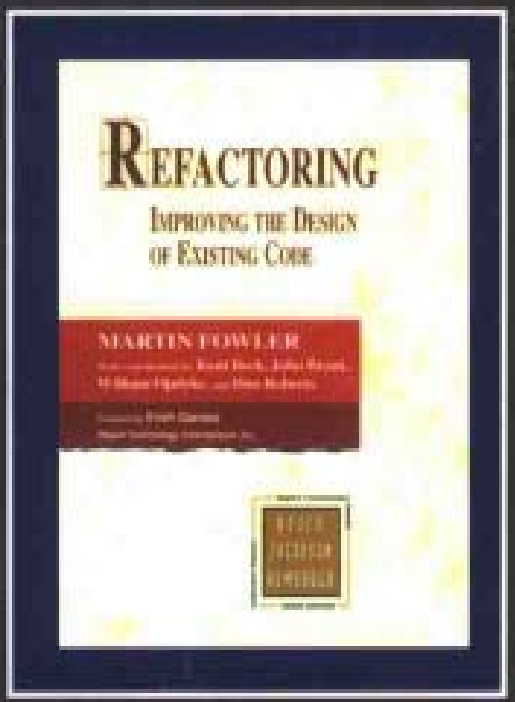
软 件 工 程 系 列

Refactoring: Improving the Design of Existing Code

重构——改善既有代码的设计（中文版）

[美]Martin Fowler 著
侯捷 熊节 译



- 与《设计模式》齐名的经典巨著
- 《设计模式》作者 Erich Gamma 为本书作序
- 超过70种行之有效的重构方法

中国电力出版社
www.infopower.com.cn

Refactoring: Improving the Design of Existing Code

重构——改善既有代码的设计（中文版）

当对象技术成为老生常谈之后——尤其在 Java 编程语言之中，新的问题也在软件开发社区中浮现了出来。缺乏经验的开发人员完成了大量粗劣设计，获得的程序不但缺乏效率，也难以维护和扩展。渐渐地，软件系统专家发现，与这些沿袭下来的、质量不佳的程序共处，是多么艰难。对象专家运用许多（而且日渐更多）技术来改善既有程序的结构完善性与性能，已有数年之久。但是这些被称为“重构”（refactoring）的实践技术，一直（只）流传于专家领域内，因为没有人愿意将全部这些知识录写为所有开发人员可读的形式。这种情况如今终于结束。在本书中，知名的对象技术者 Martin Fowler 闯入新的领域，揭去那些名家实践手法的神秘面纱，展示软件从业人员领悟这种新过程的重大意义。

只要受过适度训练，一位技巧娴熟的系统程序员可以在拿到一个糟糕的设计之后，把它翻新为设计良好、稳健强固的代码。本书之中，Martin Fowler 告诉你重构机会通常可以在哪里找到，以及如何将一个糟糕的设计重新修订为一个良好的设计。每个重构步骤都十分简单——简单到了似乎不值得去做的程度。重构涉及将值域（field）从一个 class 搬移到另一个 class，或将某些代码拉出来独立为另一个函数（method），或甚至将某些代码上下移动于继承体系（hierarchy）之中。这些个别步骤虽然可能十分基本，积累下来的影响却能够彻底改善设计。重构已经被证明可以阻止软件的腐朽与衰败。

除了讨论各式各样的重构技术，作者还提供了一份详细名录（catalog），其中有超过 70 个已被证明效果的重构手法，以饶富帮助的重点，教导你实施的时机，实施时的逐步指令。并各自携带一个例子，显示重构的运转。这些富有良好解说价值的实例都以 Java 写就，其中的观念适用于任何面向对象编程语言。

Martin Fowler 是一位独立咨询顾问，他运用对象技术解决企业问题已经超过十年。他的顾问领域包括健康管理、金融贸易，以及法人财务。他的客户包括 Chrysler, Citibank, UK National Health Service, Andersen Consulting, Netscape Communications。此外 Fowler 也是 objects、UML、patterns 技术的一位合格讲师。他是《Analysis Patterns》和《UML Distilled》的作者。

Kent Beck 是一位知名的程序员、测试员、重构员、作家、五弦琴专家。

John Brant 和 **Don Roberts** 是《Refactoring Browser for Smalltalk》的作者，此书可从 http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser 获得。他们两人也是咨询顾问，研究重构的实践与理论有六年之久。

William Opdyke 在伊利诺斯大学所做的 object-oriented frameworks（面向对象框架）博士研究，导出了重构领域的第一份重要出版品。他目前是 Lucent Technologies/Bell Laboratories 的一名卓越技术人员。

译者侯捷，致力计算机技术教育超过十年——以著作、翻译、评论、专栏、授课等多重形式。对于各种层级、各种定位、各种技术领域之 Framework Libraries 有浓烈兴趣和钻研。

译者熊节，普通程序员，喜编程，乐此而不疲。酷爱读书，好求新知。记性好忘性大，故凡有所得必记诸文字，有小得，无大成。胸有点墨，心无大志，惟愿宁静淡泊而已。夜阑人静，一杯清水，几本闲书，神交于各方名士，献曝于天下同好，吾愿足矣。

中文版（本书）支持网站：http://www.jhou.com（繁体）http://jjhou.csdn.net（简体）

ISBN 7-5083-1554-5

9 787508 315546 >

责任编辑/程 驹 乔 晶
封面设计/王红卿

PEARSON
Addison Wesley

www.PearsonEd.com

ISBN 7-5083-1554-5

定价：68.00 元

重构

— 改善既有代码的设计 —

Refactoring

Improving the Design of Existing Code

Martin Fowler 著

（以及 Kent Beck, John Brant , William Opdyke,

Don Roberts 对最后三章的贡献）

侯捷 / 熊节 合译

Refactoring: Improving the Design of Existing Code (ISBN 0-201-48567-2)
Martin Fowler

图书在版编目（CIP）数据

重构：改善既有代码的设计 /（美）福勒（Fowler, M.）著；侯捷，熊节译.
-北京：中国电力出版社，2003（软件工程系列）
ISBN 7-5083-1554-5
Ⅰ.重… Ⅱ.①福…②侯…③熊… Ⅲ.代码－程序设计 Ⅳ.TP311.11
中国版本图书馆 CIP 数据核字（2003）第 057350 号

责任编辑：程璐 乔晶
书　　名：重构：改善既有代码的设计
原　　著：（美）Martin Fowler
翻　　译：侯捷 熊节
出版发行：中国电力出版社
地址：北京市三里河路6号 邮政编码：100044
电话：（010）88515918 传真：（010）88518169
印　　刷：汇鑫印务有限公司印刷
开　　本：787×1092 1/16　　**印　　张：**29　　**字　　数：**540 千字
书　　号：ISBN 7-5083-1554-5
版　　次：2003年8月北京第一版
印　　次：2003年8月第一次印刷
定　　价：68.00 元

Refactorings（重构）列表

译序

by 侯捷

看过铁路道班工人吗？提着手持式砸道机，机身带着钝钝扁扁的钻头，在铁道上、枕木间卖力地「砍劈钻凿」。他们在做什么？他们在使路基上的碎石块（道碴）因持续剧烈的震动而翻转方向、滑动位置，甚至震碎为更小石块填满缝隙，以求道碴更紧密契合，提供铁道更安全更强固的体质。

当「重构」（refactoring）映入眼帘，我的大脑牵动「道班工人+电动砸道机+枕木道碴」这样一幅联想画面。「重构」一词非常清楚地说明了它自身的意义和价值：在不破坏可察功能的前提下，借由搬移、提炼、打散、凝聚…，改善事物的体质。很多人认同这样一个信念：「非常的建设需要非常的破坏」，但是现役的应用软件、构筑过半的项目、运转中的系统，容不得推倒重来。这时候，在不破坏可察功能的前提下改善体质、强化当前的可读性、为将来的扩充性和维护性做准备、乃至于在过程中找出潜伏的「臭虫」，就成了大受欢迎的稳步前进的良方。

作为一个程序员，任谁都有看不顺眼手上代码的经验 — 代码来自你邻桌那个菜鸟，或三个月前的自己。面临此境，有人选择得过且过；然而根据我对「程序员」人格特质的了解，更多人盼望插手整顿。挽起袖子剑及履及，其勇可嘉，其虑未缜。过去或许不得小暴虎凭河，忍受风险。现在，有了严谨的重构准则和严密的重构手法，「稳定中求发展」终于有了保障。

是的，把重构的概念和想法逐一落实在严谨的准则和严密的手法之中，正是这本《*Refactoring*》的最大贡献。重构?! 呵呵，上进的程序员每天的进行式，从来不新鲜，但要强力保证「维持程序原有的可察功能，不带进新臭虫」，重构就不能是一项靠着天份挥洒的艺术，必须是一项工程。

我对本书的看法

初初阅读本书，屡屡感觉书中所列的许多重构目标过于平淡，重构步骤过于琐屑。这些我们平常也都做、习惯大气挥洒的动作，何必以近乎枯燥的过程小步前进？然后，渐渐我才体会，正是这样的小步与缓步前进，不过激，不躁进，再加上完整的测试配套（是的，测试之于重构极其重要），才是「不带来破坏，不引入臭虫」的最佳保障。我个人其实不敢冒信有谁能够乖乖地按步遵循实现本书所列诸多被我（从人的角度）认为平淡而琐屑的重构步骤。我个人认为，本书的最大价值，除了呼吁对软件质量的追求态度，以及对重构「工程性」的认识，最终最重要的价值还在于：建立起吾人对于「目前和未来之自动化重构工具」的基本理论和实现技术上的认识与信赖。人类眼中平淡琐屑的步骤，正是自动化重构工具的基础。机器缺乏人类的「大局观」智慧，机器需要的正是切割为一个一个极小步骤的指令。一板一眼，一次一点点，这正是机器所需要的，也正是机器的专长。

本书第 14 章提到，Smalltalk 开发环境已含自动化重构工具。我并非 Smalltalk guy，我没有用过这些工具。基于技术的飞快滚动（或我个人的孤陋寡闻），或许如今你已经可以在 Java, C++ 等面向对象编程环境中找到这一类自动化重构工具。

软件技术圈内，重构（refactoring）常常被拿来与设计模式（design patterns）并论。书籍市场上，《*Refactoring*》也与《*Design Patterns*》齐名。GoF 曾经说『设计模式为重构提供了目标』，但本书作者 Martin 亦言『本书并没有提供助你完成所有知名模式的重构手法，甚至连 GoF 的 23 个知名模式都没有能够全部覆盖。』我们可以从这些话中理解技术的方向，以及书籍所反映的局限。我并不完全赞同 Martin 所言『哪怕你手上有一个糟糕的设计或甚至一团混乱，你也可以借由重构将它加工成设计良好的代码。』但我十分同意 Martin 说『你会发现所谓设计不再是一切动作的前提，而是在整个开发过程中逐渐浮现出来。』我比较担心，阅历不足的程序员在读过本书后可能发酵出「先动手再说，死活可重构」的心态，轻忽了事前优秀设计的重要性。任何技术上的说法都必须有基本假设；虽然重构（或更向上说 XP，eXtreme Programming）的精神的确是「不妨先动手」，但若草率行事，代价还是很高的。重型开发和轻型开发各有所长，各有应用，世间并无万应灵药，任何东西都不能极端。过犹不及，皆不可取！

当然，「重构工程」与「自动化重构工具」可为我们带来相当大幅度的软件质量提升，这一点我毫无异议，并且非常期待☺。

关于本书制作

此书在翻译与制作上保留了所有坏味道（bad smell）、重构（refactoring）、设计模式（design patterns）的英文名称，并表现以特殊字体；只在封面内页、目录、小节标题中相应地给出一个根据字面或技术意义而做的中文译名。各种「坏味道」名称尽量就其意义选用负面字眼，如泥团、夸夸、过长、过大、过多、情结、偏执、惊悚、狎昵、纯稚、冗赘…。这些其实都是助忆之用，与茶余饭后的谈资（以及读者批评的根据☺）。

原书各小节并无序号。为参考、检索或讨论时的方便，我为译本加上了序号。

本书保留相当份量的英文术语，时而英中并陈（英文为主，中文为辅）。这么做的考量是，本书读者不可能不知道 class, final, reference, public, package…这些简短的、与 Java 编程息息相关的用词。另一方面，我确实认为，中文书内保留经过挑选的某些英文术语，有利于整体阅读效果。

两个需要特别说明的用词是 Java 编程界惯用的 "field" 和 "method"。它们相当于 C++ 的 "data member" 和 "member function"。由于出现次数实在频繁，为降低中英夹杂程度，我把它们分别译为「值域」和「函数」。如果将 "method" 译为「方法」，恐怕术语突出性不高。本书将 "type" 译为「型别」而非「类型」，亦是为了中文术语之突出性；"instance" 译为「实体」而非「实例」、"argument" 译为「引数」而非「实参」，有意义上的考量。「*static* 值域与 *reference* 值域」、「*reference* 对象与 *value* 对象」等等则保留部分英文，并选用如上的特殊字体。凡此种种，相信一进入书中您很快可以感受本书术语风格。

本书还有诸多地方采中英并陈（中文为主，英文为辅）方式，意在告诉读者，我们（译者）深知自己的不足与局限，惟恐造成您对中译名词的误解或不习惯，所以附上原文。

中文版（本书）已将英文版截至 2003/06/18 为止之勘误，修正于纸本。

一点点感想

Martin Fowler 表现于原书的写作风格是：简洁，爱用代名词和略称。这使得读者往往需要在字面上揣度推敲。我期盼（并相信）经过技术意义的反刍、中英术语的并陈、中文表述的努力，中文版（本书）在阅读时间、理解时间和记忆深度上，较之英文版，能够为以华文为母语的读者提高 10 倍以上的成效。

本书由我和熊节先生合译。熊节负责第一个 pass，我负责后继工作。中文版（本书）为读者带来的阅读和理解上的效益，熊节居上首功 — 虽说做的是第一个 pass，我从初稿质量便可看出他多次反复推敲和文字琢磨的刻痕。至于整体风格、中英术语的选定、版面的呈现、乃至于全盘技术内涵的表现，如果有任何差错，责任都是我的☺。

作为一个信息技术教育者，以及一个信息技术传播者，我在超过 10 年的写译历程中，观察了不同级别的技术书品在读书市场上的兴衰起伏。这些适可反映大环境下技术从业人员及学子们的某些面向和取向。我很高兴看到我们的中文技术书籍（著译皆含）从早期盈盈满满的初阶语言用书，逐渐进化到中高阶语言用书、操作系统、技术内核、程序库/框架、再至设计/分析、软件工程。我很高兴看到这样的变化，我很高兴看到《*Design Patterns*》、《*Refactoring*》、《*Agile...*》、《*UML...*》、《*XP...*》之类的书在中文书籍市场中现身，并期盼它们有丰富的读者。

中文版（本书）支援网站有一个「术语 英中繁简」对照表。如果您有需要，欢迎访问，网址如下，并欢迎给我任何意见。谢谢。

侯捷 2003/06/18 于台湾.新竹

jjhou@jjhou.com（电子邮箱）
http://www.jjhou.com（繁体）（术语对照表 http://www.jjhou.com/terms.htm）
http://jjhou.csdn.net（简体）（术语对照表 http:// jjhou.csdn.net/terms.htm）

译序

by 熊节

重构的生活方式

还记得那一天，当我把《重构》的全部译稿整理完毕，发送给侯老师时，心里竟然不经意地有了一丝惆然。我是一只习惯的动物，总是安于一种习惯的生活方式。在那之前的很长一段时间里，习惯了每天晚上翻译这本书，习惯了随手把问题写成mail发给Martin Fowler先生，习惯了阅读Martin及时而耐心的回信，习惯了在那本复印的、略显粗糙的书本上勾勾画画，习惯了躺在床上咀嚼回味那些带有一点点英国绅士矜持口吻的词句，习惯了背后嗡嗡作响的老空调…当深秋的风再次染红了香山的叶，这种生活方式也就告一段落了。

只有几位相熟的朋友知道我在翻译这本书，他们不太明白为什么常把经济学挂在嘴边的我会乐于帮侯老师翻译这本书 — 我自己也不明白，大概只能用爱好来解释吧。既然已经衣食无忧，既然还有一点属于自己的时间，能够亲手把这本《重构》翻译出来，也算是给自己的一个交代。

第一次听到「重构」这个词，是在2001年10月。在当时，它的思想足以令我感到震撼。软件自有其美感所在。软件工程希望建立完美的需求与设计，按照既有的规范编写标准划一的代码，这是结构的美；快速迭代和RAD颠覆「全知全能」的神话，用近乎刀劈斧砍（crack）的方式解决问题，在混沌的循环往复中实现需求，这是解构的美；而Kent Beck与Martin Fowler两人站在一起，XP那敏捷而又严谨的方法论演绎了重构的美 — 我不知道是谁最初把refactoring一词翻译为「重构」，或许无心插柳，却成了点睛之笔。

我一直是设计模式的爱好者。曾经在我的思想中，软件开发应该有一个「理想国」— 当然，在这个理想国维持着完美秩序的，不是哲学家，而是模式。设计模式给我们

的，不仅仅是一些问题的解决方案，更有追求完美「理型」的渴望。但是，Joshua Kerievsky 在那篇著名的《模式与 XP》（收录于《极限编程研究》一书）中明白地指出：在设计前期使用模式常常导致过度工程（over-engineering），这是一个残酷的现实，单凭对完美的追求无法写出实用的代码，而「实用」是软件压倒一切的要素。从一篇《停止过度工程》开始，Joshua 撰写了"Refactoring to Patterns"系列文章。这位犹太人用他民族性的睿智头脑，敏锐地发现了软件的后结构主义道路。而让设计模式在飞速变化的 Internet 时代重新闪现光辉的，又是重构的力量。

在一篇流传甚广的帖子里，有人把《重构》与《设计模式》并列为「Java 行业的圣经」。在我看来这种并列其实并不准确。实际上，尽管我如此喜爱这本《重构》，但自从完成翻译之后，我再也没有读过它。不，不是因为我已经对它烂熟于心，而是因为重构已经变成了我的另一种生活方式，变成了我每天的「面包与黄油」，变成了我们整个团队的空气与水，以至于无须再到书中寻找任何「神谕」。而《设计模式》，我倒是放在手边时常翻阅，因为总是记得不那么真切。

所以，在你开始阅读本书之前，我有两个建议要给你：首先，把你的敬畏扔到大西洋里去，对于即将变得像空气与水一样普通的技术，你无须对它敬畏；其次，找到合适的开发工具（如果你和我一样是 Java 人，那么这个「合适的工具」就是 Eclipse），学会使用其中的自动测试和重构功能，然后再尝试使用本书介绍的任何技术。懒惰是程序员的美德之一，绝不要因为这本书让你变得勤快。

最后，即使你完全掌握了这本书中的所有东西，也千万不要跟别人吹嘘。在我们的团队里，程序员常常会说：『如果没有单元测试和重构，我没办法写代码。』

好了，感谢你耗费一点点的时间来倾听我现在对重构、对这本《重构》的想法。Martin Fowler 经常说，花一点时间来重构是值得的，希望你会觉得花一点时间看我的文字也是值得的。

熊节 2003 年 6 月 11 日
夜 杭州

P.S. 我想借这个难得的机会感谢一个人：我亲爱的女友马姗姗。在北京的日子里，是她陪伴着我度过每个日日夜夜，照顾我的生活，使我能够有精力做些喜欢的事（包括翻译这本书）。当我埋头在屏幕前敲打键盘时，当我抱着书本冥思苦想时，她无私地容忍了我的痴迷与冷淡。谢谢你，姗姗，我永远爱你。

序言

by Erich Gamma

重构（**refactoring**）这个概念来自 Smalltalk 圈子，没多久就进入了其他语言阵营之中。由于重构是 framework（框架）开发中不可缺少的一部分，所以当 framework 开发人员讨论自己的工作时，这个术语就诞生了。当他们精炼自己的 class hierarchies（类阶层体系）时，当他们叫喊自己可以拿掉多少多少行代码时，重构的概念慢慢浮出水面。framework 设计者知道，这东西不可能一开始就完全正确，它将随着设计者的经验成长而进化；他们也知道，代码被阅读和被修改的次数远远多于它被编写的次数。保持代码易读、易修改的关键，就是重构 — 对 framework 而言如此，对一般软件也如此。

好极了，还有什么问题吗？很显然：重构具有风险。它必须修改运作中的程序，这可能引入一些幽微的错误。如果重构方式不恰当，可能毁掉你数天甚至数星期的成果。如果重构时不做好准备，不遵守规则，风险就更大。你挖掘自己的代码，很快发现了一些值得修改的地方，于是你挖得更深。挖得愈深，找到的重构机会就越多…于是你的修改也愈多。最后你给自己挖了个大坑，却爬不出去了。为了避免自掘坟墓，重构必须系统化进行。我在《*Design Patterns*》书中和另外三位（协同）作者曾经提过：design patterns（设计模式）为 refactoring（重构）提供了目标。然而「确定目标」只是问题的一部分而已，改造程序以达目标，是另一个难题。

Martin Fowler 和本书另几位作者清楚揭示了重构过程，他们为面向对象软件开发所做的贡献，难以衡量。本书解释重构的原理（principles）和最佳实践方式（best practices），并指出何时何地你应该开始挖掘你的代码以求改善。本书的核心是一份完整的重构名录（catalog of refactoring），其中每一项都介绍一种经过实证的代码变换手法（code transformation）的动机和技术。某些项目如 *Extract Method* 和

*Move Field*看起来可能很浅显，但不要掉以轻心，因为理解这类技术正是有条不紊地进行重构的关键。本书所提的这些重构准则将帮助你一次一小步地修改你的代码，这就减少了过程中的风险。很快你就会把这些重构准则和其名称加入自己的开发词典中，并且朗朗上口。

我第一次体验有纪律的、一次一小步的重构，是在 30000 英呎高空和 Kent Beck 共同编写程序（译注：原文为 pair-programming，应该指的是 *eXtreme Programming* 中的所谓「成对搭档编程」）。我们运用本书收录的重构准则，保证每次只走一步。最后，我对这种实践方式的效果感到十分惊讶。我不但对最后结果更有信心，而且开发压力也小了很多。所以，我高度推荐你试试这些重构准则，你和你的程序都将因此更美好。

— *Erich Gamma*

Object Technology International, Inc.

Java 高级架构师

每天晚上 365 天提供在线直播教程：

1、具有 1-8 工作经验的，面对目前流行的技术不知从何下手，需要突破技术瓶颈的可以学习。

2、在公司待久了，过得很安逸，但跳槽时面试碰壁。需要在短时间内进修、跳槽拿高薪的可以学习。

3、没有工作经验，但基础非常扎实，对 java 工作机制，常用设计思想，常用 java 开发框架掌握熟练的可以学习。

4、工作中一般需求都能搞定。但是所学的知识点没有系统化，很难在技术领域继续突破的可以学习。

课程涉及知识点内容：

①源码分析专题　②高并发/高性能　③高可用/可扩展

④性能优化　⑤Spring Cloud　⑥Spring Boot

Java 高级架构师专题课程（加 QQ 获取教程地址：2621000066）

或者扫描二维码添加

前言

by Martin Fowler

从前，有位咨询顾问参访一个开发项目。系统核心是个 class hierarchy（类阶层体系），顾问看了开发人员所写的一些代码。他发现整个体系相当凌乱，上层 classes 对于 classes 的运作做了一些假设，下层（继承）classes 实现这些假设。但是这些假设并不适合所有 subclasses，导致覆写（overridden）行为非常繁重。只要在 superclass 内做点修改，就可以减少许多覆写必要。在另一些地方，superclass 的某些意图并未被良好理解，因此其中某些行为在 subclasses 内重复出现。还有一些地方，好几个 subclasses 做相同的事情，其实可以把它们搬到 class hierarchy 的上层去做。

这位顾问于是建议项目经理看看这些代码，把它们整理一下，但是经理并不热衷于此，毕竟程序看上去还可以运行，而且项目面临很大的进度压力。于是经理说，晚些时候再抽时间做这些整理工作。

顾问也把他的想法告诉了在这个 class hierarchy 上工作的程序员，告诉他们可能发生的事情。程序员都很敏锐，马上就看出问题的严重性。他们知道这并不全是他们的错，有时候的确需要借助外力才能发现问题。程序员立刻用了一两天的时间整理好这个 class hierarchy，并删掉了其中一半代码，功能毫发无损。他们对此十分满意，而且发现系统速度变得更快，更容易加入新 classes 或使用其他 classes。

项目经理并不高兴。进度排得很紧，许多工作要做。系统必须在几个月之后发布，许多功能还等着加进去，这些程序员却白白耗费两天时间，什么活儿都没干。原先的代码运行起来还算正常，他们的新设计显然有点过于「理论」且过于「无瑕」。项目要出货给客户的，是可以有效运行的代码，不是用以取悦学究们的完美东西。顾问接下来又建议应该在系统的其他核心部分进行这样的整理工作，这会使整个项目停顿一至二个星期。所有这些工作只是为了让代码看起来更漂亮，并不能给系统

添加任何新功能。

你对这个故事有什么看法？你认为这个顾问的建议（更进一步整理程序）是对的吗？你会因循那句古老的工程谚语吗：「如果它还可以运行，就不要动它」。

我必须承认我自己有某些偏见，因为我就是那个顾问。六个月之后这个项目宣告失败，很大的原因是代码太复杂，无法除错，也无法获得可被接受的性能。

后来，项目重新启动，几乎从头开始编写整个系统，Kent Beck 被请去做了顾问。他做了几件迥异以往的事，其中最重要的一件就是坚持以持续不断的重构行为来整理代码。这个项目的成功，以及重构（refactoring）在这个成功项目中扮演的角色，促成了我写这本书的动机，如此一来我就能够把 Kent 和其他一些人已经学会的「以重构方式改进软件质量」的知识，传播给所有读者。

什么是重构（Refactoring）？

所谓重构是这样一个过程：「在不改变代码外在行为的前提下，对代码做出修改，以改进程序的内部结构」。重构是一种有纪律的、经过训练的、有条不紊的程序整理方法，可以将整理过程中不小心引入错误的机率降到最低。本质上说，重构就是「在代码写好之后改进它的设计」。

「在代码写好之后改进它的设计」？这种说法有点奇怪。按照目前对软件开发的理解，我们相信应该先设计而后编码（coding）。首先得有一个良好的设计，然后才能开始编码。但是，随着时间流逝，人们不断修改代码，于是根据原先设计所得的系统，整体结构逐渐衰弱。代码质量慢慢沉沦，编码工作从严谨的工程堕落为胡砍乱劈的随性行为。

「重构」正好与此相反。哪怕你手上有一个糟糕的设计，甚至是一堆混乱的代码，你也可以借由重构将它加工成设计良好的代码。重构的每个步骤都很简单，甚至简单过了头，你只需要把某个值域（field）从一个 class 移到另一个 class，把某些代码从一个函数（method）拉出来构成另一个函数，或是在 class hierarchy 中把某些代码推上推下就行了。但是，聚沙成塔，这些小小的修改累积起来就可以根本改善设计质量。这和一般常见的「软件会慢慢腐烂」的观点恰恰相反。

通过重构（refactoring），你可以找出改变的平衡点。你会发现所谓设计不再是一切动作的前提，而是在整个开发过程中逐渐浮现出来。在系统构筑过程中，你可以学习如何强化设计；其间带来的互动可以让一个程序在开发过程中持续保有良好的设计。

本书有些什么？

本书是一本重构指南（guide to refactoring），为专业程序员而写。我的目的是告诉你如何以一种可控制且高效率的方式进行重构。你将学会这样的重构方式：不引入「臭虫」（错误），并且有条不紊地改进程序结构。

按照传统，书籍应该以一个简介开头。尽管我也同意这个原则，但是我发现以概括性的讨论或定义来介绍重构，实在不是件容易的事。所以我决定拿一个实例做为开路先锋。第 1 章展示一个小程序，其中有些常见的设计缺陷，我把它重构为更合格的面向对象程序。其间我们可以看到重构的过程，以及数个很有用的重构准则。如果你想知道重构到底是怎么回事，这一章不可不读。

第 2 章涵盖重构的一般性原则、定义，以及进行原因，我也大致介绍了重构所存在的一些问题。第 3 章由 Kent Beck 介绍如何嗅出代码中的「坏味道」，以及如何运用重构清除这些坏味道。「测试」在重构中扮演非常重要的角色，第 4 章介绍如何运用一个简单的（源码开放的）Java 测试框架，在代码中构筑测试环境。

本书的核心部分，重构名录（catalog of refactorings），从第 5 章延伸至第 12 章。这不是一份全面性的名录，只是一个起步，其中包括迄今为止我在工作中整理下来的所有重构准则。每当我想做点什么 — 例如 *Replace Conditional with Polymorphism* — 的时候，这份名录就会提醒我如何一步一步安全前进。我希望这是值得你日后一再回顾的部分。

本书介绍了其他人的许多研究成果，最后数章就是由他们之中的几位所客串写就。Bill Opdyke 在第 13 章记述他将重构技术应用于商业开发过程中遇到的一些问题。Don Roberts 和 John Brant 在第 14 章展望重构技术的未来 — 自动化工具。我把最后一章（第 15 章）留给重构技术的顶尖大师，Kent Beck。

在 Java 中运用重构

本书全部以 Java 撰写实例。重构当然也可以在其他语言中实现，而且我也希望这本书能够给其他语言使用者带来帮助。但我觉得我最好在本书中只使用 Java，因为那是我最熟悉的语言。我会不时写下一些提示，告诉读者如何在其他语言中进行重构，不过我真心希望看到其他人在本书基础上针对其他语言写出更多重构方面的书籍。

为了最大程度地帮助读者理解我的想法，我不想使用 Java 语言中特别复杂的部分。所以我避免使用 inner class（内嵌类）、reflection（反射机制）、thread（线程）以及很多强大的 Java 特性。这是因为我希望尽可能清楚展现重构的核心。

我应该提醒你，这些重构准则并不针对并发（concurrent）或分布式（distributed）编程。那些主题会引出更多重要的事，超越了本书的关心范围。

谁该阅读本书？

本书瞄准专业程序员，也就是那些以编写软件为生的人。书中的示例和讨论，涉及大量需要详细阅读和理解的代码。这些例子都以 Java 完成。之所以选择 Java，因为它是一种应用范围愈来愈广的语言，而且任何具备 C 语言背景的人都可以轻易理解它。Java 是一种面向对象语言，而面向对象机制对于重构有很大帮助。

尽管关注对象是代码，重构（refactoring）对于系统设计也有巨大影响。资深设计师（senior designers）和架构规划师（architects）也很有必要了解重构原理，并在自己的项目中运用重构技术。最好是由老资格、经验丰富的开发人员来引入重构技术，因为这样的人最能够良好理解重构背后的原理，并加以调整，使之适用于特定工作领域。如果你使用 Java 以外的语言，这一点尤其必要，因为你必须把我给出的范例以其他语言改写。

下面我要告诉你：如何能够在不遍读全书的情况下得到最多知识。

- 如果你想知道重构是什么，请阅读第 1 章，其中示例会让你清楚重构过程。
- 如果你想知道为什么应该重构，请阅读前两章。它们告诉你「重构是什么」以及「为什么应该重构」。

- 如果你想知道该在什么地方重构，请阅读第 3 章。它会告诉你一些代码特征，这些特征指出「这里需要重构」。
- 如果你想真正（实际）进行重构，请完整阅读前四章，然后选择性地阅读重构名录（refactoring catalog）。一开始只需概略浏览名录，看看其中有些什么，不必理解所有细节。一旦真正需要实施某个准则，再详细阅读它，让它来帮助你。名录是一种具备查询价值的章节，你也许并不想一次把它全部读完。此外你还应该读一读名录之后的「客串章节」，特别是第 15 章。

站在前人的肩膀上

就在本书一开始的此刻，我必须说：这本书让我欠了一大笔人情债，欠那些在过去十年中做了大量研究工作并开创重构领域的人一大笔债。这本书原本应该由他们之中的某个人来写，但最后却是由我这个有时间有精力的人捡了便宜。

重构技术的两位最早拥护者是 Ward Cunningham 和 Kent Beck。他们很早就把重构作为开发过程的一个核心成份，并且在自己的开发过程中运用它。尤其需要说明的是，正因为和 Kent 的合作，才让我真正看到了重构的重要性，并直接激励了我写这一本书。

Ralph Johnson 在 University of Illinois, Urbana-Champaign（伊利诺斯大学乌尔班纳分校）领导了一个小组，这个小组因其对对象技术（object technology）的实际贡献而闻名。Ralph 很早就是重构技术的拥护者，他的一些学生也一直在研究这个课题。Bill Opdyke 的博士论文是重构研究领域的第一份详细书面成果。John Brant 和 Don Roberts 则早已不满足于写文章了，他们写了一个工具 — 重构浏览器（Refactoring Browser），对 Smalltalk 程序实施重构工程。

致谢

尽管有这些研究成果帮忙，我还需要很多协助才能写出这本书。首先，并且也是最重要的，Kent Beck 给了我巨大的帮助。Kent 在底特律（Detroit）和我谈起他正在为 *Smalltalk Report* 撰写一篇论文 [Beck, hanoi]，从此播下本书的第一颗种子。那篇论文不但让我开始注意到重构技术，而且我还从中「偷」了许多想法放到本书第 1 章。Kent 也在其他地方帮助我，想出「代码味道」这个概念的是他，当我遇到各种困难时，鼓励我的人也是他，常常和我一起工作助我完成这本书的，还是他。我常

常忍不住这么想：他完全可以自己把这本书写得更好。可惜有时间写书的人是我，所以我也只能希望自己不要做得太差。

写这本书的时候，我希望能把一些专家经验直接与你分享，所以我非常感激那些花时间为本书添加材料的人。Kent Beck, John Brant, William Opdyke 和 Don Roberts 编撰或合著了本书部分章节。此外 Rich Garzaniti 和 Ron Jeffries 帮我添加了一些有用的补充资料。

在任何像这样的一本书里，作者都会告诉你，技术审阅者提供了巨大的帮助。一如以往，Addison-Wesley 的 Carter 和他的优秀团队是一群精明的审阅者。他们是：

- Ken Auer, Rolemodel Software, Inc.
- Joshua Bloch, Sun Microsystems, Java Software
- John Brant, University of Illinois at Urbana-Champaign
- Scott Corley, High Voltage Software, Inc.
- Ward Cunningham, Cunningham & Cunningham, Inc.
- Stéphane Ducasse
- Erich Gamma, Object Technology International, Inc.
- Ron Jeffries
- Ralph Johnson, University of Illinois
- Joshua Kerievsky, Industrial Logic, Inc.
- Doug Lea, SUNY Oswego
- Sander Tichelaar

他们大大提高了本书的可读性和准确性，并且至少去掉了一些任何手稿都可能会有的潜在错误。在此我要特别感谢两个效果显著的建议，这两个建议让我的书看上去耳目一新：Ward 和 Ron 建议我以重构前后效果（包括代码和 UML 图）并列的方式写第 1 章，Joshua 建议我在重构名录中画出代码梗概（code sketches）。

除了正式审阅小组，还有很多非正式的审阅者。这些人或看过我的手稿，或关注我的网页并留下对我很有帮助的意见。他们是 Leif Bennett, Michael Feathers, Michael Finney, Neil Galarneau, Hisham Ghazouli, Tony Gould, John Isner, Brian Marick, Ralf Reissing, John Salt, Mark Swanson, Dave Thomas 和 Don Wells。我相信肯定还有一些被我遗忘的人，请容我在此向你们道歉，并致上我的谢意。

有一个特别有趣的审阅小组，就是「恶名昭彰」☺ 的 University of Illinois at Urbana-Champaign 读书小组。由于本书反映出他们的众多研究成果，我要特别感谢他们的成就。这个小组成员包括 Fredrico "Fred" Balaguer, John Brant, Ian Chai, Brian Foote, Alejandra Garrido, Zhijiang "John" Han, Peter Hatch, Ralph Johnson, Songyu "Raymond" Lu, Dragos-Anton Manolescu, Hiroaki Nakamura, James Overturf, Don Roberts, Chieko Shirai, Les Tyrell 和 Joe Yoder。

任何好想法都需要在严酷的生产环境中接受检验。我看到重构对于 Chrysler Comprehensive Compensation（C3）系统起了巨大的影响。我要感谢那个团队的所有成员：Ann Anderson, Ed Anderi, Ralph Beattie, Kent Beck, David Bryant, Bob Coe, Marie DeArment, Margaret Fronczak, Rich Garzaniti, Dennis Gore, Brian Hacker, Chet Hendrickson, Ron Jeffries, Doug Joppie, David Kim, Paul Kowalsky, Debbie Mueller, Tom Murasky, Richard Nutter, Adrian Pantea, Matt Saigeon, Don Thomas 和 Don Wells。和他们一起工作所获得的第一手数据，巩固了我对重构原理和利益的认识。他们在重构技术上不断进步，极大程度地帮助我看到：一旦重构技术应用于历时多年的大型项目中，可以起怎样的作用。

再一次，我得到了 Addison-Wesley 的 J. Carter Shanklin 和其团队的帮助，包括 Krysia Bebick, Susan Cestone, Chuck Dutton, Kristin Erickson, John Fuller, Christopher Guzikowski, Simone Payment 和 Genevieve Rajewski。与优秀出版商合作是一个令人愉快的经验，他们会提供给作者大量的支援和帮助。

谈到支援，为一本书付出最多的，总是距离作者最近的人。对我来说，那就是我（现在）的妻子 Cindy。感谢你，当我埋首工作的时候，你还是一样爱我。当我投入书中，总会不断想起你。

Martin Fowler
Melrose, Massachusetts
fowler@acm.org
http://www.martinfowler.com
http://www.refactoring.com

坏味道（smell）	常用的重构手法（Common Refactoring）
Alternative Classes with Different Interfaces, p85	*Rename Method* (273), *Move Method* (142)
Comments, p87	*Extract Method* (110), *Introduce Assertion* (267)
Data Class, p86	*Move Method* (142), *Encapsulate Field* (206), *Encapsulate Collection* (208)
Data Clumps, p81	*Extract Class* (149), *Introduce Parameter Object* (295), *Preserve Whole Object* (288)
Divergent Change, p79	*Extract Class* (149)
Duplicated Code, p76	*Extract Method* (110), *Extract Class* (149), *Pull Up Method* (322), *Form Template Method* (345)
Feature Envy, p80	*Move Method* (142), *Move Field* (146), *Extract Method* (110)
Inappropriate Intimacy, p85	*Move Method* (142), *Move Field* (146), *Change Bidirectional Association to Unidirectional* (200), *Replace Inheritance with Delegation* (352), *Hide Delegate* (157)
Incomplete Library Class, p86	*Introduce Foreign Method* (162), *Introduce Local Extension* (164)
Large Class, p78	*Extract Class* (149), *Extract Subclass* (330), *Extract Interface* (341), *Replace Data Value with Object* (175)
Lazy Class, p.83	*Inline Class* (154), *Collapse Hierarchy* (344)
Long Method, p76	*Extract Method* (110), *Replace Temp With Query* (120), *Replace Method with Method Object* (135), *Decompose Conditional* (238)

※译注：各种坏味道（smell）和各种重构手法（refactorings）之中文译名见目录及正文。

坏味道（smell）	常用的重构手法（Common Refactoring）
Long Parameter List, p78	*Replace Parameter with Method* (292), *Introduce Parameter Object* (295), *Preserve Whole Object* (288)
Message Chains, p84	*Hide Delegate* (157)
Middle Man, p85	*Remove Middle Man* (160), *Inline Method* (117), *Replace Delegation with Inheritance* (355)
Parallel Inheritance Hierarchies, p83	*Move Method* (142), *Move Field* (146)
Primitive Obsession, p81	*Replace Data Value with Object* (175), *Extract Class* (149), *Introduce Parameter Object* (295), *Replace Array with Object* (186), *Replace Type Code with Class* (218), *Replace Type Code with Subclasses* (223), *Replace Type Code with State/Strategy* (227)
Refused Bequest, p87	*Replace Inheritance with Delegation* (352)
Shotgun Surgery, p80	*Move Method* (142), *Move Field* (146), *Inline Class* (154)
Speculative Generality, p83	*Collapse Hierarchy* (344), *Inline Class* (154), *Remove Parameter* (277), *Rename Method* (273)
Switch Statements, p82	*Replace Conditional with Polymorphism* (255), *Replace Type Code with Subclasses* (223), *Replace Type Code with State/Strategy* (227), *Replace Parameter with Explicit Methods* (285), *Introduce Null Object* (260)
Temporary Field, p84	*Extract Class* (149), *Introduce Null Object* (260)

※译注：各种坏味道（smell）和各种重构手法（refactorings）之中文译名见目录及正文。

目录

Contents

1

重构，第一个案例

Refactoring, a First Example

我该怎么开始介绍重构（**refactoring**）呢？按照传统作法，一开始介绍某个东西时，首先应该大致讲讲它的历史、主要原理等等。可是每当有人在会场上介绍这些东西，总是诱发我的瞌睡虫。我的思绪开始游荡，我的眼神开始迷离，直到他或她拿出实例，我才能够提起精神。实例之所以可以拯救我于太虚之中，因为它让我看见事情的真正行进。谈原理，很容易流于泛泛，又很难说明如何实际应用。给出一个实例，却可以帮助我把事情认识清楚。

所以我决定以一个实例作为本书起点。在此过程中我将告诉你很多重构原理，并且让你对重构过程有一点感觉。然后我才能向你提供普通惯见的原理介绍。

但是，面对这个介绍性实例，我遇到了一个大问题。如果我选择一个人型程序，对程序自身的描述和对重构过程的描述就太复杂了，任何读者都将无法掌握（我试了一下，哪怕稍微复杂一点的例子都会超过100页）。如果我选择一个够小以至于容易理解的程序，又恐怕看不出重构的价值。

和任何想要介绍「应用于真实世界中的有用技术」的人一样，我陷入了一个十分典型的两难困境。我将带引你看看如何在一个我所选择的小程序中进行重构，然而坦白说，那个程序的规模根本不值得我们那么做。但是如果我给你看的代码是大系统的一部分，重构技术很快就变得重要起来。所以请你一边观赏这个小例子，一边想像它身处于一个大得多的系统。

1.1 起点

实例非常简单。这是一个影片出租店用的程序，计算每一位顾客的消费金额并打印报表（statement）。操作者告诉程序：顾客租了哪些影片、租期多长，程序便根据租赁时间和影片类型算出费用。影片分为三类：普通片、儿童片和新片。除了计算费用，还要为常客计算点数；点数会随着「租片种类是否为新片」而有不同。

我以数个 classes 表现这个例子中的元素。图 1.1 是一张 UML class diagram（类图），用以显示这些 classes。我会逐一列出这些 classes 的代码。

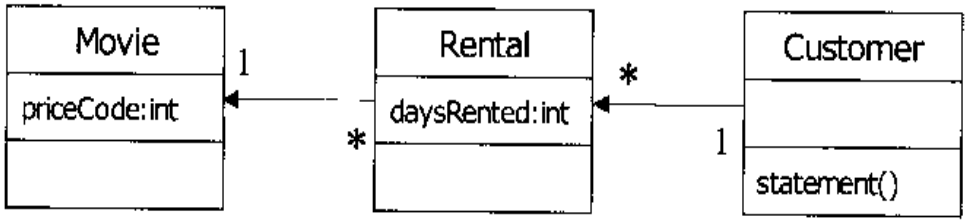


图 1.1 本例一开始的各个 classes。此图只显示最重要的特性。图中所用符号是 UML（Unified Modeling Language，统一建模语言，[Fowler, UML]）。

Movie（影片）

Movie 只是一个简单的 data class（纯数据类）。

```
public class Movie {
  public static final int CHILDRENS = 2;
  public static final int REGULAR = 0;
  public static final int NEW_RELEASE = 1;

  private String _title;    // 名称
  private int _priceCode;   // 价格（代号）

  public Movie(String title, int priceCode){
     _title = title;
     _priceCode = priceCode;
  }
```

```
    public int getPriceCode(){
        return _priceCode;
    }

    public void setPriceCode(int arg){
        _priceCode = arg;
    }

    public String getTitle(){
        return _title;
    }
}
```

Rental（租赁）

Rental class 表示「某个顾客租了一部影片」。

```
class Rental {
  private Movie _movie;          // 影片
  private int _daysRented;       // 租期

  public Rental(Movie movie, int daysRented) {
    _movie = movie;
    _daysRented = daysRented;
  }

  public int getDaysRented() {
    return _daysRented;
  }

  public Movie getMovie() {
    return _movie;
  }
}
```

译注：中文版（本书）支持网站提供本章重构过程中的各阶段完整代码（共分七个阶段），并含测试。网址见于封底。

Customer（顾客）

Customer class 用来表示顾客。就像其他 classes 一样，它也拥有数据和相应的访问函数（accessor）：

```
class Customer {
  private String _name;                          // 姓名
  private Vector _rentals = new Vector();         // 租借记录

  public Customer(String name) {
      _name = name;
  }

  public void addRental(Rental arg) {
      _rentals.addElement(arg);
  }

  public String getName() {
      return _name;
  }

  // 译注：续下页...
```

Customer 还提供了一个用以制造报表的函数（method），图 1.2 显示这个函数带来的交互过程（interactions）。完整代码显示于下一页。

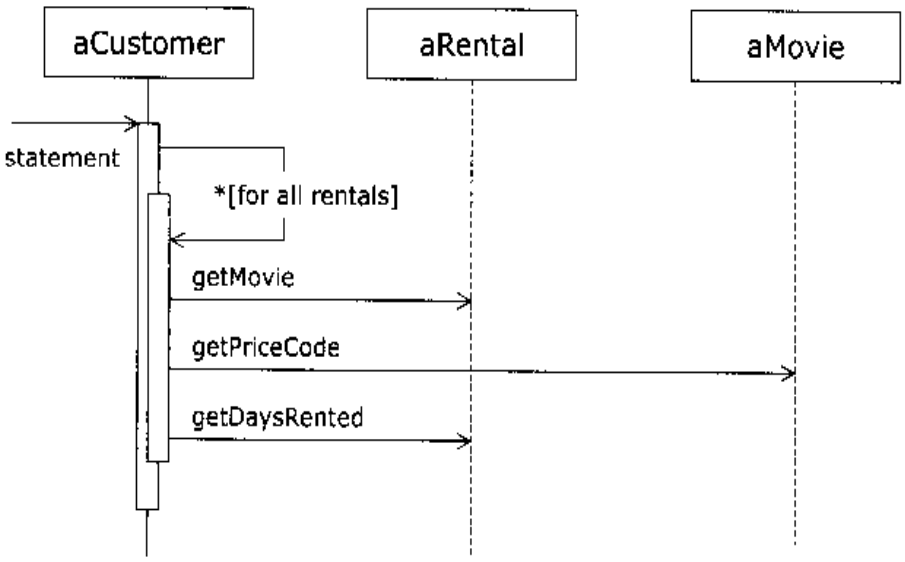


图 1.2 statement() 的交互过程（interactions）

```
public String statement() {
  double totalAmount = 0;           // 总消费金额
  int frequentRenterPoints = 0;     // 常客积点
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";

  while(rentals.hasMoreElements()){
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement(); // 取得一笔租借记录

    //determine amounts for each line
    switch(each.getMovie().getPriceCode()) {  // 取得影片出租价格
      case Movie.REGULAR:               // 普通片
        thisAmount += 2;
        if(each.getDaysRented()>2)
          thisAmount += (each.getDaysRented()-2)*1.5;
        break;

      case Movie.NEW_RELEASE:           // 新片
        thisAmount += each.getDaysRented()*3;
        break;

      case Movie.CHILDRENS:             // 儿童片
        thisAmount += 1.5;
        if(each.getDaysRented()>3)
          thisAmount += (each.getDaysRented()-3)*1.5;
        break;
    }

    // add frequent renter points (累加 常客积点)
    frequentRenterPoints ++;
    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE &&
       each.getDaysRented() > 1)
      frequentRenterPoints ++;

    // show figures for this rental (显示此笔租借数据)
    result += "\t" + each.getMovie().getTitle() + "\t" +
              String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }

  // add footer lines (结尾打印)
  result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points";
  return result;
}
```

对此起始程序的评价

这个起始程序给你留下什么印象？我会说它设计得不好，而且很明显不符合面向对象精神。对于这样一个小程序，这些缺点其实没有什么关系。快速而随性（quick and dirty）地设计一个简单的程序并没有错。但如果这是复杂系统中具有代表性的一段，那么我就真的要对这个程序信心动摇了。Customer 里头那个长长的 `statement()` 做的事情实在太多了，它做了很多原本应该由其他 class 完成的事情。

即便如此，这个程序还是能正常工作。所以这只是美学意义上的判断，只是对丑陋代码的厌恶，是吗？在我们修改这个系统之前的确如此。编译器才不会在乎代码好不好看呢。但是当我们打算修改系统的时候，就涉及到了人，而人在乎这些。差劲的系统是很难修改的，因为很难找到修改点。如果很难找到修改点，程序员就很有可能犯错，从而引入「臭虫」（bugs）。

在这个例子里，我们的用户希望对系统做一点修改。首先他们希望以 HTML 格式打印报表，这样就可以直接在网页上显示，这非常符合潮流。现在请你想一想，这个变化会带来什么影响。看看代码你就会发现，根本不可能在打印 HTML 报表的函数中复用（*reuse*）目前 `statement()` 的任何行为。你惟一可以做的就是编写一个全新的 `htmlStatement()`，大量重复 `statement()` 的行为。当然，现在做这个还不太费力，你可以把 `statement()` 复制一份然后按需要修改就是。

但如果计费标准发生变化，又会发生什么事？你必须同时修改 `statement()` 和 `htmlStatement()`，并确保两处修改的一致性。当你后续还要再修改时，剪贴（copy-paste）问题就浮现出来了。如果你编写的是一个永不需要修改的程序，那么剪剪贴贴就还好，但如果程序要保存很长时间，而且可能需要修改，剪贴行为就会造成潜在的威胁。

现在，第二个变化来了：用户希望改变影片分类规则，但是还没有决定怎么改。他们设想了几种方案，这些方案都会影响顾客消费和常客积点的计算方式。作为一个经验丰富的开发者，你可以肯定：不论用户提出什么方案，你惟一能够获得的保证就是他们一定会在六个月之内再次修改它。

为了应付分类规则和计费规则的变化，程序必须对 statement() 作出修改。但如果我们把 statement() 内的代码拷贝到用以打印 HTML 报表的函数中，我们就必须确保将来的任何修改在两个地方保持一致。随着各种规则变得愈来愈复杂，适当的修改点愈来愈难找，不犯错的机会也愈来愈少。

