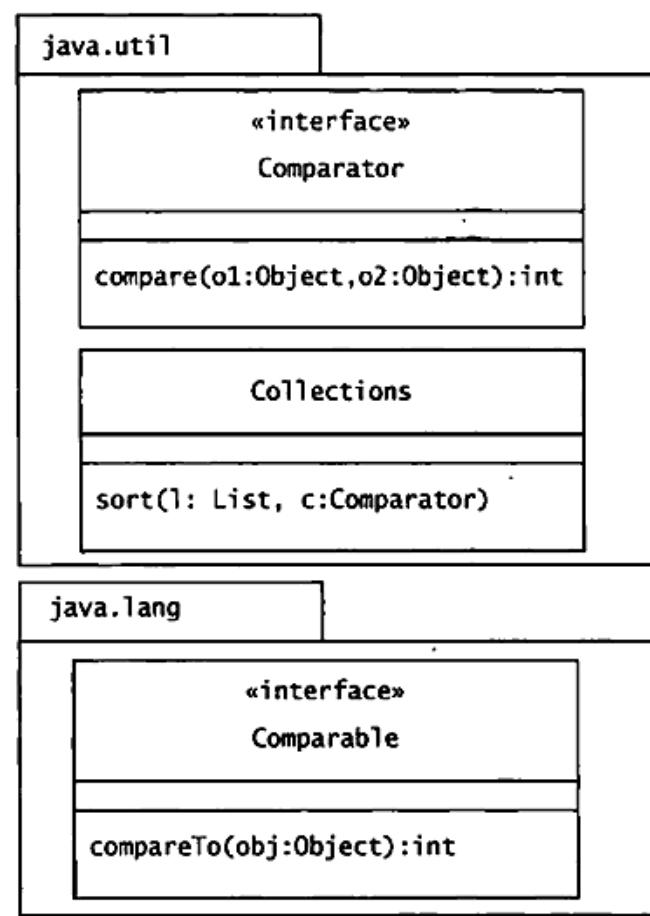


图 21.1 Collections 类中的 `sort()` 方法使用图中的接口

如果需要，`Arrays` 类和 `Collections` 类的 `sort()` 方法可以提供一个 `Comparator` 接口的实例。如果在使用 `sort()` 方法时，没有提供 `Comparator` 接口的实例，该方法将会依赖 `Comparable` 接口的 `compareTo()` 方法。如果某些元素没有实现 `Comparable` 接口，而你却又

未曾提供 `Comparator` 接口实例，对这些元素排序就会出现异常。当然，在 Java 语言中，绝大多数基础类型，如 `String` 都实现了 `Comparable` 接口。

`sort()` 方法为模板方法模式提供了一个范例：类库提供算法，可以让你自己实现比较两个元素的关键步骤。`Compare()` 方法返回一个小于、等于或者大于 0 的数字。这些数字分别对应着对象 `o1` 小于、等于或者大于对象 `o2`。例如，下面的代码分别根据火箭的最远点和名称对火箭集合进行排序（火箭构造函数接收的参数包括名字、质量、价格、最远点以及推动力）。

```
package app.templateMethod;

import java.util.Arrays;
import com.oozinoz.firework.Rocket;
import com.oozinoz.utility.Dollars;

public class ShowComparator {
    public static void main(String args[]) {
        Rocket r1 = new Rocket(
            "Sock-it", 0.8, new Dollars(11.95), 320, 25);
        Rocket r2 = new Rocket(
            "Sprocket", 1.5, new Dollars(22.95), 270, 40);
        Rocket r3 = new Rocket(
            "Mach-it", 1.1, new Dollars(22.95), 1000, 70);
        Rocket r4 = new Rocket(
            "Pocket", 0.3, new Dollars(4.95), 150, 20);
        Rocket[] rockets = new Rocket[] { r1, r2, r3, r4 };

        System.out.println("Sorted by apogee: ");
        Arrays.sort(rockets, new ApogeeComparator());
        for (int i = 0; i < rockets.length; i++)
            System.out.println(rockets[i]);
        System.out.println();
        System.out.println("Sorted by name: ");
        Arrays.sort(rockets, new NameComparator());
        for (int i = 0; i < rockets.length; i++)
            System.out.println(rockets[i]);
    }
}
```

下面是 `ApogeeComparator` 比较器的代码：

```
package app.templateMethod;
```

```
import java.util.Comparator;
import com.oozinoz.firework.Rocket;

public class ApogeeComparator implements Comparator {
    // 挑战!
}
```

NameComparator 的代码如下：

```
package app.templateMethod;

import java.util.Comparator;
import com.oozinoz.firework.Rocket;

public class NameComparator implements Comparator {
    // 挑战!
}
```

程序的输出取决于 Rocket 是如何实现 `toString()` 方法的，但火箭的显示则依照两种排序方式：

Sorted by apogee:

Pocket
Sprocket
Sock-it
Mach-it

Sorted by name:

Mach-it
Pocket
Sock-it
Sprocket

挑战 21.1

请补充完成 `ApogeeComparator` 类和 `NameComparator` 类的代码，使程序可以正确地对火箭集合进行排序。

答案参见第 347 页

排序是一种通用算法，除了一个关键步骤外，和你的领域与应用几乎没有关系。该关键步骤就是对元素的排序。没有哪个通用排序算法包括诸如比较两个火箭最远点的步骤，而是由应用程序来实现这一步骤。`Sort()`方法和`Comparator`接口可以为通用排序算法提供对该特定步骤的支持。

模板方法模式并不局限于仅缺少领域步骤的情况。有时，会将整个算法应用到具体的应用领域。

完成一个算法

模板方法模式与适配器模式类似，它们都允许开发者简化代码，并允许其他开发者来完成代码的设计。在适配器模式中，设计者可能会为特定对象指定设计需要的接口，而由其他开发者提供该接口的实现。在实现时，应使用实现了不同接口的现有类所提供的服务。在模板方法模式中，设计者可能会提供一个通用的算法，而其他开发者仅提供该算法关键步骤的实现。考虑图 21.2 所示的 Aster 火药球填压机。

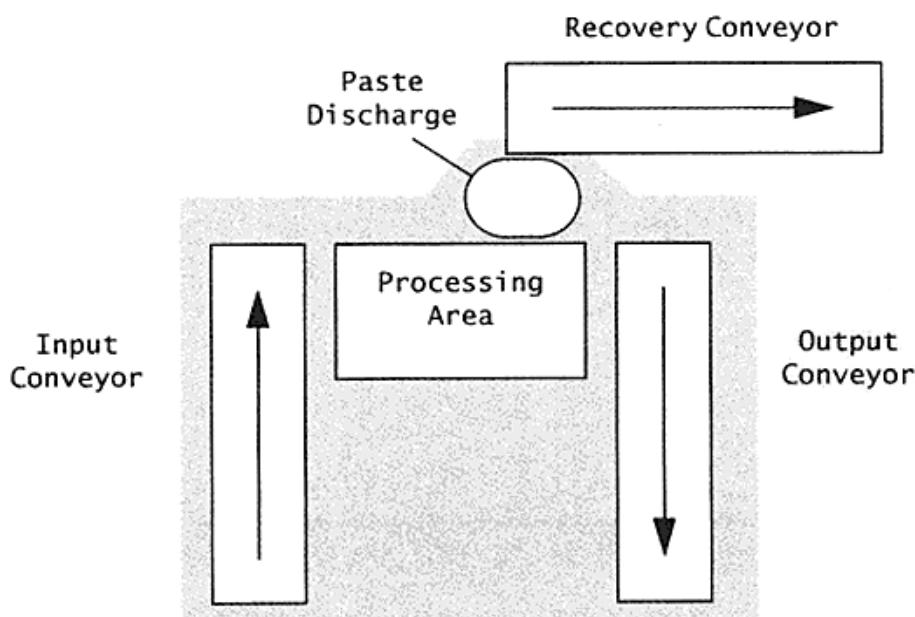


图 21.2 Aster 火药球填压机利用传送带的输入输出来传送火药球模具。Oozinoz 公司增加了一个回收传送带，用于回收被丢弃的火药糊

Aster 公司的火药球填压机接收一个空的金属模具，并填充火药。这个机器有一些料斗，可

以将化学药品混合成糊，然后压进模具。当机器关闭时，它将停止在处理区工作的模具。输入传送带上的模具不会被处理，而是被送到输出传送带上。接着，填压机将卸下当前的火药糊，并用水清洗工作区。填压机通过板载的计算机程序来完成这些操作。**AsterStarPress** 类如图 21.3 所示。

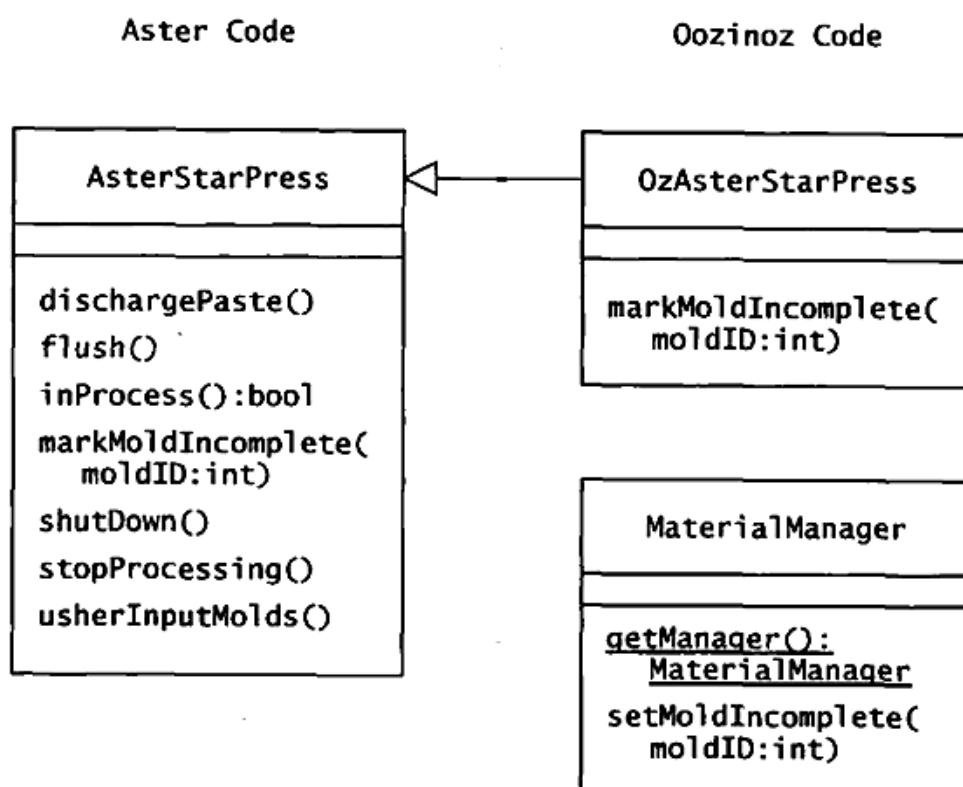


图 21.3 Aster 公司的火药球填压机有一个抽象类，在 Oozinoz 系统中，必须实现该抽象类才能工作

Aster 公司的火药球填压机属于可独立运行的智能机器，公司意识到该机器可能会被用于智能化工厂，并且需要具备通信能力。例如，如果正在处理的填压机尚未处理完毕，**shutDown()** 方法会通知工厂。

```
public void shutdown() {
    if (inProcess()) {
        stopProcessing();
        markMoldIncomplete(currentMoldID);
    }
    usherInputMolds();
    dischargePaste();
    flush();
}
```

`markMoldIncomplete()`方法和 `AsterStarPress` 类都是抽象的。Oozinoz 系统需要创建一个子类来实现这些方法，并将这些代码下载到火药球填压机的板载计算机中。可以通过向 `MaterialManager` 单例传递未完成的模具信息，实现 `markMoldIncomplete()` 方法，以便跟踪原料的状态。

挑战 21.2

写出 `OzAsterStarPress` 类的 `markMoldIncomplete()` 代码实现：

```
public class OzAsterStarPress extends AsterStarPress {  
    public void markMoldIncomplete(int id) {  
        // 挑战!  
    }  
}
```

答案参见第 347 页

Aster 公司负责开发火药球填压机的技术人员对焰火工厂的工作方式了如指掌，并完成了填压机与工厂之间的通信工作。但是，依然有些地方是 Aster 公司的开发人员所没有预见到的，因此需要你去完成这些通信点。

模板方法钩子

钩子（hook）是一个方法回调，它可以让其他开发者将自己的代码插入到程序的指定位置。当你希望使用其他开发者的代码时，或者希望在别的程序中插入自己的代码时，就可以使用钩子。开发者可以在你需要的时候增加一个方法调用。通常，开发者会为钩子方法提供一个存根实现。一旦其他客户端不再需要钩子方法，就没有必要再重写它了。

考虑如下情况：当 Aster 公司的火药球填压机被关闭时，它会倒出火药糊，并自动用水清洗自身。压缩机必须倒出火药糊，以防止它干燥之后会堵塞机器。Oozinoz 公司会回收这些火药糊，以便将它们切成罗马蜡烛中的小火药球（罗马蜡烛是一个固定立方体，它由炸药混合而成，点燃后会有焰火效果）。在火药填压机倒出火药糊后，可以安排一个机器人来将火药糊移动到一个独立的传送带上，如图 21.2 所示。有一点至关重要，就是一定要在机器冲洗工作区前将

火药糊移走。这带来的问题是：需要在 `shutdown()` 方法的两条语句之间获得控制。

```
dischargePaste();  
flush();
```

你可能需要重写 `dischargePaste()` 方法，使之增加一个调用，用于收集火药糊。

```
public void dischargePaste() {  
    super.dischargePaste();  
    getFactory().collectPaste();  
}
```

这个方法会在倒出火药糊之后插入一个步骤。增加的步骤使用一个单例工厂方法来收集倾倒的火药糊。执行 `shutdown()` 方法时，工厂机器人会在清洗填压机前收集完所有倾倒的火药糊。不幸的是，`dischargePaste()` 方法存在危险性，因为收集火药糊的方法会产生一些边际效应。Aster 公司的程序员们显然并不知道你是如何定义 `dischargePaste()` 方法的。如果他们修改代码，使得在你不想收集火药糊的时候倾倒它，就会产生错误。

程序员们都会竭力编写代码来解决问题。但是，这里面临的挑战是需要在与其他程序员沟通过后，再编写代码解决问题。

挑战 21.3

请为 Aster 公司的程序员写一个通知，要求他们在清洗工作区前，确保已经安全地收集完倾倒的火药糊。

答案参见第 348 页

在模板方法模式中，子类支持的步骤可能被用来完成某些算法；但也可能这些步骤是可选的，通常是根据其他程序员的要求，在子类代码中实现这些钩子方法。尽管该模式的意图是将算法中的某部分分离为一个单独的类，但是，倘若程序多处出现重复的算法方法，也可以将它们重构为模板方法模式。

重构为模板方法模式

在使用模板方法模式时，你会发现在类继承关系中，超类提供的是算法的描述，而子类提

供的是算法具体的步骤。当发现不同的方法具备相似的算法时，就可以将它们重构为模板方法模式（重构是在不改变程序功能的前提下，用更好的设计来重新实现）。考虑第 16 章中的 Machine 和 MachinePlanner 的并行层次。如图 21.4 所示，Machine 类提供了一个 createPlanner() 工厂方法，返回一个合适的 MachinePlanner 类的子类。

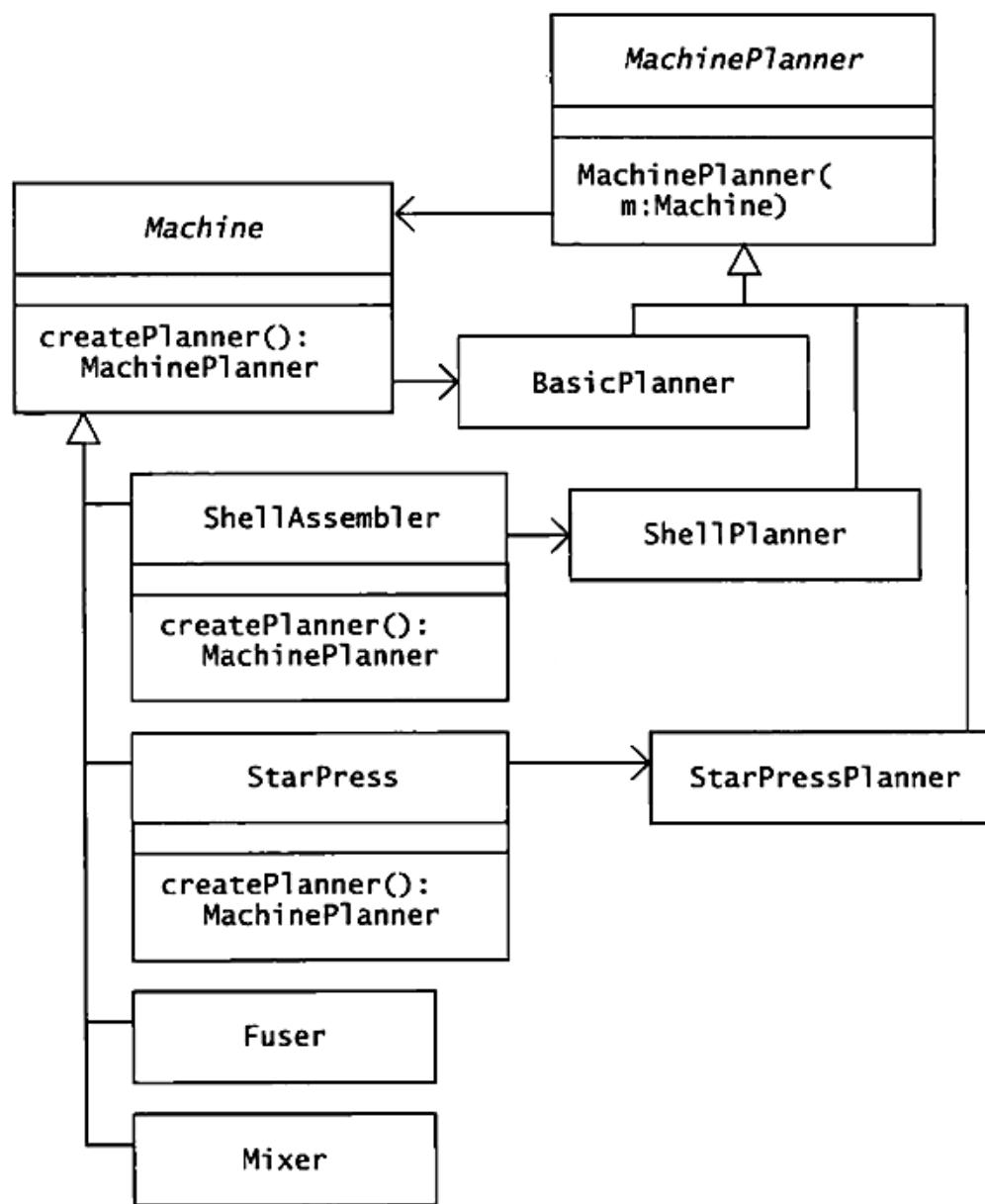


图 21.4 Machine 对象可以为自己创建合适的 MachinePlanner 对象实例

当要求创建一个规划器时，Machine 类的两个子类会实例化 MachinePlanner 类层次关系中指定的子类。ShellAssembler 类和 StarPress 这两个类有一个共同的需求：它们都希望在需要的时候创建 MachinePlanner 类。

阅读这些类的代码，你会发现子类都使用了同样的延迟初始化技术来创建规划器。例如，

`ShellAssembler` 类有一个 `getPlanner()` 方法来延迟初始化 `planner` 成员。

```
public ShellPlanner getPlanner() {  
    if (planner == null)  
        planner = new ShellPlanner(this);  
    return planner;  
}
```

在 `ShellPlanner` 类中，`planner` 的类型是 `ShellPlanner`。`StarPress` 类也有一个 `planner` 成员，但是类型为 `StarPressPlanner`。`StarPress` 类的 `getPlanner()` 方法也会延迟初始化 `planner` 属性：

```
public StarPressPlanner getPlanner() {  
    if (planner == null)  
        planner = new StarPressPlanner(this);  
    return planner;  
}
```

`Machine` 类的其他子类使用了相同的技术，仅在第一次需要使用规划器的时候才创建它。这就可以通过重构来清理代码，减少我们需要维护的代码。假设你决定提供一个 `Machine` 类，其中包含类型为 `MachinePlanner` 的 `planner` 属性，就可以从子类中删除这个属性，以及已有的 `getPlanner()` 方法。

挑战 21.4

请写出 `Machine` 类中 `getPlanner()` 方法的代码。

答案参见第 348 页

我们经常可以通过抽象出相似的方法轮廓，将其移到超类，并让不同的子类实现算法所需要的不同步骤，从而将代码重构为模板方法模式。

小结

模板方法模式的意图是在一个方法里定义一个算法，抽象某些步骤，或者将它们定义在接

口中，以便其他类可以实现这些步骤。

模板方法模式可以作为开发者之间的一种契约。一些开发者开发算法的框架，而另一些开发者负责实现算法的具体步骤。这些步骤可以是完成算法所必需的步骤，也可以是算法开发者在程序中特定位置设置的钩子。

模板方法模式的意图并不是要求我们在定义子类前写出模板方法。你可能会在已有的类层次关系中发现一些相似的方法。在这种情况下，你可以提取算法的框架，并将它移到超类中，从而运用模板方法模式来简化和组织你的代码。

第 22 章

状态（State）模式

对象状态是指对象属性的当前值的组合。在调用对象的 `set` 方法，或给对象的成员变量赋值时，都是在改变对象的状态。通常在执行对象的方法时，其自身状态也会改变。

我们通常使用状态一词来代表对象中独立的、可改变的属性。例如，我们可以说机器的状态是打开还是关闭。此时，对象状态中的可变部分可能是它所拥有的行为中最突出的部分。因此，与对象状态相关的逻辑可能在许多类的方法中蔓延。多次重复出现相似或者相同的逻辑，会带来维护上的沉重负担。

有一种办法可以获知与状态相关的逻辑分布情况：引入一组新的类，每个类代表不同的状态，然后将与状态相关的行为分别写到每个类中。

状态模式的意图是将表示对象状态的逻辑分散到代表状态的不同类中。

对状态进行建模

对那些更为关注状态的对象进行建模时，需要一个依赖于状态的变量，来跟踪对象的行为。这个变量可能会出现在复杂的、层叠的 `if` 语句中，该语句可以对对象收到的事件进行处理。使用这种方式对状态建模，会让代码变得很复杂，尤其是当调整状态模型时，不得不调整很多方法的代码。状态模式利用分散的操作提供了一种整洁简单的操作方式。它将状态建模为对象，

并把状态相关的逻辑封装到独立的类中。为了了解状态模式的工作方式，我们在系统建模时先不使用状态模式。在下一小节中，我们对这些代码进行重构，以此验证状态模式是否能改进我们的设计。

假设 Oozinoz 公司要对传送带的入口门进行建模。传送带是一个大型的智能设备，它上面有一个入口门可以接收原料，并且通过原料上的条形码对原料进行存储。入口门只有一个操作按钮。如果门关闭，按键后门会自动打开。如果在门尚未完全打开前再次按键，门将再次关闭。如果门一直开着，2 秒后门会自动超时关闭。可以在门打开的时候通过再次按键来阻止门的自动关闭。图 22.1 展示了传送带入口门的状态和转换效果。

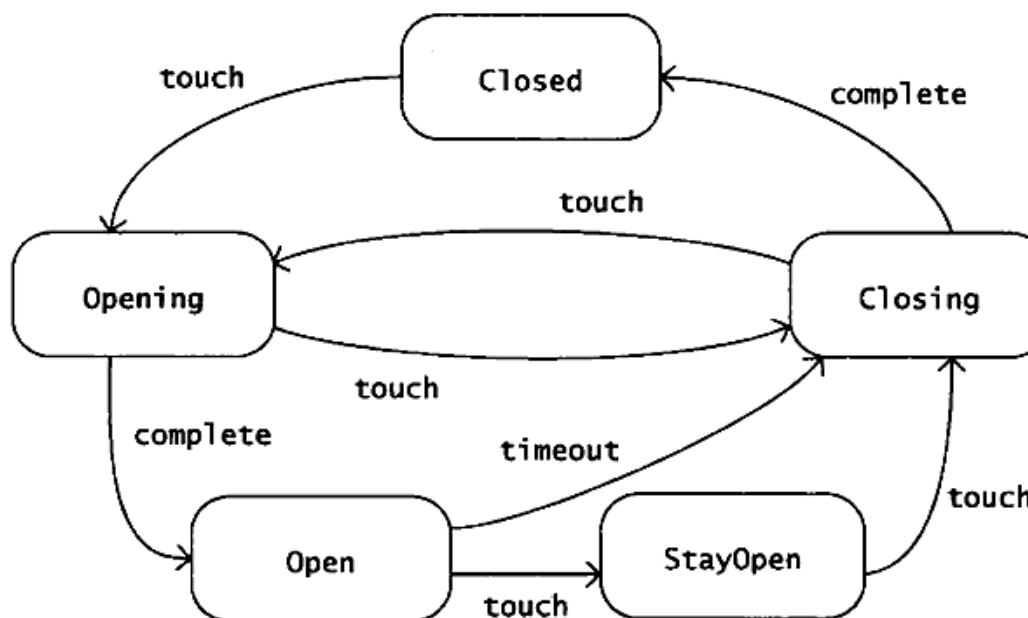


图 22.1 传送带入口门通过触发一个单独的按钮来对门的状态进行控制

图 22.1 是一个 UML 状态图。相对于文字表达来讲，这幅图描述得更加清楚。

挑战 22.1

假设打开了传送带入口门，并将材料投进去。请问除了等待超时让门自动关闭外，是否还有其他方式可以关门？

答案参见第 349 页

可以为传送软件提供一个 `Door` 对象，随着状态的变化，传送软件可以更新对象的状态。

图 22.2 展示了 `Door` 类。

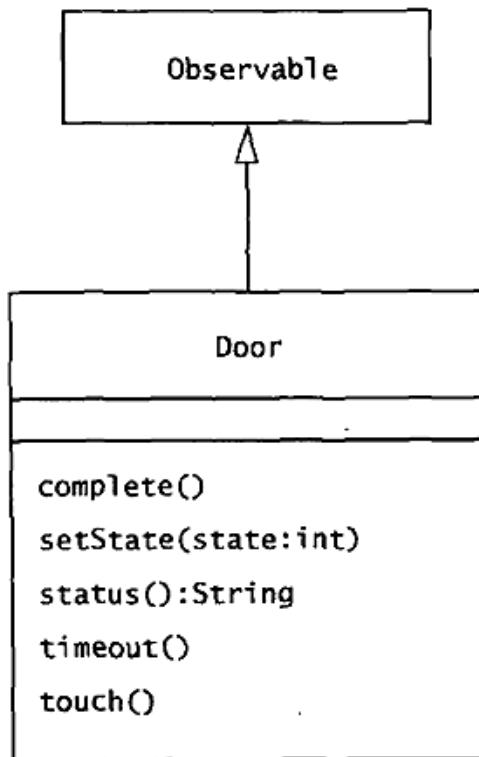


图 22.2 Door 类依赖于传送带触发的状态改变事件，对传送带入口门进行建模

Door 类是 observable 的子类，诸如 GUI 等客户对象，可以显示门的状态。该类中定义了门的几种状态：

```
package com.oozinoz.carousel;
import java.util.Observable;

public class Door extends Observable {
    public final int CLOSED = -1;
    public final int OPENING = -2;
    public final int OPEN = -3;
    public final int CLOSING = -4;
    public final int STAYOPEN = -5;

    private int state = CLOSED;
    // ...
}
```

（如果使用 Java 5，可以选择使用枚举类型。）

显而易见，状态的文本描述取决于门所处的状态。

```
public String status() {
    switch (state) {
```

```
    case OPENING:
        return "Opening";
    case OPEN:
        return "Open";
    case CLOSING:
        return "Closing";
    case STAYOPEN:
        return "StayOpen";
    default:
        return "Closed";
    }
}
```

当用户触发传送带入口门的按键时，程序会调用 `Door` 对象的 `touch()` 方法。图 22.1 模拟了 `Door` 类中的状态转移。

```
public void touch() {
    switch (state) {
        case OPENING:
        case STAYOPEN:
            setState(CLOSING);
            break;
        case CLOSING:
        case CLOSED:
            setState(OPENING);
            break;
        case OPEN:
            setState(STAYOPEN);
            break;
        default:
            throw new Error("can't happen");
    }
}
```

`Door` 类中的 `setState()` 方法负责通知那些监听门状态改变的观察者。

```
private void setState(int state) {
    this.state = state;
    setChanged();
    notifyObservers();
}
```

挑战 22.2

请写出 Door 类中 complete() 和 timeout() 的代码。

答案参见第 349 页

重构为状态模式

Door 类的代码有些复杂，因为 state 变量是类的全局变量。此外，你可能发现很难比较状态转移的方法，特别是图 22.1 所示的状态机中的 touch() 方法。此时，引入状态模式可以帮助你简化代码。要在本例中使用状态模式，需要将门的每种状态都写到一个独立的类中，如图 22.3 所示。

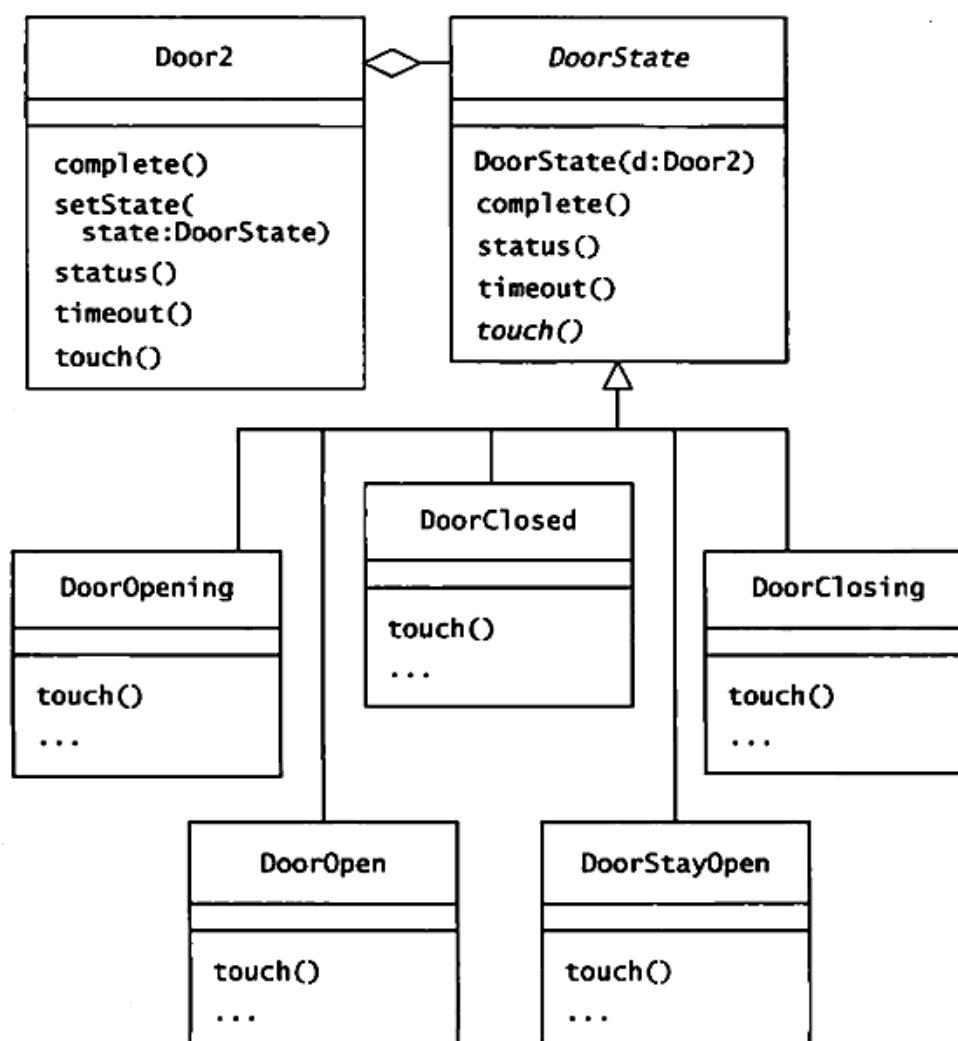


图 22.3 该图展示了将门的状态机映射成独立类后的样子

图 22.3 展示的重构将门的各种状态分别创建成相应的类。每个类都包含相应的逻辑来处理状态行为，即当门处于某个特定状态时，触动按钮后应有的行为。例如，DoorClosed.java 文件包含如下代码：

```
package com.oozinoz.carousel;
public class DoorClosed extends DoorState {
    public DoorClosed(Door2 door) {
        super(door);
    }
    public void touch() {
        door.setState(door.OPENING);
    }
}
```

DoorClosed 类中的 touch() 方法通知 Door2 门对象的最新状态。Door2 对象通过 DoorClosed 类的构造函数传入。当前的设计要求每个状态对象都要保留对 Door2 对象的引用，确保状态对象可以通知门对象的状态变化。当前设计要求每个状态对象都引用某个特定的门对象，以便每个状态对象只能操作一个单独的门对象。下一节将指出如何修改该设计，以便一组单独的状态集可以满足任意数量的门对象。当前的设计则要求在创建 Door2 对象的同时，创建一组属于门对象的状态集。

```
package com.oozinoz.carousel;
import java.util.Observable;

public class Door2 extends Observable {
    public final DoorState CLOSED = new DoorClosed(this);
    public final DoorState CLOSING = new DoorClosing(this);
    public final DoorState OPEN = new DoorOpen(this);
    public final DoorState OPENING = new DoorOpening(this);
    public final DoorState STAYOPEN = new DoorStayOpen(this);

    private DoorState state = CLOSED;
    // ...
}
```

抽象类 DoorState 需要一些子类来实现 touch() 方法。每个状态类都有一个 touch() 转换函数，这和状态机是一致的。DoorState 类的其他转换方法都是无关方法，因此子类可以重写或者忽略不相干的消息。

```
package com.oozinoz.carousel;

public abstract class DoorState {
    protected Door2 door;

    public abstract void touch();

    public void complete() { }

    public void timeout() { }

    public String status() {
        String s = getClass().getName();
        return s.substring(s.lastIndexOf('.') + 1);
    }

    public DoorState(Door2 door) {
        this.door = door;
    }
}
```

注意，所有的状态对象都包含了 `status()` 方法，它比重构之前要简单许多。

从接收状态的改变来看，新的设计并没有改变 `Door2` 对象的角色。现在，`Door2` 对象只是简单地将这些改变传递给当前的 `state` 对象：

```
package com.oozinoz.carousel;
import java.util.Observable;

public class Door2 extends Observable {
    // 变量与构造函数 ...

    public void touch() {
        state.touch();
    }

    public void complete() {
        state.complete();
    }
}
```

```
public void timeout() {
    state.timeout();
}

public String status() {
    return state.status();
}

protected void setState(DoorState state) {
    this.state = state;
    setChanged();
    notifyObservers();
}
}
```

在当前设计中，`touch()`、`complete()`、`timeout()`和 `status()`方法展示了多态性的应用。每个方法都表示一种状态的迁移。操作是固定的，但是接收的类——状态类，却是不一样的。多态的原则是方法的执行依赖于接收的类以及该类中的方法签名。当调用 `touch()` 方法时会发生什么呢？答案取决于门对象的状态。代码依然可以有效地进行转换，由于引入了多态，代码比之以前更为简洁。

`Door2` 类中的 `setState()` 方法现在被用于 `DoorState` 的子类。这些子类类似于图 22.1 状态机中对应的状态。例如，`Dooropen` 类处理有关 `touch()` 和 `timeout()` 的调用，对应状态机中 `Open` 状态的两个转换：

```
package com.oozinoz.carousel;
public class DoorOpen extends DoorState {
    public DoorOpen(Door2 door) {
        super(door);
    }

    public void touch() {
        door.setState(door.STAYOPEN);
    }

    public void timeout() {
        door.setState(door.CLOSING);
    }
}
```

挑战 22.3

写出 `DoorClosing.java` 的代码。

答案参见第 349 页

新的设计使代码变得更加简单，但或许你对现在的设计仍不满意，因为 `Door` 类使用的“常量”实际上是一个局部变量。

使状态成为常量

状态模式将状态相关的逻辑转移到独立的类中，用以表示对象的状态。然而该模式并没有指出如何管理状态和主对象之间的通信和依赖关系。在之前的设计中，每个状态类都在构造函数中接收一个 `Door` 对象。状态对象保存这个对象的引用，并利用该对象更新门的状态。这样的设计并不算坏，但它确实造成了在实例化 `Door` 对象时，伴随着对一组 `DoorState` 对象的实例化。若能创建一组独立的静态 `DoorState` 对象，然后让 `Door` 状态管理因为状态改变产生的所有更新，会是更好的一种设计。

使状态对象变成常量的一种做法是，让当前的状态类标识下一个状态，让 `Door` 类去更新 `state` 变量。根据这样的设计，`Door` 类的 `touch()` 方法就会采用如下方式更新 `state` 变量：

```
public void touch() {
    state = state.touch();
}
```

注意，`Door` 类的 `touch()` 方法返回了 `void`，`DoorState` 的子类也会实现 `touch()` 方法，但是，这些方法会返回 `DoorState` 值。例如，`DoorOpen` 类的 `touch()` 方法：

```
public DoorState touch() {
    return DoorState.STAYOPEN;
}
```

在这个设计中，`DoorState` 对象不包含任何 `Door` 对象的引用。因此，应用程序的所有 `DoorState` 对象只需要引用一个 `Door` 对象的实例。

使 `DoorState` 对象变成常量的另一种方式是在状态迁移时传递核心的 `Door` 对象。你可以为 `complete()`、`timeout()`、`touch()` 这些与状态改变有关的方法都加上一个 `Door` 参数。这些方法接收一个 `Door` 对象来更新其状态，而不需要在它的内部保留 `Door` 对象的引用。

挑战 22.4

完成图 22.4，将 `DoorState` 对象设计为常量，并在状态迁移时传递 `Door` 对象。

答案参见第 350 页

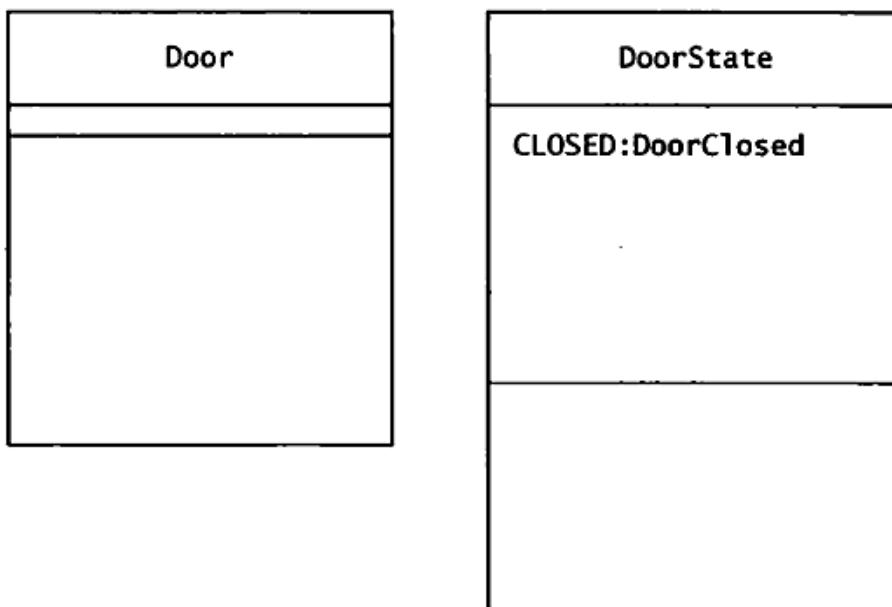


图 22.4 完成这个设计图，使之可以表示连续的门状态

在应用状态模式时，可以针对状态改变时的通信机制进行完全自由的设计。状态类维护了一个主对象的引用，该对象的状态已经建模。或者，可以在状态迁移时传递这个对象，也可以让这些状态子类决定下一个状态，但不更新主对象。使用哪种方式取决于你的应用场景或者个人喜好。

如果这些状态类在多线程环境下使用，要确保这些状态迁移方法是同步的，并保证在同一时间，两个线程在更改状态时不会有冲突。

状态模式的作用是让任何一个状态的逻辑都集中放在一个单独的类中。

小结

总体而言，对象的状态取决于对象实例中变量值的集合。在一些场景下，对象中的绝大多数属性一旦被赋值后都不会改变，但其中某个属性值经常变化，并在类逻辑中扮演重要角色。该属性可能会代表整个对象的状态，甚至被命名为 `state`。

在对现实世界中拥有重要状态的实体进行建模时，可能会发生一些显著的状态迁移，例如事务或机器的状态迁移。在这些场景下，依赖于对象状态的逻辑可能会分布在多个方法中。可以通过将状态相关的行为转移到状态对象类中，以此来简化这些代码。这会让每个状态类都包含领域中一个状态的行为，同时也让状态类对应状态机中的相应状态。

为了处理状态间的迁移，可以让主对象包含一组状态的引用。或者在状态迁移的调用中，将主对象传递给状态改变的类。你也可以让状态类的信息提供者仅仅给出下一个状态，而不去更新主对象。无论怎样管理状态迁移，状态模式都会通过将对象的不同状态操作，分散到一组类的集合中，从而简化代码。

第 23 章

策略（Strategy）模式

策略模式是一个计划或者方式，根据给定的输入条件达成一个目标。策略和算法很相似，算法是一段程序，它可以对一组输入进行处理，获得一个输出。通常情况下，策略提供的范围比算法要广泛。这就意味着策略通常会提供一组或者一族可互换的方法。

当一个计算机程序存在多种策略时，代码就会变得复杂。要从一组策略中选择合适的策略，选择逻辑自身可能会很复杂。当多种策略的选择与执行导致复杂度增加时，就可以使用策略模式来简化它。

策略操作定义了策略的输入与输出，实现则由各个独立的类完成。这些类的实现虽然不同，但是由于接口是一致的，因此可以使用相同的接口给用户提供不同的策略进行互换。策略模式可以让一组策略共存，代码互不干扰。它还将选择策略的逻辑从策略本身中分离了出来。

策略模式的意图是将可互换的方法封装在各自独立的类中，并且让每个方法都实现一个公共的操作。

策略建模

策略模式通过将问题的不同解决方法封装在不同的类，以帮助我们组织和简化代码。为了理解策略模式是如何工作的，首先来了解一下不使用它的情况下，如何对策略进行建模。在下

一小节，我们再重构这块代码，使用策略模式来提高代码质量。

考虑 Oozinoz 公司焰火的广告策略，当用户访问该公司的网站或者给呼叫中心打电话时，都会建议客户购买该焰火产品。Oozinoz 公司使用两款现有的商业推荐引擎，帮助选择正确的广告，然后推荐给客户。Customer 类负责选择与使用其中的一款引擎，来决定给客户推荐哪一种焰火。

Re18 是其中的一款推荐引擎，它基于用户之间的相似度进行推荐。为使该引擎能正常工作，需要让已注册的用户填写焰火弹及其他娱乐设施的相关预设信息。

如果用户没有注册，Oozinoz 公司使用另一款供应商提供的 LikeMyStuff 引擎向用户推荐，它根据用户最近购买的产品进行决策。如果系统收集的数据太少，不足以提供推荐，该软件将会随机选择一个焰火广告进行投放。然而，Oozinoz 公司出于销售方面的考虑，可能需要推出某项促销业务，而该促销业务的规则将会覆盖所有引擎的业务规则，并推出指定的焰火广告。图 23.1 展示了各个类之间的关系，用于向客户提供焰火广告。

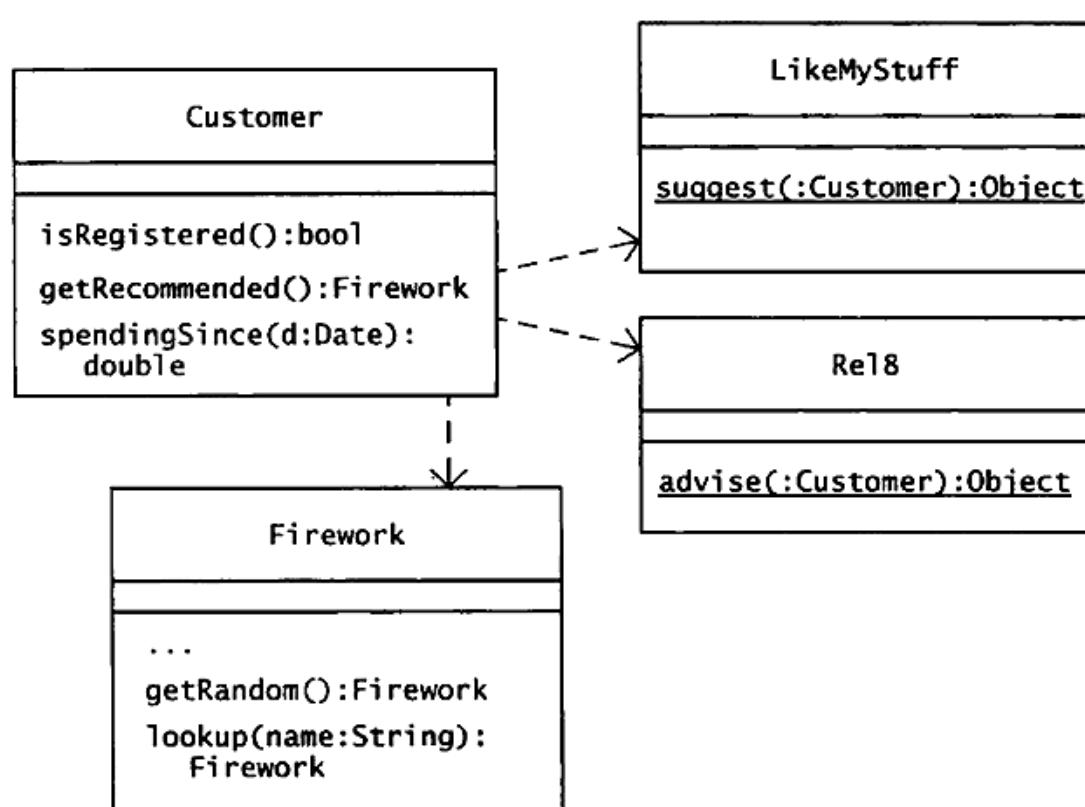


图 23.1 Customer 类依赖于其他类来获取推荐的焰火广告，包括两个现成的推荐引擎

LikeMyStuff 引擎和 Re18 引擎都接收一个 Customer 对象，向客户推荐广告。Oozinoz 公司的这两款引擎的配置均用于提供焰火弹广告，但 LikeMyStuff 引擎依赖于数据库，而 Re18 则依赖于对象模型。Customer 类的 getRecommended() 方法体现了 Oozinoz 公司的广告策略，代码如下：

```
public Firework getRecommended() {
    // 如果对指定的焰火进行了促销，则返回
    try {
        Properties p = new Properties();
        p.load(ClassLoader.getSystemResourceAsStream(
            "config/strategy.dat"));
        String promotedName = p.getProperty("promote");

        if (promotedName != null) {
            Firework f = Firework.lookup(promotedName);
            if (f != null) return f;
        }
    } catch (Exception ignored) {
        // 如果资源丢失或加载失败，就进入下一种方式
    }

    // 如果注册了，则与其他客户进行比较
    if (isRegistered()) {
        return (Firework) Re18.advise(this);
    }

    // 检查上一年的花费
    Calendar cal = Calendar.getInstance();
    cal.add(Calendar.YEAR, -1);
    if (spendingSince(cal.getTime()) > 1000)
        return (Firework) LikeMyStuff.suggest(this);

    // 好的，不错！
    return Firework.getRandom();
}
```