你的态度也许倾向于「尽量少修改程序」：不管怎么说，它还运行得很好。你心里头牢牢记着那句古老的工程学格言：「如果它没坏，就别动它」。这个程序也许还没坏掉，但它带来了伤害。它让你的生活比较难过，因为你发现很难完成客户所需的修改。这时候就该重构技术粉墨登场了。

如果你发现自己需要为程序添加一个特性，而代码结构使你无法很方便地那么做，那就先重构那个程序，使特性的添加比较容易进行，然后再添加特性。

1.2 重构的第一步

每当我要进行重构的时候，第一个步骤永远相同：我得为即将修改的代码建立一组可靠的测试环境。这些测试是必要的，因为尽管遵循重构准则可以使我避免绝大多数的臭虫引入机会，但我毕竟是人，毕竟有可能犯错。所以我需要可靠的测试。

由于 statement()的运作结果是个字符串（string），所以我首先假设一些顾客，让他们每个人各租几部不同的影片，然后产生报表字符串。然后我就可以拿新字符串和手上已经检查过的参考字符串做比较。我把所有测试都设置好，使得以在命令行输入一条 Java 命令就把它们统统运行起来。运行这些测试只需数秒钟，所以一如你即将见到，我经常运行它们。

测试过程中很重要的一部分，就是测试程序对于结果的回报方式。它们要不说"OK"，表示所有新字符串都和参考字符串一样，要不就印出一份失败清单，显示问题字符串的出现行号。这些测试都属于自我检验（self-checking）。是的，你必须让测试有能力自我检验，否则就得耗费大把时间来回比对，这会降低你的开发速度。

进行重构的时候，我们需要倚赖测试，让它告诉我们是否引入了「臭虫」。好的测试是重构的根本。化时间建立一个优良的测试机制是完全值得的，因为当你修改程序时，好测试会给你必要的安全保障。测试机制在重构领域的地位实在太重要了，我将在第 4 章详细讨论它。

重构之前，首先检查自己是否有一套可靠的测试机制。这些测试必须有自我检验（self-checking）能力。

1.3 分解并重组 statement()

第一个明显引起我注意的就是长得离谱的 statement()。每当看到这样长长的函数，我就想把它大卸八块。要知道，代码区块愈小，代码的功能就愈容易管理，代码的处理和搬移也都愈轻松。

本章重构过程的第一阶段中，我将说明如何把长长的函数切开，并把较小块的代码移至更合适的 class 内。我希望降低代码重复量，从而使新的（打印 HTML 报表用的）函数更容易撰写。

第一个步骤是找出代码的逻辑泥团（*logical clump*）并运用 *Extract Method*（110）。本例一个明显的逻辑泥团就是 `switch` 语句，把它提炼（*extract*）到独立函数中似乎比较好。

和任何重构准则一样，当我提炼一个函数时，我必须知道可能出什么错。如果我提炼得不好，就可能给程序引入臭虫。所以重构之前我需要先想出安全作法。由于先前我已经进行过数次这类重构，所以我已经把安全步骤记录于书后的重构名录（refactoring catalog）中了。

首先我得在这段代码里头找出函数内的局部变量（local variables）和参数（parameters）。我找到了两个：`each` 和 `thisAmount`，前者并未被修改，后者会被修改。任何不会被修改的变量都可以被我当成参数传入新的函数，至于会被修改的变量就需格外小心。如果只有一个变量会被修改，我可以把它当作返回值。`thisAmount` 是个临时变量，其值在每次循环起始处被设为 0，并且在 `switch` 语句之前不会改变，所以我可以直接把新函数的返回值赋予它。

下面两页展示重构前后的代码。重构前的代码在左页，重构后的代码在右页。凡是从函数提炼出来的代码，以及新代码所做的任何修改，只要我觉得不是明显到可以一眼看出，就以粗体字标示出来特别提醒你。本章剩余部分将延续这种左右比对形式。

```
public String statement() {
  double totalAmount = 0;           // 总消费金额
  int frequentRenterPoints = 0;     // 常客积点
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";

  while(rentals.hasMoreElements()){
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement(); // 取得一笔租借记录

    //determine amounts for each line
    switch(each.getMovie().getPriceCode()) {  // 取得影片出租价格
      case Movie.REGULAR:               // 普通片
        thisAmount += 2;
        if(each.getDaysRented()>2)
          thisAmount += (each.getDaysRented()-2)*1.5;
        break;

      case Movie.NEW_RELEASE:           // 新片
        thisAmount += each.getDaysRented()*3;
        break;

      case Movie.CHILDRENS:             // 儿童片
        thisAmount += 1.5;
        if(each.getDaysRented()>3)
          thisAmount += (each.getDaysRented()-3)*1.5;
        break;
    }

    // add frequent renter points (累加 常客积点)
    frequentRenterPoints ++;
    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
        each.getDaysRented() > 1)
      frequentRenterPoints ++;

    // show figures for this rental (显示此笔租借数据)
    result += "\t" + each.getMovie().getTitle() + "\t" +
              String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }

  // add footer lines (结尾打印)
  result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points";
  return result;
}
```

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
  while (rentals.hasMoreElements()){
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    thisAmount = amountFor(each);    // 计算一笔租片费用

    // add frequent renter points
    frequentRenterPoints ++;
    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
        each.getDaysRented() > 1)
      frequentRenterPoints ++;

    // show figures for this rental
    result += "\t" + each.getMovie().getTitle() + "\t" +
              String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }
  // add footer lines
  result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) +
            " frequent renter points";
  return result;
}

private int amountFor(Rental each) {     // 计算一笔租片费用
  int thisAmount = 0;
  switch(each.getMovie().getPriceCode()) {
    case Movie.REGULAR:          // 普通片
      thisAmount += 2;
      if(each.getDaysRented() > 2)
        thisAmount += (each.getDaysRented()-2)*1.5;
      break;
    case Movie.NEW_RELEASE:      // 新片
      thisAmount += each.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:        // 儿童片
      thisAmount += 1.5;
      if(each.getDaysRented() > 3)
        thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
  }
  return thisAmount;
}
```

每次做完这样的修改之后，我都要编译并测试。这一次起头不算太好 — 测试失败了，有两笔测试数据告诉我发生错误。一阵迷惑之后我明白了自己犯的错误。我愚蠢地将 amountFor() 的返回值型别声明为 int，而不是 double。

```
private double amountFor(Rental each) {  // 计算一笔租片费用
  double thisAmount = 0;
  switch(each.getMovie().getPriceCode()) {
    case Movie.REGULAR:              // 普通片
      thisAmount += 2;
      if(each.getDaysRented() > 2)
        thisAmount += (each.getDaysRented()-2)*1.5;
      break;
    case Movie.NEW_RELEASE:          // 新片
      thisAmount += each.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:            // 儿童片
      thisAmount += 1.5;
      if(each.getDaysRented() > 3)
        thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
  }
  return thisAmount;
}
```

我经常犯这种愚蠢可笑的错误，而这种错误往往很难发现。在这里，Java 无怨无尤地把 double 型别转换为 int 型别，而且还愉快地做了取整动作 [Java Spec]。还好此处这个问题很容易发现，因为我做的修改很小，而且我有很好的测试。借着这个意外疏忽，我要阐述重构步骤的本质：由于每次修改的幅度都很小，所以任何错误都很容易发现。你不必耗费大把时间调试，哪怕你和我一样粗心。

重构技术系以微小的步伐修改程序。如果你犯下错误，很容易便可发现它。

由于我用的是 Java，所以我需要对代码做一些分析，决定如何处理局部变量。如果拥有相应的工具，这个工作就超级简单了。Smalltalk 的确拥有这样的工具：Refactoring Browser。运用这个工具，重构过程非常轻松，我只需标示出需要重构的代码，在选单中点选 *Extract Method*，输入新的函数名称，一切就自动搞定。而且工具决不会像我那样犯下愚蠢可笑的错误。我非常盼望早日出现 Java 版本的重构工具！

现在，我已经把原本的函数分为两块，可以分别处理它们。我不喜欢 amountFor() 内的某些变量名称，现在是修改它们的时候。

下面是原本的代码：

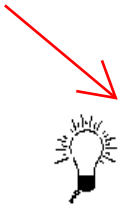
```
private double amountFor(Rental each) {  // 计算一笔租片费用
  double thisAmount = 0;
  switch(each.getMovie().getPriceCode()) {
    case Movie.REGULAR:              // 普通片
      thisAmount += 2;
      if(each.getDaysRented() > 2)
         thisAmount += (each.getDaysRented()-2)*1.5;
      break;
    case Movie.NEW_RELEASE:          // 新片
      thisAmount += each.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:            // 儿童片
      thisAmount += 1.5;
      if(each.getDaysRented() > 3)
         thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
  }
  return thisAmount;
}
```

下面是易名后的代码：

```
private double amountFor(Rental aRental) {    // 计算一笔租片费用
    double result= 0;
    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:          // 普通片
            result += 2;
            if (aRental.getDaysRented() > 2)
                result += (aRental.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:      // 新片
            result += aRental.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:        // 儿童片
            result += 1.5;
            if (aRental.getDaysRented() > 3)
                result += (aRental.getDaysRented() - 3) * 1.5;
            break;
    }
    return result;
}
```

易名之后我需要重新编译并测试，确保没有破坏任何东西。

更改变量名称是值得的行为吗？绝对值得。好的代码应该清楚表达出自己的功能，变量名称是代码清晰的关键。如果为了提高代码的清晰度，需要修改某些东西的名字，大胆去做吧。只要有良好的查找/替换工具，更改名称并不困难。语言所提供的强型别检验（strong typing）以及你自己的测试机制会指出任何你遗漏的东西。记住：



> 任何一个傻瓜都能写出计算机可以理解的代码。惟有写出人类容易理解的代码，才是优秀的程序员。

代码应该表现自己的目的，这一点非常重要。阅读代码的时候，我经常进行重构。这样，随着对程序的理解逐渐加深，我也就不断地把这些理解嵌入代码中，这么一来才不会遗忘我曾经理解的东西。

搬移「金额计算」代码

观察 amountFor()时，我发现这个函数使用了来自 Rental class 的信息，却没有使用来自 Customer class 的信息。

```
class Customer...
  private double amountFor(Rental aRental) {
        double result = 0;
        switch (aRental.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (aRental.getDaysRented() > 2)
                    result += (aRental.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += aRental.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (aRental.getDaysRented() > 3)
                    result += (aRental.getDaysRented() - 3) * 1.5;
                break;
        }
        return result;
    }
```

这立刻使我怀疑它是否被放错了位置。绝大多数情况下，函数应该放在它所使用的数据的所属 object（或说 class）内，所以 amountFor()应该移到 Rental class 去。为了这么做，我要运用 *Move Method*（142）。首先把代码拷贝到 Rental class 内，调整代码使之适应新家，然后重新编译。像下面这样：

```
class Rental...
        double getCharge() {
            double result = 0;
            switch (getMovie().getPriceCode()) {
                case Movie.REGULAR:
                    result += 2;
                    if (getDaysRented() > 2)
                        result += (getDaysRented() - 2) * 1.5;
                    break;
                case Movie.NEW_RELEASE:
                    result += getDaysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    result += 1.5;
                    if (getDaysRented() > 3)
                        result += (getDaysRented() - 3) * 1.5;
                    break;
            }
            return result;
        }
```

在这个例子里，「适应新家」意味去掉参数。此外，我还要在搬移的同时变更函数名称。

现在我可以测试新函数是否正常工作。只要改变 Customer.amountFor()函数内容，使它委托（*delegate*）新函数即可：

```
class Customer...
        private double amountFor(Rental aRental) {
            return aRental.getCharge();
        }
```

现在我可以编译并测试，看看有没有破坏了什么东西。

下一个步骤是找出程序中对于旧函数的所有引用（*reference*）点，并修改它们，让它们改用新函数。下面是原本的程序：

```
class Customer...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();

            thisAmount = amountFor(each);

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                                    each.getDaysRented() > 1)
                frequentRenterPoints ++;

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;

    }
```

本例之中，这个步骤很简单，因为我才刚刚产生新函数，只有一个地方使用了它。一般情况下你得在可能运用该函数的所有 classes 中查找一遍。

```
class Customer
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();

            thisAmount = each.getCharge();

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                                    each.getDaysRented() > 1)
                frequentRenterPoints ++;

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;

    }
```

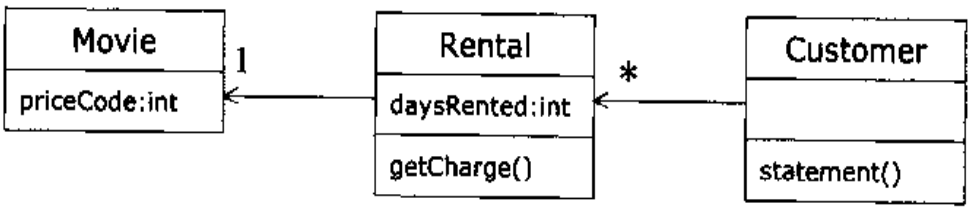


图 1.3 搬移「金额计算」函数后，所有 classes 的状态（*state*）

做完这些修改之后（图 1.3），下一件事就是去掉旧函数。编译器会告诉我是否我漏掉了什么。然后我进行测试，看看有没有破坏什么东西。

有时候我会保留旧函数，让它调用新函数。如果旧函数是一个 public 函数，而我又不想修改其他 class 的接口，这便是一种有用的手法。

当然我还想对 Rental.getCharge()做些修改，不过暂时到此为止，让我们回到 Customer.statement()：

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        thisAmount = each.getCharge();

        // add frequent renter points
        frequentRenterPoints ++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                                each.getDaysRented() > 1)
            frequentRenterPoints ++;

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle()+ "\t" +
                  String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;

    }
    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";
    return result;

}
```

下一件引我注意的事是：thisAmount 如今变成多余了。它接受 each.getCharge() 的执行结果，然后就不再有任何改变。所以我可以运用 *Replace Temp with Query*（120）把 thisAmount 除去：

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        // add frequent renter points
        frequentRenterPoints ++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
          each.getDaysRented() > 1) frequentRenterPoints ++;

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle()+ "\t" +
                  String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();

    }
    //add footer lines
    result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints)
           + " frequent renter points";
    return result;

}
}
```

做完这份修改，我立刻编译并测试，保证自己没有破坏任何东西。

我喜欢尽量除去这一类临时变量。临时变量往往形成问题，它们会导致大量参数被传来传去，而其实完全没有这种必要。你很容易失去它们的踪迹，尤其在长长的函数之中更是如此。当然我这么做也需付出性能上的代价，例如本例的费用就被计算了两次。但是这很容易在 Rental class 中被优化。而且如果代码有合理的组织和管理，优化会有很好的效果。我将在 p.69 的「重构与性能」一节详谈这个问题。

提炼「常客积点计算」代码

下一步要对「常客积点计算」做类似处理。点数的计算视影片种类而有不同，不过不像收费规则有那么多变化。看来似乎有理由把积点计算责任放在 Rental class 身上。首先我们需要针对「常客积点计算」这部分代码（以下粗体部分）运用 *Extract Method*（110）重构准则：

```
    public String statement() {

        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
              && each.getDaysRented() > 1) frequentRenterPoints ++;

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();

        }
        //add footer lines
        result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints)
            + " frequent renter points";
        return result;

    }
}
```

再一次我又要寻找局部变量。这里再一次用到了 each，而它可以被当作参数传入新函数中。另一个临时变量是 frequentRenterPoints。本例中的它在被使用之前已经先有初值，但提炼出来的函数并没有读取该值，所以我们不需要将它当作参数传进去，只需对它执行「附添赋值动作」（*appending assignment*, operator+=）就行了。

我完成了函数的提炼，重新编译并测试；然后做一次搬移，再编译、再测试。重构时最好小步前进，如此一来犯错的几率最小。

```
class Customer...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
        }

        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;
    }

class Rental...
        int getFrequentRenterPoints() {
            if ((getMovie().getPriceCode() == Movie.NEW_RELEASE)
                && getDaysRented() > 1)
                return 2;
            else
                return 1;
        }
```

我利用重构前后的 UML（Unified Modeling Language，统一建模语言）图形（图 1.4 至图 1.7）总结刚才所做的修改。和先前一样，左页是修改前的图，右页是修改后的图。

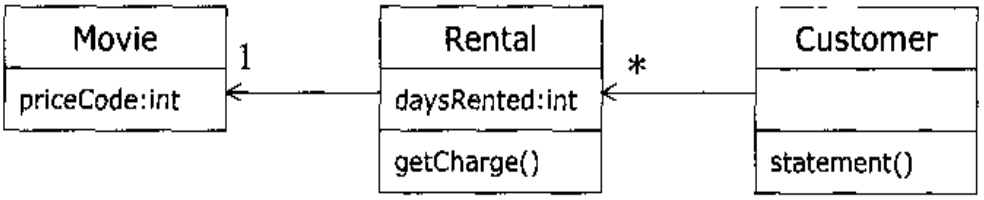


图 1.4 「常客积点计算」函数被提炼及搬移之前的 class diagrams

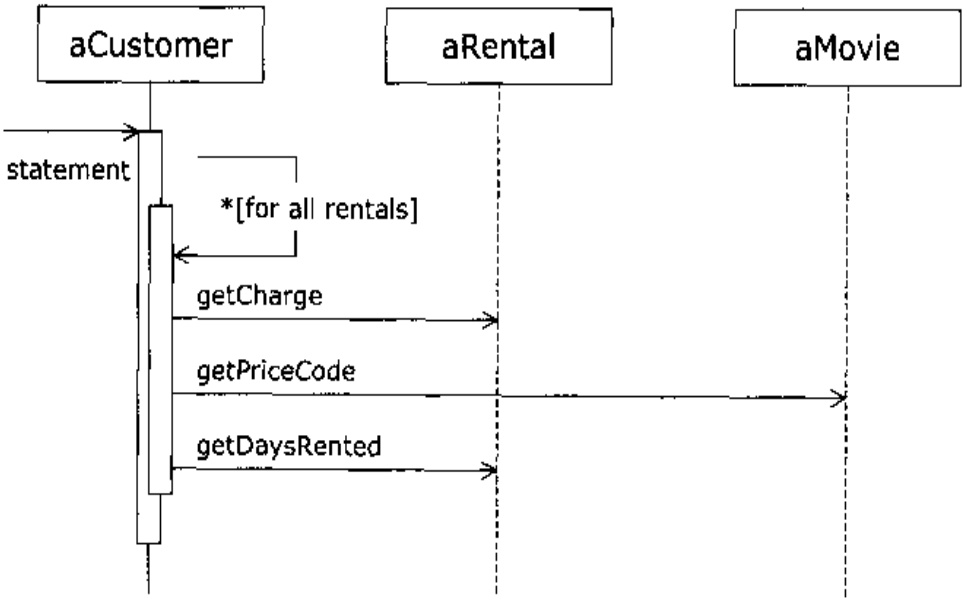


图 1.5 「常客积点计算」函数被提炼及搬移之前的 sequence diagrams

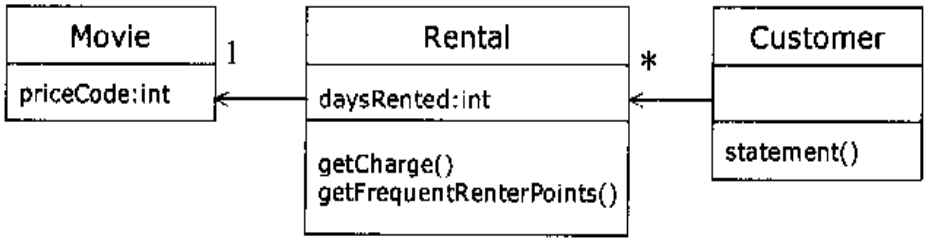


图 1.6 「常客积点计算」函数被提炼及搬移之后的 class diagrams

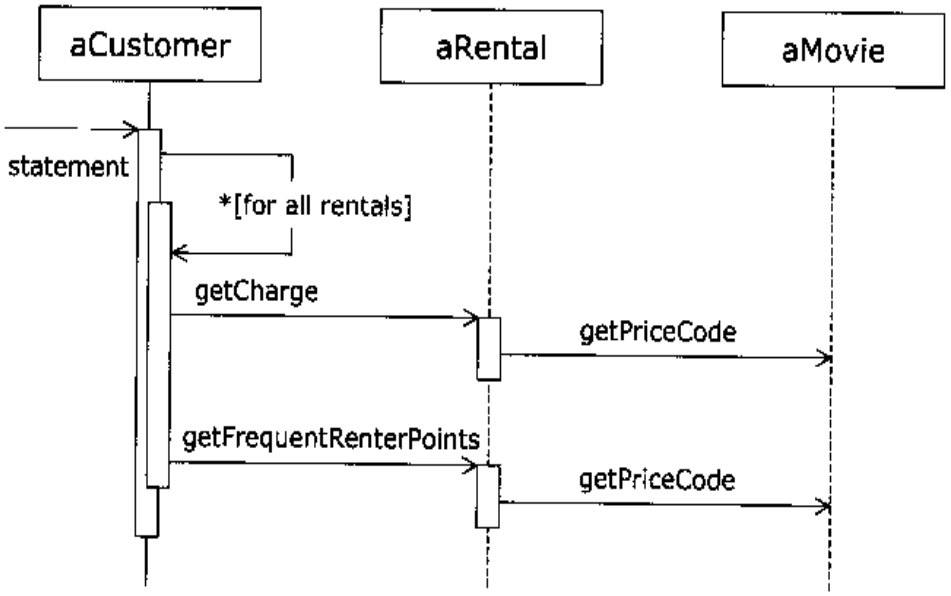


图 1.7 「常客积点计算」函数被提炼及搬移之后的 sequence diagrams

去除临时变量

正如我在前面提过的，临时变量可能是个问题。它们只在自己所属的函数中有效，所以它们会助长「冗长而复杂」的函数。这里我们有两个临时变量，两者都是用来从 Customer 对象相关的 Rental 对象中获得某个总量。不论 ASCII 版或 HTML 版都需要这些总量。我打算运用 *Replace Temp with Query*(120)，并利用所谓的 ***query method*** 来取代 totalAmount 和 frequentRentalPoints 这两个临时变量。由于 class 内的任何函数都可以取用（调用）上述所谓 ***query methods***，所以它能够促进较干净的设计，而非冗长复杂的函数：

```
class Customer...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
        }

        //add footer lines
        result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;
    }
```

首先我以 Customer class 的 getTotalCharge()取代 totalAmount:

```
class Customer...

    public String statement() {
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }

        //add footer lines
        result += "Amount owed is " +
                  String.valueOf(getTotalCharge()) + "\n";
        result += "You earned " +
                  String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;
    }

    // 译注：此即所谓 query method
    private double getTotalCharge() {
        double result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.getCharge();
        }
        return result;
    }
```

这并不是 *Replace Temp with Query*（120）的最简单情况。由于 totalAmount 在循环内部被赋值，我不得不把循环复制到 ***query method*** 中。

重构之后，重新编译并测试，然后以同样手法处理 frequentRenterPoints：

```
class Customer...
    public String statement() {
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }

        //add footer lines
        result += "Amount owed is " +
                  String.valueOf(getTotalCharge()) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";
        return result;
    }
```

```
public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle()+ "\t" +
                  String.valueOf(each.getCharge()) + "\n";
    }

    //add footer lines
    result += "Amount owed is " +
              String.valueOf(getTotalCharge()) + "\n";
    result += "You earned " +
              String.valueOf(getTotalFrequentRenterPoints()) +
              " frequent renter points";
    return result;
}

// 译注：此即所谓 query method
private int getTotalFrequentRenterPoints(){
    int result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += each.getFrequentRenterPoints();
    }
    return result;
}
```

图 1.8 至图 1.11 分别以 UML class diagram（类图）和 interaction diagram（交互作用图）展示 `statement()` 重构前后的变化。

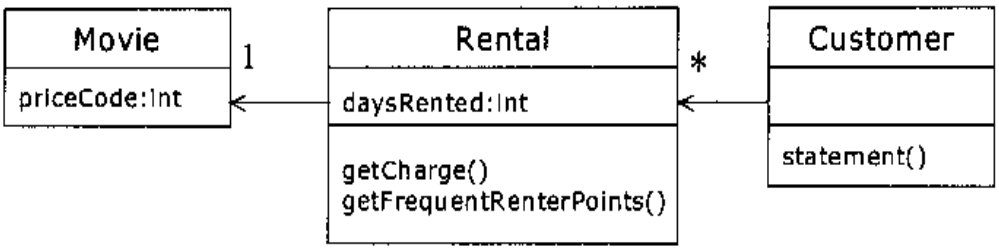


图 1.8 「总量计算」函数被提炼前的 class diagram

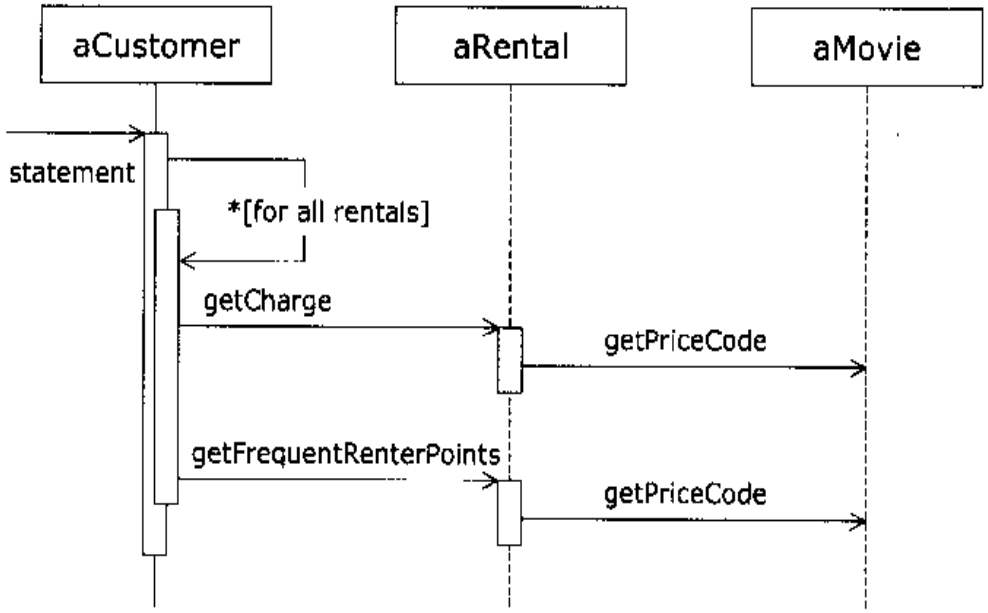


图 1.9 「总量计算」函数被提炼前的 sequence diagram

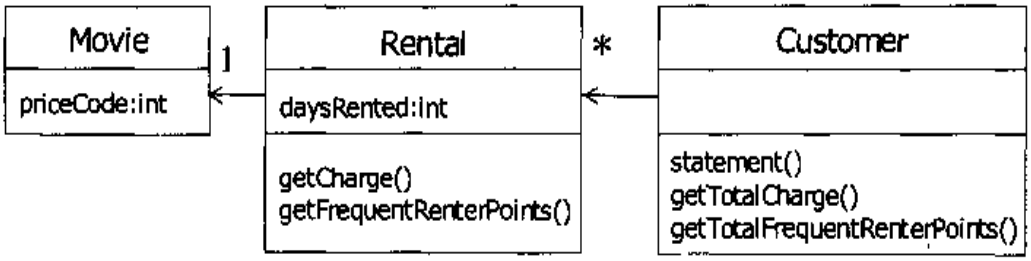


图 1.10 「总量计算」函数被提炼后的 class diagram

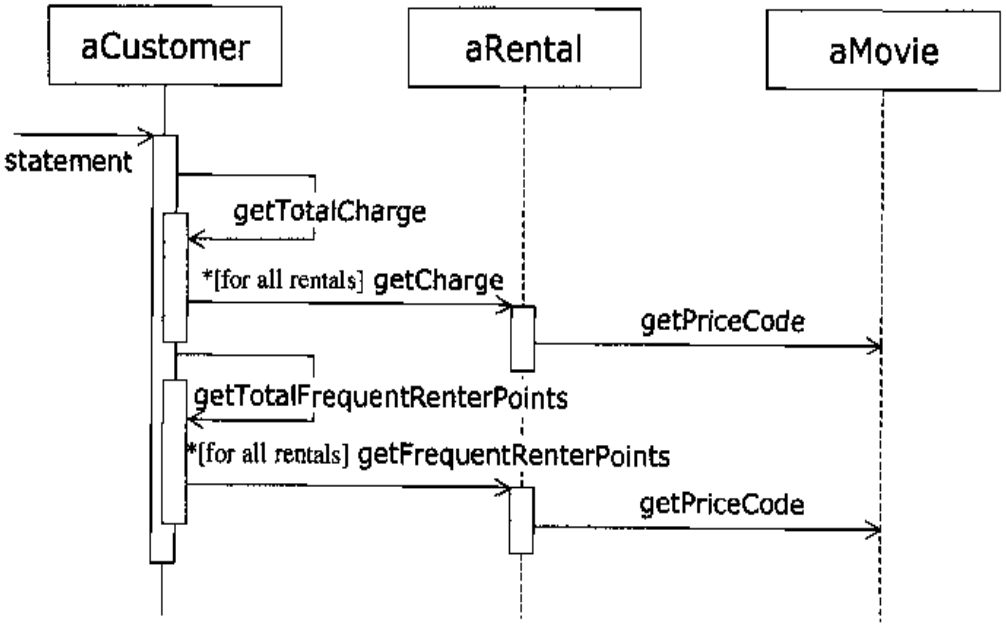


图 1.11 「总量计算」函数被提炼后的 sequence diagram

做完这次重构，有必要停下来思考一下。大多数重构都会减少代码总量，但这次却增加了代码总量，那是因为 Java 1.1 需要大量语句（statements）来设置一个总和（*summing*）循环。哪怕只是一个简单的总和循环，每个元素只需一行代码，外围的支持代码也需要六行之多。这其实是任何程序员都熟悉的习惯写法，但代码数量还是太多了。

这次重构存在另一个问题，那就是性能。原本代码只执行 while 循环一次，新版本要执行三次。如果 while 循环耗时很多，就可能大大降低程序的性能。单单为了这个原因，许多程序员就不愿进行这个重构动作。但是请注意我的用词：如果和可能。除非我进行评测（*profile*），否则我无法确定循环的执行时间，也无法知道这个循环是否被经常使用以至于影响系统的整体性能。重构时你不必担心这些，优化时你才需要担心它们，但那时候你已处于一个比较有利的位置，有更多选择可以完成有效优化（见 p.69 的讨论）。

现在，Customer class 内的任何代码都可以取用这些 ***query methods*** 了。如果系统他处需要这些信息，也可以轻松地将 ***query methods*** 加入 Customer class 接口。如果没有这些 ***query methods***，其他函数就必须了解 Rental class，并自行建立循环。在一个复杂系统中，这将使程序的编写难度和维护难度大大增加。

你可以很明显看出来，htmlStatement()和 statement()是不同的。现在，我应该脱下「重构」的帽子，戴上「添加功能」的帽子。我可以像下面这样编写 htmlStatement()，并添加相应测试：

```
public String htmlStatement() {
        Enumeration rentals = _rentals.elements();
        String result = "<H1>Rentals for <EM>" + getName() +
                        "</EM></H1><P>\n";
        while (rentals.hasMoreElements()) {
                Rental each = (Rental) rentals.nextElement();
                //show figures for each rental
                result += each.getMovie().getTitle()+ ": " +
                  String.valueOf(each.getCharge()) + "<BR>\n";
        }
        //add footer lines
        result +=  "<P>You owe <EM>" +
                   String.valueOf(getTotalCharge()) + .
                   "</EM><P>\n";
        result += "On this rental you earned <EM>" -
                String.valueOf(getTotalFrequentRenterPoints()) +
                "</EM> frequent renter points<P>";
        return result;
}
```

通过计算逻辑的提炼，我可以完成一个 `htmlStatement()`，并复用（*reuse*）原本 `statement()`内的所有计算。我不必剪剪贴贴，所以如果计算规则发生改变，我只需在程序中做一处修改。完成其他任何类型的报表也都很快而且很容易。这次重构并不花很多时间，其中大半时间我用来弄清楚代码所做的事，而这是我无论如何都得做的。

前述有些重构码系从 ASCII 版本里头拷贝过来 — 主要是循环设置部分。更深入的重构动作可以清除这些重复代码。我可以把处理表头（header）、表尾（footer）和报表细目的代码都分别提炼出来。在 *Form Template Method*（345）实例中，你可以看到如何做这些动作。但是，现在用户又开始嘀咕了，他们准备修改影片分类规则。我们尚未清楚他们想怎么做，但似乎新分类法很快就要引入，现有的分类法马上就要变更。与之相应的费用计算方式和常客积点计算方式都还待决定，现在就对程序做修改，肯定是愚蠢的。我必须进入费用计算和常客积点计算中，把「因条件而异的代码」（译注：指的是 `switch` 语句内的 `case` 子句）替换掉，这样才能为将来的改变镀上一层保护膜。现在，请重新戴回「重构」这顶帽子。

1.4 运用多态（polymorphism）取代与价格相关的条件逻辑

这个问题的第一部分是 switch 语句。在另一个对象的属性（*attribute*）基础上运用 switch 语句，并不是什么好主意。如果不得不使用，也应该在对象自己的数据上使用，而不是在别人的数据上使用。

```
class Rental...
  double getCharge() {
      double result = 0;
      switch (getMovie().getPriceCode()) {
          case Movie.REGULAR:
              result += 2;
              if (getDaysRented() > 2)
                  result += (getDaysRented() - 2) * 1.5;
              break;
          case Movie.NEW_RELEASE:
              result += getDaysRented() * 3;
              break;
          case Movie.CHILDRENS:
              result += 1.5;
              if (getDaysRented() > 3)
                  result += (getDaysRented() - 3) * 1.5;
              break;
      }
      return result;
  }
```

这暗示 getCharge()应该移到 Movie class 里头去：

```
class Movie...
  double getCharge(int daysRented) {
     double result = 0;
     switch (getPriceCode()) {
        case Movie.REGULAR:
           result += 2;
           if (daysRented > 2)
              result += (daysRented - 2) * 1.5;
           break;
        case Movie.NEW_RELEASE:
           result += daysRented * 3;
           break;
        case Movie.CHILDRENS:
           result += 1.5;
           if (daysRented > 3)
              result += (daysRented - 3) * 1.5;
           break;
     }
     return result;
  }
```

为了让它得以运作，我必须把「租期长度」作为参数传递进去。当然，「租期长度」来自 Rental 对象。计算费用时需要两份数据：「租期长度」和「影片类型」。为什么我选择「将租期长度传给 Movie 对象」而不是「将影片类型传给 Rental 对象」呢？因为本系统可能发生的变化是加入新影片类型，这种变化带有不稳定倾向。如果影片类型有所变化，我希望掀起最小的涟漪，所以我选择在 Movie 对象内计算费用。

我把上述计费方法放进 Movie class 里头，然后修改 Rental 的 getCharge()，让它使用这个新函数（图 1.12 和图 1.13）：

```
class Rental...
  double getCharge() {
        return _movie.getCharge(_daysRented);
  }
```

搬移 getCharge()之后，我以相同手法处理常客积点计算。这样我就把根据影片类型而变化的所有东西，都放到了影片类型所属的 class 中。以下是重构前的代码：

```
class Rental...
 int getFrequentRenterPoints() {
     if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
          getDaysRented() > 1)
         return 2;
     else
         return 1;
 }
```

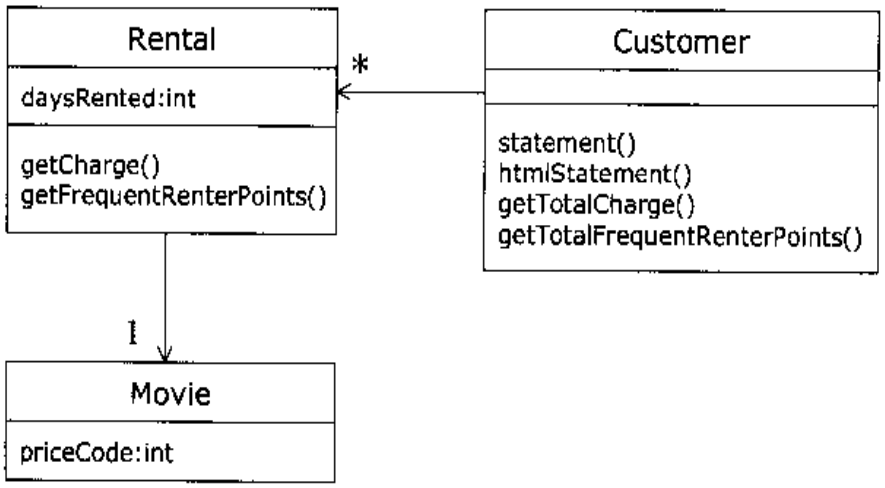


图 1.12 本节所讨论的两个函数被移到 Movie class 内之前系统的 class diagram

重构如下：

```
class Rental...
  int getFrequentRenterPoints() {
     return _movie.getFrequentRenterPoints(_daysRented);
  }

class Movie...
  int getFrequentRenterPoints(int daysRented) {
        if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
            return 2;
        else
            return 1;
  }
```

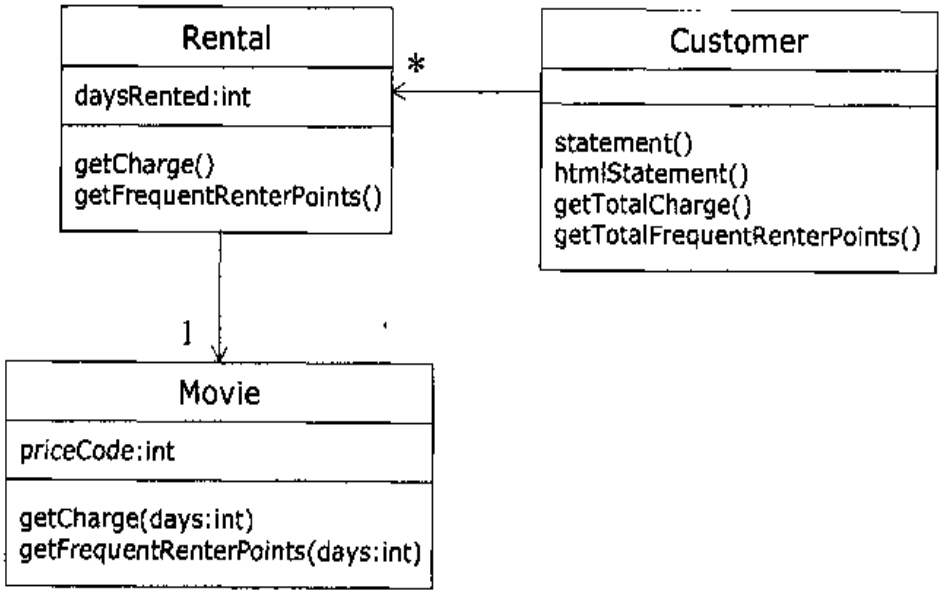


图 1.13 本节所讨论的两个函数被移到 Movie class 内之后系统的 class diagram

终于……我们来到继承（inheritance）

我们有数种影片类型，它们以不同的方式回答相同的问题。这听起来很像 subclasses 的工作。我们可以建立 Movie 的三个 subclasses，每个都有自己的计费法（图 1.14）。

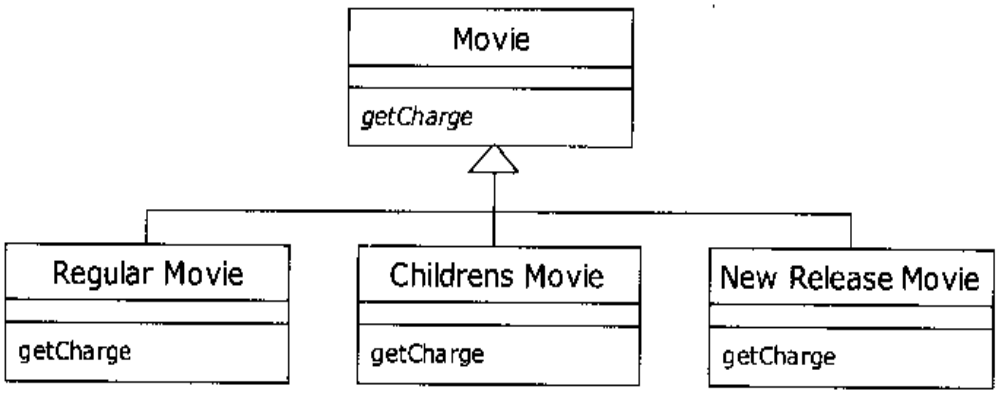


图 1.14 以继承机制表现不同的影片类型

这么一来我就可以运用多态（polymorphism）来取代 switch 语句了。很遗憾的是这里有个小问题，不能这么干。一部影片可以在生命周期内修改自己的分类，一个对象却不能在生命周期内修改自己所属的 class。不过还是有一个解决方法：***State*** pattern（模式）[Gang of Four]。运用它之后，我们的 classes 看起来像图 1.15。

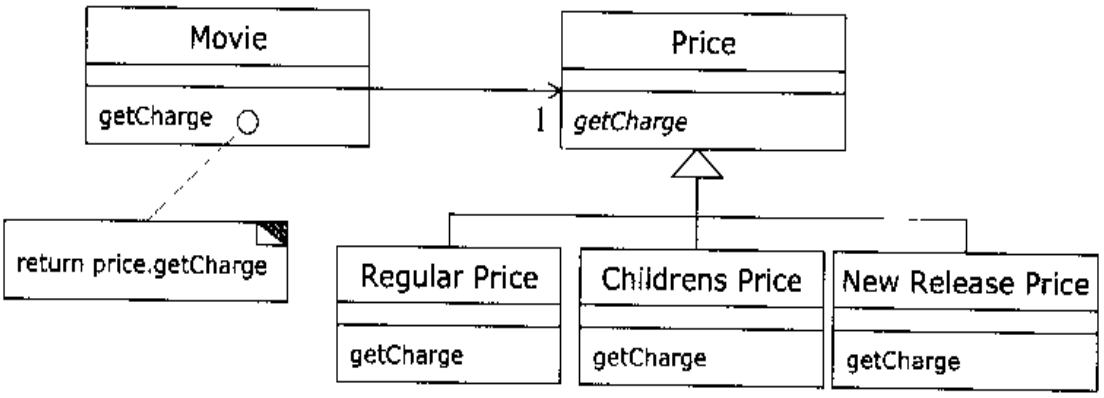


图 1.15 运用 ***State*** pattern（模式）表现不同的影片

加入这一层间接件，我们就可以在 `Price` 对象内进行 subclassing 动作（译注：一如图 1.15），于是便可在任何必要时刻修改价格。

如果你很熟悉 Gang of Four 所列的各种模式（patterns），你可能会问：「这是一个 ***State*** 还是一个 ***Strategy***？」答案取决于 `Price` class 究竟代表计费方式（此时我喜欢把它叫做 `Pricer` 或 `PricingStrategy`），或是代表影片的某个状态（*state*，例如「*Star Trek X* 是一部新片」）。在这个阶段，对于模式（和其名称）的选择反映出你对结构的想法。此刻我把它视为影片的某种状态（*state*）。如果未来我觉得 ***Strategy*** 能更好地说明我的意图，我会再重构它，修改名字，以形成 ***Strategy***。

为了引入 ***State*** 模式，我使用三个重构准则。首先运用 *Replace Type Code with State/Strategy*（227），将「与型别相依的行为」（type code behavior）搬移至 ***State*** 模式内。然后运用 *Move Method*（142）将 `switch` 语句移到 `Price` class 里头。最后运用 *Replace Conditional with Polymorphism*（255）去掉 `switch` 语句。

首先我要使用 *Replace Type Code with State/Strategy*（227）。第一步骤是针对「与型别相依的行为」使用 *Self Encapsulate Field*（171），确保任何时候都通过 **getting** 和 **setting** 两个函数来运用这些行为。由于多数代码来自其他 classes，所以多数函数都已经使用 **getting** 函数。但构造函数（constructor）仍然直接访问价格代号（译注：程序中的 _priceCode）：

```
class Movie...
    public Movie(String name, int priceCode) {
        _title = name;
        _priceCode = priceCode;
    }
```

我可以用一个 **setting** 函数来代替：

```
class Movie
     public Movie(String name, int priceCode) {
        _title = name;
        setPriceCode(priceCode);     // 译注：这就是一个 set method
     }
```

然后编译并测试，确保没有破坏任何东西。现在我加入新 class，并在 Price 对象中提供「与型别相依的行为」。为了实现这一点，我在 Price 内加入一个抽象函数（abstract method），并在其所有 subclasses 中加上对应的具体函数（concrete method）：

```
abstract class Price {
       abstract int getPriceCode();       // 取得价格代号
}
class ChildrensPrice extends Price {
       int getPriceCode() {
              return Movie.CHILDRENS;
       }
}
class NewReleasePrice extends Price {
       int getPriceCode() {
              return Movie.NEW_RELEASE;
       }
}
class RegularPrice extends Price {
       int getPriceCode() {
              return Movie.REGULAR;
       }
}
```

现在我可以编译这些新 classes 了。

现在，我需要修改 Movie class 内的「价格代号」访问函数（***get/set*** 函数，如下），让它们使用新 class。下面是重构前的样子：

```
public int getPriceCode() {
        return _priceCode;
}
public setPriceCode (int arg) {
         priceCode = arg;
}
private int _priceCode;
```

这意味我必须在 Movie class 内保存一个 Price 对象，而不再是保存一个 _priceCode 变量。此外我还需要修改访问函数（译注：即 get/set 函数）：

```
class Movie...
  public int getPriceCode() {              // 取得价格代号
     return _price.getPriceCode();
  }
  public void setPriceCode(int arg) {      // 设定价格代号
      switch (arg) {
          case REGULAR:                    // 普通片
               _price = new RegularPrice();
               break;
          case CHILDRENS:                  // 儿童片
               _price = new ChildrensPrice();
               break;
          case NEW_RELEASE:                // 新片
               _price = new NewReleasePrice();
               break;
          default:
             throw new IllegalArgumentException("Incorrect Price Code");
      }
  }
  private Price _price;
```

现在我可以重新编译并测试，那些比较复杂的函数根本不知道世界已经变了个样儿。

现在我要对getCharge()实施 *Move Method*（142）。下面是重构前的代码：

```
class Movie...
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
        return result;
    }
```

搬移动作很简单。下面是重构后的代码：

```
class Movie...
    double getCharge(int daysRented) {
        return _price.getCharge(daysRented);
    }

class Price...
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
    return result;
}
```

搬移之后，我就可以开始运用 *Replace Conditional with Polymorphism*（255）了。下面是重构前的代码：

```
class Price...
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
        return result;
    }
```

我的作法是一次取出一个case分支，在相应的class内建立一个覆写函数(overriding method)。先从 **RegularPrice** 开始：

```
class RegularPrice...
    double getCharge(int daysRented) {
        double result = 2;
        if (daysRented > 2)
            result += (daysRented - 2) * 1.5;
        return result;
    }
```

这个函数覆写（*overrides*）了父类中的 case 语句，而我暂时还把后者留在原处不动。现在编译并测试，然后取出下一个 case 分支，再编译并测试。（为了保证被执行的的确是subclass代码，我喜欢故意丢一个错误进去，然后让它运行，让测试失败。噢，我是不是有点太偏执了？）

```
class ChildrensPrice...
    double getCharge(int daysRented) {
        double result = 1.5;
        if (daysRented > 3)
            result += (daysRented - 3) * 1.5;
        return result;
    }

class NewReleasePrice...
    double getCharge(int daysRented) {
        return daysRented * 3;
    }
```

处理完所有 case 分支之后，我就把 Price.getCharge()声明为 abstract：

```
class Price...
    abstract double getCharge(int daysRented);
```

现在我可以运用同样手法处理 getFrequentRenterPoints()。重构前的样子如下（译注：其中有「与型别相依的行为」，也就是「判断是否为新片」那个动作）：

```
class Movie...
     int getFrequentRenterPoints(int daysRented) {
       if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
            return 2;
       else
            return 1;
     }
```

首先我把这个函数移到 Price class 里头：

```
Class Movie...
     int getFrequentRenterPoints(int daysRented) {
       return _price.getFrequentRenterPoints(daysRented);
     }
Class Price...
     int getFrequentRenterPoints(int daysRented) {
       if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
            return 2;
       else
            return 1;
     }
```

但是这一次我不把 superclass 函数声明为 abstract。我只是为「新片类型」产生一个覆写函数（*overriding method*），并在 superclass 内留下一个已定义的函数，使它成为一种缺省行为。

```
// 译注：在新片中产生一个覆写函数（overriding method）
Class NewReleasePrice
     int getFrequentRenterPoints(int daysRented) {
       return (daysRented > 1) ? 2: 1;
     }

// 译注：在 superclass 内保留它，使它成为一种缺省行为
Class Price...
    int getFrequentRenterPoints(int daysRented);
       return 1;
    }
```

引入 ***State*** 模式花了我不少力气，值得吗？这么做的收获是：如果我要修改任何与价格有关的行为，或是添加新的定价标准，或是加入其他取决于价格的行为，程序的修改会容易得多。这个程序的其余部分并不知道我运用了 ***State*** 模式。对于我目前拥有的这么几个小量行为来说，任何功能或特性上的修改也许都称不上什么困难，但如果在一个更复杂的系统中，有十多个与价格相关的函数，程序的修改难易度就会有很大的区别。以上所有修改都是小步骤进行，进度似乎太过缓慢，但是没有任何一次我需要打开调试器（debugger），所以整个过程实际上很快就过去了。我书写本章所用的时间，远比修改那些代码的时间多太多了。

现在我已经完成了第二个重要的重构行为。从此，修改「影片分类结构」，或是改变「费用计算规则」、改变常客积点计算规则，都容易多了。图 1.16 和图 1.17 描述 ***State*** 模式对于价格信息所起的作用。

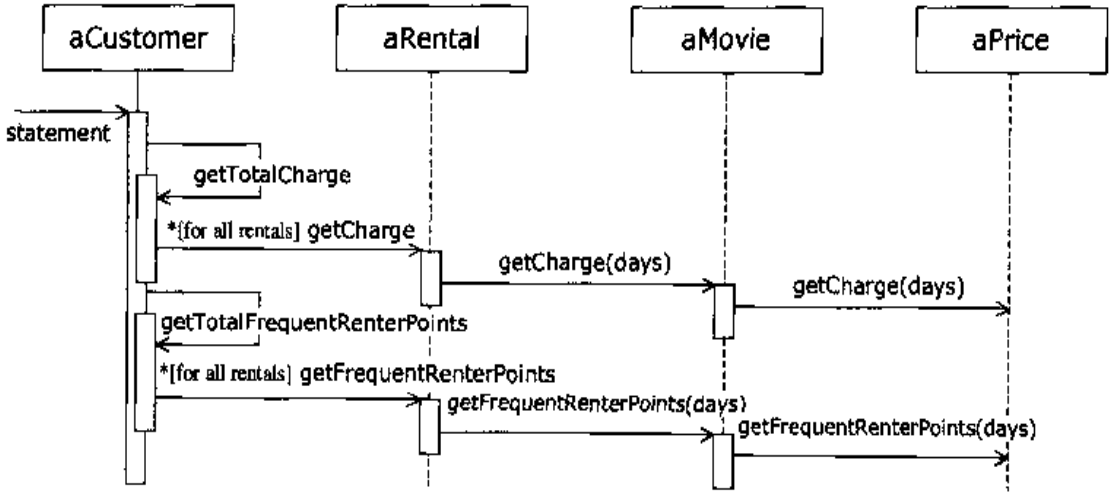


图 1.16 运用 ***State*** pattern（模式）当时的 interaction diagram

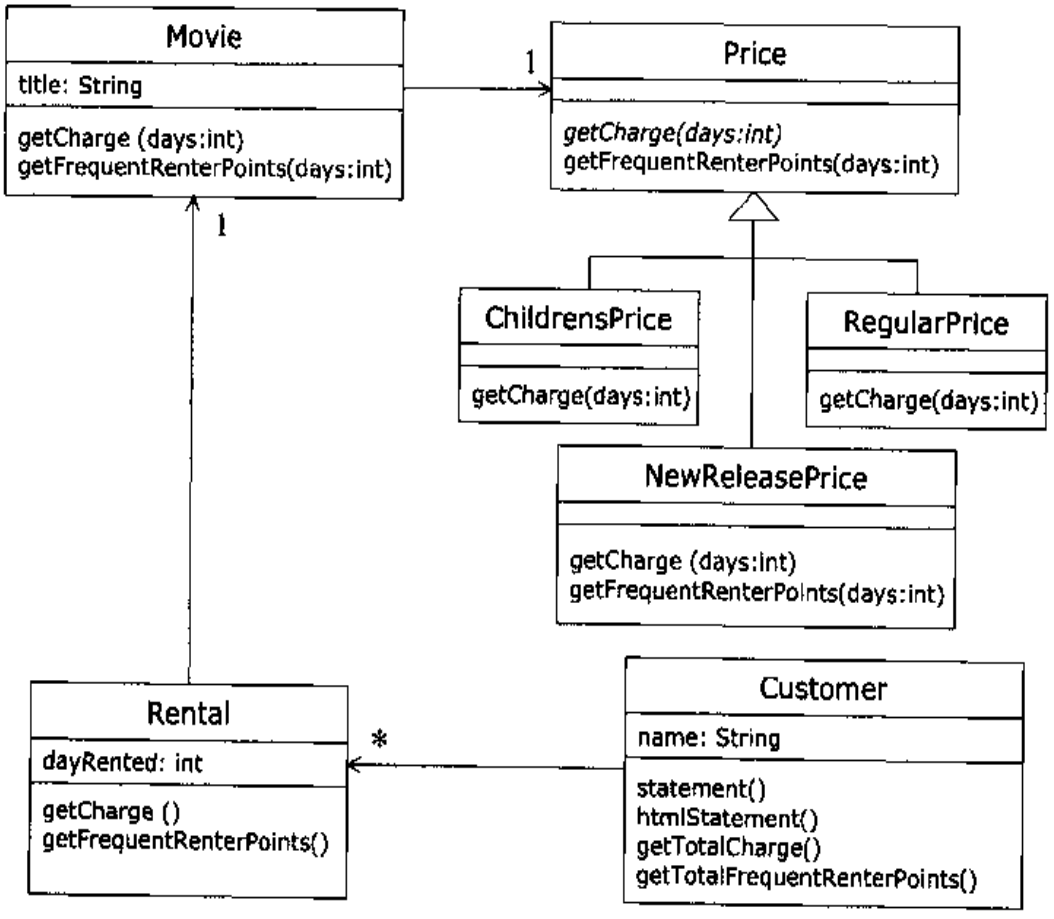


图 1.17 加入 ***State*** pattern（模式）之后的 class diagram

1.5 结语

这是一个简单的例子，但我希望它能让你对于「重构是什么样子」有一点感觉。例中我已经示范了数个重构准则，包括 *Extract Method*（110）、*Move Method*（142）、*Replace Conditional with Polymorphism*（255）、*Self Encapsulate Field*（171）、*Replace Type Code with State/Strategy*（227）。所有这些重构行为都使责任的分配更合理，代码的维护更轻松。重构后的程序风格，将十分不同于过程化（*procedural*）风格，后者也许是某些人习惯的风格。不过一旦你习惯了这种重构后的风格，就很难再回到（再满足于）结构化风格了。

这个例子给你上的最重要一课是「重构的节奏」：测试、小修改、测试、小修改、测试、小修改……。正是这种节奏让重构得以快速而安全地前进。

如果你看懂了前面的例子，你应该已经理解重构是怎么回事了。现在，让我们了解一些背景、原理和理论（不太多！）。

> 译注：中文版（本书）支持网站提供本章重构过程中的各阶段完整代码（共分七个阶段），并含测试。网址见于封底。

2

重构原则

Principles in Refactoring

前面所举的例子应该已经让你对重构（**refactoring**）有了一个良好的感受。现在，我们应该回头看看重构的关键原则，以及重构时需要考虑的某些问题。

2.1 何谓重构

我总是不太乐意为什么东西下定义，因为每个人对任何东西都有自己的定义。但是当你写一本书时，你总得选择自己满意的定义。在重构这个题目上，我的定义以Ralph Johnson团队和其他相关研究成果为基础。

首先要说明的是：视上下文不同，「重构」这个词有两种不同的定义。你可能会觉得这挺烦人的（我就是这么想），不过处理自然语言本来就是件烦人的事，这只不过是又一个实例而已。

第一个定义是名词形式：

> 重构（名词）：对软件内部结构的一种调整，目的是在不改变「软件之可察行为」前提下，提高其可理解性，降低其修改成本。

你可以在后续章节中找到许多重构范例，诸如 *Extract Method*（110）和 *Pull Up Field*（320）等等。一般而言重构都是对软件的小改动，但重构可以包含另一个重构。例如 *Extract Class*（149）通常包含 *Move Method*（142）和 *Move Field*（146）。

「重构」的另一个用法是动词形式：

> 重构（动词）：使用一系列重构准则（手法），在不改变「软件之可察行为」前提下，调整其结构。

所以，在软件开发过程中，你可能会花上数小时的时间进行重构，其间可能用上数十个不同的重构准则。

曾经有人这样问我：『重构就只是整理代码吗？』从某种角度来说，是的！但我认为重构不止于此，因为它提供了一种更高效且受控的代码整理技术。自从运用重构技术后，我发现自己对代码的整理比以前更有效率。这是因为我知道该使用哪些重构准则，我也知道以怎样的方式使用它们才能够将错误减到最少，而且在每一个可能出错的地方我都加以测试。

我的定义还需要往两方面扩展。首先，重构的目的是使软件更容易被理解和修改。你可以在软件内部做很多修改，但必须对软件「可受观察之外部行为」只造成很小变化，或甚至不造成变化。与之形成对比的是「性能优化」。和重构一样，性能优化通常不会改变组件的行为（除了执行速度），只会改变其内部结构。但是两者出发点不同：性能优化往往使代码较难理解，但为了得到所需的性能你不得不那么做。

我要强调的第二点是：重构不会改变软件「可受观察之行为」— 重构之后软件功能一如以往。任何用户，不论最终用户或程序员，都不知道已有东西发生了变化。（译注：「可受观察之行为」其实也包括性能，因为性能是可以被观察的。不过我想我们无需太挑剔这些用词。）

两顶帽子

上述第二点引出了 Kent Beck 的「两顶帽子」比喻。使用重构技术开发软件时，你把自己的时间分配给两种截然不同的行为：「添加新功能」和「重构」。添加新功能时，你不应该修改既有代码，只管添加新功能。通过测试（并让测试正常运行），你可以衡量自己的工作进度。重构时你就不能再添加功能，只管改进程序结构。此时你不应该添加任何测试（除非发现先前遗漏的任何东西），只在绝对必要（用以处理接口变化）时才修改测试。

软件开发过程中，你可能会发现自己经常变换帽子。首先你会尝试添加新功能，然后你会意识到：如果把程序结构改一下，功能的添加会容易得多。于是你换一顶帽子，做一会儿重构工作。程序结构调整好后，你又换上原先的帽子，继续添加新功能。新功能正常工作后，你又发现自己的编码造成程序难以理解，于是你又换上重构帽子……。整个过程或许只花十分钟，但无论何时你都应该清楚自己戴的是哪一顶帽子。

2.2 为何重构？

我不想把重构说成治百病的万灵丹，它绝对不是所谓的「银弹」[1]。不过它的确很有价值，虽不是一颗银子弹却是一把「银钳子」，可以帮助你始终良好地控制自己的代码。重构是个工具，它可以（并且应该）为了以下数个目的而被运用：

「重构」改进软件设计

如果没有重构，程序的设计会逐渐腐败变质。当人们只为短期目的，或是在完全理解整体设计之前，就贸然修改代码，程序将逐渐失去自己的结构，程序员愈来愈难通过阅读源码而理解原本设计。重构很像是在整理代码，你所做的就是让所有东西回到应该的位置上。代码结构的流失是累积性的。愈难看出代码所代表的设计意涵，就愈难保护其中设计，于是该设计就腐败得愈快。经常性的重构可以帮助代码维持自己该有的形态。

同样完成一件事，设计不良的程序往往需要更多代码，这常常是因为代码在不同的地方使用完全相同的语句做同样的事。因此改进设计的一个重要方向就是消除重复代码（Duplicate Code）。这个动作的重要性着眼于未来。代码数量减少并不会使系统运行更快，因为这对程序的运行轨迹几乎没有任何明显影响。然而代码数量减少将使未来可能的程序修改动作容易得多。代码愈多，正确的修改就愈困难，因为有更多代码需要理解。你在这儿做了点修改，系统却不如预期那样工作，因为你未曾修改另一处 — 那儿的代码做着几乎完全一样的事情，只是所处环境略有不同。如果消除重复代码，你就可以确定代码将所有事物和行为都只表述一次，惟一一次，这正是优秀设计的根本。

[1] 译注：「银弹」（silver bullet）是美国家喻户晓的比喻。美国民间流传月圆之夜狼人出没，只有以纯银子弹射穿狼人心脏，才能制服狼人。

「重构」使软件更易被理解

从许多角度来说，所谓程序设计，便是与计算机交谈。你编写代码告诉计算机做什么事，它的响应则是精确按照你的指示行动。你得及时填补「想要它做什么」和「告诉它做什么」之间的缝隙。这种编程模式的核心就是「准确说出吾人所欲」。除了计算机外，你的源码还有其他读者：数个月之后可能会有另一位程序员尝试读懂你的代码并做一些修改。我们很容易忘记这第二位读者，但他才是最重要的。计算机是否多花了数个钟头进行编译，又有什么关系呢？如果一个程序员花费一周时间来修改某段代码，那才关系重大。如果他理解你的代码，这个修改原本只需一小时。

问题在于，当你努力让程序运转的时候，你不会想到未来出现的那个开发者。是的，是应该改变一下我们的开发节奏，对代码做适当修改，让代码变得更易理解。重构可以帮助我们让代码更易读。一开始进行重构时，你的代码可以正常运行，但结构不够理想。在重构上花一点点时间，就可以让代码更好地表达自己的用途。这种编程模式的核心就是「准确说出你的意思」。

关于这一点，我没必要表现得如此无私。很多时候那个「未来的开发者」就是我自己。此时重构就显得尤其重要了。我是个很懒惰的程序员，我的懒惰表现形式之一就是：总是记不住自己写过的代码。事实上对于任何立可查阅的东西我都故意不去记它，因为我怕把自己的脑袋塞爆。我总是尽量把该记住的东西写进程序里头，这样我就不必记住它了。这么一来我就不必太担心 Old Peculier （译注：一种有名的麦芽酒）[Jackson] 杀光我的脑细胞。

这种可理解性还有另一方面的作用。我利用重构来协助我理解不熟悉的代码。当我看到不熟悉的代码，我必须试着理解其用途。我先看两行代码，然后对自己说：『噢，是的，它做了这些那些……』。有了重构这个强大武器在手，我不会满足于这么一点脑中体会。我会真正动手修改代码，让它更好地反映出我的理解，然后重新执行，看它是否仍然正常运作，以此检验我的理解是否正确。

一开始我所做的重构都像这样停留在细枝末节上。随着代码渐趋简洁，我发现自己可以看到一些以前看不到的设计层面的东西。如果不对代码做这些修改，也许我永远看不见它们，因为我的聪明才智不足以在脑子里把这一切都想像出来。Ralph Johnson 把这种「早期重构」描述为「擦掉窗户上的污垢，使你看得更远」。研究代码时我发现，重构把我带到更高的理解层次上。如果没有重构，我达不到这种层次。

「重构」助你找到臭虫（bugs）

对代码的理解，可以帮助我找到臭虫。我承认我不太擅长调试。有些人只要盯着一大段代码就可以找出里面的臭虫，我可不行。但我发现如果我对代码进行重构，我就可以深入理解代码的作为，并恰到好处地把新的理解反馈回去。搞清楚程序结构的同时，我也清楚了自己所做的一些假设，从这个角度来说，不找到臭虫都难矣。

这让我想起了 Kent Beck 经常形容自己的一句话：「我不是个伟大的程序员；我只是个有着一些优秀习惯的好程序员而已。」重构能够帮助我更有效地写出强固稳健（robust）的代码。

「重构」助你提高编程速度

终于，前面的一切都归结到了这最后一点：重构帮助你更快速地开发程序。

听起来有点违反直觉。当我谈到重构，人们很容易看出它能够提高质量。改善设计、提升可读性、减少错误，这些都是提高质量。但这难道不会降低开发速度吗？

我强烈相信：良好设计是快速软件开发的根本。事实上拥有良好设计才可能达成快速的开发。如果没有良好设计，或许某一段时间内你的进展迅速，但恶劣的设计很快就让你的速度慢下来。你会把时间花在调试上面，无法添加新功能。修改时间愈来愈长，因为你必须花愈来愈多的时间去理解系统、寻找重复代码。随着你给最初程序打上一个又一个的补丁（patch），新特性需要更多代码才能实现。真是个恶性循环。

良好设计是维持软件开发速度的根本。重构可以帮助你更快速地开发软件，因为它阻止系统腐败变质，它甚至还可以提高设计质量。

2.3 何时重构？

当我谈论重构，常常有人问我应该怎样安排重构时间表。我们是不是应该每两个月就专门安排两个星期来进行重构呢？

几乎任何情况下我都反对专门拨出时间进行重构。在我看来，重构本来就不是一件「特别拨出时间做」的事情，重构应该随时随地进行。你不应该为重构而重构，你之所以重构，是因为你想做别的什么事，而重构可以帮助你把那些事做好。

三次法则（The Rule of Three）

Don Roberts 给了我一条准则：第一次做某件事时只管去做；第二次做类似的事会产生反感，但无论如何还是做了；第三次再做类似的事，你就应该重构。

> 事不过三，三则重构。（*Three strikes and you refactor.*）

添加功能时一并重构

最常见的重构时机就是我想给软件添加新特性的时候。此时，重构的第一个原因往往是为了帮助我理解需要修改的代码。这些代码可能是别人写的，也可能是我自己写的。无论何时只要我想理解代码所做的事，我就会问自己：是否能对这段代码进行重构，使我能更快理解它。然后我就会重构。之所以这么做，部分原因是为了让我下次再看这段代码时容易理解，但最主要的原因是：如果在前进过程中把代码结构理清，我就可以从中理解更多东西。

在这里，重构的另一个原动力是：代码的设计无法帮助我轻松添加我所需要的特性。我看着设计，然后对自己说：「如果用某种方式来设计，添加特性会简单得多」。这种情况下我不会因为自己过去的错误而懊恼 — 我用重构来弥补它。之所以这么做，部分原因是为了让未来增加新特性时能够更轻松一些，但最主要的原因还是：我发现这是最快捷的途径。重构是一个快速流畅的过程，一旦完成重构，新特性的添加就会更快速、更流畅。

修补错误时一并重构

调试过程中运用重构，多半是为了让代码更具可读性。当我看着代码并努力理解它的时候，我用重构帮助改善自己的理解。我发现以这种程序来处理代码，常常能够帮助我找出臭虫。你可以这么想：如果收到一份错误报告，这就是需要重构的信号，因为显然代码还不够清晰 — 不够清晰到让你一目了然发现臭虫。

复审代码时一并重构

很多公司都会做常态性的代码复审工作（code reviews），因为这种活动可以改善开发状况。这种活动有助于在开发团队中传播知识，也有助于让较有经验的开发者把知识传递给比较欠缺经验的人，并帮助更多人理解大型软件系统中的更多部分。代码复审工作对于编写清晰代码也很重要。我的代码也许对我自己来说很清晰，对他人则不然。这是无法避免的，因为要让开发者设身处地为那些不熟悉自己所做所为的人设想，实在太困难了。代码复审也让更多人有机会提出有用的建议，毕竟我在一个星期之内能够想出的好点子很有限。如果能得到别人的帮助，我的生活会舒服得多，所以我总是期待更多复审。

我发现，重构可以帮助我复审别人的代码。开始重构前我可以先阅读代码，得到一定程度的理解，并提出一些建议。一旦想到一些点子，我就会考虑是否可以通过重构立即轻松地实现它们。如果可以，我就会动手。这样做了几次以后，我可以把代码看得更清楚，提出更多恰当的建议。我不必想像代码「应该是什么样」，我可以「看见」它是什么样。于是我可以获得更高层次的认识。如果不进行重构，我永远无法得到这样的认识。

重构还可以帮助代码复审工作得到更具体的结果。不仅获得建议，而且其中许多建议能够立刻实现。最终你将从实践中得到比以往多得多的成就感。

为了让过程正常运转，你的复审团队必须保持精练。就我的经验，最好是一个复审者搭配一个原作者，共同处理这些代码。复审者提出修改建议，然后两人共同判断这些修改是否能够通过重构轻松实现。果真能够如此，就一起着手修改。

如果是比较大的设计复审工作，那么，在一个较大团队内保留多种观点通常会更好一些。此时直接展示代码往往不是最佳办法。我喜欢运用 UML 示意图展现设计，并以 CRC 卡展示软件情节。换句话说，我会和某个团队进行设计复审，而和个别（单一）复审者进行代码复审。

极限编程（Extreme Programming）[Beck, XP] 中的「搭档（成对）编程」（Pair Programming）形式，把代码复审的积极性发挥到了极致。一旦采用这种形式，所有正式开发任务都由两名开发者在同一台机器上进行。这样便在开发过程中形成随时进行的代码复审工作，而重构也就被包含在开发过程内了。

为什么重构有用（Why Refactoring Works）

— *Kent Beck*

程序有两面价值：「今天可以为你做什么」和「明天可以为你做什么」。大多数时候，我们都只关注自己今天想要程序做什么。不论是修复错误或是添加特性，我们都是为了让程序能力更强，让它在今天更有价值。

但是系统今天（当下）的行为，只是整个故事的一部分，如果没有认清这一点，你无法长期从事编程工作。如果你「为求完成今天任务」而采取的手法使你不可能在明天完成明天的任务，那么你还是失败。但是，你知道自己今天需要什么，却不一定知道自己明天需要什么。也许你可以猜到明天的需求，也许吧，但肯定还有些事情出乎你的意料。

对于今天的工作，我了解得很充分；对于明天的工作，我了解得不够充分。但如果我纯粹只是为今天工作，明天我将完全无法工作。

重构是一条摆脱束缚的道路。如果你发现昨天的决定已经不适合今天的情况，放心改变这个决定就是，然后你就可以完成今天的工作了。明天，喔，明天回头看今天的理解也许觉得很幼稚，那时你还可以改变你的理解。

是什么让程序如此难以相与？下笔此刻，我想起四个原因，它们是：

- 难以阅读的程序，难以修改。
- 逻辑重复（duplicated logic）的程序，难以修改。
- 添加新行为时需修改既有代码者，难以修改。
- 带复杂条件逻辑（complex conditional logic）的程序，难以修改。

因此，我们希望程序：(1) 容易阅读；(2) 所有逻辑都只在惟一地点指定；(3) 新的改动不会危及现有行为；(4) 尽可能简单表达条件逻辑（conditional logic）。

重构是这样一个过程：它在一个目前可运行的程序上进行，企图在「不改变程序行为」的情况下赋予上述美好性质，使我们能够继续保持高速开发，从而增加程序的价值。

2.4 怎么对经理说？

「该怎么跟经理说重构的事？」这是我最常被问到的问题之一。如果这位经理懂技术，那么向他介绍重构应该不会很困难。如果这位经理只对质量感兴趣，那么问题就集中到了「质量」上面。此时，在复审过程中使用重构，就是一个不错的办法。大量研究结果显示，「技术复审」是减少错误、提高开发速度的一条重要途径。随

便找一本关于复审、审查或软件开发程序的书看看，从中找些最新引证，应该可以让大多数经理认识复审的价值。然后你就可以把重构当作「将复审意见引入代码内」的方法来使用，这很容易。

当然，很多经理嘴巴上说自己「质量驱动」，其实更多是「进度驱动」。这种情况下我会给他们一个较有争议的建议：不要告诉经理！

这是在搞破坏吗？我不这样想。软件开发者都是专业人士。我们的工作就是尽可能快速创造出高效软件。我的经验告诉我，对于快速创造软件，重构可带来巨大帮助。如果需要添加新功能，而原本设计却又使我无法方便地修改，我发现先「进行重构」再「添加新功能」会更快些。如果要修补错误，我需得先理解软件工作方式，而我发现重构是理解软件的最快方式。受进度驱动的经理要我尽可能快速完事，至于怎么完成，那就是我的事了。我认为最快的方式就是重构，所以我就重构。

间接层和重构（Indirection and Refactoring）

— *Kent Beck*

『计算机科学是这样一门科学：它相信所有问题都可以通过多一个间接层（indirection）来解决。』— *Dennis DeBruler*

由于软件工程师对间接层如此醉心，你应该不会惊讶大多数重构都为程序引入了更多间接层。重构往往把大型对象拆成数个小型对象，把大型函数拆成数个小型函数。

但是，间接层是一柄双刃剑。每次把一个东西分成两份，你就需要多管理一个东西。如果某个对象委托（*delegate*）另一对象，后者又委托另一对象，程序会愈加难以阅读。基于这个观点，你会希望尽量减少间接层。

别急，伙计！间接层有它的价值。下面就是间接层的某些价值：

- **允许逻辑共享**（To enable sharing of logic）。比如说一个子函数（submethod）在两个不同的地点被调用，或 superclass 中的某个函数被所有 subclasses 共享。
- **分开解释「意图」和「实现」**（To explain intention and implementation separately）。你可以选择每个 class 和函数的名字，这给了你一个解释自己意图的机会。class 或函数内部则解释实现这个意图的作法。如果 class 和函数内部又以「更小单元的意图」来编写，你所写的代码就可以「与其结构中的大部分重要信息沟通」。
- **将变化加以隔离**（To isolate change）。很可能我在两个不同地点使用同一对象，其中一个地点我想改变对象行为，但如果修改了它，我就要冒「同时影响两处」的风险。为此我做出一个 subclass，并在需要修改处引用这个 subclass。现在，我可以修

改这个 subclass 而不必承担「无意中影响另一处」的风险。

- **将条件逻辑加以编码**（To encode conditional logic）。对象有一种匪夷所思的机制：多态消息（polymorphic messages），可以灵活弹性而清晰地表达条件逻辑。只要显式条件逻辑被转化为消息（message[2]）形式，往往便能降低代码的重复、增加清晰度并提高弹性。

这就是重构游戏：在保持系统现有行为的前提下，如何才能提高系统的质量或降低其成本，从而使它更有价值？

这个游戏中最常见的变量就是：你如何看待你自己的程序。找出一个缺乏「间接层利益」之处，在不修改现有行为的前提下，为它加入一个间接层。现在你获得了一个更有价值的程序，因为它有较高的质量，让我们在明天（未来）受益。

请将这种方法与「小心翼翼的事前设计」做个比较。推测性设计总是试图在任何一行代码诞生之前就先让系统拥有所有优秀质量，然后程序员将代码塞进这个强健的骨架中就行了。这个过程的问题在于：太容易猜错。如果运用重构，你就永远不会面临全盘错误的危险。程序自始至终都能保持一致的行为，而你又有机会为程序添加更多价值不菲的质量。

还有一种比较少见的重构游戏：找出不值得的间接层，并将它拿掉。这种间接层常以中介函数（intermediate methods）形式出现，也许曾经有过贡献，但芳华已逝。它也可能是个组件，你本来期望在不同地点共享它，或让它表现出多态性（polymorphism），最终却只在一处使用之。如果你找到这种「寄生式间接层」，请把它扔掉。如此一来你会获得一个更有价值的程序，不是因为它取得了更多（先前所列）的四种优秀质量，而是因为它以更少的间接层获得一样多的优秀质量。

2.5 重构的难题

学习一种可以大幅提高生产力的新技术时，你总是难以察觉其不适用的场合。通常你在一个特定场景中学习它，这个场景往往是个项目。这种情况下你很难看出什么会造成这种新技术成效不彰或甚至形成危害。十年前，对象技术（object tech.）的情况也是如此。那时如果有人问我「何时不要使用对象」，我很难回答。并非我认为对象十全十美、没有局限性 — 我最反对这种盲目态度，而是尽管我知道它的好处，但确实不知道其局限性在哪儿。

[2] 译注：此处的「消息」（message）是指面向对象古典论述中的意义。在那种场合中，「调用某个函数（method）」就是「送出消息（message）给某个对象（object）」。

现在，重构的处境也是如此。我们知道重构的好处，我们知道重构可以给我们的工作带来垂手可得的改变。但是我们还没有获得足够的经验，我们还看不到它的局限性。

这一小节比我希望的要短。暂且如此吧。随着更多人学会重构技巧，我们也将对它有更多了解。对你而言这意味：虽然我坚决认为你应该尝试一下重构，获得它所提供的利益，但在此同时，你也应该时时监控其过程，注意寻找重构可能引入的问题。请让我们知道你所遭遇的问题。随着对重构的了解日益增多，我们将找出更多解决办法，并清楚知道哪些问题是真正难以解决的。

数据库（Databases）

「重构」经常出问题的一个领域就是数据库。绝大多数商用程序都与它们背后的database schema（数据库表格结构）紧密耦合（coupled）在一起，这也是database schema 如此难以修改的原因之一。另一个原因是数据迁移（migration）。就算你非常小心地将系统分层（layered），将 database schema 和对象模型（object model）间的依赖降至最低，但 database schema 的改变还是让你不得不迁移所有数据，这可能是件漫长而烦琐的工作。

在「非对象数据库」（nonobject databases）中，解决这个问题的办法之一就是：在对象模型（object model）和数据库模型（database model）之间插入一个分隔层（separate layer），这就可以隔离两个模型各自的变化。升级某一模型时无需同时升级另一模型，只需升级上述的分隔层即可。这样的分隔层会增加系统复杂度，但可以给你很大的灵活度。如果你同时拥有多个数据库，或如果数据库模型较为复杂使你难以控制，那么即使不进行重构，这分隔层也是很重要的。

你无需一开始就插入分隔层，可以在发现对象模型变得不稳定时再产生它。这样你就可以为你的改变找到最好的杠杆效应。

对开发者而言，对象数据库既有帮助也有妨碍。某些面向对象数据库提供不同版本的对象之间的自动迁移功能，这减少了数据迁移时的工作量，但还是会损失一定时间。如果各数据库之间的数据迁移并非自动进行，你就必须自行完成迁移工作，这个工作量可是很大的。这种情况下你必须更加留神 classes 内的数据结构变化。你仍然可以放心将 classes 的行为转移过去，但转移值域（field）时就必须格外小心。数据尚未被转移前你就得先运用访问函数（accessors）造成「数据已经转移」的假象。一旦你确定知道「数据应该在何处」时，就可以一次性地将数据迁移过去。这时惟

一需要修改的只有访问函数（accessors），这也降低了错误风险。

修改接口（Changing Interfaces）

关于对象，另一件重要事情是：它们允许你分开修改软件模块的实现（implementation）和接口（interface）。你可以安全地修改某对象内部而不影响他人，但对于接口要特别谨慎 — 如果接口被修改了，任何事情都有可能发生。

一直对重构带来困扰的一件事就是：许多重构手法的确会修改接口。像 *Rename Method*（273）这么简单的重构手法所做的一切就是修改接口。这对极为珍贵的封装概念会带来什么影响呢？

如果某个函数的所有调用动作都在你的控制之下，那么即使修改函数名称也不会有任何问题。哪怕面对一个 public 函数，只要能取得并修改其所有调用者，你也可以安心地将这个函数易名。只有当需要修改的接口系被那些「找不到，即使找到也不能修改」的代码使用时，接口的修改才会成为问题。如果情况真是如此，我就会说：这个接口是个「已发布接口」（published interface）— 比公开接口（public interface）更进一步。接口一旦发布，你就再也无法仅仅修改调用者而能够安全地修改接口了。你需要一个略为复杂的程序。

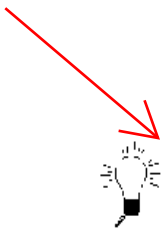
这个想法改变了我们的问题。如今的问题是：该如何面对那些必须修改「已发布接口」的重构手法？

简言之，如果重构手法改变了已发布接口（published interface），你必须同时维护新旧两个接口，直到你的所有用户都有时间对这个变化做出反应。幸运的是这不太困难。你通常都有办法把事情组织好，让旧接口继续工作。请尽量这么做：让旧接口调用新接口。当你要修改某个函数名称时，请留下旧函数，让它调用新函数。千万不要拷贝函数实现码，那会让你陷入「重复代码」（duplicated code）的泥淖中难以自拔。你还应该使用 Java 提供的 deprecation（反对）设施，将旧接口标记为 "deprecated"。这么一来你的调用者就会注意到它了。

这个过程的一个好例子就是 Java 容器类（群集类，collection classes）。Java 2 的新容器取代了原先一些容器。当 Java 2 容器发布时，JavaSoft 花了很大力气来为开发者提供一条顺利迁徙之路。

「保留旧接口」的办法通常可行，但很烦人。起码在一段时间里你必须建造（build）并维护一些额外的函数。它们会使接口变得复杂，使接口难以使用。还好我们有另一个选择：不要发布（publish）接口。当然我不是说要完全禁止，因为很明显你必

得发布一些接口。如果你正在建造供外部使用的 APIs，像 Sun 所做的那样，肯定你必得发布接口。我之所以说尽量不要发布，是因为我常常看到一些开发团队公开了太多接口。我曾经看到一支三人团队这么工作：每个人都向另外两人公开发布接口。这使他们不得不经常来回维护接口，而其实他们原本可以直接进入程序库，径行修改自己管理的那一部分，那会轻松许多。过度强调「代码拥有权」的团队常常会犯这种错误。发布接口很有用，但也有代价。所以除非真有必要，别发布接口。这可能意味需要改变你的代码拥有权观念，让每个人都可以修改别人的代码，以运应接口的改动。以搭档（成对）编程（Pair Programming）完成这一切通常是个好主意。



不要过早发布（publish）接口。请修改你的代码拥有权政策，使重构更顺畅。

Java 之中还有一个特别关于「修改接口」的问题：在 throws 子句中增加一个异常。这并不是对签名式（signature）的修改，所以你无法以 *delegation*（委托手法）隐藏它。但如果用户代码不做出相应修改，编译器不会让它通过。这个问题很难解决。你可以为这个函数选择一个新名字，让旧函数调用它，并将这个新增的 *checked exception*（可控式异常）转换成一个 *unchecked exception*（不可控异常）。你也可以抛出一个 *unchecked* 异常，不过这样你就会失去检验能力。如果你那么做，你可以警告调用者：这个 *unchecked* 异常日后会变成一个 *checked* 异常。这样他们就有时间在自己的代码中加上对此异常的处理。出于这个原因，我总是喜欢为整个 package 定义一个 superclass 异常（就像 `java.sql` 的 `SQLException`），并确保所有 public 函数只在自己的 throws 子句中声明这个异常。这样我就可以随心所欲地定义 subclass 异常，不会影响调用者，因为调用者永远只知道那个更具一般性的 superclass 异常。

难以通过重构手法完成的设计改动

通过重构，可以排除所有设计错误吗？是否存在某些核心设计决策，无法以重构手法修改？在这个领域里，我们的统计数据尚不完整。当然某些情况下我们可以很有效地重构，这常常令我们倍感惊讶，但的确也有难以重构的地方。比如说在一个项目中，我们很难（但还是有可能）将「无安全需求（no security requirements）情况下构造起来的系统」重构为「安全性良好的（good security）系统」。

这种情况下我的办法就是「先想像重构的情况」。考虑候选设计方案时，我会问自己：将某个设计重构为另一个设计的难度有多大？如果看上去很简单，我就不必太

担心选择是否得当，于是我就会选最简单的设计，哪怕它不能覆盖所有潜在需求也没关系。但如果预先看不到简单的重构办法，我就会在设计上投入更多力气。不过我发现，这种情况很少出现。

何时不该重构？

有时候你根本不应该重构 — 例如当你应该重新编写所有代码的时候。有时候既有代码实在太混乱，重构它还不如重新写一个来得简单。作出这种决定很困难，我承认我也没有什么好准则可以判断何时应该放弃重构。

重写（而非重构）的一个清楚讯号就是：现有代码根本不能正常运作。你可能只是试着做点测试，然后就发现代码中满是错误，根本无法稳定运作。记住，重构之前，代码必须起码能够在大部分情况下正常运作。

一个折衷办法就是：将「大块头软件」重构为「封装良好的小型组件」。然后你就可以逐一对组件做出「重构或重建」的决定。这是一个颇具希望的办法，但我还没有足够数据，所以也无法写出优秀的指导原则。对于一个重要的古老系统，这肯定会是一个很好的方向。

另外，如果项目已近最后期限，你也应该避免重构。在此时机，从重构过程赢得的生产力只有在最后期限过后才能体现出来，而那个时候已经时不我予。Ward Cunningham 对此有一个很好的看法，他把未完成的重构工作形容为「债务」。很多公司都需要借债来使自己更有效地运转。但是借债就得付利息，过于复杂的代码所造成的「维护和扩展的额外开销」就是利息。你可以承受一定程度的利息，但如果利息太高你就会被压垮。把债务管理好是很重要的，你应该随时通过重构来偿还部分债务。

如果项目已经非常接近最后期限，你不应该再分心于重构，因为已经没有时间了。不过多个项目经验显示：重构的确能够提高生产力。如果最后你没有足够时间，通常就表示你其实早该进行重构。

2.6 重构与设计

「重构」肩负一项特别任务：它和设计彼此互补。初学编程的时候，我埋头就写程序，浑浑噩噩地进行开发。然而很快我便发现，「事先设计」（upfront design）可以助我节省回头工的高昂成本。于是我很快加强这种「预先设计」风格。许多人都

把设计看做软件开发的关键环节，而把编程（programming）看做只是机械式的低级劳动。他们认为设计就像画工程图而编码就像施工。但是你要知道，软件和真实器械有着很大的差异。软件的可塑性更强，而且完全是思想产品。正如 Alistair Cockburn 所说：「有了设计，我可以思考更快，但是其中充满小漏洞。」

有一种观点认为：重构可以成为「预先设计」的替代品。这意思是你根本不必做任何设计，只管按照最初想法开始编码，让代码有效运作，然后再将它重构成型。事实上这种办法真的可行。我的确看过有人这么做，最后获得设计良好的软件。极限编程（Extreme Programming）[Beck, XP] 的支持者极力提倡这种办法。

尽管如上所言，只运用重构也能收到效果，但这并不是最有效的途径。是的，即使极限编程（Extreme Programming）爱好者也会进行预先设计。他们会使用 CRC 卡或类似的东西来检验各种不同想法，然后才得到第一个可被接受的解决方案，然后才能开始编码，然后才能重构。关键在于：重构改变了「预先设计」的角色。如果没有重构，你就必须保证「预先设计」正确无误，这个压力太大了。这意味如果将来需要对原始设计做任何修改，代价都将非常高昂。因此你需要把更多时间和精力放在预先设计上，以避免日后修改。

如果你选择重构，问题的重点就转变了。你仍然做预先设计，但是不必一定找出正确的解决方案。此刻的你只需要得到一个足够合理的解决方案就够了。你很肯定地知道，在实现这个初始解决方案的时候，你对问题的理解也会逐渐加深，你可能会察觉最佳解决方案和你当初设想的有些不同。只要有重构这项武器在手，就不成问题，因为重构让日后的修改成本不再高昂。

这种转变导致一个重要结果：软件设计朝向简化前进了一大步。过去未曾运用重构时，我总是力求得到灵活的解决方案。任何一个需求都让我提心吊胆地猜疑：在系统寿命期间，这个需求会导致怎样的变化？由于变更设计的代价非常高昂，所以我希望建造一个足够灵活、足够强固的解决方案，希望它能承受我所能预见的所有需求变化。问题在于：要建造一个灵活的解决方案，所需的成本难以估算。灵活的解决方案比简单的解决方案复杂许多，所以最终得到的软件通常也会更难维护 — 虽然它在我预先设想的方向上的确是更加灵活。就算幸运走在预先设想的方向上，你也必须理解如何修改设计。如果变化只出现在一两个地方，那不算大问题。然而变化其实可能出现在系统各处。如果在所有可能的变化出现地点都建立起灵活性，整个系统的复杂度和维护难度都会大大提高。当然，如果最后发现所有这些灵活性都毫无必要，这才是最大的失败。你知道，这其中肯定有些灵活性的确派不上用场，

但你却无法预测到底是哪些派不上用场。为了获得自己想要的灵活性，你不得不加入比实际需要更多的灵活性。

有了重构，你就可以通过一条不同的途径来应付变化带来的风险。你仍旧需要思考潜在的变化，仍旧需要考虑灵活的解决方案。但是你不必再逐一实现这些解决方案，而是应该问问自己：「把一个简单的解决方案重构成这个灵活的方案有多大难度？」如果答案是「相当容易」（大多数时候都如此），那么你就只需实现目前的简单方案就行了。

重构可以带来更简单的设计，同时又不损失灵活性，这也降低了设计过程的难度，减轻了设计压力。一旦对重构带来的简单性有更多感受，你甚至可以不必再预先思考前述所谓的灵活方案——一旦需要它，你总有足够的信心去重构。是的，当下只管建造可运行的最简化系统，至于灵活而复杂的设计，唔，多数时候你都不会需要它。

劳而无获

— Ron Jeffries

Chrysler Comprehensive Compensation（克莱斯勒综合薪资系统）的支付过程太慢了。虽然我们的开发还没结束，这个问题却已经开始困扰我们，因为它已经拖累了测试速度。

Kent Beck、Martin Fowler 和我决定解决这个问题。等待大伙儿会合的时间里，凭着我对这个系统的全盘了解，我开始推测：到底是什么让系统变慢了？我想到数种可能，然后和伙伴们谈了几种可能的修改方案。最后，关于「如何让这个系统运行更快」，我们提出了一些真正的好点子。

然后，我们拿 Kent 的量测工具度量了系统性能。我一开始所想的可能性竟然全都不是问题肇因。我们发现：系统把一半时间用来创建「日期」实体（instance）。更有趣的是，所有这些实体都有相同的值。

于是我们观察日期的创建逻辑，发现有机会将它优化。日期原本是由字符串转换而生，即使无外部输入也是如此。之所以使用字符串转换方式，完全是为了方便键盘输入。好，也许我们可以将它优化。

于是我们观察日期怎样被这个程序运用。我们发现，很多日期对象都被用来产生「日期区间」实体（instance）。「日期区间」是个对象，由一个起始日期和一个结束日期组成。仔细追踪下去，我们发现绝大多数日期区间是空的！

处理日期区间时我们遵循这样一个规则：如果结束日期在起始日期之前，这个日期区间就该是空的。这是一条很好的规则，完全符合这个 class 的需要。采用此一规则后不久，

我们意识到，创建一个「起始日期在结束日期之后」的日期区间，仍然不算是清晰的代码，于是我们把这个行为提炼到一个 ***factory method***（译注：一个著名的设计模式，见《*Design Patterns*》），由它专门创建「空的日期区间」。

我们做了上述修改，使代码更加清晰，却意外得到了一个惊喜。我们创建一个固定不变的「空日期区间」对象，并让上述调整后的 ***factory method*** 每次都返回该对象，而不再每次都创建新对象。这一修改把系统速度提升了几乎一倍，足以让测试速度达到可接受程度。这只花了我们大约五分钟。

我和团队成员（Kent 和 Martin 谢绝参加）认真推测过：我们了若指掌的这个程序中可能有什么错误？我们甚至凭空做了些改进设计，却没有先对系统的真实情况进行量测。

我们完全错了。除了一场很有趣的交谈，我们什么好事都没做。

教训：哪怕你完全了解系统，也请实际量测它的性能，不要臆测。臆测会让你学到一些东西，但十有八九你是错的。

2.7 重构与性能（Performance）

译注：在我的接触经验中，performance 一词被不同的人予以不同的解释和认知：效率、性能、效能。不同地区（例如台湾和大陆）的习惯用法亦不相同。本书一遇 performance 我便译为性能。efficient 译为高效，effective 译为有效。

关于重构，有一个常被提出的问题：它对程序的性能将造成怎样的影响？为了让软件易于理解，你常会做出一些使程序运行变慢的修改。这是个重要的问题。我并不赞成为了提高设计的纯洁性或把希望寄托于更快的硬件身上，而忽略了程序性能。已经有很多软件因为速度太慢而被用户拒绝，日益提高的机器速度亦只不过略微放宽了速度方面的限制而已。但是，换个角度说，虽然重构必然会使软件运行更慢，但它也使软件的性能优化更易进行。除了对性能有严格要求的实时（real time）系统，其他任何情况下「编写快速软件」的秘密就是：首先写出可调（tunable）软件，然后调整它以求获得足够速度。

我看过三种「编写快速软件」的方法。其中最严格的是「时间预算法」（time budgeting），这通常只用于性能要求极高的实时系统。如果使用这种方法，分解你的设计时就要做好预算，给每个组件预先分配一定资源 — 包括时间和执行轨迹（footprint）。每个组件绝对不能超出自己的预算，就算拥有「可在不同组件之间调度预配时间」的机制也不行。这种方法高度重视性能，对于心律调节器一类的系统是必须的，因为在这样的系统中迟来的数据就是错误的数据。但对其他类系统（例如我经常开发的企业信息系统）而言，如此追求高性能就有点过份了。

第二种方法是「持续关切法」（constant attention）。这种方法要求任何程序员在任何时间做任何事时，都要设法保持系统的高性能。这种方式很常见，感觉上很有吸引力，但通常不会起太大作用。任何修改如果是为了提高性能，通常会使程序难以维护，因而减缓开发速度。如果最终得到的软件的确更快了，那么这点损失尚有所值，可惜通常事与愿违，因为性能改善一旦被分散到程序各角落，每次改善都只不过是从「对程序行为的一个狭隘视角」出发而已。

关于性能，一件很有趣的事情是：如果你对大多数程序进行分析，你会发现它把大半时间都耗费在一小半代码身上。如果你一视同仁地优化所有代码，90% 的优化工作都是白费劲儿，因为被你优化的代码有许多难得被执行起来。你花时间做优化是为了让程序运行更快，但如果因为缺乏对程序的清楚认识而花费时间，那些时间都是被浪费掉了。

第二种性能提升法系利用上述的 "90%" 统计数据。采用这种方法时，你以一种「良好的分解方式」（well-factored manner）来建造自己的程序，不对性能投以任何关切，直至进入性能优化阶段 — 那通常是在开发后期。一旦进入该阶段，你再按照某个特定程序来调整程序性能。

在性能优化阶段中，你首先应该以一个量测工具监控程序的运行，让它告诉你程序中哪些地方大量消耗时间和空间。这样你就可以找出性能热点（hot spot）所在的一小段代码。然后你应该集中关切这些性能热点，并使用前述「持续关切法」中的优化手段来优化它们。由于你把注意力都集中在热点上，较少的工作量便可显现较好的成果。即便如此你还是必须保持谨慎。和重构一样，你应该小幅度进行修改。每走一步都需要编译、测试、再次量测。如果没能提高性能，就应该撤销此次修改。你应该继续这个「发现热点、去除热点」的过程，直到获得客户满意的性能为止。关于这项技术，McConnell [McConnell] 为我们提供了更多信息。

一个被良好分解（well-factored）的程序可从两方面帮助此种优化形式。首先，它让你有比较充裕的时间进行性能调整（performance tuning），因为有分解良好的代码在手，你就能够更快速地添加功能，也就有更多时间用在性能问题上（准确的量测则保证你把这些时间投资在恰当地点）。其次，面对分解良好的程序，你在进行性能分析时便有较细的粒度（granularity），于是量测工具把你带入范围较小的程序段落中，而性能的调整也比较容易些。由于代码更加清晰，因此你能够更好地理解自己的选择，更清楚哪种调整起关键作用。

我发现重构可以帮助我写出更快的软件。短程看来，重构的确会使软件变慢，但它使优化阶段中的软件性能调整更容易。最终我还是有赚头。

2.8 重构起源何处？

我曾经努力想找出重构（refactoring）一词的真正起源，但最终失败了。优秀程序员肯定至少会花一些时间来清理自己的代码。这么做是因为，他们知道简洁的代码比杂乱无章的代码更容易修改，而且他们知道自己几乎无法一开始就写出简洁的代码。

重构不止如此。本书中我把重构看做整个软件开发过程的一个关键环节。最早认识重构重要性的两个人是 Ward Cunningham 和 Kent Beck，他们早在 1980s 之前就开始使用 Smalltalk，那是个特别适合重构的环境。Smalltalk 是一个十分动态的环境，你可以很快写出极具功能的软件。Smalltalk 的「编译/连结/执行」周期非常短，因此很容易快速修改代码。它是面向对象，所以也能够提供强大工具，最大限度地将修改的影响隐藏于定义良好的接口背后。Ward 和 Kent 努力发展出一套适合这类环境的软件开发过程（如今 Kent 把这种风格叫作极限编程 [Beck, XP]）。他们意识到：重构对于提高他们的生产力非常重要。从那时起他们就一直在工作中运用重构技术，在严肃而认真的软件项目中使用它，并不断精炼这个程序。

Ward 和 Kent 的思想对 Smalltalk 社群产生了极大影响，重构概念也成为 Smalltalk 文化中的一个重要元素。Smalltalk 社群的另一位领袖是 Ralph Johnson，伊利诺斯大学乌尔班纳分校教授，著名的「四巨头」[3][Gang of Four] 之一。Ralph 最大的兴趣之一就是开发软件框架（framework）。他揭示了重构对于灵活高效框架的开发帮助。

Bill Opdyke 是 Ralph 的博士研究生，对框架也很感兴趣。他看到重构的潜在价值，并看到重构应用于 Smalltalk 之外的其他语言的可能性。他的技术背景是电话交换系统的开发。在这种系统中，大量的复杂情况与时俱增，而且非常难以修改。Bill 的博士研究就是从工具构筑者的角度来看待重构。通过研究，Bill 发现：在 C++ framework 开发项目中，重构很有用。他也研究了极有必要的「语义保持性（semantics-preserving）重构」及其证明方式，以及如何以工具实现重构。时至今日，Bill 的博士论文 [Opdyke] 仍然是重构领域中最有价值、最丰硕的研究成果。此外他为本书撰写了第 13 章。

[3] 译注：Ralph Johnson 和另外三位先生 Erich Gamma, Richard Helm, John Vlissides 合写了软件开发界驰名的《*Design Patterns*》，人称四巨头（Gang of Four）。

我还记得 1992 年 OOPSLA 大会上见到 Bill 的情景。我们坐在一间咖啡厅里，讨论当时我正为保健业务构筑的一个概念框架（conceptual framework）中的某些工作。Bill 跟我谈起他的研究成果，我还记得自己当时的想法：「有趣，但并非真的那么重要。」唉，我完全错了。

John Brant 和 Don Roberts 将重构中的「工具」构想发扬光大，开发了一个名为「重构浏览器」（Refactoring Browser）的 Smalltalk 重构工具。他们撰写了本书第 14 章，其中对重构工具做了更多介绍。

那么，我呢？我一直有清理代码的倾向，但从来没有想到这会有那么重要。后来我和 Kent 一起做了个项目，看到他使用重构手法，也看到重构对生产性能和产品质量带来的影响。这份体验让我相信：重构是一门非常重要的技术。但是，在重构的学习和推广过程中我遇到了挫折，因为我拿不出任何一本书给程序员看，也没有任何一位专家打算写出这样一本书。所以，在这些专家的帮助下，我写下了这本书。

优化一个薪资系统

— Rich Garzaniti

将 Chrysler Comprehensive Compensation（克莱斯勒综合薪资系统）交给 GemStone 公司之前，我们用了相当长的时间开发它。开发过程中我们无可避免地发现程序不够快，于是找了 Jim Haungs — GemSmith 中的一位好手 — 请他帮我们优化这个系统。

Jim 先用一点时间让他的团队了解系统运作方式，然后以 GemStone 的 ProfMonitor 特性编写出一个性能量测工具，将它插入我们的功能测试中。这个工具可以显示系统产生的对象数量，以及这些对象的诞生点。

令我们吃惊的是：创建量最大的对象竟是字符串。其中最大的工作量则是反复产生 12,000-bytes 的字符串。这很特别，因为这字符串实在太大了，连 GemStone 惯用的垃圾回收设施都无法处理它。由于它是如此巨大，每当被创建出来，GemStone 都会将它分页（paging）至磁盘上。也就是说字符串的创建竟然用上了 I/O 子系统（译注：分页机制会动用 I/O），而每次输出记录时都要产生这样的字符串三次！

我们的第一个解决办法是把一个 12,000-bytes 字符串缓存（cached）起来，这可解决一大半问题。后来我们又加以修改，将它直接写入一个 file stream，从而避免产生字符串。

解决了「巨大字符串」问题后，Jim 的量测工具又发现了一些类似问题，只不过字符串稍微小一些：800-bytes、500-bytes……等等，我们也都对它们改用 file stream，于是问题都解决了。

使用这些技术，我们稳步提高了系统性能。开发过程中原本似乎需要 1,000 小时以上才

能完成的薪资计算，实际运作时只花 40 小时。一个月后我们把时间缩短到 18 小时。正式投入运转时只花 12 小时。经过一年的运行和改善后，全部计算只需 9 小时。

我们的最大改进就是：将程序放在多处理器（multi-processor）计算机上，以多线程（multiple threads）方式运行。最初这个系统并非按照多线程思维来设计，但由于代码有良好分解（well factored），所以我们只花三天时间就让它得以同时运行多个线程了。现在，薪资的计算只需 2 小时。

在 Jim 提供工具使我们得以在实际操作中量度系统性能之前，我们也猜测过问题所在。但如果只靠猜测，我们需要很长的时间才能试出真正的解法。真实的量测指出了一个完全不同的方向，并大大加快了我们的进度。

3

代码的坏味道

Bad smells in Code, by Kent Beck and Martin Fowler

If it stinks, change it.（如果尿布臭了，就换掉它。）

— 语出 *Beck* 奶奶，讨论小孩抚养哲学

现在，对于「重构如何运作」，你已经有了相当好的理解。但是知道 How 不代表知道 When。决定何时重构、何时停止和知道重构机制如何运转是一样重要的。

难题来了！解释「如何删除一个 *instance* 变量」或「如何产生一个 class hierarchy（阶层体系）」很容易，因为这些都是很简单的事情。但要解释「该在什么时候做这些动作」就没那么顺理成章了。除了露几手含混的编程美学（说实话，这就是咱这些顾问常做的事），我还希望让某些东西更具说服力一些。

去苏黎士拜访 Kent Beck 的时候，我正在为这个微妙的问题大伤脑筋。也许是因为受到刚出生的女儿的气味影响吧，他提出「用味道来形容重构时机」。『味道』，他说，『听起来是不是比含混的美学理论要好多了？』啊，是的。我们看过很多很多代码，它们所属的项目从大获成功到奄奄一息都有。观察这些代码时，我们学会了从中找寻某些特定结构，这些结构指出（有时甚至就像尖叫呼喊）重构的可能性。（本章主词换成「我们」，是为了反映一个事实：Kent 和我共同撰写本章。你应该可以看出我俩的文笔差异 — 插科打诨的部分是我写的，其余都是他的。）

我们并不试图给你一个「重构为时晚矣」的精确衡量标准。从我们的经验看来，没有任何量度规矩比得上一个见识广博者的直觉。我们只会告诉你一些迹象，它会指出「这里有一个可使用重构解决的问题」。你必须培养出自己的判断力，学会判断一个 class 内有多少 *instance* 变量算是太大、一个函数内有多少行代码才算太长。

如果你无法确定该进行哪一种重构手法，请阅读本章内容和封底内页表格来寻找灵感。你可以阅读本章（或快速浏览封底内页表格）来判断自己闻到的是什么味道，然后再看看我们所建议的重构手法能否帮助你。也许这里所列的「臭味条款」和你所检测的不尽相符，但愿它们能够为你指引正确方向。

3.1 Duplicated Code（重复的代码）

臭味行列中首当其冲的就是 Duplicated Code。如果你在一个以上的地点看到相同的程序结构，那么当可肯定：设法将它们合而为一，程序会变得更好。

最单纯的 Duplicated Code 就是「同一个 class 内的两个函数含有相同表达式（expression）」。这时候你需要做的就是采用 *Extract Method*（110）提炼出重复的代码，然后让这两个地点都调用被提炼出来的那一段代码。

另一种常见情况就是「两个互为兄弟（sibling）的 subclasses 内含相同表达式」。要避免这种情况，只需对两个 classes 都使用 *Extract Method*（110），然后再对被提炼出来的代码使用 *Pull Up Method*（332），将它推入 superclass 内。如果代码之间只是类似，并非完全相同，那么就得运用 *Extract Method*（110）将相似部分和差异部分割开，构成单独一个函数。然后你可能发现或许可以运用 *Form Template Method*（345）获得一个 ***Template Method*** 设计模式。如果有些函数以不同的算法做相同的事，你可以择定其中较清晰的一个，并使用 *Substitute Algorithm*（139）将其他函数的算法替换掉。

如果两个毫不相关的 classes 内出现 Duplicated Code，你应该考虑对其中一个使用 *Extract Class*（149），将重复代码提炼到一个独立 class 中，然后在另一个 class 内使用这个新 class。但是，重复代码所在的函数也可能的确只应该属于某个 class，另一个 class 只能调用它，抑或这个函数可能属于第三个 class，而另两个 classes 应该引用这第三个 class。你必须决定这个函数放在哪儿最合适，并确保它被安置后就不会再在其他任何地方出现。

3.2 Long Method（过长函数）

拥有「短函数」（short methods）的对象会活得比较好、比较长。不熟悉面向对象技术的人，常常觉得对象程序中只有无穷无尽的 *delegation*（委托），根本没有进行任何计算。和此类程序共同生活数年之后，你才会知道，这些小小函数有多大价值。「间接层」所能带来的全部利益 — 解释能力、共享能力、选择能力 — 都是

由小型函数支持的（请看 p.61 的「间接层和重构」）。

很久以前程序员就已认识到：程序愈长愈难理解。早期的编程语言中，「子程序调用动作」需要额外开销，这使得人们不太乐意使用 small method。现代 OO 语言几乎已经完全免除了进程（process）内的「函数调用动作额外开销」。不过代码阅读者还是得多费力气，因为他必须经常转换上下文去看看子程序做了什么。某些开发环境允许用户同时看到两个函数，这可以帮助你省去部分麻烦，但是让 small method 容易理解的真正关键在于一个好名字。如果你能给函数起个好名字，读者就可以通过名字了解函数的作用，根本不必去看其中写了些什么。

最终的效果是：你应该更积极进取地分解函数。我们遵循这样一条原则：每当感觉需要以注释来说明点什么的时候，我们就把需要说明的东西写进一个独立函数中，并以其用途（而非实现手法）命名。我们可以对一组或甚至短短一行代码做这件事。哪怕替换后的函数调用动作比函数自身还长，只要函数名称能够解释其用途，我们也该毫不犹豫地那么做。关键不在于函数的长度，而在于函数「做什么」和「如何做」之间的语义距离。

百分之九十九的场合里，要把函数变小，只需使用 *Extract Method*（110）。找到函数中适合集在一起的部分，将它们提炼出来形成一个新函数。

如果函数内有大量的参数和临时变量，它们会对你的函数提炼形成阻碍。如果你尝试运用 *Extract Method*（110），最终就会把许多这些参数和临时变量当作参数，传递给被提炼出来的新函数，导致可读性几乎没有任何提升。啊是的，你可以经常运用 *Replace Temp with Query*（120）来消除这些暂时元素。*Introduce Parameter Object*（295）和 *Preserve Whole Object*（288）则可以将过长的参数列变得更简洁些。

如果你已经这么做了，仍然有太多临时变量和参数，那就应该使出我们的杀手锏：*Replace Method with Method Object*（135）。

如何确定该提炼哪一段代码呢？一个很好的技巧是：寻找注释。它们通常是指出「代码用途和实现手法间的语义距离」的信号。如果代码前方有一行注释，就是在提醒你：可以将这段代码替换成一个函数，而且可以在注释的基础上给这个函数命名。就算只有一行代码，如果它需要以注释来说明，那也值得将它提炼到独立函数去。

条件式和循环常常也是提炼的信号。你可以使用 *Decompose Conditional*（238）处理条件式。至于循环，你应该将循环和其内的代码提炼到一个独立函数中。

3.3 Large Class（过大类）

如果想利用单一 class 做太多事情，其内往往就会出现太多 *instance* 变量。一旦如此，Duplicated Code 也就接踵而至了。

你可以运用 *Extract Class*（149）将数个变量一起提炼至新 class 内。提炼时应该选择 class 内彼此相关的变量，将它们放在一起。例如 "depositAmount" 和 "depositCurrency" 可能应该隶属同一个 class。通常如果 class 内的数个变量有着相同的前缀或字尾，这就意味有机会把它们提炼到某个组件内。如果这个组件适合作为一个 subclass，你会发现 *Extract Subclass*（330）往往比较简单。

有时候 class 并非在所有时刻都使用所有 *instance* 变量。果真如此，你或许可以多次使用 *Extract Class*（149）或 *Extract Subclass*（330）。

和「太多 *instance* 变量」一样，class 内如果有太多代码，也是「代码重复、混乱、死亡」的绝佳滋生地点。最简单的解决方案（还记得吗，我们喜欢简单的解决方案）是把赘余的东西消弭于 class 内部。如果有五个「百行函数」，它们之中很多代码都相同，那么或许你可以把它们变成五个「十行函数」和十个提炼出来的「双行函数」。

和「拥有太多 *instance* 变量」一样，一个 class 如果拥有太多代码，往往也适合使用 *Extract Class*（149）和 *Extract Subclass*（330）。这里有个有用技巧：先确定客户端如何使用它们，然后运用 *Extract Interface*（341）为每一种使用方式提炼出一个接口。这或许可以帮助你看清楚如何分解这个 class。

如果你的 Large Class 是个 GUI class，你可能需要把数据和行为移到一个独立的领域对象（domain object）去。你可能需要两边各保留一些重复数据，并令这些数据同步（sync.）。*Duplicate Observed Data*（189）告诉你该怎么做。这种情况下，特别是如果你使用旧式 Abstract Windows Toolkit（AWT）组件，你可以采用这种方式去掉 GUI class 并代以 Swing 组件。

3.4 Long Parameter List（过长参数列）

刚开始学习编程的时候，老师教我们：把函数所需的所有东西都以参数传递进去。这可以理解，因为除此之外就只能选择全局数据，而全局数据是邪恶的东西。对象技术改变了这一情况，因为如果你手上没有你所需要的东西，总可以叫另一个对象

给你。因此，有了对象，你就不必把函数需要的所有东西都以参数传递给它了，你只需传给它足够的东西、让函数能从中获得自己需要的所有东西就行了。函数需要的东西多半可以在函数的宿主类（host class）中找到。面向对象程序中的函数，其参数列通常比在传统程序中短得多。

这是好现象，因为太长的参数列难以理解，太多参数会造成前后不一致、不易使用，而且一旦你需要更多数据，就不得不修改它。如果将对象传递给函数，大多数修改都将没有必要，因为你很可能只需（在函数内）增加一两条请求（requests），就能得到更多数据。

如果「向既有对象发出一条请求」就可以取得原本位于参数列上的一份数据，那么你应该激活重构准则 *Replace Parameter with Method*（292）。上述的既有对象可能是函数所属 class 内的一个值域（field），也可能是另一个参数。你还可以运用 *Preserve Whole Object*（288）将来自同一对象的一堆数据收集起来，并以该对象替换它们。如果某些数据缺乏合理的对象归属，可使用 *Introduce Parameter Object*（295）为它们制造出一个「参数对象」。

此间存在一个重要的例外。有时候你明显不希望造成「被调用之对象」与「较大对象」间的某种依存关系。这时候将数据从对象中拆解出来单独作为参数，也很合情合理。但是请注意其所引发的代价。如果参数列太长或变化太频繁，你就需要重新考虑自己的依存结构（dependency structure）了。

3.5 Divergent Change（发散式变化）

我们希望软件能够更容易被修改 — 毕竟软件再怎么说本来就该是「软」的。一旦需要修改，我们希望能够跳到系统的某一点，只在该处做修改。如果不能做到这点，你就嗅出两种紧密相关的刺鼻味道中的一种了。

单一职责

如果某个 class 经常因为不同的原因在不同的方向上发生变化，Divergent Change 就出现了。当你看着一个 class 说：『呃，如果新加入一个数据库，我必须修改这三个函数；如果新出现一种金融工具，我必须修改这四个函数』，那么此时也许将这个对象分成两个会更好，这么一来每个对象就可以只因一种变化而需要修改。当然，往往只有在加入新数据库或新金融工具后，你才能发现这一点。针对某一外界变化的所有相应修改，都只应该发生在单一 class 中，而这个新 class 内的所有内容都应该反应该外界变化。为此，你应该找出因着某特定原因而造成的所有变化，然后运用 *Extract Class*（149）将它们提炼到另一个 class 中。

3.6 Shotgun Surgery（霰弹式修改）

Shotgun Surgery 类似 Divergent Change，但恰恰相反。如果每遇到某种变化，你都必须在许多不同的 classes 内做出许多小修改以响应之，你所面临的坏味道就是 Shotgun Surgery。如果需要修改的代码散布四处，你不但很难找到它们，也很容易忘记某个重要的修改。

这种情况下你应该使用 *Move Method*（142）和 *Move Field*（146）把所有需要修改的代码放进同一个 class。如果眼下没有合适的 class 可以安置这些代码，就创造一个。通常你可以运用 *Inline Class*（154）把一系列相关行为放进同一个 class。这可能会造成少量 Divergent Change，但你可以轻易处理它。

Divergent Change 是指「一个 class 受多种变化的影响」，Shotgun Surgery 则是指「一种变化引发多个 classes 相应修改」。这两种情况下你都会希望整理代码，取得「外界变化」与「待改类」呈现一对一关系的理想境地。

3.7 Feature Envy（依恋情结）

对象技术的全部要点在于：这是一种「将数据和加诸其上的操作行为包装在一起」的技术。有一种经典气味是：函数对某个 class 的兴趣高过对自己所处之 host class 的兴趣。这种倾慕之情最通常的焦点便是数据。无数次经验里，我们看到某个函数为了计算某值，从另一个对象那儿调用几乎半打的取值函数（getting method）。疗法显而易见：把这个函数移至另一个地点。你应该使用 *Move Method*（142）把它移到它该去的地方。有时候函数中只有一部分受这种依恋之苦，这时候你应该使用 *Extract Method*（110）把这一部分提炼到独立函数中，再使用 *Move Method*（142）带它去它的梦中家园。

当然，并非所有情况都这么简单。一个函数往往会用上数个 classes 特性，那么它究竟该被置于何处呢？我们的原则是：判断哪个 class 拥有最多「被此函数使用」的数据，然后就把这个函数和那些数据摆在一起。如果先以 *Extract Method*（110）将这个函数分解为数个较小函数并分别置放于不同地点，上述步骤也就比较容易完成了。

有数个复杂精巧的模式（patterns）破坏了这个规则。说起这个话题，「四巨头」[Gang of Four] 的 ***Strategy*** 和 ***Visitor*** 立刻跳入我的脑海，Kent Beck 的 ***Self Delegation*** [Beck] 也在此列。使用这些模式是为了对抗坏味道 Divergent Change。最根本的原则是：将总是一起变化的东西放在一块儿。「数据」和「引用这些数据」的行为

总是一起变化的，但也有例外。如果例外出现，我们就搬移那些行为，保持「变化只在一地发生」。***Strategy*** 和 ***Visitor*** 使你得以轻松修改函数行为，因为它们将少量需被覆写（overridden）的行为隔离开来 — 当然也付出了「多一层间接性」的代价。

3.8 Data Clumps（数据泥团）

数据项（data items）就像小孩子：喜欢成群结队地待在一块儿。你常常可以在很多地方看到相同的三或四笔数据项：两个 classes 内的相同值域（field）、许多函数签名式（signature）中的相同参数。这些「总是绑在一起出现的数据」真应该放进属于它们自己的对象中。首先请找出这些数据的值域形式（field）出现点，运用 *Extract Class*（149）将它们提炼到一个独立对象中。然后将注意力转移到函数签名式（signature）上头，运用 *Introduce Parameter Object*（295）或 *Preserve Whole Object*（288）为它减肥。这么做的直接好处是可以将很多参数列缩短，简化函数调用动作。是的，不必因为 Data Clumps 只用上新对象的一部分值域而在意，只要你以新对象取代两个（或更多）值域，你就值回票价了。

一个好的评断办法是：删掉众多数据中的一笔。其他数据有没有因而失去意义？如果它们不再有意义，这就是个明确信号：你应该为它们产生一个新对象。

缩短值域个数和参数个数，当然可以去除一些坏味道，但更重要的是：一旦拥有新对象，你就有机会让程序散发出一种芳香。得到新对象后，你就可以着手寻找 Feature Envy，这可以帮你指出「可移至新 class」中的种种程序行为。不必太久，所有 classes 都将在它们的小小社会中充分发挥自己的生产力。

3.9 Primitive Obsession（基本型别偏执）

大多数编程环境都有两种数据：结构型别（record types）允许你将数据组织成有意义的形式；基本型别（primitive types）则是构成结构型别的积木块。结构总是会带来一定的额外开销。它们有点像数据库中的表格，或是那些得不偿失（只为做一两件事而创建，却付出太大额外开销）的东西。

对象的一个极具价值的东西是：它们模糊（甚至打破）了横亘于基本数据和体积较大的 classes 之间的界限。你可以轻松编写出一些与语言内置（基本）型别无异的小型 classes。例如 Java 就以基本型别表示数值，而以 class 表示字符串和日期 — 这两个型别在其他许多编程环境中都以基本型别表现。

对象技术的新手通常不愿意在小任务上运用小对象 — 像是结合数值和币别的 money class、含一个起始值和一个结束值的 range class、电话号码或邮政编码（ZIP）等等的特殊 strings。你可以运用 *Replace Data Value with Object*（175）将原本单独存在的数据值替换为对象，从而走出传统的洞窟，进入炙手可热的对象世界。如果欲替换之数据值是 type code（型别码），而它并不影响行为，你可以运用 *Replace Type Code with Class*（218）将它换掉。如果你有相依于此 *type code* 的条件式，可运用 *Replace Type Code with Subclass*（213）或 *Replace Type Code with State/Strategy*（227）加以处理。

如果你有一组应该总是被放在一起的值域（fields），可运用 *Extract Class*（149）。如果你在参数列中看到基本型数据，不妨试试 *Introduce Parameter Object*（295）。如果你发现自己正从 array 中挑选数据，可运用 *Replace Array with Object*（186）。

3.10 Switch Statements（switch 惊悚现身）

面向对象程序的一个最明显特征就是：少用 switch（或 case）语句。从本质上说，switch 语句的问题在于重复（duplication）。你常会发现同样的 switch 语句散布于不同地点。如果要为它添加一个新的 case 子句，你必须找到所有 switch 语句并修改它们。面向对象中的多态（polymorphism）概念可为此带来优雅的解决办法。

大多数时候，一看到 switch 语句你就应该考虑以「多态」来替换它。问题是多态该出现在哪儿？switch 语句常常根据 type code（型别码）进行选择，你要的是「与该 type code 相关的函数或 class」。所以你应该使用 *Extract Method*（110）将 switch 语句提炼到一个独立函数中，再以 *Move Method*（142）将它搬移到需要多态性的那个 class 里头。此时你必须决定是否使用 *Replace Type Code with Subclasses*（223）或 *Replace Type Code with State/Strategy*（227）。一旦这样完成继承结构之后，你就可以运用 *Replace Conditional with Polymorphism*（255）了。

如果你只是在单一函数中有些选择事例，而你并不想改动它们，那么「多态」就有点杀鸡用牛刀了。这种情况下 *Replace Parameter with Explicit Methods*（285）是个不错的选择。如果你的选择条件之一是 null，可以试试 *Introduce Null Object*（260）。

3.11 Parallel Inheritance Hierarchies（平行继承体系）

Parallel Inheritance Hierarchies 其实是 Shotgun Surgery 的特殊情况。在这种情况下，每当你为某个 class 增加一个 subclass，必须也为另一个 class 相应增加一个 subclass。如果你发现某个继承体系的 class 名称前缀和另一个继承体系的 class 名称前缀完全相同，便是闻到了这种坏味道。

消除这种重复性的一般策略是：让一个继承体系的实体（instances）指涉（参考、引用、*refer to*）另一个继承体系的实体（instances）。如果再接再厉运用 *Move Method*（142）和 *Move Field*（146），就可以将指涉端（referring class）的继承体系消弭于无形。

3.12 Lazy Class（冗赘类）

你所创建的每一个 class，都得有人去理解它、维护它，这些工作都是要花钱的。如果一个 class 的所得不值其身价，它就应该消失。项目中经常会出现这样的情况：某个 class 原本对得起自己的身价，但重构使它身形缩水，不再做那么多工作；或开发者事前规划了某些变化，并添加一个 class 来应付这些变化，但变化实际上没有发生。不论上述哪一种原因，请让这个 class 庄严赴义吧。如果某些 subclass 没有做满足够工作，试试 *Collapse Hierarchy*（344）。对于几乎没用的组件，你应该以 *Inline Class*（154）对付它们。

3.13 Speculative Generality（夸夸其谈未来性）

这个令我们十分敏感的坏味道，命名者是 Brian Foote。当有人说『噢，我想我们总有一天需要做这事』并因而企图以各式各样的挂勾（hooks）和特殊情况来处理一些非必要的事情，这种坏味道就出现了。那么做的结果往往造成系统更难理解和维护。如果所有装置都会被用到，那就值得那么做；如果用不到，就不值得。用不上的装置只会挡你的路，所以，把它搬开吧。

如果你的某个 abstract class 其实没有太大作用，请运用 *Collapse Hierarchy*（344）。非必要之 delegation（委托）可运用 *Inline Class*（154）除掉。如果函数的某些参数未被用上，可对它实施 *Remove Parameter*（277）。如果函数名称带有多余的抽象意味，应该对它实施 *Rename Method*（273）让它现实一些。

如果函数或 class 的惟一用户是 test cases（测试用例），这就飘出了坏味道 Speculative Generality。如果你发现这样的函数或 class，请把它们连同其 *test*

cases 都删掉。但如果它们的用途是帮助 test cases 检测正当功能，当然必须刀下留人。

3.14 Temporary Field（令人迷惑的暂时值域）

有时你会看到这样的对象：其内某个 instance 变量仅为某种特定情势而设。这样的代码让人不易理解，因为你通常认为对象在所有时候都需要它的所有变量。在变量未被使用的情况下猜测当初其设置目的，会让你发疯。

请使用 Extract Class（149）给这个可怜的孤儿创造一个家，然后把所有和这个变量相关的代码都放进这个新家。也许你还可以使用 Introduce Null Object（260）在「变量不合法」的情况下创建一个 Null 对象，从而避免写出「条件式代码」。

如果 class 中有一个复杂算法，需要好几个变量，往往就可能导致坏味道 Temporary Field 的出现。由于实现者不希望传递一长串参数（想想为什么），所以他把这些参数都放进值域（fields）中。但是这些值域只在使用该算法时才有效，其他情况下只会让人迷惑。这时候你可以利用 Extract Class（149）把这些变量和其相关函数提炼到一个独立 class 中，提炼后的新对象将是一个 method object [Beck]（译注：其存在只是为了提供调用函数的途径，class 本身并无抽象意味）。

3.15 Message Chains（过度耦合的消息链）

如果你看到用户向一个对象索求（request）另一个对象，然后再向后者索求另一个对象，然后再索求另一个对象……这就是 Message Chain。实际代码中你看到的可能是一长串 `getThis()` 或一长串临时变量。采取这种方式，意味客户将与查找过程中的航行结构（structure of navigation）紧密耦合。一旦对象间的关系发生任何变化，客户端就不得不做出相应修改。

这时候你应该使用 Hide Delegate（157）。你可以在 Message Chain 的不同位置进行这种重构手法。理论上你可以重构 Message Chain 上的任何一个对象，但这么做往往会把所有中介对象（intermediate object）都变成 Middle Man。通常更好的选择是：先观察 Message Chain 最终得到的对象是用来干什么的，看看能否以 Extract Method（110）把使用该对象的代码提炼到一个独立函数中，再运用 Move Method（142）把这个函数推入 Message Chain。如果这条链上的某个对象有多位客户打算航行此航线的剩余部分，就加一个函数来做这件事。

有些人把任何函数链（method chain。译注：就是 Message Chain；面向对象领域中所谓「发送消息」就是「调用函数」）都视为坏东西，我们不这样想。呵呵，我们的冷静镇定是出了名的，起码在这件事情上是这样。

3.16 Middle Man（中间转手人）

对象的基本特征之一就是封装（encapsulation）－对外部世界隐藏其内部细节。封装往往伴随 delegation（委托）。比如说你问主管是否有时间参加一个会议，他就把这个消息委托给他的记事簿，然后才能回答你。很好，你没必要知道这位主管到底使用传统记事簿或电子记事簿抑或秘书来记录自己的约会。

但是人们可能过度运用 delegation。你也许会看到某个 class 接口有一半的函数都委托给其他 class，这样就是过度运用。这时你应该使用 *Remove Middle Man*（160），直接和实责对象打交道。如果这样「不干实事」的函数只有少数几个，可以运用 *Inline Method*（117）把它们 "*inlining*"，放进调用端。如果这些 Middle Man 还有其他行为，你可以运用 *Replace Delegation with Inheritance*（355）把它变成实责对象的 subclass，这样你既可以扩展原对象的行为，又不必负担那么多的委托动作。

3.17 Inappropriate Intimacy（狎昵关系）

有时你会看到两个 classes 过于亲密，花费太多时间去探究彼此的 private 成分。如果这发生在两个「人」之间，我们不必做卫道之士；但对于 classes，我们希望它们严守清规。

就像古代恋人一样，过份狎昵的 classes 必须拆散。你可以采用 *Move Method*（142）和 *Move Field*（146）帮它们划清界线，从而减少狎昵行径。你也可以看看是否运用 *Change Bidirectional Association to Unidirectional*（200）让其中一个 class 对另一个斩断情丝。如果两个 classes 实在是情投意合，可以运用 *Extract Class*（149）把两者共同点提炼到一个安全地点，让它们坦荡地使用这个新 class。或者也可以尝试运用 *Hide Delegate*（157）让另一个 class 来为它们传递相思情。

继承（inheritance）往往造成过度亲密，因为 subclass 对 superclass 的了解总是超过 superclass 的主观愿望。如果你觉得该让这个孩子独自生活了，请运用 *Replace Inheritance with Delegation*（352）让它离开继承体系。

3.18 Alternative Classes with Different Interfaces（异曲同工的类）

如果两个函数做同一件事，却有着不同的签名式（signature），请运用 *Rename Method*

（273）根据它们的用途重新命名。但这往往不够，请反复运用 *Move Method*（142）将某些行为移入 classes，直到两者的协议（protocols）一致为止。如果你必须重复而赘余地移入代码才能完成这些，或许可运用 *Extract Superclass*（336）为自己赎点罪。

3.19 Incomplete Library Class（不完美的程序库类）

复用（reuse）常被视为对象的终极目的。我们认为这实在是过度估计了（我们只是使用而已）。但是无可否认，许多编程技术都建立在 library classes（程序库类）的基础上，没人敢说是不是我们都把排序算法忘得一干二净了。

library classes 构筑者没有未卜先知的能力，我们不能因此责怪他们。毕竟我们自己也几乎总是在系统快要构筑完成的时候才能弄清楚它的设计，所以 library 构筑者的任务真的很艰巨。麻烦的是 library 的形式（form）往往不够好，往往不可能让我们修改其中的 classes 使它完成我们希望完成的工作。这是否意味那些经过实践检验的战术如 *Move Method*（142）等等，如今都派不上用场了？

幸好我们有两个专门应付这种情况的工具。如果你只想修改 library classes 内的一两个函数，可以运用 *Introduce Foreign Method*（162）；如果想要添加一大堆额外行为，就得运用 *Introduce Local Extension*（164）。

3.20 Data Class（纯稚的数据类）

所谓 Data Class 是指：它们拥有一些值域（fields），以及用于访问（读写）这些值域的函数，除此之外一无长物。这样的 classes 只是一种「不会说话的数据容器」，它们几乎一定被其他 classes 过份细琐地操控着。这些 classes 早期可能拥有 public 值域，果真如此你应该在别人注意到它们之前，立刻运用 *Encapsulate Field*（206）将它们封装起来。如果这些 classes 内含容器类的值域（collection fields），你应该检查它们是不是得到了恰当的封装：如果没有，就运用 *Encapsulate Collection*（208）把它们封装起来。对于那些不该被其他 classes 修改的值域，请运用 *Remove Setting Method*（300）。

然后，找出这些「取值/设值」函数（getting and setting methods）被其他 classes 运用的地点。尝试以 *Move Method*（142）把那些调用行为搬移到 Data Class 来。如果无法搬移整个函数，就运用 *Extract Method*（110）产生一个可被搬移的函数。不久之后你就可以运用 *Hide Method*（303）把这些「取值/设值」函数隐藏起来了。

Data Class 就像小孩子。作为一个起点很好，但若要让它们像「成年（成熟）」的对象那样参与整个系统的工作，它们就必须承担一定责任。

3.21 Refused Bequest（被拒绝的遗赠）

subclasses 应该继承 superclass 的函数和数据。但如果它们不想或不需要继承，又该怎么办呢？它们得到所有礼物，却只从中挑选几样来玩！

按传统说法，这就意味继承体系设计错误。你需要为这个 subclass 新建一个兄弟（sibling class），再运用 *Push Down Method*（328）和 *Push Down Field*（329）把所有用不到的函数下推给那兄弟。这样一来 superclass 就只持有所有 subclasses 共享的东西。常常你会听到这样的建议：所有 superclasses 都应该是抽象的（abstract）。

既然使用「传统说法」这个略带贬义的词，你就可以猜到，我们不建议你这么做，起码不建议你每次都这么做。我们经常利用 subclassing 手法来复用一些行为，并发现这可以很好地应用于日常工作。这也是一种坏味道，我们不否认，但气味通常并不强烈。所以我们说：如果 Refused Bequest 引起困惑和问题，请遵循传统忠告。但不必认为你每次都得那么做。十有八九这种坏味道很淡，不值得理睬。

如果 subclass 复用了 superclass 的行为（实现），却又不愿意支持 superclass 的接口，Refused Bequest 的坏味道就会变得浓烈。拒绝继承 superclass 的实现，这一点我们不介意；但如果拒绝继承 superclass 的接口，我们不以为然。不过即使你不愿意继承接口，也不要胡乱修改继承体系，你应该运用 *Replace Inheritance with Delegation*（352）来达到目的。

3.22 Comments（过多的注释）

别担心，我们并不是说你不该写注释。从嗅觉上说，Comments 不是一种坏味道；事实上它们还是一种香味呢。我们之所以要在这里提到 Comments，因为人们常把它当作除臭剂来使用。常常会有这样的情况：你看到一段代码有着长长的注释，然后发现，这些注释之所以存在乃是因为代码很糟糕。这种情况的发生次数之多，实在令人吃惊。

Comments 可以带我们找到本章先前提到的各种坏味道。找到坏味道后，我们首先应该以各种重构手法把坏味道去除。完成之后我们常常会发现：注释已经变得多余了，因为代码已经清楚说明了一切。

如果你需要注释来解释一块代码做了什么，试试 *Extract Method*（110）；如果 method 已经提炼出来，但还是需要注释来解释其行为，试试 *Rename Method*（273）；如果你需要注释说明某些系统的需求规格，试试 *Introduce Assertion*（267）。

当你感觉需要撰写注释，请先尝试重构，试着让所有注释都变得多余。

如果你不知道该做什么，这才是注释的良好运用时机。除了用来记述将来的打算之外，注释还可以用来标记你并无十足把握的区域。你可以在注释里写下自己「为什么做某某事」。这类信息可以帮助将来的修改者，尤其是那些健忘的家伙。

4

构筑测试体系

Building Test

如果你想进行重构（refactoring），首要前提就是拥有一个可靠的测试环境。就算你够幸运，有一个可以自动进行重构的工具，你还是需要测试。而且短时间内不可能有任何工具可以为我们自动进行所有可能的重构。

我并不把这视为缺点。我发现，编写优良的测试程序，可以极大提高我的编程速度，即使不进行重构也一样如此。这让我很吃惊，也违反许多程序员的直觉，所以我有必要解释一下这个现象。

4.1 自我测试代码（Self-testing Code）的价值

如果认真观察程序员把最多时间耗在哪里，你就会发现，编写代码其实只占非常小的一部分。有些时间用来决定下一步干什么，另一些时间花在设计上面，最多的时间则是用来调试（debug）。我敢肯定每一位读者都还记得自己花在调试上面的无数个小时，无数次通宵达旦。每个程序员都能讲出「花一整天（甚至更多）时间只找出一只小小臭虫」的故事。修复错误通常是比较快的，但找出错误却是噩梦一场。当你修好一个错误，总是会有另一个错误出现，而且肯定要很久以后才会注意到它。彼时你又要花上大把时间去寻找它。

我走上「自我测试代码」这条路，肇因于 1992 年 OOPSLA 大会上的一次演讲。会场上有人（我记得好像是 Dave Thomas）说：『class 应该包含它们自己的测试代码。』这激发了我的灵感，让我想到一种组织测试的好方法。我这样解释它：每个 class 都应该有一个测试函数，并以它来测试自己这个 class。

那时候我还着迷于增量式开发（incremental development），所以我尝试在结束每次增量时，为每个 class 添加测试。当时我开发的项目很小，所以我们大约每周增量一次。执行测试变得相当直率，但尽管如此，做这些测试还是很烦人，因为每个测

试都把结果输出到控制台（console），而我必须逐一检查它们。我是个很懒的人，我情愿当下努力工作以免除日后的工作。我意识到我其实完全不必自己盯着屏幕检验测试所得信息是否正确，我大可让计算机来帮我做这件事。我需要做的就是把我所期望的输出放进测试代码中，然后做一个比较就行了。于是我可以舒服地执行每个 class 的测试函数，如果一切都没问题，屏幕上就只出现一个 "OK"。现在，这些 classes 都变成「自我测试」了。

确保所有测试都完全自动化，让它们检查自己的测试结果。

此后再进行测试就简单多了，和编译一样简单。于是我开始在每次编译之后都进行测试。很快我发现自己的生产性能大大提高。我意识到那是因为我没有花太多时间去调试。如果我不小心引入一个可被原测试捕捉到的错误，那么只要我执行测试，它就会向我报告这个错误。由于测试本来是可以正常运行的，所以我知道这个错误必定是在前一次执行测试后引入。由于我频繁地进行测试，每次测试都在不久之前，因此我知道错误的源头就是我刚刚写下的代码。而由于我对那段代码记忆犹新，份量也很小，所以轻松就能找到错误。从前需要一小时甚至更多时间才能找到的错误，现在最多只需两分钟就找到了。之所以能够拥有如此强大的侦错能力，不仅仅因为我构筑了 self-testing classes（自我测试类），也因为我频繁地运行它们。

注意到这一点后，我对测试的积极性更高了。我不再等待每次增量结束，只要写好一点功能，我就立即添加测试。每天我都会添加一些新功能，同时也添加相应的测试。那些日子里，我很少花一分钟以上的时间在调试上面。

一整组（a suite of）测试就是一个强大的「臭虫」侦测器，能够大大缩减查找「臭虫」所需要的时间。

当然，说服别人也这么做，并不容易。编写测试程序，意味要写很多额外代码。除非你确切体验到这种方法对编程速度的提升，否则自我测试就显不出它的意义。很多人根本没学过如何编写测试程序，甚至根本没考虑过测试，这对于编写自我测试代码也很不利。如果需要手动运行测试，那更是令人烦闷欲呕；但如果可以自动运行，编写测试代码就真的很有趣。

实际上，撰写测试代码的最有用时机是在开始编程之前。当你需要添加特性的时候，先写相应测试代码。听起来离经叛道，其实不然。编写测试代码其实就是在问自己：添加这个功能需要做些什么。编写测试代码还能使你把注意力集中于接口而非实现上头（这永远是件好事）。预先写好的测试代码也为你的工作安上一个明确的结束标志：一旦测试代码正常运行，工作就可以结束了。

「频繁进行测试」是极限编程（eXtreme Programming，XP） [Beck, XP] 的重要一环。「极限编程」一词容易让人联想起那些编码飞快、自由而散漫的黑客（hackers），实际上极限编程者都是十分专注的测试者。他们希望尽可能快速开发软件，而他们也知道「测试」可协助他们尽可能快速地前进。

争论至此可休矣。尽管我相信每个人都可以从编写自我测试代码中受益，但这并不是本书重点。本书谈的是重构，而重构需要测试。如果你想重构，你就必须编写测试代码。本章将教你「以 Java 编写测试代码」的起步知识。这不是一本专讲测试的书，所以我不想讲得太仔细。但我发现，少量测试就足以带来惊人的利益。

和本书其他内容一样，我以实例来介绍测试手法。开发软件的时候，我一边撰写代码，一边撰写测试代码。但是当我和他人并肩重构时，往往得面对许多无自我测试的代码。所以重构之前我们首先必须把这些代码改造为「自我测试」。

Java 之中的测试惯用手法是 "testing main"，意思是每个 class 都应该有一个用于测试的 `main()`。这是一个合理的习惯（尽管并不那么值得称许），但可能不好操控。这种作法的问题是很难轻松运行多个测试。另一种作法是：建立一个独立 class 用于测试，并在一个框架（framework）中运行它，使测试工作更轻松。

4.2 JUnit 测试框架（Testing Framework）

译注：本段将使用英文词：test-suite（测试套件）、test-case（测试用例）和 test-fixture（测试装备），期能直接对应图 4.1 的 JUnit 结构组件，并有助于阅读 JUnit 文档。

我用的是 JUnit，一个由 Erich Gamma 和 Kent Beck [JUnit] 开发的开放源码测试框架。这个框架非常简单，却可让你进行测试所需的所有重要事情。本章中我将运用这个测试框架来为一些 IO classes 开发测试代码。

首先我创建一个 `FileReaderTester` class 来测试文件读取器。任何「包含测试代码」的 class 都必须衍生自测试框架所提供的 `TestCase` class。这个框架运用 ***Composite***

模式 [Gang of Four]，允许你将测试代码聚集到 suites（套件）中，如图 4.1。这些套件可以包含未加工的 test-cases（测试用例），或其他 test-suits（测试套件）。如此一来我就可以轻松地将一系列庞大的 test-suits 结合在一起，并自动运行它们。

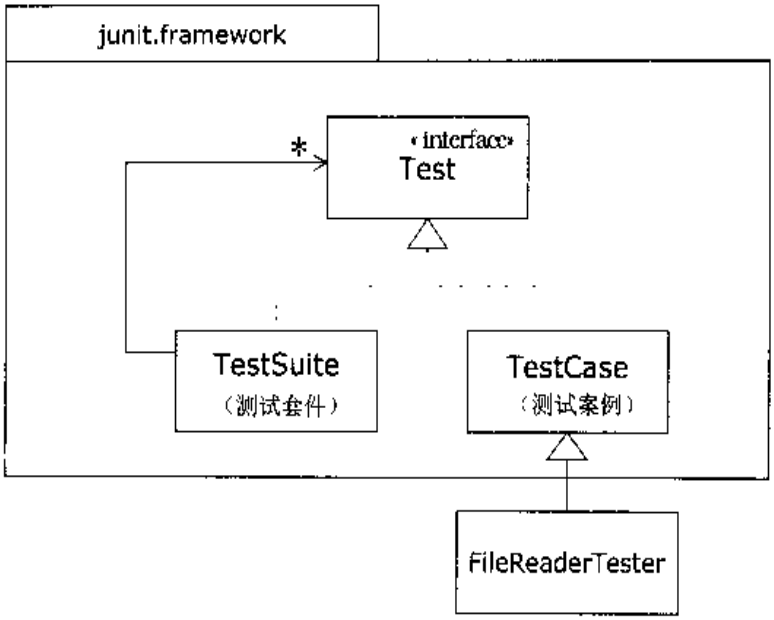


图 4.1 测试框架的 ***Composite*** 结构

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

这个新建的 class 必须有一个构造函数。完成之后我就可以开始添加测试代码了。我的第一件工作是设置 test fixture（测试装备），那是指「用于测试的对象样本」。由于我要读一个文件，所以先备妥一个测试文件如下：