这段代码放在 Oozinoz 代码库的 com.oozinoz.recommendation 包中，可以从 www.oozinoz.com 处获取。如果对某次焰火弹进行过促销，getRecommended()方法期望将焰火弹的名字保存在 config 目录的 strategy.dat 文件中。如下所示：

```
promote=JSquirrel
```

如果该文件不存在，getRecommended()的代码在用户已经注册的前提下，将会使用 Re18 引擎。如果没有促销策略，客户也没有注册，但用户在过去一年内通过本系统购买过商品，代码将会使用 LikeMyStuff 引擎。如果所有的推荐条件都不具备，系统将会随机选择推荐一种焰

火炮。`getRecommended()`方法可以工作，也许你认为它并不糟糕，但是我们希望精益求精。

重构到策略模式

`getRecommended()`方法有很多问题。首先，方法太长——长到需要通过注释来解释各个部分。短方法更容易被理解，几乎不需要注释，完全优于长方法。此外，`getRecommended()`同时完成了选择策略和执行策略两件事情，这是两件不同的事情，属于独立的功能。可以使用策略模式来简化代码。为此，需要执行以下几个步骤：

- 创建一个接口来定义策略操作。
- 分别用不同的类实现该策略接口。
- 重构代码，选择使用正确的策略类。

假设你创建了一个 `Advisor` 接口，如图 23.2 所示。

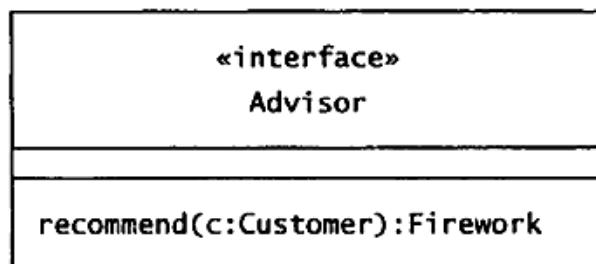


图 23.2 `Advisor` 接口定义了一个操作，多个类都可以使用不同的策略来实现它

`Advisor` 接口要求实现该接口的类接收一个客户对象，并返回一种推荐的焰火炮。在下一步重构 `Customer` 类的 `getRecommended()` 代码时，需要创建几个表示推荐策略的类。每个类都会分别实现 `Advisor` 接口的 `recommend()` 方法。

挑战 23.1

请填充图 23.3，该图展示了推荐逻辑被重构为一组策略类。

答案参见第 350 页

创建好策略类后，下一步就是将代码从 `Customer` 类的 `getRecommended()` 方法中移到这些新类中。`GroupAdvisor` 和 `ItemAdvisor` 是两个最简单的类。它们只是简单地包装了一下对

两个现有推荐引擎的调用。接口只能定义实例方法，因此，`GroupAdvisor` 和 `ItemAdvisor` 必须被实例化，以支持 `Advisor` 接口。由于，我们始终需要这样一个对象，因此可以让 `Customer` 类持有每个实现类唯一的静态实例。图 23.4 展示了该类的一个设计。

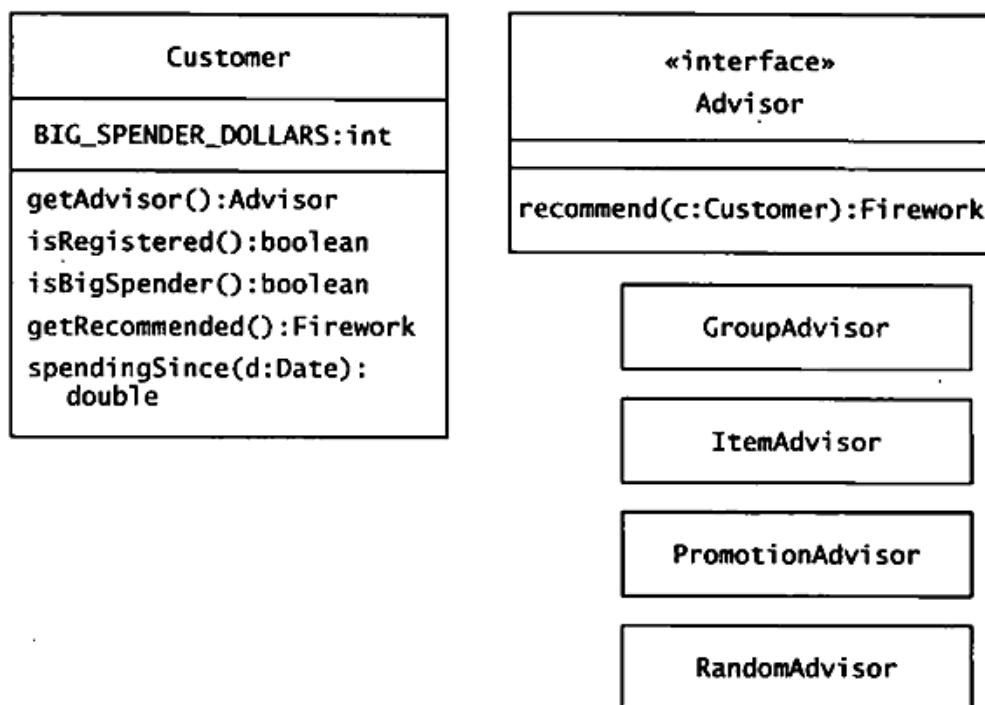


图 23.3 通过完成这个图可以展示重构推荐软件的过程。图中不同的策略都实现了同一个接口

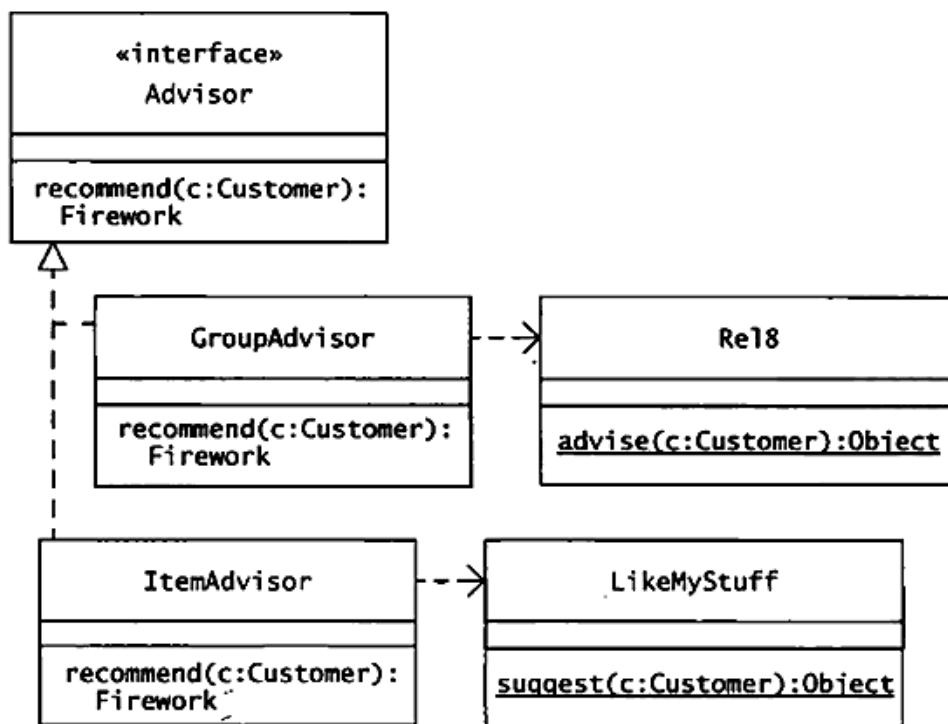


图 23.4 Advisor 接口的两个实现类提供了 recommend() 方法的不同策略，它们都依赖于现有的推荐引擎

这些 `advisor` 类将对 `recommend()` 方法的调用传递给了背后所需的引擎类。例如，`GroupAdvisor` 类将对 `recommend()` 的调用传递给 `Re18` 引擎需要的类。

```
public Firework recommend(Customer c) {  
    return (Firework) Re18.advise(c);  
}
```

挑战 23.2

除了策略模式外，`GroupAdvisor` 和 `ItemAdvisor` 类还使用了哪种模式？

答案参见第 350 页

`GroupAdvisor` 类和 `ItemAdvisor` 类的工作将对 `recommend()` 方法的调用传递给某个推荐引擎。我们还需要创建 `PromotionAdvisor` 类和 `RandomAdvisor` 类，这可以通过对 `Customer` 类的 `getRecommended()` 方法进行重构来完成。与 `GroupAdvisor` 类和 `ItemAdvisor` 类一样，这两个类都会实现 `recommend()` 操作。

`PromotionAdvisor` 类的构造函数应该判断是否有促销活动。你可能需要为它提供一个 `hasItem()` 方法，用以展示是否存在促销的产品：

```
public class PromotionAdvisor implements Advisor {  
    private Firework promoted;  
  
    public PromotionAdvisor() {  
        try {  
            Properties p = new Properties();  
            p.load(ClassLoader.getSystemResourceAsStream(  
                "config/strategy.dat"));  
            String promotedFireworkName = p.getProperty("promote");  
            if (promotedFireworkName != null)  
                promoted = Firework.lookup(promotedFireworkName);  
        } catch (Exception ignored) {  
            // 资源未找到或加载失败  
            promoted = null;  
        }  
    }  
}
```

```
public boolean hasItem() {
    return promoted != null;
}

public Firework recommend(Customer c) {
    return promoted;
}
}
```

RandomAdvisor 类很简单：

```
public class RandomAdvisor implements Advisor {
    public Firework recommend(Customer c) {
        return Firework.getRandom();
    }
}
```

通过运用策略模式对 Customer 类进行重构，将对策略的选择和执行分离开。Customer 对象的 advisor 属性保存了当前选择的策略。重构后的 Customer2 类延迟初始化了该类，它的逻辑反映了 Oozinoz 公司的广告策略：

```
private Advisor getAdvisor() {
    if (advisor == null) {
        if (promotionAdvisor.hasItem())
            advisor = promotionAdvisor;
        else if (isRegistered())
            advisor = groupAdvisor;
        else if (isBigSpender())
            advisor = itemAdvisor;
        else
            advisor = randomAdvisor;
    }
    return advisor;
}
```

挑战 23.3

请写出 Customer.getRecommended() 方法的最新代码。

答案参见第 351 页

比较策略模式与状态模式

重构后的代码几乎都是简单的类和方法。这就是重构的好处，并且，重构后的代码可以方便添加新的策略。这次重构的主要目的是将某个操作分散到一组相关的类中。就这点而言，策略模式和状态模式是一致的。事实上，一些开发人员甚至认为这两种模式并无区别。

一方面，状态模式和策略模式的模型区别微乎其微。显然，多态使得状态模式和策略模式的结构看起来几乎完全一致。

另一方面，在现实生活中，策略和状态是两种完全不同的思想。在对状态和策略进行建模时，这种现实的差异就会带来不同的问题。例如，在对状态进行建模时，状态的迁移至关重要；而对策略的建模却无须考虑这一问题。另一个区别是，策略模式允许客户选择或者提供某个策略，而状态模式却很少这样设计。

状态模式和策略模式的意图迥然不同，因此我们仍将认为它们是不同的模式。但是你必须意识到这并非所有人的共识。

比较策略模式和模板方法模式

第 21 章的模板方法模式，使用了排序作为例子来描述。可以使用 `sort()` 算法对任意的 `Arrays` 或者 `Collection` 类进行排序，只要你能够比较集合中的两个对象。但也可以声称：改变两个对象的比较算法，其实就是在改变策略。例如，在销售火箭时，按照价格排序和按照推动力排序就是两种不同的策略^{译注1}。

译注1：从结构上讲，策略模式的抽象策略通常定义为接口，而模板方法模式则定义为抽象类。虽然二者都可以看做是对算法或策略的抽象，但模板方法模式往往规定好整个策略的框架，子类实现的常常是策略的一部分；而策略模式则是对整个算法进行抽象。对于第 21 章给出的例子而言，如果针对排序，这自然是采用模板方法模式，但如果将比较看做是我们要抽象的算法，并将比较逻辑分离为单独的接口，则可以认为在模板方法模式中针对比较算法又运用了策略模式。

挑战 23.4

请问 `Arrays.sort()` 方法属于模板方法模式，还是策略模式？

答案参见第 351 页

小结

不同策略模型的逻辑可能会放到同一个类中，并常常写在一个方法里。这样的方法可能过于复杂，并且混合了策略的选择逻辑和执行逻辑^{译注2}。

为了简化这样的代码，需要创建一组类，每个类代表一种策略。定义一个接口方法，并让这些类都实现它。这就使得每个类都封装了一种策略，代码也会变得简单。同时，还需要为策略提供选择逻辑。该逻辑在重构后仍然可能比较复杂，但是可以简化这些代码，使其等价于问题域中描述策略的伪代码。

典型情况下，客户会使用一个上下文变量来维护对策略的选择。通过将策略操作的调用转发给上下文，并使用多态执行正确的策略，可以使得策略的执行变得更简单。策略模式将可相互替换的策略封装到不同的类，并让这些类实现一个公共接口，这样就可以帮助我们创建简洁的代码，为解决问题的各种方式建立统一的模型。

译注2：这事实上也违背了单一职责原则。

第24章

命令（Command）模式

直接调用是执行方法的一般方式。然而，有时我们无法控制方法执行的时机与上下文。这种情况下，可以将方法封装在对象的内部。通过在对象内部存储调用方法所需要的信息，就可以让客户端或者服务决定何时调用该方法。

命令模式的意图是将请求封装在对象内部。

经典范例：菜单命令

支持菜单的工具包通常都使用了命令模式。每个菜单项关联一个对象，以便在用户单击该菜单项时，可以执行相关的行为。这种设计分离了 GUI 逻辑与应用逻辑。Swing 库就采用了这种模式，允许你为每个 `JMenuItem` 关联一个 `ActionListener`。

当用户单击菜单时，该如何让类调用相关的方法呢？答案是使用多态：固定操作的名称，针对不同的类给出同名方法的不同实现。对 `JMenuItem` 而言，方法名是 `actionPerformed()`，当用户选择任意一个菜单项时，`JMenuItem` 就去调用注册到侦听器上任意对象的 `actionPerformed()` 方法。

挑战 24.1

Java 的菜单机制使你可以方便地使用命令模式，而并不要求你将自己的代码组织为命令模式。事实上，在开发过程中，用一个对象监听 GUI 中的所有事件是很常见的。请问这用的是哪种模式？

答案参见第 352 页

当在开发 Swing 应用时，可能会为所有的 GUI 事件注册同一个监听器对象，尤其是在 GUI 组件交互时。然而，对于菜单而言，这并非好的选择。倘若让一个单独的对象侦听所有的菜单事件，当 GUI 对象发出事件时，就需要对这些事件进行筛选。相反，如果让每个菜单项都用独立的对象来处理事件，那么最好使用命令模式。

当用户选择菜单项时，会调用 `actionPerformed()` 方法。在创建菜单项时，可以使用指定了命令行为的 `actionPerformed()` 方法，为其绑定一个 `ActionListener`。相对于定义一个新类来实现这个小功能，使用匿名类是更好的选择。

考虑 `com.oozinoz.visualization` 包中的 `visualization2` 类。该类提供了一个菜单栏，包含了文件菜单，使得用户可以保存或者恢复模拟 Oozinoz 工厂的效果。该菜单包含两个菜单项，分别是 `Save As...` 和 `Restore From....`。它们都注册了监听器对用户事件进行侦听。这两个监听器分别调用 `visualization2` 类的 `save()` 和 `load()` 方法，来实现 `actionPerformed()` 方法：

```
package com.oozinoz.visualization;
import java.awt.event.*;
import javax.swing.*;
import com.oozinoz.ui.*;
public class Visualization2 extends Visualization {
    public static void main(String[] args) {
        Visualization2 panel = new Visualization2(UI.NORMAL);
        JFrame frame =
            SwingFacade.launch(panel, "Operational Model");
        frame.setJMenuBar(panel.menus());
        frame.setVisible(true);
    }
    public Visualization2(UI ui) {
        super(ui);
```

```
}

public JMenuBar menus() {
    JMenuBar menuBar = new JMenuBar();

    JMenu menu = new JMenu("File");
    menuBar.add(menu);

    JMenuItem menuItem = new JMenuItem("Save As...");
    menuItem.addActionListener(new ActionListener() {
        // 挑战!
    });
    menu.add(menuItem);

    menuItem = new JMenuItem("Restore From...");
    menuItem.addActionListener(new ActionListener() {
        // 挑战!
    });
    menu.add(menuItem);

    return menuBar;
}

public void save() { /* 忽略 */ }
public void restore() { /* 忽略 */ }
}
```

挑战 24.2

请写出匿名类 `ActionListener` 的 `actionPerformed()` 方法。注意，该方法需要一个 `ActionEvent` 参数。

答案参见第 352 页

当使用命令模式来装配菜单时，应将这些命令应用到其他开发者提供的上下文，即 Java 菜单框架中。在某些命令模式的应用场合，需要自己提供命令执行的上下文。例如，你可能需要提供一个时间服务，它可以记录方法执行的时间。

使用命令模式来提供服务

假设你想让开发者统计某个方法执行的时间。现在，我们有一个 `Command` 接口，实现如下：

```
public abstract void execute();
```

然后，定义一个 `CommandTimer` 类：

```
package com.oozinoz.utility;

import com.oozinoz.robotInterpreter.Command;

public class CommandTimer {
    public static long time(Command command) {
        long t1 = System.currentTimeMillis();
        command.execute();
        long t2 = System.currentTimeMillis();
        return t2 - t1;
    }
}
```

可以使用 JUnit 测试 `time()` 方法，如下所示。注意，这并非一个精确的测试，如果计时器是“jittery”，测试就会失败。

```
package app.command;

import com.oozinoz.robotInterpreter.Command;
import com.oozinoz.utility.CommandTimer;

import junit.framework.TestCase;
public class TestCommandTimer extends TestCase {
    public void testSleep() {
        Command doze = new Command() {
            public void execute() {
                try {
                    Thread.sleep(
                        2000 + Math.round(10 * Math.random()));
                } catch (InterruptedException ignored) {

```

```
        }
    }

};

long actual = // 挑战！

long expected = 2000;
long delta = 5;
assertTrue(
    "Should be " + expected + " +/- " + delta + " ms",
    expected - delta <= actual
    && actual <= expected + delta);
}
```

挑战 24.3

请写出上面用于统计 `doze` 命令执行时间的赋值语句。

答案参见第 354 页

命令钩子

第 21 章的模板方法模式中介绍了 Aster 火药填压机：一个应用模板方法的智能机器。如果机器关闭时，当前模具还未处理完，则填压机中的代码会自动将当前模具设置为未完成。

`AsterStarPress` 类是一个抽象类，需要定义一个子类来重写其 `markMoldIncomplete()` 方法。其中，`shutDown()` 方法依赖于该方法，用于确保当机器关闭时，让领域对象知道哪些模具未完成。

```
public void shutdown() {
    if (inProcess()) {
        stopProcessing();
        markMoldIncomplete(currentMoldID);
    }
}
```

```

    usherInputMolds();
    dischargePaste();
    flush();
}

```

当需要将某些类移植到火药填压机的板载计算机上时，创建 `AsterStarPress` 类的子类会显得很不方便。假定你要求 Aster 公司的程序员使用命令模式来重新提供一个钩子。图 24.1 展示了 `AsterStarPress` 类可以使用的 Hook 命令，使你可以在运行时对火药填压机的代码进行参数化。

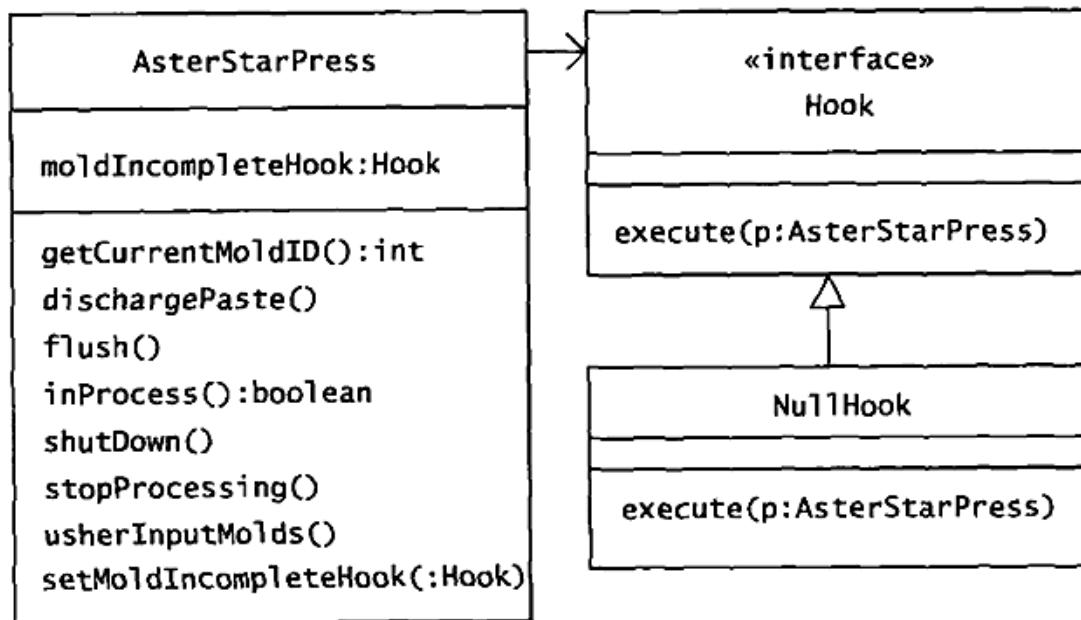


图 24.1 该类可以提供一个钩子——一种插入自定义代码的方式，通过程序执行时在一个指定点来调用支持的命令

在原始的 `AsterStarPress` 类中，`shutDown()` 方法依赖其子类提供的某个步骤。在新的设计中，`shutDown()` 方法在将停未停之时，会使用钩子来执行客户代码。

```

public void shutDown() {
    if (inProcess()) {
        stopProcessing();
        // 挑战!
    }
    usherInputMolds();
    dischargePaste();
    flush();
}

```

挑战 24.4

请写出新 `shutdown()` 方法的实现。

答案参见第 355 页

这个例子展示了另一种模式——Null Object 模式^{#注1}，该模式没有 *Design Patterns* 一书中提到的模式那么知名。它通过引入一个默认的对象来避免空指针的检查。请参考 *Refactoring*（由 Fowler 等人在 1999 年编写）一书，书中介绍了如何在代码中引入该模式。

命令模式提供了除模板方法模式外的另一种设计钩子的方法，它在意图和结构上也和其他几种模式相似。

命令模式与其他模式的关系

命令模式类似于解释器模式。下一章我们会比较这两个模式。命令模式还和另一个模式很相似，两者都是知道创建对象的时间，但都不知道应该创建哪个对象。

挑战 24.5

哪个模式描述了这种情况：知道何时去创建对象，但却不知道需要实例化哪个类？

答案参见第 356 页

命令模式除了与某些模式具有异曲同工之妙，它也常常会与其他模式协作。例如，在 MVC 设计中可以将命令模式与调停者模式混合使用。第 19 章的备忘录模式就给出了这个例子。`visualization` 类处理 GUI 控制逻辑，任何与模型相关的逻辑都由调停者来处理。例如，`visualization` 类用如下代码来延迟初始化 Undo 按钮：

译注1：即空对象模式，由 Bobby Woolf 提出。该模式用于处理判断对象为 null 的情况。它为指定对象提供了一个特定对象，用以替换指定对象为 null 的情况。引入 Null Object 模式，可以消除判断对象是否为 null 的语句。

```
protected JButton undoButton() {  
    if (undoButton == null) {  
        undoButton = ui.createButtonCancel();  
        undoButton.setText("Undo");  
        undoButton.setEnabled(false);  
        undoButton.addActionListener(mediator.undoAction());  
    }  
    return undoButton;  
}
```

上面的代码使用了命令模式，将 `undo()` 方法封装在 `ActionListener` 类的实例中。该代码也使用了调停者模式，让中心对象去协调与底层模型相关的事件。为了让 `undo()` 方法生效，调停者对象必须恢复模拟工厂的前一个版本，这正是将命令模式与其他模式混合使用的最好时机。

挑战 24.6

哪个模式可以实现对象状态的存储与恢复？

答案参见第 356 页

小结

命令模式可以将请求封装在一个对象中，允许你可以像管理对象一样去管理方法，传递并且在合适的时机去调用它们。菜单是应用命令模式的一个经典案例。菜单项知道何时执行一个请求，但却不知道应该去执行什么请求。命令模式可以让你使用与菜单标签相关的方法调用作为参数，传递给一个菜单。

命令模式的另一个用处是，允许在服务执行的上下文中执行客户类代码。服务经常在调用客户代码的前后执行。最后，除了可以控制方法的执行时间和上下文外，命令模式还可以提供一个钩子机制，允许客户代码作为算法执行的一部分。

命令模式将请求封装在了对象中，因此你可以像操作对象那样来操作请求。或许是因为这种想法如此普遍，通常命令模式会和其他模式一起使用。例如，命令模式可以作为模板方法模式的一种替代模式，它也经常和调停者模式与备忘录模式配合使用。

第 25 章

解释器 (Interpreter) 模式

和命令模式一样，解释器模式也产生一个可执行的对象。不同的是，解释器模式创建了一个类层次，该层次中的每个类都实现或者解释了一个公共操作，并且该操作的名称与类名相同。从这点来看，解释器模式与状态模式和策略模式都很相似。在这些模式各自的类集合中，都有一个公共的方法，但是每个类对该方法的实现都不一样。

解释器模式和合成模式也很相似，合成模式通常会定义一个公共接口，该接口用于单个对象或者复合对象。合成模式并不要求必须以不同的方式组织结构，尽管该模式本身支持这种做法。例如第 5 章中的 `ProcessComponent` 层级，就允许顺序和可交替的两种工作流程。合成模式的要点是组合不同的类型（解释器模式通常是基于合成模式的）。一个类组合其他组件的方式定义了解释器类的实现方式。

要了解解释器模式，极具挑战。你可能需要重温合成模式，因为在本章，我们会经常用到该模式。

解释器模式的意图是让你根据事先定义好的一系列组合规则，组合可执行对象。

一个解释器示例

Oozinoz 公司使用解释器来控制机器人沿着生产线移动材料。你或许会认为解释器是一门

编程语言，其实它的核心是提供一组类，可以组合指令。Oozinoz 公司的机器人解释器就是一组封装了机器人指令集的类层次。该层次的起点是一个抽象的 `Command` 类。分布在层次中的是一个 `execute()` 方法。图 25.1 展示了 `Robot` 类以及机器人解释器支持的两个命令。

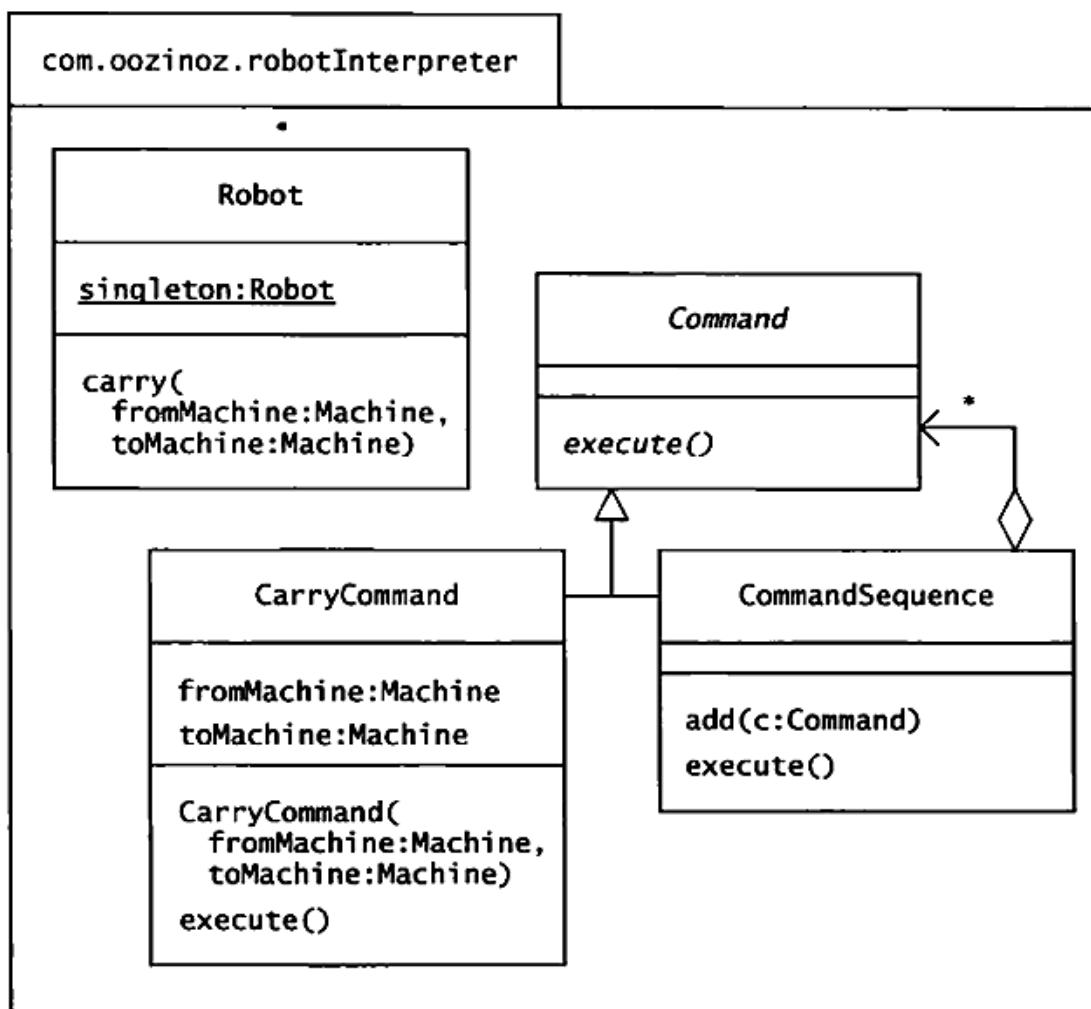


图 25.1 该解释器层次提供了对工厂机器人运行时编程的能力

粗略一看，你会以为图 25.1 是在介绍命令模式，因为层次结构的顶端定义了一个 `Command` 类。然而，命令模式的意图是将方法封装在对象的内部，而图 25.1 所示的 `Command` 层次并非如此，设计层次的目的是要求 `Command` 子类重新阐释 `execute()` 操作的含义。这正是解释器模式的意图：允许你组合可执行的对象。

典型的解释器层次包含的子类通常超过两个，我们马上就会扩展 `Command` 层次。图 25.1 所示的两个类作为第一个例子来讲已经足够了：

```

package app.interpreter;
import com.oozinoz.machine.*;
import com.oozinoz.robotInterpreter.*;

```

```
public class ShowInterpreter {
    public static void main(String[] args) {
        MachineComposite dublin = OozinozFactory.dublin();
        ShellAssembler assembler =
            (ShellAssembler) dublin.find("ShellAssembler:3302");
        StarPress press = (StarPress) dublin.find("StarPress:3404");
        Fuser fuser = (Fuser) dublin.find("Fuser:3102");

        assembler.load(new Bin(11011));
        press.load(new Bin(11015));

        CarryCommand carry1 = new CarryCommand(assembler, fuser);
        CarryCommand carry2 = new CarryCommand(press, fuser);

        CommandSequenceseq = new CommandSequence();
        seq.add(carry1);
        seq.add(carry2);

        seq.execute();
    }
}
```

该示意代码让工厂机器人将两箱原材料从操作机上转移到卸载缓冲区里，需要用到 `OozinozFactory` 类的 `dublin()` 方法所返回的机器组合对象。该数据模型代表 Oozinoz 公司在爱尔兰都柏林的一个新的工厂设备。代码定位了工厂里的三台机器，将材料箱放在其中两台机器上，然后创建 `Command` 层次中的命令。该程序的最后一句调用了 `CommandSequence` 对象的 `execute()` 方法，让机器人按照 `seq` 中的指令执行操作。

`CommandSequence` 对象通过依次调用各个子命令来完成对 `execute()` 操作的解释。

```
package com.oozinoz.robotInterpreter;

import java.util.ArrayList;
import java.util.List;

public class CommandSequence extends Command {
    protected List commands = new ArrayList();
```

```
public void add(Command c) {  
    commands.add(c);  
}  
  
public void execute() {  
    for (int i = 0; i < commands.size(); i++) {  
        Command c = (Command) commands.get(i);  
        c.execute();  
    }  
}
```

`CarryCommand` 类的 `execute()` 方法实现如下：与工厂机器人交互，将箱子从一个机器人移到另一个机器人。

```
package com.oozinoz.robotInterpreter;  
import com.oozinoz.machine.Machine;  
  
public class CarryCommand extends Command {  
    protected Machine fromMachine;  
    protected Machine toMachine;  
  
    public CarryCommand(  
        Machine fromMachine, Machine toMachine) {  
        this.fromMachine = fromMachine;  
        this.toMachine = toMachine;  
    }  
  
    public void execute() {  
        Robot.singleton.carry(fromMachine, toMachine);  
    }  
}
```

`CarryCommand` 类是为机器人控制生产线领域提供的专用设计。我们很容易想到与另一个领域相关的类，比如 `StartUpCommand` 类或者 `ShutdownCommand` 类，用来控制机器。创建能操作一组机器的 `ForCommand` 类也很有用。图 25.2 展示了 `Command` 类层次的这些扩展。

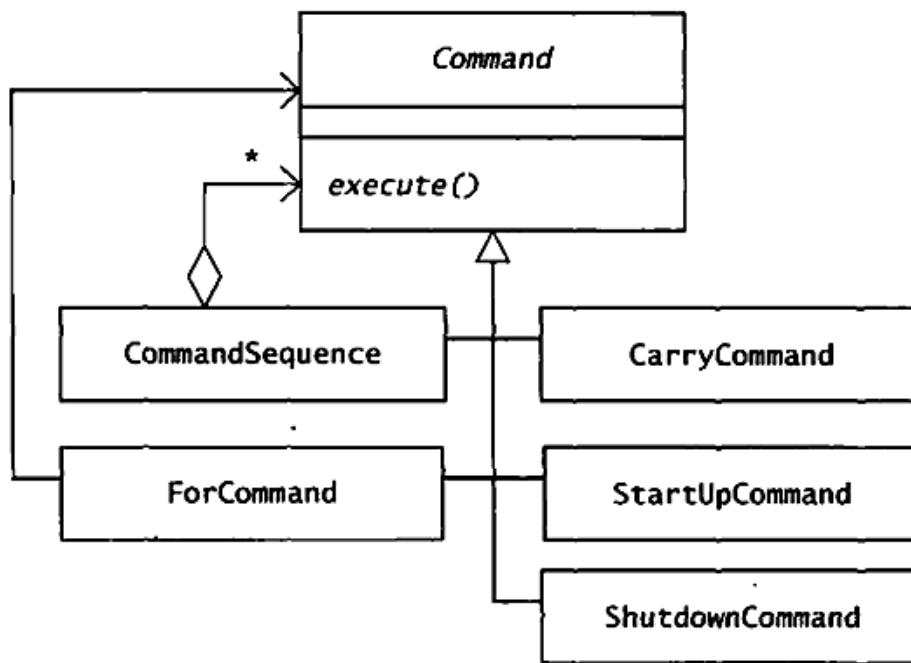


图 25.2 解释器模式允许多个子类重新解释通用操作的意义

ForCommand 类的一部分设计清晰明了。该类的构造函数假设接收一组机器和一个 **Command** 对象，以便执行 **for** 循环。该设计更加复杂的部分是如何将循环和方法体关联起来。Java 5 扩展了 **for** 语句，建立了一个变量，每次循环的时候都会给该变量附上新值。我们用下面的语句模拟该方法。

```
for (Command c: commands)
    c.execute();
```

Java 将 **c** 标识符与循环体中的 **c** 变量联系起来。为了创建一个解释器类来模拟该效果，我们需要一个方法对变量进行处理和运算。图 25.3 的 **Term** 层次做到了这点。

Term 层次与 **Command** 层次的相似之处在于，都有一个公共方法 **eval()** 贯穿整个层次。你可能会说该层次本身就是解释器模式自身的一个例子，尽管它不包含合成类。像 **CommandSequence** 这样的合成类常常会出现在解释器模式中。

Term 层次允许将单台机器都命名为常量，并允许我们将变量赋值给这些常量或者其他变量。该层次允许我们构建出更为灵活的与领域相关的解释器类。例如，**StartUpCommand** 代码可以和 **Term** 对象一起工作，而不仅仅是特定的机器：

```
package com.oozinoz.robotInterpreter2;
import com.oozinoz.machine.Machine;
```

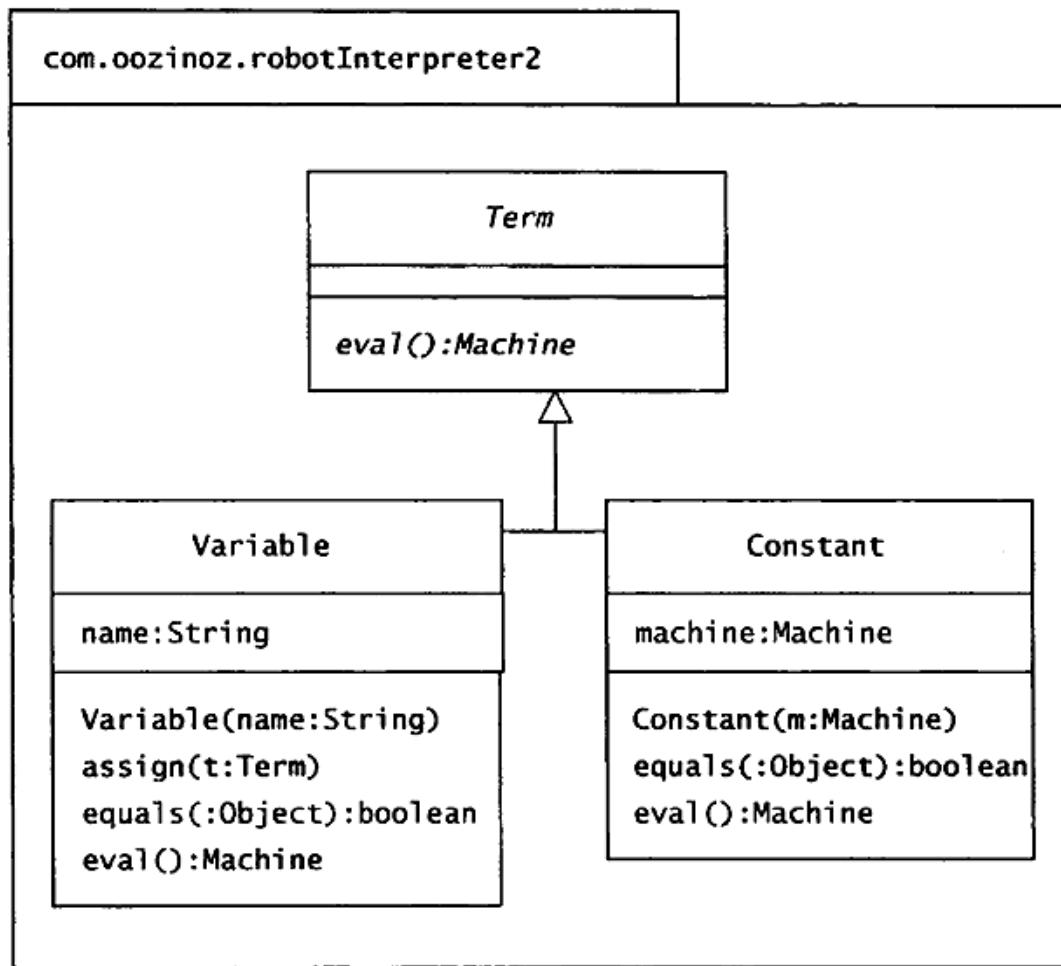


图 25.3 Term 类层次允许变量对机器进行处理

```

public class StartUpCommand extends Command {
    protected Term term;

    public StartUpCommand(Term term) {
        this.term = term;
    }

    public void execute() {
        Machine m = term.eval();
        m.startup();
    }
}

```

类似的，可以通过修改 CarryCommand 类，使其变得更加灵活：

```
package com.oozinoz.robotInterpreter2;
```

```
public class CarryCommand extends Command {  
    protected Term from;  
    protected Term to;  
  
    public CarryCommand(Term fromTerm, Term toTerm) {  
        from = fromTerm;  
        to = toTerm;  
    }  
  
    public void execute() {  
        Robot.singleton.carry(from.eval(), to.eval());  
    }  
}
```

一旦将 Command 层次设计成可以操作 term 对象，就可以写一个 ForCommand 类，使之允许设置变量的值，并能够在循环中执行方法体命令：

```
package com.oozinoz.robotInterpreter2;  
  
import java.util.List;  
import com.oozinoz.machine.Machine;  
import com.oozinoz.machine.MachineComponent;  
import com.oozinoz.machine.MachineComposite;  
  
public class ForCommand extends Command {  
    protected MachineComponent root;  
    protected Variable variable;  
    protected Command body;  
  
    public ForCommand(  
        MachineComponent mc, Variable v, Command body) {  
        this.root = mc;  
        this.variable = v;  
        this.body = body;  
    }  
  
    public void execute() {  
        execute(root);  
    }  
}
```

```
private void execute(MachineComponent mc) {  
    if (mc instanceof Machine) {  
        // 挑战!  
        return;  
    }  
  
    MachineComposite comp = (MachineComposite) mc;  
    List children = comp.getComponents();  
    for (int i = 0; i < children.size(); i++) {  
        MachineComponent child =  
            (MachineComponent) children.get(i);  
        execute(child);  
    }  
}
```

ForCommand 类的 execute() 方法遍历了机器的整棵组件树。第 28 章的迭代器模式将提供另一种更为优雅的方式来迭代组合结构。对解释器模式而言，要点是正确地解释树中每个节点的 execute() 请求。

挑战 25.1

完成 ForCommand 类中 execute() 方法的代码。

答案参见第 356 页

ForCommand 类允许我们为工厂编写“程序”或“脚本”命令。例如，如下的程序就编写了一个解释器，用来关闭工厂的所有机器：

```
package app.interpreter;  
  
import com.oozinoz.machine.*;  
import com.oozinoz.robotInterpreter2.*;  
  
class ShowDown {  
    public static void main(String[] args) {  
        MachineComposite dublin = oozinozFactory.dublin();  
        Variable v = new Variable("machine");  
        Command c = new ForCommand(  
    }  
}
```

```

        dublin, v, new ShutDownCommand(v));
c.execute();
}
}

```

当程序调用 `execute()` 方法时, `ForCommand` 对象 `c` 会通过遍历所提供的机器组件, 为每个机器解释 `execute()` 方法:

- 为变量 `v` 赋值。
- 调用已经提供的 `ShutDownCommand` 对象的 `execute()` 方法。

如果再添加一些控制逻辑流程的类, 比如 `IfCommand` 类和 `whileCommand` 类, 就能构建一个完善的解释器了。这些类需要某些方法来建模 Boolean 条件。例如, 可能需要对机器变量是否等于某台特殊机器进行建模。我们可以为此引入一套新的术语层次结构, 但也可以借用 C 语言的想法来简化设计: 让 `null` 代表假, 其余值都代表真。有了这个想法, 就可以扩展 `Term` 类的层次结构, 如图 25.4 所示。

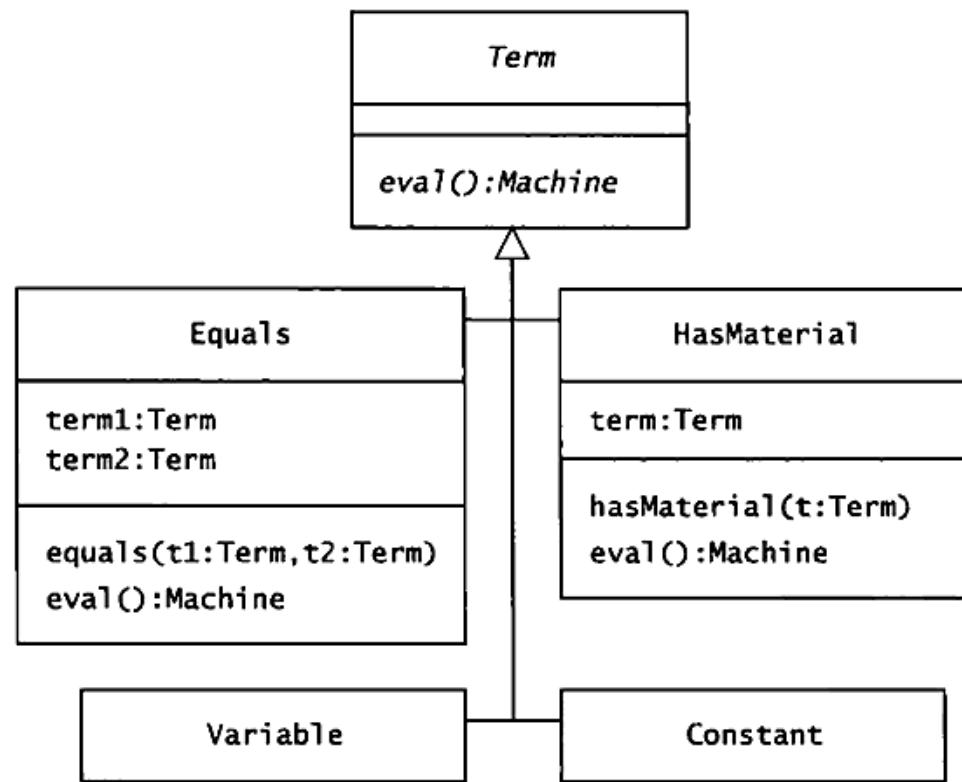


图 25.4 包括建模 Boolean 条件的 `Term` 类层次结构

`Equals` 类比较两个 `term` 对象, 并且返回 `null` 表示假。一个合理的设计是让 `Equals` 类包含 `eval()` 方法, 并且在两个 `term` 对象结果相等时, 返回其中一个结果:

```
package com.oozinoz.robotInterpreter2;
import com.oozinoz.machine.Machine;
public class Equals extends Term {
    protected Term term1;
    protected Term term2;
    public Equals(Term term1, Term term2) {
        this.term1 = term1;
        this.term2 = term2;
    }

    public Machine eval() {
        Machine m1 = term1.eval();
        Machine m2 = term2.eval();
        return m1.equals(m2) ? m1 : null;
    }
}
```

HasMaterial 类将 Boolean 类中值的概念扩展到了特定领域，代码如下所示：

```
package com.oozinoz.robotInterpreter2;

import com.oozinoz.machine.Machine;

public class HasMaterial extends Term {
    protected Term term;

    public HasMaterial(Term term) {
        this.term = term;
    }

    public Machine eval() {
        Machine m = term.eval();
        return m.hasMaterial() ? m : null;
    }
}
```