```
Bradman     99.94    52   80   10   6996     334      29
Pollock     60.97    23   41   4    2256     274      7
Headley     60.83    22   40   4    2190     270*     10
Sutcliffe   60.73    54   84   9    4555     194      16
```

进一步运用这个文件之前，我得先准备好 test fixture（测试装备）。TestCase class 提供两个函数专门针对此一用途：setUp()用来产生相关对象、tearDown()负责删除它们。在 TestCase class 中这两个函数都只有空壳。大多数时候你不需要操心 test fixture 的拆除（垃圾回收器会扛起责任），但是在这里，以 tearDown() 关闭文件无疑是明智之举：

```
class FileReaderTester...
    protected void setUp() {
        try {
            _input = new FileReader("data.txt");
        } catch (FileNotFoundException e) {
            throw new RuntimeException ("unable to open test file");
        }
    }

    protected void tearDown() {
        try {
            _input.close();
        } catch (IOException e) {
            throw new RuntimeException ("error on closing test file");
        }
    }
```

现在我有了适当的 test fixture（测试装备），可以开始编写测试代码了。首先要测试的是 read()，我要读取一些字符，然后检查后续读取的字符是否正确：

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assert('d' == ch);
}
```

assert()扮演自动测试角色。如果 assert()的参数值为 true，一切良好；否则我们就会收到错误通知。稍后我会让你看看测试框架怎么向用户报告错误消息。现在我要先介绍如何将测试过程运行起来。

第一步是产生一个 test suite（测试套件）。为达目的，请设计一个 suite()如下：

```
class FileReaderTester...
    public static Test suite() {
        TestSuite suite= new TestSuite();
        suite.addTest(new FileReaderTester("testRead"));
        return suite;
    }
```

这个测试套件只含一个 test-case（测试用例）对象，那是个 FileReaderTester 实体。创建 test-case 对象时，我传给其构造函数一个字符串，这正是待测函数的名称。这会创建出一个对象，用以测试被指定的函数。这个测试系通过 Java 反射机制（reflection）和对象系结在一起。你可以自由下载 JUnit 源码，看看它究竟如何做到。至于我，我只把它当作一种魔法。

要将整个测试运行起来，还需要一个独立的 TestRunner class。TestRunner 有两个版本，其中一个有漂亮的图形用户界面（GUI），另一个采用文字界面。我可以在 main 函数中调用「文字界面」版：

```
class FileReaderTester...
    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
}
```

这段代码创建出一个 TestRunner，并要它去测试 FileReaderTester class。当我执行它，我看到：

```
.
Time: 0.110
OK (1 tests)
```

对于每个运行起来的测试，JUnit 都会输出一个句点，这样你就可以直观看到测试进展。它会告诉你整个测试用了多长时间。如果所有测试都没有出错，它就会说"OK"，并告诉你运行了多少笔测试。我可以运行上千笔测试，如果一切良好，我会看到那个 "OK"。对于自我测试代码来说，这个简单的响应至关重要，没有它我就不可能经常运行这些测试。有了这个简单响应，你可以执行一大堆测试然后去吃个午饭（或开个会），回来之后再看看测试结果。

频繁地运行测试。每次编译请把测试也考虑进去 — 每天至少执行每个测试一次。

重构过程中，你可以只运行少数几项测试，它们主要用来检查当下正在开发或整理的代码。是的，你可以只运行少数几项测试，这样肯定比较快，否则整个测试会减低你的开发速度，使你开始犹豫是否还要这样下去。千万别屈服于这种诱惑，否则你一定会付出代价。

如果测试出错，会发生什么事？为了展示这种情况，我故意放一只臭虫进去：

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assert([illegible] == ch);          // deliberate error
}
```

得到如下结果：

```
.F
Time: 0.220
!!!FAILURES!!!
```

```
Test Results:
Run: 1 Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead
test.framework.AssertionFailedError
```

JUnit 警告我测试失败，并告诉我这项失败具体发生在哪个测试身上。不过这个错误消息并不特别有用。我可以使用另一种形式的 assert，让错误消息更清楚些：

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assertEquals('m',ch);
}
```

你做的绝大多数 asserts 都是对两个值进行比较，检验它们是否相等，所以 JUnit 框架为你提供 assertEquals()。这个函数很简单：以 equals()进行对象比较，以操作符 == 进行数值比较 — 我自己常忘记区分它们。这个函数也输出更具意义的错误消息：

```
.F
Time: 0.170
!!!FAILURES!!!
Test Results:
Run: 1 Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead "expected:<m> but was:<d>"
```

我应该提一下：编写测试代码时，我往往一开始先让它们失败。面对既有代码，要不我就修改它（如果我能碰触源码的话），使它测试失败，要不就在 assertions 中放一个错误期望值，造成测试失败。之所以这么做，是为了向自己证明：测试机制的确可以运行，并且的确测试了它该测试的东西（这就是为什么上面两种作法中我比较喜欢修改待测码的原因）。这可能有些偏执，或许吧，但如果测试代码所测的东西并非你想测的东西，你真的有可能被搞得迷迷糊糊。

除了捕捉「失败」（failures，也就是 assertions 之结果为 "false"），JUnit 还可以捕捉「错误」（errors，意料外的异常）。如果我关闭 input stream，然后试图读取它，就应该得到一个异常（exception）。我可以这样测试：

```
public void testRead() throws IOException {
    char ch = '&';
    _input.close();
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();    // will throw exception
    assertEquals('m',ch);
}
```

执行上述测试，我得到这样的结果：

```
.E
Time: 0.110
!!!FAILURES!!!
Test Results:
Run: 1 Failures: 0 Errors: 1
There was 1 error:
1) FileReaderTester.testRead
java.io.IOException: Stream closed
```

区分失败（failures）和错误（errors）是很有用的，因为它们的出现形式不同，排除的过程也不同。

JUnit 还包含一个很好的图形用户界面（GUI，见图 4.2）。如果所有测试都顺利通过，窗口下端的进度杆（progress bar）就呈绿色；如果有任何一个测试失败，进度杆就呈红色。你可以丢下这个 GUI 不管，整个环境会自动将你在代码所做的任何修改连接（links）进来。这是一个非常方便的测试环境。

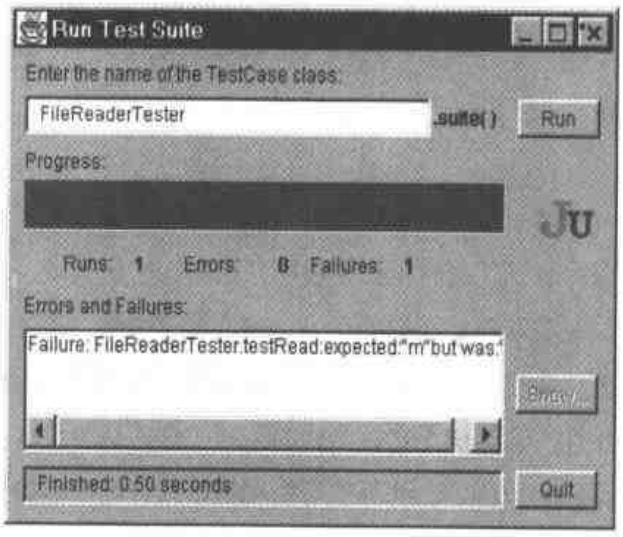


图 4.2 JUnit 的图形用户界面

单元测试（Unit Tests）和功能测试（Functional Tests）

JUnit 框架的用途是单元测试，所以我应该讲讲单元测试和功能测试之间的差异。我一直挂在嘴上的其实是「单元测试」，编写这些测试的目的是为了提高身为一个程序员的生产性能。至于让质保部门开心，那只是附带效果而已。单元测试

是高度本地化（localized）的东西，每个 test class 只对单一 package 运作。它能够测试其他 packages 的接口，除此之外它将假设其他 package 一切正常。

功能测试就完全不同。它们用来保证软件能够正常运作。它们只负责向客户提供质量保证，并不关心程序员的生产力。它们应该由一个喜欢寻找臭虫的个别团队来开发。这个团队应该使用重量级工具和技术来帮助自己开发良好的功能测试。

一般而言，功能测试尽可能把整个系统当作一个黑箱。面对一个 GUI 待测系统，它们通过 GUI 来操作那个系统。面对文件更新程序或数据库更新程序，功能测试只观察特定输入所导致的数据变化。

一旦功能测试者或最终用户找到软件中的一只臭虫，要除掉它至少需要做两件事。当然你必须修改代码，才得以排除错误，但你还应该添加一个单元测试，让它揭发这只臭虫。事实上，每当收到臭虫提报（bug report），我都首先编写一个单元测试，使这只臭虫浮现。如果需要缩小臭虫出没范围，或如果出现其他相关失败（failures），我就会编写不只一个测试。我使用单元测试来帮助我盯住臭虫，并确保我的单元测试不会有类似的漏网之……呃……臭虫。

> **每当你接获臭虫提报（bug report），请先撰写一个单元测试来揭发这只臭虫。**

JUnit 框架被设计用来编写单元测试。功能测试往往以其他工具辅助进行，例如某些拥有 GUI（图形用户界面）的测试工具，然而通常你还得撰写一些与你的应用程序息息相关的测试工具，使能够比单纯使用 GUI scripts（脚本语言）更轻松地管理 test cases（测试用例）。你也可以运用 JUnit 来执行功能测试，但这通常不是最有效的形式。当我要进行重构时，我倚赖程序员的好朋友：单元测试。

4.3 添加更多测试

现在，我们应该继续添加更多测试。我遵循的风格是：观察 class 该做的所有事情，然后针对任何一项功能的任何一种可能失败情况，进行测试。这不同于某些程序员提倡的「测试所有 public 函数」。记住，测试应该是一种风险驱动（risk driven）行为，测试的目的是希望找出现在或未来可能出现的错误。所以我不会去测试那些仅仅读或写一个值域的访问函数（accessors），因为它们太简单了，不大可能出错。

这一点很重要，因为如果你撰写过多测试，结果往往测试量反而不够。我常常阅读许多测试相关书籍，我的反应是：测试需要做那么多工作，令我退避三舍。这种书起不了预期效果，因为它让你觉得测试有大量工作要做。事实上，哪怕只做一点点测试，你也能从中受益。测试的要诀是：测试你最担心出错的部分。这样你就能从测试工作中得到最大利益。

> **编写未臻完善的测试并实际运行，好过对完美测试的无尽等待。**

现在，我的目光落到了 read()。它还应该做些什么？文档上说，当 input stream 到达文件尾端，read()应该返回 -1（在我看来这并不是个很好的协议，不过我猜这会让 C 程序员倍感亲切）。让我们来测试一下。我的文本编辑器告诉我，我的测试文件共有 141 个字符，于是我撰写测试代码如下：

```
public void testReadAtEnd() throws IOException {
  int ch = -1234;
  for (int i = 0; i < 141; i++)
     ch = _input.read();
  assertEquals("read at end", -1, _input.read());
}
```

为了让这个测试运行起来，我必须把它添加到 test suit（测试套件）中：

```
public static Test suite() {    // 译注：原本在p.93
  TestSuite suite= new TestSuite();
  suite.addTest(new FileReaderTester("testRead"));
  suite.addTest(new FileReaderTester("testReadAtEnd"));
  return suite;
}
```

当 test suit（测试套件）运行起来，它会告诉我它的每个成分 — 也就是这两个 test cases（测试用例） 的运行情况。每个用例都会调用 setUp()，然后执行测试代码，最终调用 tearDown()。每次测试都调用 setUp()和 tearDown()是很重要的，因为这样才能保证测试之间彼此隔离。也就是说我们可以按任意顺序运行它们，不会对它们的结果造成任何影响。

老要记住将 test cases 添加到 suite()，实在是件痛苦的事。幸运的是 Erich Gamma 和 Kent Beck 和我一样懒，所以他们提供了一条途径来避免这种痛苦。TestSuite class 有个特殊构造函数，接受一个 class 为参数，创建出来的 test suite 会将该 class

内所有以 "test" 起头的函数都当作 test cases 包含进来。如果遵循这一命名习惯，就可以把我的 main() 改为这样：

```
public static void main (String[] args) {      // 译注：原下p94
  junit.textui.TestRunner.run (new TestSuite(FileReaderTester.class));
}
```

这样，我写的每一个测试函数便都被自动添加到 test suite 中。

测试的一项重要技巧就是「寻找边界条件」。对 read() 而言，边界条件应该是第一个字符、最后一个字符、倒数第二个字符：

```
public void testReadBoundaries()throws IOException {
    assertEquals("read first char",'B', _input.read());
    int ch;
    for (int i = 1; i <140; i++)
        ch = _input.read();
    assertEquals("read last char",'6',_input.read());
    assertEquals("read at end",-1,_input.read());
}
```

你可以在 assertions 中加入一条消息。如果测试失败，这条消息就会被显示出来。

> 考虑可能出错的边界条件，把测试火力集中在那儿。

「寻找边界条件」也包括寻找特殊的、可能导致测试失败的情况。对于文件相关测试，空文件是个不错的边界条件：

```
public void testEmptyRead()throws IOException {
  File empty = new File ("empty.txt");
  FileOutputStream out = new FileOutputStream (empty);
  out.close();
  FileReader in = new FileReader (empty);
  assertEquals (-1, in.read());
}
```

现在我为这个测试产生一些额外的 test fixture（测试装备）。如果以后还需要空文件，我可以把这些代码移至 setUp()，从而将「空文件」加入常规 test fixture。

```
protected void setUp(){
  try {
     _input = new FileReader("data.txt");
     _empty = newEmptyFile();
  } catch(IOException e){
     throw new RuntimeException(e.toString());
  }
}
```

```
private FileReader newEmptyFile() throws IOException {
  File empty = new File ("empty.txt");
  FileOutputStream out = new FileOutputStream(empty);
  out.close();
  return newFileReader(empty);
}

public void testEmptyRead() throws IOException {
  assertEquals (-1, _empty.read());
}
```

如果读取文件末尾之后的位置，会发生什么事？同样应该返回-1。现在我再加一个测试来探测这一点：

```
public void testReadBoundaries() throws IOException {
  assertEquals("read first char",'B', _input.read());
  int ch;
  for (int i = 1;i <140; i++)
     ch = _input.read();
  assertEquals("read last char", '6', _input.read());
  assertEquals("read at end",-1,_input.read());
  assertEquals ("readpast end", -1, _input.read());
}
```

注意，我在这里扮演「程序公敌」的角色。我积极思考如何破坏代码。我发现这种思维能够提高生产力，并且很有趣。它纵容了我心智中比较促狭的那一部分。

测试时，别忘了检查预期的错误是否如期出现。如果你尝试在 stream 被关闭后再读取它，就应该得到一个 **IOException** 异常，这也应该被测试出来：

```
public void testReadAfterClose() throws IOException{
  _input.close();
  try {
     _input.read();
     fail ("no exception for read past end");
  } catch (IOException io) {}
}
```

IOException 之外的任何异常都将以一般方式形成一个错误。

当事情被大家认为应该会出错时，别忘了检查彼时是否有异常如预期般地被抛出。

请遵循这些规则，不断丰富你的测试。对于某些比较复杂的 classes，可能你得花费一些时间来浏览其接口，但是在此过程中你可以真正理解这个接口。而且这对于考虑错误情况和边界情况特别有帮助。这是在编写代码的同时（甚至之前）编写测试代码的另一个好处。

随着 tester classes 愈来愈多，你可以产生另一个 class，专门用来包含「由其他 tester classes 所形成」的测试套件（test suite）。这很容易做到，因为一个测试套件本来就可以包含其他测试套件。这样，你就可以拥有一个「主控的」（master）test class：

```
class MasterTester extends TestCase {
  public static void main (String[] args) {
      junit.textui.TestRunner.run (suite());
  }
  public static Test suite() {
      TestSuite result = new TestSuite();
      result.addTest(new TestSuite(FileReaderTester.class));
      result.addTest(new TestSuite(FileWriterTester.class));
      // and so on...
      return result;
  }
}
```

什么时候应该停下来？我相信这样的话你听过很多次：「任何测试都不能证明一个程序没有臭虫」。这是真的，但这不会影响「测试可以提高编程速度」。我曾经见过数种测试规则建议，其目的都是保证你能够测试所有情况的一切组合。这些东西值得一看，但是别让它们影响你。当测试数量达到一定程度之后，测试效益就会呈现递减态势，而非持续递增；如果试图编写太多测试，你也可能因为工作量太大而气馁，最后什么都写不成。你应该把测试集中在可能出错的地方。观察代码，看哪儿变得复杂；观察函数，思考哪些地方可能出错。是的，你的测试不可能找出所有臭虫，但一旦进行重构，你可以更好地理解整个程序，从而找到更多臭虫。虽然我总是以单独一个测试套件（test suite）开始重构，但前进途中我总会加入更多测试。

不要因为「测试无法捕捉所有臭虫」，就不撰写测试代码，因为测试的确可以捕捉到大多数臭虫。

对象技术有个微妙处：继承（inheritance）和多态（polymorphism）会让测试变得比较困难，因为将有许多种组合需要测试。如果你有三个彼此合作的 abstract classes，每个 abstract class 有三个 subclasses，那么你总共拥有九个可供选择的 classes，和

27种组合。我并不总是试着测试所有可能组合，但我会尽量测试每一个classes，这可以大大减少各种组合所造成的风险。如果这些classes之间彼此有合理的独立性，我很可能不会尝试所有组合。是的，我总有可能遗漏些什么，但我觉得「花合理时间抓出大多数臭虫」要好过「穷尽一生抓出所有臭虫」。

测试代码和产品代码（待测代码）之间有个区别：你可以放心地拷贝、编辑测试代码。处理多种组合情况以及面对多个可供选择的classes时，我经常这么做。首先测试「标准发薪过程」，然后加上「资历」和「年底前停薪」条件，然后又去掉这两个条件……。只要在合理的测试装备（test fixture）上准备好一些简单的替换样本，我就能够很快生成不同的test case（测试用例），然后就可以利用重构手法分解出真正常用的各种东西。

我希望这一章能够让你对于「撰写测试代码」有一些感觉。关于这个主题，我可以说上很多，但如果那么做，就有点喧宾夺主了。总而言之，请构筑一个良好的臭虫检测器（bug detector）并经常运行它；这对任何开发工作都是一个美好的工具，并且是重构的前提。

5

重构名录

Toward a Catalog of Refactorings

本书5~12章构成了一份重构名录草案（initial catalog of refactorings）。其中所列的重构手法来自我最近数年的心得。这份名录并非钜细靡遗，但应该足可为你提供一个坚实的起点，让你得以开始自己的重构工作。

5.1 重构的记录格式（Format of Refactorings）

介绍重构时，我采用一种标准格式。每个重构手法都有如下五个部分：

- 首先是名称（**name**）。建造一个重构词汇表，名称是很重要的。这个名称也就是我将在本书其他地方使用的名称。
- 名称之后是一个简短概要（**summary**），简单介绍此一重构手法的适用情景，以及它所做的事情。这部分可以帮助你更快找到你所需要的重构手法。
- 动机（**motivation**），为你介绍「为什么需要这个重构」和「什么情况下不该使用这个重构」。
- 作法（**mechanics**），简明扼要地一步一步介绍如何进行此一重构。
- 范例（**examples**），以一个十分简单的例子说明此重构手法如何运作。

「概要」（**summary**）包括二个部分：(1) 一个简短文句，介绍这个重构能够帮助的问题；(2) 一段简短陈述，介绍你应该做的事；(3) 一幅速写图，简单展现重构前后示例：有时候我展示代码，有时候我展示统一建模语言（UML）图。哪一种形式能更好呈现该重构的本质，我就使用该种形式（本书所有UML图都根据实现观点（implementation perspective）而画 [Fowler, UML]。）如果你以前见过这一重构手法，那么速写图能够让你迅速了解这一重构的概况；如果你不曾见过这个重构，可能就需要浏览整个范例，才能得到较好的认识。

「作法」（**mechanics**）出自我自己的笔记。这些笔记是为了让我在一段时间不做某项重构之后还能记得怎么做。它们也颇为简洁，通常不会解释「为什么要这么做那么做」。我会在「范例」（**examples**）给出更多解释。这么一来「作法」就成了简短的笔记。如果你知道该使用哪个重构，但记不清具体步骤，可以参考「作法」部分（至少我是这么使用它们的）；如果你初次使用某个重构，可能「作法」对你还不够，你还需要阅读「范例」。

撰写「作法」的时候，我尽量将重构的每个步骤都写得简短。我强调安全的重构方式，所以应该采用非常小的步骤，并且在每个步骤之后进行测试。真正工作时我通常会采用比这里介绍的「婴儿学步」稍大些的步骤，然而一旦遇上臭虫，我就会撤销上一步，换用比较小的步骤。这些步骤还包含一些特定状况的参考，所以它们也有检验表（checklist）的作用：我自己经常忘掉这些该做的事情。

「范例」（**examples**）像是简单而有趣的教科书。我使用这些范例是为了帮助解释重构的基本要素，最大限度地避免其他枝节，所以我希望你能原谅其中的简化工作（它们当然不是优秀商用对象设计的适当例子）。不过我敢肯定你一定能在你手上那些更复杂的情况中使用它们。某些十分简单的重构干脆没有范例，因为我觉得为它们加上一个范例不会有多大意义。

更明确地说，加上「范例」仅仅是为了阐释当时讨论的重构手法。通常那些代码最终仍有其他问题，但修正那些问题需要用到其他重构手法。某些情况下数个重构经常被一并运用，这时候我会把某个范例拿到另一个重构中继续使用。大部分时候，一个范例只为一项重构而设计，这么做是为了让每一项重构手法自给自足（self-contained），因为这份重构名录的首要目的还是作为参考工具。

这些例子不会告诉你「如何设计一个 "employee" 对象或一个 "order" 对象」。这些例子的存在纯粹只是为了说明重构，除此之外别无用途。例如你会发现，我在这些例子中用 `double` 数据来表示货币金额。我之所以这样做，只是为了让例子简单一些，因为「以什么形式表示金额」对于重构自身并不重要。在真正的商用软件中，我强烈建议你不要以 `double` 表现金额。如果真要表示货币金额，我会使用 ***Quantity*** 模式 [Fowler, AP]。

撰写本书之际，商业开发中使用得最多的是 Java 1.1，所以我的大多数例子也以 Java 1.1 写就，这从我对群集（collections）的使用就可以明显看出来。本书即将完成之时，Java 2 已经正式发布。但我不觉得有必要修改所有这些例子，因为对重构来说，群集（collections）也是次要的。但是有些重构手法，例如 *Encapsulate Collection*（208），在 Java 1.2 中有所不同。这时候我会同时解释 Java 2 和 Java 1.1。

修改后的代码可能被埋没在未修改的代码中，难以一眼看出，所以我使用粗体（**blodface code**）突显修改过的代码。但我并没有对所有修改过的代码都使用粗体字，因为一旦修改过的代码太多，全都粗体反而不能突显重点。

5.2 寻找引用点（Finding References）

很多重构都要求你找到对于某个函数（methods）、某个值域（fields）或某个 class 的所有引用点（指涉点）。做这件事的时候，记得寻求计算机的帮助。有了计算机的帮助，你可以减少「遗漏某个引用点」的几率，而且通常比人工查找更快。

大多数语言都把计算机代码当作文本文件来处理，所以最好的帮手就是一个适当的文本查找工具。许多编程环境都允许你在一个文件或者一组文件中进行文本查找，你的查找目标的访问控制（access control）会告诉你需要查找的文件范围。

不要盲目地「查找-替换」。你应该检查每一个引用点，确定它的确指向你想要替换的东西。或许你很擅长运用查找手法，但我总是用心去检查，以确保替换时不出错。要知道，你可以在不同的 classes 中使用相同函数名称，也可以在同一个 class 中使用名称相同但签名（signature）不同的函数，所以「替换」出错机会是很高的。

在强型别（strongly typed）语言中，你可以让编译器帮助你捕捉漏网之鱼。你往往可以直接删除旧部分，让编译器帮你找出因此而被悬挂起来（dangling）的引用点。这样做的好处是：编译器会找到所有被悬挂的引用点。但是这种技巧也存在问题。

首先，如果被删除的部分在继承体系（hierarchy）中声明不止一次，那么编译器也会被迷惑。尤其当你处理一个被覆写（overridden）多次的函数时，情况更是如此。所以如果你在一个继承体系中工作，请先利用文本查找工具，检查是否有其他 class 声明了你正在处理的那个函数。

第二个问题是：编译器可能太慢，从而使你的工作失去性能。如果真是这样，请先使用文本查找工具，最起码编译器可以复查你的工作。只有当你想移除某个部分时，

才请你这样做。常常你会想先观察这一部分的所有运用情况，然后才决定下一步。这种情况下你必须使用文本查找法（而不是倚赖编译器）。

第三个问题是：编译器无法找到通过反射机制（reflection）而得到的引用点。这也是我们应该小心使用反射的原因之一。如果系统中使用了反射，你就必须以文本查找找出你想找的东西，测试份量也因此加重。有些时候我会建议你只编译，不测试，因为编译器通常会捕捉到可能的错误。如果使用反射（reflection），所有这些便利都没有了，你必须为许多编译搭配测试。

某些 Java 开发环境（特别值得一提的是 IBM 的 VisualAge）承受了 Smalltalk 浏览器的影响。在这些开发环境中你应该使用菜单选项（menu options）来查找引用点，而不是使用文本查找工具。因为这些开发环境并不以文本文件保存代码，而是使用一个内置数据库。只要习惯了这些菜单选项，你会发现它们往往比难用的文本查找工具出色得多。

5.3 这些重构准则有多成熟？

任何技术作家都会面对这样一个问题：该在何时发表自己的想法？发表愈早，人们愈快能够运用新想法、新观念。但只要是人，总是不断在学习。如果过早发表半生不熟的想法，这些思想可能并不完善，甚至可能给那些尝试采用它们的人带来麻烦。

重构的基本技巧——小步前进、频繁测试——已经得到多年的实践检验，特别是在 Smalltalk 社群中。所以，我敢保证，重构的这些基础思想是非常可靠的。

本书中的重构准则是我自己使用重构的笔记。是的，我全都用过它们。但是「使用某个重构手法」和「将它浓缩成可在这里给出之作法步骤」是有区别的。特别是在一些十分特殊的情况下，偶尔你会看见一些问题突然涌现。我并没有让很多人进行我所写下的这些技术步骤以图发现这一类问题。所以，使用重构的时候，请随时知道自己在做什么。记住，就像看着食谱做菜一样，你必须让这些重构准则适应你自己的情况。如果你遇上一个有趣的问题，请以电子邮件告诉我，我会试着把你的情况告诉其他人。

关于这些重构手法，另一个需要记住的就是：我是在「单进程」（single-process）软件这一大前提下考虑并介绍它们的。我很希望看到有人介绍用于并发式（concurrent）和分布式（distributed）程序设计的重构技术。这样的重构将是完全不同的。举个例子，在单进程软件中，你永远不必操心多么频繁地调用某个函数，

因为函数的调用成本很低，但在分布式软件中，函数的往返必须被减至最低限度。在这些特殊编程领域中有着完全不同的重构技术，这已超越本书主题。

许多重构手法，例如 *Replace Type Code with State/Strategy*（227）和 *Form Template Method*（345），都涉及一向系统引入模式（patterns）。正如 GoF（Gang of Four，四巨头）的经典著作中所说，「设计模式……为你的重构行为提供了目标」。模式和重构之间有着一种与生俱来的关系。模式是你希望到达的目标，重构则是到达之路。本书并没有提供「助你完成所有知名模式」的重构手法，甚至连 GoF 的 23 个知名模式 [Gang of Four] 都没有能够全部覆盖。这也从某个侧面反映出这份名录的不完整。我希望有一天这个缺陷能够被填补。

运用重构的时候，请记住：它们仅仅是一个起点。毋庸置疑，你一定可以找出个中缺陷。我之所以选择现在发表它们，因为我相信，尽管它们还不完美，但的确有用。我相信它们能给你一个起点，然后你可以不断提高自己的重构能力——这正是它们带给我的。

随着你用过愈来愈多的重构手法，我希望，你也开始发展属于你自己的重构手法。但愿本书例子能够激发你的创造力，并给你一个起点，让你知道从何入手。我很清楚现实存在的重构，比我这里介绍的还要多得多。如果你真的提出了一些新的重构手法，请给我一封电子邮件。

6

重新组织你的函数

Composing Methods

我的重构手法中，很大一部分是对函数进行整理，使之更恰当地包装代码。几乎所有时刻，问题都源于 Long Methods（过长函数）。这很讨厌，因为它们往往包含太多信息，这些信息又被函数错综复杂的逻辑掩盖，不易鉴别。对付过长函数，一项重要的重构手法就是 *Extract Method*（110），它把一段代码从原先函数中提取出来，放进一个单独函数中。*Inline Method*（117）正好相反：将一个函数调用动作替换为该函数本体。如果在进行多次提炼之后，意识到提炼所得的某些函数并没有做任何实质事情，或如果需要回溯恢复原先函数，我就需要 *Inline Method*（117）。

Extract Method（110）最大的困难就是处理局部变量，而临时变量则是其中一个主要的困难源头。处理一个函数时，我喜欢运用 *Replace Temp with Query*（120）去掉所有可去掉的临时变量。如果很多地方使用了某个临时变量，我就会先运用 *Split Temporary Variable*（128）将它变得比较容易替换。

但有时候临时变量实在太混乱，难以替换。这时候我就需要使用 *Replace Method with Method Object*（135）。它让我可以分解哪怕最混乱的函数，代价则是引入一个新 class。

参数带来的问题比临时变量稍微少一些，前提是你不在函数内赋值给它们。如果你已经这样做了，就得使用 *Remove Assignments to Parameters*（131）。

函数分解完毕后，我就可以知道如何让它工作得更好。也许我还会发现算法可以改进，从而使代码更清晰。这时我就使用 *Substitute Algorithm*（139）引入更清晰的算法。

6.1 Extract Method

你有一段代码可以被组织在一起并独立出来。

将这段代码放进一个独立函数中，并让函数名称解释该函数的用途。

```
void printOwing(double amount) {
    printBanner();

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails (double amount) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

动机（Motivation）

Extract Method 是我最常用的重构手法之一。当我看见一个过长的函数或者一段需要注释才能让人理解用途的代码，我就会将这段代码放进一个独立函数中。

有数个原因造成我喜欢简短而有良好命名的函数。首先，如果每个函数的粒度都很小（finely grained），那么函数之间彼此复用的机会就更大；其次，这会使高层函数码读起来就像一系列注释；再者，如果函数都是细粒度，那么函数的覆写（override）也会更容易些。

的确，如果你习惯看大型函数，恐怕需要一段时间才能适应这种新风格。而且只有当你能给小型函数很好地命名时，它们才能真正起作用，所以你需要在函数名称下点功夫。人们有时会问我，一个函数多长才算合适？在我看来，长度不是问题，关

键在于函数名称和函数本体之间的语义距离（semantic distance）。如果提炼动作（extracting）可以强化代码的清晰度，那就去做，就算函数名称比提炼出来的代码还长也无所谓。

作法（Mechanics）

- 创造一个新函数，根据这个函数的意图来给它命名（以它「做什么」来命名，而不是以它「怎样做」命名）

  ⇨ 即使你想要提炼（extract）的代码非常简单，例如只是一条消息或一个函数调用，只要新函数的名称能够以更好方式昭示代码意图，你也应该提炼它。但如果你想不出一个更有意义的名称，就别动。

- 将提炼出的代码从源函数（source）拷贝到新建的目标函数（target）中。
- 仔细检查提炼出的代码，看看其中是否引用了「作用域（scope）限于源函数」的变量（包括局部变量和源函数参数）。
- 检查是否有「仅用于被提炼码」的临时变量（temporary variables）。如果有，在目标函数中将它们声明为临时变量。
- 检查被提炼码，看看是否有任何局部变量（local-scope variables）的值被它改变。如果一个临时变量值被修改了，看看是否可以将被提炼码处理为一个查询（query），并将结果赋值给相关变量。如果很难这样做，或如果被修改的变量不止一个，你就不能仅仅将这段代码原封不动地提炼出来。你可能需要先使用 *Split Temporary Variable*（128），然后再尝试提炼。也可以使用 *Replace Temp with Query*（120）将临时变量消灭掉（请看「范例」中的讨论）。
- 将被提炼码中需要读取的局部变量，当作参数传给目标函数。
- 处理完所有局部变量之后，进行编译。
- 在源函数中，将被提炼码替换为「对目标函数的调用」。

  ⇨ 如果你将任何临时变量移到目标函数中，请检查它们原本的声明式是否在被提炼码的外围。如果是，现在你可以删除这些声明式了。

- 编译，测试。

范例（Examples）：无局部变量（No Local Variables）

在最简单的情况下，*Extract Method*（110）易如反掌。请看下列函数：

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

我们可以轻松提炼出“打印banner”的代码。我只需要剪切、粘贴、再插入一个函数调用动作就行了：

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}

void printBanner() {
    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");
}
```

范例（Examples）：有局部变量（Using Local Variables）

果真这么简单，这个重构手法的困难点在哪里？是的，就在局部变量，包括传进源函数的参数和源函数所声明的临时变量。局部变量的作用域仅限于源函数，所以当我使用 *Extract Method*（110）时，必须花费额外功夫去处理这些变量。某些时候它们甚至可能妨碍我，使我根本无法进行这项重构。

局部变量最简单的情况是：被提炼码只是读取这些变量的值，并不修改它们。这种情况下我可以简单地将它们当作参数传给目标函数。所以如果我面对下列函数：

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

我就可以将「打印详细信息」这一部分提炼为「带一个参数的函数」：

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}

void printDetails (double outstanding) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

必要的话，你可以用这种手法处理多个局部变量。

如果局部变量是个对象，而被提炼码调用了会对该对象造成修改的函数，也可以如法炮制。你同样只需将这个对象作为参数传递给目标函数即可。只有在被提炼码真的对一个局部变量赋值的情况下，你才必须采取其他措施。

范例（Examples）：对局部变量再赋值（Reassigning）

如果被提炼码对局部变量赋值，问题就变得复杂了。这里我们只讨论临时变量的问题。如果你发现源函数的参数被赋值，应该马上使用 *Remove Assignments to Parameters*（131）。

被赋值的临时变量也分两种情况。较简单的情况是：这个变量只在被提炼码区段中使用。果真如此，你可以将这个临时变量的声明式移到被提炼码中，然后一起提炼出去。另一种情况是：被提炼码之外的代码也使用了这个变量。这又分为两种情况：如果这个变量在被提炼码之后未再被使用，你只需直接在目标函数中修改它就可以了；如果被提炼码之后的代码还使用了这个变量，你就需要让目标函数返回该变量改变后的值。我以下列代码说明这几种不同情况：

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

现在我把「计算」代码提炼出来：

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

double getOutstanding() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}
```

Enumeration 变量 e 只在被提炼码中用到，所以我可以将它整个搬到新函数中。double 变量 outstanding 在被提炼码内外都被用到，所以我必须让提炼出来的新函数返回它。编译测试完成后，我就把回传值改名，遵循我的一贯命名原则：

```
double getOutstanding() {
    Enumeration e = _orders.elements();
    double result = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result = each.getAmount();
    }
    return result;
}
```

本例中的 outstanding 变量只是很单纯地被初始化为一个明确初值，所以我可以只在新函数中对它初始化。如果代码还对这个变量做了其他处理，我就必须将它的值作为参数传给目标函数。对于这种变化，最初代码可能是这样：

```
void printOwing(double previousAmount) {
    Enumeration e = _orders.elements();
    double outstanding = previousAmount * 1.2;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    printDetails(outstanding);
}
```

提炼后的代码可能是这样：

```
void printOwing(double previousAmount) {
    double outstanding = previousAmount * 1.2;
    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

double getOutstanding(double initialValue) {
    double result = initialValue;
    Enumeration e = _orders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }
    return result;
}
```

编译并测试后，我再将变量 outstanding 的初始化过程整理一下：

```
void printOwing(double previousAmount) {
    printBanner();
    double outstanding = getOutstanding(previousAmount * 1.2);
    printDetails(outstanding);
}
```

这时候，你可能会问：『如果需要返回的变量不止一个，又该怎么办呢？』

你有数种选择。最好的选择通常是：挑选另一块代码来提炼。我比较喜欢让每个函数都只返回一个值，所以我会安排多个函数，用以返回多个值。如果你使用的语言支持「输出式参数」（output parameters），你可以使用它们带回多个回传值。但我还是尽可能选择单一返回值。

临时变量往往为数众多，甚至会使提炼工作举步维艰。这种情况下，我会尝试先运用 *Replace Temp with Query*（120）减少临时变量。如果即使这么做了提炼依旧困难重重，我就会动用 *Replace Method with Method Object*（135），这个重构手法不在乎代码中有多少临时变量，也不在乎你如何使用它们。

6.2 Inline Method

一个函数，其本体（method body）应该与其名称（method name）同样清楚易懂。

在函数调用点插入函数本体，然后移除该函数。

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

动机（Motivation）

本书经常以简短的函数表现动作意图，这样会使代码更清晰易读。但有时候你会遇到某些函数，其内部代码和函数名称同样清晰易读。也可能你重构了该函数，使得其内容和其名称变得同样清晰。果真如此，你就应该去掉这个函数，直接使用其中的代码。间接性可能带来帮助，但非必要的间接性总是让人不舒服。

另一种需要使用 *Inline Method*（117）的情况是：你手上有一群组织不甚合理的函数。你可以将它们都 *inline* 到一个大型函数中，再从中提炼出组织合理的小型函数。Kent Beck 发现，实施 *Replace Method with Method Object*（135）之前先这么做，往往可以获得不错的效果。你可以把你所要的函数（有着你要的行为）的所有调用对象的函数内容都 *inline* 到 method object（函数对象）中。比起既要移动一个函数，又要移动它所调用的其他所有函数，「将大型函数作为单一整体来移动」会比较简单。

如果别人使用了太多间接层，使得系统中的所有函数都似乎只是对另一个函数的简单委托（delegation），造成我在这些委托动作之间晕头转向，那么我通常都会使用 *Inline Method*（117）。当然，间接层有其价值，但不是所有间接层都有价值。试着使用 *inlining*，我可以找出那些有用的间接层，同时将那些无用的间接层去除。

作法（Mechanics）

- 检查函数，确定它不具多态性（is not polymorphic）。
  - ⇨ 如果subclass继承了这个函数，就不要将此函数inline化，因为subclass无法覆写（override）一个根本不存在的函数。
- 找出这个函数的所有被调用点。
- 将这个函数的所有被调用点都替换为函数本体（代码）。
- 编译，测试。
- 删除该函数的定义。

被我这样一写，*Inline Method*（117）似乎很简单。但情况往往并非如此。对于递归调用、多返回点、*inlining* 至另一个对象中而该对象并无提供访问函数（accessors）……，每一种情况我都可以写上好几页。我之所以不写这些特殊情况，原因很简单：如果你遇到了这样的复杂情况，那么就不应该使用这个重构手法。

6.3 Inline Temp

你有一个临时变量，只被一个简单表达式赋值一次，而它妨碍了其他重构手法。

将所有对该变量的引用动作，替换为对它赋值的那个表达式自身。

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```

动机（Motivation）

Inline Temp（119）多半是作为 *Replace Temp with Query*（120）的一部分来使用，所以真正的动机出现在后者那儿。惟一单独使用 *Inline Temp*（119）的情况是：你发现某个临时变量被赋予某个函数调用的返回值。一般来说，这样的临时变量不会有任何危害，你可以放心地把它留在那儿。但如果这个临时变量妨碍了其他的重构手法 — 例如 *Extract Method*（110），你就应该将它 inline 化。

作法（Mechanics）

- 如果这个临时变量并未被声明为 `final`，那就将它声明为 `final`，然后编译。

  ⇨ 这可以检查该临时变量是否真的只被赋值一次。

- 找到该临时变量的所有引用点，将它们替换为「为临时变量赋值」之语句中的等号右侧表达式。
- 每次修改后，编译并测试。
- 修改完所有引用点之后，删除该临时变量的声明式和赋值语句。
- 编译，测试。

6.4 Replace Temp with Query

你的程序以一个临时变量（temp）保存某一表达式的运算结果。

将这个表达式提炼到一个独立函数（译注：所谓查询式，query）中。将这个临时变量的所有「被引用点」替换为「对新函数的调用」。新函数可被其他函数使用。

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
   return basePrice * 0.95;
else
   return basePrice * 0.98;
```

```
if (basePrice() > 1000)
   return basePrice() * 0.95;
else
   return basePrice() * 0.98;
...
double basePrice() {
   return _quantity * _itemPrice;
}
```

动机（Motivation）

临时变量的问题在于：它们是暂时的，而且只能在所属函数内使用。由于临时变量只有在所属函数内才可见，所以它们会驱使你写出更长的函数，因为只有这样你才能访问到想要访问的临时变量。如果把临时变量替换为一个查询式（query method），那么同一个 class 中的所有函数都将可以获得这份信息。这将带给你极大帮助，使你能够为这个 class 编写更清晰的代码。

Replace Temp with Query（120）往往是你运用 *Extract Method*（110）之前必不可少的一个步骤。局部变量会使代码难以被提炼，所以你应该尽可能把它们替换为查询式。

这个重构手法较为直率的情况就是：临时变量只被赋值一次，或者赋值给临时变量的表达式不受其他条件影响。其他情况比较棘手，但也有可能发生。你可能需要先运用 *Split Temporary Variable*（128）或 *Separate Query from Modifier*（279）使情况变得简单一些，然后再替换临时变量。如果你想替换的临时变量是用来收集结果

的（例如循环中的累加值），你就需要将某些程序逻辑（例如循环）拷贝到查询式（query method）去。

作法（Mechanics）

首先是简单情况：

- 找出只被赋值一次的临时变量。
  - ⇨ 如果某个临时变量被赋值超过一次，考虑使用 *Split Temporary Variable*（128）将它分割成多个变量。
- 将该临时变量声明为 final。
- 编译。
  - ⇨ 这可确保该临时变量的确只被赋值一次。
- 将「对该临时变量赋值」之语句的等号右侧部分提炼到一个独立函数中。
  - ⇨ 首先将函数声明为 private。日后你可能会发现有更多 class 需要使用它，彼时你可轻易放松对它的保护。
  - ⇨ 确保提炼出来的函数无任何连带影响（副作用），也就是说该函数并不修改任何对象内容。如果它有连带影响，就对它进行 *Separate Query from Modifier*（279）。
- 编译，测试。
- 在该临时变量身上实施 *Inline Temp*（119）。

我们常常使用临时变量保存循环中的累加信息。在这种情况下，整个循环都可以被提炼为一个独立函数，这也使原本的函数可以少掉几行扰人的循环码。有时候，你可能会用单一循环累加好几个值，就像本书 p.26 的例子那样。这种情况下你应该针对每个累加值重复一遍循环，这样就可以将所有临时变量都替换为查询式（query）。当然，循环应该很简单，复制这些代码时才不会带来危险。

运用此手法，你可能会担心性能问题。和其他性能问题一样，我们现在不管它，因为它十有八九根本不会造成任何影响。如果性能真的出了问题，你也可以在优化时期解决它。如果代码组织良好，那么你往往能够发现更有效的优化方案；如果你没有进行重构，好的优化方案就可能与你失之交臂。如果性能实在太糟糕，要把临时变量放回去也是很容易的。

范例（Example）

首先，我从一个简单函数开始：

```
double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

我希望将两个临时变量都替换掉，当然，每次一个。

尽管这里的代码十分清楚，我还是先把临时变量声明为 final，检查它们是否的确只被赋值一次：

```
double getPrice() {
    final int basePrice = _quantity * _itemPrice;
    final double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

这么一来，如果有任何问题，编译器就会警告我。之所以先做这件事，因为如果临时变量不只被赋值一次，我就不该进行这项重构。接下来我开始替换临时变量，每次一个。首先我把赋值（assignment）动作的右侧表达式提炼出来：

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
private int basePrice() {
    return _quantity * _itemPrice;
}
```

编译并测试，然后开始使用 *Inline Temp*（119）。首先把临时变量 basePrice 的第一个引用点替换掉：

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;
    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

编译、测试、下一个（听起来像在指挥人们跳乡村舞蹈一样）。由于「下一个」已经是 basePrice 的最后一个引用点，所以我把 basePrice 临时变量的声明式一并摘除：

```
double getPrice() {
    final double discountFactor;
    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice() * discountFactor;
}
```

搞定 basePrice 之后，我再以类似办法提炼出一个 discountFactor()：

```
double getPrice() {
    final double discountFactor = discountFactor();
    return basePrice() * discountFactor;
}
private double discountFactor() {
    if (basePrice() > 1000) return 0.95;
    else return 0.98;
}
```

你看，如果我没有把临时变量 basePrice 替换为一个查询式，将多么难以提炼 discountFactor()！

最终，getPrice()变成了这样：

```
double getPrice() {
    return basePrice() * discountFactor();
}
```

6.5 Introduce Explaining Variable

你有一个复杂的表达式。

将该复杂表达式（或其中一部分）的结果放进一个临时变量，以此变量名称来解释表达式用途。

```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
     (browser.toUpperCase().indexOf("IE") > -1) &&
      wasInitialized() && resize > 0 )
{
   // do something
}
```

```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized = resize > 0;

if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
   // do something
}
```

动机（Motivation）

表达式有可能非常复杂而难以阅读。这种情况下，临时变量可以帮助你将表达式分解为比较容易管理的形式。

在条件逻辑（conditional logic）中，*Introduce Explaining Variable*（124）特别有价值：你可以用这项重构将每个条件子句提炼出来，以一个良好命名的临时变量来解释对应条件子句的意义。使用这项重构的另一种情况是，在较长算法中，可以运用临时变量来解释每一步运算的意义。

Introduce Explaining Variable（124）是一个很常见的重构手法，但我得承认，我并不常用它。我几乎总是尽量使用 *Extract Method*（110）来解释一段代码的意义。毕竟临时变量只在它所处的那个函数中才有意义，局限性较大，函数则可以在对象的整个生命中都有用，并且可被其他对象使用。但有时候，当局部变量使 *Extract Method*（110）难以进行时，我就使用 *Introduce Explaining Variable*（124）。

作法（Mechanics）

- 声明一个 final 临时变量，将待分解之复杂表达式中的一部分动作的运算结果赋值给它。
- 将表达式中的「运算结果」这一部分，替换为上述临时变量。
  - ⇨ 如果被替换的这一部分在代码中重复出现，你可以每次一个，逐一替换。
- 编译，测试。
- 重复上述过程，处理表达式的其他部分。

范例（Examples）

我们从一个简单计算开始：

```
double price() {
    // price is base price - quantity discount + shipping
    return _quantity * _itemPrice
         Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
         Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

这段代码还算简单，不过我可以让它变得更容易理解。首先我发现，底价（base price）等于数量（quantity）乘以单价（item price）。于是我把这一部分计算的结果放进一个临时变量中：

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
         Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
         Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

稍后也用上了「数量乘以单价」运算结果，所以我同样将它替换为 basePrice 临时变量：

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
         Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
         Math.min(basePrice * 0.1, 100.0);
}
```

然后，我将批发折扣（quantity discount）的计算提炼出来，将结果赋予临时变量quantityDiscount：

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    final double quantityDiscount = Math.max(0, _quantity - 500) *
                                    _itemPrice * 0.05;
    return basePrice - quantityDiscount +
        Math.min(basePrice * 0.1, 100.0);
}
```

最后，我再把运费（shipping）计算提炼出来，将运算结果赋予临时变量shipping。同时我还可以删掉代码中的注释，因为现在代码已经可以完美表达自己的意义了：

```
double price() {
    final double basePrice = _quantity * _itemPrice;
    final double quantityDiscount = Math.max(0, _quantity - 500) *
                                    _itemPrice * 0.05;
    final double shipping = Math.min(basePrice * 0.1, 100.0);
    return basePrice - quantityDiscount + shipping;
}
```

运用 Extract Method 处理上述范例

面对上述代码，我通常不会以临时变量来解释其动作意图，我更喜欢使用 *Extract Method*（110）。让我们回到起点：

```
double price() {
    // price is base price - quantity discount + shipping
    return _quantity * _itemPrice
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

这一次我把底价计算提炼到一个独立函数中：

```
double price() {
    // price is base price - quantity discount + shipping
    return basePrice() -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(basePrice() * 0.1, 100.0);
}
private double basePrice() {
    return _quantity * _itemPrice;
}
```

我继续我的提炼，每次提炼出一个新函数。最后得到下列代码：

```
double price() {
    return basePrice() - quantityDiscount() + shipping();
}
private double quantityDiscount() {
    return Math.max(0, _quantity - 500) * _itemPrice * 0.05;
}
private double shipping() {
    return Math.min(basePrice() * 0.1, 100.0);
}
private double basePrice() {
    return _quantity * _itemPrice;
}
```

我比较喜欢使用 *Extract Method*（110），因为同一对象中的任何部分，都可以根据自己的需要去取用这些提炼出来的函数。一开始我会把这些新函数声明为 private；如果其他对象也需要它们，我可以轻易释放这些函数的访问限制。我还发现，*Extract Method*（110）的工作量通常并不比 *Introduce Explaining Variable*（124）来得大。

那么，应该在什么时候使用 *Introduce Explaining Variable*（124）呢？答案是：在 *Extract Method*（110）需要花费更大工作量时。如果我要处理的是一个拥有大量局部变量的算法，那么使用 *Extract Method*（110）绝非易事。这种情况下我会使用 *Introduce Explaining Variable*（124）帮助我理清代码，然后再考虑下一步该怎么办。搞清楚代码逻辑之后，我总是可以运用 *Replace Temp with Query*（120）把被我引入的那些解释性临时变量去掉。况且，如果我最终使用 *Replace Method with Method Object*（135），那么被我引入的那些解释性临时变量也有其价值。

6.6 Split Temporary Variable（剖解临时变量）

你的程序有某个临时变量被赋值超过一次，它既不是循环变量，也不是一个集用临时变量（collecting temporary variable）。

针对每次赋值，创造一个独立的、对应的临时变量。

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

⇩

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

动机（Motivation）

临时变量有各种不同用途，其中某些用途会很自然地导致临时变量被多次赋值。「循环变量」和「集用临时变量」就是两个典型例子：循环变量（loop variable）[Beck]会随循环的每次运行而改变（例如 `for (int i=0; i<10; i++)`语句中的 `i`）；集用临时变量（collecting temporary variable）[Beck] 负责将「通过整个函数的运算」而构成的某个值收集起来。

除了这两种情况，还有很多临时变量用于保存一段冗长代码的运算结果，以便稍后使用。这种临时变量应该只被赋值一次。如果它们被赋值超过一次，就意味它们在函数中承担了一个以上的责任。如果临时变量承担多个责任，它就应该被替换（剖解）为多个临时变量，每个变量只承担一个责任。同一个临时变量承担两件不同的事情，会令代码阅读者糊涂。

作法（Mechanics）

- 在「待剖解」之临时变量的声明式及其第一次被赋值处，修改其名称。
  - ⇨ 如果稍后之赋值语句是「`i = i` 某表达式」形式，就意味这是个集用临时变量，那么就不要剖解它。集用临时变量的作用通常是累加、字符串接合、写入 stream 或者向群集（collection）添加元素。

- 将新的临时变量声明为 final。
- 以该临时变量之第二次赋值动作为界，修改此前对该临时变量的所有引用点，让它们引用新的临时变量。
- 在第二次赋值处，重新声明原先那个临时变量。
- 编译，测试。
- 逐次重复上述过程。每次都在声明处对临时变量易名，并修改下次赋值之前的引用点。

范例（Examples）

下面范例中我要计算一个苏格兰布丁（haggis）运动的距离。在起点处，静止的苏格兰布丁会受到一个初始力的作用而开始运动。一段时间后，第二个力作用于布丁，让它再次加速。根据牛顿第二定律，我可以这样计算布丁运动的距离：

```
double getDistanceTravelled (int time) {
    double result;
    double acc = _primaryForce / _mass;          // 译注：第一次赋值处
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * acc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;
    if (secondaryTime > 0) {
        double primaryVel = acc * _delay;  // 译注：以下是第二次赋值处
        acc = (_primaryForce + _secondaryForce) / _mass;
        result += primaryVel * secondaryTime - 0.5 * acc * secondaryTime
                * secondaryTime;
    }
    return result;
}
```

真是个绝佳的丑陋小东西。注意观察此例中的 acc 变量如何被赋值两次。acc 变量有两个责任：第一是保存第一个力造成的初始加速度；第二是保存两个力共同造成的加速度。这就是我想要剖解的东西。

首先，我在函数开始处修改这个临时变量的名称，并将新的临时变量声明为 final。接下来我把第二次赋值之前对 acc 变量的所有引用点，全部改用新的临时变量。最后，我在第二次赋值处重新声明 acc 变量：

```
double getDistanceTravelled (int time) {
    double result;
    final double primaryAcc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * primaryAcc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;
```

```
    if (secondaryTime > 0) {
        double primaryVel = primaryAcc * _delay;
        double acc = (_primaryForce + _secondaryForce) / _mass;
        result += primaryVel * secondaryTime + 0.5 * acc * secondaryTime
                * secondaryTime;
    }
    return result;
}
```

新的临时变量的名称指出，它只承担原先 acc 变量的第一个责任。我将它声明为 final，确保它只被赋值一次。然后，我在原先 acc 变量第二次被赋值处重新声明 acc。现在，重新编译并测试，一切都应该没有问题。

然后，我继续处理 acc 临时变量的第二次赋值。这次我把原先的临时变量完全删掉，代之以一个新的临时变量。新变量的名称指出，它只承担原先 acc 变量的第二个责任：

```
double getDistanceTravelled (int time) {
    double result;
    final double primaryAcc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * primaryAcc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;
    if (secondaryTime > 0) {
        double primaryVel = primaryAcc * _delay;
        final double secondaryAcc = (_primaryForce + _secondaryForce)
        / _mass;
        result += primaryVel * secondaryTime + 0.5 *
        secondaryAcc * secondaryTime * secondaryTime;
    }
    return result;
}
```

现在，这段代码肯定可以让你想起更多其他重构手法、尽情享受吧。（我敢保证，这比吃苏格兰布丁强多了 —— 你知道他们都在里面放了些什么东西吗？[4]）

[4] 译注：苏格兰布丁（haggis）是一种苏格兰菜，把羊心等内脏装在羊胃里煮成。由于它被羊胃包成一个球体，因此可以像球一样踢来踢去，这就是本例的由来。「把羊心装在羊胃里煮成…」，呃，有些人难免对这道菜恶心，Martin Fowler 想必是其中之一。

6.7 Remove Assignments to Parameters

你的代码对一个参数进行赋值动作。

以一个临时变量取代该参数的位置。

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
```

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
```

动机（Motivation）

首先，我要确定大家都清楚「对参数赋值」这个说法的意思。如果你把一个名为 foo 的对象作为参数传给某个函数，那么「对参数赋值」意味改变 foo，使它引用（参考、指涉、指向）另一个对象。如果你在「被传入对象」身上进行什么操作，那没问题，我也总是这样干。我只针对「foo 被改而指向（引用）完全不同的另一个对象」这种情况来讨论：

```
void aMethod(Object foo) {
    foo.modifyInSomeWay();  // that's OK
    foo = anotherObject;    // trouble and despair will follow you
```

我之所以不喜欢这样的作法，因为它降低了代码的清晰度，而且混淆了 *pass by value*（传值）和 *pass by reference*（传址）这两种参数传递方式。Java 只采用 *pass by value* 传递方式（稍后讨论），我们的讨论也正是基于这一点。

在 *pass by value* 情况下，对参数的任何修改，都不会对调用端造成任何影响。那些用过 *pass by reference* 的人可能会在这一点上犯糊涂。

另一个让人糊涂的地方是函数本体内。如果你只以参数表示「被传递进来的东西」，那么代码会清晰得多，因为这种用法在所有语言中都表现出相同语义。

在 Java 中，不要对参数赋值：如果你看到手上的代码已经这样做了，请使用 ***Remove Assignments to Parameters***（131）。

当然，面对那些使用「输出式参数」（output parameters）的语言，你不必遵循这条规则。不过在那些语言中我会尽量少用输出式参数。

作法（Mechanics）

- 建立一个临时变量，把待处理的参数值赋予它。
- 以「对参数的赋值动作」为界，将其后所有对此参数的引用点，全部替换为「对此临时变量的引用动作」。
- 修改赋值语句，使其改为对新建之临时变量赋值。
- 编译，测试。

  ⇨ 如果代码的语义是 *pass by reference*，请在调用端检查调用后是否还使用了这个参数。也要检查有多少个 *pass by reference* 参数「被赋值后又被使用」。请尽量只以 return 方式返回一个值。如果需要返回的值不只一个，看看可否把需返回的大堆数据变成单一对象，或干脆为每个返回值设计对应的一个独立函数。

范例（Examples）

我从下列这段简单代码开始：

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    if (quantity > 100) inputVal -= 1;
    if (yearToDate > 10000) inputVal -= 4;
    return inputVal;
}
```

以临时变量取代对参数的赋值动作，得到下列代码：

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    if (quantity > 100) result -= 1;
    if (yearToDate > 10000) result -= 4;
    return result;
}
```

还可以为参数加上关键词 final，从而强制它遵循「不对参数赋值」这一惯例：

```
int discount (final int inputVal, final int quantity,
              final int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    if (quantity > 100) result -= 1;
    if (yearToDate > 10000) result -= 4;
    return result;
}
```

不过我得承认，我并不经常使用 final 来修饰参数，因为我发现，对于提高短函数的清晰度，这个办法并无太大帮助。我通常会在较长的函数中使用它，让它帮助我检查参数是否被做了修改。

Java 的 *pass by Value*（传值）

Java 使用 "*pass by value*"「函数调用」方式，这常常造成许多人迷惑。在所有地点，Java 都严格采用 *pass by value*，所以下列程序：

```
class Param {
    public static void main(String[] args) {
        int x = 5;
        triple(x);
        System.out.println ("x after triple: " + x);
    }
    private static void triple(int arg) {
        arg = arg * 3;
        System.out.println ("arg in triple: " + arg);
    }
}
```

会产生这样的输出：

```
arg in triple: 15
x after triple: 5
```

这段代码还不至于让人糊涂。但如果参数中传递的是对象，就可能把人弄迷糊了。如果我在程序中以 Date 对象表示日期，那么下列程序：

```
class Param {
    public static void main(String[] args) {
        Date d1 = new Date ("1 Apr 98");
        nextDateUpdate(d1);
        System.out.println ("d1 after nextDay: " + d1);

        Date d2 = new Date ("1 Apr 98");
        nextDateReplace(d2);
        System.out.println ("d2 after nextDay: " + d2);
    }
    private static void nextDateUpdate (Date arg) {
        arg.setDate(arg.getDate() + 1);
        System.out.println ("arg in nextDay: " + arg);
    }
    private static void nextDateReplace (Date arg) {
        arg = new Date (arg.getYear(),arg.getMonth(),arg.getDate()+1);
        System.out.println ("arg in nextDay: " + arg);
    }
}
```

产生的输出是：

```
arg in nextDay: Thu Apr 02 00:00:00 EST 1998
d1 after nextDay: Thu Apr 02 00:00:00 EST 1998
arg in nextDay: Thu Apr 02 00:00:00 EST 1998
d2 after nextDay: Wed Apr 01 00:00:00 EST 1998
```

从本质上说，object reference 是按值传递的（*pass by value*）。因此我可以修改参数对象的内部状态，但对参数对象重新赋值，没有意义。

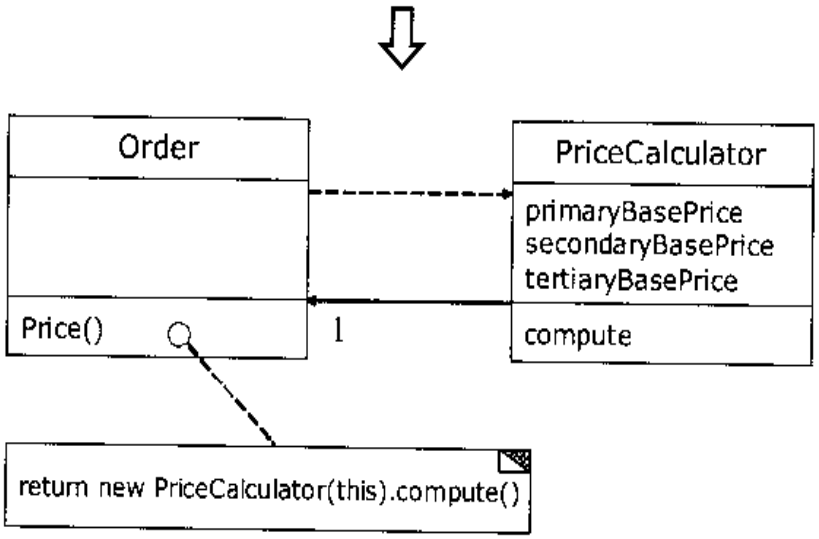
Java 1.1 及其后版本，允许你将参数标示为 `final`，从而避免函数中对参数赋值。即使某个参数被标示为 final，你仍然可以修改它所指向的对象。我总是把参数视为 final，但是我得承认，我很少在参数列（parameter list）中这样标示它们。

6.8 Replace Method with Method Object

你有一个大型函数，其中对局部变量的使用，使你无法采用 *Extract Method*（110）。

将这个函数放进一个单独对象中，如此一来局部变量就成了对象内的值域（field）。**然后你可以在同一个对象中将这个大型函数分解为数个小型函数。**