既然已经将 Boolean 条件术语引入到解释器包中，我们就可以添加流程控制类，如图 25.5 所示。

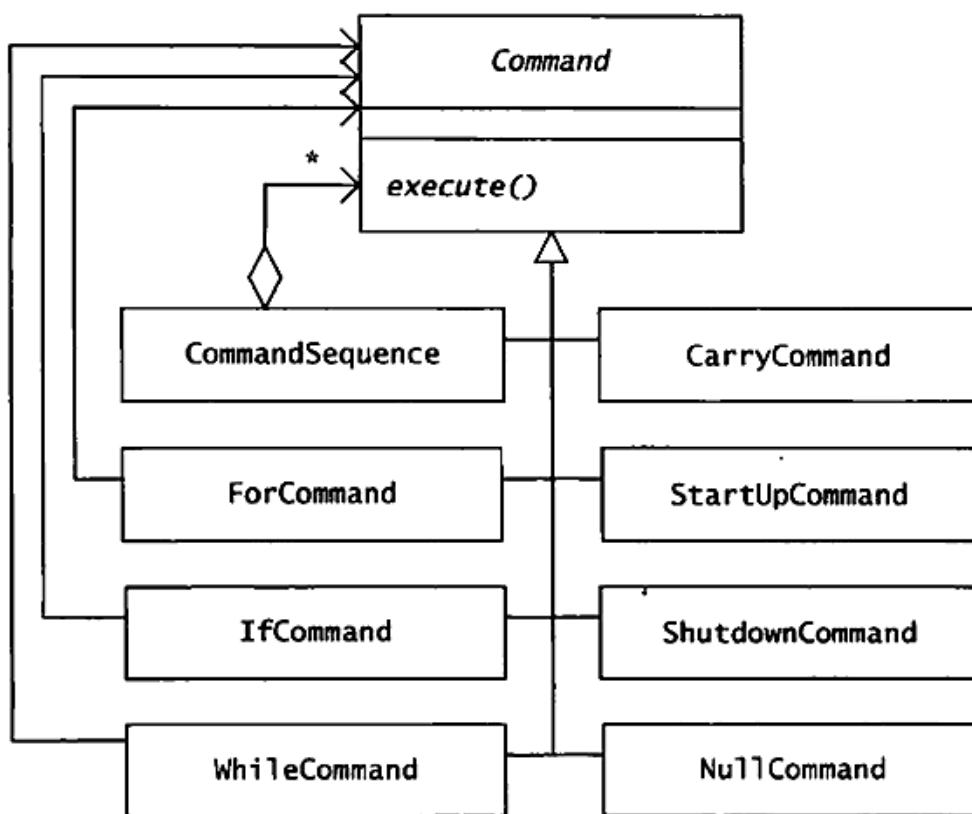


图 25.5 可以通过为解释器层次结构增加一个逻辑控制流类，使其更加完善

当需要一个什么也不执行的命令时，NullCommand 将很有用。例如，if 命令的 else 分支为空时：

```
package com.oozinoz.robotInterpreter2;
public class NullCommand extends Command {
    public void execute() {
    }
}
package com.oozinoz.robotInterpreter2;

public class IfCommand extends Command {
    protected Term term;
    protected Command body;
    protected Command elseBody;

    public IfCommand(
        Term term, Command body, Command elseBody) {
        this.term = term;
        this.body = body;
        this.elseBody = elseBody;
    }
}
```

```
public void execute() {  
    // 挑战!  
}  
}
```

挑战 25.2

完成 IfCommand 类的 execute()方法。

答案参见第 357 页

挑战 25.3

写出 WhileCommand 类的代码。

答案参见第 357 页

你可能会将 WhileCommand 类与卸载火药填压机的解释器一起使用：

```
package app.interpreter;  
  
import com.oozinoz.machine.*;  
import com.oozinoz.robotInterpreter2.*;  
  
public class Showwhile {  
    public static void main(String[] args) {  
        MachineComposite dublin = OozinozFactory.dublin();  
        Term starPress = new Constant(  
            (Machine) dublin.find("StarPress:1401"));  
        Term fuser = new Constant(  
            (Machine) dublin.find("Fuser:1101"));  
  
        starPress.eval().load(new Bin(77));  
        starPress.eval().load(new Bin(88));  
  
        whileCommand command = new WhileCommand(  
            new HasMaterial(starPress),  
            new CarryCommand(starPress, fuser));
```

```
    command.execute();
}
}

command 对象是一个解释器，它的 execute()方法会将所有的箱子从火药填压机 1401 上卸载下来。
```

挑战 25.4

合上本书，请写出命令模式和解释器模式的区别。

答案参见第 357 页

可以为解释器类层次增加更多的类，以此增加对更多领域相关任务类型控制的支持。也可以扩展 Term 层次结构。譬如增加一个 Term 的子类，使其可以找到其他机器附近的卸载缓冲区。

Command 和 Term 类层次的用户可以任意组合丰富而复杂的执行“程序”。例如，可以轻而易举地创建这样一个对象：在执行时，可以卸载工厂中除卸载缓冲区外的其他机器上的原材料。可以用如下伪代码来描述该程序：

```
for (m in factory)
    if (not (m is unloadBuffer))
        ub = findUnload for m
        while (m hasMaterial)
            carry (m, ub)
```

如果用 Java 代码来实现这些功能，会比这些伪代码要冗余复杂得多。那么，为何不创建一个领域相关的解析器，负责将这些伪代码转换成实际的代码，使其可以管理工厂的原材料，并为我们创建相应的解释器对象呢？

解释器、语言和解析器

解释器模式仅仅展现了解释器是如何工作的，却并没有指出应该如何实例化以及如何组合这些对象。在本章，我们通过几行 Java 代码“手工”地创建了一个新的解释器；然而，更为常见的方式却是通过解析器 (Parser) 来创建。解析器对象可以根据一组规则识别和分解文本结构，

使其转换为更加适宜的形式，以便进一步进行处理。例如，可以为早期的伪代码文本程序创建一个机器命令的解释器对象。

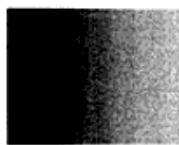
在撰写本书时，市面上关于用 Java 写解析器的资料很少。想要获取更多的资料，可以尝试在网上搜索“Java 解析器工具（Java parser tools）”。大多数的解析器工具都包含一个解析器的生成器。要使用该生成器，需要使用指定的语法或语言来描述该模式，工具会根据你的描述生成一个解析器。生成的解析器会识别你语言中的实例。相对于使用解析器生成器，还可以运用解析器模式编写通用目的的解析器。*Building Parsers with Java™*（由 Metsker 在 2001 年编写）一书使用 Java 语言解释了该技术。

小结

解释器模式可以让你根据创建的类层次结构来组合可执行对象。层次结构中的每个类都实现了一个公共的操作，比如 `execute()`。尽管之前的例子没有讨论，但该方法通常都需要一个额外的“上下文”对象，用来存储重要的状态信息。

每个类的名称通常暗指该类所要实现或解释的通用操作。每个类要么定义合成命令的方式，要么表示执行某些操作的终端命令。

设计解释器常常会引入变量，以及布尔或者算术表达式。解释器也经常和解析器一起使用，用来创建一个轻量级的语言，简化新的解释器对象的创建。



第 5 部分

扩展型模式

第 26 章

扩展型模式介绍

在使用 Java 编程时，你并不是从零开始，而是“继承”了 Java 类库的所有功能。通常，还可以使用同事和前人写好的代码。只要你不是在重新组织或改进遗留代码，你就是在对它进行扩展。可以认为：Java 编程本身就是一种扩展。

如果你曾经继承过一个代码库，或许你会抱怨糟糕的代码质量。然而，你添加的新代码质量就一定好吗？答案通常是主观的。不过，本章介绍的面向对象软件开发的几个原则，却可以用来评估你的工作。

除了扩展代码库的一些常见技术外，还可以使用设计模式添加新的行为。在了解了基本的面向对象的设计原则后，本章会回顾前面提到的具有扩展功能的模式，然后介绍其他面向扩展的模式。

面向对象设计的原则

石桥的存在已有悠悠数千年的历史，为了设计它们，我们拥有太多久经考验并得到一致认可的原则。面向对象编程的诞生不过才 50 年左右，因此它的设计无法达到石桥的设计水平，也就不足为奇了。然而，我们已经拥有了许多讨论设计原则的优秀论坛。其中，www.c2.com 的 Portland Pattern Repository 就是其中之一。访问该网站，你会发现在评估面向对象的设计方面，一些原则业已体现了自身价值。在设计时，需要考虑的其中一条原则是 Liskov 替换原则（LSP）。

Liskov 替换原则

子类应该在逻辑性上与其超类保持一致，但是什么才是逻辑性（logical）与一致性（consistent）呢？Java 编译器将会保证一定级别的一致性，但是，任何一个一致性的原则又都会绕过编译器。Liskov 替换原则（由 Liskov 在 1987 年提出）能够帮助我们改善设计，它的主要思想是：一个类的实例应该具有其父类的所有功能。

诸如 Java 这样的面向对象设计语言，已经满足基本的 LSP。例如，如果 `UnloadBuffer` 类是 `Machine` 类的子类，就可以将 `UnloadBuffer` 对象当做 `Machine` 对象来使用：

```
Machine m = new UnloadBuffer(3501);
```

LSP 的某些方面还需要人的参与，因为今天的编译器显得还不够智能。考虑图 26.1 所示的类层次关系。

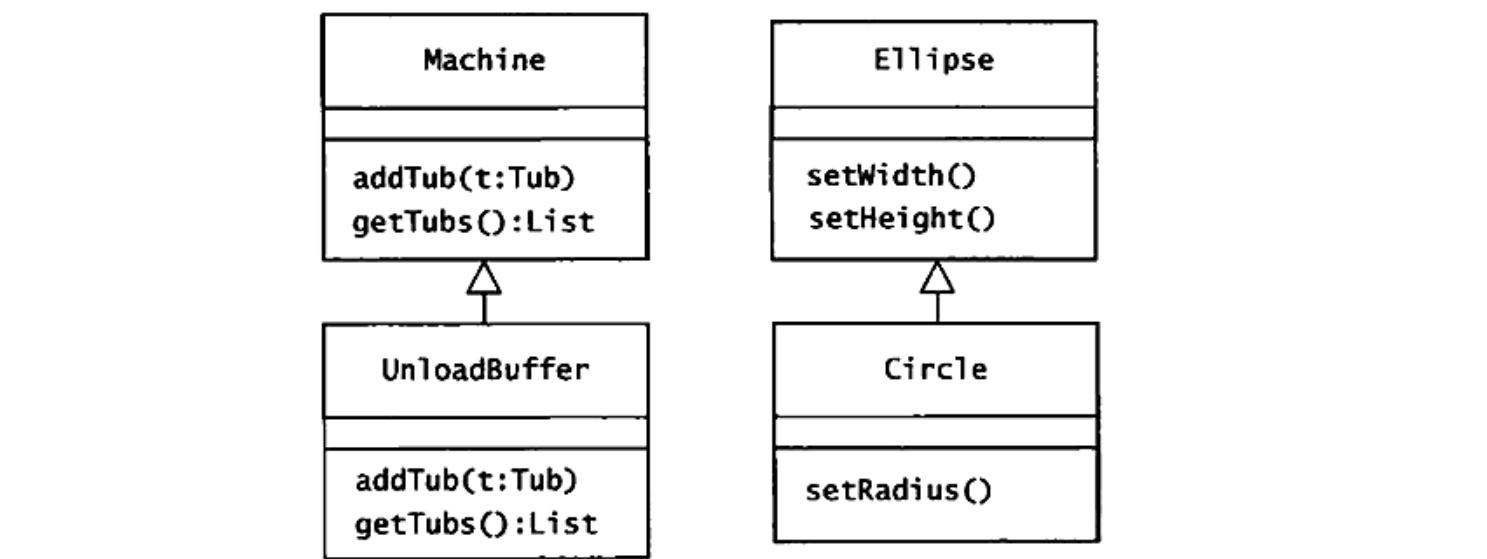


图 26.1 这幅图有两个问题：卸载缓冲池是机器吗？圆是一个椭圆吗？

卸载缓冲池当然是一个机器，但是在类层次中这样建模就会有问题。对于 Oozinoz 公司的机器来说，除了卸载缓冲区，其他的机器都可以接收一桶原料。几乎每台机器都可以接收原料桶，并将其拥有的原料桶集合，报告给附近的机器。将此行为迁移到 `Machine` 类中是很有价值的。但是，如果调用 `UnloadBuffer` 对象的 `addTub()` 或者 `getTubs()` 方法，就会出现错误。倘若出现了错误，我们在调用方法时是否应该抛出异常呢？

假设其他的开发者编写了一个方法，用来查询车间里所有机器拥有的原料桶的集合。一旦访问到卸载缓冲池，如果 `UnloadBuffer` 类的 `getTubs()` 方法抛出异常，这段代码就会出现问题。这严重违反了 LSP：当你将 `UnloadBuffer` 对象当做 `Machine` 对象来使用时，程序可能会崩溃！假设不抛出异常，我们需要简单地忽略对 `UnloadBuffer` 类中 `getTubs()` 和 `addTub()` 的调用。这一做法依然违反了 LSP，因为在你给机器添加一个原料桶时，这个原料桶可能会消失！

违反 LSP 并不一定是设计缺陷。针对 Oozinoz 公司的这种情况，需要权衡一下，让 `Machine` 类拥有大多数机器的行为和违反 LSP 原则，究竟哪个价值更大。重要的一点是意识到 LSP，并且清楚为什么其他设计可能违反了 LSP。

挑战 26.1

圆当然是椭圆的一种特殊形态，不是吗？请说明图 26.1 中 `Ellipse` 类和 `Circle` 类的关系是否违背了 LSP。

答案参见第 358 页

迪米特法则

在 20 世纪 80 年代后期，美国东北大学 Demeter Project 的成员尝试编辑了一些规则，用于确保健康的面向对象编程。项目团队将这些规则称为迪米特法则（Law of Demeter, LoD）。Karl Lieberherr 和 Ian Holland 在 *Assuring Good Style for Object-Oriented Programs* 一文中全面地总结了这一规则。声明认为：非正式地说，迪米特法则要求每个方法只能给有限的对象发送消息，包括参数对象、`[this]` 伪变量，以及 `[this]` 的直接子部分。文章随后给出了该法则的正式定义。相对于完全理解迪米特法则的意图，识别设计是否违反该法则更加容易。

假设你有一个 `MaterialManager` 对象，该对象有一个方法接收一个 `Tub` 对象作为参数。`Tub` 对象有一个 `Location` 属性，该属性返回一个 `Machine` 对象，用以表示桶被放在哪个位置。假设在 `MaterialManager` 的方法中，想要知道机器是否可用，你可能会写如下代码：

```
if (tub.getLocation().isUp()) {  
    //...  
}
```

这段代码违反了迪米特法则，因为它调用了一个方法，将消息发送给 `tub.getLocation()`。而 `tub.getLocation()` 对象不是一个参数，不是 `this`——`MaterialManager` 对象正在执行的方法，也不是 `this` 的一个属性。

挑战 26.2

请解释为何 `tub.getLocation().isup()` 表达式不合适？

答案参见第 358 页

如果这个挑战仅仅让你觉得形如 `a.b.c` 的表达式是错误的，那就降低了迪米特法则的价值。事实上，Lieberherr 和 Holland 希望迪米特法则能够进一步确定无误地回答这样的问题：是否可以遵循某种公式或法则来写出更好的面向对象程序？这篇阐释迪米特法则的早期论文非常值得我们拜读。就像 Liskov 替换原则一样，如果知道并遵循这些规则，它就会帮助你写出更好的代码，一旦你的设计违背了这些原则，就能够及时获知。

你会发现，遵循这些指导原则，扩展性就能够自然而然产生好的代码。但是，对于很多开发者而言，面向对象的开发依然是一门艺术。对代码库的艺术扩展源于艺术家们的实践，而这些大师们依然在不断地改进他们的艺术。重构就是诸多技艺中的一种工具，它可以在不改变既有功能的前提下，改善代码的质量。

消除代码的坏味道

你可能寄希望于 Liskov 替换原则与迪米特法则，能够永远地防止你写出拙劣的代码。不过，更实际的做法是运用这些准则来帮助发现代码的坏味道，然后修复它。这是一种通用的实践：先写出可工作的代码，然后找出代码的问题，并且修复它，以提升代码质量。但是该如何准确地识别问题呢？答案就是找到代码的坏味道。在 *Refactoring: Improving the Design of Existing Code*（由 Fowler 等人在 1999 年编写）一书中描述了 22 种坏味道，并给出了相应的重构手法。

本书在运用模式时，多次使用重构来重新组织和改善已有的代码。但是在重构时，并不需要总是运用设计模式。任何时候，只要发现代码的坏味道，就值得去重构。

挑战 26.3

请提供一个有坏味道的方法，它需改进质量，改进时不能违反 LSP 或 LoD。

答案参见第 358 页

超越常规的扩展

很多设计模式，包括本书已经讲到的很多模式，都与扩展行为有关。面向扩展的模式通常涉及两种开发角色。例如在适配器模式中，一个开发者通过接口为对象提供服务，而另一个开发者则使用该服务。

挑战 26.4

请填充表 26.1 的空白处，它们都使用了设计模式来扩展类或者对象的行为。

表 26.1 使用设计模式扩展类或对象的行为

| 例 子 | 使用的模式 |
|---|--------|
| 焰火模拟器的开发者创建了一个接口，该接口定义了对象为了参与模拟所必须处理的行为 | 适配器模式 |
| 一种工具，可以让你在运行时组合可执行的对象 | ？ |
| ？ | 模板方法模式 |
| ？ | 命令模式 |
| 一个代码生成器，通过插入行为提供一种假象，使得运行在其他机器上的对象就像在本地执行一样 | ？ |
| ？ | 观察者模式 |
| 该设计允许定义一个抽象的操作，该操作取决于一个定义良好的接口，并且允许添加新的驱动器实现该接口 | ？ |

答案参见第 359 页

除了已经介绍的模式，还剩下三个模式主要用于扩展，如表 26.2 所示。

表 26.2 用于扩展的模式

| 如果你的意图是 | 使用该模式 |
|----------------------------|-------|
| 让开发者动态组合对象的行为 | 装饰器模式 |
| 提供一个方法来顺序访问集合中的元素 | 迭代器模式 |
| 允许开发者定义一个新的操作，而无须改变分层体系中的类 | 访问者模式 |

小结

编写代码的主要目的是扩展新的功能，这需要重新组织代码，改善代码的质量。世上并不存在完整而客观评估代码质量的方法，但是，面向对象的一些好的设计原则业已提出。

Liskov 替换原则要求子类的实例应该具有父类的全部功能。你应该能够识别和评估违反该原则的代码。迪米特法则是一组规则，可以降低类之间的依赖，使代码变得更加整洁。

Martin Fowler 等人在 1999 年归纳了代码坏味道的一组例子。每种坏味道都可以通过重构来消除，一些坏味道可以重构为设计模式。很多设计模式都可以用来标识、简化或者帮助扩展。

第 27 章

装饰器（Decorator）模式

通常，你会添加一些新类或新的方法去扩展已有的代码库。然而有时，你也想要在程序运行期间为某个对象组合新的行为。比如，借助解释器模式，你可以组合一个行为变化非常快的对象，这取决于你如何组合它。在某些情况下，你需要对象的行为发生一些细小的变化，并且这些变化可以进行组合。这时，装饰器模式就可以满足这个需求。

装饰器模式的意图是在运行时组合操作的新变化。

经典范例：流和输出器

Java 类库中输入输出流的设计思路是应用装饰器模式的一个经典范例。流是一系列字节或字符的集合，就像文档中的字符集合一样。在 Java 中，`writer` 类是支持流的一个方式。某些输出器类的构造函数参数是其他的输出器，因此你可以基于一个输出器创建另一个输出器。像这种简单的组合就是装饰器模式的典型结构。Java 中的输出器就体现了装饰器模式的思路。同时，我们也应该看到，仅仅需要很少的代码量，借助于装饰器模式就可以大大扩展我们在读（输入）和写（输出）操作中融合小变化的能力。

下面是 Java 中的一个应用装饰器模式的例子，代码创建了一个小的文本文件：

```
package app.decorator;
import java.io.*;

public class ShowDecorator {
```

```
public static void main(String[] args)
    throws IOException {
    FileWriter file = new FileWriter("sample.txt");
    BufferedWriter writer = new BufferedWriter(file);
    writer.write("a small amount of sample text");
    writer.newLine();
    writer.close();
}
```

运行这段程序将会产生一个名为 `sample.txt` 的文本文件，其内容为很少量的简单文本。它使用 `FileWriter` 对象创建新文件，并把这个对象包含在 `BufferedWriter` 对象中。其中，最值得注意的是，我们可以从一个流组合另一个流：这段代码从 `FileWriter` 对象组合得到 `BufferedWriter` 对象。

在 Oozinoz 公司中，销售人员需要从产品数据库的文本文件中格式化出客户信息。这些信息没有使用形式多样的字体或者风格，但是现在销售人员希望信息能够进行格式化调整，使之更加整洁漂亮。为支持这个计划，我们创建了装饰器框架。这些装饰器类允许组合很多不同种类的输出过滤器。

为了开发过滤器类集合，需要首先创建一个抽象类，它将定义一些过滤器所支持的操作。通过选择已经存在于 `Writer` 类中的操作，可以轻松创建一个类，该类可以继承 `Writer` 类的所有行为。图 27.1 说明了这种思路。

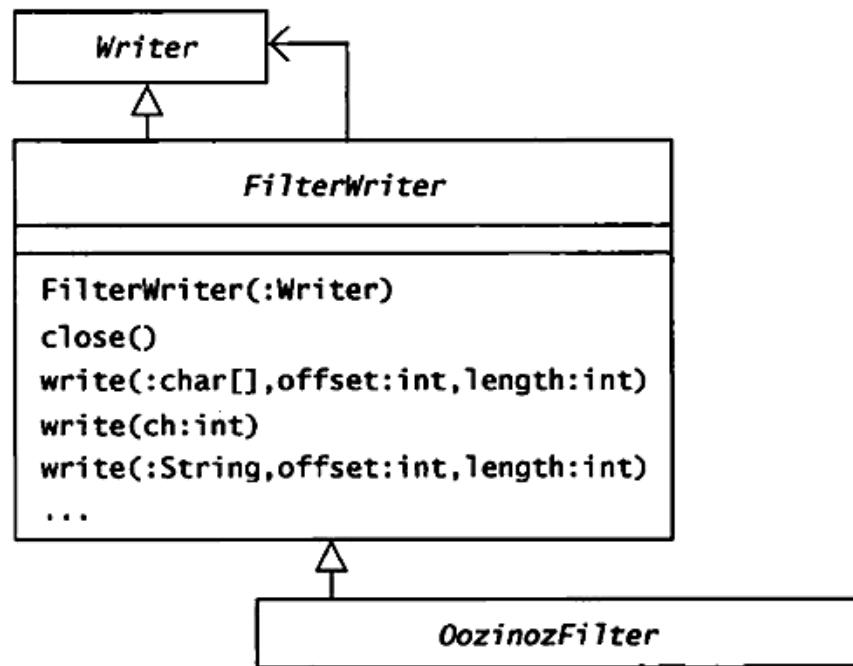


图 27.1 `OozinozFilter` 类是装饰输出字符流的类的超类

我们将定义一个过滤器类，将输出器作为它的构造函数参数，并且在 `write()` 方法中混入新的行为。

为了创建一个可组合的输出流工具箱，下一步将创建一个具有多个关键属性的过滤器超类。这个过滤器类将：

- 接收一个 `Writer` 对象作为其构造函数参数。
- 作为一个过滤器类层次的超类。
- 提供所有 `Writer` 方法 (`write(:int)` 除外) 的默认实现。

这种设计思路如图 27.2 所示。

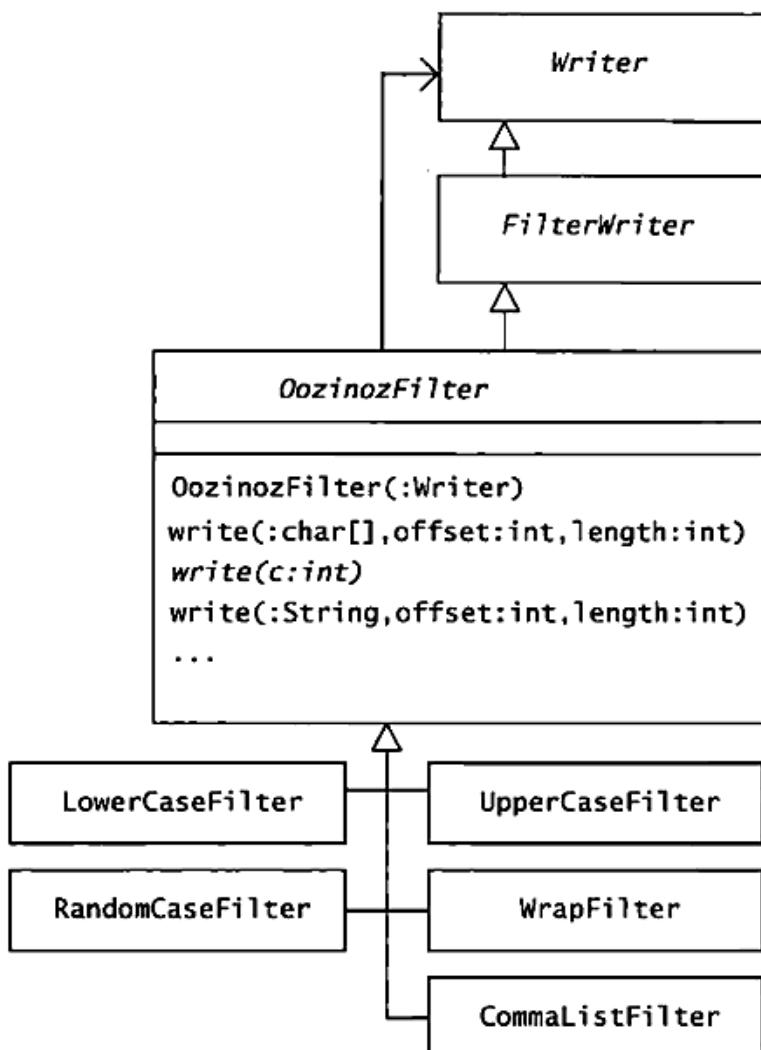


图 27.2 OozinozFilter 类的构造函数接收 Writer 类的任何子类的实例

OozinozFilter 类的设计编码实现如下所示：

```
package com.oozinoz.filter;
```

```
import java.io.*;
public abstract class OozinozFilter extends FilterWriter {
    protected OozinozFilter(Writer out) {
        super(out);
    }

    public void write(char cbuf[], int offset, int length)
        throws IOException {
        for (int i = 0; i < length; i++)
            write(cbuf[offset + i]);
    }

    public abstract void write(int c) throws IOException;

    public void write(String s, int offset, int length)
        throws IOException {
        write(s.toCharArray(), offset, length);
    }
}
```

这段代码足以保证装饰器模式的实现。OozinozFilter 类的子类可以实现 write(:int) 方法，在传递给底层流的 write(:int) 方法之前对字符进行修改。OozinozFilter 类中的其他方法提供了子类需要的行为。该类把 close() 方法和 flush() 方法的调用丢给了它的超类 FilterWriter。OozinozFilter 类通过将 write(:int) 方法定义为抽象方法，对 write(:char[]) 方法进行了说明。

现在很容易创建和使用新的流过滤器。比如，下面的代码把文本全部变成了小写格式：

```
package com.oozinoz.filter;
import java.io.*;

public class LowerCaseFilter extends OozinozFilter {
    public LowerCaseFilter(Writer out) {
        super(out);
    }

    public void write(int c) throws IOException {
        out.write(Character.toLowerCase((char) c));
    }
}
```

如下代码使用了这个小写过滤器：

```
package app.decorator;

import java.io.IOException;
import java.io.Writer;

import com.oozinoz.filter.ConsoleWriter;
import com.oozinoz.filter.LowerCaseFilter;

public class ShowLowerCase {
    public static void main(String[] args)
        throws IOException {
        Writer out = new ConsoleWriter();
        out = new LowerCaseFilter(out);
        out.write("This Text, notably ALL in LoWeR casE!");
        out.close();
    }
}
```

这段程序向控制台输出“this text, notably all in lower case!”。

`UpperCaseFilter` 类的实现代码与 `LowerCaseFilter` 类类似，除了 `write()` 方法的实现略有不同，差别如下：

```
public void write(int c) throws IOException {
    out.write(Character.toUpperCase((char) c));
}
```

`TitleCaseFilter` 类的代码实现稍微复杂一些，因为它需要跟踪空格。

```
package com.oozinoz.filter;
import java.io.*;

public class TitleCaseFilter extends OozinozFilter {
    boolean inwhite = true;
    public TitleCaseFilter(Writer out) {
        super(out);
    }

    public void write(int c) throws IOException {
```

```
        out.write(
            i < white
                ? Character.toUpperCase((char) c)
                : Character.toLowerCase((char) c));
        inwhite = Character.isWhitespace((char) c)
        || c == '\"';
    }
}
```

CommaListFilter 类在元素之间添加了逗号:

```
package com.oozinoz.filter;

import java.io.IOException;
import java.io.Writer;

public class CommaListFilter extends OozinozFilter {
    protected boolean needComma = false;

    public CommaListFilter(Writer writer) {
        super(writer);
    }

    public void write(int c) throws IOException {
        if (needComma) {
            out.write(',');
            out.write(' ');
        }
        out.write(c);
        needComma = true;
    }

    public void write(String s) throws IOException {
        if (needComma)
            out.write(", ");
        out.write(s);
        needComma = true;
    }
}
```

这些过滤器的主题是一样的：开发任务主要是重写合适的 `write()` 方法。`write()` 方法装饰已经接收到的文本流，并把修改后的文本传递给接下来的流。

挑战 27.1

完成 `RandomCaseFilter.java` 的程序代码。

答案参见第 359 页

`WrapFilter` 类的代码比其他过滤器复杂很多。它需要将输出集中处理，因此在传递给接下来的流之前，需要缓存输出并计算字符数量。你可以从 www.oozinoz.com 网站下载查看 `WrapFilter.java` 代码（下载的详细信息参见附录 C）。

`WrapFilter` 类的构造函数接收一个 `Writer` 对象和一个 `width` 参数，其中 `width` 参数可以告知从何处换行。你可以将这个过滤器和其他过滤器混合使用去输出各种各样的效果。比如，如下程序会对输入文件中的文本进行换行、居中以及首字母大写等处理。

```
package app.decorator;
import java.io.*;
import com.oozinoz.filter.TitleCaseFilter;
import com.oozinoz.filter.WrapFilter;

public class ShowFilters {
    public static void main(String args[])
        throws IOException {
        BufferedReader in = new BufferedReader(
            new FileReader(args[0]));
        Writer out = new FileWriter(args[1]);
        out = new WrapFilter(new BufferedWriter(out), 0);
        out = new TitleCaseFilter(out);

        String line;
        while ((line = in.readLine()) != null)
            out.write(line + "\n");

        out.close();
        in.close();
    }
}
```

为了查看这段程序的动态运行效果，假定输入文件 `adcopy.txt` 包含下面的内容：

```
The "SPACESHOT" shell hovers
    at 100 meters for 2 to 3
minutes,      erupting star bursts every 10 seconds that
generate          ABUNDANT reading-level light for a
typical stadium.
```

以命令行的方式执行 `ShowFilters` 程序：

```
>ShowFilters adcopy.txt adout.txt
```

`adout.txt` 文件的内容将会类似于：

```
The "Spaceshot" Shell Hovers At 100
Meters For 2 To 3 Minutes, Erupting Star
Bursts Every 10 Seconds That Generate
Abundant Reading-level Light For A
Typical Stadium.
```

如果不写入文件，也可以把输出字符直接写到控制台。图 27.3 给出了 `Writer` 类的子类 `ConsoleWriter` 的设计，它能将输出字符直接输出到控制台。

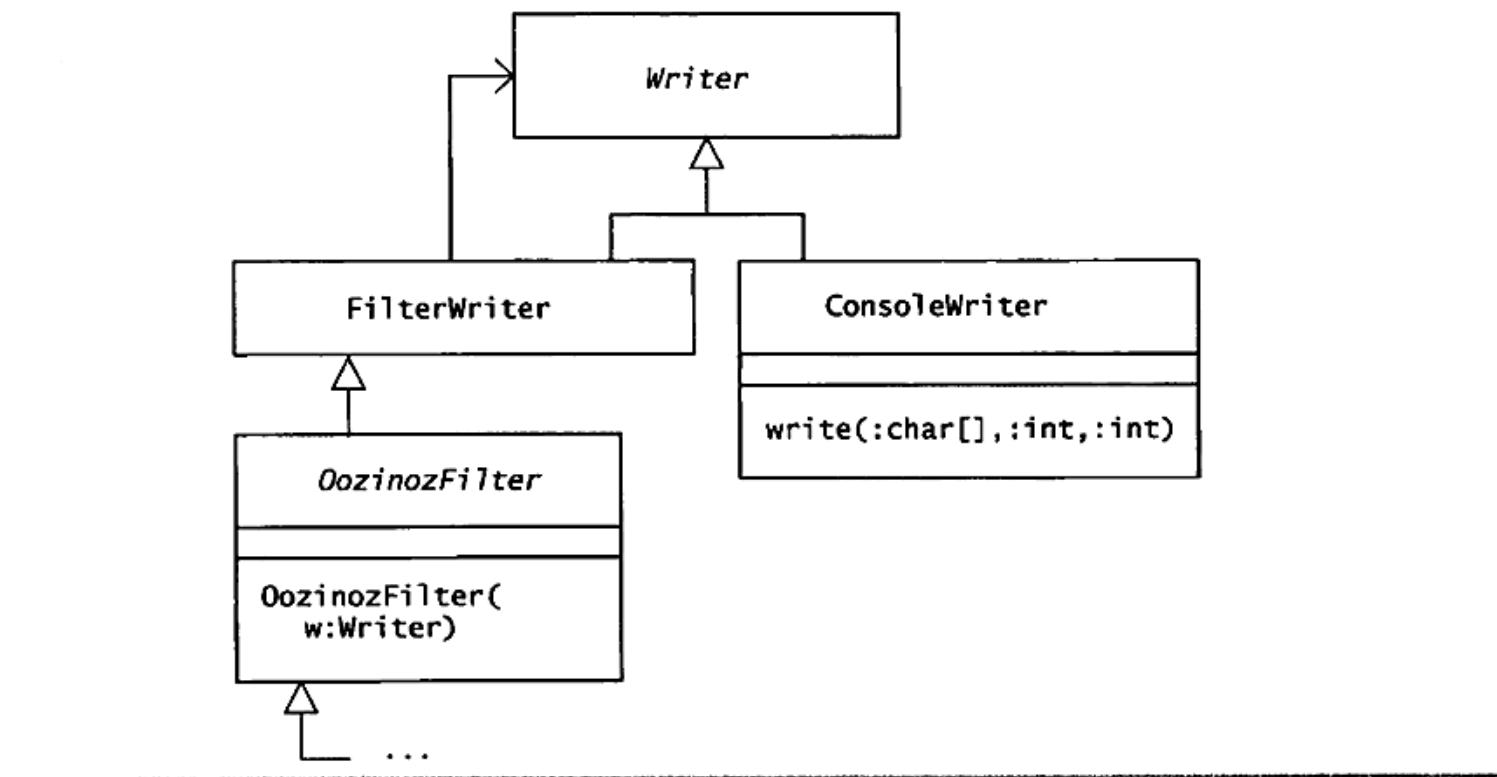


图 27.3 `ConsoleWriter` 对象可以作为任何 `OozinozFilter` 子类的构造函数参数

挑战 27.2

写出 `ConsoleWriter.java` 的代码实现。

答案参见第 360 页

输入输出流是关于装饰器模式如何在程序运行时组合对象行为的经典实例。装饰器模式的另一个重要应用则是在运行时创建数学函数。

函数包装器

如同装饰器模式在输入输出流应用中获得的巨大成功，通过使用装饰器模式在运行时组合新的行为并创建数学函数也取得了不错的效果。运行时创建新函数的能力可以传递给用户，使得他们通过 GUI 或者简单语言就可以实现新函数。通过把数学函数作为对象而不是新的方法来创建，可以减少代码里方法的数量，还可以获得更好的灵活性。

为了创建函数装饰器库（函数包装器），可以使用一个类似于 I/O 流的类结构。对于函数包装器超类名，我们使用 `Function`。对于 `Function` 类的初始设计，可以模仿 `OozinozFilter` 类的设计思路，如图 27.4 所示。

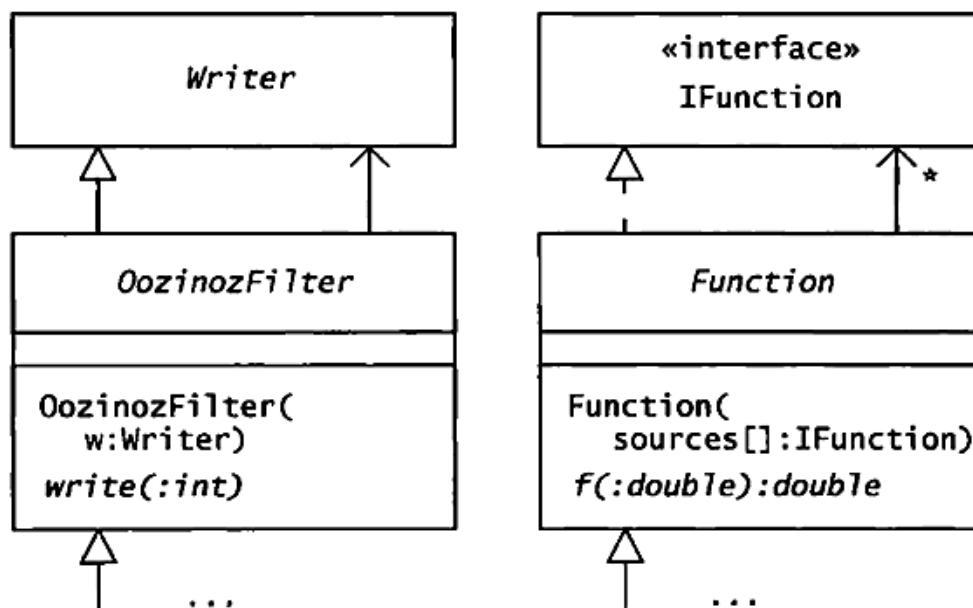


图 27.4 函数包装器（类结构）的初始设计类似于 I/O 过滤器的设计

`OozinozFilter` 类是 `Filterwriter` 类的子类，它的构造函数能够接收其他 `writer` 对象。`Function` 类的设计思路与此类相似：不同于接收单个 `IFunction` 对象，`Function` 类接收一个数组。一些函数，比如算术函数，需要更多的下一级函数共同作用。

对于函数包装器，没有类像 `writer` 类一样已经实现了我们需要的操作。因此，并不真正需要 `IFunction` 接口。通过删去该接口的定义，我们可以简化定义 `Function` 类的层次结构，如图 27.5 所示。

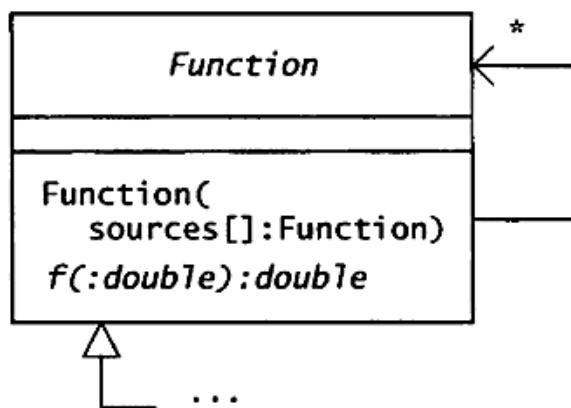


图 27.5 无须定义单独的接口，`Function` 类的简单设计同样可以实现需求

和 `OozinozFilter` 类一样，`Function` 类定义公共操作，其子类必须实现此公共操作。我们很自然地将这个公共操作命名为 `f`。同时，也可以计划实现参数函数，使所有函数基于一个标准的介于 0 到 1 之间的“time”参数。可以使用工具栏中的 Parametric Equations 菜单，在第 4 章还可以了解使用参数方程的影响力背景。

对于希望包装的每个函数，我们都将为它们创建一个 `Function` 类的子类。图 27.6 是初始的 `Function` 类层次结构的设计思路。

`Function` 超类的代码主要用于声明 `sources` 数组：

```
package com.oozinoz.function;

public abstract class Function {
    protected Function[] sources;

    public Function(Function f) {
        this(new Function[] { f });
    }
}
```

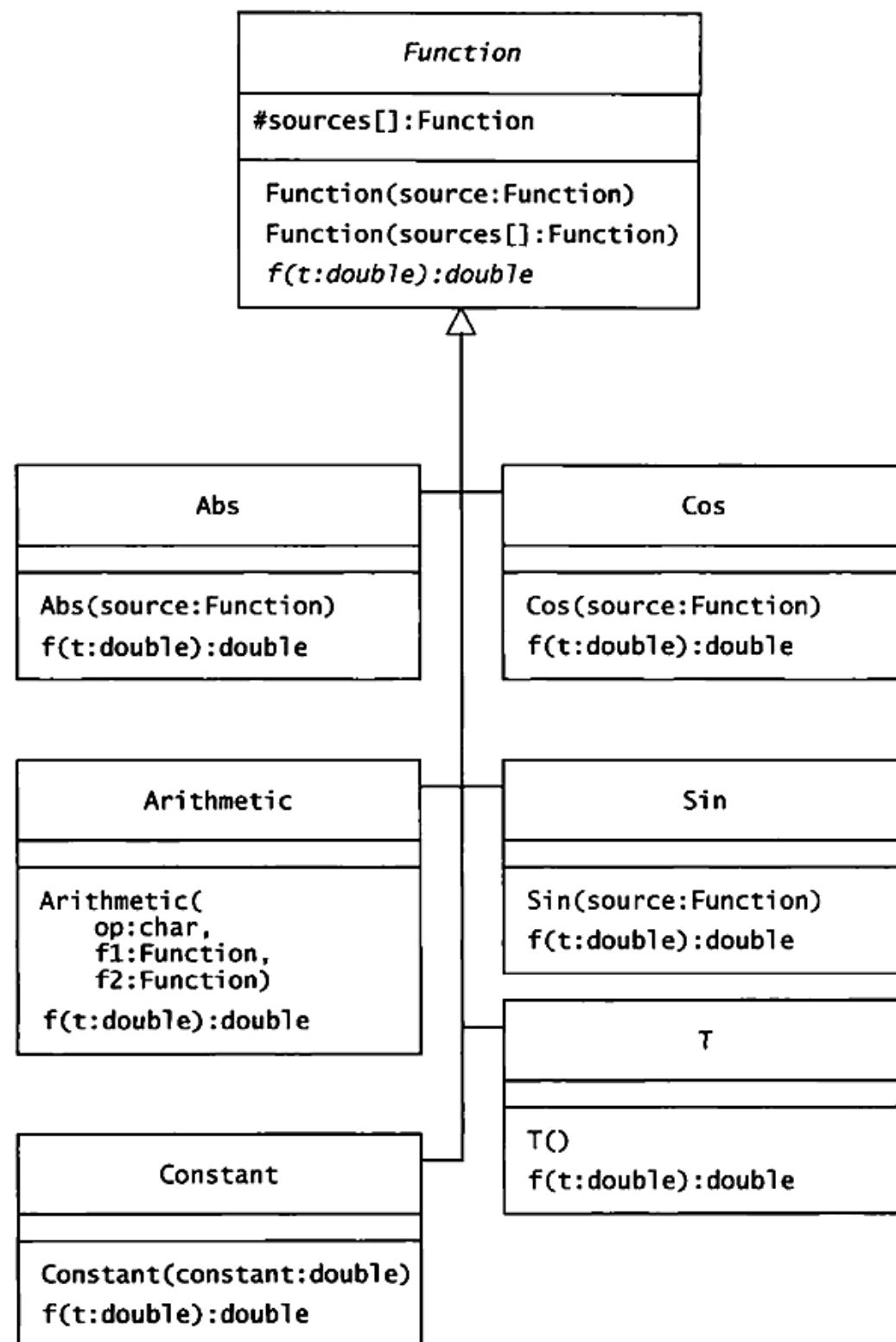


图 27.6 Function 类的每一个子类按照类名的不同含义实现 f(t)

```

public Function(Function[] sources) {
    this.sources = sources;
}
  
```

```
public abstract double f(double t);

public String toString() {
    String name = this.getClass().toString();
    StringBuffer buf = new StringBuffer(name);
    if (sources.length > 0) {
        buf.append('(');
        for (int i = 0; i < sources.length; i++) {
            if (i > 0)
                buf.append(", ");
            buf.append(sources[i]);
        }
        buf.append(')');
    }
    return buf.toString();
}
```

通常，Function 子类都比较简单。例如，下面是 Cos 类的代码：

```
package com.oozinoz.function;
public class Cos extends Function {
    public Cos(Function f) {
        super(f);
    }

    public double f(double t) {
        return Math.cos(sources[0].f(t));
    }
}
```

Cos 类的构造函数期待接收一个 Function 对象参数，并把这个参数传递给它的超类构造函数，参数存储在 sources 数组中。Cos.f() 方法在时间 t 对原函数求值，将值传递给 Math.Cos()，并返回结果。

Abs 类和 Sin 类与 Cos 类几乎完全一样。借助 Constant 类，可以创建拥有一个常量值的 Function 对象，并作为 f() 方法的调用返回。Arithmetic 类接收一个操作符指示器，并将其运用到 f() 方法中。Arithmetic 类的代码如下所示：

```
package com.oozinoz.function;
```

```
public class Arithmetic extends Function {  
    protected char op;  
  
    public Arithmetic(char op, Function f1, Function f2) {  
        super(new Function[] { f1, f2 });  
        this.op = op;  
    }  
  
    public double f(double t) {  
        switch (op){  
            case '+':  
                return sources[0].f(t) + sources[1].f(t);  
            case '-':  
                return sources[0].f(t) - sources[1].f(t);  
            case '*':  
                return sources[0].f(t) * sources[1].f(t);  
            case '/':  
                return sources[0].f(t) / sources[1].f(t);  
            default:  
                return 0 ;  
        }  
    }  
}
```

T类返回传入的t值。如果希望一个变量随着时间线性变化，那么这样做会很有用。比如，下面的代码创建了一个Function对象，随着时间从0变化到1，该对象的f()值从0变化到 2π ：

```
new Arithmetic('*', new T(), new Constant(2 * Math.PI))
```

可以使用Function类去组合新的数学函数，而无须编写新的方法。FunPanel类为它的x和y函数接收Function参数。这个类也使得这些函数可以在固定面板范围内使用。这个面板还可以和如下代码结合使用：

```
package app.decorator;  
  
import app.decorator.brightness.FunPanel;  
  
import com.oozinoz.function.*;  
import com.oozinoz.ui.SwingFacade;
```

```
public class ShowFun {  
    public static void main(String[] args) {  
        Function theta = new Arithmetic(  
            '*', new T(), new Constant(2 * Math.PI));  
        Function theta2 = new Arithmetic(  
            '*', new T(), new Constant(2 * Math.PI * 5));  
        Function x = new Arthmetic(  
            '+', new Cos(theta), new Cos(theta2));  
        Function y = new Arithmetric(  
            '+', new Sin(theta), new Sin(theta2));  
  
        FunPanel panel = new FunPanel(1000);  
        panel.setPreferredsize(  
            new java.awt.Dimension(200, 200));  
  
        panel.setXY(x, y);  
        SwingFacade.launch(panel, "Chrysanthemum");  
    }  
}
```

这段程序画出了一组相互嵌套的圆形，程序运行后的效果如图 27.7 所示。

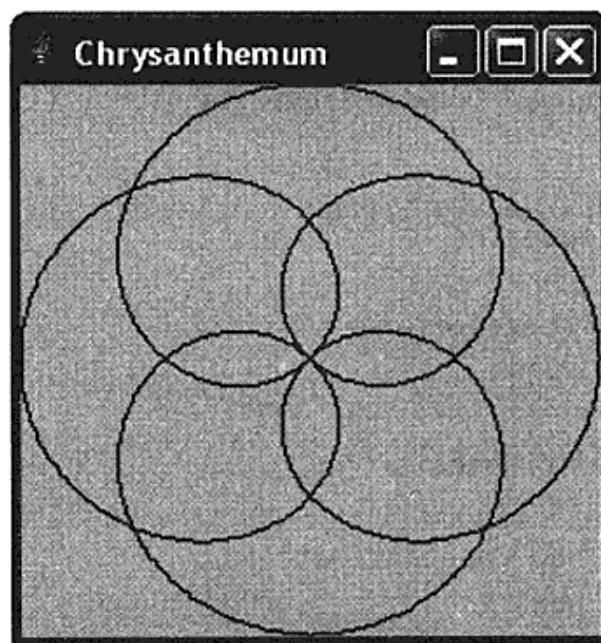


图 27.7 在没有引入任何新方法的情况下，一个复杂的数学函数被创建

通过简单地向 `Function` 类层次结构中加入新的数学函数，你就可以扩展你自己的函数包装器。

挑战 27.3

请写出 Exp 函数包装器类的代码实现。(试着合上书去编写代码!)