```
class Order...
  double price() {
      double primaryBasePrice;
      double secondaryBasePrice;
      double tertiaryBasePrice;
      // long computation;
      ...
  }
```



动机（Motivation）

我在本书中不断向读者强调小型函数的优美动人。只要将相对独立的代码从大型函数中提炼出来，就可以大大提高代码的可读性。

但是，局部变量的存在会增加函数分解难度。如果一个函数之中局部变量泛滥成灾，那么想分解这个函数是非常困难的。*Replace Temp with Query*（120）可以助你减轻这一负担，但有时候你会发现根本无法拆解一个需要拆解的函数。这种情况下，你应该把手深深地伸入你的工具箱（好洞沉瓮底呢），祭出函数对象（method object）[Beck] 这件法宝。

Replace Method with Method Object（135）会将所有局部变量都变成函数对象（method object）的值域（field）。然后你就可以对这个新对象使用 *Extract Method*（110）创造出新函数，从而将原本的大型函数拆解变短。

作法（Mechanics）

我厚着脸皮从 Kent Beck [Beck] 那里偷来了下列作法：

- 建立一个新 class，根据「待被处理之函数」的用途，为这个 class 命名。
- 在新 class 中建立一个 final 值域，用以保存原先大型函数所驻对象。我们将这个值域称为「源对象」。同时，针对原（旧）函数的每个临时变量和每个参数，在新 class 中建立一个个对应的值域保存之。
- 在新 class 中建立一个构造函数（constructor），接收源对象及原函数的所有参数作为参数。
- 在新 class 中建立一个 `compute()`函数。
- 将原（旧）函数的代码拷贝到 `compute()`函数中。如果需要调用源对象的任何函数，请以「源对象」值域调用。
- 编译。
- 将旧函数的函数本体替换为这样一条语句：「创建上述新 class 的一个新对象，而后调用其中的 `compute()`函数」。

现在进行到很有趣的部分了。由于所有局部变量现在都成了值域，所以你可以任意分解这个大型函数，不必传递任何参数。

范例（Examples）

如果要给这一重构手法找个合适例子，需要很长的篇幅。所以我以一个不需要长篇幅（那也就是说可能不十分完美）的例子展示这项重构。请不要问这个函数的逻辑是什么，这完全是我且战且走的产品。

```
class Account...
  int gamma (int inputVal, int quantity, int yearToDate) {
      int importantValue1 = (inputVal * quantity) + delta();
      int importantValue2 = (inputVal * yearToDate) + 100;
      if ((yearToDate - importantValue1) > 100)
          importantValue2 -= 20;
      int importantValue3 = importantValue2 * 7;
      // and so on.
      return importantValue3 - 2 * importantValue1;
  }
```

为了把这个函数变成一个函数对象（method object），我首先需要声明一个新 class。在此新 class 中我应该提供一个 final 值域用以保存原先对象（源对象）；对于函数的每一个参数和每一个临时变量，也以一个个值域逐一保存。

```
class Gamma...
  private final Account _account;
  private int inputVal;
  private int quantity;
  private int yearToDate;
  private int importantValue1;
  private int importantValue2;
  private int importantValue3;
```

按惯例，我通常会以下划线作为值域名称的前缀。但为了保持小步前进，我暂时先保留这些值域的原名。

接下来，加入一个构造函数：

```
Gamma (Account source, int inputValArg, int quantityArg,
       int yearToDateArg) {
   _account = source;
   inputVal = inputValArg;
   quantity = quantityArg;
   yearToDate = yearToDateArg;
}
```

现在可以把原本的函数搬到 compute() 了。函数中任何调用 Account class 的地方，我都必须改而使用 _account 值域：

```
int compute () {
   importantValue1 = (inputVal * quantity) + _account.delta();
   importantValue2 = (inputVal * yearToDate) + 100;
   if ((yearToDate - importantValue1) > 100)
      importantValue2 -= 20;
   int importantValue3 = importantValue2 * 7;
   // and so on.
   return importantValue3 - 2 * importantValue1;
}
```

然后，我修改旧函数，让它将它的工作转发（委托，delegate）给刚完成的这个函数对象（method object）：

```
int gamma (int inputVal, int quantity, int yearToDate) {
   return new Gamma(this, inputVal, quantity, yearToDate).compute();
}
```

这就是本项重构的基本原则。它带来的好处是：现在我可以轻松地对 compute() 函数采取 *Extract Method*（110），不必担心引数（argument）传递。

```
int compute () {
    importantValue1 = (inputVal * quantity) + _account.delta();
    importantValue2 = (inputVal * yearToDate) + 100;
    importantThing();
    int importantValue3 = importantValue2 * 7;
    // and so on.
    return importantValue3 - 2 * importantValue1;
}
void importantThing() {
    if ((yearToDate - importantValue1) > 100)
    importantValue2 -= 20;
}
```

6.9 Substitute Algorithm（替换你的算法）

你想要把某个算法替换为另一个更清晰的算法。

将函数本体（method body）替换为另一个算法。

```
String foundPerson(String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")) {
            return "Don";
        }
        if (people[i].equals ("John")) {
            return "John";
        }
        if (people[i].equals ("Kent")) {
            return "Kent";
        }
    }
    return "";
}
```

```
String foundPerson(String[] people){
    List candidates = Arrays.asList(new String[]
                                    {"Don", "John",   "Kent"});
    for (int i=0; i<people.length; i++)
        if (candidates.contains(people[i]))
            return people[i];
    return "";
}
```

动机（Motivation）

我没试过给猫剥皮，不过我听说这有好几种方法，我敢打赌其中某些方法会比另一些简单。算法也是如此。如果你发现做一件事可以有更清晰的方式，就应该以较清晰的方式取代复杂方式。「重构」可以把一些复杂东西分解为较简单的小块，但有时你就是必须壮士断腕，删掉整个算法，代之以较简单的算法。随着对问题有了更多理解，你往往会发现，在你的原先作法之外，有更简单的解决方案，此时你就需要改变原先的算法。如果你开始使用程序库，而其中提供的某些功能/特性与你自己的代码重复，那么你也需要改变原先的算法。

有时候你会想要修改原先的算法，让它去做一件与原先动作略有差异的事。这时候你也可以先把原先的算法替换为一个较易修改的算法，这样后续的修改会轻松许多。

使用这项重构手法之前，请先确定自己已经尽可能分解了原先函数。替换一个巨大而复杂的算法是非常困难的，只有先将它分解为较简单的小型函数，然后你才能很有把握地进行算法替换工作。

作法（Mechanics）

- 准备好你的另一个（替换用）算法，让它通过编译。
- 针对现有测试，执行上述的新算法。如果结果与原本结果相同，重构结束。
- 如果测试结果不同于原先，在测试和调试过程中，以旧算法为比较参照标准。
  - ⇨ 对于每个 test case（测试用例），分别以新旧两种算法执行，并观察两者结果是否相同。这可以帮助你看到哪一个 test case 出现麻烦，以及出现了怎样的麻烦。

7

在对象之间搬移特性

Moving Features Between Objects

在对象的设计过程中，「决定把责任放在哪儿」即使不是最重要的事，也是最重要的事之一。我使用对象技术已经十多年了，但还是不能一开始就保证做对。这曾经让我很烦恼，但现在我知道，在这种情况下，我可以运用重构（refactoring），改变自己原先的设计。

常常我可以只运用 *Move Method*（142）和 *Move Field*（146）简单地移动对象行为，就可以解决这些问题。如果这两个重构手法都需要用到，我会首先使用 *Move Field*（146），再使用 *Move Method*（142）。

class往往会因为承担过多责任而变得臃肿不堪。这种情况下，我会使用 *Extract Class*（149）将一部分责任分离出去。如果一个 class 变得太「不负责任」，我就会使用 *Inline Class*（154）将它融入另一个 class。如果一个 class 使用了另一个 class，运用 *Hide Delegate*（157）将这种关系隐藏起来通常是有帮助的。有时候隐藏 delegate class 会导致拥有者的接口经常变化，此时需要使用 *Remove Middle Man*（160）。

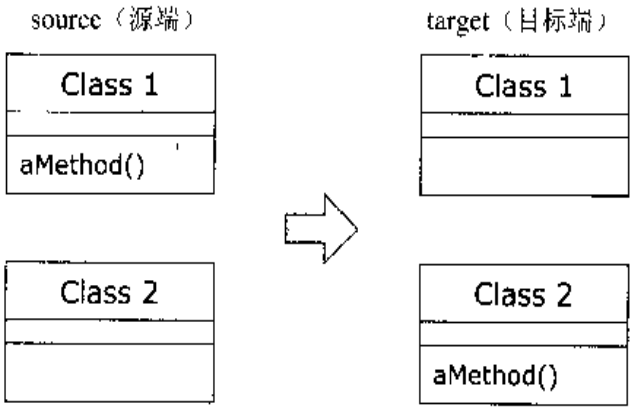
本章的最后两项重构 — *Introduce Foreign Method*（162）和 *Introduce Local Extension*（164）— 比较特殊。只有当我不能访问某个 class 的源码，却又想把其他责任移进这个不可修改的 class 时，我才会使用这两个重构手法。如果我想加入的只是一或两个函数，我会使用 *Introduce Foreign Method*（162）；如果不止一两个函数，我就使用 *Introduce Local Extension*（164）。

7.1 Move Method（搬移函数）

（译注：本节大量保留 class, method, source, target 字眼）

你的程序中，有个函数与其所驻 class 之外的另一个 class 进行更多交流：调用后者，或被后者调用。

在该函数最常引用（指涉）的 class 中建立一个有着类似行为的新函数。将旧函数变成一个单纯的委托函数（delegating method），或是将旧函数完全移除。



动机（Motivation）

「函数搬移」是重构理论的支柱。如果一个 class 有太多行为，或如果一个 class 与另一个 class 有太多合作而形成高度耦合（highly coupled），我就会搬移函数。通过这种手段，我可以使系统中的 classes 更简单，这些 classes 最终也将更干净利落地实现系统交付的任务。

常常我会浏览 class 的所有函数，从中寻找这样的函数：使用另一个对象的次数比使用自己所驻对象的次数还多。一旦我移动了一些值域，就该做这样的检查。一旦发现「有可能被我搬移」的函数，我就会观察调用它的那一端、它调用的那一端，以及继承体系中它的任何一个重定义函数。然后，我会根据「这个函数与哪个对象的交流比较多」，决定其移动路径。

这往往不是一个容易做出的决定。如果不能肯定是否应该移动一个函数，我就会继续观察其他函数。移动其他函数往往会让这项决定变得容易一些。有时候，即使你移动了其他函数，还是很难对眼下这个函数做出决定。其实这也没什么大不了的。如果真的很难做出决定，那么或许「移动这个函数与否」并不那么重要。所以，我会凭本能去做，反正以后总是可以修改的。

作法（Mechanics）

- □ 检查 source class 定义之 source method 所使用的一切特性（features），考虑它们是否也该被搬移。（译注：此处所谓特性泛指 class 定义的所有东西，包括值域和函数。）
  - ⇨ 如果某个特性只被你打算搬移的那个函数用到，你应该将它一并搬移。如果另有其他函数使用了这个特性，你可以考虑将使用该特性的所有函数全都一并搬移。有时候搬移一组函数比逐一搬移简单些。
- □ 检查 source class 的 subclass 和 superclass，看看是否有该函数的其他声明。
  - ⇨ 如果出现其他声明，你或许无法进行搬移，除非 target class 也同样表现出多态性（polymorphism）。
- □ 在 target class 中声明这个函数。
  - ⇨ 你可以为此函数选择一个新名称——对 target class 更有意义的名称。
- □ 将 source method 的代码拷贝到 target method 中。调整后者，使其能在新家中正常运行。
  - ⇨ 如果 target method 使用了 source 特性，你得决定如何从 target method 引用 source object。如果 target class 中没有相应的引用机制，就把 source object reference 当作参数，传给新建立的 target method。
  - ⇨ 如果 source method 包含异常处理式（exception handler），你得判断逻辑上应该由哪个 class 来处理这一异常。如果应该由 source class 来负责，就把异常处理式留在原地。
- □ 编译 target class。
- □ 决定如何从 source 正确引用 target object。
  - ⇨ 可能会有一个现成的值域或函数帮助你取得 target object。如果没有，就看能否轻松建立一个这样的函数。如果还是不行，你得在 source class 中新建一个新值域来保存 target object。这可能是一个永久性修改，但你也可以让它保持暂时的地位，因为后继的其他重构项目可能会把这个新建值域去掉。
- □ 修改 source method，使之成为一个 delegating method（纯委托函数）。
- □ 编译，测试。
- □ 决定「删除 source method」或将它当作一个 delegating method 保留下来。
  - ⇨ 如果你经常要在 source object 中引用 target method，那么将 source method 作为 delegating method 保留下来会比较简单。

- 如果你移除 source method，请将 source class 中对 source method 的所有引用动作，替换为「对 target method 的引用动作」。
  - ⇨ 你可以每修改一个引用点就编译并测试一次。也可以通过一次「查找/替换」改掉所有引用点，这通常简单一些。
- 编译，测试。

范例（Examples）

我用一个表示「帐户」的 account class 来说明这项重构：

```
class Account...
    double overdraftCharge() {          //译注：透支金计费，它和其他 class
        if (_type.isPremium()) {        //      的关系似乎比较密切。
            double result = 10;
            if (_daysOverdrawn > 7)
                result += (_daysOverdrawn - 7) * 0.85;
            return result;
        }
        else return _daysOverdrawn * 1.75;
    }
    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0) result += overdraftCharge();
        return result;
    }
    private AccountType _type;
    private int _daysOverdrawn;
```

假设有数种新帐户，每一种都有自己的「透支金计费规则」。所以我希望将 overdraftCharge()搬移到 AccountType class 去。

第一步要做的是：观察被 overdraftCharge()使用的每一特性（features），考虑是否值得将它们与 overdraftCharge()一起移动。此例之中我需要让 _daysOverdrawn 值域留在 Account class，因为其值会随不同种类的帐户而变化。然后，我将 overdraftCharge()函数码拷贝到 AccountType 中，并做相应调整。

```
class AccountType...
    double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7)
                result += (daysOverdrawn - 7) * 0.85;
            return result;
    }
    else return daysOverdrawn * 1.75;
}
```

在这个例子中，「调整」的意思是：(1) 对于「使用 AccountType 特性」的语句，去掉 _type；(2) 想办法得到依旧需要的 Account class 特性。当我需要使用 source

class 特性，我有四种选择：(1) 将这个特性也移到 target class；(2) 建立或使用一个从 target class 到 source 的引用（指涉）关系；(3) 将 source object 当作参数传给 target method；(4) 如果所需特性是个变量，将它当作参数传给 target method。

本例中我将 _daysOverdrawn 变量作为参数传给 target method（上述 (4)）。

调整 target method 使之通过编译，而后我就可以将 source method 的函数本体替换为一个简单的委托动作（delegation），然后编译并测试：

```
class Account...
    double overdraftCharge() {
    return _type.overdraftCharge(_daysOverdrawn);
}
```

我可以保留代码如今的样子，也可以删除 source method。如果决定删除，就得找出 source method 的所有调用者，并将这些调用重新定向，改调用 Account 的 bankCharge()：

```
class Account...
    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0)
            result += _type.overdraftCharge(_daysOverdrawn);
        return result;
    }
```

所有调用点都修改完毕后，我就可以删除 source method 在 Account 中的声明了。我可以在每次删除之后编译并测试，也可以一次性批量完成。如果被搬移的函数不是 private，我还需要检查其他 classes 是否使用了这个函数。在强型（strongly typed）语言中，删除 source method 声明式后，编译器会帮我发现任何遗漏。

此例之中被移函数只取用（指涉）一个值域，所以我只需将这个值域作为参数传给 target method 就行了。如果被移函数调用了 Account 中的另一个函数，我就不能这么简单地处理。这种情况下我必须将 source object 传递给 target method：

```
class AccountType...
    double overdraftCharge(Account account) {
        if (isPremium()) {
            double result = 10;
            if (account.getDaysOverdrawn() > 7)
                result += (account.getDaysOverdrawn() - 7) * 0.85;
            return result;
        }
        else return account.getDaysOverdrawn() * 1.75;
    }
```

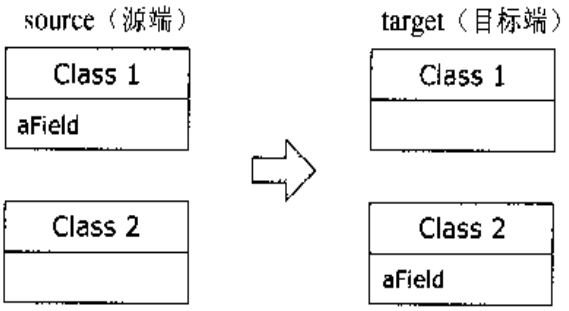
如果我需要 source class 的多个特性，那么我也会将 source object 传递给 target method。不过如果 target method 需要太多 source class 特性，就得进一步重构。通常这种情况下我会分解 target method，并将其中一部分移回 source class。

7.2 Move Field（搬移值域）

（译注：本节大量保留 class, object, field, source, target 等字眼）

你的程序中，某个 field（值域）被其所驻 class 之外的另一个 class 更多地用到。

在 target class 建立一个 new field，修改 source field 的所有用户，令它们改用 new field。



动机（Motivation）

在 classes 之间移动状态（states）和行为，是重构过程中必不可少的措施。随着系统发展，你会发现自己需要新的 classes，并需要将原本的工作责任拖到新的 class 中。这个星期中合理而正确的设计决策，到了下个星期可能不再正确。这没问题；如果你从来没遇到这种情况，那才有问题。

如果我发现，对于一个 field（值域），在其所驻 class 之外的另一个 class 中有更多函数使用了它，我就会考虑搬移这个 field。上述所谓「使用」可能是通过设值/取值（setting/getting）函数间接进行。我也可能移动该 field 的用户（某函数），这取决于是否需要保持接口不受变化。如果这些函数看上去很适合待在原地，我就选择搬移 field。

使用 *Extract Class*（149）时，我也可能需要搬移 field。此时我会先搬移 field，然后再搬移函数。

作法（Mechanics）

- 如果 field 的属性是 public，首先使用 *Encapsulate Field*（206）将它封装起来。
  - ⇨ 如果你有可能移动那些频繁访问该 field 的函数，或如果有许多函数访问某个 field，先使用 *Self Encapsulate Field*（171）也许会有帮助。
- 编译，测试。
- 在 target class 中建立与 source field 相同的 field，并同时建立相应的设值/取值（setting/getting）函数。

- 编译 target class。
- 决定如何在 source object 中引用 target object。
  - ⇨ 一个现成的 field 或 method 可以助你得到 target object。如果没有，就看能否轻易建立这样一个函数。如果还不行，就得在 source class 中新建一个 field 来存放 target object。这可能是个永久性修改，但你也可以暂不公开它，因为后续重构可能会把这个新建 field 除掉。
- 删除 source field。
- 将所有「对 source field 的引用」替换为「对 target 适当函数的调用」。
  - ⇨ 如果是「读取」该变量，就把「对 source field 的引用」替换为「对 target 取值函数（getter）的调用」；如果是「赋值」该变量，就把「对 source field 的引用」替换成「对设值函数（setter）的调用」。
  - ⇨ 如果 source field 不是 private，就必须在 source class 的所有 subclasses 中查找 source field 的引用点，并进行相应替换。
- 编译，测试。

范例（Examples）

下面是 Account class 的部分代码：

```
class Account...
    private AccountType _type;
    private double _interestRate;
    double interestForAmount_days (double amount, int days) {
        return _interestRate * amount * days / 365;
    }
```

我想把表示利率的 _interestRate 搬移到 AccountType class 去。目前已有数个函数引用了它，interestForAmount_days() 就是其一。下一步我要在 AccountType 中建立 _interestRate field 以及相应的访问函数：

```
class AccountType...
    private double _interestRate;

    void setInterestRate (double arg) {
        _interestRate = arg;
    }
    double getInterestRate () {
        return _interestRate;
    }
```

这时候我可以编译新的 AccountType class。

现在，我需要让 Account class 中访问 _interestRate field 的函数转而使用 AccountType 对象，然后删除 Account class 中的 _interestRate field。我必须删除 source field，才能保证其访问函数的确改变了操作对象，因为编译器会帮我指出未正确获得修改的函数。

```
private double _interestRate;
double interestForAmount_days (double amount, int days) {
    return _type.getInterestRate() * amount * days / 365;
}
```

范例：使用 Self-Encapsulation（自我封装）

如果有很多函数已经使用了_interestRate field，我应该先运用 *Self Encapsulate Field*（171）：

```
class Account...
  private AccountType _type;
  private double _interestRate;
  double interestForAmount_days (double amount, int days) {
      return getInterestRate() * amount * days / 365;
  }
  private void setInterestRate (double arg) {
      _interestRate = arg;
  }
  private double getInterestRate () {
      return _interestRate;
  }
```

这样，在搬移 field 之后，我就只需要修改访问函数（accessors）就行了：

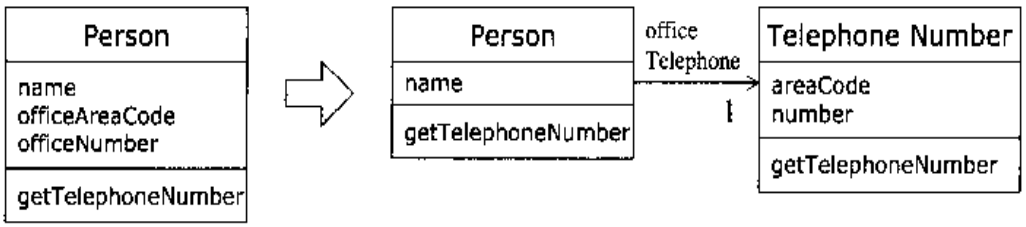
```
double interestForAmountAndDays (double amount, int days) {
    return getInterestRate() * amount * days / 365;
}
private void setInterestRate (double arg) {
    _type.setInterestRate(arg);
}
private double getInterestRate () {
    return _type.getInterestRate();
}
```

如果以后有必要，我可以修改访问函数（accessors）的用户，让它们使用新对象。*Self Encapsulate Field*（171）使我得以保持小步前进。如果我需要对 class 做许多处理，保持小步前进是有帮助的。特别值得一提的是：首先使用 *Self Encapsulate Field*（171）使我得以更轻松使用 *Move Method*（142）将函数搬移到 target class 中。如果待移函数引用了 field 的访问函数（accessors），那么那些引用点是无须修改的。

7.3 Extract Class（提炼类）

某个 class 做了应该由两个 classes 做的事。

建立一个新 class，将相关的值域和函数从旧 class 搬移到新 class。



动机（Motivation）

你也许听过类似这样的教诲：一个 class 应该是一个清楚的抽象（abstract），处理一些明确的责任。但是在实际工作中，class 会不断成长扩展。你会在这儿加入一些功能，在那儿加入一些数据。给某个 class 添加一项新责任时，你会觉得不值得为这项责任分离出一个单独的 class。于是，随着责任不断增加，这个 class 会变得过份复杂。很快，你的 class 就会变成一团乱麻。

这样的 class 往往含有大量函数和数据。这样的 class 往往太大而不易理解。此时你需要考虑哪些部分可以分离出去，并将它们分离到一个单独的 class 中。如果某些数据和某些函数总是一起出现，如果某些数据经常同时变化甚至彼此相依，这就表示你应该将它们分离出去。一个有用的测试就是问你自己，如果你搬移了某些值域和函数，会发生什么事？其他值域和函数是否因此变得无意义？

另一个往往在开发后期出现的信号是 class 的「subtyped 方式」。如果你发现 subtyping 只影响 class 的部分特性，或如果你发现某些特性「需要以此方式 subtyped」，某些特性「需要以彼方式 subtyped」，这就意味你需要分解原来的 class。

作法（Mechanics）

- 决定如何分解 class 所负责任。
- 建立一个新 class，用以表现从旧 class 中分离出来的责任。
  - ⇨ 如果旧 class 剩下的责任与旧 class 名称不符，为旧 class 易名。

- 建立「从旧 class 访问新 class」的连接关系（link）。
  - ⇨ 也许你有可能需要一个双向连接。但是在真正需要它之前，不要建立「从新 class 通往旧 class」的连接。
- 对于你想搬移的每一个值域，运用 *Move Field*（146）搬移之。
- 每次搬移后，编译、测试。
- 使用 *Move Method*（142）将必要函数搬移到新 class。先搬移较低层函数（也就是「被其他函数调用」多于「调用其他函数」者），再搬移较高层函数。
- 每次搬移之后，编译、测试。
- 检查，精简每个 class 的接口。
  - ⇨ 如果你建立起双向连接，检查是否可以将它改为单向连接。
- 决定是否让新 class 曝光。如果你的确需要曝光它，决定让它成为 *reference* object（引用型对象）或 immutable *value* object（不可变之「实值型对象」）。

范例（Examples）

让我们从一个简单的 Person class 开始：

```
class Person...
  public String getName() {
      return _name;
  }
  public String getTelephoneNumber() {
      return ("(" + _officeAreaCode + ") " + _officeNumber);
  }
  String getOfficeAreaCode() {
      return _officeAreaCode;
  }
  void setOfficeAreaCode(String arg) {
      _officeAreaCode = arg;
  }
  String getOfficeNumber() {
      return _officeNumber;
  }
  void setOfficeNumber(String arg) {
      _officeNumber = arg;
  }

  private String _name;
  private String _officeAreaCode;
  private String _officeNumber;
```

在这个例子中，我可以将「与电话号码相关」的行为分离到一个独立 class 中。首先我要定义一个 TelephoneNumber class 来表示「电话号码」这个概念：

```
class TelephoneNumber {
}
```

易如反掌！然后，我要建立从 Person 到 TelephoneNumber 的连接：

```
class Person...
  private TelephoneNumber _officeTelephone =
                                          new TelephoneNumber();
```

现在，我运用 *Move Field*（146）移动一个值域：

```
class TelephoneNumber {
    String getAreaCode() {
        return _areaCode;
    }
    void setAreaCode(String arg) {
        _areaCode = arg;
    }
    private String _areaCode;
}
class Person...
    public String getTelephoneNumber() {
        return ("(" + getOfficeAreaCode() + ") " + _officeNumber);
    }
    String getOfficeAreaCode() {
        return _officeTelephone.getAreaCode();
    }
    void setOfficeAreaCode(String arg) {
        _officeTelephone.setAreaCode(arg);
    }
```

然后我可以移动其他值域，并运用 *Move Method*（142）将相关函数移动到 TelephoneNumber class 中：

```
class Person...
    public String getName() {
        return _name;
    }
    public String getTelephoneNumber(){
        return _officeTelephone.getTelephoneNumber();
    }
    TelephoneNumber getOfficeTelephone() {
        return _officeTelephone;
    }

    private String _name;
    private TelephoneNumber _officeTelephone = new TelephoneNumber();
```

```
class TelephoneNumber...
    public String getTelephoneNumber() {
        return ("(" + _areaCode + ") " + _number);
    }
    String getAreaCode() {
        return _areaCode;
    }
    void setAreaCode(String arg) {
        _areaCode = arg;
    }
    String getNumber() {
        return _number;
    }
    void setNumber(String arg) {
        _number = arg;
    }
    private String _number;
    private String _areaCode;
```

下一步要做的决定是：要不要对客户揭示这个新class？我可以将Person中「与电话号码相关」的函数委托（delegating）至TelephoneNumber，从而完全隐藏这个新class；也可以直接将它对用户曝光。我还可以将它暴露给部分用户（位于同一个package中的用户），而不暴露给其他用户。

如果我选择暴露新class，我就需要考虑别名（aliasing）带来的危险。如果我暴露了TelephoneNumber，而有个用户修改了对象中的_areaCode值域值，我又怎么能知道呢？而且，做出修改的可能不是直接用户，而是用户的用户的用户。

面对这个问题，我有下列数种选择：

1. 允许任何对象修改TelephoneNumber对象的任何部分。这就使得TelephoneNumber对象成为引用对象（reference object），于是我应该考虑使用 *Change Value to Reference*（179）。这种情况下，Person应该是TelephoneNumber的访问点。
2. 不许任何人「不通过Person对象就修改TelephoneNumber对象」。为了达到目的，我可以将TelephoneNumber设为不可修改的（immutable），或为它提供一个不可修改的接口（immutable interface）。
3. 另一个办法是：先复制一个TelephoneNumber对象，然后将复制得到的新对象传递给用户。但这可能会造成一定程度的迷惑，因为人们会认为他们可以修改TelephoneNumber对象值。此外，如果同一个TelephoneNumber对象被传递给多个用户，也可能在用户之间造成别名（aliasing）问题。

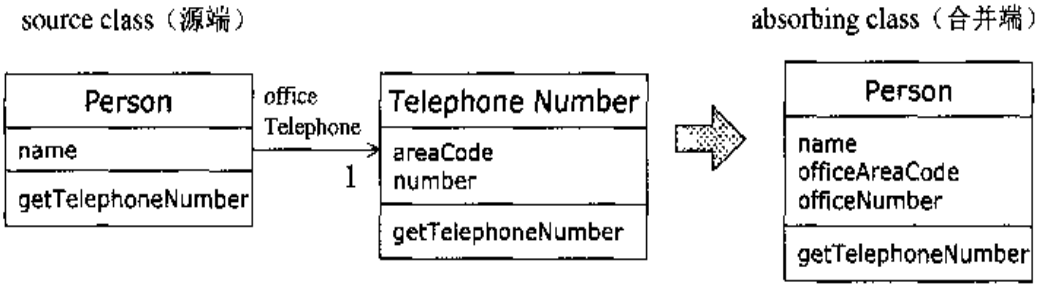
Extract Class（149）是改善并发（concurrent）程序的一种常用技术，因为它使你可以为提炼后的两个 classes 分别加锁（locks）。如果你不需要同时锁定两个对象，你就不必这样做。这方面的更多信息请看 Lea[Lea], 3.3 节。

这里也存在危险性。如果需要确保两个对象被同时锁定，你就面临事务（transaction）问题，需要使用其他类型的共享锁（shared locks）。正如 Lea[Lea] 8.1 节所讨论，这是一个复杂领域，比起一般情况需要更繁重的机制。事务（transaction）很有实用性，但是编写事务管理程序（transaction manager）则超出了大多数程序员的职责范围。

7.4 Inline Class（将类内联化）

你的某个 class 没有做太多事情（没有承担足够责任）。

将 class 的所有特性搬移到另一个 class 中，然后移除原 class。



动机（Motivation）

Inline Class（154）正好与 *Extract Class*（149）相反。如果一个 class 不再承担足够责任、不再有单独存在的理由（这通常是因为此前的重构动作移走了这个 class 的责任），我就会挑选这一「萎缩 class」的最频繁用户（也是个 class），以 *Inline Class*（154）手法将「萎缩 class」塞进去。

作法（Mechanics）

- 在 absorbing class（合并端的那个 class）身上声明 source class 的 public 协议，并将其中所有函数委托（*delegate*）至 source class。
  - ⇨ 如果「以一个独立接口表示 source class 函数」更合适的话，就应该在 *inlining* 之前先使用 *Extract Interface*（341）。
- 修改所有 source class 引用点，改而引用 absorbing class。
  - ⇨ 将 source class 声明为 private，以斩断 package 之外的所有引用可能。同时并修改 source class 的名称，这便可使编译器帮助你捕捉到所有对于 source class 的 "dangling references"（虚悬引用点）。
- 编译，测试。
- 运用 *Move Method*（142）和 *Move Field*（146），将 source class 的特性全部搬移到 absorbing class。
- 为 source class 举行一个简单的丧礼。

范例（Example）

先前（上个重构项）我从 TelephoneNumber 提炼出另一个 class，现在我要将它 *inlining* 塞回到 Person 去。 开始这两个 classes 是分离的：

```
class Person...
    public String getName() {
        return _name;
    }
    public String getTelephoneNumber(){
        return _officeTelephone.getTelephoneNumber();
    }
    TelephoneNumber getOfficeTelephone() {
        return _officeTelephone;
    }

    private String _name;
    private TelephoneNumber _officeTelephone =
        new TelephoneNumber();

class TelephoneNumber...
    public String getTelephoneNumber() {
        return ("(" + _areaCode + ") " + _number);
    }
    String getAreaCode() {
        return _areaCode;
    }
    void setAreaCode(String arg) {
        _areaCode = arg;
    }
    String getNumber() {
        return _number;
    }
    void setNumber(String arg) {
        _number = arg;
    }
    private String _number;
    private String _areaCode;
```

首先我在 Person 中声明 TelephoneNumber 的所有「可见」（public）函数：

```
class Person...
    String getAreaCode() {
        return _officeTelephone.getAreaCode(); //译注：请注意其变化
    }
    void setAreaCode(String arg) {
        _officeTelephone.setAreaCode(arg);      //译注：请注意其变化
    }
    String getNumber() {
```

```
        return _officeTelephone.getNumber();    //译注：请注意其变化
    }
    void setNumber(String arg) {
        _officeTelephone.setNumber(arg);        //译注：请注意其变化
    }
```

现在，我要找出 TelephoneNumber 的所有用户，让它们转而使用 Person 接口。于是下列代码：

```
Person martin = new Person();
martin.getOfficeTelephone().setAreaCode ("781");
```

就变成了：

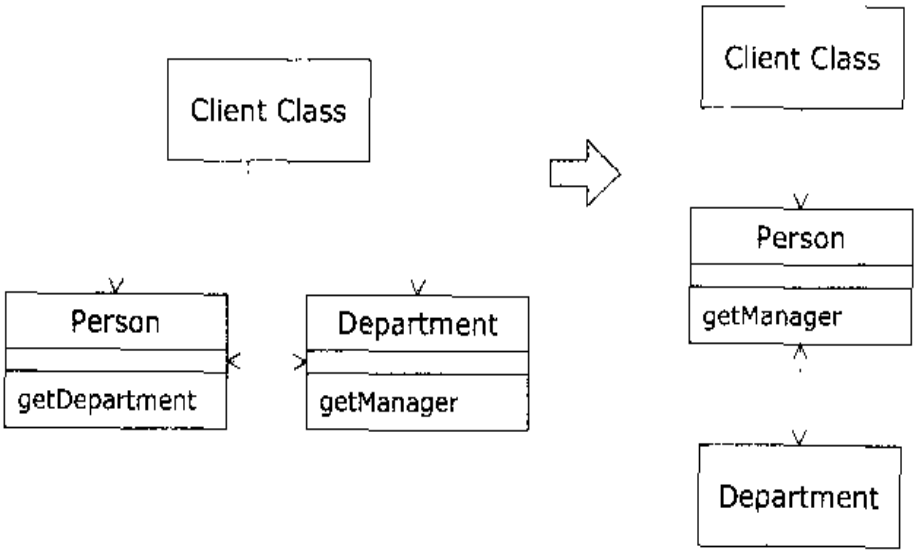
```
Person martin = new Person();
martin.setAreaCode ("781");
```

现在，我可以持续使用 *Move Method*（142）和 *Move Field*（146），直到 TelephoneNumber 不复存在。

7.5 Hide Delegate（隐藏「委托关系」）

客户直接调用其 server object（服务对象）的 delegate class。

在 server 端（某个 class）建立客户所需的所有函数，用以隐藏委托关系（delegation）。



动机（Motivation）

「封装」即使不是对象的最关键特征，也是最关键特征之一。「封装」意味每个对象都应该尽可能少了解系统的其他部分。如此一来，一旦发生变化，需要了解这一变化的对象就会比较少 — 这会使变化比较容易进行。

任何学过对象技术的人都知道：虽然 Java 允许你将值域声明为 public，但你还是应该隐藏对象的值域。随着经验日渐丰富，你会发现，有更多可以（并值得）封装的东西。

如果某个客户调用了「建立于 server object（服务对象）的某个值域基础之上」的函数，那么客户就必须知晓这一委托对象（delegate object。译注：即 server object 的那个特殊值域）。万一委托关系发生变化，客户也得相应变化。你可以在 server 端放置一个简单的委托函数（delegating method），将委托关系隐藏起来，从而去除这种依存性（图 7.1）。这么一来即便将来发生委托关系上的变化，变化将被限制在 server 中，不会波及客户。

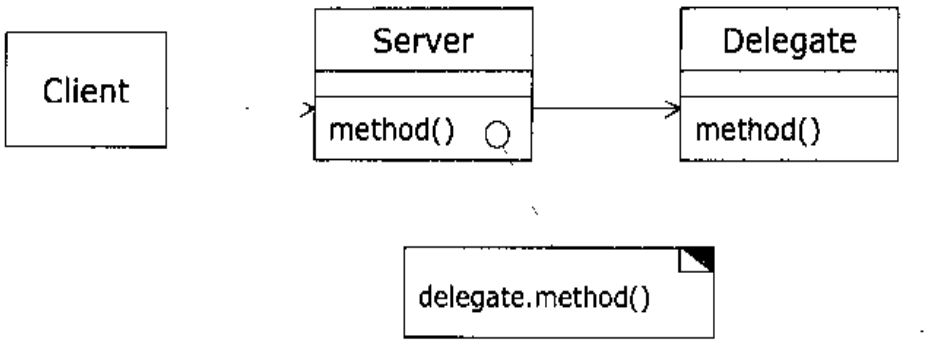


图 7.1 简单的委托关系（delegation）

对于某些客户或全部客户，你可能会发现，有必要先使用 *Extract Class*（149）。一旦你对所有客户都隐藏委托关系（delegation），你就可以将 server 接口中的所有委托都移除。

作法（Mechanics）

- 对于每一个委托关系中的函数，在 server 端建立一个简单的委托函数（delegating method）。
- 调整客户，令它只调用 server 提供的函数（译注：不得跳过径自调用下层）。
  - ⇨ 如果 client（客户）和 server 不在同一个 package，考虑修改委托函数（delegate method）的访问权限，让 client 得以在 package 之外调用它。
- 每次调整后，编译并测试。
- 如果将来不再有任何客户需要取用图 7.1 的 `Delegate`（受托类），便可移除 server 中的相关访问函数（accessor for the delegate）。
- 编译，测试。

范例（Examples）

本例从两个 classes 开始，代表「人」的 `Person` 和代表「部门」的 `Department`：

```
class Person {
    Department _department;

    public Department getDepartment() {
        return _department;
    }
```

```
    public void setDepartment(Department arg) {
        _department = arg;
    }
}

class Department {
    private String _chargeCode;
    private Person _manager;

    public Department (Person manager) {
        _manager = manager;
    }

    public Person getManager() {
        return _manager;
    }
...
```

如果客户希望知道某人的经理是谁，他必须先取得 Department 对象：

```
manager = john.getDepartment().getManager();
```

这样的编码就是对客户揭露了 Department 工作原理，于是客户知道：Department 用以追踪「经理」这条信息。如果对客户隐藏 Department，可以减少耦合（coupling）。为了这一目的，我在 Person 中建立一个简单的委托函数：

```
public Person getManager() {
    return _department.getManager();
}
```

现在，我得修改 Person 的所有客户，让它们改用新函数：

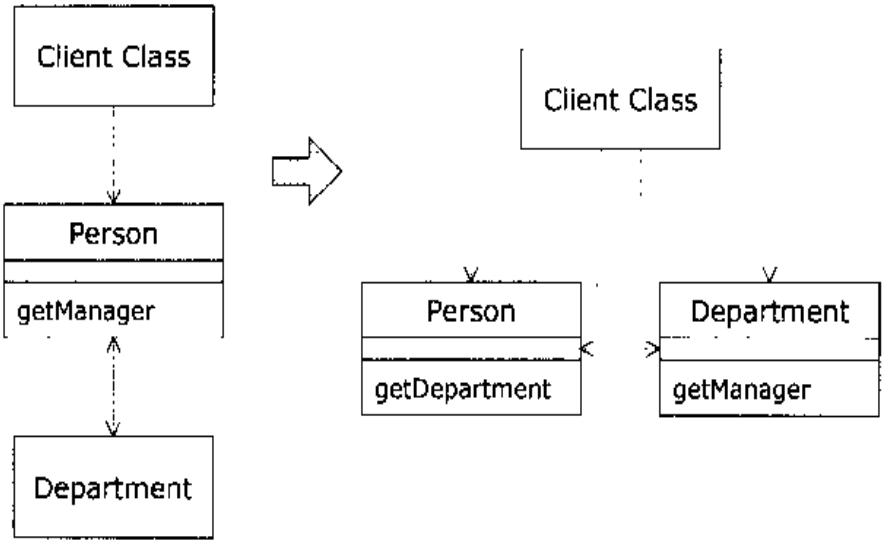
```
manager = john.getManager();
```

只要完成了对 Department 所有函数的委托关系，并相应修改了 Person 的所有客户，我就可以移除 Person 中的访问函数 getDepartment() 了。

7.6 Remove Middle Man（移除中间人）

某个 class 做了过多的简单委托动作（simple delegation）。

让客户直接调用 delegate（受托类）。



动机（Motivation）

在 *Hide Delegate*（157）的「动机」栏，我谈到了「封装 delegated object（受托对象）」的好处。但是这层封装也是要付出代价的，它的代价就是：每当客户要使用 delegate（受托类）的新特性时，你就必须在 server 端添加一个简单委托函数。随着 delegate 的特性（功能）愈来愈多，这一过程会让你痛苦不已。server 完全变成了一个「中间人」，此时你就应该让客户直接调用 delegate。

很难说什么程度的隐藏才是合适的。还好，有了 *Hide Delegate*（157）和 *Remove Middle Man*（160），你大可不必操心这个问题，因为你可以在系统运行过程中不断进行调整。随着系统的变化，「合适的隐藏程度」这个尺度也相应改变。六个月前恰如其分的封装，现今可能就显得笨拙。重构的意义就在于：你永远不必说对不起 — 只要把出问题的地方修补好就行了。

作法（Mechanics）

- 建立一个函数，用以取用 delegate（受托对象）。
- 对于每个委托函数（delegate method），在 server 中删除该函数，并将「客户对该函数的调用」替换为「对 delegate（受托对象）的调用」。
- 处理每个委托函数后，编译、测试。

范例（Examples）

我将以另一种方式使用先前用过的「人与部门」例子。还记得吗，上一项重构结束时，Person 将 Department 隐藏起来了：

```
class Person...
  Department _department;
  public Person getManager() {
     return _department.getManager();

class Department...
  private Person _manager;
  public Department (Person manager) {
     _manager = manager;
  }
```

为了找出某人的经理，客户代码可能这样写：

```
manager = john.getManager();
```

像这样，使用和封装 Department 都很简单。但如果大量函数都这么做，我就不得不在 Person 之中安置大量委托行为（delegations）。这就是移除中间人的时候了。首先在 Person 中建立一个「受托对象（delegate）取得函数」：

```
class Person...
  public Department getDepartment() {
     return _department;
  }
```

然后逐一处理每个委托函数。针对每一个这样的函数，我要找出通过 Person 使用的函数，并对它进行修改，使它首先获得受托对象（delegate），然后直接使用之：

```
manager = john.getDepartment().getManager();
```

然后我就可以删除 Person 的 getManager()函数。如果我遗漏了什么，编译器会告诉我。

为方便起见，我也可能想要保留一部分委托关系（delegations）。此外我也可能希望对某些客户隐藏委托关系，并让另一些用户直接使用受托对象。基于这些原因，一些简单的委托关系（以及对应的委托函数）也可能被留在原地。

7.7 Introduce Foreign Method（引入外加函数）

你所使用的 server class 需要一个额外函数，但你无法修改这个 class。

在 client class 中建立一个函数，并以一个 server class 实体作为第一引数（argument）。

```
Date newStart = new Date (previousEnd.getYear(),
                          previousEnd.getMonth(),previousEnd.getDate()+1);
```

```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
   return new Date (arg.getYear(),arg.getMonth(),
                    arg.getDate() + 1);
}
```

动机（Motivation）

这种事情发生过太多次了：你正在使用一个 class，它真的很好，为你提供了你想要的所有服务。而后，你又需要一项新服务，这个 class 却无法供应。于是你开始咒骂：「为什么不能做这件事？」如果可以修改源码，你便可以自行添加一个新函数；如果不能，你就得在客户端编码，补足你要的那个函数。

如果 client class 只使用这项功能一次，那么额外编码工作没什么大不了，甚至可能根本不需要原本提供服务的那个 class。然而如果你需要多次使用这个函数，你就得不断重复这些代码。还记得吗，重复的代码是软件万恶之源。这些重复性代码应该被抽出来放进同一个函数中。进行本项重构时，如果你以外加函数实现一项功能，那就是一个明确信号：这个函数原本应该在提供服务的（server）class 中加以实现。

如果你发现自己为一个 server class 建立了大量外加函数，或如果你发现有许多 classes 都需要同样的外加函数，你就不应该再使用本项重构，而应该使用 *Introduce Local Extension*（164）。

但是不要忘记：外加函数终归是权宜之计。如果有可能，你仍然应该将这些函数搬移到它们的理想家园。如果代码拥有权（code ownership）是个需要考量的问题，就把外加函数交给 server class 的拥有者，请他帮你在 server class 中实现这个函数。

作法（Mechanics）

- 在 client class 中建立一个函数，用来提供你需要的功能。
  - ⇨ 这个函数不应该取用 client class 的任何特性。如果它需要一个值，把该值当作参数传给它。
- 以 server class 实体作为该函数的第一个参数。
- 将该函数注释为：「外加函数（foreign method），应在 server class 实现。」
  - ⇨ 这么一来，将来如果有机会将外加函数搬移到 server class 中，你便可以轻松找出这些外加函数。

范例（Examples）

程序中，我需要跨过一个收费周期（billing period）。原本代码像这样：

```
Date newStart = new Date (previousEnd.getYear(),
   previousEnd.getMonth(), previousEnd.getDate() + 1);
```

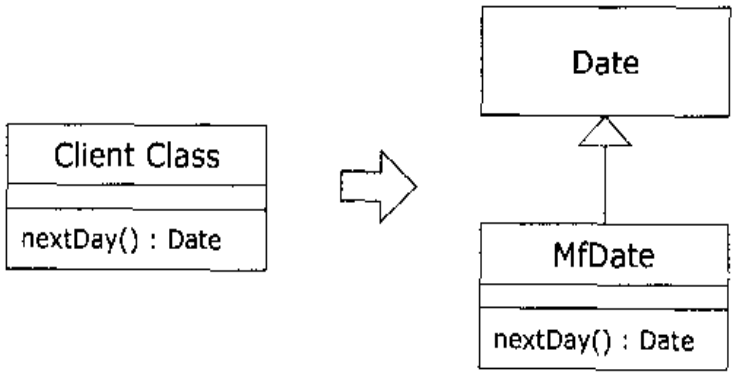
我可以将赋值运算右侧代码提炼到一个独立函数中。这个函数就是 Date class 的一个外加函数：

```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
   // foreign method, should be on date
   return new Date (arg.getYear(),arg.getMonth(),arg.getDate()+1);
}
```

7.8 Introduce Local Extension（引入本地扩展）

你所使用的 server class 需要一些额外函数，但你无法修改这个 class。

建立一个新 class，使它包含这些额外函数。让这个扩展品成为 source class 的 subclass（子类）或 wrapper（外覆类）。



original class（原始类）
extended class（被扩展类）

subclass（子类）或 wrapper（外覆类）

动机（Motivation）

很遗憾，classes 的作者无法预知未来，他们常常没能为你预先准备一些有用的函数。如果你可以修改源码，最好的办法就是直接加入自己需要的函数。但你经常无法修改源码。如果只需要一两个函数，你可以使用 *Introduce Foreign Method*（162）。但如果你需要的额外函数超过两个，外加函数（foreign methods）就很难控制住它们了。所以，你需要将这些函数组织在一起，放到一个恰当地方去。要达到这一目的，标准对象技术 subclassing 和 wrapping 是显而易见的办法。这种情况下我把 subclass 或 wrapper 称为 local extension（本地扩展）。

所谓 local extension，是一个独立的 class，但也是其 extended class 的 subtype（译注：这里的 subtype 不同于 subclass；它和 extended class 并不一定存在严格的继承关系，只要能够提供 extended class 的所有特性即可）。这意味它提供 original class 的一切特性，同时并额外添加新特性。在任何使用 original class 的地方，你都可以使用 local extension 取而代之。

使用 local extension（本地扩展）使你得以坚持「函数和数据应该被包装在形式良好的单元内」这一原则。如果你一直把本该放在 extension class 中的代码零散放置于其他 classes 中，最终只会让其他这些 classes 变得过分复杂，并使得其中函数难以被复用。

在 subclass 和 wrapper 之间做选择时，我通常首选 subclass，因为这样的工作量比较少。制作 subclass 的最大障碍在于，它必须在对象创建期（object-creation time）实施。如果我可以接管对象创建过程，那当然没问题；但如果你想在对象创建之后再使用 local extension，就有问题了。此外，"subclassing" 还迫使我必须产生一个 subclass 对象，这种情况下如果有其他对象引用了旧对象，我们就同时有两个对象保存了原数据！如果原数据是不可修改的（immutable），那也没问题，我可以放心进行拷贝；但如果原数据允许被修改，问题就来了，因为这时候闹了双包，一个修改动作无法同时改变两份拷贝。这时候我就必须改用 wrapper。但使用 wrapper 时，对 local extension 的修改会波及原物（original），反之亦然。

作法（Mechanics）

- 建立一个 extension class，将它作为原物（原类）的 subclass 或 wrapper。
- 在 extension class 中加入转型构造函数（converting constructors）。
  - ⇨ 所谓「转型构造函数」是指接受原物（original）作为参数。如果你采用 subclassing 方案，那么转型构造函数应该调用适当的 superclass 构造函数；如果你采用 wrapper 方案，那么转型构造函数应该将它所获得之引数（argument）赋值给「用以保存委托关系（delegate）」的那个值域。
- 在 extension class 中加入新特性。
- 根据需要，将原物（original）替换为扩展物（extension）。
- 将「针对原始类（original class）而定义的所有外加函数（foreign methods）」搬移到扩展类（extension）中。

范例（Examples）

我将以 Java 1.0.1 的 `Date` class 为例。Java 1.1 已经提供了我想要的功能，但是在它到来之前的那段日子，很多时候我需要扩展 Java 1.0.1 的 `Date` class。

第一件待决事项就是使用 subclass 或 wrapper。subclassing 是比较显而易见的办法：

```
class mfDate extends Date {
   public nextDay()...
   public dayOfYear()...
```

wrapper 则需用上委托（delegation）：