答案参见第 361 页

假定星形火药的亮度是 (随时间) 呈指数下降的正弦波函数:

$$\text{brightness} = e^{-4t} \cdot \sin(\pi t)$$

和前面一样, 不需要编写新的类或方法就可以组合函数 (绘制亮度随时间变化的曲线):

```
package app.decorator.brightness;

import com.oozinoz.function.*;
import com.oozinoz.ui.SwingFacade;

public class ShowBrightness {
    public static void main(String args[]) {
        FunPanel panel = new FunPanel();
        panel.setPreferredSize(
            new java.awt.Dimension(200, 200));

        Function brightness = new Arithmetic(
            '*',
            new Exp(
                new Arithmetic(
                    '*',
                    new Constant(-4),
                    new T())),
            new Sin(
                new Arithmetic(
                    '*',
                    new Constant(Math.PI),
                    new T())));

        panel.setXY(new T(), brightness);

        SwingFacade.launch(panel, "Brightness");
    }
}
```

这段代码所绘制出的曲线如图 27.8 所示。

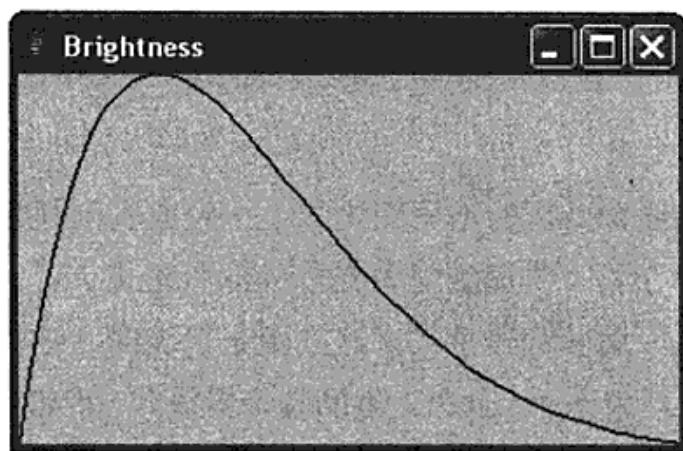


图 27.8 星形火药的亮度迅速达到顶点随后便开始衰减

挑战 27.4

请编写代码来定义表示亮度函数的 `Brightness` 对象。

答案参见第 361 页

你可以根据需要向 `Function` 类层次结构加入其他函数。例如，可能会加入 `Random` 类、`Sqrt` 类以及 `Tan` 类。也可以创建新的类层次结构来处理其他不同的类型，比如字符串类型，或者，采用不同的方法来定义 `f()` 操作。例如，你可以把 `f()` 定义为时间的二维或者三维函数。总而言之，不论创建了什么样的类层次结构，都可以使用装饰器模式在运行时组合出丰富的函数。

装饰器模式和其他设计模式的关系

装饰器模式的机制中包含一个跨越类层次结构实现的公共操作。从这个角度来讲，装饰器模式类似于 `State`（状态）模式、`Strategy`（策略）模式和 `Interpreter`（解释器）模式。在装饰器模式中，类通常拥有需要其他后继装饰器对象的构造函数。从这一点上讲，装饰器模式类似于合成模式。装饰器模式也类似于代理模式，因为装饰器类通常通过把调用转发给后继装饰器对象以实现公共操作。

小结

借助装饰器模式，可以混合操作的不同变化。一个经典实例是输入输出流，可以从其他流组合一个新的流。Java 类库在其 I/O 流实现中支持装饰器模式。可以扩展这个思路，创建自己的 I/O 过滤器集合。还可以应用装饰器模式来建立函数包装器，这样就能够根据有限的函数类集合来创建数量庞大的函数对象系列。此外，借助装饰器模式，能够灵活设计具有公共操作（这些公共操作往往具有不同的实现方式）的类，这样就可以在运行时集成新的混合的变化。

第28章

迭代器（Iterator）模式

若要通过新增集合类型扩展代码库，可以增加一个迭代器来完成扩展。本章主要讨论对集合类型进行迭代的例子。除了对新集合类型的迭代之外，在多线程环境下的迭代也会引出一系列有趣的问题。迭代看似简单，但是它并不能完全解决问题。

迭代器模式的意图是为顺序访问集合元素提供一种方式。

普通的迭代

Java 语言中提供了一系列方式来进行迭代：

- `for`、`while` 和 `repeat` 循环，通常使用整数索引。
- `Enumeration` 类（在 `java.util` 中）。
- `Iterator` 类（也在 `java.util` 中），在 JDK 1.2 中支持集合。
- `for` 循环的扩展（`foreach`），在 JDK 1.5 中添加。

我们将重点介绍迭代器类，而本小节将展示对循环的扩展。

一个 `Iterator` 类包含三个方法：`hasNext()`、`next()` 和 `remove()`，如果迭代器不支持 `remove()` 操作，会抛出 `UnsupportedOperationException()` 异常。

for 循环扩展后的形式如下：

```
for (Type element : collection)
```

它将针对集合创建一个循环，每次取出集合中的一项：这里称为 `element`。它并不需要显式地将元素转换为特定类型，转换过程都是隐式完成的。该结构同样适用于数组。如果一个类希望支持 for 循环，必须实现 `Iterable` 接口，并提供一个 `iterator()` 方法。

下面的程序演示了 `Iterator` 类和增强后的 for 循环：

```
package app.iterator;

import java.util.ArrayList
import java.util.Iterator;
import java.util.List;

public class ShowForeach {
    public static void main(String[] args) {
        ShowForeach example = new ShowForeach();
        example.showIterator();
        System.out.println();
        example.showForeach();
    }

    public void showIterator() {
        List names = new ArrayList();
        names.add("Fuser:1101");
        names.add("StarPress:991");
        names.add("Robot:1");

        System.out.println("JDK 1.2-style Iterator:");
        for (Iterator it = names.iterator(); it.hasNext();) {
            String name = (String) it.next();
            System.out.println(name);
        }
    }

    public void showForeach() {
        List<String> names = new ArrayList<String>();
        names.add("Fuser:1101");
    }
}
```

```
names.add("StarPress:991");
names.add("Robot:1");

System.out.println(
    "JDK 1.2-style Iterator:");
for (String name: names)
    System.out.println(name);
}

}
```

当运行该程序后，将得到如下输出：

```
JDK 1.2-style Iterator:
```

```
Fuser:1101
```

```
StarPress:991
```

```
Robot:1
```

```
JDK 1.5-style Extended For Loop:
```

```
Fuser:1101
```

```
StarPress:991
```

```
Robot:1
```

到目前为止，Oozinoz 公司还在继续使用旧版本的 `Iterator` 类，并且暂时无法升级，除非确认客户有新版本的编译器。尽管如此，从今天开始你就该为学习扩展的 `for` 循环做好准备了。

线程安全的迭代

交互良好的应用程序通常使用线程来同时执行多个任务。对于那些耗费时间的任务，让它们在后台执行是极为常见的做法，因为这样就不会拖延 GUI 的响应时间。线程很有用，但也很危险。许多应用程序崩溃的原因都是由于没有协调好线程之间的交互。毫无疑问，迭代整个集合的方法常常是导致多线程程序失败的原因。

`java.util.Collections` 包中的集合类，通过提供一个 `synchronized()` 方法来保证线程执行的安全。从本质上讲，该方法将确保两个线程在底层访问的是同一个集合版本，并会阻止多个线程在同一时间更改集合。

在迭代过程中，集合和它的迭代器相互协作共同检测列表是否变化，即判断列表是否被同步。为了说明这种行为的工作方式，以 Oozinoz 系统中的 Factory 单例为例，该对象能告诉我们在指定时间内启动了哪些机器，并显示这一组启动了的机器。`app.iterator.concurrent` 包中的示例程序在 `upMachineNames()` 方法中硬编码了这段代码。

下面的代码显示了当前已经启动的机器列表，当新机器启动时，它还可以模拟列表的显示情况：

```
package app.iterator.concurrent;
import java.util.*;

public class ShowConcurrentIterator implements Runnable {
    private List list;

    protected static List upMachineNames() {
        return new ArrayList(Arrays.asList(new String[] {
            "Mixer1201", "ShellAssembler1301",
            "StarPress1401", "UnloadBuffer1501 }));
    }

    public static void main(String[] args) {
        new ShowConcurrentIterator().go();
    }

    protected void go() {
        list = Collections.synchronizedList(
            upMachineNames());
        Iterator iter = list.iterator();
        int i = 0;
        while (iter.hasNext()) {
            i++;
            if (i == 2) { // 模拟唤醒线程
                new Thread(this).start();
                try { Thread.sleep(100); }
                catch (InterruptedException ignored) {}
            }
            System.out.println(iter.next());
        }
    }
}
```

```
/**  
 * 在一个单独的线程中，向 list 插入一个元素  
 */  
public void run() {  
    list.add(0, "Fuser1101");  
}  
}
```

这段代码中的 `main()` 函数创建了该类的实例，并且调用了该实例的 `go()` 方法。该方法负责迭代已经启动的机器列表，并构造该列表的同步版本。这段代码模拟了方法在迭代这一列表时，其他机器启动的情况。`run()` 方法会修改列表，并运行在一个独立的线程中。

在输出一台或者两台机器后，`ShowConcurrentIterator` 程序就崩溃了：

```
Mixer1201  
java.util.ConcurrentModificationException  
at java.util.AbstractList$Itr.checkForComodification(Unknown Source)  
at java.util.AbstractList$Itr.next(Unknown Source)  
at com.oozinoz.app.ShowConcurrent.ShowConcurrentIterator.go(Show-  
ConcurrentIterator.java:49)  
at com.oozinoz.app.ShowConcurrent.ShowConcurrentIterator.main(Show-  
ConcurrentIterator.java:29)  
Exception in thread "main" .
```

程序崩溃的原因是因为集合和迭代对象在迭代过程中检测到了集合的变化。其实，无须使用新的线程也可以展示这种行为：创建一个程序，在迭代过程中改变其集合元素，就可以让该程序崩溃。实际上，一个多线程程序在迭代器遍历集合的过程中，更可能意外地更改该集合。

我们可以在迭代集合的过程中，设计一个线程安全的方法；然而，引起 `ShowConcurrentIterator` 程序崩溃的关键原因是它使用了迭代器。在 `for` 循环中使用同步列表来进行迭代，就不会引发前面程序中出现的异常，但是依然会带来一些问题。考虑下面给出的另一个版本的程序：

```
package app.iterator.concurrent;  
import java.util.*;  
  
public class ShowConcurrentFor implements Runnable {  
    private List list;
```

```
protected static List upMachineNames() {
    return new ArrayList(Arrays.asList(new String[] {
        "Mixer1201", "ShellAssembler1301",
        "StarPress1401", "UnloadBuffer1501" }));
}

public static void main(String[] args) {
    new ShowConcurrentFor().go();
}

protected void go() {
    System.out.println(
        "This version lets new things "
        + "be added in concurrently:");

    list = Collections.synchronizedList(
        upMachineNames());
    display();
}

private void display() {
    for (int i = 0; i < list.size(); i++) {
        if (i == 1) { // 模拟唤醒线程
            new Thread(this).start();
            try { Thread.sleep(100); }
            catch (InterruptedException ignored) {}
        }
        System.out.println(list.get(i));
    }
}

/**
 * 在一个单独的线程中，向 list 中插入一个元素
 */
public void run() {
    list.add(0, "Fuser1101");
}
}
```

运行该程序，产生的输出如下所示：

This version lets new things be added in concurrently:

```
Mixer1201  
Mixer1201  
ShellAssembler1301  
StarPress1401  
UnloadBuffer1501
```

挑战 28.1

请解释 ShowConcurrentFor 程序的输出。

答案参见第 362 页

让我们来对比一下这两个版本的程序：一个程序在运行过程中崩溃，而另一个则产生了错误的输出。这两种结果我们都不能接受，因此在迭代过程中，还需要另外的方式来保护集合列表。

在多线程程序中，有两种常见的方式为集合提供安全的迭代。这两种方式都会引入一个对象，有时称为互斥(mutex)对象。该对象被多个线程共享，用于竞争对象的控制权。在其中一种设计方式中，需要所有的线程在访问集合前都要首先获得互斥锁。下面的程序展示了这种方式：

```
package app.iterator.concurrent;  
import java.util.*;  
  
public class ShowConcurrentMutex implements Runnable {  
    private List list;  
  
    protected static List upMachineNames() {  
        return new ArrayList(Arrays.asList(new String[] {  
            "Mixer1201", "ShellAssembler1301",  
            "StarPress1401", "UnloadBuffer1501" }));  
    }  
  
    public static void main(String[] args) {  
        new ShowConcurrentMutex().go();  
    }  
  
    protected void go() {  
        System.out.println(  
            "This version synchronizes properly.");  
    }  
}
```

```
list = Collections.synchronizedList(upMachineNames());
synchronized (list) {
    display();
}
}

private void display() {
    for (int i = 0; i < list.size(); i++) {
        if (i == 1) { // 模拟唤醒线程
            new Thread(this).start();
            try { Thread.sleep(100);
            } catch (InterruptedException ignored) {}
        }
        System.out.println(list.get(i));
    }
}

/**
 * 在一个单独的线程中，向 list 插入一个元素
 */
public void run() {
    synchronized (list) {
        list.add(0, "Fuser1101");
    }
}
}
```

该程序将输出原始的列表：

```
This version synchronizes properly:
Mixer1201
ShellAssembler1301
StarPress1401
UnloadBuffer1501
```

这份集合的输出结果，与插入新对象之前运行 `run()` 方法的集合结果是一致的。程序的输出是连续的，没有重复，因为该程序的逻辑要求 `run()` 方法在等待 `display()` 方法完成后才能执行。尽管输出正确，但这种设计却有些华而不实，毕竟当一个线程正在迭代集合时，不可能让其他线程等待。

一个替代方案是在互斥操作中克隆该集合，并对克隆后的集合进行迭代。这样做的好处是

效率高。在操作集合之前克隆集合，常常要比等待其他操作集合内容的方法执行完成要快得多。遗憾的是，在迭代前对集合进行克隆也可能会带来一些问题。

`ArrayList` 的 `clone()` 方法会产生一个浅拷贝 (shallow copy)：新集合和原始集合引用相同的对象。当其他线程通过某种方式改变了引用的对象后，我们所依赖的克隆就会失效。但在某些场景下，这种风险非常小。例如，如果只是希望显示集合中的机器名，在对克隆集合进行迭代时，修改机器名就是不合理的^{译注1}。

总结下来，我们已经讨论了 4 种在多线程环境下迭代集合的方式。其中，使用 `synchronized()` 方法的两种方式都失败了，不是程序崩溃，就是产生错误的结果。后两种方式使用了锁和克隆，都能产生正确的结果，但是也会带来一些问题。

挑战 28.2

请解释反对 `synchronized()` 方法的理由，或者基于锁的方式为何不是我们要寻求的答案？

答案参见第 362 页

Java 类库对多线程环境下的迭代提供了大量支持，但这种并发设计都未能消除复杂度。对于内置集合的迭代，Java 类库提供了很好的支持，但是倘若希望引入自己的集合类型，就可能需要引入相应的迭代器。

基于合成结构的迭代

要设计遍历合成结构的算法，访问其中的每个节点，执行一些操作，其实并不困难。可以回顾一下挑战 5.3 中多种递归访问合成结构的算法。创建一个迭代器可能比创建一个递归算法

译注1：对于并发处理过程中相关问题的解决，最佳方式就是使用不变值。这里提到的对集合进行克隆的方式，其实就是保证每个集合对象都是不变的。只是在 Java 中，由于集合对象的特点，我们很难保证集合对象的不变性。而在 JVM 中的另一门语言 Scala 中，则通过 `val` 关键字，很好地保证了各种类型对象的不变性。

更加复杂。复杂之处在于返回对程序中其他部分的控制，保存多种状态，并让迭代器能从上次结束的地方继续开始迭代。

合成模式为迭代器提供了一个颇具挑战性的好例子。

你或许会认为需要给领域相关的每个组件都创建一个新的迭代器。事实上，可以先设计出一个可重用的合成迭代器，然后修改各组件类，使其利用该迭代器能进行正确的迭代。

这种合成迭代器的设计从本质上讲都是递归的。为了迭代该合成结构，我们需要迭代其子节点。乍一听，这似乎有些复杂。首先，要确定是先返回前序节点还是后续节点（分别称为前序遍历和后续遍历）。如果选择前序遍历，就必须在返回节点头后，遍历其子节点。注意，或许每个子节点本身就是一个合成结构。这种方式的关键在于需要维护两个迭代器。一个迭代器负责记录当前是哪个子节点（图 28.1 中的标记 1）。它就是一个简单的列表迭代器，负责迭代子合成结构的列表。另一个迭代器（图 28.1 中的标记 2）将当前的子节点当做合成结构进行迭代。图 28.1 给出了在合成迭代中，决定当前节点的三个方面。

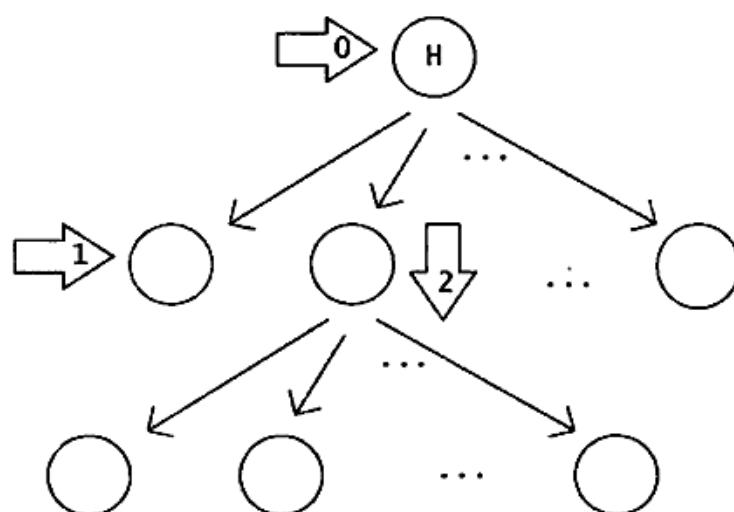


图 28.1 对合成结构进行迭代需要注意几个方面：0 代表头节点，1 代表顺序迭代其子节点，2 代表将每个子节点当做合成结构来进行迭代

要设计出合成迭代器，对叶子节点的迭代并不难，但对合成节点的迭代则可能比较困难。这些迭代器的设计如图 28.2 所示。

可以看到，类中包含了 `hasNext()` 和 `next()` 等方法，因此 `ComponentIterator` 类实现了 `java.util` 包中的 `Iterator` 接口。

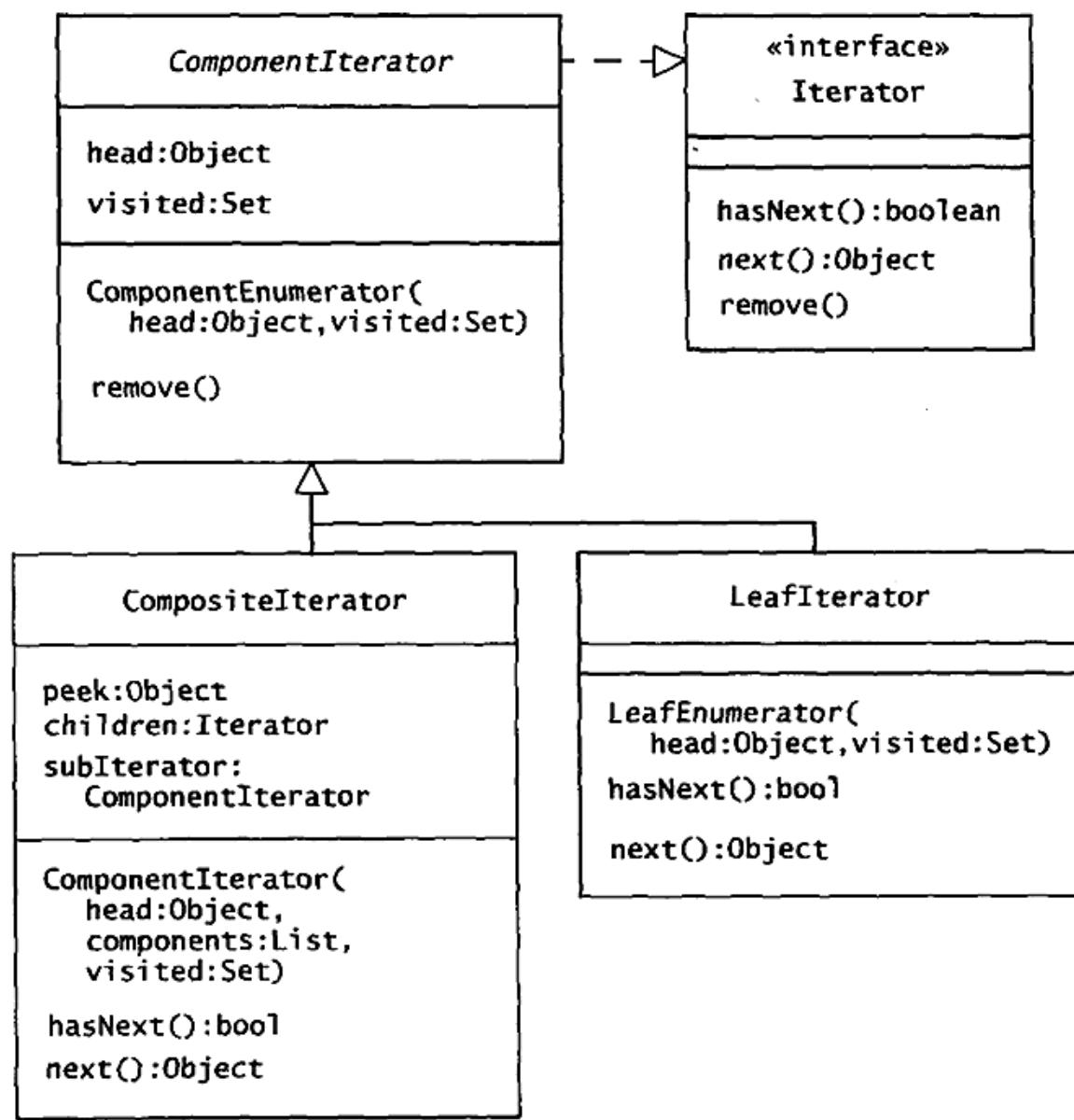


图 28.2 合成迭代器的最初设计

这一设计展现了迭代类的构造函数接收了一个对象进行迭代。在实际中，该对象将是一个合成对象，比如机器合成对象或者流程合成对象。该设计还使用了一个 `visited` 变量，用以记录已经迭代过的节点。这可以防止在合成结构包含了环时，不会进入无限循环。位于类层次结构顶部的 **ComponentIterator** 类的代码开始时如下所示：

```

package com.oozinoz.iterator;
import java.util.*;

public abstract class ComponentIterator
    implements Iterator {
    protected Object head;
}

```

```
protected Set visited;
protected boolean returnInterior = true;

public ComponentIterator(Object head, Set visited) {
    this.head = head;
    this.visited = visited;
}

public void remove() {
    throw new UnsupportedOperationException(
        "ComponentIterator.Remove");
}

}
```

该类将多数复杂的操作交给了子类去完成。

在 `CompositeIterator` 的子类中，我们期望使用一个列表迭代器，用来遍历合成节点的子节点。图 28.1 中标记为 1 的就是该迭代器，在图 28.2 中，使用了 `children` 变量来表示。合成结构也需要如图 28.1 中标记为 2 的迭代器。在图 28.2 中使用 `subiterator` 变量来表示。`CompositeIterator` 类的构造函数采用了如下形式初始化子迭代器：

```
public CompositeIterator(
    Object head, List components, Set visited) {
    super(head, visited);
    children = components.iterator();
}
```

当开始遍历一个合成结构时，返回的第一个节点是头节点（图 28.1 中标记为 H）。`CompositeIterator` 类中的 `next()` 方法如下所示：

```
public Object next() {
    if (peek != null) {
        Object result = peek;
        peek = null;
        return result;
    }

    if (!visited.contains(head)) {
        visited.add(head);
```

```
        return head;
    }

    return nextDescendant();
}
```

`next()` 方法使用 `visited` 集合来记录迭代器是否已经返回了头节点。如果是，`nextDescendant()` 方法就必须找到下一个节点。

无论何时，只要节点是合成节点，`subiterator` 变量就对其进行遍历。如果遍历的迭代器是有效的，`CompositeIterator` 类的 `next()` 方法就可以移动这个子迭代器。倘若某个子迭代器无法移动，代码就必须移到子节点列表的下一个元素，为其分配一个新的子迭代器，并且移动该迭代器。`nextDescendant()` 方法中的代码逻辑如下所示：

```
protected Object nextDescendant() {
    while (true) {
        if (subiterator != null) {
            if (subiterator.hasNext())
                return subiterator.next();
        }

        if (!children.hasNext()) return null;

        ProcessComponent pc =
            (ProcessComponent) children.next();
        if (!visited.contains(pc)) {
            subiterator = pc.iterator(visited);
        }
    }
}
```

对于支持遍历的对象类型，该方法引入了第一个约束：代码要求合成对象的子节点必须实现 `iterator(:Set)` 方法。考虑合成结构的一个例子，比如第 5 章介绍的流程组件类层次结构。图 28.3 展示了 Oozinoz 公司对制造各种焰火弹组件的流程进行建模所形成的合成层次结构。

`CompositeIterator` 类的 `next()` 方法需要去遍历合成对象的每个子节点。我们需要子类实现 `iterator(:Set)` 方法，以便使用 `next()` 中的代码。图 28.2 展示了这些类和接口之间的关系。

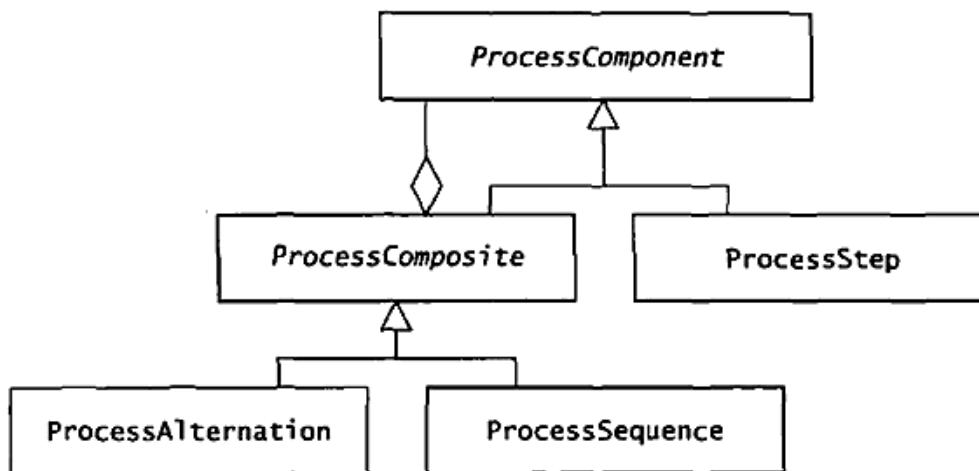


图 28.3 Oozinoz 公司的制造流程为合成结构

为了更新 `ProcessComponent` 类层次关系，以便能够遍历它，需要提供一个 `iterator()` 方法。

```

public ComponentIterator iterator() {
    return iterator(new HashSet());
}

public abstract ComponentIterator iterator(Set visited);
  
```

`ProcessComponent` 类是一个抽象类，它的抽象方法 `iterator(:Set)` 由其子类实现。
`ProcessComposite` 类的代码如下所示：

```

public ComponentIterator iterator(Set visited) {
    return new CompositeIterator(this, subprocesses, visited);
}
  
```

实现了 `iterator()` 方法的 `ProcessStep` 类如下所示：

```

public ComponentIterator iterator(Set visited) {
    return new LeafIterator(this, visited);
}
  
```

挑战 28.3

如果让 `ProcessComponent` 类层次结构中的类实现 `iterator()` 方法，从而创建合适的迭代器类实例，你会使用哪个模式？

答案参见第 362 页

在调整了 `ProcessComponent` 类层次结构的相应位置后，就可以为合成流程的遍历编写代码了。图 28.4 展示了 Oozinoz 公司典型的流程对象模型。

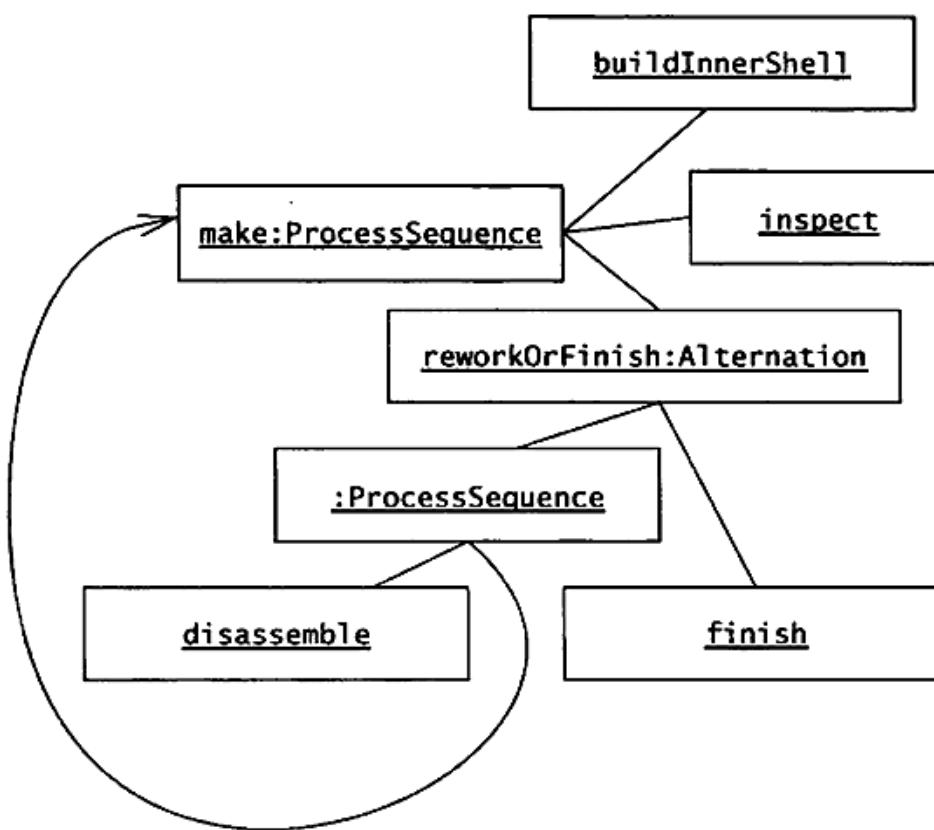


图 28.4 制作高空焰火弹的过程是一个循环的合成结构。图中的每个叶子节点都是一个 `ProcessStep` 类的实例。剩下的节点是 `ProcessComposite` 类的实例

`com.oozinoz.processes` 包中的 `ShellProcess` 类定义了一个静态的 `make()` 方法，该方法返回图 28.4 所示的对象模型。下面的程序将会迭代模型中的所有节点。

```
package app.iterator.process;

import com.oozinoz.iterator.ComponentIterator;
import com.oozinoz.process.ProcessComponent;
import com.oozinoz.process.ShellProcess;

public class ShowProcessIteration {
    public static void main(String[] args) {
        ProcessComponent pc = ShellProcess.make();
        ComponentIterator iter = pc.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

程序产生如下输出：

```
Make an aerial shell
Build inner shell
Inspect
Rework inner shell, or complete shell
Rework
Disassemble
Finish: Attach lift, insert fusing, wrap
```

输出的名字是 `ShellProcess` 类分配给每个步骤的名字。注意，在对象模型中，`Disassemble` 后的步骤是 `make`。输出忽略了这个步骤，因为第一行输出中已经输出过该步骤。

为合成迭代器增加深度

如果我们对每个步骤按其在模型中的深度进行缩进，则该程序的输出结果看起来将更加清晰。可以将叶迭代器的深度定义为 0，合成迭代器的当前深度是子迭代器的深度加 1。可以将超类 `ComponentIterator` 中的抽象方法 `getDepth()` 定义为如下形式：

```
public abstract int getDepth();
```

`LeafIterator` 类的 `getDepth()` 方法如下所示：

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

`CompositeIterator.getDepth()` 中的代码如下所示：

```
public int getDepth() {
    if (subiterator != null)
        return subiterator.getDepth() + 1;
    return 0;
}
```

下面程序的输出具有更高的可读性：

```
package app.iterator.process;

import com.oozinoz.iterator.ComponentIterator;
import com.oozinoz.process.ProcessComponent;
```

```
import com.oozinoz.process.ShellProcess;

public class ShowProcessIteration2 {
    public static void main(String[] args) {
        ProcessComponent pc = ShellProcess.make();
        ComponentIterator iter = pc.iterator();
        while (iter.hasNext()) {
            for (int i = 0; i < 4 * iter.getDepth(); i++)
                System.out.print(' ');
            System.out.println(iter.next());
        }
    }
}
```

程序的输出如下：

```
Make an aerial shell
    Build inner shell
    Inspect
    Rework inner shell, or complete shell
        Rework
        Disassemble
    Finish: Attach lift, insert fusing, wrap
```

另一种改进 `ComponentIterator` 类层次结构的方法是，只允许其遍历合成结构的叶子节点。

遍历叶子节点

假设只允许迭代返回叶子节点。如果需要了解叶子节点的属性，例如单个步骤的执行时间，这种做法是有价值的。可以为 `ComponentIterator` 类增加一个 `returnInterior` 字段，用来记录内部（非叶子）节点是否应该从遍历中返回：

```
protected boolean returnInterior = true;

public boolean shouldShowInterior() {
    return returnInterior;
}
```

```
public void setShowInterior(boolean value) {  
    returnInterior = value;  
}
```

在 `CompositeIterator` 类的 `nextDescendant()` 方法中，当需要为合成节点的子节点创建一个新的迭代器时，需要将这一属性传递下去：

```
protected Object nextDescendant() {  
    while (true) {  
        if (subiterator != null) {  
            if (subiterator.hasNext())  
                return subiterator.next();  
        }  
        if (!children.hasNext()) return null;  
  
        ProcessComponent pc =  
            (ProcessComponent) children.next();  
        if (!visited.contains(pc)) {  
            subiterator = pc.iterator(visited);  
            subiterator.setShowInterior(  
                shouldShowInterior());  
        }  
    }  
}
```

我们同样需要更新 `CompositeEnumerator` 类的 `next()` 方法。代码如下：

```
public Object next() {  
    if (peek != null) {  
        Object result = peek;  
        peek = null;  
        return result;  
    }  
  
    if (!visited.contains(head)) {  
        visited.add(head);  
    }  
  
    return nextDescendant();  
}
```

挑战 28.4

更新 `CompositeIterator` 类中的 `next()` 方法，使其获取 `returnInterior` 字段的值。

答案参见第 363 页

为某个新的集合类型创建一个迭代器 (`iterator`)，是一件非常有意义的设计任务。这样做的好处是自定义集合能够像 Java 类库中的集合那样轻松使用。

小结

迭代器模式的意图是让客户端类可以顺序地访问集合元素。Java 类库中的集合类为集合的操作提供了强大的支持，包括支持迭代或者遍历。若要创建新的集合类型，通常需要为其创建一个迭代器。与领域相关的合成结构就是一种新的集合类型。对通用集合的支持，以及对 `for` 循环的增强，可以使代码的可读性更好。可以设计一个通用的迭代器，并将其用到各种组件的层次结构中。

在实例化一个迭代器时，应该考虑对其集合的遍历是否更改了集合的内容。在单线程程序中，通常不会发生这种情况。但在多线程程序中，就需要确保对集合的访问是同步的。为了保证在多线程环境下迭代的安全，可以通过一个互斥对象来同步对集合的访问。当进行迭代时，可以阻塞其他对象对该集合的访问，或者通过克隆集合的方式来阻塞其他对象对该集合的访问。如果设计合理，可以为客户提供线程安全的迭代器代码。

第29章

访问者（Visitor）模式

要对已存在的类层次进行扩展，通常的做法是为需要的行为增加方法。然而，有时需要增加的行为与现有对象模型并不一致，又或者无法修改现有代码。在这种情况下，不更改类的层次结构，就无法扩展该层次结构的行为。倘若类层次结构的开发者运用了访问者模式，就可以支持其他开发人员扩展该类层次结构的行为。

和解释器模式一样，访问者模式通常是基于合成模式的。你可能需要复习一下合成模式，因为我们在整章中使用这一模式。

访问者模式的意图是在不改变类层次结构的前提下，对该层次结构进行扩展。

访问者模式机制

在开发类层次结构时，访问者模式的运用为代码的扩展开启了一扇大门，即使后来的开发人员无权访问源代码，仍然可以对类的层次结构进行各种扩展。访问者模式的机制如下：

- 为类层次结构中的部分或全部类增加 `accept()` 方法。该方法的每个实现都需要接收一个参数，参数类型为自定义的接口。
- 创建定义了一组操作的接口，这些操作通常都会公用一个名字，如 `visit`，每个操作所需的参数类型并不相同。只要类层次结构中的类需要扩展，就可以为其声明这样的操作。

图 29.1 展示了这样一个类图，它修改了 **MachineComponent** 的类层次，使其支持访问者模式。

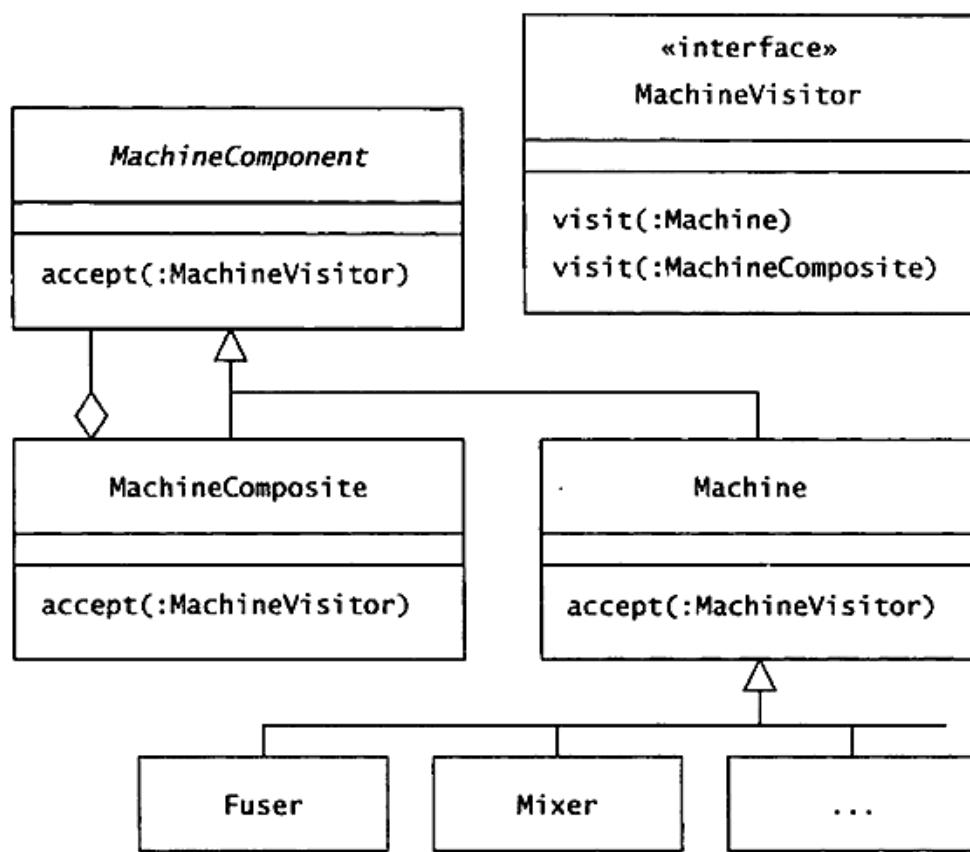


图 29.1 为了让 **MachineComponent** 层次结构支持访问者模式，在该图中增加了 **accept()**方法和 **MachineVisitor** 接口

图 29.1 并未解释访问者模式是如何工作的，在本章的下一小节会详细讲解。该图简单地展示了使用访问者模式的基本原理。

请注意，在 **MachineComponent** 的类图中，并非所有的类都实现了 **accept()**方法。访问者模式并不要求层次结构中的每个类都实现自己的 **accept()**方法。我们将会看到，实现 **accept()**方法的每个类，都会作为 **visit()**方法的参数被传递给 **visitor** 接口。

MachineComponent 类的 **accept()**方法是一个抽象方法，每个子类对该方法的实现几乎完全相同：

```
public void accept(MachineVisitor v) {  
    v.visit(this);  
}
```

你可能会想，既然该方法在 **Machine** 和 **MachineComposite** 类中完全相同，就可以将其提

取到抽象的 `MachineComponent` 类中。然而，编译器却会认为这两份“完全相同的代码”其实并不相同。

挑战 29.1

Java 编译器会认为 `Machine` 和 `MachineComposite` 类的 `accept()` 方法有何不同？（不要试图忽略这一点，这是理解访问者模式的关键。）

答案参见第 363 页

`MachineVisitor` 接口要求实现者定义访问 `machines` 和 `machine composites` 的方法：

```
package com.oozinoz.machine;
public interface MachineVisitor {
    void visit(Machine m);
    void visit(MachineComposite mc);
}
```

`MachineComponent` 类的 `accept()` 方法与 `MachineVisitor` 接口的结合，使得开发者能够为层次结构提供新的操作。

常规的访问者模式

假设你就职于 Oozinoz 公司在爱尔兰都柏林新成立的工厂。这里的开发人员已经为新工厂的机器组合建立了对象模型，并且可以通过 `oozinozFactory` 类的静态方法 `dublin()` 访问该模型。为了能正常显示该合成模型，开发人员创建了一个 `MachineTreeModel` 类，它可以为 `JTree` 对象提供需要的信息。（`MachineTreeModel` 类的代码在 `com.oozinoz.dublin` 包中。）

为了显示工厂的机器，需要从工厂组合中创建一个 `MachineTreeModel` 类的实例，并将其包装为 Swing 组件：

```
package app.visitor;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import com.oozinoz.machine.OozinozFactory;
```

```
import com.oozinoz.ui.SwingFacade;

public class ShowMachineTreeModel {
    public ShowMachineTreeModel() {
        MachineTreeModel model = new MachineTreeModel(
            OozinozFactory.dublin());
        JTree tree = new JTree(model);
        tree.setFont(SwingFacade.getStandardFont());
        SwingFacade.launch(
            new JScrollPane(tree),
            " A New Oozinoz Factory");
    }

    public static void main(String[] args) {
        new ShowMachineTreeModel();
    }
}
```

运行这一程序，显示结果如图 29.2 所示。

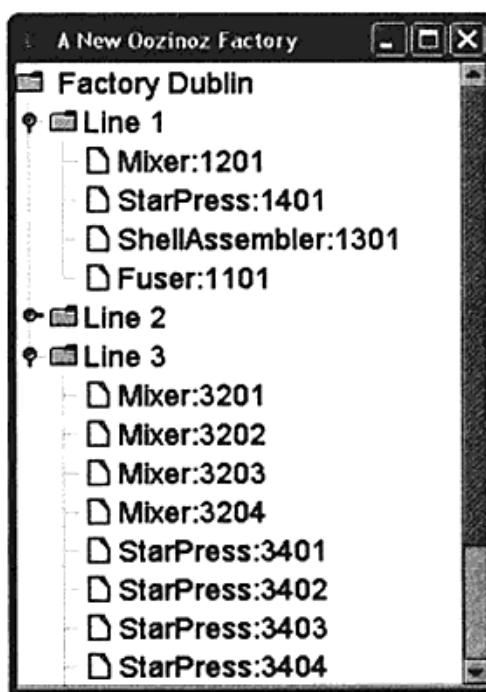


图 29.2 该 GUI 程序展示了在都柏林新工厂中的机器组合

机器组合有许多有用的行为，假设需要在工厂模型中找到一台指定的机器，在不改变 `MachineComponent` 层次结构的前提下增加这项功能，就可以创建一个如图 29.3 所示的 `FindVisitor` 类。

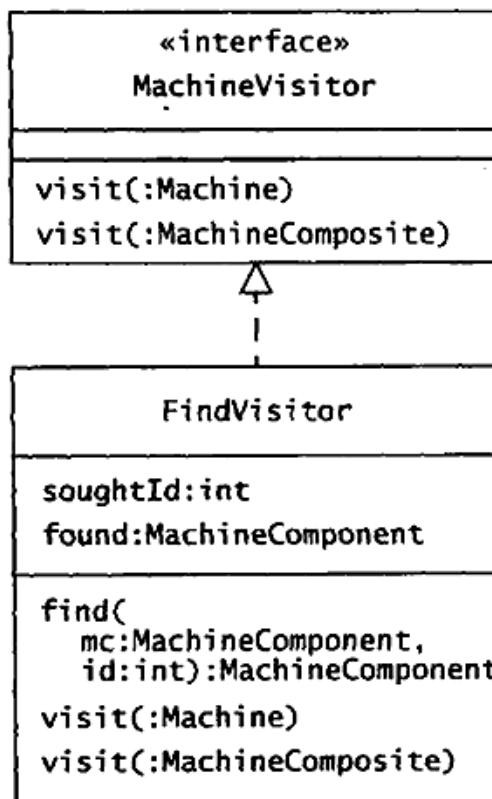


图 29.3 FindVisitor 类向 MachineComponent 层次结构添加了 find()方法

visit()方法不返回任何对象，因此 FindVisitor 类在 found 实例变量中记录搜索状态。

```

package app.visitor;

import com.oozinoz.machine.*;
import java.util.*;

public class FindVisitor implements MachineVisitor {
    private int soughtId;
    private MachineComponent found;

    public MachineComponent find(
        MachineComponent mc, int id) {
        found = null;
        soughtId = id;
        mc.accept(this);
        return found;
    }

    public void visit(Machine m) {
        if (found == null &&m.getId() == soughtId)
    }
}
  
```

```
        found = m;
    }

    public void visit(MachineComposite mc) {
        if (found == null && mc.getId() == soughtId) {
            found = mc;
            return;
        }
        Iterator iter = mc.getComponents().iterator();
        while (found == null && iter.hasNext())
            ((MachineComponent) iter.next()).accept(this);
    }
}
```

`visit()`方法会检查 `found` 变量，在找到目标组件后，树的遍历就会结束。

挑战 29.2

写一个程序用来查找并打印出 `OozinozFactory.dublin()` 方法所返回的 `MachineComponent` 实例中的 `StarPress:3404` 对象。

答案参见第 363 页

`find()`方法并不关心接收到的 `MachineComponent` 对象究竟是 `Machine` 类型，还是 `MachineComposite` 类型。该方法只是简单地调用 `accept()`，而 `accept()`方法则负责轮流调用 `visit()`方法。

注意，`visit(:MachineComposite)`方法内的循环，同样不关心它的子组件的实例是 `Machine` 类型，还是 `MachineComposite` 类型。`visit()`方法简单地调用每个组件的 `accept()` 操作。究竟调用会执行哪个方法，取决于子组件的类型。图 29.4 展示了方法调用的典型顺序。

执行 `visit(:MachineComposite)`方法时，它会调用每个合成子对象的 `accept()`方法。子对象会通过调用 `visitor`对象的 `visit()`方法来回应。图 29.4 展示了从 `visitor` 对象到接收 `accept()` 调用的对象，以及各自接收对象之间的来回调用。这种技术被称为双重委派(double dispatch)，用于确保正确的 `visitor`类的 `visit()`方法被调用。

访问者模式中的双重委派使你可以创建访问类，这些访问类包含的方法可以指向被访问的类结构中不同的类型。

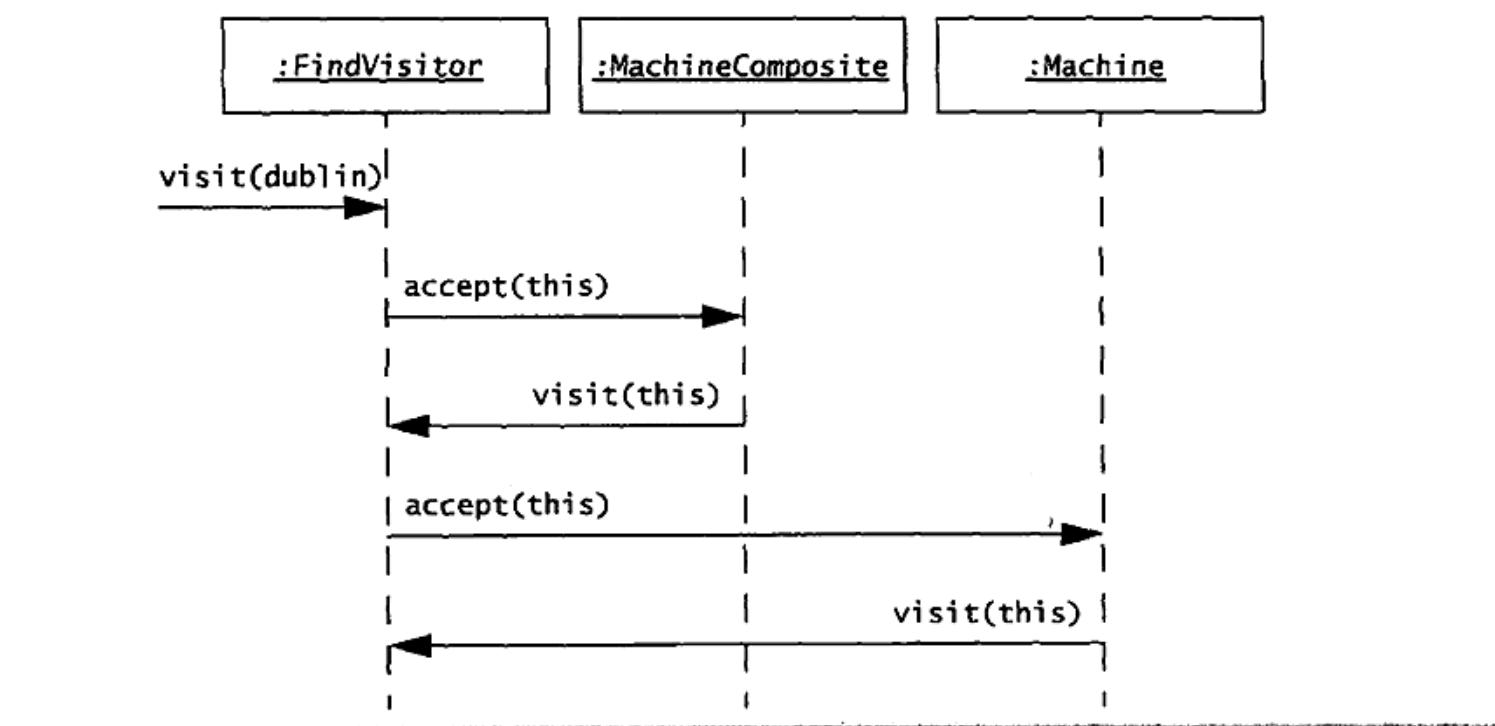


图 29.4 `FindVisitor` 对象调用 `accept()`方法，以决定执行哪个 `visit()`方法

倘若你能够控制源代码，通过运用访问者模式，几乎可以为源代码增加任意的行为。例如，可以增加一个访问者，它能够查找机器组件中的所有机器（叶子节点）：

```

package app.visitor;
import com.oozinoz.machine.*;
import java.util.*;

public class RakeVisitor implements MachineVisitor {
    private Set leaves;

    public Set getLeaves(MachineComponent mc) {
        leaves = new HashSet();
        mc.accept(this);
        return leaves;
    }

    public void visit(Machine m) {
        // 挑战!
    }

    public void visit(MachineComposite mc) {
        // 挑战!
    }
}
  
```

挑战 29.3

完成 RakeVisitor 类的代码，使其可以收集机器组件中的所有叶子节点。

答案参见第 364 页

这段简短的程序可以找到机器组件的叶子节点，并将其打印出来：

```
package app.visitor;

import com.oozinoz.machine.*;
import java.io.*;
import com.oozinoz.filter.WrapFilter;

public class ShowRakeVisitor {
    public static void main(String[] args)
        throws IOException {
        MachineComponent f = OozinozFactory.dublin();
        Writer out = new PrintWriter(System.out);
        out = new WrapFilter(new BufferedWriter(out), 60);
        out.write(
            new RakeVisitor().getLeaves(f).toString());
        out.close();
    }
}
```

这段程序使用逗号作为过滤器，产生如下输出：

```
[StarPress:3401, Fuser:3102, StarPress:3402, Mixer:3202,
Fuser:3101, StarPress:3403, ShellAssembler:1301,
ShellAssembler:2301, Mixer:1201, StarPress:2401, Mixer:3204,
Mixer:3201, Fuser:1101, Fuser:2101, ShellAssembler:3301,
ShellAssembler:3302, StarPress:1401, Mixer:3203, Mixer:2202,
StarPress:3404, Mixer:2201, StarPress:2402]
```

FindVisitor 类和 Rakevisitor 类都向 MachineComponent 类结构中添加了新的行为。看起来，这些类似乎工作正常。然而，在编写访问者类时可能存在障碍：这些访问者需要知道你想要扩展的类层次结构。一旦类层次结构发生变化，就可能破坏这些访问者类；但往往在一开

始时，我们对类结构机制的了解并非一清二楚，尤其是当需要访问的组合结构存在环时，更需要额外的处理。

Visitor 环

Oozinoz 公司使用 `ProcessComponent` 层次结构对工作流建模，该层级结构是另一种合成结构，并可以让其支持访问者模式而获益。与机器组合不同，它天生需要支持包含环结构的工作流，访问者必须保证在遍历过程组件时，不会陷入死循环。图 29.5 展示了 `ProcessComponent` 的层次结构。

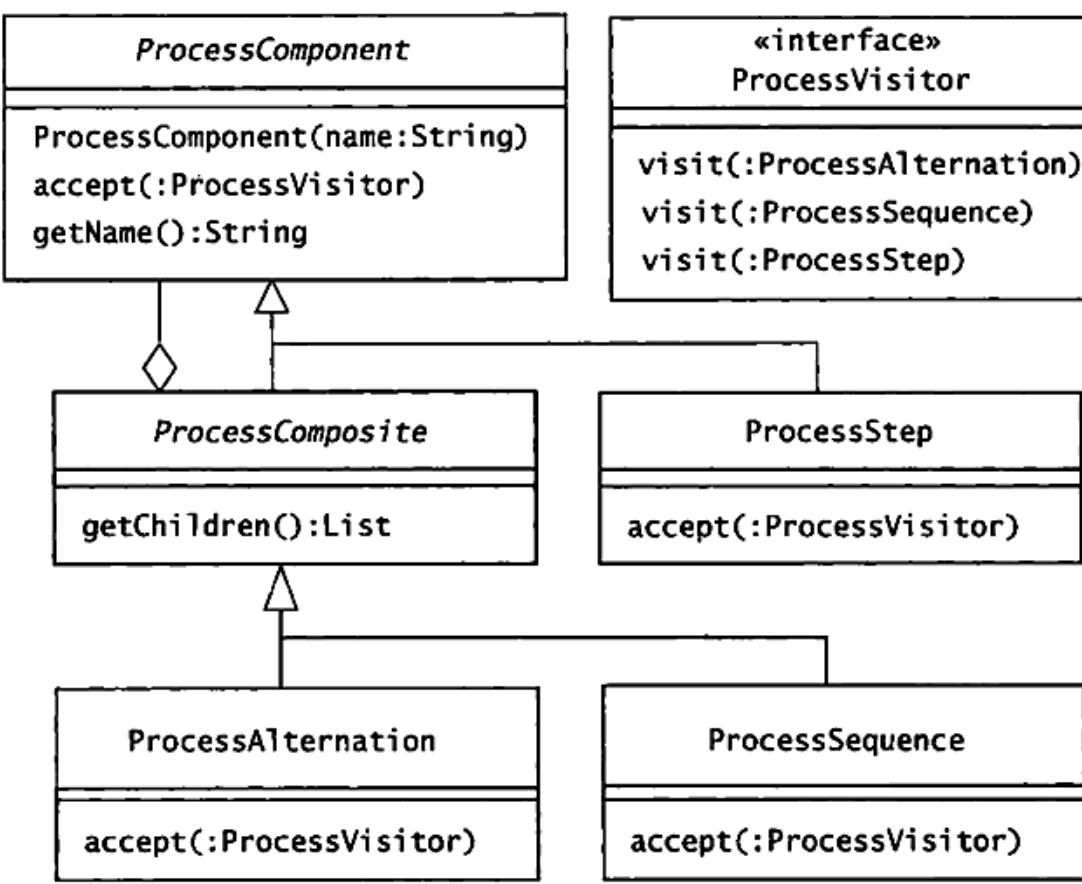


图 29.5 和 `MachineComponent` 层次结构一样，`ProcessComponent` 的层次结构可以支持访问者模式

假设希望打印出来的过程组件具有缩进的格式。在第 28 章的迭代器模式中，使用了迭代器来打印出一个工作流的步骤。打印输出如下所示：

```

Make an aerial shell
    Build inner shell
  
```

```
Inspect
Rework inner shell, or complete shell
    Rework
        Disassemble
    Finish: Attach lift, insert fusing, wrap
```

若要重新做一个焰火弹，需要先拆除，然后再重新组装。“拆除”之后的步骤是“制作焰火弹”。然而，输出的结果并未体现出这一步，因为迭代器发现该步骤在之前已经出现过一次。如果输出的步骤名称能够展现当前工作的步骤形成了一个环，将更有意义。若能标示出那些交替出现而非顺序出现的组件，同样很有帮助。

为了支持对工作流程的直观输出，可以创建一个访问者类，它会初始化一个 `StringBuilder` 对象，并把访问者类访问工作组件的过程记录在该对象中。为了标示出交替出现的组合步骤，访问者可以在这些步骤的名称前标记一个问号（?）。为了标示已经出现过的步骤，访问者可以在该步骤的名称后面加上省略号（…）。经过这些转换，焰火弹的生产过程会产生如下输出：

```
Make an aerial shell
Build inner shell
Inspect
?Rework inner shell, or complete shell
    Rework
        Disassemble
        Make an aerial shell...
Finish: Attach lift, insert fusing, wrap
```

工作流组件的访问者类需要处理环的情况，通过使用 `Set` 对象来跟踪访问者业已访问的节点，可以轻而易举地实现。该类的代码如下：

```
package app.visitor;
import java.util.List;
import java.util.HashSet;
import java.util.Set;
import com.oozinoz.process.*;

public class PrettyVisitor implements ProcessVisitor {
    public static final String INDENT_STRING = " ";
    private StringBuffer buf;
    private int depth;
```

```
private Set visited;
public StringBuffer getPretty(ProcessComponent pc) {
    buf = new StringBuffer();
    visited = new HashSet();
    depth = 0;
    pc.accept(this);
    return buf;
}

protected void printIndentedString(String s) {
    for (int i = 0; i < depth; i++)
        buf.append(INDENT_STRING);
    buf.append(s);
    buf.append("\n");
}
// ... visit() 方法 ...
}
```

该类使用了 `getPretty()` 方法来初始化实例变量，并实现访问算法。该算法使用 `printIndentedString()` 方法来处理缩进。当访问 `ProcessStep` 对象时，代码只是打印出该步骤的名字：

```
public void visit(ProcessStep s) {
    printIndentedString(s.getName());
}
```

从图 29.5 中你可能会注意到，`ProcessComposite` 类没有实现 `accept()` 方法，但它的子类却实现了。访问交替执行的流程与顺序执行的流程在逻辑上几乎完全相同，如下所示：

```
public void visit(ProcessAlternation a) {
    visitComposite("?", a);
}

public void visit(ProcessSequence s) {
    visitComposite("", s);
}
protected void visitComposite(
    String prefix, ProcessComposite c) {
    if (visited.contains(c)) {
        printIndentedString(prefix + c.getName() + "...");
```

```
    } else {
        visited.add(c);
        printIndentedString(prefix + c.getName());
        depth++;

        List children = c.getChildren();
        for (int i = 0; i < children.size(); i++) {
            ProcessComponent child =
                (ProcessComponent) children.get(i);
            child.accept(this);
        }

        depth--;
    }
}
```

二者的区别在于，交替节点会打印问号作为前缀。无论是哪一种类型，只要当前访问的节点曾经被访问过，就打印出该节点的名称和省略号。否则，就将该节点添加到已访问的节点集合中，并打印出前缀——问号或者什么都没有，然后调用子节点的 accept() 方法。对于一个典型的访问者模式而言，代码使用多态来决定子节点是 ProcessStep、ProcessAlternation 还是 ProcessSequence 类的实例。

如下的小程序可以很好地打印出工作流：

```
package app.visitor;

import com.oozinoz.process.ProcessComponent;
import com.oozinoz.process.ShellProcess;

public class ShowPrettyVisitor {
    public static void main(String[] args) {
        ProcessComponent p = ShellProcess.make();
        PrettyVisitor v = new PrettyVisitor();
        System.out.println(v.getPretty(p));
    }
}
```

运行该程序，产生如下输出：

```
Make an aerial shell
```

```
Build inner shell
Inspect
?Rework inner shell, or complete shell
    Rework
        Disassemble
        Make an aerial shell...
Finish: Attach lift, insert fusing, wrap
```

比起简单地迭代整个工作模型而言，这种方式的输出结果所能涵盖的信息更多。如果输出中出现问号，则表示该组合步骤是交替出现的。一旦节点出现两次，后面就会加上省略号，这比直接省略该重复步骤表达得更为清楚。

`ProcessComponent` 类结构的开发者通过定义 `ProcessVisitor` 接口以及 `accept()` 方法，使得该层级结构支持访问者模式。这些开发者需要竭力避免在遍历工作流时出现死循环。如 `Prettyvisitor` 类所示，访问者类的开发者需要识别工作流组件中可能存在的环。倘若在开发 `ProcessComponent` 时，能够在一定程度上支持对环结构的管理，并将其作为支持访问者模式的一部分，就可以规避这些错误。

挑战 29.4

`ProcessComponent` 的开发者应该如何设计，才能保证该组件在支持访问者模式的同时支持对环的管理？

答案参见第 365 页

访问者模式的危机

一直以来，访问者模式颇具争议。一些开发者反对使用该模式，而另一些人则拥护该模式，并竭力寻找强化该模式的方法，尽管这些方法通常都会增加系统的复杂性。事实上，诸多设计问题都与使用访问者模式有关。

本章给出的例子同样彰显了访问者模式的脆弱之处。例如，在 `MachineComponent` 的层次结构中，开发者区分了 `Machine` 节点和 `Machinecomposite` 节点，但却没有区分 `Machine` 子

类。如果需要区分访问类中不同机器的类型，就需要使用类型检查或其他技术，以辨别 `visit()` 方法接收的到底是哪种机器类型。或许你会认为，类层次的开发者应该在访问者接口中，提供一个 `visit(:Machine)` 方法，以包含所有的机器类型。然而，随时都可能增加新的机器类型，因而这种设计并不健壮。

访问者模式是否是一个好的选择，取决于系统变化的特征：如果层次结构相对稳定，变化的总是行为，那么就是好的选择。如果行为相对稳定，但是结构总是变化，选择它就不恰当了，因为你需要更新现有的访问者类，使得它们可以支持新的节点类型。

`ProcessComponent` 层次结构中还存在另一种脆弱性。该结构的开发者知道工作流模型中环的危险性。但是，他们应该如何将这种顾虑告知访问者类的开发者呢？

这些问题可能暴露出访问者模式的一个根本问题：当需要扩展层次结构的行为时，通常需要一些层次设计的专业知识。如果缺乏这些专业知识，就可能陷入陷阱，例如无法避免在工作流中出现死循环。即使你拥有层次结构的专业知识，也可能会引入某种危险的依赖关系。一旦层次结构发生变化，这些依赖关系就可能会失效。这种专业知识以及对代码的控制使得访问者模式很难被应用。

计算机语言解析器是一个经典的运用访问者模式的范例，而且不会产生后续问题。解析器的开发者通常会开发一个抽象语法树，它可以根据语言的语法规则组织输入文字。如果希望为这棵树开发多种行为，则访问者模式就是最佳选择。在这个例子中，被访问的层次结构通常只有很少的行为，甚至没有行为。访问者类有责任处理所有设计好的行为，而避免像本章中的例子那样分离这些职责。

和任何模式一样，访问者模式不是必需的；如果需要使用该模式，就应该物尽其用。对于访问者模式而言，通常还有其他的替换方案提供更为健壮的设计。

挑战 29.5

请列出两种可替换的方案，用来替换 Oozinoz 公司的机器层次结构和工作流结构。

答案参见第 365 页

因此，若要判断是否应该使用访问者模式，最好满足如下条件：

- 节点类型的集合是稳定的。
- 共同发生的变化是为不同的节点添加新的功能。
- 新功能必须适用于所有节点类型。

小结

访问者模式使你可以在不改变类层次结构的前提下，为该结构增加新的行为。该模式的机制包括为访问者类定义一个接口，为层次关系中的所有访问者增加一个 `accept()` 方法。`accept()` 方法将使用双重委派技术，将其调用委派给访问者。类层次结构中的对象可以根据其类型调用合适的 `visit()` 方法。

访问者类的开发者必须了解被访问者层次结构设计的微妙之处。特别的，访问者类需要注意被访问对象模型中的环结构。归咎于此，一些开发者会避免使用访问者模式，转而运用一些替换方式。是否应该运用访问者模式，通常取决于你的设计理念、团队情况以及具体的应用。

附录 A

指南

如果你已经阅读完本书所有的章节，请允许我们向你道一声祝贺！如果你还完成了本书给出的所有挑战，那么我们更要向你表示致敬！劳而有获，我们相信你已对设计模式有了更深入的理解。那么，到了这里，你还能得到什么呢？

从本书学到更多

如果没有完成本书所有的挑战练习，不用担心，你并不会是这唯一的一个。我们都很忙，在经过一番思考后，总是会忍不住去翻阅附录给出的答案。这很正常，即使如此，你仍有可能成为一名出色的开发人员。回到练习部分，再一次仔细思考，只有在你认为获得了正确答案之后，或者在你经历了一番冥思苦想仍然没有寻找到答案之后，才去求助于本书给出的参考答案。不要自欺欺人，总是认为以后还有时间再来思考。通过这些挑战来实践你的模式知识，才能具有足够的信心将这些模式运用到你的日常工作中。

除了完成本书给出的挑战之外，我们建议你到 www.oozinoz.com 去下载本书的代码，在自己机器上运行一遍，检查一下书中范例的结果。让代码跑起来要比纸上谈兵更能提高我们对设计模式的信心。你还可以为自己提出新的挑战。或许你希望通过一种新的方法组织装饰过滤器，又或者实现一个数据适配器，以展现你所熟知领域的数据。

当你已经熟练掌握了设计模式后，就可以开始检查一下你对设计模式经典范例的理解。你还可以尝试在自己的代码中合理地运用设计模式。

理解经典

设计模式通常能使我们的设计更健壮。在 Java 类库中就大量运用了设计模式。如果你能够指出代码中运用的设计模式，就说明你已经掌握了这种设计模式，并能与其他理解设计模式的人沟通。例如，如果开发人员理解装饰器模式的原理，就能够很好地解释 Java 流为何是装饰器。

下面的测试可以考查你对 Java 类库中各种典型设计模式范例的理解程度。

- GUI 控件是如何运用观察者模式的？
- 菜单为何常常运用命令模式？
- 为什么说驱动器是桥接模式的最好范例？每个特定的驱动器，是否都是适配器模式的范例？
- Java 流使用了装饰器模式，这意味着什么？
- 为何代理模式是 RMI 设计的基础？
- 如果排序算法是模板方法模式的一个典型例子，那么算法的哪一步没有被规定实现？

如果不用查阅本书，你就能准确地回答这些问题，就说明你很好地达到了理解设计模式的目标。写下答案，并与同行进行讨论，会是一种很好的锻炼。

在你的代码中运用模式

学习设计模式的目的是成为一名更优秀的软件开发人员。我们可以通过代码库来考虑如何运用设计模式。这里有两种选择：对新加的代码运用设计模式；或者通过对已有代码进行重构来运用设计模式。如果现有代码异常复杂且难以维护，就可以通过重构现有代码并运用设计模式对其进行改善。不过，在重构之前，需要事先确定这一活动能够为客户带来益处；其次还需要为这些需要重构的代码创建自动化测试。

倘若你对设计模式已有较深入的理解，并准备在适合的场景运用这些设计模式。那么，你能寻找到运用的时机吗？我们常常可以发现一些运用设计模式的时机。如果你正在寻找运用设计模式的时机，可以考虑如下场景。

- 代码库中用于维护系统状态或用户状态的代码是否复杂？如果复杂，可以考虑运用状态模式对其进行改进。
- 代码是否将策略的选择与执行混杂在一起？如果是，可以运用策略模式改进这段代码。
- 客户或分析人员提供的流程图是否难以理解，无法用代码实现？如果是，就可以考虑运用解释器模式，把流程图中的每个节点表示为解释器类层次结构中的一个实例，这样可以直接将流程图转换为代码。
- 你所实现的组合结构是否不允许子节点自身又是一个组合结构？此时可以运用合成模式来改进代码。
- 你的对象模型中是否出现关系一致性的错误？如果有，可以运用调停者模式对对象关系进行建模，以避免出现关系一致性错误。
- 代码是包含客户端正在使用的服务，它将决定实例化哪个类？此时，可以运用工厂方法模式改进和简化这样的代码。

通过学习设计模式，我们就能够积累有关软件设计思想的丰富词汇。如果你正在寻找运用它们的时机，可能无须花费很长的时间就能找到这样的机会。但是切忌不要设计过度。

持续学习

无论出于什么目的，你规划好时间，动力十足而又踌躇满志地读完了本书。现在，我们要给你的建议就是“持续学习！”可以认真考虑一下每周花在工作上的时间，抽出 5 个小时的时间进行学习。离开办公室，去阅读一些书籍和杂志，或者编写一些自己感兴趣的软件，并将这种方式养成习惯。如果能够认真地对待这种工作方式，你就会变得更加优秀，也能够激发你对工作的热情。

假设你有足够的时间，就需要确定学习的方向。在更深入地学习设计模式之前，你需要确认是否已经掌握了学习模式的基础知识。如果没有读过 *Java™ Programming Language*（由 Arnold 和 Gosling 在 1998 年编写），也没有读过 *The Unified Modeling Language User Guide*（由 Booch、

Rumbaugh 和 Jacobsen 在 1999 年编写），我建议你先阅读这两本书。如果希望了解更多模式的内容，可以有许多选择。如果希望了解运用设计模式的真实案例，可以阅读 *Pattern Hatching: Patterns Applied*（由 Vlissides 在 1998 年编写）。如果能够了解与并发模式有关的内容，会更有帮助。并发程序设计是软件开发中一个重要却常被忽略的内容，如果期望了解这方面的知识，建议你阅读 *Concurrent Programming in Java™*（由 Lea 在 2000 年编写）这本书。

关于重构的基础知识以及重构目录，可以阅读 *Refactoring*（由 Fowler 等人在 1999 年编写）。*Refactoring Workbook*（由 Wake 在 2004 年编写）一书则以与本书相似的格式介绍了重构的实践。一旦理解了重构，又希望了解如何从重构到模式，可以阅读 *Refactoring to Patterns*（由 Kerievsky 在 2005 年编写）。

学习的方向有很多，然而无论学习什么内容，关键还是要坚持下去。无论是与工作内容有关，还是自己感兴趣的领域，都要坚持学习。想到就能做到，要想成为一名优秀的开发人员，那就坚持不断地学习吧！

Steve.Metsker@acm.org

William.Wake@acm.org

附录 B

答案

第 2 章 接口型模式介绍

挑战 2.1 的答案（第 9 页）

一个没有抽象方法的抽象类从功能来看与接口相似。然而，需要注意如下区别：

- 一个类可以实现多个接口，但却只能继承最多一个抽象类。
- 抽象类可以包含具体方法；接口的所有方法都是抽象的。
- 抽象类可以声明和使用字段；接口则不能，但可以创建静态的 `final` 常量。
- 抽象类中的方法可以是 `public`、`protected`、`private` 或者默认的 `package`；接口的方法都是 `public`。
- 抽象类可以定义构造函数；接口不能。

挑战 2.2 的答案（第 9 页）

A. 正确。接口方法总是抽象的，不管你是否对此进行声明。

B. 正确。接口方法总是公开的，不管你是否对此进行声明。

- C. 错误。接口的可见性会被限制在它所在的包中。对于本例，接口应该被标记为 `public`，这样，`com.oozinoz.simulation` 包外的类才能够访问它。
- D. 正确。例如，`List` 和 `Set` 接口都继承自 `java.util` 的 `Collection` 接口。
- E. 错误。没有方法的接口称之为标记接口（marker interface）。有时候，一个方法处于类继承层次的高处，例如 `Object.clone()`，但它却不适用于所有子类。如果希望子类在此样式下能够有选择地实现，就可以创建一个标记接口。通过声明对 `Cloneable` 标记接口的实现，就能够要求子类也实现 `Object` 的 `clone()` 方法。
- F. 错误。接口虽然能够声明静态的 `static` 和 `final` 字段作为常量，但不能声明实例字段。
- G. 错误。虽然是个不错的主意，但 Java 接口却不能这样做，这会使得实现类必须提供一个特定的构造函数。

挑战 2.3 的答案（第 10 页）

一个例子被当做一个类被注册成为事件的监听器时，这些监听器类会收到它们关心的通知，而不是调用者。例如，我们需要在触发 `MouseListener.mouseDragged()` 方法时采取某个动作，但对于同一个监听器而言，`MouseListener.mouseMoved()` 方法却是一个空的实现。

第 3 章 适配器（Adapter）模式

挑战 3.1 的答案（第 15 页）

解决方案如图 B.1 所示。

`OozinozRocket` 类的实例具有 `PhysicalRocket` 和 `RocketSim` 对象的功能。适配器模式能够将已经实现的方法适配为客户端需要的方法。

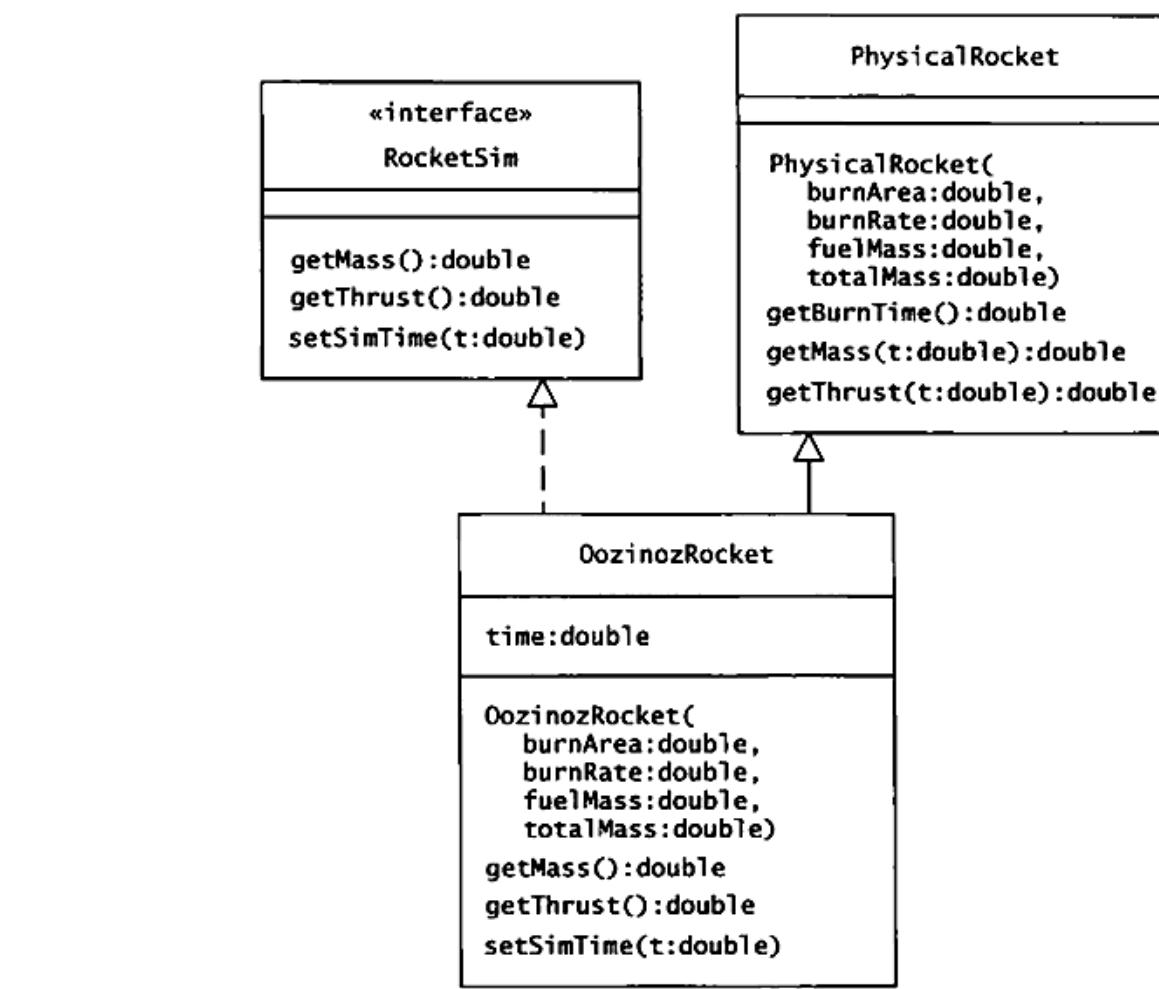


图 B.1 OozinozRocket 类将 PhysicalRocket 类适配为满足 RocketSim 接口的需要

挑战 3.2 的答案（第 16 页）

完整的类代码如下所示：

```
package com.oozinoz.firework;

import com.oozinoz.simulation.*;

public class OozinozRocket
    extends PhysicalRocket implements RocketSim {
    private double time;
    public OozinozRocket(
        double burnArea,
        double burnRate,
        double fuelMass,
        double totalMass) {
        super(burnArea, burnRate, fuelMass, totalMass);
    }
}
```

```

public double getMass() {
    return getMass(time);
}

public double getThrust() {
    return getThrust(time);
}

public void setSimTime(double time) {
    this.time = time;
}
}

```

你可以在本书提供的源代码 com.oozinoz.firework 包中找到这个类。

挑战 3.3 的答案 (第 19 页)

图 B.2 给出了解决方案。

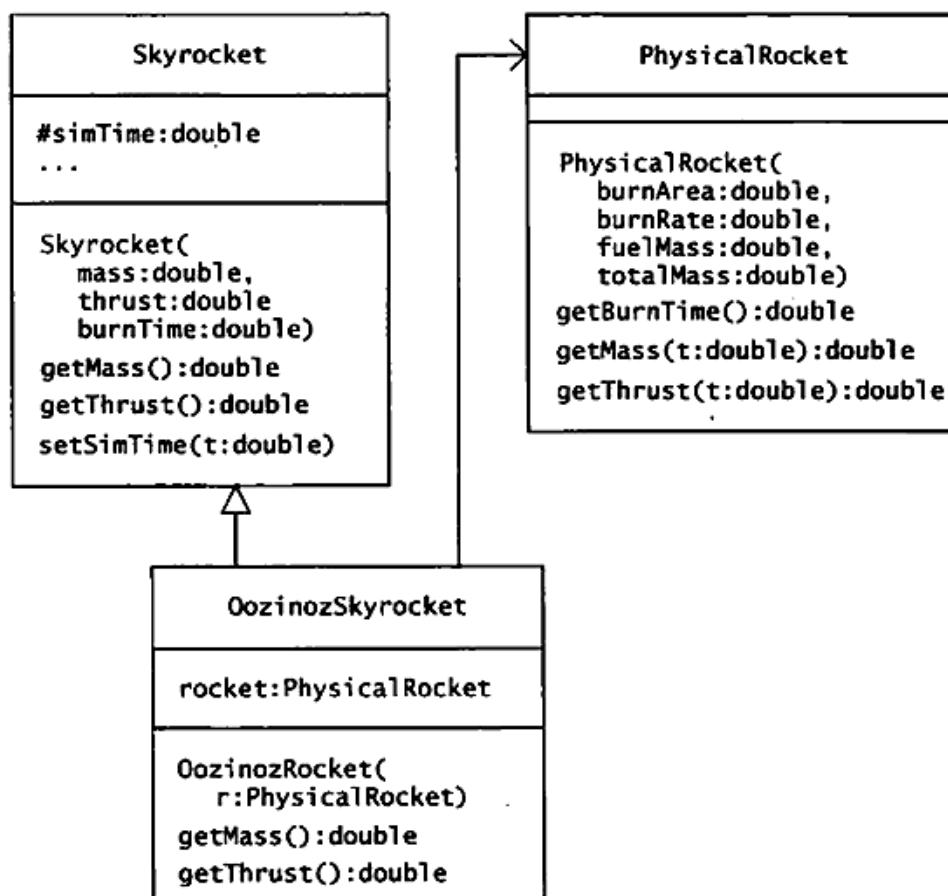


图 B.2 OozinozSkyrocket 对象是一个 Skyrocket 对象，但它的实现却是将调用转给 PhysicalRocket 对象

`OozinozSkyrocket` 类是一个对象适配器，它继承自 `Skyrocket`，因此它的实例拥有 `Skyrocket` 对象的功能。

挑战 3.4 的答案（第 20 页）

对象适配器使得 `OozinozSkyrocket` 类比采用类适配器更加脆弱的原因为：

- 没有 `OozinozSkyrocket` 类所提供的接口规范。由于 `Skyrocket` 的变化，可能在运行时出现编译时无法检测到的问题。
- `OozinozSkyrocket` 需要借助于访问其超类的 `simTime` 变量，但我们却无法保证该变量总是被声明为 `protected`，也不能保证处于 `Skyrocket` 类中的这一字段符合子类的意图。（我们不能期望提供者不会修改我们所依赖的 `Skyrocket` 代码，换言之，很难约束和控制它们所要做的事情。）

挑战 3.5 的答案（第 23 页）

代码实现大约如下所示：

```
package app.adapter;
import javax.swing.table.*;
import com.oozinoz.firework.Rocket;

public class RocketTable extends AbstractTableModel {
    protected Rocket[] rockets;
    protected String[] columnNames = new String[] {
        "Name", "Price", "Apogee" };

    public RocketTable(Rocket[] rockets) {
        this.rockets = rockets;
    }

    public int getColumnCount() {
        return columnNames.length;
    }

    public String getColumnName(int i) {
        return columnNames[i];
    }
}
```

```
public int getRowCount() {
    return rockets.length;
}

public Object getValueAt(int row, int col) {
    switch (col) {
        case 0:
            return rockets[row].getName();
        case 1:
            return rockets[row].getPrice();
        case 2:
            return new Double(rockets[row].getApogee());
        default:
            return null;
    }
}
```

`TableModel` 接口很好地展示了如何适应未来可能发生的变化。该接口以及实现了部分功能的 `AbstractTableModel`，减少了在标准 GUI 表控件中显示领域对象的实现工作。在接口的支持下，这种解决方案很容易进行适配，以应对未来的变化。

挑战 3.6 的答案（第 25 页）

- 一种观点：当用户单击鼠标时，我需要将 Swing 调用的结果转换或适配给对应的动作。换言之，当需要将 GUI 事件适配给应用程序的接口时，我使用了 Swing 的适配器类。我将一个接口转换为另一个，从而实现了适配器模式的意图。
- 一种争论：Swing 中的“适配器”类是桩（Stub），它们并没有真正转换或适配。在定义这些类的子类时，重写了你需要的方法。这些重写的方法和类才是适配器模式的例子。如果将“Adapter”改名为 `DefaultMouseListener`，就不会出现这样的争论了。

第 4 章 外观（Facade）模式

挑战 4.1 的答案（第 28 页）

示例与外观的区别如下：

- 示例通常是一个单独运行的应用程序，而外观不是。
- 示例通常包含了样本数据，而外观则没有。
- 外观通常是可配置的，示例不是。
- 外观的意图是为了重用，示例不是。
- 外观用在产品代码中，示例不是。

挑战 4.2 的答案（第 29 页）

`JOptionPane` 类是 Java 类库中少有的几个运用了外观模式的类。它属于产品代码，可配置，设计的目的是为了重用。除此之外，`JOptionPane` 类通过提供一个简单的接口使得对 `JDialog` 类的使用变得简单，满足外观模式的意图。你可能会认为该类是一个简化了的“子系统”，但事实上，一个独立的 `JDialog` 类并不能认为是子系统。但确切地讲，该类提供的丰富特性体现了外观类的价值。

Sun 公司在 JDK 中提供了多个示例程序。然而，这些类都不是 Java 类库的一部分。也就是说，它们并没有被放在 `java` 包中。外观属于 Java 类库，但示例不是。

`JOptionPane` 有多个静态方法，这使得它成为运用了外观模式的工具类。严格地讲，这样的类并不符合 UML 中工具类的定义，因为 UML 要求工具类只能处理静态方法。

挑战 4.3 的答案（第 29 页）

Java 类库之所以较少运用外观模式，可能存在如下几个合理但观点截然对立的理由：

- 作为 Java 开发者，通常要求对库中的工具做整体的了解。外观模式可能会限制这种运用系统的方式。它们可能会分散开发人员的注意力，并对类库提供的功能产生误解。
- 外观类介于丰富的工具包与特定应用程序之间。为了创建外观类，需要了解它所支持的应用程序类型。然而 Java 类库的用户如此之多，这种预先的支持是不可能的。
- Java 类库提供的外观类过少，这是一个缺陷。应该加入更多的外观类，提供更好的支持。

挑战 4.4 的答案（第 35 页）

最后的结果可能如图 B.3 所示。

注意，`createTitledBorder()` 方法不是静态方法。你给出的解决方案是否将这些方法声明为静态呢？不管是否声明为静态，都要分析其中的原因。

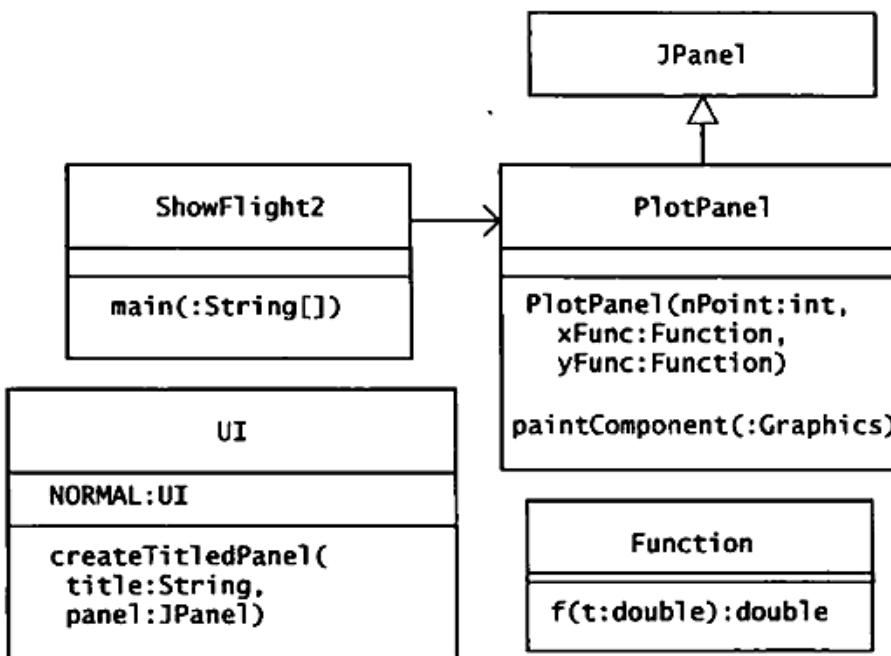


图 B.3 该图展示了将计算飞行路径的应用程序重构为多个类，每个类承担一个职责

本书的代码将 **UI** 方法声明为非静态的，这样 **UI** 的子类就可以重写它们，从而能够为用户界面的构建提供不同的工具实现。为了让标准的用户界面是可用的，设计参考了单例的 **NORMAL** 对象。

如下是与 **UI** 有关的代码：

```

public class UI {
    public static final UI NORMAL = new UI();
    protected Font font =
        new Font("Book Antiqua", Font.PLAIN, 18);

    // 省略部分实现

    public Font getFont() {
        return font;
    }

    public TitledBorder createTitledBorder(String title) {
        TitledBorder border =
            BorderFactory.createTitledBorder(
                BorderFactory.createBevelBorder(
                    BevelBorder.RAISED),
                title,
                TitledBorder.LEFT,
                TitledBorder.TOP);
        border.setTitleColor(Color.black);
    }
}
  
```

```
        border.setTitleFont(getFont());
        return border;
    }

    public JPanel createTitledPanel(
        String title, JPanel in) {
        JPanel out = new JPanel(); out.add(in);
        out.setBorder(createTitledBorder(title));
        return out;
    }
}
```

若要了解 GUI 工具包的更多内容，可以参考本书第 17 章抽象工厂模式中的内容，而第 8 章的单例模式则提供了关于单例的内容。

第 5 章 合成 (Composite) 模式

挑战 5.1 的答案（第 40 页）

设计合成类可用于维护合成对象的集合，使得合成对象既可以支持叶子对象，又能支持组合对象。

换言之，合成模式的设计使得我们能够将一种分组建模为另一种分组的集合。例如，我们可以将用户的系统权限定义为特定权限的集合或权限组。合成模式的另一个例子是对工作进程的定义，可以将其定义为进程步骤的集合以及其他进程。相比于将其定义为叶子对象集合的合成对象，这样的定义更为灵活。

若只支持叶子对象的集合，则合成对象只能有一层的深度。

挑战 5.2（第 41 页）

对于 Machine 类，getMachineCount() 方法的实现如下：

```
public int getMachineCount() {
    return 1;
}
```

类图显示 MachineComposite 使用了一个 List 对象，用以跟踪它的组件。要计算合成对

象中机器的数量，可以这样实现：

```
public int getMachineCount(){
    int count = 0;
    Iterator i = components.iterator();
    while (i.hasNext()) {
        MachineComponent mc = (MachineComponent) i.next();
        count += mc.getMachineCount();
    }
    return count;
}
```

如果使用 JDK 1.5，可以使用对 `for` 循环的扩展。

挑战 5.3 的答案（第 41 页）

答案见表 B.1 所示。

表 B.1 方法的定义

| 方 法 | 类 | 定 义 |
|--------------------------------|-------------------------------|--------------------------------------|
| <code>getMachineCount()</code> | <code>MachineComposite</code> | 返回组合中每个组件的数量和 |
| | <code>Machine</code> | 返回 1 |
| <code>isCompletelyUp()</code> | <code>MachineComposite</code> | 如果所有组件为“completely up”，则返回 true |
| | <code>Machine</code> | 如果机器为“up”，则返回 true |
| <code>stopAll()</code> | <code>MachineComposite</code> | 停止所有的组件 |
| | <code>Machine</code> | 停止机器 |
| <code>getOwners()</code> | <code>MachineComposite</code> | 创建一个 set，而非 list；为所有组件添加负责人，然后返回 set |
| | <code>Machine</code> | 返回机器的负责人 |
| <code>getMaterial()</code> | <code>MachineComposite</code> | 返回组件中所有材料的集合 |
| | <code>Machine</code> | 返回当前的材料 |

挑战 5.4 的答案（第 45 页）

程序输出为：

```
Number of machines: 4
```

事实上，在 `plant` 工厂中，只有三台机器，但是 `mixer` 对象同时被 `plant` 与 `bay` 对象计数。这些对象包含的机器组件列表都引用了 `mixer` 对象。

结果出现了错误。工程师将添加的 `plant` 对象看做是 `bay` 合成对象的组件，调用 `getMachineCount()` 会导致死循环。

挑战 5.5 的答案（第 47 页）

`machineComposite.isTree()` 方法的合理实现为：

```
protected boolean isTree(Set visited) {  
    visited.add(this);  
    Iterator i = components.iterator();  
    while (i.hasNext()) {  
        MachineComponent c = (MachineComponent) i.next();  
        if (visited.contains(c) || !c.isTree(visited))  
            return false;  
    }  
    return true;  
}
```

挑战 5.6 的答案（第 49 页）

答案应显示如图 B.4 中的连线。

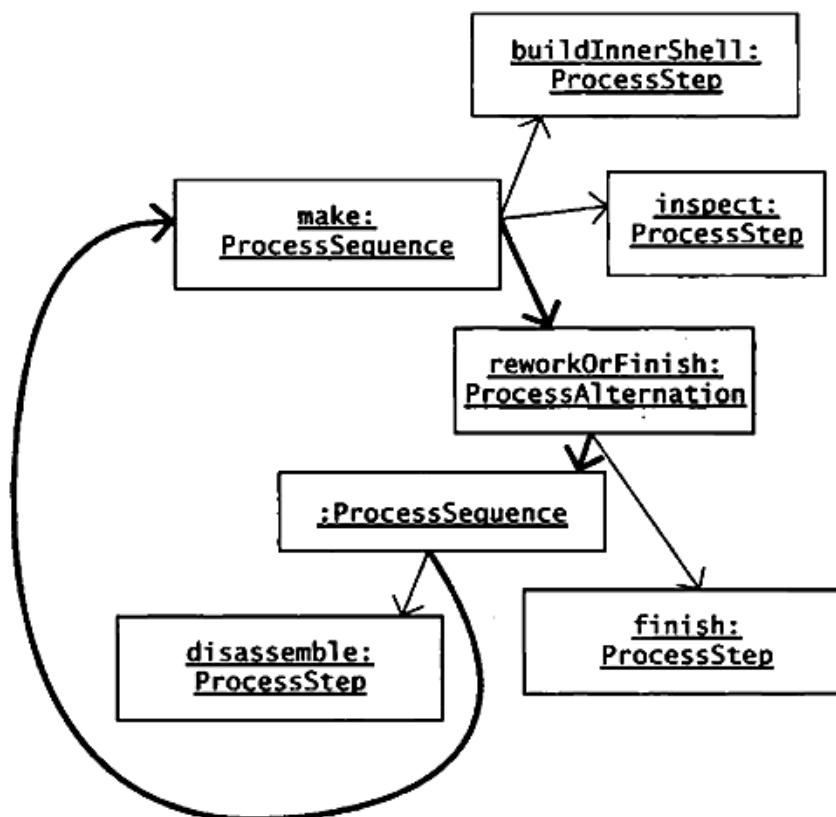


图 B.4 对象图环中的粗线表示这种循环是高空焰火弹生产流程与生俱来的

第 6 章 桥接 (Bridge) 模式

挑战 6.1 的答案 (第 53 页)

若要通过共同的接口控制多种机器，可以使用适配器模式，为每个控制器创建一个适配器类。每个适配器类都可以将标准的接口调用转换为对现有控制器的调用。

挑战 6.2 的答案 (第 54 页)

你所实现的代码可能会是这样：