```
class mfDate {
   private Date _original;
```

范例：使用 Subclass（子类）

首先，我要建立一个新的 MfDateSub class 来表示「日期」（译注："Mf" 是作者 Martin Fowler 的姓名缩写），并使其成为 Date 的 subclass：

```
class MfDateSub extends Date
```

然后，我需要处理 Date 和我的 extension class 之间的不同处。MfDateSub 构造函数需要委托（delegating）给 Date 构造函数：

```
public MfDateSub (String dateString) {
    super (dateString);
};
```

现在，我需要加入一个转型构造函数，其参数是一个隶属原类的对象：

```
public MfDateSub (Date arg) {
    super (arg.getTime());
}
```

现在，我可以在 extension class 中添加新特性，并使用 *Move Method*（142）将所有外加函数（foreign methods）搬移到 extension class。于是，下面的代码：

```
client class...
  private static Date nextDay(Date arg) {
      // foreign method, should be on date
      return new Date (arg.getYear(),arg.getMonth(),
      arg.getDate() + 1);
  }
```

经过搬移之后，就成了：

```
class MfDate...
  Date nextDay() {
      return new Date (getYear(),getMonth(), getDate() + 1);
  }
```

范例：使用 Wrapper（外覆类）

首先声明一个 wrapping class：

```
class MfDateWrap {
    private Date _original;
}
```

使用 wrapping 方案时，我对构造函数的设定与先前有所不同。现在的构造函数将只是执行一个单纯的委托动作（delegation）：

```
public MfDateWrap (String dateString) {
    _original = new Date(dateString);
};
```

而转型构造函数则只是对其 *instance* 变量赋值而已：

```
public MfDateWrap (Date arg) {
    _original = arg;
}
```

接下来是一项枯燥乏味的工作：为原始类的所有函数提供委托函数。我只展示两个函数，其他函数的处理依此类推。

```
public int getYear() {
    return _original.getYear();
}
public boolean equals (MfDateWrap arg) {
    return (toDate().equals(arg.toDate()));
}
```

完成这项工作之后，我就可以后使用 ***Move Method***（142）将日期相关行为搬移到新 class 中。于是以下代码：

```
client class...
  private static Date nextDay(Date arg) {
      // foreign method, should be on date
      return new Date (arg.getYear(),arg.getMonth(),
                       arg.getDate()+1);
  }
```

经过搬移之后，就成了：

```
class MfDate...
  Date nextDay() {
      return new Date (getYear(),getMonth(), getDate() + 1);
  }
```

使用 wrappers 有一个特殊问题：如何处理「接受原始类之实体为参数」的函数？例如：

```
public boolean after (Date arg)
```

由于无法改变原始类（original），所以我只能以一种方式使用上述的 `after()`：

```
aWrapper.after(aDate)        // can be made to work
aWrapper.after(anotherWrapper) // can be made to work
aDate.after(aWrapper)        // will not work
```

这样覆写（overriding）的目的是为了向用户隐藏 wrapper 的存在。这是一个好策略，因为 wrapper 的用户的确不应该关心 wrapper 的存在，的确应该可以同样地对待 wrapper（外覆类）和 original（原始类）。但是我无法完全隐藏此一信息，因为某些系统所提供的函数（例如 `equals()`）会出问题。你可能会认为：你可以在

MfDateWrap class 中覆写 equals()，像这样：

```
public boolean equals (Date arg) // causes problems
```

但这样做是危险的，因为尽管我达到了自己的目的，Java 系统的其他部分都认为 equals()符合交换律：如果 a.equals(b)为真，那么 b.equals(a)也必为真。违反这一规则将使我遭遇一大堆莫名其妙的错误。要避免这样的尴尬境地，惟一办法就是修改 Date class。但如果我能够修改 Date，我又何必进行此项重构？所以，在这种情况下，我只能（必须）向用户暴露「我进行了包装」这一事实。我将以一个新函数来进行日期之间的相等性检查（equality tests）：

```
public boolean equalsDate (Date arg)
```

我可以重载 equalsDate()，让一个重载版本接受 Date 对象，另一个重载版本接受 MfDateWrap 对象。这样我就不必检查未知对象的型别了：

```
public boolean equalsDate (MfDateWrap arg)
```

subclassing 方案中就没有这样的问题，只要我不覆写原函数就行了。但如果我覆写了 original class 中的函数，那么寻找函数时，我会被搞得晕头转向。一般来说，我不会在 extension class 中覆写 original class 的函数，我只会添加新函数。

> 译注：**equality**（相等性）是一个很基础的大题目。《*Effective Java*》by Joshua Bloch 第 3 章，以及《*Practical Java*》by Peter Haggar 第 2 章，对此均有很深入的讨论。这两本书对于其他的基础大题目如 Serializable, Comparable, Cloneable, hashCode() 也都有深刻讨论。

8

重新组织数据

Organizing Data

本章之中，我将讨论数个「能让你更轻松运用数据」的重构手法。很多人或许会认为 *Self Encapsulate Field*（171）有点多余，但是关于「对象应该直接访问其中的数据，抑或应该通过访问函数（accessor）来访问」这一问题，争论的声音从来不曾停止。有时候你确实需要访问函数，此时你就可以通过 *Self Encapsulate Field*（157）得到它们。通常我会选择「直接访问」方式，因为我发现，只要我想做，任何时候进行这项重构都是很简单的。

面向对象语言有一个很有用的特征：除了允许使用传统语言提供的简单数据型别，它们还允许你定义新型别。不过人们往往需要一段时间才能习惯这种编程方式。开始你常会使用一个简单数值来表示某个概念；随着对系统的深入了解，你可能会明白，以对象表示这个概念，可能更合适。*Replace Value with Object*（175）让你可以将「哑」数据（dumb data）变成会说话的对象（articulate objects）。如果你发现程序中有太多地方需要这一类对象，你也可以使用 *Change Value to Reference*（179）将它们变成 *reference* object。

如果你看到一个 array 的行为方式很像一个数据结构，你可以使用 *Replace Array with Object*（186）把 array 变成对象，从而使这个数据结构更清晰地显露出来。但这只是第一步，当你使用 *Move Method*（142）为这个新对象加入相应行为时，真正的好处才得以体现。

魔法数（magic numbers），也就是带有特殊含义的数字，从来都是个问题。我还清楚记得，一开始学习编程的时候，老师就告诉我不要使用魔法数。但它们还是不时出现。因此，只要弄清楚魔法数的用途，我就运用 *Replace Magic Number with Symbolic Constant*（204）将它们除掉，以绝后患。

对象之间的关联（links）可以单向，也可以双向。单向关联比较简单，但有时为了支持一项新功能，你需要以 *Change Unidirectional Association to Bidirectional*

（197）将它变成双向关联。*Change Bidirectional Association to Unidirectional*（200）则恰恰相反：如果你发现不再需要双向关联，可以使用这项重构将它变成单向关联。

我常常遇到这样的情况：GUI classes 竟然去处理不该它们处理的业务逻辑（business logic）。为了把这些处理业务逻辑的行为移到合适的 domain class 去，你需要在 domain class 中保存这些逻辑的相关数据，并运用 *Duplicate Observed Data*（189）提供对 GUI 的支持。一般来说，我不喜欢重复的数据，但这是一个例外，因为这里的重复数据通常是不可避免的。

面向对象编程（OOP）的关键原则之一就是封装。如果一个 class 暴露了任何 public 数据，你就应该使用 *Encapsulate Field*（206）将它高雅而正派地包装起来。如果被暴露的数据是个群集（collection），你就应该使用 *Encapsulate Collection*（208），因为群集有其特殊协议。如果一整笔记录（record）都被裸露在外，你就应该使用 *Replace Record with Data Class*（217）。

需要特别对待的一种数据是 type code（型别码）：这是一种特殊数值，用来指出「与实体所属之型别相关的某些东西」。Type code 通常以枚举（enumeration）形式出现，并且通常以 static final 整数实现之。如果这些 type code 用来表现某种信息，并且不会改变所属 class 的行为，你可以运用 *Replace Type Code with Class*（218）将它们替换掉，这项重构会为你提供更好的型别检查，以及一个更好的平台，使你可以在未来更方便地将相关行为添加进去。另一方面，如果 class 的行为受到 type code 的影响，你就应该尽可能使用 *Replace Type Code with Subclasses*（223）。如果做不到，就只好使用更复杂（同时也更灵活）的 *Replace Type Code with State/Strategy*（227）。

8.1 Self Encapsulate Field（自封装值域）

你直接访问一个值域（field），但与值域之间的耦合关系逐渐变得笨拙。

为这个值域建立取值/设值函数（getting/setting methods），**并且只以这些函数来访问值域。**

```
private int _low, _high;
boolean includes (int arg) {
    return arg >= _low && arg <= _high;
}
```

```
private int _low, _high;
boolean includes (int arg) {
    return arg >= getLow() && arg <= getHigh();
}
int getLow() {return _low;}
int getHigh() {return _high;}
```

动机（Motivation）

在「值域访问方式」这个问题上，存在两种截然不同的观点：其中一派认为，在该变量定义所在的 class 中，你应该自由（直接）访问它；另一派认为，即使在这个 class 中你也应该只使用访问函数间接访问之。两派之间的争论可以说是如火如荼。（参见 Auer 在 [Auer] p.413 和 Beck 在 [Beck] 上的讨论。）

本质而言，「间接访问变量」的好处是，subclass 得以通过「覆写一个函数」而改变获取数据的途径；它还支持更灵活的数据管理方式，例如 lazy initialization（意思是：只有在需要用到某值时，才对它初始化）。

「直接访问变量」的好处则是：代码比较容易阅读。阅读代码的时候，你不需要停下来说：『啊，这只是个取值函数。』

面临选择时，我总是做两手准备。通常情况下我会很乐意按照团队中其他人的意愿来做。就我自己而言，我比较喜欢先使用「直接访问」方式，直到这种方式给我带来麻烦为止。如果「直接访问」给我带来麻烦，我就会转而使用「间接访问」方式。重构给了我改变主意的自由。

如果你想访问 superclass 中的一个值域，却又想在 subclass 中将「对这个变量的访问」改为一个计算后的值，这就是最该使用 *Self Encapsulate Field*（171）的时候。「值域自我封装」只是第一步。完成自我封装之后，你可以在 subclass 中根据自己的需要随意覆写取值/设值函数（getting/setting methods）。

作法（Mechanics）

- 为「待封装值域」建立取值/设值函数（getting/setting methods）。
- 找出该值域的所有引用点，将它们全部替换为「对于取值/设值函数的调用」。
  - ⇨ 如果引用点是「读取」值域值，就将它替换为「调用取值函数」；如果引用点是「设定」值域值，就将它替换为「调用设值函数」。
  - ⇨ 你可以暂时为该值域改名，让编译器帮助你查找引用点。
- 将该值域声明为 private。
- 复查，确保找出所有引用点。
- 编译，测试。

范例（Example）

下面这个例子看上去有点过分简单。不过，嘿，起码它写起来很快：

```
class IntRange {

    private int _low, _high;

    boolean includes (int arg) {
        return arg >= _low && arg <= _high;
    }

    void grow(int factor) {
        _high = _high * factor;
    }
```

```
IntRange (int low, int high) {
    _low = low;
    _high = high;
}
```

为了封装 _low 和_high 这两个值域，我先定义「取值/设值函数」（如果此前没有定义的话），并使用它们：

```
class IntRange {

    boolean includes (int arg) {
        return arg >= getLow() && arg <= getHigh();
    }
    void grow(int factor) {
        setHigh (getHigh() * factor);
    }

    private int _low, _high;

    int getLow() {
        return _low;
    }
    int getHigh() {
        return _high;
    }
    void setLow(int arg) {
        _low = arg;
    }
    void setHigh(int arg) {
        _high = arg;
    }
```

使用本项重构时，你必须小心对待「在构造函数中使用设值函数」的情况。一般说来，设值函数被认为应该在「对象创建后」才使用。所以初始化过程中的行为有可能与设值函数的行为不同。这种情况下，我也许在构造函数中直接访问值域，要不就是建立另一个独立的初始化函数：

```
IntRange (int low, int high) {
    initialize (low, high);
}
private void initialize (int low, int high) {
    _low = low;
    _high = high;
}
```

一旦你拥有一个subclass，上述所有动作的价值就体现出来了。如下所示：

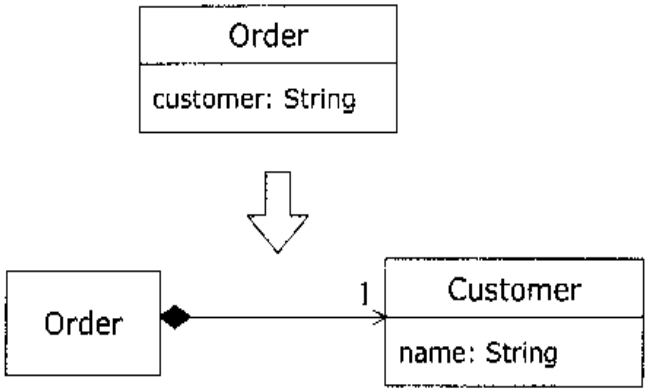
```
c.ass CappedRange extends IntRange {

    CappedRange (int low, int high, int cap) {
        super (low, high);
        _cap = cap;
    }

    private int _cap;

    int getCap() {
        return _cap;
    }
    int getHigh() {
        return Math.min(super.getHigh(), getCap());
    }
}
```

现在，我可以在 CappedRange class 中覆写 getHigh()，从而加入对 cap 的考虑，而不必修改 IntRange class 的任何行为。

8.2 Replace Data Value with Object（以对象取代数据值）

你有一笔数据项（data item），需要额外的数据和行为。

将这笔数据项变成一个对象。



动机（Motivation）

开发初期，你往往决定以简单的数据项（data item）表示简单的行为。但是，随着开发的进行，你可能会发现，这些简单数据项不再那么简单了。比如说，一开始你可能会用一个字符串来表示「电话号码」概念，但是随后你就会发现，电话号码需要「格式化」、「抽取区号」之类的特殊行为。如果这样的数据项只有一二个，你还可以把相关函数放进数据项所属的对象里头；但是 Duplicate Code 臭味和 Feature Envy 臭味很快就会从代码中散发出来。当这些臭味开始出现，你就应该将数据值（data value）变成对象（object）。

作法（Mechanics）

- 为「待替换数值」新建一个 class，在其中声明一个 final 值域，其型别和 source class 中的「待替换数值」型别一样。然后在新 class 中加入这个值域的取值函数（getter），再加上一个「接受此值域为参数」的构造函数。
- 编译。
- 将 source class 中的「待替换数值值域」的型别改为上述的新建 class。
- 修改 source class 中此一值域的取值函数（getter），令它调用新建 class 的取值函数。

- 如果 source class 构造函数中提及这个「待替换值域」（多半是赋值动作），我们就修改构造函数，令它改用新 class 的构造函数来对值域进行赋值动作。
- 修改 source class 中「待替换值域」的设值函数（setter），令它为新 class 创建一个实体。
- 编译，测试。
- 现在，你有可能需要对新 class 使用 *Change Value to Reference*（179）。

范例（Example）

下面有一个代表「定单」的 order class，其中以一个字符串记录定单客户。现在，我希望改以一个对象来表示客户信息，这样我就有充裕的弹性保存客户地址、信用等级等等信息，也得以安置这些信息的操作行为。Order class 最初如下：

```
class Order...
  public Order (String customer) {
     _customer = customer;
  }
  public String getCustomer() {
     return _customer;
  }
  public void setCustomer(String arg) {
     _customer = arg;
  }
  private String _customer;
```

Order class 的客户代码可能像下面这样：

```
private static int numberOfOrdersFor(Collection orders,
                                     String customer) {
    int result = 0;
    Iterator iter = orders.iterator();
    while (iter.hasNext()) {
       Order each = (Order) iter.next();
       if (each.getCustomer().equals(customer)) result++;
    }
    return result;
}
```

首先，我要新建一个 Customer class 来表示「客户」概念。然后在这个 class 中建立一个 final 值域，用以保存一个字符串，这是 Order class 目前所使用的。我将这个新值域命名为 _name，因为这个字符串的用途就是记录客户名称。此外我还要为这个字符串加上取值函数（getter）和构造函数（constructor）。

```
class Customer {
    public Customer (String name) {
       _name = name;
    }
```

```
    public String getName() {
        return _name;
    }
    private final String _name;
}
```

现在，我要将 Order 中的 _customer 值域的型别修改为 Customer；并修改所有引用此一值域的函数，让它们恰当地改而引用 Customer 实体。其中取值函数和构造函数的修改都很简单；至于设值函数（setter），我让它创建一份 Customer 实体。

```
class Order...
  public Order (String customer) {                    // constructor
      _customer = new Customer(customer);
  }
  public String getCustomer() {                       // getter
      return _customer.getName();
  }
  private Customer _customer;

  public void setCustomer(String arg) {               //setter
      _customer = new Customer(arg);
  }
```

设值函数需要创建一份 Customer 实体，这是因为以前的字符串是个实值对象（*value* object），所以现在的 Customer 对象也应该是个实值对象。这也就意味每个 Order 对象都包含自己的一个 Customer 对象。注意这样一条规则：实值对象应该是不可修改内容的。这便可以避免一些讨厌的「别名」（aliasing）错误。日后或许我会想让 Customer 对象成为引用对象（*reference* object），但那是另一项重构手法的责任。现在我可以编译并测试了。

我需要观察 Order class 中的 _customer 值域的操作函数，并作出一些修改，使它更好地反映出修改后的新形势。对于取值函数，我会使用 *Rename Method*（273）改变其名称，让它更清晰地表示，它所返回的是消费者名称，而不是个 Customer 对象。

```
public String getCustomerName() {
    return _customer.getName();
}
```

至于构造函数和设值函数，我就不必修改其签名（signature）了，但参数名称得改：

```
public Order (String customerName) {
    _customer = new Customer(customerName);
}
public void setCustomer(String customerName) {
    _customer = new Customer(customerName);
}
```

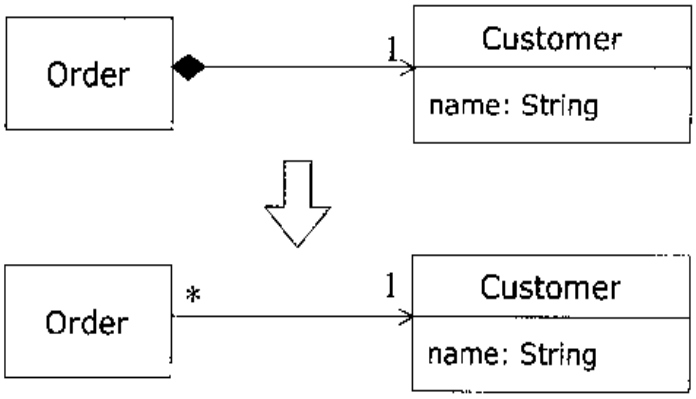
后继的其他重构也许会让我添加新的、「接受既有 Customer 对象作为参数」的构造函数和设值函数。

本次重构到此为止。但是，这个案例和其他很多案例一样，还需要一个后续步骤。如果想在 Customer 中加入信用等级、地址之类的其他信息，现在还做不到，因为目前的 Customer 还是被作为实值对象（*value* object）来对待，每个 Order 对象都拥有自己的 Customer 对象。为了给 Customer class 加上信用等级、地址之类的属性，我必须运用 *Change Value to Reference*（179），这么一来属于同一客户的所有 Order 对象就可以共享同一个 Customer 对象。马上你就可以看到这个例子。

8.3 Change Value to Reference（将实值对象改为引用对象）

你有一个 class，衍生出许多相等实体（equal instances），你希望将它们替换为单一对象。

将这个 *value* object（实值对象）变成一个 *reference* object（引用对象）。



动机（Motivation）

在许多系统中，你都可以对对象做一个有用的分类：*reference* object 和 *value* objects。前者就像「客户」、「帐户」这样的东西，每个对象都代表真实世界中的一个实物，你可以直接以相等操作符（==，用来检验同一性，identity）检查两个对象是否相等。后者则是像「日期」、「钱」这样的东西，它们完全由其所含的数据值来定义，你并不在意副本的存在：系统中或许存在成百上千个内容为 "1/1/2000" 的「日期」对象。当然，你也需要知道两个 *value* objects 是否相等，所以你需要覆写 `equals()`（以及 `hashCode()`）。

要在 *reference* object 和 *value* object 之间做选择有时并不容易。有时候，你会从一个简单的 *value* object 开始，在其中保存少量不可修改的数据。而后，你可能会希望给这个对象加入一些可修改数据，并确保对任何一个对象的修改都能影响到所有引用此一对象的地方。这时候你就需要将这个对象变成一个 *reference* object。

作法（Mechanics）

- 使用 *Replace Constructor with Factory Method*（304）。
- 编译，测试。

- 决定由什么对象负责提供访问新对象的途径。
  - ⇨ 可能是个静态字典（static dictionary）或一个注册对象（registry object）。
  - ⇨ 你也可以使用多个对象作为新对象的访问点（access point）。
- 决定这些 *reference* object 应该预先创建好，或是应该动态创建。
  - ⇨ 如果这些 *reference* object 是预先创建好的，而你必须从内存中将它们读取出来，那么就得确保它们在被需要的时候能够被及时加载。
- 修改 ***factory method***[5]，令它返回 *reference* object。
  - ⇨ 如果对象是预先创建好的，你就需要考虑：万一有人索求一个其实并不存在的对象，要如何处理错误？
  - ⇨ 你可能希望对 ***factory method*** 使用 *Rename Method*（273），使其传达这样的信息：它返回的是一个既存对象。
- 编译，测试。

范例（Example）

在 *Replace Data Value with Object*（175）一节中，我留下了一个重构后的程序，本节范例就从它开始。我们有下列的 `Customer` class：

```
class Customer {
    public Customer (String name) {
        _name = name;
    }
    public String getName() {
        return _name;
    }
    private final String _name;
}
```

它被以下的 `Order` class 使用：

```
class Order...
  public Order (String customerName) {
      _customer = new Customer(customerName);
```

[5]译注：此处之 factory method 不等同于 GoF 在《*Design Patterns*》书中提出的 ***Factory Method***。为避免混淆，读者应该将此处的 factory method 理解为 "Creational Method"，亦即「用以创建某种实体」的函数，这个概念包含 GoF 的 ***Factory Method***，而又比 ***Factory Method*** 意义广泛。

```
    }
    public void setCustomer(String customerName) {
        _customer = new Customer(customerName);
    }
    public String getCustomerName() {
        return _customer.getName();
    }
    private Customer _customer;
```

此外，还有一些代码也会使用 Customer 对象：

```
private static int numberOfOrdersFor(Collection orders,
                                     String customer) {
    int result = 0;
    Iterator iter = orders.iterator();
    while (iter.hasNext()) {
        Order each = (Order) iter.next();
        if (each.getCustomerName().equals(customer)) result++;
    }
    return result;
}
```

到目前为止，Customer 对象还是 *value* object。就算多份定单属于同一客户，但每个 Order 对象还是拥有各自的 Customer 对象。我希望改变这一现状，使得一旦同一客户拥有多份不同定单，代表这些定单的所有 Order 对象就可以共享同一个 Customer 对象。本例中这就意味：每一个客户名称只该对应一个 Customer 对象。

首先我使用 *Replace Constructor with Factory Method*（304）。这样，我就可以控制 Customer 对象的创建过程，这在以后会是非常重要的。我在 Customer class 中定义这个 ***factory method***：

```
class Customer {
    public static Customer create (String name) {
        return new Customer(name);
    }
```

然后我把「对构造函数的调用」替换成「对 ***factory method*** 的调用」：

```
class Order {
    public Order (String customer) {
        _customer = Customer.create(customer);
    }
```

然后我再把构造函数声明为 private：

```
class Customer {
    private Customer (String name) {
        _name = name;
    }
```

现在，我必须决定如何访问 Customer 对象。我比较喜欢通过另一个对象（例如 Order class 中的一个值域）来访问它。但是本例并没有这样一个明显的值域可用于访问 Customer 对象。在这种情况下，我通常会创建一个注册（登录）对象，作为访问点。为了简化我们的例子，我把 Customer 对象保存在 Customer class 的一个 static 值域中，让 Customer class 作为访问点：

```
private static Dictionary _instances = new Hashtable();
```

然后我得决定：应该在接到请求时创建新的 Customer 对象，还是应该预先将它们创建好。这里我选择后者。在应用程序的启动代码（start-up code）中，我先把需要使用的 Customer 对象加载妥当。这些对象可能来自数据库，也可能来自文件。为求简单起见，我在代码中明确生成这些对象。反正以后我总是可以使用 *Substitute Algorithm*（139）来改变它们的创建方式。

```
class Customer...
  static void loadCustomers() {
     new Customer ("Lemon Car Hire").store();
     new Customer ("Associated Coffee Machines").store();
     new Customer ("Bilston Gasworks").store();
  }
  private void store() {
     _instances.put(this.getName(), this);
  }
```

现在，我要修改 ***factory method***，让它返回预先创建好的 Customer 对象：

```
public static Customer create (String name) {
   return (Customer) _instances.get(name);
}
```

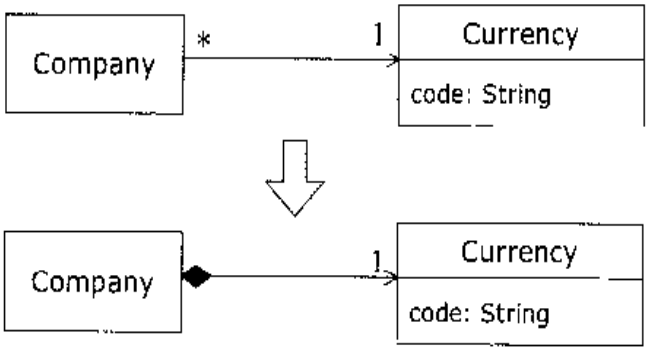
由于 create()总是返回既有的 Customer 对象，所以我应该使用 *Rename Method*（273）修改这个 ***factory method*** 的名称，以便强调（说明）这一点。

```
class Customer...
  public static Customer getNamed (String name) {
     return (Customer) _instances.get(name);
  }
```

8.4 Change Reference to Value（将引用对象改为实值对象）

你有一个 *reference* object（引用对象），很小且不可变（immutable），而且不易管理。

将它变成一个 *value* object（实值对象）。



动机（Motivation）

正如我在 *Change Value to Reference*（179）中所说，要在 *reference* object 和 *value* object 之间做选择，有时并不容易。作出选择后，你常会需要一条回头路。

如果 *reference* object 开始变得难以使用，也许你就应该将它改为 *value* object。*reference* object 必须被某种方式控制，你总是必须向其控制者请求适当的 *reference* object。它们可能造成内存区域之间错综复杂的关联。在分布系统和并发系统中，不可变的 *value* object 特别有用，因为你无须考虑它们的同步问题。

value object 有一个非常重要的特性：它们应该是不可变的（immutable）。无论何时只要你调用同一对象的同一个查询函数，你都应该得到同样结果。如果保证了这一点，就可以放心地以多个对象表示相同事物（same thing）。如果 *value* object 是可变的（mutable），你就必须确保你对某一对象的修改会自动更新其他「代表相同事物」的其他对象。这太痛苦了，与其如此还不如把它变成 *reference* object。

这里有必要澄清一下「不可变（immutable）」的意思。如果你以 Money class 表示「钱」的概念，其中有「币种」和「金额」两条信息，那么 Money 对象通常是一个不可变的 *value* object。这并非意味你的薪资不能改变，而是意味：如果要改变你的薪资，你需要使用另一个崭新的 Money 对象来取代现有的 Money 对象，而不是在现有的 Money 对象上修改。你和 Money 对象之间的关系可以改变，但 Money 对象自身不能改变。

> 译注：《*Practical Java*》by Peter Haggar 第 6 章对于 mutable/immutable 有深入讨论。

作法（Mechanics）

- 检查重构对象是否为 *immutable*（不可变）对象，或是否可修改为不可变对象。
  - ⇨ 如果该对象目前还不是 *immutable*，就使用 *Remove Setting Method*（300），直到它成为 *immutable* 为止。
  - ⇨ 如果无法将该对象修改为 *immutable*，就放弃使用本项重构。
- 建立 equals() 和 hashCode()。
- 编译，测试。
- 考虑是否可以删除 ***factory method***，并将构造函数声明为 public。

范例（Example）

让我们从一个表示「货币种类」的 Currency class 开始：

```
class Currency...
  private String  code;

  public String getCode() {
      return _code;
  }
  private Currency (String code) {
      _code = code;
  }
```

这个 class 所做的就是保存并返回一个货币种类代码。它是一个 *reference* object，所以如果要得到它的一份实体，必须这么做：

```
Currency usd = Currency.get("USD");
```

Currency class 维护一个实体链表（list of instances）；我不能直接使用构造函数创建实体，因为 Currency 构造函数是 private。

```
new Currency("USD").equals(new Currency("USD"))  // returns false
```

要把一个 *reference* object 变成 *value* object，关键动作是：检查它是否为 *immutable*（不可变）。如果不是，我就不能使用本项重构，因为 *mutable*（可变的）*value* object 会造成令人苦恼的别名现象（aliasing）。

在这里，Currency对象是不可变的，所以下一步就是为它定义equals()：

```
public boolean equals(Object arg) {
    if (! (arg instanceof Currency)) return false;
    Currency other = (Currency) arg;
    return (_code.equals(other._code));
}
```

如果我定义equals()，我必须同时定义hashCode()。实现hashCode()有个简单办法：读取equals()使用的所有值域的hash codes，然后对它们进行 *bitwise xor*（^）操作。本例中这很容易实现，因为equals()只使用了一个值域：

```
public int hashCode() {
    return _code.hashCode();
}
```

完成这两个函数后，我可以编译并测试。这两个函数的修改必须同时进行，否则倚赖hashing的任何群集对象（collections，例如Hashtable、HashSet和HashMap）可能会产生意外行为。

现在，我想创建多少个等值的Currency对象就创建多少个。我还可以把构造函数声明为public，直接以构造函数获取Currency实体，从而去掉Currency class中的 ***factory method*** 和「控制实体创建」的行为。

```
new Currency("USD").equals(new Currency("USD"))  // now returns true
```

8.5 Replace Array with Object（以对象取代数组）

你有一个数组（array），其中的元素各自代表不同的东西。

以对象替换数组。对于数组中的每个元素，以一个值域表示之。

```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```

```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

动机（Motivation）

数组（array）是一种常见的用以组织数据的结构体。不过，它们应该只用于「以某种顺序容纳一组相似对象」。有时候你会发现，一个数组容纳了数种不同对象，这会给 array 用户带来麻烦，因为他们很难记住像「数组的第一个元素是人名」这样的约定。对象就不同了，你可以运用值域名称和函数名称来传达这样的信息，因此你无需死记它，也无需倚赖注释。而且如果使用对象，你还可以将信息封装起来，并使用 *Move Method*（142）为它加上相关行为。

作法（Mechanics）

- 新建一个 class 表示数组所示信息，并在该 class 中以一个 public 值域保存原先的数组。
- 修改数组的所有用户，让它们改用新建的 class 实体。
- 编译，测试。
- 逐一为数组元素添加取值/设值函数（getters/setters）。根据元素的用途，为这些访问函数命名。修改客户端代码，让它们通过访问函数取用数组内的元素。每次修改后，编译并测试。

- 当所有「对数组的直接访问」都被取代为「对访问函数的调用」后，将 class 之中保存该数组的值域声明为 private。
- 编译。
- 对于数组内的每一个元素，在新 class 中创建一个型别相当的值域；修改该元素的访问函数，令它改用上述的新建值域。
- 每修改一个元素，编译并测试。
- 数组的所有元素都在对应的 class 内有了相应值域之后，删除该数组。

范例（Example）

我们的范例从一个数组开始，其中有三个元素，分别保存一支球队的名称、获胜场次和失利场次。这个数组的声明可能像这样：

```
String[] row = new String[3];
```

客户端代码可能像这样：

```
row [0] = "Liverpool";
row [1] = "15";

String name = row[0];
int wins = Integer.parseInt(row[1]);
```

为了将数组变成对象，我首先建立一个对应的 class：

```
class Performance {}
```

然后为它声明一个 public 值域，用以保存原先数组。（我知道 public 值域十恶不赦，请放心，稍后我便让它改邪归正。）

```
public String[] _data = new String[3];
```

现在，我要找到创建和访问数组的地方。在创建地点，我将它替换为下列代码：

```
Performance row = new Performance();
```

对于数组使用地点，我将它替换为以下代码：

```
row._data [0] = "Liverpool";
row._data [1] = "15";

String name = row._data[0];
int wins = Integer.parseInt(row._data[1]);
```

然后我要逐一为数组元素加上有意义的取值/设值函数（getters/setters）。首先从「球队名称」开始：

```
class Performance...
  public String getName() {
     return _data[0];
```

```
    }
    public void setName(String arg) {
        _data[0] = arg;
    }
```

然后修改row对象的用户，让他们改用「取值/设值函数」来访问球队名称：

```
row.setName("Liverpool");
row._data[1] = "15";

String name = row.getName();
int wins = Integer.parseInt(row._data[1]);
```

第一个元素也如法炮制。为了简单起见，我还可以把数据型别的转换也封装起来：

```
class Performance...
  public int getWins() {
      return Integer.parseInt(_data[1]);
  }
  public void setWins(String arg) {
      _data[1] = arg;
  }
...
client code...
  row.setName("Liverpool");
  row.setWins("15");

  String name = row.getName();
  int wins = row.getWins();
```

处理完所有元素之后，我就可以将保存该数组的值域声明为 private 了。

```
private String[] _data = new String[3];
```

现在，本次重构最重要的部分（接口修改）已经完成。但是「将对象内的数组替换掉」的过程也同样重要。我可以针对每个数组元素，在 class 内建立一个型别相当的值域，然后修改该数组元素的访问函数，令它直接访问新建值域，从而完全摆脱对数组元素的依赖。

```
class Performance...
  public String getName() {
      return _name;
  }
  public void setName(String arg) {
      _name = arg;
  }
private String _name;
```

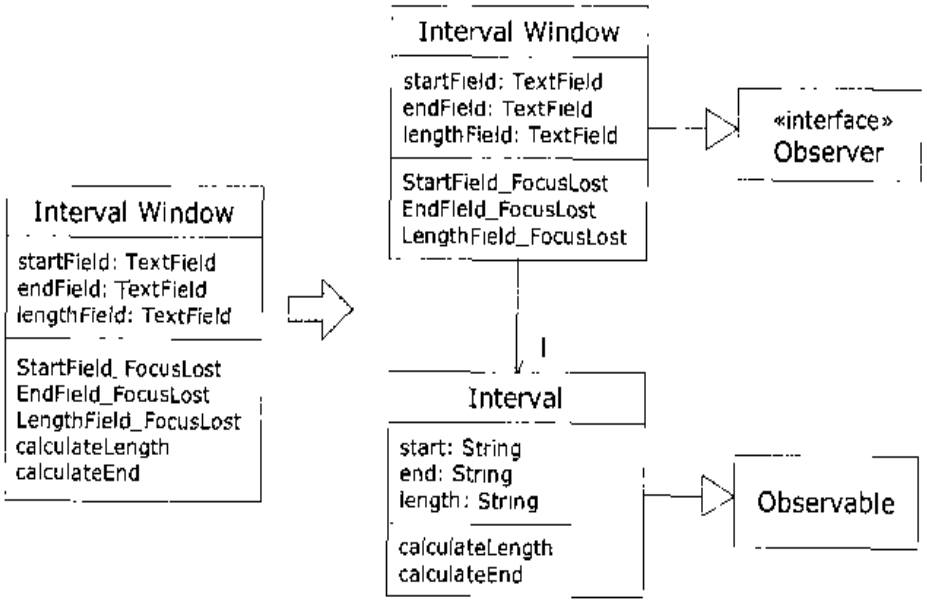
对数组中的每一个元素都如法炮制。全部处理完毕后，我就可以将数组从我的 Performance class 中删掉了。

8.6 Duplicate Observed Data（复制「被监视数据」）

（译注：本节大量保留 domain, presentation, event, getter/setter, observed 等字眼。所谓 presentation class，用以处理「数据表现形式」；所谓 domain class，用以处理业务逻辑。）

你有一些 domain data 置身于 GUI 控件中，而 domain method 需要访问之。

将该笔数据拷贝到一个 domain object 中。建立一个 *Observer* 模式，用以对 domain object 和 GUI object 内的重复数据进行同步控制（sync.）。



动机（Motivation）

一个分层良好的系统，应该将处理用户界面（UI）和处理业务逻辑（business logic）的代码分开。之所以这样做，原因有以下几点：(1) 你可能需要使用数个不同的用户界面来表现相同的业务逻辑；如果同时承担两种责任，用户界面会变得过分复杂；(2) 与 GUI 隔离之后，domain objects 的维护和演化都会更容易；你甚至可以让不同的开发者负责不同部分的开发。

尽管你可以轻松地将「行为」划分到不同部位，「数据」却往往不能如此。同一笔数据有可能既需要内嵌于 GUI 控件，也需要保存于 domain model 里头。自从 MVC（Model-View-Controller）模式出现后，用户界面框架都使用多层系统（multitiered system）来提供某种机制，使你不但可以提供这类数据，并保持它们同步（sync.）。

如果你遇到的代码是以双层（two-tiered）方式开发，业务逻辑（business logic）被内嵌于用户界面（UI）之中，你就有必要将行为分离出来。其中的主要工作就是函

数的分解和搬移。但数据就不同了：你不能仅仅只是移动数据，你必须将它复制到新建部位中，并提供相应的同步机制。

作法（Mechanics）（译注：建议搭配范例阅读）

- 修改 **presentation class**，使其成为 **domain class** 的 ***Observer*** [GoF]。
  - ⇨ 如果尚未有 domain class，就建立一个。
  - ⇨ 如果没有「从 presentation class 到 domain class」的关联性（link），就将 domain class 保存于 presentation class 的一个值域中。
- 针对 GUI class 内的 domain data，使用 *Self Encapsulate Field*（171）。
- 编译，测试。
- 在事件处理函数（event handler）中加上对设值函数（setter）的调用，以「直接访问方式」（译注：亦即直接调用组件提供的相关函数）更新 GUI 组件。
  - ⇨ 在事件处理函数中放一个设值函数（setter），利用它将 GUI 组件更新为 domain data 的当前值。当然这其实没有必要，你只不过是拿它的值设定它自己。但是这样使用 setter，便是允许其中的任何动作得以于日后被执行起来，这是这一步骤的意义所在。
  - ⇨ 进行这个改变时，对于组件，不要使用取值函数（getter），应该采取「直接取用」方式（译注：亦即直接调用 GUI 组件所提供的函数），因为稍后我们将修改取值函数（getter），使其从 domain object（而非 GUI 组件）取值。设值函数（setter）也将遭受类似修改。
  - ⇨ 确保测试代码能够触发新添加的事件处理（event handling）机制。
- 编译，测试。
- 在 domain class 中定义数据及其相关访问函数（accessors）。
  - ⇨ 确保 domain class 中的设值函数（setter）能够触发 ***Observer*** 模式的通报机制（notify mechanism）。
  - ⇨ 对于被观察（被监视）的数据，在 domain class 中使用「与 presentation class 所用的相同型别」（通常是字符串）来保存。后续重构中你可以自由改变这个数据型别。
- 修改 presentation class 中的访问函数（accessors），将它们的操作对象改为 domain object（而非 GUI 组件）。

- 修改observer（译注：亦即presentation class）的`update()`，使其从相应的domain object中将所需数据拷贝给GUI组件。
- 编译，测试。

范例（Example）

我们的范例从图8.1所示窗口开始。其行为非常简单：当用户修改文本框中的数值，另两个文本框就会自动更新。如果你修改**Start**或**End**，**length**就会自动成为两者计算所得的长度；如果你修改**length**，**End**就会随之变动。

一开始，所有函数都放在`IntervalWindow` class中。所有文本框都能够响应「失去键盘焦点」（loss of focus）这一事件。

```
public class IntervalWindow extends Frame...
  java.awt.TextField _startField;
  java.awt.TextField _endField;
  java.awt.TextField _lengthField;

  class SymFocus extends java.awt.event.FocusAdapter
  {
     public void focusLost(java.awt.event.FocusEvent event)
     {
        Object object = event.getSource();
        // 译注：侦测到哪一个文本框失去键盘焦点，就调用其event-handler.
        if (object == _startField)
           StartField_FocusLost(event);
        else if (object == _endField)
           EndField_FocusLost(event);
        else if (object == _lengthField)
           LengthField_FocusLost(event);
     }
  }
```

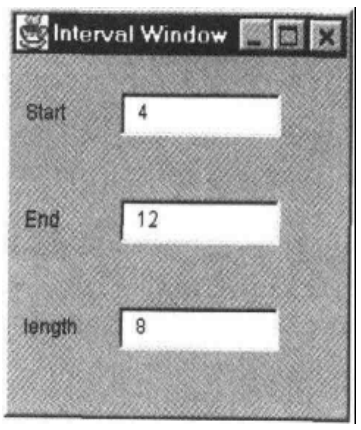


图8.1 一个简单的GUI窗口

当 **Start** 文本框失去焦点，事件监听器调用 StartField_FocusLost()。另两个文本框的处理也类似。事件处理函数大致如下：

```
void StartField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(_startField.getText()))
        _startField.setText("0");
    calculateLength();
}
void EndField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(_endField.getText()))
        _endField.setText("0");
    calculateLength();
}
void LengthField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(_lengthField.getText()))
        _lengthField.setText("0");
    calculateEnd();
}
```

你也许会奇怪，为什么我这样实现一个窗口呢？因为在我的 IDE 集成开发环境（Cafe）中，这是最简单的方式。

如果文本框内的字符串无法转换为一个整数，那么该文本框的内容将变成0。而后，调用相关计算函数：

```
void calculateLength() {
    try {
        int start = Integer.parseInt(_startField.getText());
        int end = Integer.parseInt(_endField.getText());
        int length = end - start;
        _lengthField.setText(String.valueOf(length));
    } catch (NumberFormatException e) {
        throw new RuntimeException ("Unexpected Number Format Error");
    }
}
void calculateEnd() {
    try {
        int start = Integer.parseInt(_startField.getText());
        int length = Integer.parseInt(_lengthField.getText());
        int end = start + length;
        _endField.setText(String.valueOf(end));
    } catch (NumberFormatException e) {
        throw new RuntimeException ("Unexpected Number Format Error");
    }
}
```

我的任务就是将非视觉性的计算逻辑从 GUI 中分离出来。基本上这就意味将 calculateLength()和 calculateEnd()移到一个独立的 domain class 去。为了这

一日的，我需要能够在不引用（指涉，*referring*）窗口类的前提下取用 **Start**、**End** 和 **length** 三个文本框的值。惟一办法就是将这些数据复制到 domain class 中，并保持与 GUI class 数据同步。这就是 *Duplicate Observed Data*（189）的任务。

截至目前我还没有一个 domain class，所以我着手建立一个：

```
class Interval extends Observable {}
```

IntervalWindow class 需要与此崭新的 domain class 建立一个关联：

```
private Interval _subject;
```

然后，我需要合理地初始化 _subject 值域，并把 IntervalWindow class 变成 Interval class 的一个 ***Observer***。这很简单，只需把下列代码放进 IntervalWindow 构造函数中就可以了：

```
_subject = new Interval();
_subject.addObserver(this);
update(_subject, null);
```

我喜欢把这段代码放在整个建构过程的最后。其中对 update() 的调用可以确保：当我把数据复制到 domain class 后，GUI 将根据 domain class 进行初始化。update() 是在 java.util.Observer 接口中声明的，因此我必须让 IntervalWindow class 实现这一接口：

```
public class IntervalWindow extends Frame implements Observer
```

然后我还需要为 IntervalWindow class 建立一个 update()。此刻我先令它为空：

```
public void update(Observable observed, Object arg) {
}
```

现在我可以编译并测试了。到目前为止我还没有做出任何真正的修改。呵呵，小心驶得万年船。

接下来我把注意力转移到文本框。如往常我每次只改动一点点。为了卖弄一下我的英语能力，我从 **End** 文本框开始。第一件要做的事就是实施 *Self Encapsulate Field*（171）。文本框的更新是通过 getText() 和 setText() 两函数实现的，因此我所建立的访问函数（accessors）需要调用这两个函数：

```
// 译注：class IntervalWindow...
String getEnd() {
    return _endField.getText();
}
void setEnd (String arg) {
    _endField.setText(arg);
}
```

然后，找出 _endField 的所有引用点，将它们替换为适当的访问函数：

```
void calculateLength() {
    try {
        int start = Integer.parseInt(_startField.getText());
        int end = Integer.parseInt(getEnd());
        int length = end - start;
        _lengthField.setText(String.valueOf(length));
    } catch (NumberFormatException e) {
        throw new RuntimeException ("Unexpected Number Format Error");
    }
}
void calculateEnd() {
    try {
        int start = Integer.parseInt(_startField.getText());
        int length = Integer.parseInt(_lengthField.getText());
        int end = start + length;
        setEnd(String.valueOf(end));
    } catch (NumberFormatException e) {
        throw new RuntimeException ("Unexpected Number Format Error");
    }
}
void EndField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(getEnd()))
        setEnd("0");
    calculateLength();
}
```

这是 *Self Encapsulate Field*（171）的标准过程。然而当你处理 GUI class 时，情况还更复杂些：用户可以直接（通过 GUI）修改文本框内容，不必调用 setEnd()。因此我需要在 GUI class 的事件处理函数中加上对 setEnd() 的调用。这个动作把 **End** 文本框设定为其当前值。当然，这没带来什么影响，但是通过这样的方式，我们可以确保用户的输入的确是通过设值函数（setter）进行的：

```
void EndField_FocusLost(java.awt.event.FocusEvent event) {
    setEnd(_endField.getText()); //译注：注意以下对此行的讨论
    if (isNotInteger(getEnd()))
        setEnd("0");
    calculateLength();
}
```

上述调用动作中，我并没有使用上一页的 getEnd() 取得 **End** 文本框当前内容，而是直接取用该文本框。之所以这样做是因为，随后的重构将使上一页的 getEnd() 从 domain object（而非文本框）身上取值。那时如果这里用的是 getEnd() 函数，每当用户修改文本框内容，这里就会将文本框又改回原值。所以我必须使用「直接访问文本框」的方式获取当前值。现在我可以编译并测试值域封装后的行为了。

现在，在 domain class 中加入 _end 值域：

```
class Interval...
  private String _end = "0";
```

在这里，我给它的初值和 GUI class 给它的初值是一样的。然后我再加入取值/设值函数（getter/setter）：

```
class Interval...
  String getEnd() {
      return _end;
  }
  void setEnd (String arg) {
      _end = arg;
      setChanged();
      notifyObservers();  //译注：notification code
  }
```

由于使用了 ***Observer*** 模式，我必须在设值函数（setter）中加上「发出通告」动作（即所谓 notification code）。我把 _end 声明为一个字符串，而不是一个看似更合理的整数，这是因为我希望将修改量减至最少。将来成功复制数据完毕后，我可以自由自在地于 domain class 内部把 _end 声明为整数。

现在，我可以再编译并测试一次。我希望通过所有这些预备工作，将下面这个较为棘手的重构步骤的风险降至最低。

首先，修改 IntervalWindow class 的访问函数，令它们改用 Interval 对象：

```
class IntervalWindow...
  String getEnd() {
      return _subject.getEnd();
  }
  void setEnd (String arg) {
      _subject.setEnd(arg);       //(A) 译注：本页最下对此行有些说明
  }
```

同时也修改 update() 函数，确保 GUI 对 Interval 对象发来的通告做出响应：

```
class IntervalWindow...
  public void update(Observable observed, Object arg) {
      _endField.setText(_subject.getEnd());
  }
```

这是另一个需要「直接取用文本框」的地点。如果我调用的是设值函数（setter），程序将陷入无限递归调用（译注：这是因为 IntervalWindow 的设值函数 setEnd() 调用了 Interval.setEnd()，一如稍早(A)行所示；而 Interval.setEnd() 又调用 notifyObservers()，导致 IntervalWindow.update() 又被调用）。

现在，我可以编译并测试。数据都恰如其分地被复制了。

另两个文本框也如法炮制。完成之后，我可以使用 *Move Method*（142）将`calculateEnd()`和`calculateLength()`搬到`Interval` class 去。这么一来，我就拥有一个「包容所有 domain behavior 和 domain data」并与 GUI code 分离的 domain class 了。

如果上述工作都完成了，我就会考虑彻底摆脱这个 GUI class。如果 GUI class 是个较为老旧的 AWT class，我会考虑将它换成一个比较好看的 Swing class，而且后者的坐标定位能力也比较强。我可以在 domain class 之上建立一个 Swing GUI。这样，只要我高兴，随时可以去掉老旧的 GUI class。

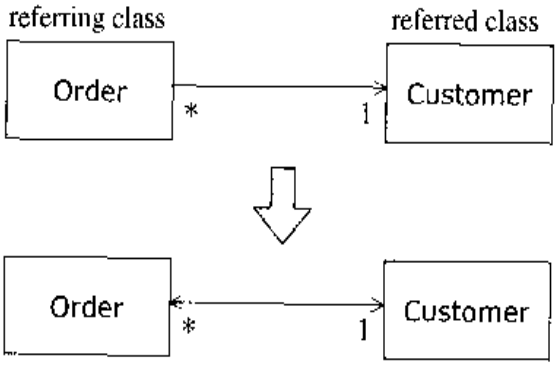
使用事件监听器（Event Listeners）

如果你使用事件监听器（event listener）而不是 ***Observer/Observable*** 模式，仍然可以实施 *Duplicate Observed Data*（189）。这种情况下，你需要在 domain model 中建立一个 **listener** class 和一个 **event** class（如果你不在意依存关系的话，也可以使用 AWT classes）。然后，你需要对 domain object 注册 listeners，就像前例对 observable 对象注册 observers 一样。每当 domain object 发生变化（类似上例的`update()`函数被调用），就向 listeners 发送一个事件（event）。`IntervalWindow` class 可以利用一个 inner class（内嵌类）来实现监听器接口（listener interface），并在适当时候调用适当的`update()`函数。

8.7 Change Unidirectional Association to Bidirectional
将单向关联改为双向

两个 classes 都需要使用对方特性，但其间只有一条单向连接（one-way link）。

添加一个反向指针，并使修改函数（modifiers）能够同时更新两条连接。（译注：这里的指针等同于句柄（handle），修改函数（modifier）指的是改变双方关系者）



动机（Motivation）

开发初期，你可能会在两个 classes 之间建立一条单向连接，使其中一个 class 可以引用另一个 class。随着时间推移，你可能发现 referred class 需要得到其引用者（某个 object）以便进行某些处理，也就是说它需要一个反向指针。但指针乃是一种单向连接，你不可能反向操作它。通常你可以绕道而行，虽然会耗费一些计算时间，成本还算合理，然后你可以在 referred class 中建立一个专职函数，负责此一行为。但是，有时候，想绕过这个问题并不容易，此时你就需要建立双向引用关系（two-way reference），或称为反向指针（back pointer）。如果你不习惯使用反向指针，它们很容易造成混乱；但只要你习惯了这种手法，它们其实并不是太复杂。

「反向指针」手法有点棘手，所以在你能够自在运用它之前，应该有相应的测试。通常我不花心思去测试访问函数（accessors），因为普通访问函数的风险没有高到需要测试的地步，但本重构要求测试访问函数，所以它是极少数需要添加测试的重构手法之一。

本重构运用反向指针（back pointer）实现双向关联（bidirectionality）。其他技术（例如连接对象，link objects）需要其他重构手法。

作法（Mechanics）

- 在 referred class 中增加一个值域，用以保存「反向指针」。
- 决定由哪个 class（引用端或被引用端）控制关联性（association）。

- 在「被控端」建立一个辅助函数，其命名应该清楚指出它的有限用途。
- 如果既有的修改函数（modifier）在「控制端」，让它负责更新反向指针。
- 如果既有的修改函数（modifier）在「被控端」，就在「控制端」建立一个控制函数，并让既有的修改函数调用这个新建的控制函数。

范例（Example）

下面是一段简单程序，其中有两个 classes：表示「定单」的 Order 和表示「客户」的 Customer。Order 引用了 Customer，Customer 则并没有引用 Order：