```
public void shutdown() {
    stopProcess();
    conveyOut();
    stopMachine();
}
```

挑战 6.3 的答案 (第 56 页)

图 B.5 给出了解决方案。

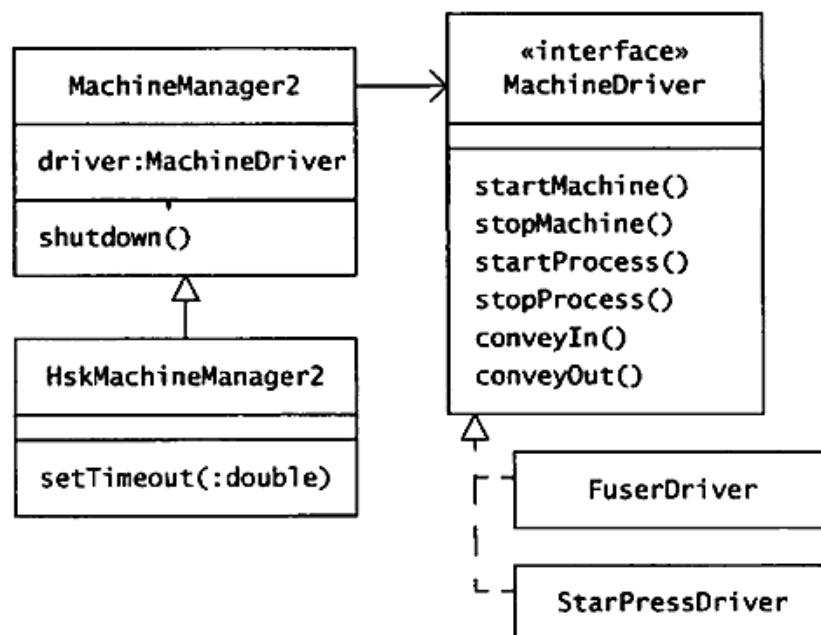


图 B.5 该图显示出了一个抽象 (MachineManager 的继承体系) 与实现 (抽象所使用的 MachineDriver 接口) 之间的分离

挑战 6.4 的答案（第 58 页）

图 B.6 给出了解决方案。

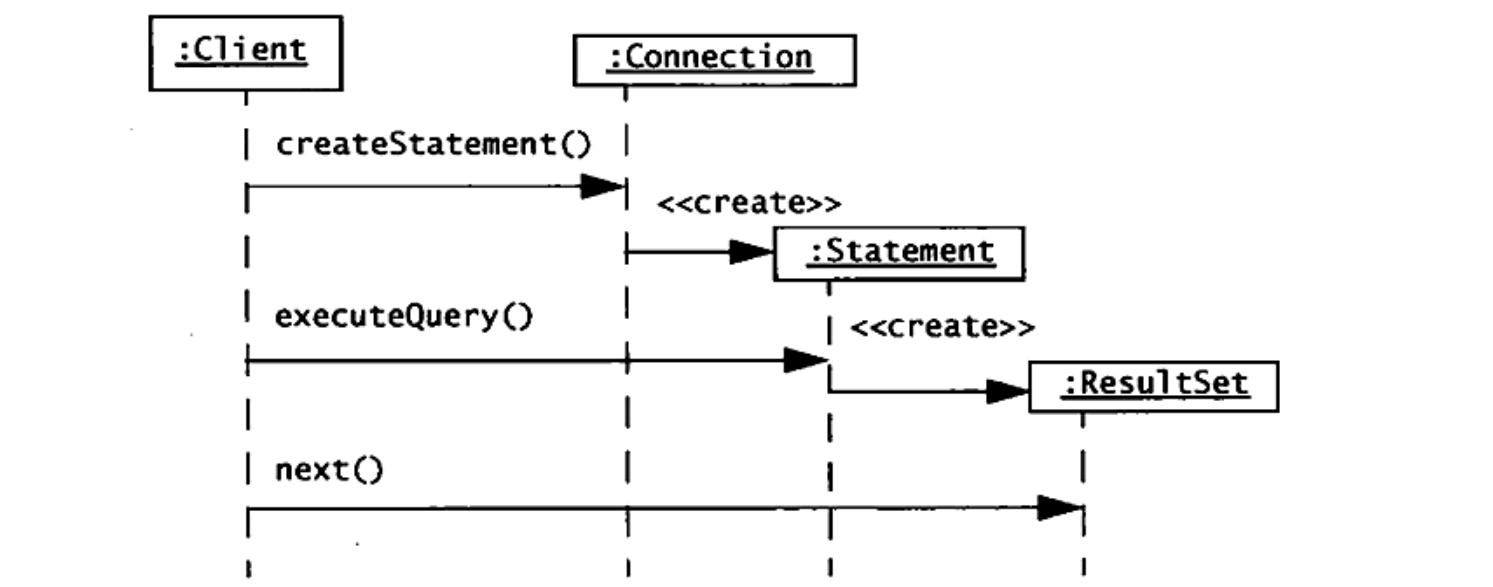


图 B.6 该图展示了在 JDBC 应用程序中常见的消息流

挑战 6.5 的答案（第 59 页）

为 SQL Server 编写专用代码有两种观点：

1. 我们不能预测未来，现在就耗费资金为将来可能永远不会发生的变化做准备，是一种常见的错误。我们现在只有 SQL Server，速度越快就意味着越短的响应时间，对目前而言就意味着节约了资金。
2. 因为仅使用 SQL Server，我们可以使用数据库提供的各种特性，而无须担心其他的数据库驱动器是否支持。

支持通用 SQL 驱动器的，也有两种观点：

1. 如果使用通用的 SQL 对象编写代码，当要改变数据库，例如使用 Oracle 时，就更容易修改。如果代码只能支持 SQL Server，就会导致无法从竞争激烈的数据库市场获利。
2. 使用通用的驱动器，使得我们可以编写一些试验性的代码，能够运行在一些便宜的数据库上，例如 MySql，而无须依赖于 SQL Server 数据库。

第 7 章 职责型模式介绍

挑战 7.1 的答案 (第 62 页)

以下是该图存在的问题:

- `Rocket.thrust()`方法返回了 `Rocket` 对象, 而不是数值类型或者物理数量值。
- `LiquidRocket` 类定义了 `getLocation()` 方法, 但是, 无论设计图还是问题域都没有说明火箭模型应该持有位置属性。即便需要该属性, 也不能只为液态火箭提供该属性, 而不管其他的火箭对象。
- `isLiquid()` 方法可以使用 `instanceof` 操作符来判断能否处理传入的对象, 但是这样就必须在超类中定义一个返回 `false` 的 `isLiquid()` 方法。
- `CheapRockets` 的类名采用了复数形式, 其他类的名称都是单数。
- `CheapRockets` 类实现了 `Runnable` 接口, 但该接口与问题域中的 `CheapRocket` 对象没有任何关系。
- 可以将模型中的廉价特征作为属性建模, 而无须为它单独创建一个类。
- 引入 `CheapRockets` 类使得我们无法区分火箭的燃料究竟是液态还是固态。例如, 无法为廉价的液态火箭建模。
- 模型将 `Firework` 定义为 `LiquidRocket` 的子类, 这就错误地表明所有焰火都属于液态火箭。
- 模型显示 `Reservation` 类与焰火类型存在直接关联, 然而问题域中却不存在这样的关系。
- `Reservation` 类拥有 `city` 对象的副本, 然而, 事实上它可以从 `Location` 对象中获得 `city` 对象。
- `CheapRockets` 由 `Runnable` 对象组成, 这有些奇怪。

挑战 7.2 的答案 (第 63 页)

这一挑战的价值不在于获得正确的答案, 而是锻炼你的思维, 以便于设计出良好的类。比较你的定义, 是否包含如下几点。

- 类的详细定义为：类是字段与方法的集合，字段持有数据值，而方法则操作这些数据值。
- 类创建了一个字段集合，即为对象定义了属性。这些属性的类型可以是其他类类型，也可以是基本数据类型，例如 `boolean` 和 `int`，还可以是接口类型。
- 类的设计者应该了解类的属性之间的关系。
- 类应该是内聚的。
- 类的名称能够反映类的意图，它既是属性的集合，又要符合类的行为。
- 类必须支持它所定义的所有行为，并实现它所继承（实现）的超类（接口）中的所有方法。（当然，是否实现超类或接口的所有方法，需依情况而定。）
- 类与其超类之间的关系必须满足继承关系。
- 类的每个方法名称都应该很好地表达方法要做的事情。

挑战 7.3 的答案（第 64 页）

有两种情况与操作调用的结果有关，要么它依赖于接收对象的状态，要么是接收对象的类。除此之外，可以使用其他人指定的名称。

方法的结果取决于对象状态的一个例子，可以参见第 6 章中 `MachineManager2` 类的 `stopMachine()` 方法。调用该方法的影响取决于 `MachineManager2` 对象使用哪个驱动程序。

当多态成为设计的一部分时，调用操作的结果部分或整体取决于接收对象的类。该原则在许多模式中都有体现，尤其是工厂方法模式、状态模式、策略模式、命令模式和解释器模式。例如，策略类的层次结构均实现了 `getRecommended()` 方法，使用不同的策略来推荐焰火。很容易预见到 `getRecommended()` 会推荐一种焰火，但并不知道接收 `getRecommended()` 调用的对象类型，只有在获知了使用哪种策略时，才会知晓。

第三种情况则是由于其他人已经定义了该名称。假设方法作为回调函数，而你已经重写了 `MouseListener` 类的 `mouseDown()` 方法。此时，就必须使用 `mouseDown()` 作为方法名，即使它的名称与意图不符。

挑战 7.4 的答案（第 65 页）

代码编译没有问题。访问权限被定义在类级别，而非对象级别。例如，一个 `Firework` 对象可以访问另外一个 `Firework` 对象的私有变量与方法。

第 8 章 单例 (Singleton) 模式

挑战 8.1 的答案（第 68 页）

为避免其他开发人员实例化你定义的类，可以创建唯一一个构造函数，并将其设置为私有的访问权限。注意，如果创建了其他非私有的构造函数，或者没有创建任何构造函数，其他对象都能够实例化该类。

挑战 8.2 的答案（第 68 页）

延迟实例化单例对象有两个原因：

1. 在静态初始化时，没有足够信息对单例对象进行初始化。例如，工厂单例就必须等待真正工厂的机器，才能建立通信通道。
2. 选择延迟初始化单例对象与获取资源有关，例如数据库连接，尤其是在一个特定会话中，它包含的应用程序并不需要该单例对象时。

挑战 8.3 的答案（第 70 页）

当两个线程几乎同时调用 `recordWipMove()` 方法时，就可能发生混乱。我们的解决方案必须解决这一问题：

```
public void recordWipMove() {  
    synchronized (classLock) {  
        wipMoves++;  
    }  
}
```

在一个线程执行递加操作时，另外的线程可以被激活并执行吗？当然是的。并非所有的机器都使用一个单独的指令对变量进行递加操作，我们无法保证编译器是在同一种情况下使用它们。一种有效策略是在多线程应用中，严格限制对单例数据的访问。对于这样的应用，可以参考 *Concurrent Programming in Java™*（由 Lea 在 2000 年编写）一书获得更多信息。

挑战 8.4 的答案（第 71 页）

OurBiggestRocket: 类名不合适。应该将类似“biggest”这样的属性表现为模型的属性，而非类名。如果开发人员必须提供这个类，可以实现为单例类。

TopSalesAssociate: 该类存在与 **OurBiggestRocket** 相同的问题。

Math: 该类是一个工具类，只有静态方法而无实例方法。它不是单例类。但请注意，它包含了一个私有构造函数。

System: 同样是一个工具类。

PrintStreem: 虽然 **System.out** 对象是一个 **PrintStream** 对象，并且承担了独一无二的职责，但它并非是 **PrintStream** 的唯一实例，所以它不是单例类。

PrintSpooler: **PrintSpoolers** 可以包含一台或多台打印机，所以很明显它不是单例类。

PrintManager: 在 Oozinoz 公司，有多台打印机，可以通过 **PrintManager** 单例对象来查询这些打印机的位置。

第 9 章 观察者 (Observer) 模式

挑战 9.1 的答案（第 75 页）

参考答案如下：

```
public JSlider slider() {
    if (slider == null) {
        slider = new JSlider();
        sliderMax = slider.getMaximum();
        sliderMin = slider.getMinimum();
        slider.addChangeListener(this);
        slider.setValue(slider.getMinimum());
    }
    return slider;
}
```

```

public void stateChanged(ChangeEvent e) {
    double val = slider.getValue();
    double tp = (val - sliderMin) / (sliderMax - sliderMin);
    burnPanel().setTPeak(tp);
    thrustPanel().setTPeak(tp);
    valueLabel().setText(Format.formatToNPlaces(tp, 2));
}

```

上述代码假定存在一个辅助类 `Format`, 用以格式化数值标签。你也许使用了诸如 ““+tp 或者 `Double.toString(tp)` 等表达式方式。(使用固定数目的小数位会使得结果看起来更逼真。)

挑战 9.2 的答案 (第 75 页)

参考答案如图 B.7 所示。为了能够注册标签对象以侦听滑动条事件, 图 B.7 的设计创建了一个实现了 `ChangeListener` 接口的 `JLabel` 子类。新的设计允许依赖于滑动条的组件注册它们感兴趣的事件, 并根据事件更新自身的状态。这种改进存在争议, 但我们还可以将设计重构为 MVC 架构。

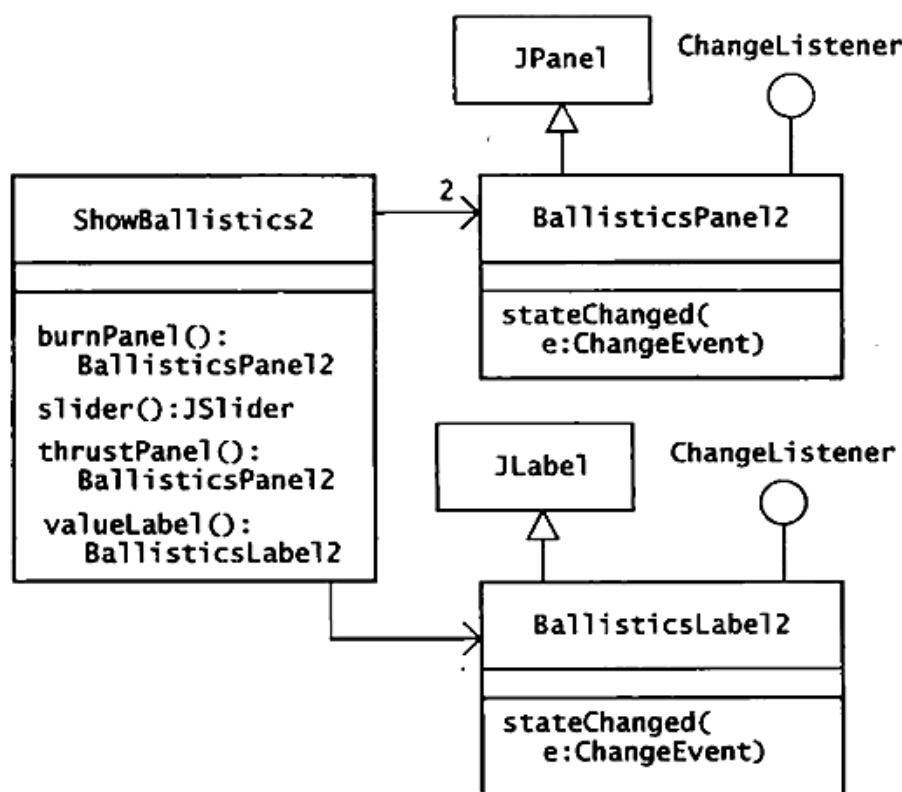


图 B.7 在本设计中, 依赖于滑动条的组件实现了 `ChangeListener`, 这样就可以注册滑动条事件

挑战 9.3 的答案（第 78 页）

图 B.8 给出了解决方案。

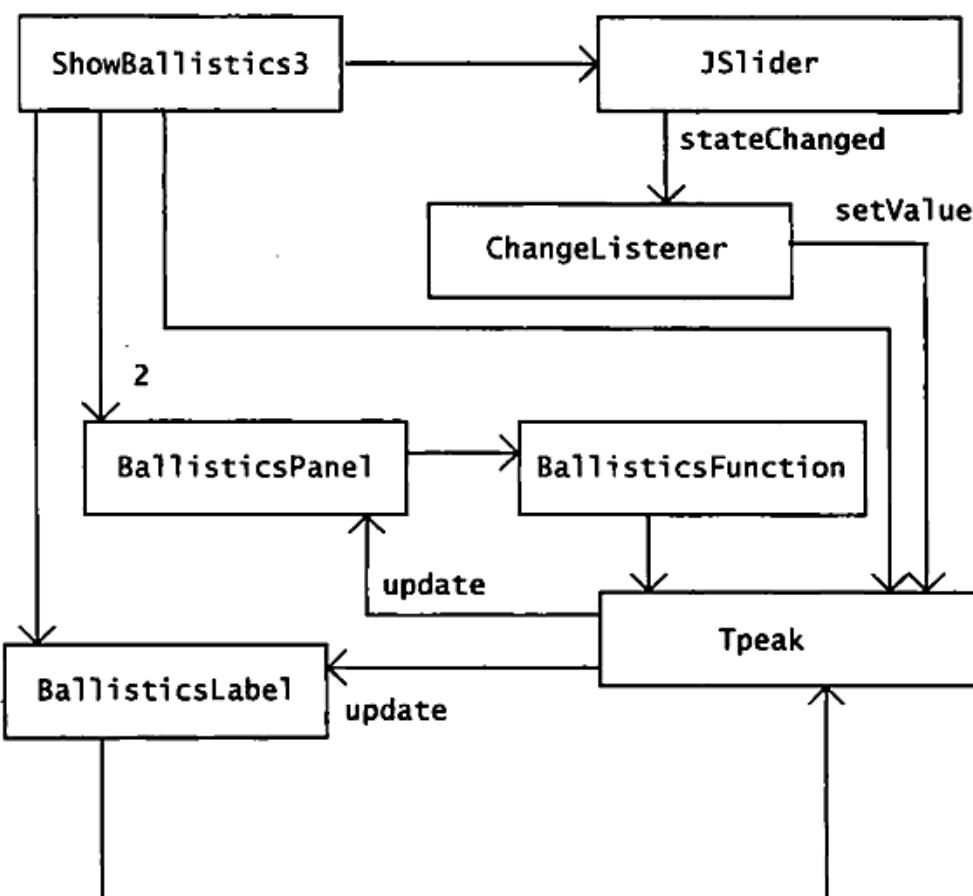


图 B.8 这一设计让应用程序观察滚动条；文本框和面板观察持有 tPeak 值的对象

持有峰值的 `Tpeak` 对象在本设计中起到了关键作用。`ShowBallistics3` 应用程序创建了 `Tpeak` 对象，只要移动了滑动条，该对象的值就会更新。显示组件（文本框与面板）被注册为 `Tpeak` 对象的观察者，实现对 `Tpeak` 对象的“侦听”。

挑战 9.4 的答案（第 80 页）

参考答案为：

```
package app.observer.ballistics3;
import javax.swing.*;
import java.util.*;

public class BallisticsLabel extends JLabel
    implements Observer {
```

```
public BallisticsLabel(Tpeak tPeak) {  
    tPeak.addObserver(this);  
}  
  
public void update(Observable o, Object arg) {  
    setText("") + ((Tpeak) o).getValue());  
    repaint();  
}  
}
```

挑战 9.5 的答案(第 80 页)

参考答案为:

```
protected JSlider slider() {  
    if (slider == null) {  
        slider = new JSlider();  
        sliderMax = slider.getMaximum();  
        sliderMin = slider.getMinimum();  
        slider.addChangeListener(new ChangeListener() {  
            public void stateChanged(ChangeEvent e) {  
                if (sliderMax == sliderMin) return;  
                tPeak.setValue(  
                    (slider.getValue() - sliderMin)  
                    / (sliderMax - sliderMin));  
            }  
        });  
        slider.setValue(slider.getMinimum());  
    }  
    return slider;  
}
```

挑战 9.6 的答案(第 81 页)

图 B.9 给出了当用户移动弹道应用程序中的滑动条时产生的调用流程。

挑战 9.7 的答案(第 83 页)

图 B.10 给出了可能的解决方案。注意，我们可以借助 `Observer` 和 `Observable` 来实现相同的设计。设计的关键在于 `Tpeak` 类，它可以通过维护一个具有侦听能力的对象，使得自己成

为可观察的对象。

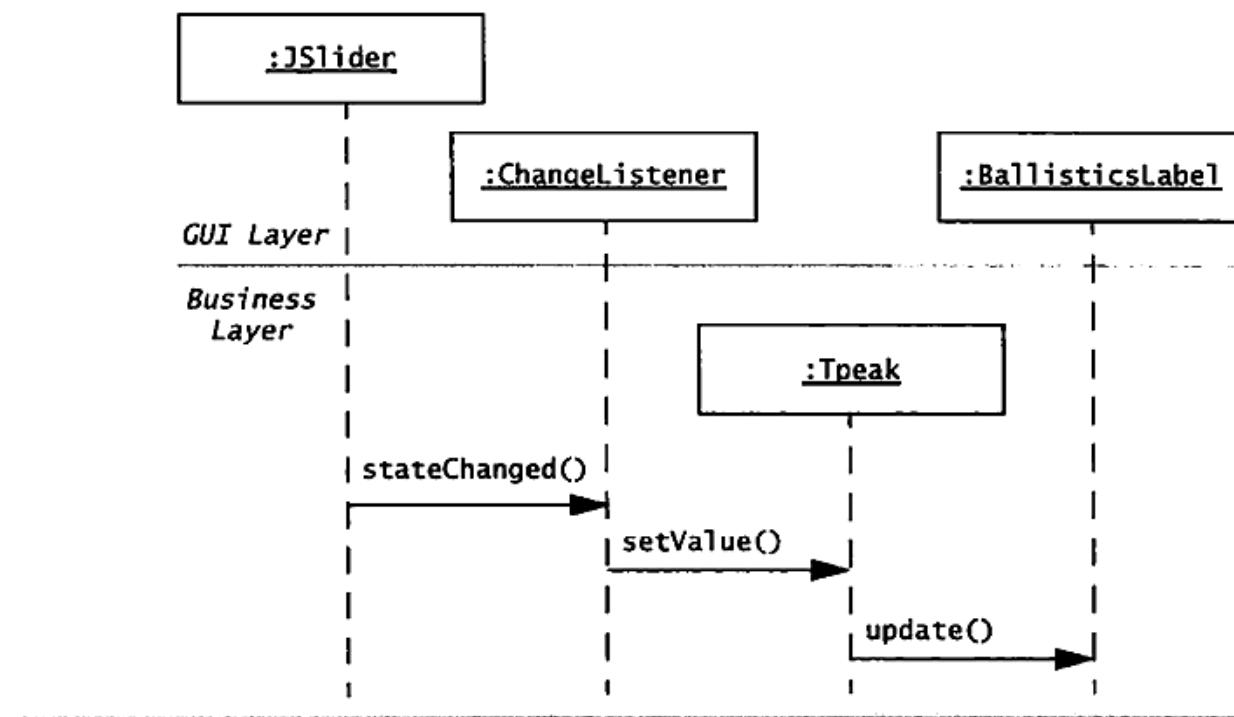


图 B.9 MVC 使得变更路径穿越了业务层

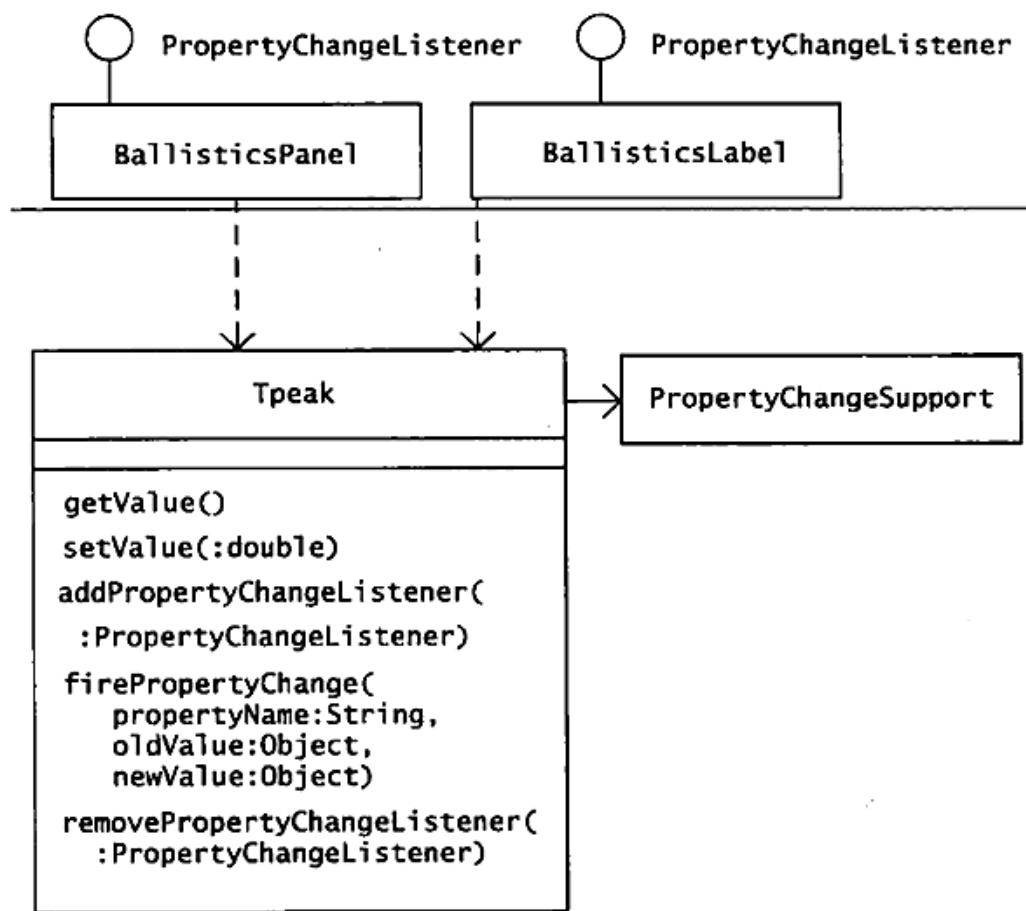
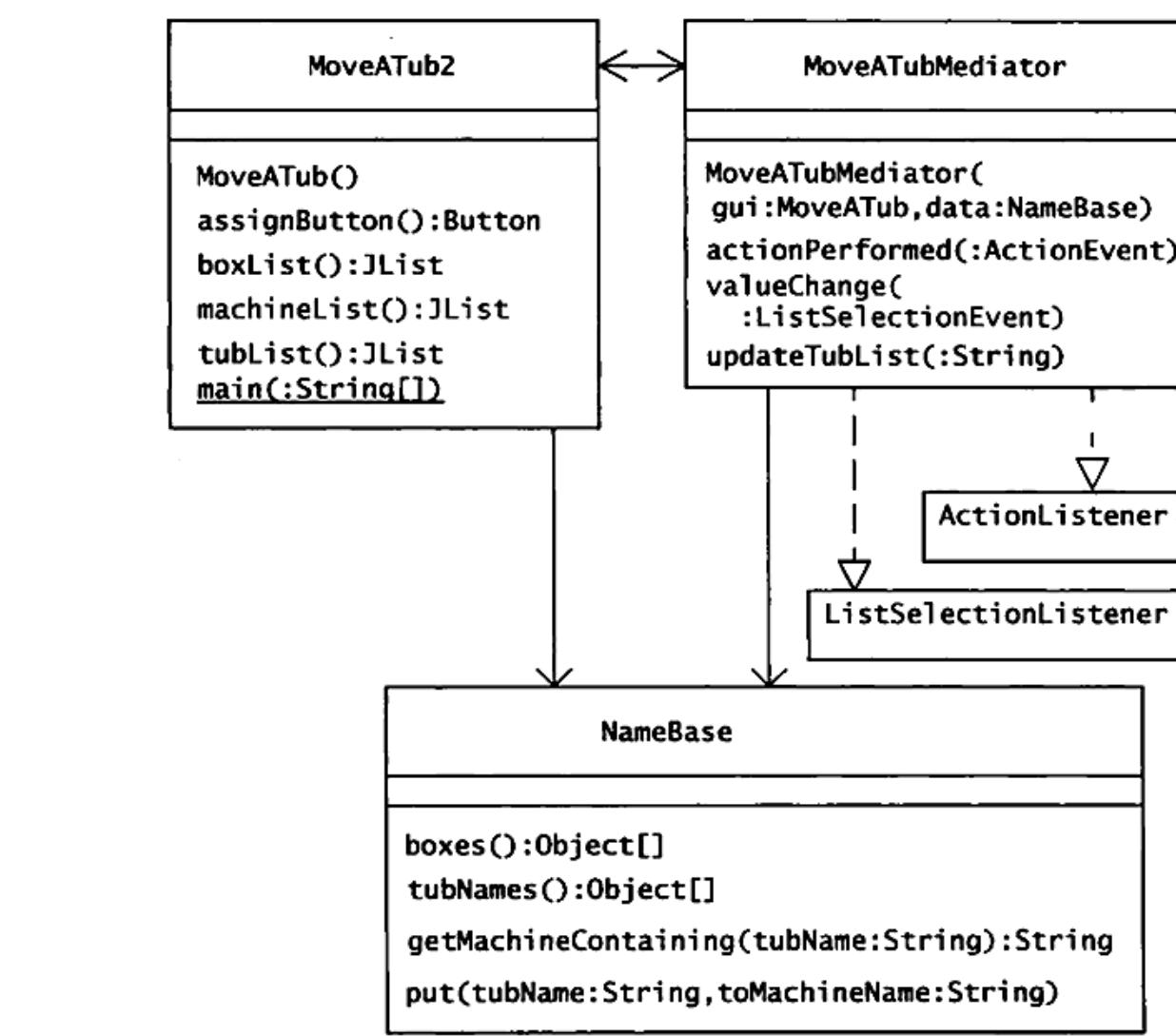


图 B.10 通过将面向侦听的调用委派给 PropertyChangeSupport 对象, Tpeak 类就能够增加侦听行为

第10章 调停者(Mediator)模式

挑战10.1的答案(第88页)

图B.11给出了一个解决方案。



图B.11 MoveATub类处理了组件的构建，MoveATubMediator类负责处理事件

在这一设计中，调停者类表现出了由Fowler等人在1999年提出的“特性依恋(feature envy)”特征。看起来，该类对GUI类的兴趣远远超过了对自身的兴趣，如valueChanged()方法所示：

```

public void valueChanged(ListSelectionEvent e) {
    // ...
  
```

```
gui.assignButton().setEnabled(  
    ! gui.tubList().isSelectionEmpty()  
    && !gui.machineList().isSelectionEmpty());  
}
```

某些开发人员并不喜欢这样的设计，但是，你可能会倾向于让一个类负责 GUI 组件的构造与布局，而将负责交互的组件以及用例流程单独分开。

挑战 10.2 的答案（第 88 页）

图 B.12 给出了一种解决方案。

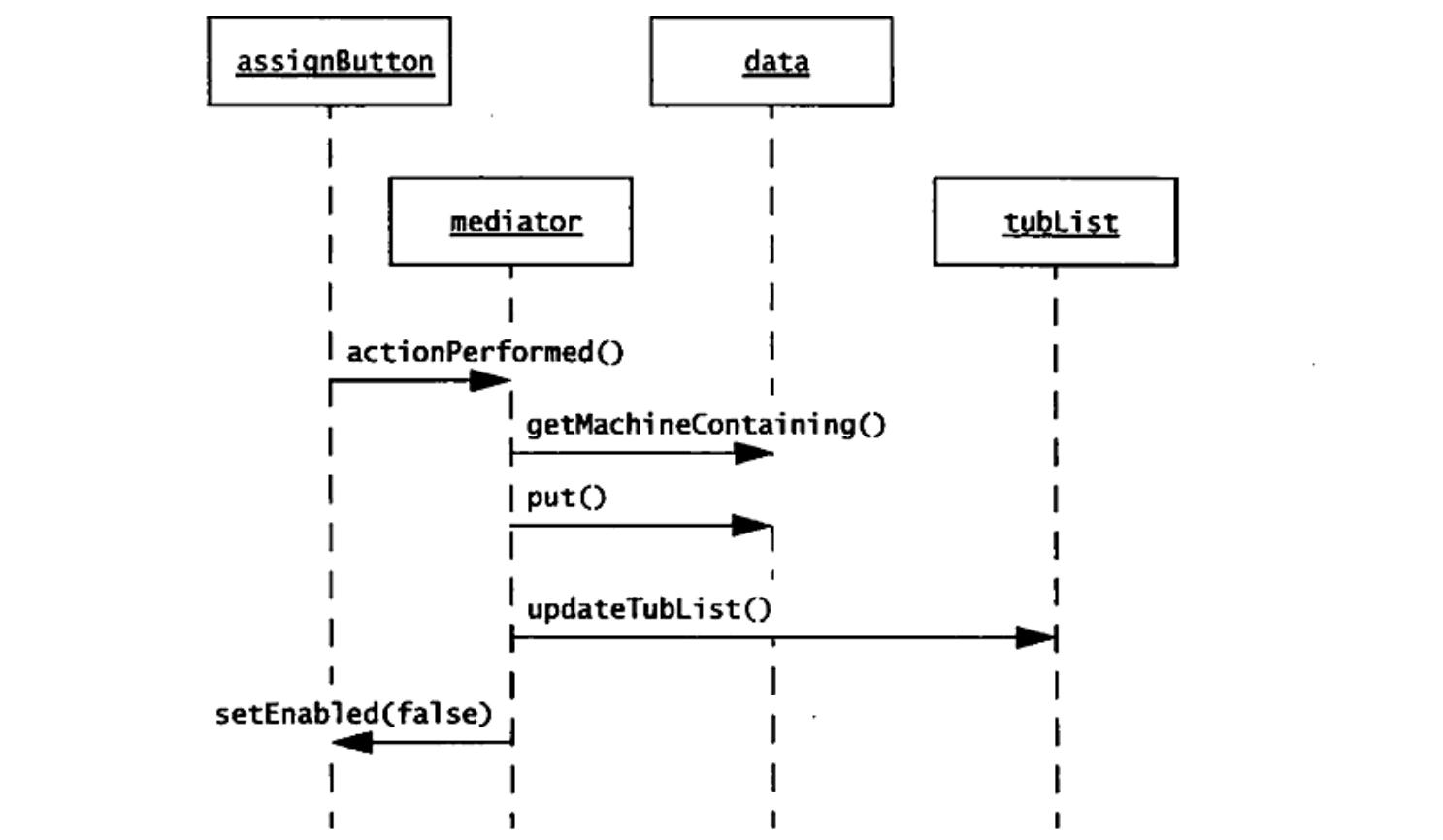


图 B.12 该图凸显了调停者的核心角色

这里给出的解决方案将调停者的角色定位为分发器，能够接收事件，并负责更新所有受到影响的对象。

挑战 10.3 的答案（第 91 页）

图 B.13 给出了被更新对象的关系图。

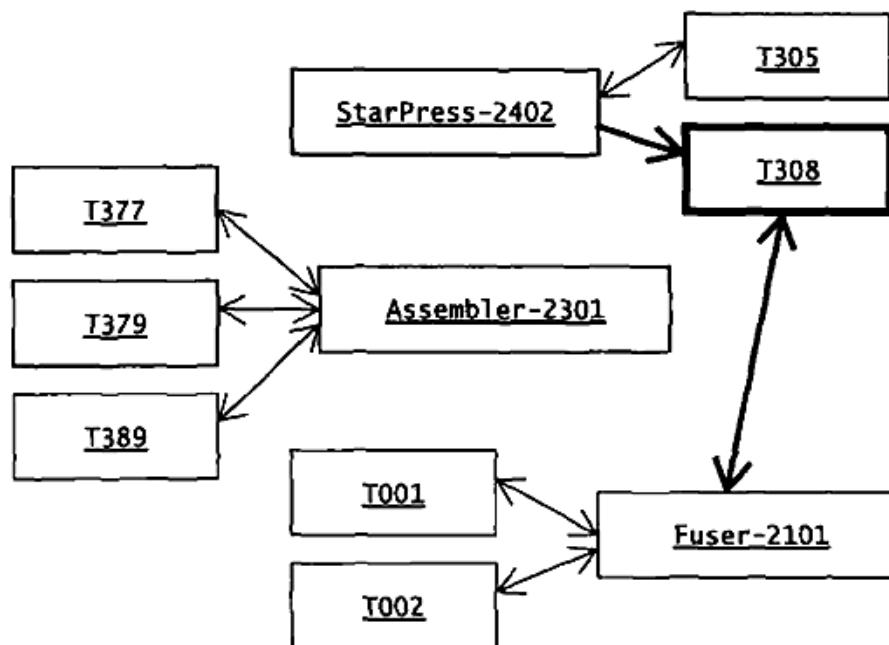


图 B.13 两台机器认为它们包含了材料箱 T308。无论是关系表还是实际应用，都不允许对象模型出现这样的情形（图中的粗线重点突出了这一问题）

开发人员的代码引入了这样的问题：StarPress-2402 仍然认为它拥有材料箱 T308。在关系表中，只要改变某一行的机器属性，就会自动地将材料箱从优先级高的机器中移除。如果关系被分散到一个分布式对象模型中，则这种自动移除工作将不会发生。材料箱/机器的关系模型需要独立的逻辑，能够转移到独立的调停者对象中。

挑战 10.4 的答案 (第 95 页)

TubMediator 类的完整代码如下所示：

```

package com.oozinoz.machine;
import java.util.*;

public class TubMediator {
    protected Map tubToMachine = new HashMap();

    public Machine getMachine(Tub t) {
        return (Machine) tubToMachine.get(t);
    }

    public Set getTubs(Machine m) {
        Set set = new HashSet();
    }
}
  
```

```
Iterator i = tubToMachine.entrySet().iterator();
while (i.hasNext()) {
    Map.Entry e = (Map.Entry) i.next();
    if (e.getValue().equals(m))
        set.add(e.getKey());
}
return set;
}

public void set(Tub t, Machine m) {
    tubToMachine.put(t, m);
}

}
```

挑战 10.5 的答案（第 95 页）

- 外观模式有助于对大型应用程序的重构。
- 桥接模式会将抽象的操作移到接口中。
- 观察者模式可以用于对 MVC 架构的支持。
- 享元模式会将对象中不变的部分分离出来，使之能够被共享。
- 构建者模式将构造对象的逻辑从类中分离出来。
- 工厂方法模式通过将类层次结构中的某方面行为转移到平行的类层次结构中，以减少类层次结构的职责范围。
- 状态模式和策略模式可以将与状态或策略有关的行为转移到单独的类中。

第 11 章 代理 (Proxy) 模式

挑战 11.1 的答案（第 101 页）

参考答案为：

```
public int getIconHeight() {
    return current.getIconHeight();
}
```

```
public int getIconwidth() {
    return current.getIconwidth();
}

public synchronized void paintIcon(
    Component c, Graphics g, int x, int y) {
    current.paintIcon(c, g, x, y);
}
```

挑战 11.2 的答案(第 101 页)

该设计存在的问题如下所示。

- 仅仅将部分调用转发给底层的 `ImageIcon` 对象是很危险的。`ImageIconProxy` 类从 `ImageIcon` 类继承了许多字段以及至少 25 个方法。要成为真正的代理，`ImageIconProxy` 对象需要转发大多数甚至所有的调用。彻底地调用转发需要许多可能存在错误的方法，随着 `ImageIcon` 类以及其超类发生变化，还需要做相应的调整。
- 你可能会质疑“缺失”的图片以及所需的图片在设计时是否真正到位。最好将图片传递过来，而不是让类负责去查找。

挑战 11.3 的答案(第 103 页)

`load()`方法将图像设置为“Loading…”，而 `run()`方法则执行在一个单独线程中，用以加载需要的图片：

```
public void load(JFrame callbackFrame) {
    this.callbackFrame = callbackFrame;
    setImage(LOADING.getImage());
    callbackFrame.repaint();
    new Thread(this).start();
}

public void run() {
    setImage(new ImageIcon(
        ClassLoader.getSystemResource(filename))
        .getImage());
    callbackFrame.pack();
}
```

挑战 11.4 的答案（第 108 页）

如类图所示，`RocketImpl` 构造函数将接收价格和最高点作为参数：

```
Rocket biggie = new RocketImpl(29.95, 820);
```

你可以将 `biggie` 声明为 `RocketImpl` 类型。然而，重要的是，客户端期望的是实现了 `Rocket` 接口的类型。

挑战 11.5 的答案（第 109 页）

完整的类图如图 B.14 所示。可以将图中 `getApogee()` 方法的接收器类型改为 `Rocket`。实际上，服务器和客户端都将这两个对象作为 `Rocket` 接口的实例进行引用。

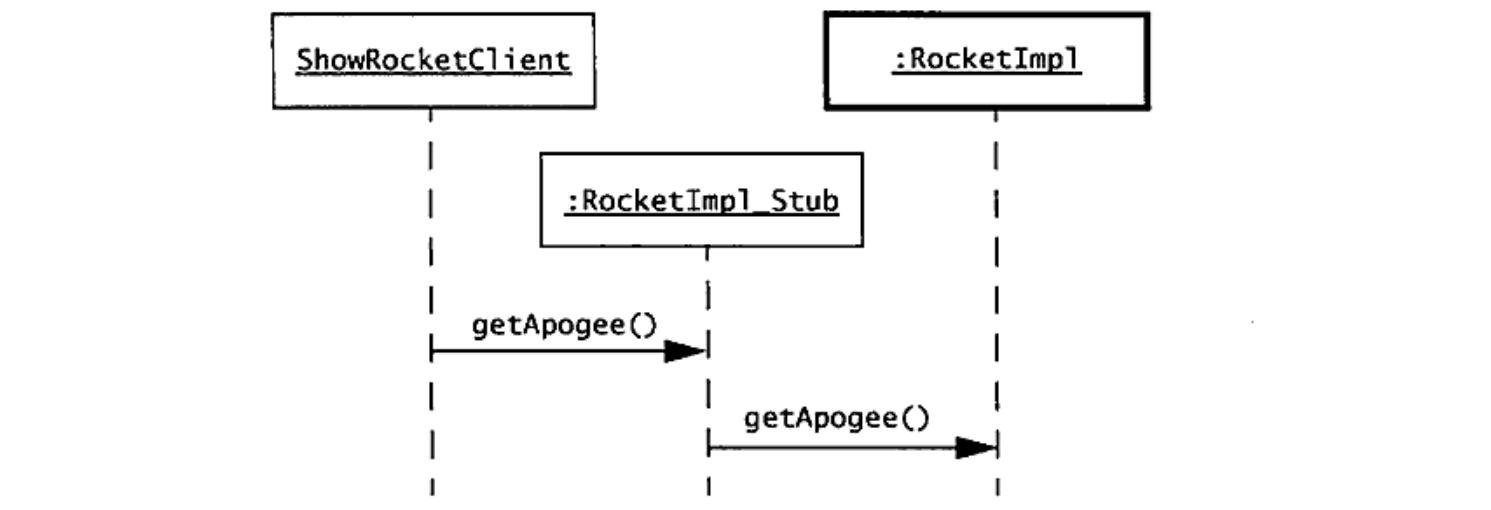


图 B.14 代理会转发客户端调用，使得远程对象看起来就像本地对象

第 12 章 职责链（Chain of Responsibility）模式

挑战 12.1 的答案（第 116 页）

在 Oozinoz 的设计中，使用了职责链去寻找负责机器的工程师，这存在一些潜在的问题。

- 我们没有指定如何设置职责链，让机器可以知道其父节点。实际上，很难确保父节点永远都不为空。

- 可以想象的是，对父对象的搜索可能会陷入到无限循环中，这取决于父对象是如何建立的。
- 并非所有对象都需要提供这些新方法所表示的行为。（例如，最顶端的元素就没有父对象。）
- 当前设计局限于细节，关心系统如何知道是哪些工程师正在工厂，并是否可用。该设计并不清楚完成责任需要的“实际时间”。

挑战 12.2 的答案（第 119 页）

你的类图应该与图 B.15 相似。

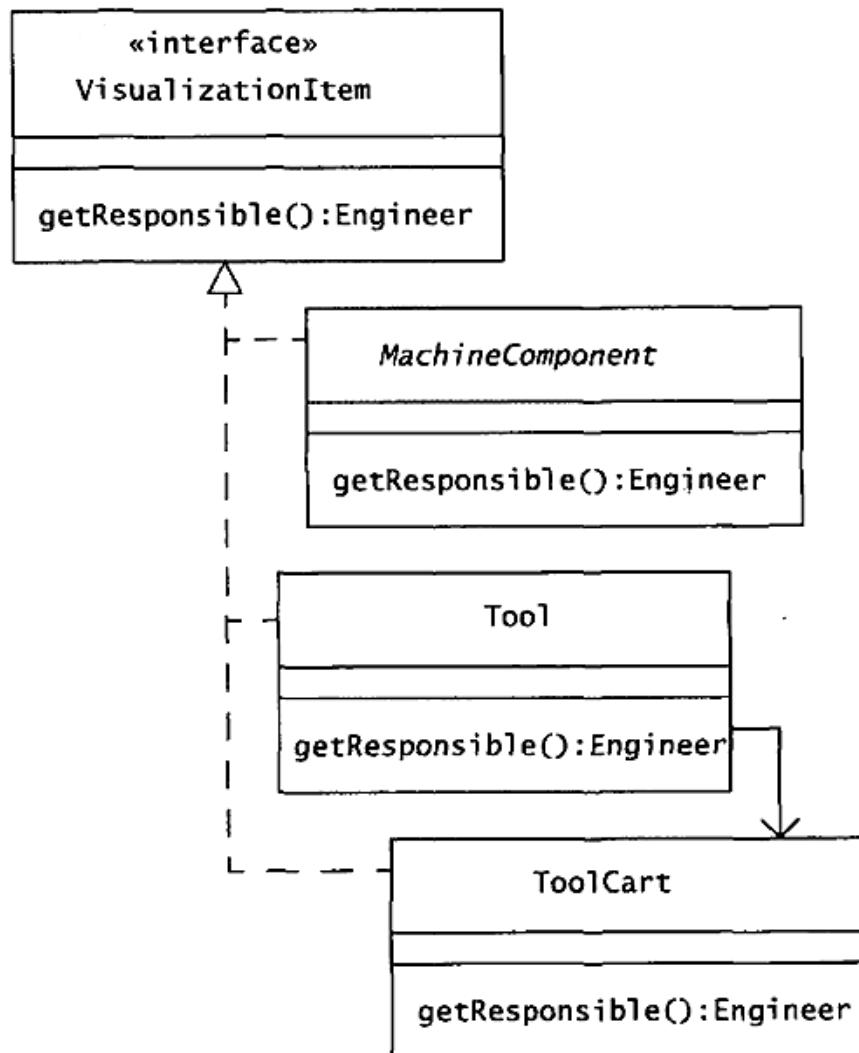


图 B.15 每个 `VisualizationItem` 对象都会告知其承担职责的工程师。从内部看，`VisualizationItem` 对象会将请求转发给其他父对象

在这个设计中，任何模拟对象的客户端都可以通过该模拟对象获知责任工程师的信息。这种方式可以让客户不必了解那个对象的职责，从而将负担转嫁给实现了 `VisualizationItem` 接口的对象。

挑战 12.3 的答案（第 119 页）

A. `MachineComponent` 对象拥有一个显式分配的责任人。如果没有，就将请求转交给它的父对象。

```
public Engineer getResponsible() {  
  
    if (responsible != null)  
        return responsible;  
    if (parent != null)  
        return parent.getResponsible();  
    return null;  
}
```

B. `Tool.Responsible` 类的代码反映了“工具总是指派给对应的工具车”原则。

```
public Engineer getResponsible() {  
    return toolCart.getResponsible();  
}
```

C. `ToolCart` 类的代码反映了“工具车都有一名责任工程师”原则。

```
public Engineer getResponsible() {  
    return responsible;  
}
```

挑战 12.4 的答案（第 120 页）

解决方案应该如图 B.16 所示。

我们提供的构造函数能够保证无论是否提供指定的工程师，都能够实例化 `Machine` 和 `MachineComposite` 对象。无论 `MachineComponent` 对象是否拥有指定的工程师，都能够从它的父对象获得责任工程师。

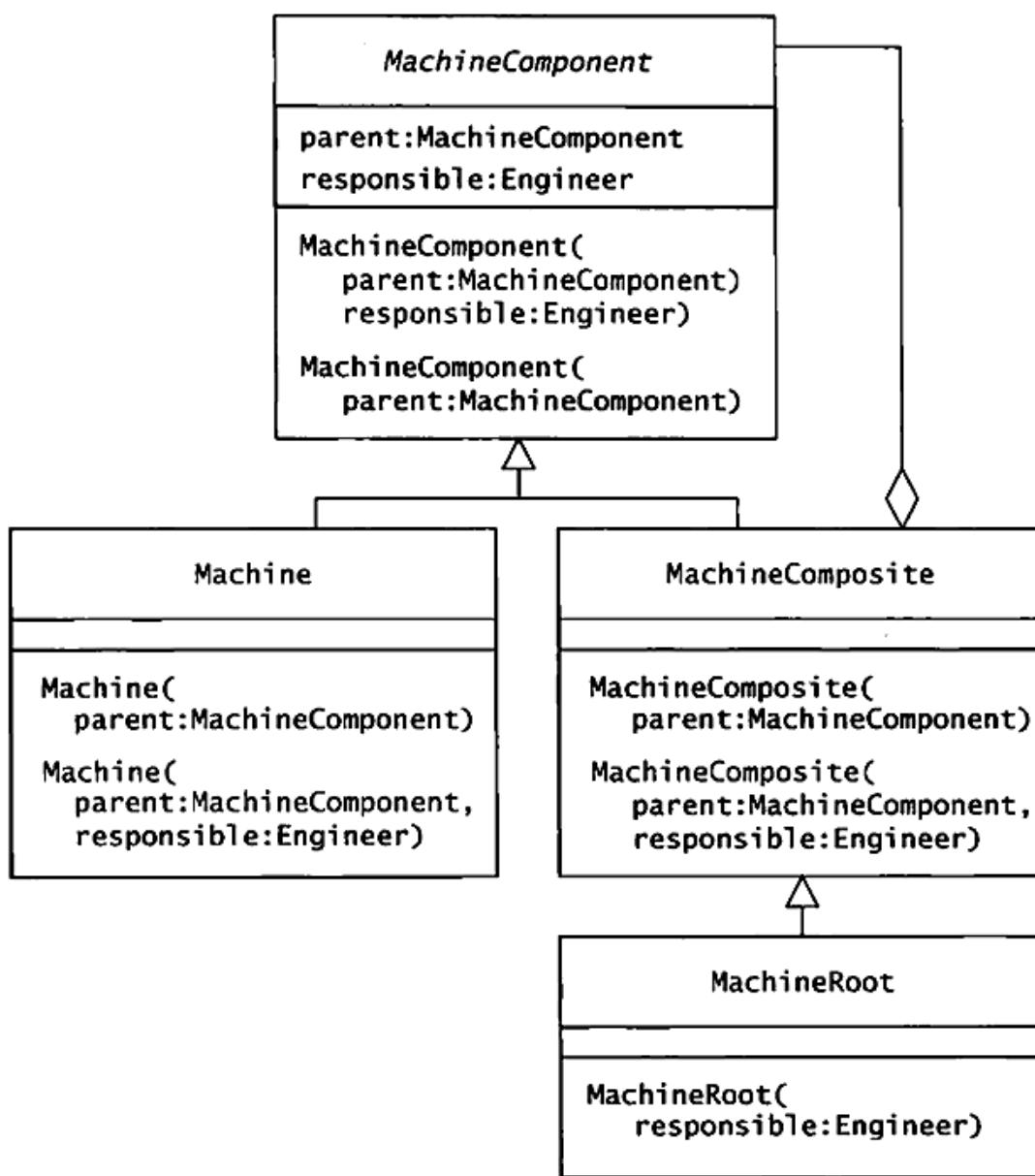


图 B.16 MachineComponent 类层次结构的构造函数必须支持两条原则：MachineRoot 对象必须拥有一个责任工程师；除了根对象的每个 MachineComponent 对象都必须具有父对象

挑战 12.5 的答案 (第 121 页)

职责链可以用于不具有组合结构的对象模型中，如：

- 一群接线工程师组成一个环状链条，他们遵守轮流提供客户服务的标准。如果当前的接线工程师在规定时间内无法回答与产品有关的问题，就通知系统进行切换，由下一个工程师提供帮助。
- 用户输入事件日期之类的信息，解析器组成一个链条对用户文本依次进行解码。

第 13 章 享元 (Flyweight) 模式

挑战 13.1 的答案（第 123 页）

支持字符串为不变类型的正方观点：在实际应用中，字符串常常被调用者共享。如果字符串能够被修改，则调用者对字符串不经意的修改就会影响到其他人。例如，当方法返回字符串类型的客户名时，它仍然持有对该名称值的引用。如果调用者将字符串转换为大写并放入哈希表中，客户名就会发生变化。在 Java 语言中，如果为字符串生成一个大写版本，就必须是一个新对象，而不能修改原来字符串的值。字符串的不变性使得它们在被多个调用者共享时，是安全的。而且，不变的字符串有助于避免系统出现安全风险。

反方观点：字符串的不变性虽然能够避免错误，但却因此付出了较大的代价。首先，即便确定需要修改字符串值，也无法修改。其次，在计算机语言中加入特殊规则，就使得语言变得难以学习和使用。Java 语言与 Smalltalk 语言一样强大，但前者学习起来更难。最后，任何计算机语言都无法避免使用者犯错。如果学习语言更加容易，就可以有更多时间来了解如何创建以及学习测试框架。

挑战 13.2 的答案（第 124 页）

可以将 Substance 中不变的部分，包括名称、符号和原子量转移到 Chemical 类中，如图 B.17 所示。

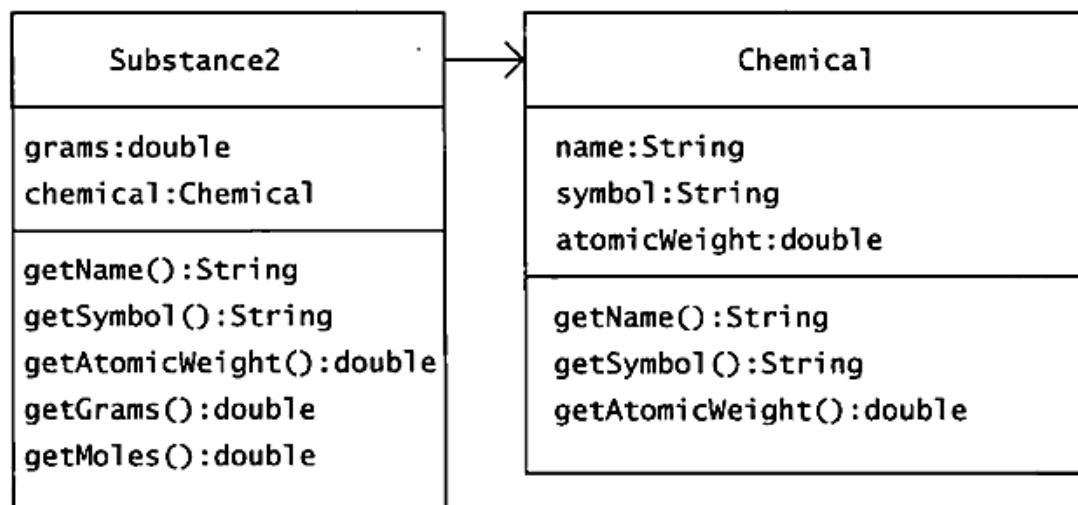


图 B.17 该类图体现了将原来属于 Substance 类中不变的部分提取到单独的类 Chemical 中

现在，类 Substance2 保持了对 Chemical 对象的引用。因此，类 Substance2 能够提供与之前 Substance 类相同的访问器。从内部来讲，这些访问器依赖于 Chemical 类，如下面的 Substance2 方法所示：

```
public double getAtomicweight() {  
    return chemical.getAtomicweight();  
}  
  
public double getGrams() {  
    return grams;  
}  
  
public double getMoles() {  
    return grams / getAtomicWeight();  
}
```

挑战 13.3 的答案（第 126 页）

一种方法是将 Chemical 的构造函数定义为私有的。这就可以防止 ChemicalFactory 类实例化 Chemical 类对象。

若要防止开发人员实例化 Chemical 类自身，可以将 Chemical 和 ChemicalFactory 类放在同一个包中，然后将 Chemical 类的构造函数设定为默认的访问限制（即包级别的访问限制）。

挑战 13.4 的答案（第 127 页）

使用嵌套类的方式过于复杂，但能够彻底确保只有 ChemicalFactory2 类才能实例化新的享元对象。最终的代码如下所示：

```
package com.oozinoz.chemical2;  
import java.util.*;  
  
public class ChemicalFactory2 {  
    private static Map chemicals = new HashMap();  
  
    class ChemicalImpl implements Chemical {  
        private String name;  
        private String symbol;  
        private double atomicweight;
```

```
ChemicalImpl(
    String name,
    String symbol,
    double atomicWeight) {
    this.name = name;
    this.symbol = symbol;
    this.atomicWeight = atomicWeight;
}

public String getName() {
    return name;
}

public String getSymbol() {
    return symbol;
}

public double getAtomicWeight() {
    return atomicWeight;
}

public String toString() {
    return name + "(" + symbol + ")["
        + atomicWeight + "]";
}
}

static {
    ChemicalFactory2 factory = new ChemicalFactory2();
    chemicals.put("carbon",
        factory.newChemicalImpl("Carbon", "C", 12));
    chemicals.put("sulfur",
        factory.newChemicalImpl("Sulfur", "S", 32));
    chemicals.put("salt peter",
        factory.newChemicalImpl(
            "Salt peter", "KN03", 101));
    //...
}

public static Chemical getChemical(String name) {
```

```
    return (Chemical) chemicals.get(
        name.toLowerCase());
    }
}
```

这段代码解决了如下三个问题。

1. `ChemicalImpl` 嵌套类应该是私有的，这样就只有 `ChemicalFactory2` 类才能使用该类。注意，嵌套类的访问范围必须是包级别或公有的，这样包含的类才可以实例化嵌套类。即使将构造函数定义为公有的，如果嵌套类本身被标记为私有，就没有其他类可以调用该构造函数。
2. `ChemicalFactory2` 的构造函数使用了一个静态的初始化器，以确保类只会构建一次化学药品列表。
3. `getChemical()` 方法可以在类的哈希表中根据名称查找化学药品。示例代码使用了小写的化学品名来存储与查询化学品。

第 14 章 构造型模式介绍

挑战 14.1 的答案（第 130 页）

与构造函数有关的特定规则包括如下几项：

- 如果不为类提供一个构造函数，Java 会提供一个默认的。
- 构造函数的名称必须与类名相匹配。（这就是说，构造函数的名称通常都以大写字母开始，而与方法名称的规定不同。）
- 构造函数可以通过 `this()` 和 `super()` 方法调用其他构造函数，只要该调用是放在构造函数的第一句。
- 构造函数的结果是该类的实例，而常规方法的返回值可以是任何类型。
- 可以使用 `new` 关键字或反射来调用构造函数。

挑战 14.2 的答案（第 131 页）

当把如下代码放到 `Fuse.java` 和 `QuickFuse.java` 中时，编译会失败：

```
package app.construction;
public class Fuse {
    private String name;
    public Fuse(String name) { this.name = name; }
}
```

和

```
package app.construction;
public class QuickFuse extends Fuse { }
```

编译器报告的错误大约如下所示：

```
Implicit super constructor Fuse() is undefined for default
constructor. Must define an explicit constructor.
```

当编译器遇到 QuickFuse 类时，就会发生错误，并为其提供一个默认的构造函数。默认的构造函数没有参数，并在默认情况下会调用其超类的无参构造函数。然而，Fuse() 构造函数却接受了一个字符串参数，这意味着编译器不再为 Fuse 类提供默认构造函数。QuickFuse 的默认构造函数无法调用超类的无参构造函数，因为该构造函数并不存在。

挑战 14.3 的答案（第 132 页）

程序会输出：

```
java.awt.Point[x=3,y=4]
```

（它成功地找到了接收两个参数的构造函数，并根据给定的参数值创建了一个新的 point 对象。）

第 15 章 构建者（Builder）模式

挑战 15.1 的答案（第 137 页）

要使解析器更加灵活，可以允许它接收逗号后的多个空格。要做到这一点，需要修改 split() 的调用格式：

```
s.split(", *");
```

或者，如果不接收逗号后的空格，也可以通过初始化如下的 `Regex` 对象，使它能够接收各种类型的空格：

```
s.split(",\\s*")
```

`\s` 字符表示为正则表达式中的“字符类”。注意，所有这些解决方案都假定字段中没有包含逗号。

为了让正则表达式更加灵活，或许你会质疑整个方法。特别的，你可能希望旅游局能够发送 XML 格式的预订信息。这就可以建立一系列的标签，在 XML 解析器中使用和读取这些标签。

挑战 15.2 的答案（第 139 页）

如果 `UnforgivingBuilder` 类的任何一个属性都是无效的，`build()` 方法就会抛出一个异常，否则返回一个有效的 `Reservation` 对象。如下是其中一种实现：

```
public Reservation build() throws BuilderException {
    if (date == null)
        throw new BuilderException("Valid date not found");

    if (city == null)
        throw new BuilderException("Valid city not found");

    if (headcount < MINHEAD)
        throw new BuilderException(
            "Minimum headcount is " + MINHEAD);

    if (dollarsPerHead.times(headcount)
        .isLessThan(MINTOTAL))
        throw new BuilderException(
            "Minimum total cost is " + MINTOTAL);

    return new Reservation(
        date,
        headcount,
        city,
        dollarsPerHead,
        hasSite);
}
```

代码会检查日期与城市值是否设置，并检查总人数以及人均费用的值是否合理。`ReservationBuilder` 超类定义了 `MINHEAD` 和 `MINTOTAL` 常量。

如果构造函数没有问题，就会返回一个有效的 `Reservation` 对象。

挑战 15.3 的答案（第 139 页）

如前所示，如果预订信息没有指定城市或日期，就会抛出异常，这是因为它无法预测这些值。至于缺失总人数与人均费用的值，需要注意如下几点：

- 如果预订请求没有指明总人数和人均费用，就将总人数设置为最小值，而将人均费用设置为最小金额除以总人数。
- 如果没有设置总人数，但却设置了人均费用，则将总人数设置为最小值，并能够保证总费用足以支持此次活动。
- 如果设置了总人数但却没有人均费用，就将人均费用的值设置为能够保证维持活动费用的最大值。

挑战 15.4 的答案（第 140 页）

参考答案如下所示：

```
public Reservation build() throws BuilderException {
    boolean noHeadcount = (headcount == 0);
    boolean noDollarsPerHead = (dollarsPerHead.isZero());

    if (noHeadcount&&noDollarsPerHead) {
        headcount = MINHEAD;
        dollarsPerHead = sufficientDollars(headcount);
    } else if (noHeadcount) {
        headcount = (int) Math.ceil(
            MINTOTAL.dividedBy(dollarsPerHead));
        headcount = Math.max(headcount, MINHEAD);
    } else if (noDollarsPerHead) {
        dollarsPerHead = sufficientDollars(headcount);
    }

    check();
}
```

```
    return new Reservation(
        date,
        headcount,
        city,
        dollarsPerHead,
        hasSite);
}
```

这段代码依赖于 `check()` 方法，它与 `UnforgivingBuilder` 类的 `build()` 方法相似：

```
protected void check() throws BuilderException {
    if (date == null)
        throw new BuilderException("Valid date not found");

    if (city == null)
        throw new BuilderException("Valid city not found");

    if (headcount < MINHEAD)
        throw new BuilderException(
            "Minimum headcount is " + MINHEAD);

    if (dollarsPerHead.times(headcount)
        .isLessThan(MINTOTAL))
        throw new BuilderException(
            "Minimum total cost is " + MINTOTAL);
}
```

第 16 章 工厂方法 (Factory Method) 模式

挑战 16.1 的答案 (第 142 页)

一个好的答案或许是你不必关心 `iterator()` 方法的返回值。重要的是，你必须知道迭代器支持什么样的接口，才能遍历集合中的元素。然而，如果你需要知道当前迭代的元素属于哪个类，可以用下面这行代码输出类名：

```
System.out.println(iter.getClass().getName());
```

结果如下：

```
java.util.AbstractList$Itr
```

Itr 类是 AbstractList 类的内部类。在使用 Java 进行编程的时候，是不会看到这个类的。

挑战 16.2 的答案（第 142 页）

有很多可能的答案，但是 `toString()` 方法可能是创建新对象时最常用的方法。例如，下面的代码创建了一个新的 `String` 对象：

```
String s = new Date().toString();
```

很多时候，`toString()` 的过程都是隐式完成的。例如：

```
System.out.println(new Date());
```

这行代码通过调用 `Date` 对象的 `toString()` 方法，创建了该对象的一个 `String` 对象。

另一种常用的创建新对象的方法是 `clone()`，该方法用于返回调用对象的一个副本。

挑战 16.3 的答案（第 143 页）

工厂方法模式的意图是让对象的提供者来决定创建哪个类。比较而言，`BorderFactory` 类的使用者通常都知道他们要创建的类型。`BorderFactory` 类使用了享元模式来有效地支持大量边界的共享使用。`BorderFactory` 类无须使用者管理对象的复用，而工厂方法模式使用户无须了解实例化哪个类。

挑战 16.4 的答案（第 144 页）

图 B.18 展示了 `CreditCheck` 接口的两个赊购审查类。工厂类提供了一个方法，用来返回 `CreditCheck` 对象。客户类在调用 `createCreditCheck()` 方法时，并不知道会实例化哪个类。

`createCreditCheck()` 方法是一个静态方法，因此客户类不需要为了获得一个 `CreditCheck` 对象而去实例化 `CreditCheckFactory` 类。如果不想让开发人员实例化该类，可以将该类定义成抽象类，或者为该类定义一个静态构造函数。

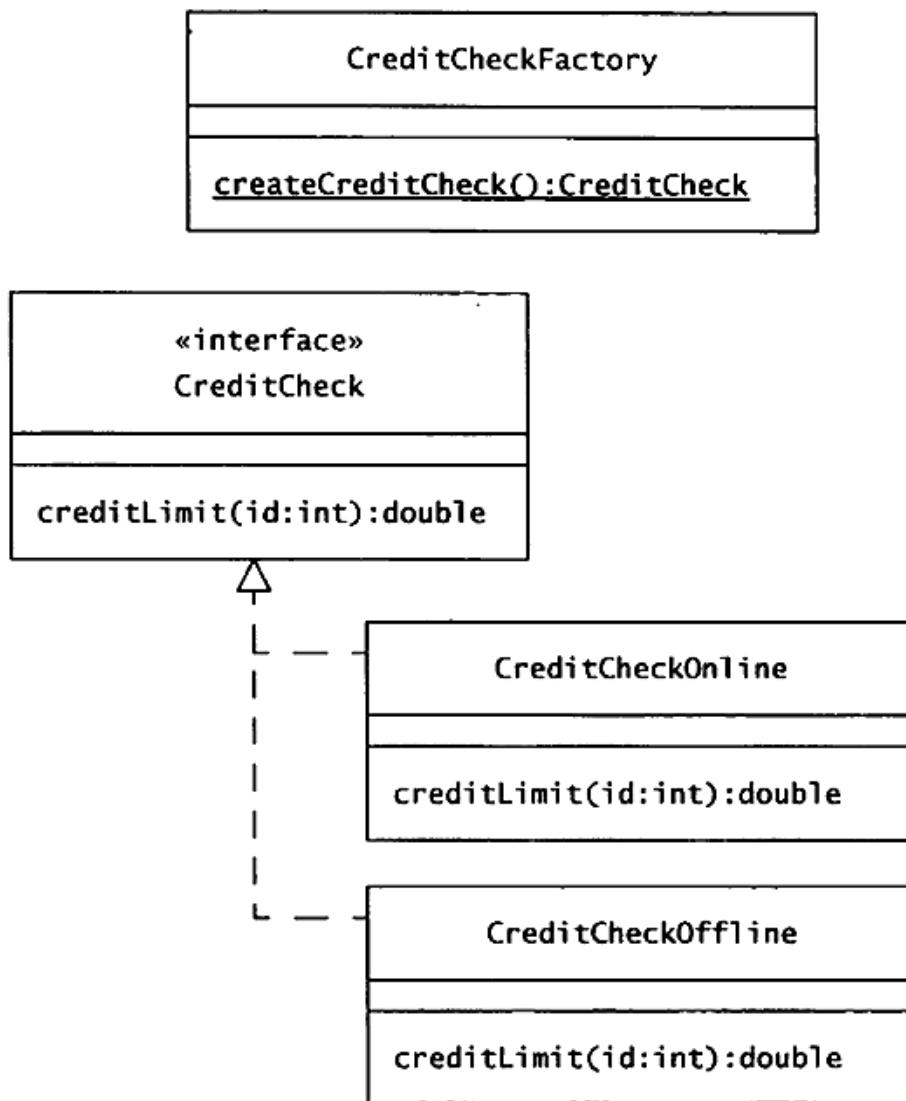


图 B.18 两个类实现了 CreditCheck 接口。实例化哪个类取决于服务提供者，而不需要客户类决定

挑战 16.5 的答案（第 145 页）

如果假定静态方法 `isAgencyUp()` 可以反映实际情况，`createCreditCheck()` 代码就可以简化成这样：

```

public static CreditCheck createCreditCheck() {
    if (isAgencyUp()) return new CreditCheckOnline();
    return new CreditCheckOffline();
}
  
```

挑战 16.6 的答案（第 146 页）

图 B.19 展示了一张合理的 Machine/MachinePlanner 并行结构图。

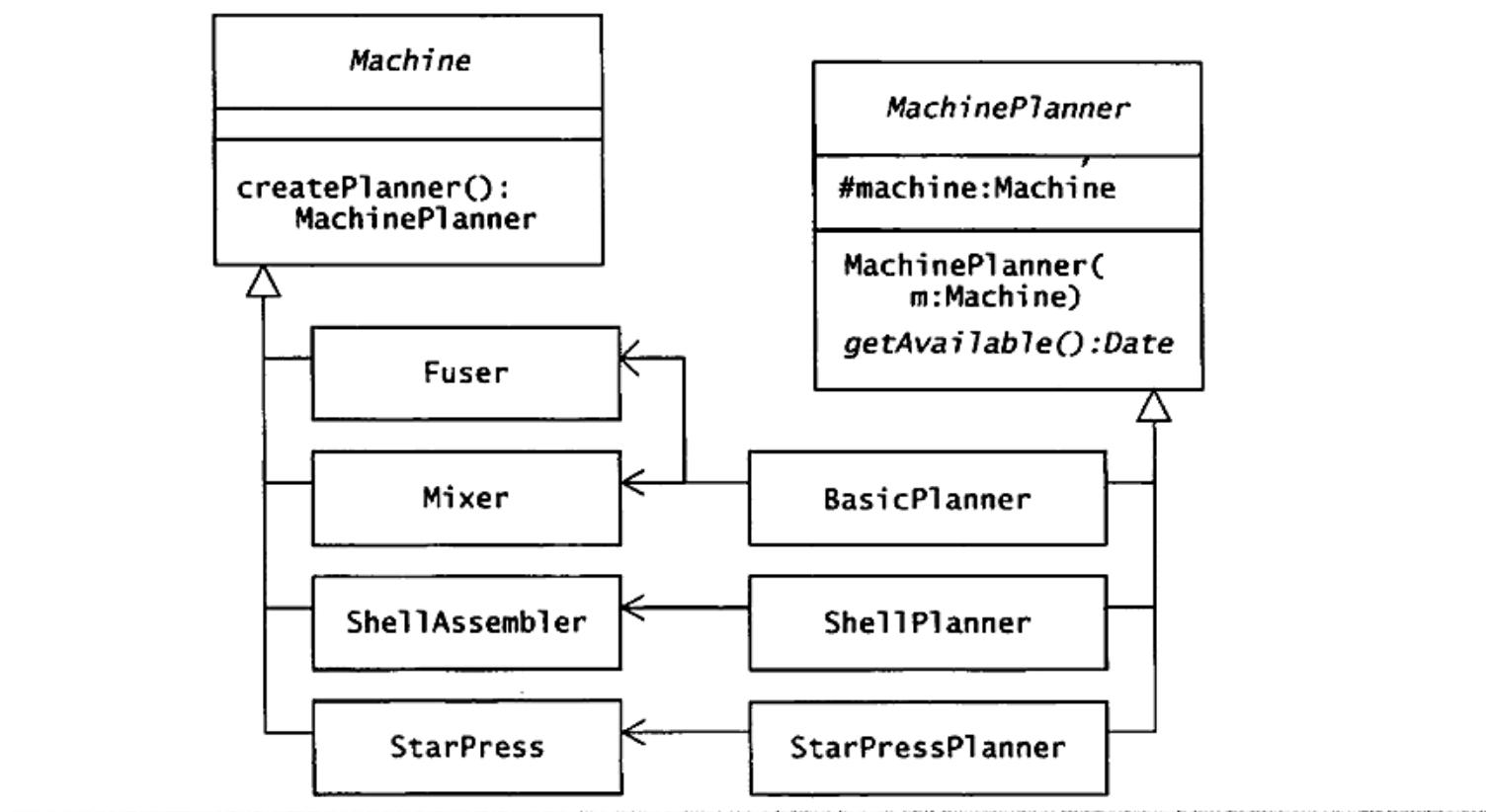


图 B.19 现在的逻辑规划被放在了一个独立的层次结构中。Machine 类的每个子类的 createPlanner() 方法都知道应该实例化哪个计划类

该图说明了 MachinePlanner 类的子类必须实现 getAvailable() 方法，还说明了 MachinePlanner 层次结构中的所有类都必须有一个接收 Machine 对象的构造函数。这样就可以让计划类获得对象的计划信息，比如机器的位置、当前处理的材料数量等。

挑战 16.7 的答案（第 147 页）

Machine 类的 createPlanner() 方法看起来如下所示：

```

public MachinePlanner createPlanner() {
    return new BasicPlanner(this);
}

```

无论 ShellAssembler 类和 StarPress 类是否需要重写该方法，Fuser 类和 Mixer 类都只能继承自该方法。对于 StarPress 类，createPlanner() 方法可以如下所示：

```

public MachinePlanner createPlanner() {
    return new StarPressPlanner(this);
}

```

这些方法用到了工厂方法模式。当我们需要一个计划对象时，就会调用机器的

`createPlanner()`方法。该方法返回什么样的计划对象，取决于具体的机器。

第 17 章 抽象工厂 (Abstract Factory) 模式

挑战 17.1 的答案 (第 151 页)

一种解决方案如下所示

```
public class BetaUI extends UI {
    public BetaUI () {
        Font oldFont = getFont();
        font = new Font(
            oldFont.getName(),
            oldFont.getStyle() | Font.ITALIC,
            oldFont.getSize());
    }

    public JButton createButtonOk() {
        JButton b = super.createButtonOk();
        b.setIcon(getIcon("images/cherry-large.gif"));
        return b;
    }

    public JButton createButtonCancel() {
        JButton b = super.createButtonCancel();
        b.setIcon(getIcon("images/cherry-large-down.gif"));
        return b;
    }
}
```

这段代码尽可能多地使用了超类提供的方法。

挑战 17.2 的答案 (第 152 页)

有一种更为灵活的设计，在接口中指定期待创建的方法和 GUI 属性，如图 B.20 所示。

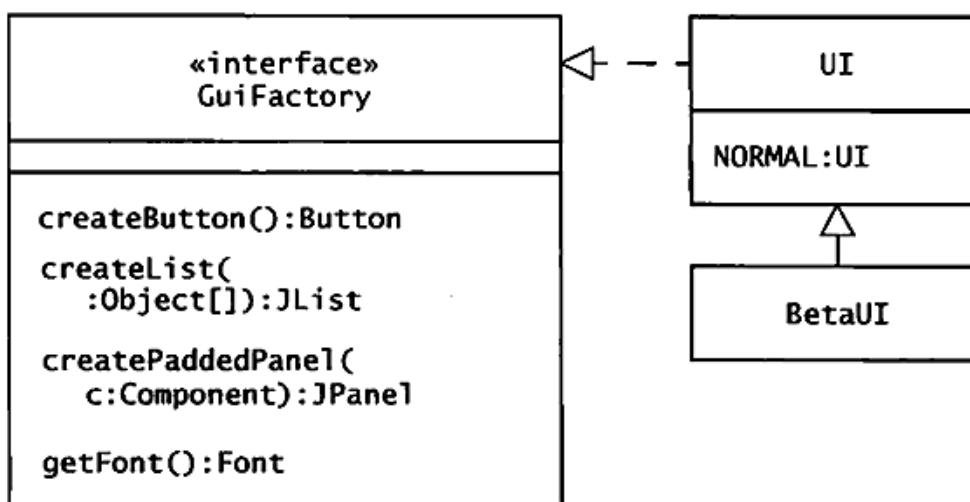


图 B.20 该图使用抽象工厂设计 GUI 控件，在 UI 类中减少了子类关于方法访问修饰符的依赖

挑战 17.3 的答案（第 155 页）

图 B.21 展示了一种解决方案，用来在 `Credit.Credit.ca` 中提供具体类以实现 `Credit` 的接口和抽象类。

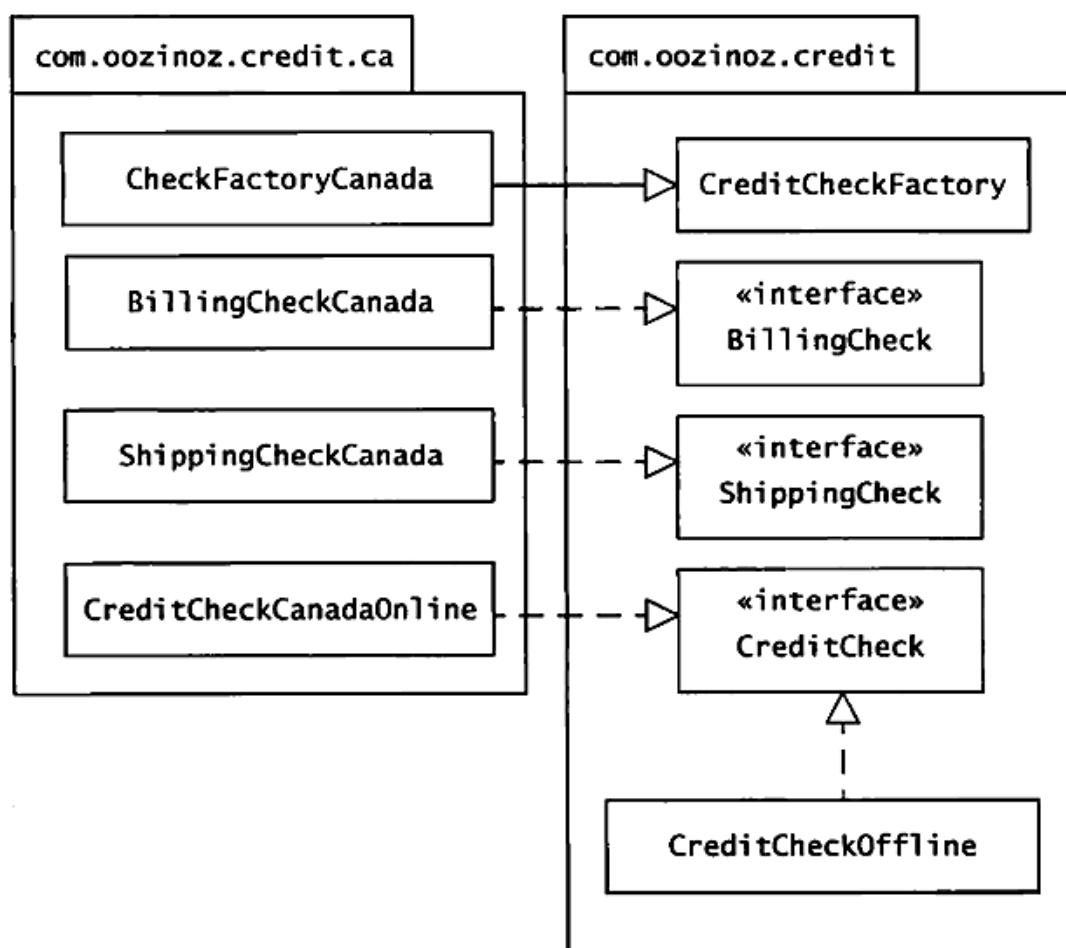


图 B.21 `com.oozinoz.credit.ca` 包提供了一组具体类，用来对加拿大来电进行各种审查

有一点需要注意的是，你只需一个具体类来进行离线电话审查。因为在 Oozinoz 公司，对来自美国和加拿大的离线电话审查是一致的。

挑战 17.4 的答案（第 156 页）

解决方案如下所示：

```
package com.oozinoz.credit.ca;
import com.oozinoz.check.*;

public class CheckFactoryCanada extends CheckFactory {
    public BillingCheck createBillingCheck() {
        return new BillingCheckCanada();
    }

    public CreditCheck createCreditCheck() {
        if (isAgencyUp())
            return new CreditCheckCanadaOnline();
        return new CreditCheckoffline();
    }

    public ShippingCheck createShippingCheck() {
        return new ShippingCheckCanada();
    }
}
```

你的方案应该满足下列要求：

- 实现继承自抽象类 `CreditCheckFactory` 的所有 `create-` 方法。
- 拥有每个 `create-` 方法返回值的合适类型。
- 如果代理离线，返回一个 `CreditCheckoffline` 对象。

挑战 17.5 的答案（第 157 页）

当前做法的好处是：将各个国家相关的类放在独立的包中，可以帮助 Oozinoz 公司的开发者来管理软件和开发成本。还可以将该包和其他包一样独立管理。我们可以很自信地说，美国相关的类不会影响到加拿大相关的类。我们也可以很容易地为新增的国家提供支持。例如，如果要开展墨西哥的业务，可以创建一个新包，来提供针对该国家所需要的审查服务。

这样做还有一个好处，就是可以将 `credit.mx` 包分配给一个有墨西哥数据服务经验的开发者来做。

当前做法的不妥之处在于：尽管理论上这种分离是很好的，但是实际上很难实施。我更加愿意把所有的类打包到一个包，直到出现 9 个或者更多的国家需要服务。当需要对包内容做出调整时，这种做法需要对每个包都进行调整，这增加了对这些包的管理任务。

第 18 章 原型（Prototype）模式

挑战 18.1 的答案（第 159 页）

这种设计的优点包括如下方面：

- 可以不创建新的类而直接创建工厂，甚至可以在运行时创建一个新的 GUI 工具箱。
- 可以通过复制一个旧的工厂，并进行轻微的调整来创建一个新的工厂。例如，可以创建一个与旧的 GUI 工具集几乎一致的新 GUI 工具集，而只存在字体上的差别。原型模式可以让新工具集从旧的工具集集成而来，比如颜色等。

缺点包括如下方面：

- 原型模式可以让我们更新属性值，比如颜色或者字体。但是却不允许为每个工厂创建不同的行为。
- 防止 UI 工具集类增长的动机并不明确，为什么这种增长是一个问题？我们必须将工具集初始化软件放在某些位置，有可能是 UI 工具类的静态方法。这种方式并没有真正减少我们需要管理的代码。

正确的答案是什么呢？类似的情况下，动手试验是很有帮助的：写出这两种设计的代码，然后再来评估这两种设计。同事们经常会在采用哪种方案上产生分歧，这是好事，实践后得出的结论更有说服力。（如果最后还是没能达成一致，你可能要引入架构师或者第三方来一起讨论。）

挑战 18.2 的答案（第 160 页）

可以这样概括 `clone()` 方法：全新的对象，但是字段相同。`clone()` 方法用原始类的类型

和字段来创建一个新的对象。新对象的字段值也都和原始类的字段值一致。如果这些字段是基本类型，比如整型，则进行值复制。但如果字段是引用类型，则进行引用复制。

`clone()`方法进行的是“浅复制”，它会在复制对象与原始对象之间共享子对象。“深复制”会包含对父对象属性的完整复制。例如，你克隆一个指向 B 对象的 A 对象，浅复制会创建一个新的 A' 对象指向原始的 B 对象，而深复制会创建一个新的 A' 对象指向新的 B' 对象。如果你想要实现一个深复制，需要自己实现复制方法。

注意，在使用 `clone()` 方法时，必须声明你的类实现 `Cloneable` 接口。这个标记接口没有方法，但是用来说明你有意支持 `clone()` 方法。

挑战 18.3 的答案（第 160 页）

这段代码将会产生三个对象，如图 B.22 所示。

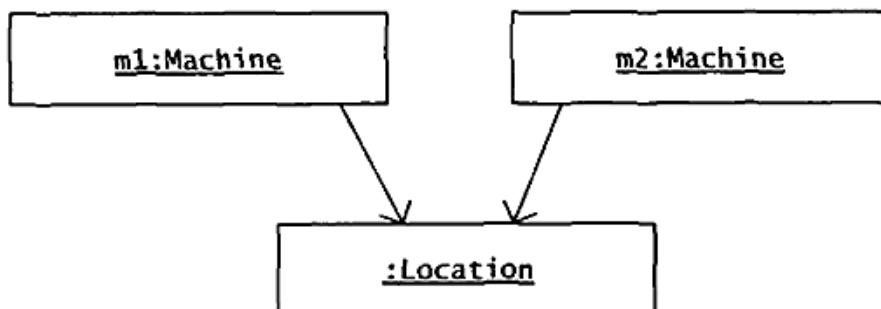


图 B.22 设计不足的克隆会创建一个不完整的复制，还是会和原始对象共享一些对象

`Machinesimulator` 类当前版本的 `clone()` 方法调用了 `super.clone()` 方法，该方法是 `Object` 实现的。这个方法用相同的属性创建了一个新的对象。基本类型，比如 `int`，都是直接复制。而对象类型，比如 `Machinesimulator` 类的 `Location` 属性，也是直接复制。注意是引用复制，而不是对象复制。`Object.clone()` 方法产生的效果如图 B.22 所示。

假设你改变了第二个机器的位置坐标和车间，因为只有一个 `Location` 对象，因此此类更改将会对第一个机器的位置造成影响。

挑战 18.4 的答案（第 162 页）

一个合理的方法如下所示：

```
public OzPanel copy2() {  
    OzPanel result = new OzPanel();
```

```
    result.setBackground(this.getBackground());
    result.setForeground(this.getForeground());
    result.setFont(this.getFont());
    return result;
}
```

copy()方法和 copy2()方法都使得 OzPanel 的客户类不需要调用构造函数，并且支持原型模式。然而，手工的 copy2()方法可能更加安全。这个方法知道需要复制哪些重要的属性，而避免了复制不清楚的属性。

第 19 章 备忘录 (Memento) 模式

挑战 19.1 的答案 (第 167 页)

下面是一个 FactoryModel 类的 undo() 方法的实现：

```
public boolean canUndo() {
    return mementos.size() > 1;
}
public void undo() {
    if (!canUndo()) return;
    mementos.pop();
    notifyListeners();
}
```

这段代码会在栈恢复到只剩一个备忘录的初始状态时，忽略 undo() 请求。栈顶始终是当前状态，因此 undo() 方法仅仅是从栈顶弹出之前的备忘录。

定义 createMemento() 方法时，必须告诉自己或者同事要求该方法返回重建对象所需全部信息。在这个例子中，机器模拟器可以在克隆过程中重建自己，而工厂模拟器可以在这些机器模拟器的克隆中重建自己。

挑战 19.2 的答案 (第 170 页)

一个方案如下：

```
public void stateChanged(ChangeEvent e) {
```

```
machinePanel().removeAll();

List locations = factoryModel.getLocations();

for (int i = 0; i < locations.size(); i++) {
    Point p = (Point) locations.get(i);
    machinePanel().add(createPictureBox(p));
}

undoButton().setEnabled(factoryModel.canUndo());
repaint();
}
```

每次状态改变时，这段代码都会从头重建 `machinePanel()` 中的机器列表。

挑战 19.3 的答案（第 170 页）

把备忘录存成一个对象的假设是，当用户想恢复原始对象时，应用程序始终是在运行的。而必须将备忘录进行持久化存储的理由如下：

- 在系统崩溃后，依然有能力恢复对象的状态。
- 程序允许用户重新进入系统后恢复之前尚未完成的任务。
- 你需要在另一台机器上重建某个对象。

挑战 19.4 的答案（第 173 页）

一个方案如下：

```
public void restore(Component source) throws Exception {
    JFileChooser dialog = new JFileChooser();
    dialog.showOpenDialog(source);

    if (dialog.getSelectedFile() == null)
        return;

    FileInputStream out = null;
    ObjectInputStream s = null;
    try {
        out = new FileInputStream(dialog.getSelectedFile());
        s = new ObjectInputStream(out);
        ArrayList list = (ArrayList) s.readObject();
    }
}
```

```
    factoryModel.setLocations(list);
} finally {
    if (s != null)
        s.close();
}
}
```

这段代码几乎就是 `save()` 方法的一个镜像方法，尽管 `restore()` 方法要求将工厂模型放入被恢复的位置列表中。

挑战 19.5 的答案（第 173 页）

封装是为了限制对对象状态以及方法的访问权限。拿工厂位置坐标集合来说，如果用文本模式来暴露对象的状态，就会造成用户可以随意修改对象状态。因此用 XML 形式来存储对象在某种程度上来说是有悖封装原则的。

在持久化存储方面违反封装原则所造成的问题，可能还需要进一步验证。而这通常取决于应用本身。为了解决这个问题，一般都会限制对对象的访问，这种方式在关系型数据库中很常见。而在某些例子中，你可能会对数据进行加密，比如传输敏感的 HTML 文本。这个问题不仅在于设计是否使用了封装或者备忘录模式，关键在于是否保证了数据的完整性。

第 20 章 操作型模式介绍

挑战 20.1 的答案（第 177 页）

职责链模式的原理是将某个操作分发在对象链中完成。对象链中的每个方法或者直接实现该操作，或者将调用转移给链中的下一个对象。

挑战 20.2 的答案（第 177 页）

下面给出了一个完整的 Java 方法修饰符的定义：

- **public**: 允许所有客户类访问。
- **protected**: 允许该类在同一个包中的子类访问。
- **private**: 只允许在该类的内部访问。

- **abstract**: 不提供实现。
- **static**: 与整个类相关，与单独的对象无关。
- **final**: 不允许被重写。
- **synchronized**: 方法会获得对对象监控器的访问权，如果方法是静态的，则将获得对类的访问权。
- **native**: 实现与平台依赖相关的代码。
- **strictfp**: 严格按照 FP 规则计算的 `double` 和 `float` 表达式，因此要求中间结果按照 IEEE 标准是有效的。

尽管一些开发者可能会在一个方法定义中使用这些修饰符，但是一些规则限制了组合使用修饰符。*Java™ Language Specification*（由 Gosling 等人在 2005 年编写）一书的第 8.4.3 小节列出了这些限制。

挑战 20.3 的答案（第 178 页）

存在如下两种情况：

在 Java 5 之前的版本中：如果你用某种方式改变了 `Bitmap.clone()` 方法的返回值，这段代码的编译将无法通过。`clone()` 签名会匹配 `Object.clone()` 的签名，因此返回类型也必须一致。

在 Java 5 中：语言本身已经支持协变返回类型，因此子类可以声明更加明确的返回类型。

挑战 20.4 的答案（第 179 页）

不在方法头声明异常的正方观点是：我们首先要注意到 Java 不要求方法声明其可能抛出的所有异常。例如任何方法都可能遇到空指针异常。Java 也承认，让程序员去声明所有可能的异常是不切实际的。应用程序需要某种策略去处理所有的异常。要求开发者声明指定的异常不是异常处理策略的最终方案。

另一方面：程序员需要获得所有可能的帮助信息。因此应用程序的架构需要一个稳定的异常处理策略。强制开发者在每个方法中声明诸如空指针类的异常也是不现实的。但是有些异常，例如打开文件时出错，此时对异常的强制要求就是有用的。C#从方法头中完全消除了程序的异常。

挑战 20.5 的答案（第 179 页）

该图展示了一种算法：用来检测对象模型是否为树形结构（包含来自 `MachineComponent`

类的两个操作) 以及 4 个方法。

第 21 章 模板方法 (Template Method) 模式

挑战 21.1 的答案 (第 185 页)

完整的程序应该与下面这段代码类似:

```
package app.templateMethod;
import java.util.Comparator;
import com.oozinoz.firework.Rocket;

public class ApogeeComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Rocket r1 = (Rocket) o1;
        Rocket r2 = (Rocket) o2;
        return Double.compare(r1.getApogee(), r2.getApogee());
    }
}
```

以及:

```
package app.templateMethod;
import java.util.Comparator;
import com.oozinoz.firework.Rocket;

public class NameComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Rocket r1 = (Rocket) o1;
        Rocket r2 = (Rocket) o2;
        return r1.toString().compareTo(r2.toString());
    }
}
```

挑战 21.2 的答案 (第 188 页)

`markMoldIncomplete()` 方法的代码将一个未完成的模具信息传递给了原材料管理器。一

种方案如下：

```
package com.oozinoz.ozAster;
import aster.*;
import com.oozinoz.businessCore.*;

public class OzAsterStarPress extends AsterStarPress {
    public MaterialManager getManager() {
        return MaterialManager.getManager();
    }

    public void markMoldIncomplete(int id) {
        getManager().setMoldIncomplete(id);
    }
}
```

挑战 21.3 的答案（第 189 页）

你需要使用的其实是一个钩子。你可能会这样描述你的请求：在机器排完药之后和开始清洗机器前，我希望能在 `shutDown()` 方法中帮我增加一个调用。如果我们把这件事写成类似 `collectPaste()` 的方法，那么它就可以被 Oozinoz 公司所复用。

开发人员会和你协商共同定出该方法的名字。问题的关键在于在一个模板方法模式中调用该方法，这样不仅可以解决我们的问题，并且使得程序更加健壮。

挑战 21.4 的答案（第 191 页）

`Machine` 类的 `getPlanner()` 方法应该充分利用抽象的 `createPlanner()` 方法：

```
public MachinePlanner getPlanner() {
    if (planner == null)
        planner = createPlanner();
    return planner;
}
```

这段代码要求为 `Machine` 类增加一个 `planner` 字段。在给 `Machine` 类增加这个字段和 `getPlanner()` 方法后，就可以在子类中删除该字段以及方法了。

这次重构创建了一个模板方法。`getPlanner()` 方法延迟初始化了 `planner` 变量，该变量的初始化由其子类的 `createPlanner()` 方法来实现。

第 22 章 状态 (State) 模式

挑战 22.1 的答案（第 194 页）

正如状态机所示，当门打开时，我们按键，门的状态会变成 StayOpen，再次按键，门将关闭。

挑战 22.2 的答案（第 197 页）

你的代码看起来应如下所示：

```
public void complete() {
    if (state == OPENING)
        setState(OPEN);
    else if (state == CLOSING)
        setState(CLOSED);
}

public void timeout() {
    setState(CLOSING);
}
```

挑战 22.3 的答案（第 201 页）

你的代码看起来应该如下所示：

```
package com.oozinoz.carousel;

public class DoorClosing extends DoorState {
    public DoorClosing(Door2 door) {
        super(door);
    }

    public void touch() {
        door.setState(door.OPENING);
    }
}
```

```

public void complete() {
    door.setState(door.CLOSED);
}
}

```

挑战 22.4 的答案（第 202 页）

图 B.23 显示了一个合理的图。

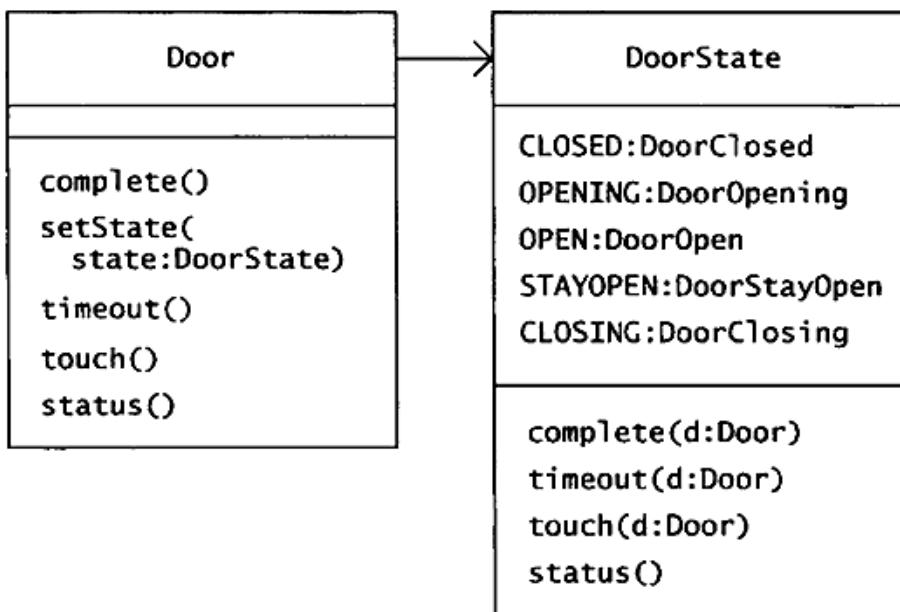


图 B.23 这个设计将 DoorState 对象设置成了常量。DoorState 状态迁移方法会更新 Door 对象的状态，并且将 Door 对象作为方法传递的参数

第 23 章 策略（Strategy）模式

挑战 23.1 的答案（第 207 页）

图 B.24 展示了一种解决方案。

挑战 23.2 的答案（第 209 页）

`GroupAdvisor` 类和 `ItemAdvisor` 类都是 `Adapter` 类的实例，提供了客户期望的接口，并且利用了不同的接口来使用类的服务。

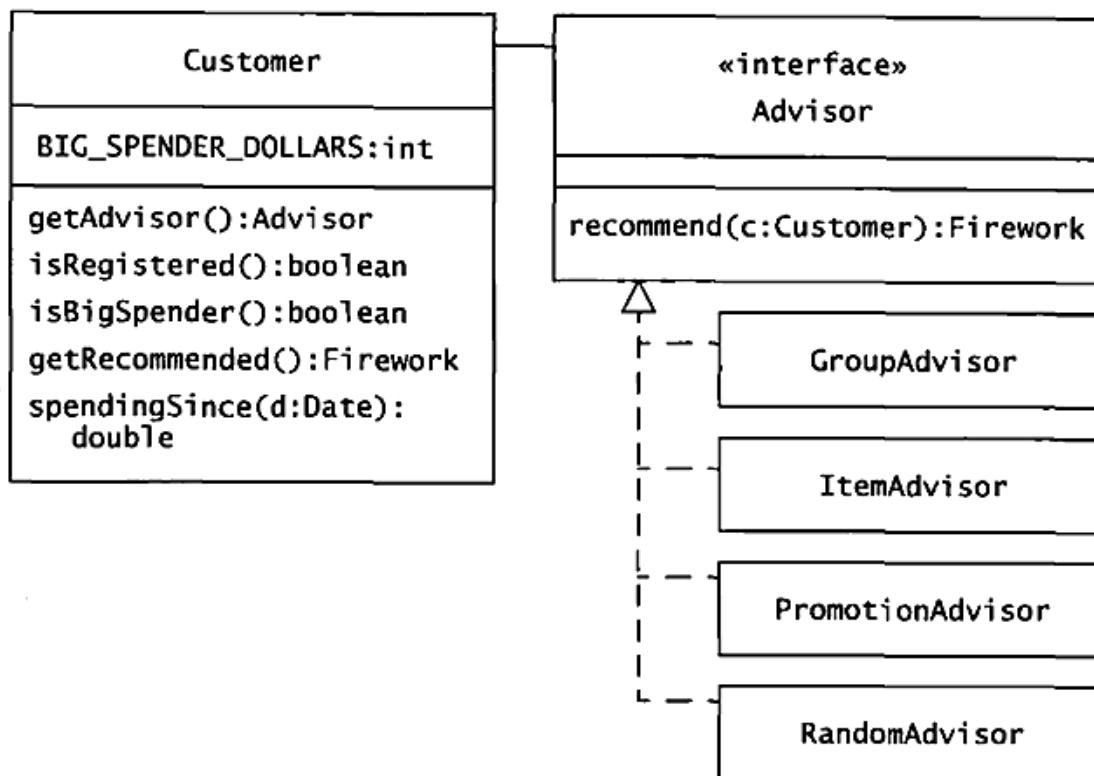


图 B.24 Oozinoz 公司的广告策略包含 4 个策略，这些策略都实现了 Advisor 接口

挑战 23.3 的答案（第 210 页）

你的代码应该看起来如下所示：

```

public Firework getRecommended() {
    return getAdvisor().recommend(this);
}
  
```

一旦知道了 advisor 对象，就可以使用多态来实现了。

挑战 23.4 的答案（第 212 页）

一个可重用的排序例子用的是模板方法模式还是策略模式？

使用策略模式：按照 *Design Patterns* 一书中的说法，模板方法的用意是让“子类”重新定义算法的具体执行步骤。但是 `Collections.sort()` 方法没有使用子类，而用的是一个 `Comparator` 实例。`Comparator` 类的每个实例都提供一个新的方法，因此也提供了一个新的算法和新的策略。所以，`sort()` 方法是策略模式的一个很好的例子。

使用模板方法模式：排序算法有很多，但是 `Collections.sort()` 仅仅用的是其中之一（快

速排序）。改变算法意味着将算法变成堆排序或者冒泡排序。策略模式的意图是让你使用多种算法，而在本例中没有体现出来。而模板方法模式的意图在于能让你将某一个步骤插入到特定算法中，而这正是 `sort()` 方法做的事情。

第 24 章 命令 (Command) 模式

挑战 24.1 的答案（第 214 页）

很多 Java Swing 程序都使用了调停者模式，注册了一个单独的对象来接收所有的 GUI 事件。该对象负责协调组件间的交互，并且将用户输入转换成控制领域对象的命令。

挑战 24.2 的答案（第 215 页）

你的代码看起来应该如下所示：

```
package com.oozinoz.visualization;
import java.awt.event.*;
import javax.swing.*;
import com.oozinoz.ui.*;

public class Visualization2 extends Visualization {
    public static void main(String[] args) {
        Visualization2 panel = new Visualization2(UI.NORMAL);
        JFrame frame = SwingFacade.launch(
            panel,
            "Operational Model");
        frame.setJMenuBar(panel.menus());
        frame.setVisible(true);
    }
    public Visualization2(UI ui) {
        super(ui);
    }
    public JMenuBar menus() {
        JMenuBar menuBar = new JMenuBar();
        JMenu menu = new JMenu("File");
        menuBar.add(menu);
        return menuBar;
    }
}
```

```
menuBar.add(menu);

JMenuItem menuItem = new JMenuItem("Save As...");
menuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        save();
    }
});
menu.add(menuItem);

menuItem = new JMenuItem("Restore From...");
menuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        restore();
    }
});
menu.add(menuItem);
return menuBar;
}

public void save() {
    try {
        mediator.save(this);
    } catch (Exception ex) {
        System.out.println("Failed save: " +
            ex.getMessage());
    }
}

public void restore() {
    try {
        mediator.restore(this);
    } catch (Exception ex) {
        System.out.println("Failed restore: " +
            ex.getMessage());
    }
}
}
```

尽管 actionPerformed()方法需要一个 ActionEvent 参数，你依然可以安全地忽略它。menus()方法为 Save 菜单和 Load 菜单都分别注册了一个匿名类的实例。当这些方法被调用时，我们也会很清楚事件的来源。

挑战 24.3 的答案(第 217 页)

testSleep()方法将 doze 命令传递给了工具类的 time()方法：

```
package app.command;

import com.oozinoz.robotInterpreter.Command;
import com.oozinoz.utility.CommandTimer;

import junit.framework.TestCase;
public class TestCommandTimer extends TestCase {
    public void testSleep() {
        Command doze = new Command() {
            public void execute() {
                try {
                    Thread.sleep(
                        2000 + Math.round(10 * Math.random()));
                } catch (InterruptedException ignored) {
                }
            }
        };
        long actual = CommandTimer.time(doze);

        long expected = 2000;
        long delta = 5;
        assertTrue(
            "Should be " + expected + " +/- " + delta + " ms",
            expected - delta <= actual
            && actual <= expected + delta);
    }
}
```

挑战 24.4 的答案（第 219 页）

你的代码应该看起来如下所示：

```
public void shutDown() {
    if (inProcess()) {
        stopProcessing();
        moldIncompleteHook.execute(this);
    }
    usherInputMolds();
    dischargePaste();
    flush();
}
```

注意，上面的代码并没有干扰对 `moldIncompleteHook` 的非空检查，因为它总是会被设置到一个真实的 Hook 对象上（初始情况下，它会被设置到一个什么都不做的 `NullHook` 对象上，但是用户可以设置到不同的钩子）。

你可能会这样使用它：

```
package app.templateMethod;

import com.oozinoz.businessCore.*;
import aster2.*;

public class ShowHook {
    public static void main(String[] args) {
        AsterStarPress p = new AsterStarPress();
        Hook h = new Hook() {
            public void execute(AsterStarPress p) {
                MaterialManager m = MaterialManager.getManager();
                m.setMoldIncomplete(p.getCurrentMoldID());
            }
        };
        p.setMoldIncompleteHook(h);
    }
}
```

挑战 24.5 的答案 (第 219 页)

在工厂方法模式下，客户类知道何时去创建一个新对象，但是不知道创建什么类型的对象。工厂方法模式将对象的创建移到了一个方法中，从而使用户不必关心应该去创建什么样的对象。这种方式也出现在抽象工厂模式中。

挑战 24.6 的答案 (第 220 页)

备忘录模式的意图是提供存储以及恢复对象的状态。通常来说，每当执行一个命令的时候，都可以将其加入到一个栈中，当需要恢复操作时，从栈中弹出这些备忘录并进行恢复即可。

第 25 章 解释器 (Interpreter) 模式

挑战 25.1 的答案 (第 228 页)

ForCommand 类的 execute() 方法的实现应该如下所示：

```
private void execute(MachineComponent mc) {
    if (mc instanceof Machine) {
        Machine m = (Machine) mc;
        variable.assign(new Constant(m));
        body.execute();
        return;
    }

    MachineComposite comp = (MachineComposite) mc;
    List children = comp.getComponents();
    for (int i = 0; i < children.size(); i++) {
        MachineComponent child =
            (MachineComponent) children.get(i);
        execute(child);
    }
}
```

execute() 方法遍历了整个机器组合结构。在遍历过程中如果遇到叶子（机器）节点时，

会将该机器分配给变量，并执行 FormMachine 对象的方法体。

挑战 25.2 的答案（第 232 页）

一种方案如下所示：

```
public void execute() {
    if (term.eval() != null)
        body.execute();
    else
        elseBody.execute();
}
```

挑战 25.3 的答案（第 232 页）

WhileCommand.java 的一种实现方式，如下所示：

```
package com.oozinoz.robotInterpreter2;

public class WhileCommand extends Command {
    protected Term term;

    protected Command body;

    public WhileCommand(Term term, Command body) {
        this.term = term;
        this.body = body;
    }

    public void execute() {
        while (term.eval() != null)
            body.execute();
    }
}
```

挑战 25.4 的答案（第 233 页）

一种答案是：解释器模式的意图是让你从类层级结构中组合可执行的对象，从而为某些公共操作提供各种解释行为。命令模式的意图仅仅是用来封装对象的请求。

解释器对象可否有命令对象的功能？当然可以！使用哪种模式取决于你的意图。你是想创建组合可执行对象的工具集，还是想封装对象的请求？

第 26 章 扩展型模式介绍

挑战 26.1 的答案（第 238 页）

从数学的角度来讲，圆是椭圆的一种特殊情况。然而在面向对象编程中，椭圆有一些圆所没有的行为。例如，椭圆的宽可能是高的两倍，而圆不可能是这样。如果这样的行为对你的程序很重要，椭圆就不可能是圆的对象，这将会违背 LSP 原则。

注意，如果你要用不可变对象，那么这样做未必适合。这只是一个简单的数学问题，而不是标准的类型层次结构。

挑战 26.2 的答案（第 239 页）

如果 `tub` 对象的 `Location` 属性有任何变化，`tub.getLocation().isUp()` 表达式可能会导致程序错误。例如，在箱子传输过程中，`location` 可能是 `null` 或者是机器人对象。此时计算 `tub.getLocation().isUp()` 的值可能会抛出异常。如果 `location` 是一个机器人对象，问题可能会变得更加糟糕，因为我们让机器人从自身收集原材料。这些潜在的问题是可以管理的，但是难道我们希望这些管理代码写在 `tub.getLocation().isUp()` 方法中吗？显然不是。这些代码可能在 `Tub` 类中都实现了。如果没有，就应该在 `Tub` 类中添加相关代码，否则就会导致我们在其他地方多次重复写这些代码。

挑战 26.3 的答案（第 240 页）

一个例子如下：

```
public static String getZip(String address) {  
    return address.substring(address.length() - 5);  
}
```

这段代码有一些坏味道，包括基本类型偏执（primitive obsession）（指一个字符串包括很多的属性）。

挑战 26.4 的答案（第 240 页）

一组解决方案如表 B.2 所示。

表 B.2 所使用的模式

| 例 子 | 使用的模式 |
|--|--------|
| 某个焰火模拟器的设计者设计了一个接口，要求所建立的对象必须实现该接口，以便于参与模拟 | 适配器模式 |
| 允许在运行时组合可执行对象的工具集 | 解释器模式 |
| 父类提供了一个方法，要求子类来实现其中的一些步骤 | 模板方法模式 |
| 一个对象允许扩展它的行为，通过对象中封装的方法，并且在合适的时机来调用 | 命令模式 |
| 通过代码生成器来插入某些行为，以便形成模拟对象在本机执行的假象 | 代理模式 |
| 该设计允许注册一些回调方法，并且在对象发生变化时触发 | 观察者模式 |
| 该设计允许提供已经定义好接口的抽象操作，并且能够添加实现该接口的新驱动器 | 桥接模式 |

第 27 章 装饰器（Decorator）模式

挑战 27.1 的答案（第 248 页）

一种解决方案如下所示：

```
package com.oozinoz.filter;
import java.io.*;

public class RandomCaseFilter extends OozinozFilter {
    public RandomCaseFilter(Writer out) {
        super(out);
    }

    public void write(int c) throws IOException {
        out.write(Math.random() < .5
            ? Character.toUpperCase(c)
            : Character.toLowerCase(c));
    }
}
```

```
    ? Character.toLowerCase((char) c)
    : Character.toUpperCase((char) c));
}
}
```

随机大小写问题也非常有趣，看看下面的程序：

```
package app.decorator;
import java.io.BufferedWriter;
import java.io.IOException;

import com.oozinoz.filter.ConsoleWriter;
import com.oozinoz.filter.RandomCaseFilter;

public class ShowRandom {
    public static void main(String[] args)
        throws IOException {
        BufferedWriter w =
            new BufferedWriter(
                new RandomCaseFilter(new ConsoleWriter()));
        w.write("buy two packs now and get a "
            + "zippie pocket rocket -- free!");
        w.newLine();
        w.close();
    }
}
```

这段程序使用了第27章定义的 `ConsoleWriter` 类。运行该程序将产生如下输出：

```
buY tWO pACKS NOW AND gET A ZiPPIE PoCkEt RoCKEt -- frEE!
```

挑战 27.2 的答案(第 250 页)

一种方案如下所示：

```
package com.oozinoz.filter;
import java.io.Writer;

public class ConsoleWriter extends Writer {
    public void close() {}
    public void flush() {}
```

```
public void write(
    char[] buffer, int offset, int length) {
    for (int i = 0; i < length; i++)
        System.out.print(buffer[offset + i]);
}
```

挑战 27.3 的答案（第 256 页）

一种方案如下所示：

```
package com.oozinoz.function;

public class Exp extends Function {
    public Exp(Function f) {
        super(f);
    }

    public double f(double t) {
        return Math.exp(sources[0].f(t));
    }
}
```

挑战 27.4 的答案（第 257 页）

一种方案如下所示：

```
package app.decorator.brightness;

import com.oozinoz.function.Function;

public class Brightness extends Function {
    public Brightness(Function f) {
        super(f);
    }

    public double f(double t) {
        return Math.exp(-4 * sources[0].f(t))
            * Math.sin(Math.PI * sources[0].f(t));
    }
}
```

第 28 章 迭代器（Iterator）模式

挑战 28.1 的答案（第 265 页）

`display()`程序启动了一个线程，该线程可以在任何时候被唤醒。尽管 `sleep()`方法的调用保证了当 `display()` 正在休眠时，`run()` 方法才会被调用。输出结果表明了当程序运行时，`display()` 方法完整地运行了一个迭代，并且打印出了列表中索引 0 的数据项：

```
Mixer1201
```

此时，第二个线程被唤醒，并且将 `Fuser1101` 放到了列表的开头，并且将其他的机器名向下移动了一位。举例来说，`Mixer1201` 从索引 0 的位置移到了索引 1 的位置。

当主线程获得了控制权，`display()` 方法将打印出列表中剩下的部分，从索引 1 到结束。

```
Mixer1201
ShellAssembler1301
StarPress1401
UnloadBuffer1501
```

挑战 28.2 的答案（第 267 页）

反对使用 `synchronized()` 方法的理由是：当用 `for` 循环进行迭代时，`synchronized()` 方法会出错，甚至会使整个程序崩溃，除非你捕获了 `InvalidOperationException` 异常并写了相关的处理逻辑。

反对使用锁机制的理由是：线程安全的迭代设计依赖于访问集合线程间的协作。`synchronized()` 方法的关键点是捕获线程没有正常合作的情况。

`synchronized()` 方法和 Java 内置的锁机制，都不会简化多线程的开发。如果要进一步学习并发编程，请参考 *Concurrent Programming in Java™*（由 Lea 在 2000 年编写）这本书。

挑战 28.3 的答案（第 272 页）

正如 16 章所提到的，迭代器是工厂方法模式的一个经典例子。如果用户想要创建

`ProcessComponent` 类的迭代器，则必须知道创建迭代器的时机，接收类则需要知道实例化哪个类。

挑战 28.4 的答案（第 277 页）

一个方案如下所示：

```
public Object next() {
    if (peek != null) {
        Object result = peek;
        peek = null;
        return result;
    }

    if (!visited.contains(head)) {
        visited.add(head);
        if (shouldShowInterior()) return head;
    }

    return nextDescendant();
}
```

第 29 章 访问者（Visitor）模式

挑战 29.1 的答案（第 280 页）

区别在于 `this` 对象的类型不同。`accept()`方法调用 `MachineVisitor` 对象的 `visit()`方法，在 `Machine` 类内，`accept()`方法将会使用 `visit(:Machine)` 签名来查询 `visit()` 方法。而在 `MachineComposite` 类内，`accept()`方法将会使用 `visit(:MachineComposite)` 签名来查询 `visit()` 方法。

挑战 29.2 的答案（第 283 页）

一种方案如下所示：

```
package app.visitor;
```

```
import com.oozinoz.machine.MachineComponent;
import com.oozinoz.machine.OozinozFactory;

public class ShowFindVisitor {
    public static void main(String[] args) {
        MachineComponent factory = OozinozFactory.dublin();
        MachineComponent machine = new FindVisitor().find(
            factory, 3404);
        System.out.println(machine != null ?
            machine.toString() : "Not found");
    }
}
```

挑战 29.3 的答案(第 285 页)

一种方案如下所示:

```
package app.visitor;
import com.oozinoz.machine.*;
import java.util.*;

public class RakeVisitor implements MachineVisitor {
    private Set leaves;

    public Set getLeaves(MachineComponent mc) {
        leaves = new HashSet();
        mc.accept(this);
        return leaves;
    }

    public void visit(Machine m) {
        leaves.add(m);
    }

    public void visit(MachineComposite mc) {
        Iterator iter = mc.getComponents().iterator();
        while (iter.hasNext())
            ((MachineComponent) iter.next()).accept(this);
    }
}
```

挑战 29.4 的答案（第 290 页）

一个解决方案是给所有的 `accept()` 方法和 `visit()` 方法都增加一个 `set` 参数，这样就可以传递访问过的节点集合。`ProcessComponent` 类可以通过给 `accept()` 方法传递一个 `set` 对象来调用其抽象的 `accept()` 方法。

```
public void accept(ProcessVisitor v) {  
    accept(v, new HashSet());  
}
```

`ProcessAlternation`、`ProcessSequence` 和 `ProcessStep` 子类的 `accept()` 方法应该为：

```
public void accept(ProcessVisitor v, Set visited) {  
    v.visit(this, visited);  
}
```

现在访问者的开发人员必须创建包含 `visit()` 方法的类，其中 `visit()` 方法需要能接收 `visited` 集合。很明显，这时候非常适合使用 `set` 作为集合。尽管该访问者的开发人员需要公开对该集合的访问。

挑战 29.5 的答案（第 291 页）

访问者模式有几种替代方案：

- 将需要的行为加入到原来的类层级结构中。可以通过两种方式达到该目的：说服该类层级结构的开发者加入你所需要的代码；在代码集体所有制的前提下，我们可以更改该代码。
- 你可以让操作机器或者流程结构的类直接遍历该层次结构。如果想知道组合结构中子对象的类型，可以使用 `instanceof` 运算符，或者写一个返回 `Boolean` 类型的函数，例如 `isLeaf()` 或者 `isComposite()`。
- 如果想让增加的行为和现有的行为有很明显的差别，需要创建一个并行的类层次结构。例如工厂方法模式中的 `MachinePlanner` 类，其机器的行为就放在独立的层次结构中。

附录 C

Oozinoz 源代码

学习设计模式的最大好处在于理解了设计模式，可以帮助我们改进代码。设计模式能够使你的代码更短小、简洁、优雅、易于维护，并且更加强大。为了更好地运用设计模式，需要在编码时时刻牢记这些模式，熟练地在代码库中构建以及重新构建设计模式。从一个可以工作的范例开始可以帮助你的学习，本书提供了许多 Java 代码范例，展示了如何运用设计模式。阅读本书时，可以构建 Oozinoz 源代码，了解整个代码范例，有助于你更好地在自己的代码中运用设计模式。

获取与使用源代码

若要获取本书提供的源代码，可以访问 www.oozinoz.com，下载 zip 格式的文件，并将其解压缩到你希望放到的地方。Oozinoz 的源代码是免费的，你可以随意使用，唯一的限制就是你不可声称这是你自己编写的。另一方面，无论是我们还是出版社都不能将这些代码用于特殊目的。

构建 Oozinoz 代码

如果没有 Java 程序的开发环境，那么需要搭建一个，并了解使用它的一些技巧，这样就能够编写、编译以及执行自己的程序。你可以购买一个开发工具，例如 IntelliJ Idea，或者使用开源工具，如 Eclipse。

使用 JUnit 对代码进行测试

Oozinoz 库包含了一个 `com.oozinoz.testing` 包，这个包使用了免费的自动测试框架 JUnit。你可以从 <http://junit.org> 下载 JUnit。如果不熟悉 JUnit，最好的办法就是求助于朋友或同事。除此之外，你还可以阅读工具的联机文档，或者翻阅相关书籍。学习 JUnit 比 Ant 更难一些，但是学会这样的自动化测试框架，对你未来的工作很有帮助。

自己查找文件

从 Oozinoz 类库中查找本书特定代码所在的文件比较困难。通常，查找特定应用程序的最简单方式是根据名称进行搜索。Oozinoz 代码经过了组织，所以可以通过查看 `oozinoz` 目录树找到文件。表 C.1 展示了 `oozinoz` 的子目录，并解释了每个目录的内容。

表 C.1 `oozinoz` 目录中子目录的名称与内容

| 目 录 名 | 目 录 内 容 |
|---------------------|--|
| <code>app</code> | “应用程序”的子目录：经过编译的 Java 可执行文件。通常，都是按照章节组织的，例如， <code>app.decorator</code> 包含了第 27 章的示例代码 |
| <code>aster</code> | 来自一个假想公司 Aster 的源代码 |
| <code>com</code> | Oozinoz 应用程序的源代码 |
| <code>images</code> | Oozinoz 应用程序使用的各种图像文件 |

小结

随着你对设计模式的学习，可能会逐渐改变你编写代码与重构代码的方式。你或许能够立即将设计模式运用到你自己的代码中，有时也有助于你理解别人编写的代码。在自己机器上运行 Oozinoz 代码是一个很好的练习，就像学习使用开源工具如 Ant 和 JUnit 一样。学习这些工具，以及运行 Oozinoz 代码或其他代码比较困难，但它们有助于提高你使用新技术的技能。祝你好运！如果你在学习本书时遇到问题，欢迎和我们联系。

Steve Metsker (Steve.Metsker@acm.org)

William Wake (William.Wake@acm.org)

附录 D

UML 概览

本附录简要介绍了本书用到的统一建模语言（Unified Modeling Language，UML）的特性。UML 提供了一套规范的表示方法，可以很好地阐释面向对象系统的设计思想。UML 并不复杂，但不能低估这些特性所包含的丰富内涵。若要快速地了解 UML 的特性，可以阅读 *UML Distilled*（由 Fowler 和 Scott 在 2003 年编写）一书。若要深入了解，则可以阅读 *The Unified Modeling Language User Guide*（由 Booch、Rumbaugh 和 Jacobson 在 1999 年编写）一书。通过学习标准的命名方法与表示方法，我们可以更好地在设计层面上进行交流，从而提高团队的生产效率。

类

图 D.1 运用了 UML 的某些特性描述类。

如下是对该类图的说明。

- 一个矩形表示一个类，类名居中。图 D.1 中显示了三个类：`Firework`、`Rocket` 与 `simulation.Rocketsim`。
- UML 并不要求在图中描述所有与该元素相关的内容，例如类的所有方法或包中包含的完整内容。

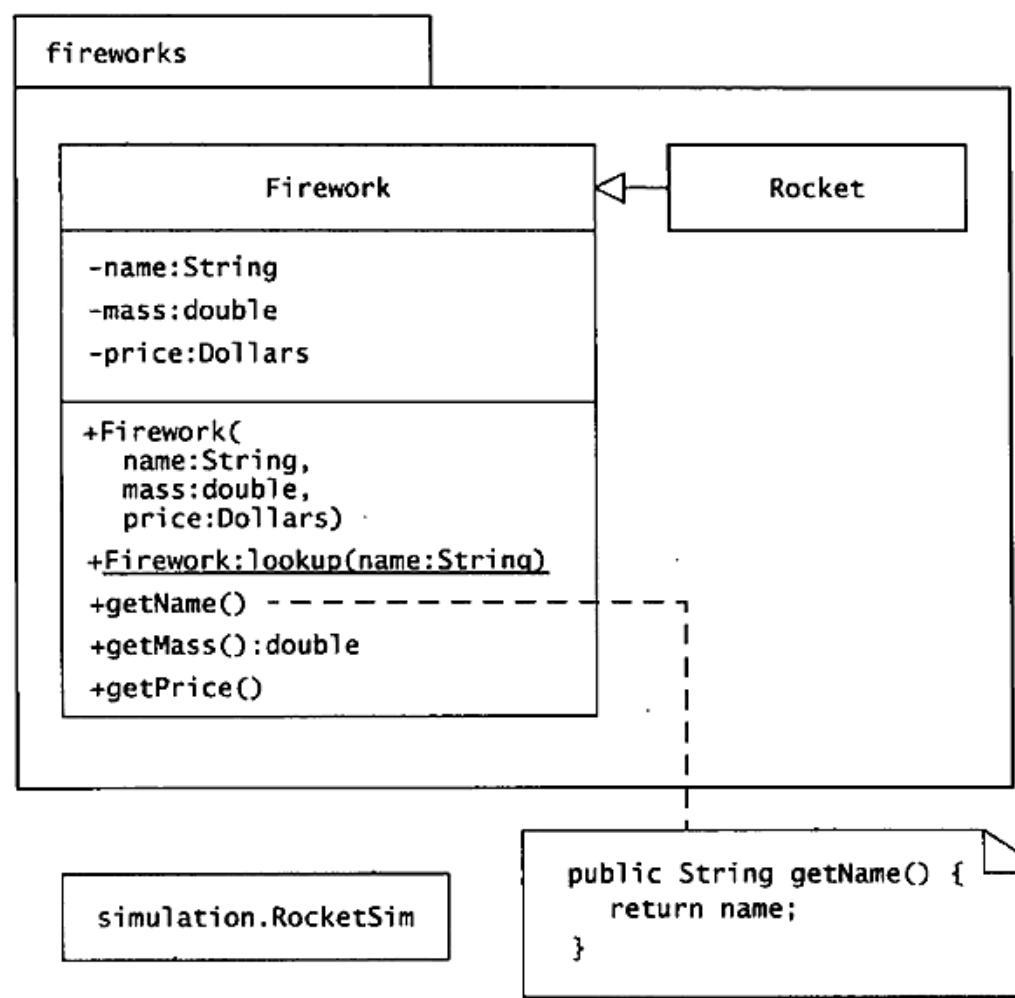


图 D.1 包含了 `Firework` 与 `Rocket` 类的 `Fireworks` 包

- 包也使用矩形表示，但包名是左对齐的，其下有更大的框，显示了这个包的类与其他类型。例如图 D.1 中所示的 `Firework` 包中的类。
- 当包图下显示一个类时，说明该类所在的命名空间。例如图 D.1 说明 `RocketSim` 类位于 `Simulation` 包中。
- 类的实例变量显示在类名下的矩形框中。`Firework` 类中包含 `name`、`price`、`mass` 等实例变量。实例变量名称后跟一个冒号，以及该实例变量的类型。
- 实例变量或方法前的减号（-），说明该变量或方法的可见性是 `private`；加号（+）说明可见性为 `public`；#号说明可见性为 `protected`。
- 类的方法显示在类名下的第二个矩形框中。
- 如果方法具有参数，通常应该将这些参数显示出来，请参见图中的 `lookup()` 方法。
- 方法签名中的变量通常被组织为这样的形式：变量名称后加冒号，再加变量的类型。如果该变量的类型能说明变量的功能，也可以省略或缩写变量名。
- 在一个实例变量或方法的名称下加下画线，表示该变量或方法是静态的，如图 D.1 中的

`lookup()`方法，它就是静态方法。

- 可以在一个卷角的矩形框中加入一段注解。其中的文字可以是注释、约束或代码，并使用虚线将它与图中的其他元素连起来。注释可以出现在任何一个 UML 图中。

类关系

图 D.2 显示了用于对类关系进行建模的 UML 特性。

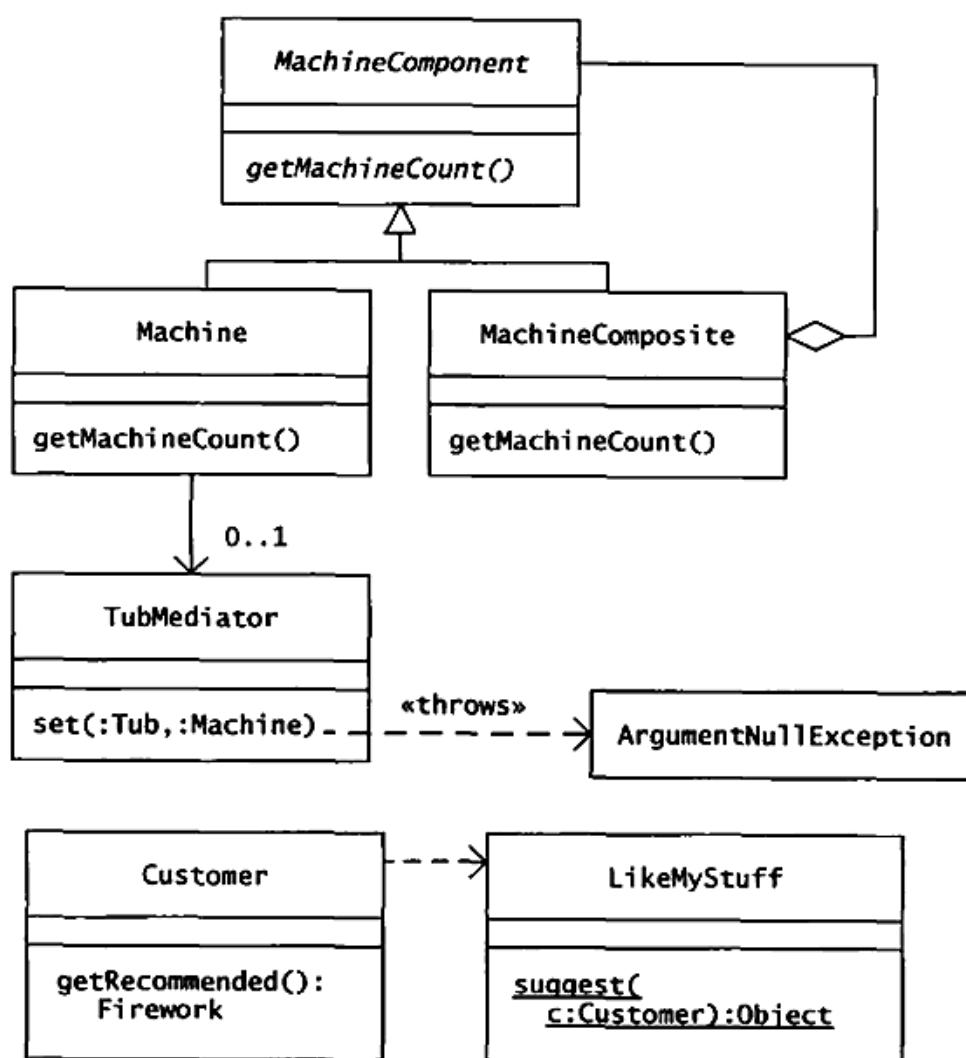


图 D.2 包含了 **Machine** 对象或其他组合对象的 **MachineComposite** 对象。**Customer** 类依赖于 **LikeMyStuff** 类，但 **Customer** 类不需要实例化它。

下面是类关系表示方法的说明。

- 显示的类名或方法名若为斜体，则表示该类或方法是抽象的。方法名下有下画线则表明它是静态的。
- 使用一个闭合的空心箭头指向类的超类。
- 两个类之间的连线表示二者的实例存在关联关系。通常，类图中的连线表示一个类中实例变量引用了另一个类。例如，Machine 类使用实例变量引用了 `TubMediator` 对象。
- 连线一端的菱形表示一个类的实例包含另一个类的实例的集合。
- 一个开放箭头表示某种引用关系，通常指一个类引用了另一个类，且被指向的类并没有反向引用前一个类。
- 多重指示符，例如 `0..1`，用于指示对象之间可能出现的连接数。星号 (*) 表示一个类有零个到多个实例与对应类的实例存在关联关系。
- 如果一个方法会抛出异常，可以从该方法引出一个带开放箭头的虚线执行这个异常类，并使用`<<throws>>`构造型标记这个箭头。
- 可以用一个带箭头的虚线表示类之间不存在对象引用的依赖关系。例如，图中的 `Customer` 类使用了 `LikeMyStuff` 的静态方法 `suggest()`，而不是通过对象引用。

接口

图 D.3 显示了接口的基本特性。

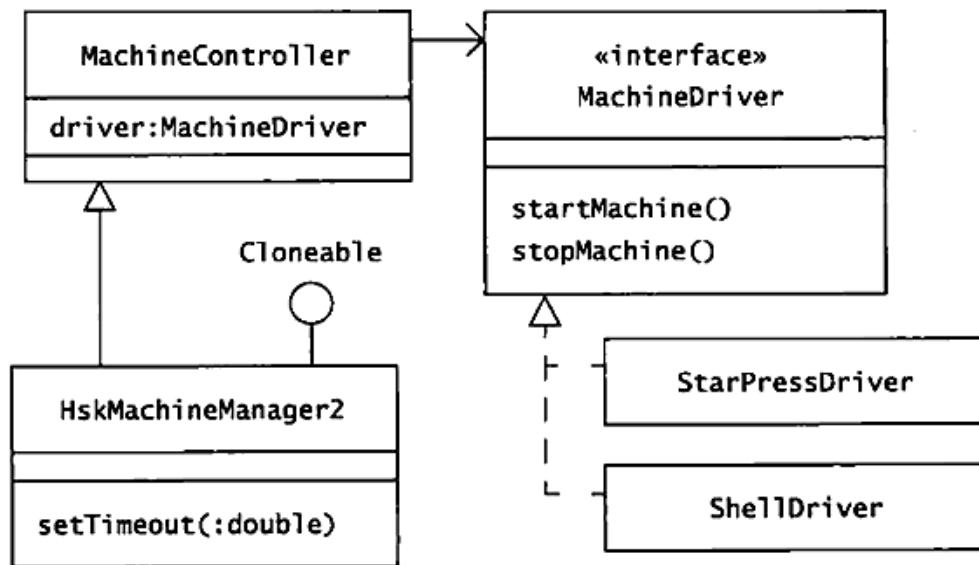


图 D.3 你可以用《interface》构造型棒棒糖图示来表示接口

如下是对接口的说明。

- 如图 D.3 所示，可以用一个包含了<<interface>>文字与接口名的矩形框来表示接口。可以使用带闭合空心箭头的虚线表示类实现了接口。
- 还可以使用一条线加一个圆形（棒棒糖形状），并在旁边标注接口名称来表示类实现了接口。
- 在 Java 语言中，接口与它的方法都是抽象的。奇怪的是，与抽象类和抽象方法不同，接口与它的方法并非斜体。

对象

对象图表示了类的特定实例，如图 D.4 所示。

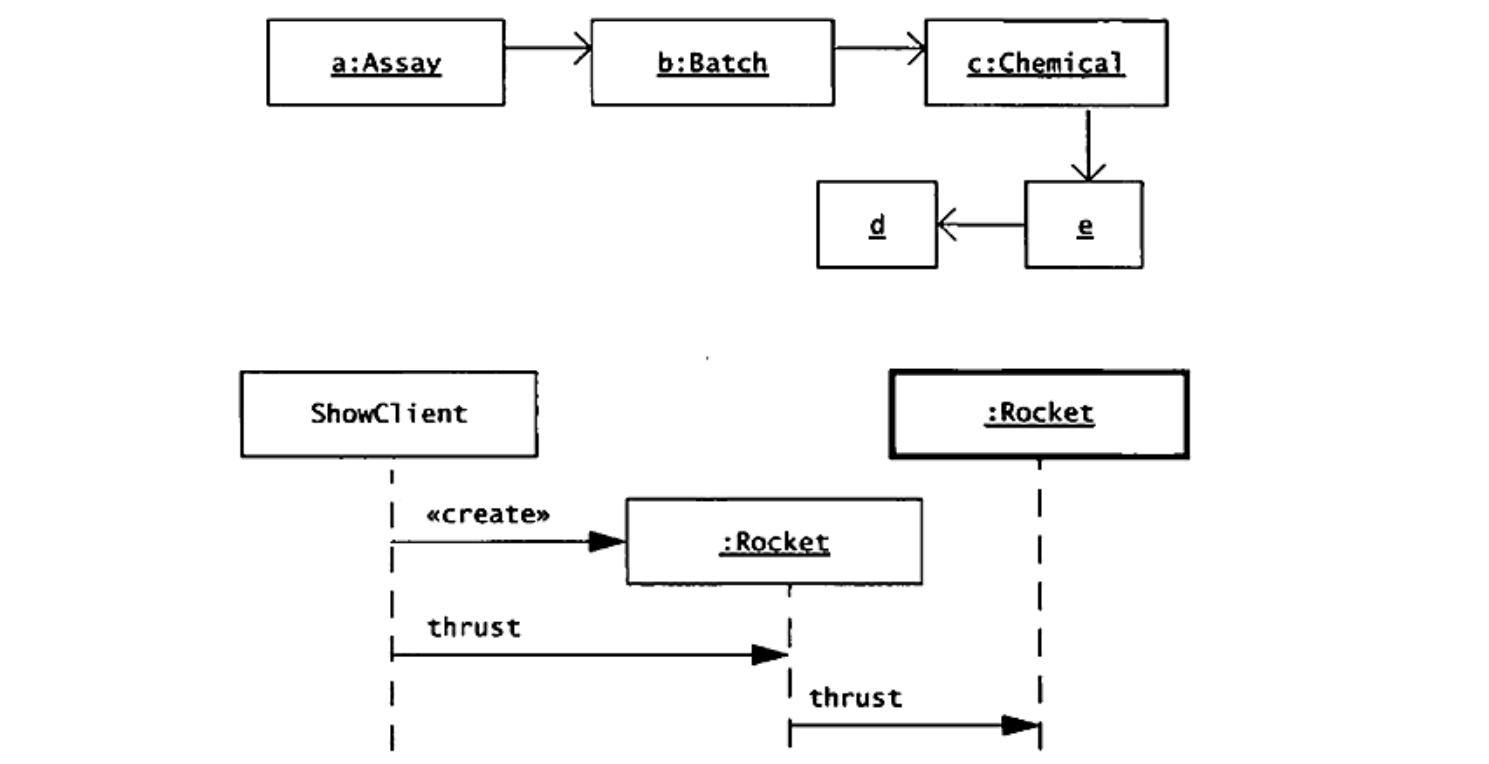


图 D.4 描述对象通常会指定对象的名称和/或对象的类型。时序图描述了连续的方法调用

如下是对对象图的说明。

- 可以通过冒号分隔的对象名与类型来描述一个对象，也可以只显示对象的名称，或者只显示冒号和类型。无论哪种情况，对象的名称与类型都要加下画线。

- 在对象之间用一条线表示一个对象引用了另一个对象。可以使用开放箭头来重点标注引用的方向。
- 可以使用图 D.4 下方的时序图表示对象发送消息给另一个对象的顺序。消息的顺序自上而下，虚线表示对象的时间区间。
- 使用<<create>>构造型表示一个对象创建了另一个对象。图 D.4 说明了 `ShowClient` 类创建了一个本地的 `Rocket` 对象。
- 可以通过为对象的矩形边框加粗表明该对象是在另一个线程、进程或另一台计算机上运行的活跃对象。图 D.4 显示了一个本地 `Rocket` 对象将 `thrust()` 调用请求转发给运行在服务器上的 `Rocket` 对象。

状态

图 D.5 显示了 UML 的状态图。

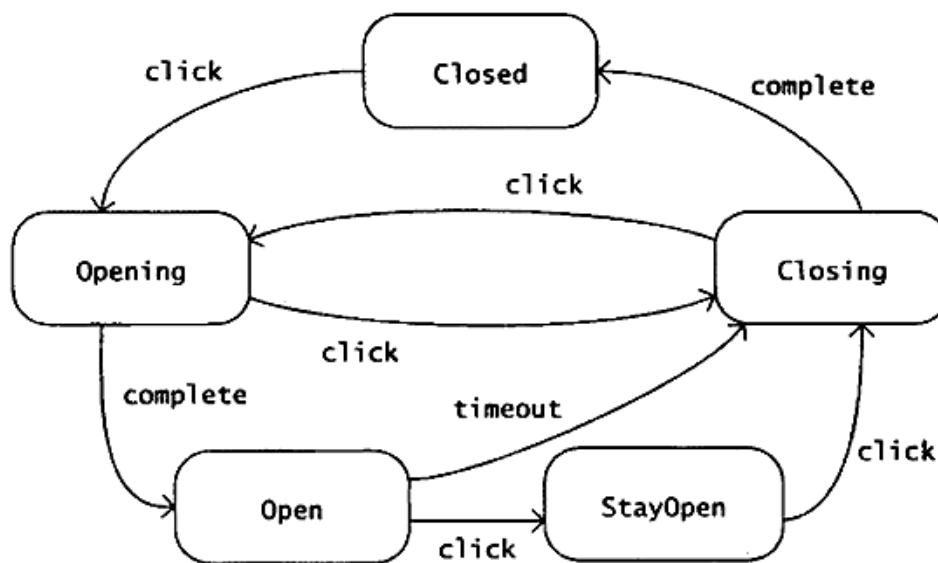


图 D.5 表示状态之间迁移的状态图

如下是对状态图的说明。

- 用带圆角的矩形框表示一种状态。
- 使用开放箭头表示状态的迁移。
- 状态图无须直接映射到类图或对象图，当然，也可以对状态图进行这样的转换，如第 22 章的图 22.3 所示。



参考文献

- Alexander, Christopher. 1979. *The Timeless Way of Building*. Oxford, England: Oxford University Press.
- Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford, England: Oxford University Press.
- Arnold, Ken, and James Gosling. 1998. *The Java™ Programming Language, Second Edition*. Reading, MA: Addison-Wesley.
- Booch, Grady, James Rumbaugh, and Ivar Jacobsen. 1999. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.
- Buschmann, Frank, et al. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, West Sussex, England: John Wiley & Sons.
- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms*. Cambridge, MA: MIT Press.
- Cunningham, Ward, ed. The Portland Patterns Repository. www.c2.com.
- Flanagan, David. 2005. *Java™ in a Nutshell*, 5th ed. Sebastopol, CA: O'Reilly & Associates.
- Flanagan, David, Jim Farley, William Crawford, and Kris Magnusson. 2002. *Java™ Enterprise in a Nutshell*, 2d ed. Sebastopol, CA: O'Reilly & Associates.

- Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.
- Fowler, Martin, with Kendall Scott. 2003. *UML Distilled, Third Edition*. Boston, MA: Addison-Wesley.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. 1995. Boston, MA: Addison-Wesley.
- Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. 2005. *The Java™ Language Specification, Third Edition*. Boston, MA: Addison-Wesley.
- Kerievsky, Joshua. 2005. *Refactoring to Patterns*. Boston, MA: Addison-Wesley.
- Lea, Doug. 2000. *Concurrent Programming in Java™, Second Edition*. Boston, MA: Addison-Wesley.
- Lieberherr, Karl J., and Ian Holland. 1989. "Assuring Good Style for Object-Oriented Programs." Washington, DC. IEEE Software.
- Liskov, Barbara. May 1987. "Data Abstraction and Hierarchy." *SIGPLAN Notices*, volume 23, number 5.
- Metsker, Steven J. 2001. *Building Parsers with Java™*. Boston, MA: Addison-Wesley.
- Russell, Michael S. 2000. *The Chemistry of Fireworks*. Cambridge, UK: Royal Society of Chemistry.
- Vlissides, John. 1998. *Pattern Hatching: Design Patterns Applied*. Reading, MA: Addison-Wesley.
- Wake, William C. 2004. *Refactoring Workbook*. Boston, MA: Addison-Wesley.
- Weast, Robert C., ed. 1983. *CRC Handbook of Chemistry and Physics*, 63rd ed. Boca Raton, FL: CRC Press.
- White, Seth, Mayderne Fisher, Rick Cattell, Graham Hamilton, and Mark Hapner. 1999. *JDBC™ API Tutorial and Reference, Second Edition*. Boston, MA: Addison-Wesley.
- Wolf, Bobby. 1998. "Null Object," in *Pattern Languages of Program Design 3*, ed. Robert Martin, Dirk Riehle, and Frank Buschmann. Reading, MA: Addison-Wesley.