```
class Order...
  Customer getCustomer() {
     return _customer;
  }
  void setCustomer (Customer arg) {
     _customer = arg;
  }
  Customer _customer; //译注：这是一个 "order" to "customer" 的连接
```

首先，我要为 Customer 添加一个值域。由于一个客户可以拥有多份定单，所以这个新增值域应该是个群集（collection）。我不希望同一份定单在同一个群集中出现一次以上，所以这里适合使用 set：

```
class Customer {
    private Set _orders = new HashSet();
```

现在，我需要决定由哪一个 class 负责控制关联性（association）。我比较喜欢让单一 class 来操控，因为这样我就可以将所有「关联处理逻辑」集中安置于一地。我将按照下列步骤做出这一决定：

1. 如果两者都是 *reference* objects，而其间的关联是「一对多」关系，那么就由「拥有单一 reference」的那一方承担「控制者」角色。以本例而言，如果一个客户可拥有多份定单，那么就由 Order class（定单）来控制关联性。
2. 如果某个对象是另一对象的组成（component），那么由后者负责控制关联性。
3. 如果两者都是 *reference* objects，而其间的关联是「多对多」关系，那么随便其中哪个对象来控制关联性，都无所谓。

本例之中由于 Order 负责控制关联性，所以我必须为 Customer 添加一个辅助函数，让 Order 可以直接访问 _orders（订单）群集。Order 的修改函数（modifier）将使用这个辅助函数对指针两端对象进行同步控制。我将这个辅助函数命名为 friendOrders()，表示这个函数只能在这种特殊情况下使用。此外，如果 Order 和 Customer 位在同一个 package 内，我还会将 friendOrders() 声明为「package 可见度」（译注：亦即不加任何修饰符的缺省访问级别），使其可见程度降到最低。

但如果这两个 classes 不在同一个 package 内，我就只好把 friendOrders()声明为 public 了。

```
class Customer...
  Set friendOrders() {
  /** should only be used by Order when modifying the association */
     return _orders;
  }
```

现在，我要改变修改函数（modifier），令它同时更新反向指针：

```
class Order...
  void setCustomer (Customer arg) ...
     if (_customer != null) _customer.friendOrders().remove(this);
     _customer = arg;
     if (_customer != null) _customer.friendOrders().add(this);
  }
```

classes 之间的关联性是各式各样的，因此修改函数（modifier）的代码也会随之有所差异。如果_customer 的值不可能是 null，我可以拿掉上述的第一个 null 检查，但仍然需要检查引数（argument）是否为 null。不过，基本形式总是相同的：先让对方删除「指向你」的指针，再将你的指针指向一个新对象，最后让那个新对象把它的指针指向你。

如果你希望在 Customer 中也能修改连接（link），就让它调用控制函数：

```
class Customer...
  void addOrder(Order arg) {
     arg.setCustomer(this);
  }
```

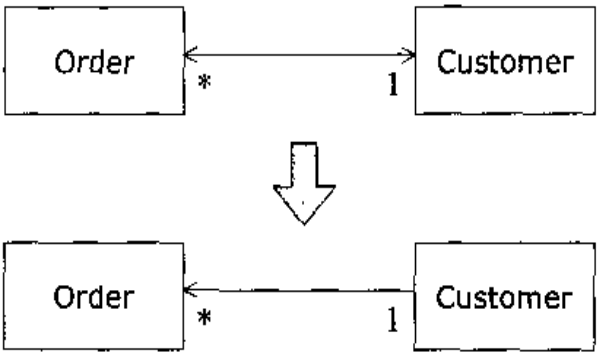
如果一份定单也可以对应多个客户，那么你所面临的就是一个「多对多」情况，重构后的函数可能是下面这样：

```
class Order... // controlling methods
  void addCustomer (Customer arg) {
     arg.friendOrders().add(this);
     _customers.add(arg);
  }
  void removeCustomer (Customer arg) {
     arg.friendOrders().remove(this);
     _customers.remove(arg);
  }
  class Customer...
    void addOrder(Order arg) {
       arg.addCustomer(this);
    }
    void removeOrder(Order arg) {
       arg.removeCustomer(this);
    }
```

8.8 Change Bidirectional Association to Unidirectional
将双向关联改为单向

两个 classes 之间有双向关联，但其中一个 class 如今不再需要另一个 class 的特性。

去除不必要的关联（association）。



动机（Motivation）

双向关联（bidirectional associations）很有用，但你也必须为它付出代价，那就是「维护双向连接、确保对象被正确创建和删除」而增加的复杂度。而且，由于很多程序员并不习惯使用双向关联，它往往成为错误之源。

大量的双向连接（two-way links）也很容易引发「僵尸对象」：某个对象本来已经该死亡了，却仍然保留在系统中，因为对它的各项引用还没有完全清除。

此外，双向关联也迫使两个 classes 之间有了相依性。对其中任一个 class 的任何修改，都可能引发另一个 class 的变化。如果这两个 classes 处在不同的 package 中，这种相依性就是 packages 之间的相依。过多的依存性（inter-dependencies）会造就紧耦合（highly coupled）系统，使得任何一点小小改动都可能造成许多无法预知的后果。

只有在你需要双向关联的时候，才应该使用它。如果你发现双向关联不再有存在价值，就应该去掉其中不必要的一条关联。

作法（Mechanics）

- 找出「你想去除的指针」的保存值域，检查它的每一个用户，判断是否可以去除该指针。

⇨ 不但要检查「直接读取点」，也要检查「直接读取点」的调用函数。

⇨ 考虑有无可能不通过指针取得「被引用对象」（referred object）。如果有可能，你就可以对取值函数（getter）使用 *Substitute Algorithm*（139），从而让客户在没有指针的情况下也可以使用该取值函数。

⇨ 对于使用该值域的所有函数，考虑将「被引用对象」（referred object）作为引数（argument）传进去。

□ 如果客户使用了取值函数（getter），先运用 *Self Encapsulate Field*（171）将「待除值域」自我封装起来，然后使用 *Substitute Algorithm*（139）对付取值函数，令它不再使用该（待除）值域。然后编译、测试。

□ 如果客户并未使用取值函数（getter），那就直接修改「待除值域」的所有被引用点：改以其他途径获得该值域所保存的对象。每次修改后，编译并测试。

□ 如果已经没有任何函数使用该（待除）值域，移除所有「对该值域的更新逻辑」，然后移除该值域。

⇨ 如果有许多地方对此值域赋值，先运用 *Self Encapsulate Field*（171）使这些地点改用同一个设值函数（setter）。编译、测试。而后将这个设值函数的本体清空。再编译、再测试。如果这些都可行，就可以将此值域和其设值函数，连同对设值函数的所有调用，全部移除。

□ 编译，测试。

范例（Example）

本例从 *Change Unidirectional Association to Bidirectional*（197）留下的代码开始进行，其中 `Customer` 和 `Order` 之间有双向关联：

```
class Order...
  Customer getCustomer() {
      return _customer;
  }
  void setCustomer (Customer arg) {
      if (_customer != null) _customer.friendOrders().remove(this);
      _customer = arg;
      if (_customer != null) _customer.friendOrders().add(this);
  }
  private Customer _customer;   //译注：这是 Order-to-Customer link
                                //      也是本例的移除对象
class Customer...
  void addOrder(Order arg) {
      arg.setCustomer(this);
  }
  private Set _orders = new HashSet();
```

```
        //译注：以上是 Customer-to-Order link
    Set friendOrders() {
        /** should only be used by Order */
        return _orders;
    }
```

后来我发现，除非先有 Customer 对象，否则不会存在 Order 对象。因此我想将「从 Order 到 Customer 的连接」移除掉。

对于本项重构来说，最困难的就是检查可行性。如果我知道本项重构是安全的，那么重构手法自身十分简单。问题在于是否有任何代码倚赖 _customer 值域的存在。如果确实有，那么在删除这个值域之后，我必须提供替代品。

首先，我需要研究所有读取这个值域的函数，以及所有使用这些函数的函数。我能找到另一条途径来供应 Customer 对象吗——这通常意味将 Customer 对象作为引数（argument）传递给其用户（某函数）。下面是一个简化例子：

```
class Order...
  double getDiscountedPrice() {
      return getGrossPrice() * (1 - _customer.getDiscount());
  }
```

改变为：

```
class Order...
  double getDiscountedPrice(Customer customer) {
      return getGrossPrice() * (1 - customer.getDiscount());
  }
```

如果待改函数是被 Customer 对象调用的，那么这样的修改方案特别容易实施，因为 Customer 对象将自己作为引数（argument）传给函数很是容易。所以下列代码：

```
class Customer...
  double getPriceFor(Order order) {
      Assert.isTrue(_orders.contains(order)); //see Introduce Assertion (267)
      return order.getDiscountedPrice();
```

变成了：

```
class Customer...
  double getPriceFor(Order order) {
      Assert.isTrue(_orders.contains(order));
      return order.getDiscountedPrice(this);
  }
```

另一种作法就是修改取值函数（getter），使其在不使用 _customer 值域的前提下返回一个 Customer 对象。如果这行得通，我就可以使用 *Substitute Algorithm*(139) 修改 Order.getCustomer()函数算法。我有可能这样修改代码：

```
Customer getCustomer() {
  Iterator iter = Customer.getInstances().iterator();
  while (iter.hasNext()) {
      Customer each = (Customer)iter.next();
```

```
        if (each.containsOrder(this)) return each;
    }
    return null;
}
```

这段代码比较慢，不过确实可行。而且，在数据库环境下，如果我需要使用数据库查询语句，这段代码对系统性能的影响可能并不显著。如果 Order class 中有些函数使用 _customer 值域，我可以实施 *Self Encapsulate Field*（171）令它们转而改用上述的 getCustomer()函数。

如果我要保留上述的取值函数（getter），那么 Order 和 Customer 的关联从接口上看虽然仍是双向，但实现上已经是单向关系了。虽然我移除了反向指针，但两个 classes 彼此之间的依存关系（inter-dependencies）仍然存在。

如果我要替换取值函数（getter），那么我就专注地替换它，其他部分留待以后处理。我会逐一修改取值函数的调用者，让它们通过其他来源取得 Customer 对象。每次修改后都编译并测试。实际工作中这一过程往往相当快。如果这个过程让我觉得很棘手很复杂，我会放弃本项重构。

一旦我消除了_customer 值域的所有读取点，我就可以着手处理「对此值域进行赋值动作」的函数了（译注：亦即设值函数，setter）。很简单，只要把这些赋值动作全部移除，再把值域一并删除，就行了。由于已经没有任何代码需要这个值域，所以删掉它并不会带来任何影响。

8.9 Replace Magic Number with Symbolic Constant

以符号常量/字面常量取代魔法数

你有一个字面数值（literal number），带有特别含义。

创造一个常量，根据其意义为它命名，并将上述的字面数值替换为这个常量。

```
double potentialEnergy(double mass, double height) {
    return mass * 9.81 * height;
}
```

```
double potentialEnergy(double mass, double height) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

动机（Motivation）

在计算科学中，魔法数（magic number）是历史最悠久的不良现象之一。所谓魔法数是指拥有特殊意义，却又不能明确表现出这种意义的数字。如果你需要在不同的地点引用同一个逻辑数，魔法数会让你烦恼不已，因为一旦这些数发生改变，你就必须在程序中找到所有魔法数，并将它们全部修改一遍，这简直就是一场噩梦。就算你不需要修改，要准确指出每个魔法数的用途，也会让你颇费脑筋。

许多语言都允许你声明常量。常量不会造成任何性能开销，却可以大大提高代码的可读性。

进行本项重构之前，你应该先寻找其他替换方案。你应该观察魔法数如何被使用，而后往往你会发现一种更好的使用方式。如果这个魔法数是个 type code（型别码），请考虑使用 *Replace Type Code with Class*（218）；如果这个魔法数代表一个数组的长度，请在遍历该数组的时候，改用 `Array.length()`。

作法（Mechanics）

- 声明一个常量，令其值为原本的魔法数值。
- 找出这个魔法数的所有引用点。
- 检查是否可以使用这个新声明的常量来替换该魔法数。如果可以，便以此一常量替换之。
- 编译。
- 所有魔法数都被替换完毕后，编译并测试。此时整个程序应该运转如常，就像没有做任何修改一样。

  ⇨ 有个不错的测试办法：检查现在的程序是否可以被你轻松地修改常量值（这可能意味某些预期结果将有所改变，以配合这一新值。实际工作中并非总是可以进行这样的测试）。如果可行，这就是一个不错的手法。

8.10 Encapsulate Field（封装值域）

你的 class 中存在一个 public 值域。

将它声明为 private，并提供相应的访问函数（accessors）。

```
public String _name
```

```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

动机（Motivation）

面向对象的首要原则之一就是封装（encapsulation），或者称为「数据隐藏」（data hidding）。按此原则，你绝不应该将数据声明为 public，否则其他对象就有可能访问甚至修改这项数据，而拥有该数据的对象却毫无察觉。这就将数据和行为分开了（不妙）。

public 数据被看做是一种不好的作法，因为这样会降低程序的模块化程度（modularity）。如果数据和使用该数据的行为被集中在一起，一旦情况发生变化，代码的修改就会比较简单，因为需要修改的代码都集中于同一块地方，而不是星罗棋布地散落在整个程序中。

Encapsulate Field（206）是封装过程的第一步。通过这项重构手法，你可以将数据隐藏起来，并提供相应的访问函数（accessors）。但它毕竟只是第一步。如果一个 class 除了访问函数（accessors）外不能提供其他行为，它终究只是一个 dumb class（哑类）。这样的 class 并不能获得对象技术的优势，而你知道，浪费任何一个对象都是很不好的。实施 *Encapsulate Field*（206）之后，我会尝试寻找那些使用「新建访问函数」的函数，看看是否可以通过简单的 *Move Method*（142）轻快地将它们移到新对象去。

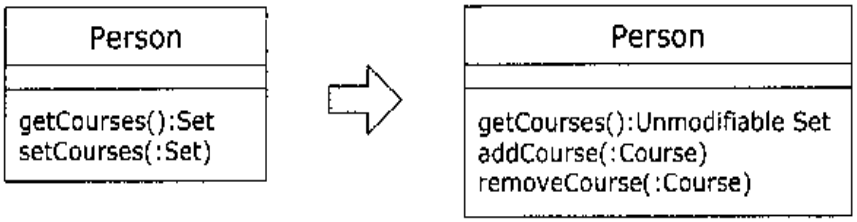
作法（Mechanics）

- 为 public 值域提供取值/设值函数（getter/setter）。
- 找到这个 class 以外使用该值域的所有地点。如果客户只是使用该值域，就把引用动作（reference）替换为「对取值函数（getter）的调用」；如果客户修改了该值域值，就将此一引用点替换为「对设值函数（setter）的调用」。
  - ⇨ 如果这个值域是个对象，而客户只不过是调用该对象的某个函数，那么无论该函数是否为修改函数（modifier，会改变对象状态），都只能算是使用该值域。只有当客户为该值域赋值时，才能将其替换为设值函数（setter）。
- 每次修改之后，编译并测试。
- 将值域的所有用户修改完毕后，把值域声明为 private。
- 编译，测试。

8.11 Encapsulate Collection（封装群集）

有个函数（method）返回一个群集（collection）。

让这个函数返回该群集的一个只读映件（read-only view），**并在这个** class **中提供「添加/移除」**（add/remove）**群集元素的函数。**



动机（Motivation）

class 常常会使用群集（collection，可能是 array、list、set 或 vector）来保存一组实体。这样的 class 通常也会提供针对该群集的「取值/设值函数」（getter/setter）。

但是，群集的处理方式应该和其他种类的数据略有不同。取值函数（getter）不该返回群集自身，因为这将让用户得以修改群集内容而群集拥有者却一无所悉。这也会对用户暴露过多「对象内部数据结构」的信息。如果一个取值函数（getter）确实需要返回多个值，它应该避免用户直接操作对象内所保存的群集，并隐藏对象内「与用户无关」的数据结构。至于如何做到这一点，视你使用的 Java 版本不同而有所不同。

另外，不应该为这整个群集提供一个设值函数（setter），但应该提供用以为群集添加/移除（add/remove）元素的函数。这样，群集拥有者（对象）就可以控制群集元素的添加和移除。

如果你做到以上数点，群集（collection）就被很好地封装起来了，这便可以降低群集拥有者（class）和用户之间的耦合度。

作法（Mechanics）

- 加入「为群集添加（add）、移除（remove）元素」的函数。
- 将「用以保存群集」的值域初始化为一个空群集。
- 编译。

- 找出「群集设值函数」的所有调用者。你可以修改那个设值函数，让它使用上述新建立的「添加/移除元素」函数；也可以直接修改调用端，改让它们调用上述新建立的「添加/移除元素」函数。
  - ⇨ 两种情况下需要用到「群集设值函数」：(1) 群集为空时；(2) 准备将原有群集替换为另一个群集时。
  - ⇨ 你或许会想运用 *Rename Method*（273）为「群集设值函数」改名，从 `setXxx()` 改为 `initializeXxx()` 或 `replaceXxx()`。
- 编译，测试。
- 找出所有「通过取值函数（getter）获得群集并修改其内容」的函数。逐一修改这些函数，让它们改用「添加/移除」（add/remove）函数。每次修改后，编译并测试。
- 修改完上述所有「通过取值函数（getter）获得群集并修改群集内容」的函数后，修改取值函数自身，使它返回该群集的一个只读映件（read-only view）。
  - ⇨ 在 Java 2 中，你可以使用 `Collection.unmodifiableXxx()` 得到该群集的只读映件。
  - ⇨ 在 Java 1.1 中，你应该返回群集的一份拷贝。
- 编译，测试。
- 找出取值函数（getter）的所有用户，从中找出应该存在于「群集之宿主对象（host object）」内的代码。运用 *Extract Method*（110）和 *Move Method*（142）将这些代码移到宿主对象去。

如果你使用 Java 2，那么本项重构到此为止。如果你使用 Java 1.1，那么用户也许会喜欢使用枚举（enumeration）。为了提供这个枚举，你应该这样做：

- 修改现有取值函数（getter）的名字，然后添加一个新取值函数，使其返回一个枚举。找出旧取值函数的所有被使用点，将它们都改为使用新取值函数。
  - ⇨ 如果这一步跨度太大，你可以先使用 *Rename Method*（273）修改原取值函数的名称；再建立一个新取值函数用以返回枚举；最后再修改所有调用者，使其调用新取值函数。
- 编译，测试。

范例（Example）

Java 2 拥有一组全新群集（collections）——并非仅仅加入一些新 classes，而是完全改变了群集的风格。所以在 Java 1.1 和 Java 2 中，封装群集的方式也完全不同。我首先讨论 Java 2 的方式，因为我认为功能更强大的 Java 2 collections 会取代 Java 1.1 collections 的地位。

范例（Example）：Java 2

假设有个人要去上课。我们用一个简单的 Course 来表示「课程」：

```
class Course...
  public Course (String name, boolean isAdvanced) {...};
  public boolean isAdvanced() {...};
```

我不关心课程其他细节。我感兴趣的是表示「人」的 Person：

```
class Person...
  public Set getCourses() {
     return _courses;
  }
  public void setCourses(Set arg) {
     _courses = arg;
  }
  private Set _courses;
```

有了这个接口，我们就可以这样为某人添加课程：

```
Person kent = new Person();
Set s = new HashSet();
s.add(new Course ("Smalltalk Programming", false));
s.add(new Course ("Appreciating Single Malts", true));
kent.setCourses(s);
Assert.equals (2, kent.getCourses().size());

Course refact = new Course ("Refactoring", true);
kent.getCourses().add(refact);
kent.getCourses().add(new Course ("Brutal Sarcasm", false));
Assert.equals (4, kent.getCourses().size());

kent.getCourses().remove(refact);
Assert.equals (3, kent.getCourses().size());
```

如果想了解高级课程，可以这么做：

```
Iterator iter = person.getCourses().iterator();
int count = 0;
while (iter.hasNext()) {
    Course each = (Course) iter.next();
    if (each.isAdvanced()) count ++;
}
```

我要做的第一件事就是为 Person 中的群集（collections）建立合适的修改函数（modifiers，亦即 add/remove 函数），如下所示，然后编译：

```
class Person...
  public void addCourse (Course arg) {
    _courses.add(arg);
  }
  public void removeCourse (Course arg) {
    _courses.remove(arg);
  }
```

如果我像下面这样初始化 _courses 值域，我的人生会轻松得多：

```
private Set _courses = new HashSet();
```

接下来我需要观察设值函数（setter）的调用者。如果有许多地点大量运用了设值函数，我就需要修改设值函数，令它调用添加/移除（add/remove）函数。这个过程的复杂度取决于设值函数的被使用方式。设值函数的用法有两种，最简单的情况就是：它被用来「对群集进行初始化动作」。换句话说，设值函数被调用之前，_courses 是个空群集。这种情况下我只需修改设值函数，令它调用添加函数（add）就行了：

```
class Person...
  public void setCourses(Set arg) {
    Assert.isTrue(_courses.isEmpty());
    Iterator iter = arg.iterator();
    while (iter.hasNext()) {
        addCourse((Course) iter.next());
    }
  }
```

修改完毕后，最好以 *Rename Method*（273）更明确地展示这个函数的意图。

```
public void initializeCourses(Set arg) {
    Assert.isTrue(_courses.isEmpty());
    Iterator iter = arg.iterator();
    while (iter.hasNext()) {
      addCourse((Course) iter.next());
    }
}
```

更普通（译注：而非上述所言对「空群集」设初值）的情况下，我必须首先以移除函数（remove）将群集中的所有元素全部移除，然后再调用添加函数（add）将元素一一添加进去。不过我发现这种情况很少出现（啊，愈是普通的情况，愈少出现）。

如果我知道初始化时，除了添加元素，不会再有其他行为，那么我可以不使用循环，直接调用addAll()函数：

```
public void initializeCourses(Set arg) {
    Assert.isTrue(_courses.isEmpty());
    _courses.addAll(arg);
}
```

我不能仅仅对这个set赋值，就算原本这个set是空的也不行。因为万一用户在「把set传递给Person对象」之后又去修改它，会破坏封装。我必须像上面那样创建set的一个拷贝。

如果用户仅仅只是创建一个set，然后使用设值函数（setter。译注：目前已改名为initializeCourses()），我可以让它们直接使用添加/移除（add/remove）函数，并将设值函数完全移除。于是，以下代码：

```
Person kent = new Person();
Set s = new HashSet();
s.add(new Course ("Smalltalk Programming", false));
s.add(new Course ("Appreciating Single Malts", true));
kent.initializeCourses(s);      //译注：setter
```

就变成了：

```
Person kent = new Person();
kent.addCourse(new Course ("Smalltalk Programming", false));
kent.addCourse(new Course ("Appreciating Single Malts", true));
```

接下来我开始观察取值函数（getter）的使用情况。首先处理「有人以取值函数修改底部群集（underlying collection）」的情况，例如：

```
kent.getCourses().add(new Course ("Brutal Sarcasm", false));
```

这种情况下我必须加以改变，使它调用新的修改函数（modifier）：

```
kent.addCourse(new Course ("Brutal Sarcasm", false));
```

修改完所有此类情况之后，我可以让取值函数（getter）返回一个只读映件（read-only view），用以确保没有任何一个用户能够通过取值函数（getter）修改群集：

```
public Set getCourses() {
    return Collections.unmodifiableSet(_courses);
}
```

这样我就完成了对群集的封装。此后，不通过Person提供的add/remove函数，谁也不能修改群集内的元素。

将行为移到这个 class 中

我拥有了合理的接口。现在开始观察取值函数（getter）的用户，从中找出应该属于 Person 的代码。下面这样的代码就应该搬移到 Person 去：

```
iterator iter = person.getCourses().iterator(); //译注：user of getter
int count = 0;
while (iter.hasNext()) {
    Course each = (Course) iter.next();
    if (each.isAdvanced()) count++;
}
```

因为以上只使用了属于 Person 的数据。首先我使用 *Extract Method*（110）将这段代码提炼为一个独立函数：

```
int numberOfAdvancedCourses(Person person) {
    Iterator iter = person.getCourses().iterator();
    int count = 0;
    while (iter.hasNext()) {
      Course each = (Course) iter.next();
      if (each.isAdvanced()) count++;
    }
    return count;
}
```

然后使用 *Move Method*（142）将这个函数搬移到 Person 中：

```
class Person...
    int numberOfAdvancedCourses() {
      Iterator iter = getCourses().iterator();
      int count = 0;
      while (iter.hasNext()) {
          Course each = (Course) iter.next();
          if (each.isAdvanced()) count++;
      }
      return count;
    }
```

举个常见例子，下列代码：

```
kent.getCourses().size()
```

可以修改成更具可读性的样子，像这样：

```
kent.numberOfCourses()
```

```
class Person...
public int numberOfCourses() {
    return _courses.size();
}
```

数年以前，我曾经担心将这样的行为搬移到 Person 中会导致 Person 变得臃肿。但是在实际工作经验中，我发现这通常并不成为问题。

范例：Java 1.1

在很多地方，Java 1.1 的情况和 Java 2 非常相似。这里我使用同一个范例，不过群集改为 vector（译注：因为 vector 属于 Java 1.1，不属于 Java 2）：

```
class Person...
  public Vector getCourses() {
    return _courses;
  }
  public void setCourses(Vector arg) {
    _courses = arg;
  }
  private Vector _courses;
```

同样地，我首先建立修改函数（modifiers；add/remove 函数），并初始化 _courses 值域，如下所示：

```
class Person...
  public void addCourse(Course arg) {
    _courses.addElement(arg);
  }
  public void removeCourse(Course arg) {
    _courses.removeElement(arg);
  }
  private Vector _courses = new Vector();
```

我可以修改 setCourses() 来初始化这个 vector：

```
public void initializeCourses(Vector arg) {
    Assert.isTrue(_courses.isEmpty());
    Enumeration e = arg.elements();
    while (e.hasMoreElements()) {
      addCourse((Course) e.nextElement());
    }
}
```

然后，我修改取值函数（getter）调用点，让它们改用新建的修改函数（modifiers）。于是下列代码：

```
kent.getCourses().addElement(new Course ("Brutal Sarcasm", false));
```

就变成了：

```
kent.addCourse(new Course ("Brutal Sarcasm", false));
```

最后一步需要有点改变，因为 Java 1.1 的 Vector class 并没有提供「不可修改版」（unmodifiable version）：

```
class Person...
    Vector getCourses() {
      return (Vector) _courses.clone();
    }
```

这样便完成了群集的封装。此后，如果不通过 Person 提供的函数，谁也不能改变群集的元素。

范例：封装数组（Encapsulating Arrays）

数组（array）很常被使用，特别是对于那些不熟悉群集（collections）的程序员而言。我很少使用数组，因为我更喜欢功能更加丰富的群集类。进行封装时，我常把数组换成其他群集。

这次我们的范例从一个字符串数组（string array）开始：

```
String[] getSkills() {
    return _skills;
}
void setSkills (String[] arg) {
     skills = arg;
}
String[] _skills;
```

同样地，首先我要提供一个修改函数（modifier）。由于用户有可能修改数组中某一特定位置上的值，所以我提供的 setSkill() 必须能对任何特定位置上的元素赋值：

```
void setSkill(int index, String newSkill) {
    _skills[index] = newSkill;
}
```

如果我需要对整个数组赋值，可以使用下列函数：

```
void setSkills (String[] arg) {
    _skills = new String[arg.length];
    for (int i=0; i < arg.length; i++)
        setSkill(i,arg[i]);
}
```

如果需要处理「被移除元素」（removed elements），就会有些困难。如果作为引数（argument）的数组和原数组长度不同，情况也会比较复杂。这也是我优先选择群集的原因之一。

现在，我需要观察取值函数（getter）的调用者，我可以把下列代码：

```
kent.getSkills()[2] = "Refactoring";
```

改成：

```
kent.setSkill(1,"Refactoring");
```

完成这一系列修改之后，我可以修改取值函数（getter），令它返回一份数组拷贝：

```
String[] getSkills() {
    String[] result = new String[_skills.length];
    System.arraycopy(_skills, 0, result, 0, _skills.length);
    return result;
}
```

现在，是把数组换成 list 的时候了：

```
class Person...
  String[] getSkills() {
     return (String[]) _skills.toArray(new String[0]);
  }
  void setSkill(int index, String newSkill) {
    _skills.set(index,newSkill);
  }
  List _skills = new ArrayList();
```

8.12 Replace Record with Data Class
以数据类取代记录

你需要面对传统编程环境中的 record structure（记录结构）。

为该 record（记录）创建一个「哑」数据对象（dumb data object）。

动机（Motivation）

record structures（记录型结构）是许多编程环境的共同性质。有一些理由使它们被带进面向对象程序之中：你可能面对的是一个老旧程序（legacy program），也可能需要通过一个传统 API 来与 structured record 交流，或是处理从数据库读出的 records。这些时候你就有必要创建一个 interfacing class，用以处理这些外来数据。最简单的作法就是先建立一个看起来类似外部记录（external record）的 class，以便日后将某些值域和函数搬移到这个 class 之中。一个不太常见但非常令人注目的情况是：数组中的每个位置上的元素都有特定含义，这种情况下你应该使用 *Replace Array with Object*（186）。

作法（Mechanics）

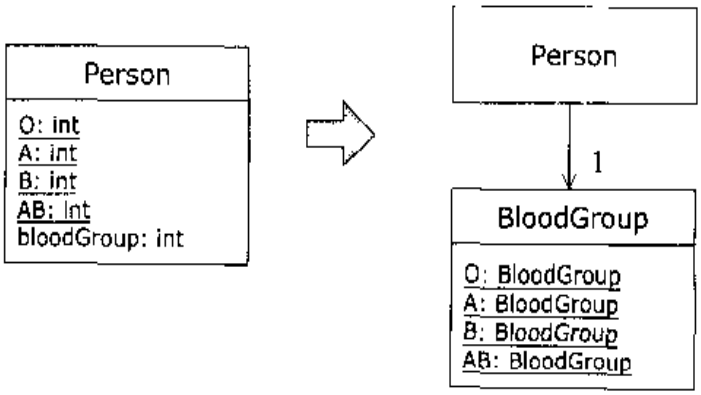
- 新建一个 class，表示这个 record。
- 对于 record 中的每一笔数据项，在新建的 class 中建立对应的一个 private 值域，并提供相应的取值/设值函数（getter/setter）。

现在，你拥有了一个「哑」数据对象（dumb data object）。这个对象现在还没有任何有用行为（函数），但是更进一步的重构会解决这个问题。

8.13 Replace Type Code with Class
以类取代型别码

class之中有一个数值型别码（numeric type code），但它并不影响class的行为。

以一个新的class替换该数值型别码（type code）。



动机（Motivation）

在以C为基础的编程语言中，type code（型别码）或枚举值（enumerations）很常见。如果带着一个有意义的符号名，type code的可读性还是不错的。问题在于，符号名终究只是个别名，编译器看见的、进行型别检验的，还是背后那个数值。任何接受type code作为引数（argument）的函数，所期望的实际上是一个数值，无法强制使用符号名。这会大大降低代码的可读性，从而成为臭虫之源。

如果把那样的数值换成一个class，编译器就可以对这个class进行型别检验。只要为这个class提供factory methods，你就可以始终保证只有合法的实体才会被创建出来，而且它们都会被传递给正确的宿主对象。

但是，在使用 *Replace Type Code with Class*（218）之前，你应该先考虑type code的其他替换方式。只有当type code是纯粹数据时（也就是type code不会在switch语句中引起行为变化时），你才能以class来取代它。Java只能以整数作为`switch`语句的「转辙」依据，不能使用任意class，因此那种情况下不能够以class替换type code。更重要的是：任何`switch`语句都应该运用 *Replace Conditional with Polymorphism*（255）去掉。为了进行那样的重构，你首先必须运用 *Replace Type Code with Subclasses*（223）或 *Replace Type Code with State/Strategy*（227）把type code处理掉。

即使一个 type code 不会因其数值的不同而引起行为上的差异，宿主类中的某些行为还是有可能更适合置放于 type code class 中，因此你还应该留意是否有必要使用 *Move Method*（142）将一两个函数搬过去。

作法（Mechanics）

- 为 type code 建立一个 class。
  - ⇨ 这个 class 内需要一个用以记录 type code 的值域，其型别应该和 type code 相同；并应该有对应的取值函数（getter）。此外还应该用一组 static 变量保存「允许被创建」的实体，并以一个 static 函数根据原本的 type code 返回合适的实体。
- 修改 source class 实现码，让它使用上述新建的 class。
  - ⇨ 维持原先以 type code 为基础的函数接口，但改变 static 值域，以新建的 class 产生代码。然后，修改 type code 相关函数，让它们也从新建的 class 中获取代码。
- 编译，测试。
  - ⇨ 此时，新建的 class 可以对 type code 进行运行期检查。
- 对于 source class 中每一个使用 type code 的函数，相应建立一个函数，让新函数使用新建的 class。
  - ⇨ 你需要建立「以新 class 实体为自变量」的函数，用以替换原先「直接以 type code 为引数」的函数。你还需要建立一个「返回新 class 实体」的函数，用以替换原先「直接返回 type code」的函数。建立新函数前，你可以使用 *Rename Method*（273）修改原函数名称，明确指出那些函数仍然使用旧式的 type code，这往往是个明智之举。
- 逐一修改 source class 用户，让它们使用新接口。
- 每修改一个用户，编译并测试。
  - ⇨ 你也可能需要一次性修改多个彼此相关的函数，才能保持这些函数之间的一致性，才能顺利地编译、测试。
- 删除「使用 type code」的旧接口，并删除「保存旧 type code」的静态变量。
- 编译，测试。

范例（Example）

每个人都拥有四种血型中的一种。我们以 Person 来表示「人」，以其中的 type code 表示「血型」：

```
class Person {
    public static final int O = 0;
    public static final int A = 1;
    public static final int B = 2;
    public static final int AB = 3;

    private int _bloodGroup;
    public Person (int bloodGroup) {
      _bloodGroup = bloodGroup;
    }
    public void setBloodGroup(int arg) {
      _bloodGroup = arg;
    }
    public int getBloodGroup() {
      return _bloodGroup;
    }
}
```

首先，我建立一个新的 BloodGroup class，用以表示「血型」，并在这个 class 实体中保存原本的 type code 数值：

```
class BloodGroup {
    public static final BloodGroup O = new BloodGroup(0);
    public static final BloodGroup A = new BloodGroup(1);
    public static final BloodGroup B = new BloodGroup(2);
    public static final BloodGroup AB = new BloodGroup(3);
    private static final BloodGroup[] _values = {O, A, B, AB};

    private final int _code;

    private BloodGroup (int code ) {
      _code = code;
    }

    public int getCode() {
      return _code;
    }

    public static BloodGroup code(int arg) {
      return _values[arg];
    }
}
```

然后，我把 Person 中的 type code 改为使用 BloodGroup class：

```
class Person {
    public static final int O = BloodGroup.O.getCode();
    public static final int A = BloodGroup.A.getCode();
    public static final int B = BloodGroup.B.getCode();
    public static final int AB = BloodGroup.AB.getCode();

    private BloodGroup _bloodGroup;

    public Person(int bloodGroup) {
      _bloodGroup = BloodGroup.code(bloodGroup);
    }
    public int getBloodGroup() {
      return _bloodGroup.getCode();
    }
    public void setBloodGroup(int arg) {
      _bloodGroup = BloodGroup.code(arg);
    }
}
```

现在，我因为 BloodGroup class 而拥有了运行期检验能力。为了真正从这些改变中获利，我还必须修改 Person 的用户，让它们以 BloodGroup 对象表示 type code，而不再使用整数。

首先，我使用 *Rename Method*（273）修改 type code 访问函数的名称，说明当前情况：

```
class Person...
    public int getBloodGroupCode() {
      return _bloodGroup.getCode();
    }
```

然后我为 Person 加入一个新的取值函数（getter），其中使用 BloodGroup：

```
public BloodGroup getBloodGroup() {
    return _bloodGroup;
}
```

另外，我还要建立新的构造函数和设值函数（setter），让它们也使用 BloodGroup：

```
public Person(BloodGroup bloodGroup) {
    _bloodGroup = bloodGroup;
}
public void setBloodGroup(BloodGroup arg) {
    _bloodGroup = arg;
}
```

现在，我要继续处理 Person 用户。此时应该注意，每次只处理一个用户，这样才可以保持小步前进。每个用户需要的修改方式可能不同，这使得修改过程更加棘手。对 Person 内的 static 变量的所有引用点也需要修改。因此，下列代码：

```
Person thePerson = new Person(Person.A)
```

就变成了：

```
Person thePerson = new Person(BloodGroup.A);
```

「调用取值函数（getter）」必须改为「调用新取值函数」。因此，下列代码：

```
thePerson.getBloodGroupCode()
```

变成了：

```
thePerson.getBloodGroup().getCode()
```

设值函数（setter）也一样。因此，下列代码：

```
thePerson.setBloodGroup(Person.AB)
```

变成了：

```
thePerson.setBloodGroup(BloodGroup.AB)
```

修改完毕 Person 的所有用户之后，我就可以删掉原本使用整数型别的那些旧的取值函数、构造函数、静态变量和设值函数了：

```
class Person ...
    public static final int O = BloodGroup.O.getCode();
    public static final int A = BloodGroup.A.getCode();
    public static final int B = BloodGroup.B.getCode();
    public static final int AB = BloodGroup.AB.getCode();
    public Person(int bloodGroup) {
      _bloodGroup = BloodGroup.code(bloodGroup);
    }
    public int getBloodGroupCode() {
      return _bloodGroup.getCode();
    }
    public void setBloodGroup(int arg) {
      _bloodGroup = BloodGroup.code(arg);
    }
```

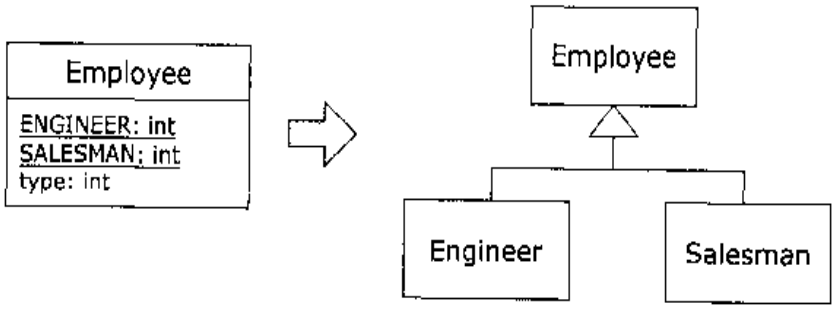
我还可以将 BloodGroup 中使用整数型别的函数声明为 private（因为再没有人会使用它们了）：

```
class BloodGroup...
    private int getCode() {
      return _code;
    }
    private static BloodGroup code(int arg) {
      return _values[arg];
    }
```

8.14 Replace Type Code with Subclasses
以子类取代型别码

你有一个不可变的（immutable）type code，它会影响 class 的行为。

以一个 subclass 取代这个 type code。



动机（Motivation）

如果你面对的 type code 不会影响宿主类的行为，你可以使用 *Replace Type Code with Class*（218）来处理它们。但如果 type code 会影响宿主类的行为，那么最好的办法就是借助多态（polymorphism）来处理变化行为。

一般来说，这种情况的标志就是像 `switch` 这样的条件式。这种条件式可能有两种表现形式：switch 语句或者 if-then-else 结构。不论哪种形式，它们都是检查 type code 值，并根据不同的值执行不同的动作。这种情况下你应该以 *Replace Conditional with Polymorphism*（255）进行重构。但为了能够顺利进行那样的重构，首先应该将 type code 替换为可拥有多态行为的继承体系。这样的一个继承体系应该以 type code 的宿主类为 base class，并针对每一种 type code 各建立一个 subclass。

为建立这样的继承体系，最简单的办法就是 *Replace Type Code with Subclasses*（223）：以type code的宿主类为base class，针对每种type code建立相应的subclass。但是以下两种情况你不能那么做：(1) type code 值在对象创建之后发生了改变；(2) 由于某些原因，type code 宿主类已经有了 subclass。如果你恰好面临这两种情况之一，就需要使用 *Replace Type Code with State/Strategy*（227）。

Replace Type Code with Subclasses（223）的主要作用其实是搭建一个舞台，让 *Replace Conditional with Polymorphism*（255）得以一展身手。如果宿主类中并没有出现条件式，那么 *Replace Type Code with Class*（218）更合适，风险也比较低。

使用 *Replace Type Code with Subclasses*（223）的另一个原因就是，宿主类中出现了「只与具备特定 type code 之对象相关」的特性。完成本项重构之后，你可以使用 *Push Down Method*（328）和 *Push Down Field*（329）将这些特性推到合适的 subclass 去，以彰显它们「只与特定情况相关」这一事实。

Replace Type Code with Subclasses（223）的好处在于：它把「对不同行为的了解」从 class 用户那儿转移到了 class 自身。如果需要再加入新的行为变化，我只需添加一个 subclass 就行了。如果没有多态机制，我就必须找到所有条件式，并逐一修改它们。因此，如果未来还有可能加入新行为，这项重构将特别有价值。

作法（Mechanics）

- 使用 *Self Encapsulate Field*（171）将 type code 自我封装起来。
  - ⇨ 如果 type code 被传递给构造函数，你就需要将构造函数换成 ***factory method***。
- 为 type code 的每一个数值建立一个相应的 subclass。在每个 subclass 中覆写（override）type code 的取值函数（getter），使其返回相应的 type code 值。
  - ⇨ 这个值被硬编码于 return 句中（例如：`return1`）。这看起来很骯脏，但只是权宜之计。当所有 case 子句都被替换后，问题就解决了。
- 每建立一个新的 subclass，编译并测试。
- 从 superclass 中删掉保存 type code 的值域。将 type code 访问函数（accessors）声明为抽象函数（abstract method）。
- 编译，测试。

范例（Example）

为简单起见，我还是使用那个恼人又不切实际的「雇员/薪资」例。我们以 `Employee` 表示「雇员」：

```
class Employee...
   private int _type;
   static final int ENGINEER = 0;
   static final int SALESMAN = 1;
   static final int MANAGER = 2;
   Employee (int type) {
       _type = type;
   }
```

第一步是以 *Self Encapsulate Field*（171）将 type code 自我封装起来：

```
int getType() {
    return _type;
}
```

由于 Employee 构造函数接受 type code 作为一个参数，所以我必须将它替换为一个 ***factory method***：

```
static Employee create(int type) {
    return new Employee(type);
}
private Employee (int type) {
    _type = type;
}
```

现在，我可以先建立一个 subclassEngineer 表示「工程师」。首先我建立这个 subclass，并在其中覆写 type code 取值函数：

```
class Engineer extends Employee {
    int getType() {
       return Employee.ENGINEER;
    }
}
```

同时我该修改 ***factory method***，令它返回一个合适的对象：

```
class Employee
    static Employee create(int type) {
       if (type == ENGINEER)  return new Engineer();
       else   return new Employee(type);
    }
```

然后，我继续逐一地处理其他 type code，直到所有 type code 都被替换成 subclass 为止。此时我就可以移除 Employee 中保存 type code 的值域，并将 getType()声明为一个抽象函数。现在，***factory method*** 看起来像这样：

```
abstract int getType();
static Employee create(int type) {
    switch (type) {
      case ENGINEER:
          return new Engineer();
      case SALESMAN:
          return new Salesman();
      case MANAGER:
          return new Manager();
```

```
        default:
            throw new IllegalArgumentException("Incorrect type
        code value");
    }
}
```

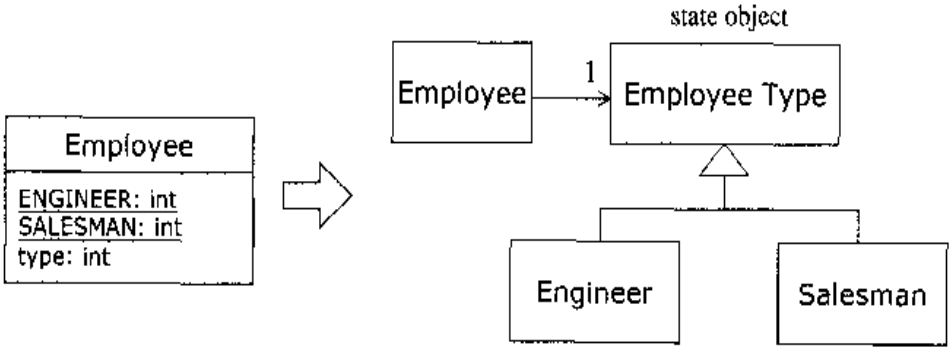
当然，我总是避免使用 switch 语句。但这里只有一处用到 switch 语句，并且只用于决定创建何种对象，这样的 switch 语句是可以接受的。

很自然地，在建立了这些 subclass 之后，你就应该使用 *Push Down Method*（328）和 *Push Down Field*（329），将「只与特定种类的雇员相关」的函数和值域推到相关的 subclass 去。

8.15 Replace Type Code with State/Strategy
以 State/Strategy 取代型别码

你有一个 type code，它会影响 class 的行为，但你无法使用 subclassing。

以 ***state*** object（专门用来描述状态的对象）取代 type code。



动机（Motivation）

本项重构和 *Replace Type Code with Subclasses*（223）很相似，但如果「type code 的值在对象生命期中发生变化」或「其他原因使得宿主类不能被 subclassing」，你也可以使用本重构。本重构使用 ***State*** 模式或 ***Strategy*** 模式 [Gang of Four]。

State 模式和 ***Strategy*** 模式非常相似，因此无论你选择其中哪一个，重构过程都是相同的。「选择哪一个模式」并非问题关键所在，你只需要选择更适合特定情境的模式就行了。如果你打算在完成本项重构之后再以 *Replace Conditional with Polymorphism*（255）简化一个算法，那么选择 ***Strategy*** 模式比较合适；如果你打算搬移与状态相关（state-specific）的数据，而且你把新建对象视为一种变迁状态（changing state），就应该选择使用 ***State*** 模式。

作法（Mechanics）

- 使用 *Self Encapsulate Field*（171）将 type code 自我封装起来。
- 新建一个 class，根据 type code 的用途为它命名。这就是一个 ***state*** object。
- 为这个新建的 class 添加 subclasses，每个 subclass 对应一种 type code。
  - ⇨ 比起逐一添加，一次性加入所有必要的 subclasses 可能更简单些。

- 在superclass中建立一个抽象的查询函数(abstract query),用以返回type code。在每个subclass中覆写该函数，返回确切的type code。
- 编译。
- 在source class中建立一个值域，用以保存新建的 ***state*** object。
- 调整source class中负责查询type code的函数，将查询动作转发给 ***state*** object。
- 调整source class中「为type code设值」的函数，将一个恰当的state object subclass赋值给「保存 ***state*** object」的那个值域。
- 编译，测试。

范例（Example）

和上一项重构一样，我仍然使用这个既无聊又弱智的「雇员/薪资」例子。同样地，我以Employee表示「雇员」：

```
class Employee {

    private int _type;
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;

    Employee (int type) {
      _type = type;
    }
```

下面的代码展示使用这些type code的条件式：

```
int payAmount() {
    switch (_type) {
      case ENGINEER:
          return _monthlySalary;
      case SALESMAN:
          return _monthlySalary + _commission;
      case MANAGER:
          return _monthlySalary + _bonus;
      default:
          throw new RuntimeException("Incorrect Employee");
    }
}
```

假设这是一家激情四溢、积极进取的公司，他们可以将表现出色的工程师擢升为经理。因此，对象的 type code 是可变的，所以我不能使用 subclassing 方式来处理 type code。和以前一样，我的第一步还是使用 *Self Encapsulate Field*（171）将表示 type code 的值域自我封装起来：

```
Employee (int type) {
   setType (type);
}
int getType() {
   return _type;
}
void setType(int arg) {
   _type = arg;
}
int payAmount() {
   switch (getType()) {
      case ENGINEER:
         return _monthlySalary;
      case SALESMAN:
         return _monthlySalary + _commission;
      case MANAGER:
         return _monthlySalary + _bonus;
      default:
         throw new RuntimeException("Incorrect Employee");
   }
}
```

现在，我需要声明一个 state class。我把它声明为一个抽象类（abstract class），并提供一个抽象函数（abstract method），用以返回 type code：

```
abstract class EmployeeType {
   abstract int getTypeCode();
}
```

现在，我可以开始创造 subclass 了：

```
class Engineer extends EmployeeType {
   int getTypeCode () {
      return Employee.ENGINEER;
   }
}
```

```
class Manager extends EmployeeType {
    int getTypeCode () {
        return Employee.MANAGER;
    }
}
class Salesman extends EmployeeType {
    int getTypeCode () {
        return Employee.SALESMAN;
    }
}
```

现在进行一次编译。前面所做的修改实在太平淡了，即使对我来说也太简单。现在，我要修改type code访问函数(accessors)，实实在在地把这些subclasses和Employee class联系起来：

```
class Employee...
  private EmployeeType _type;

  int getType() {
      return _type.getTypeCode();
  }

  void setType(int arg) {
      switch (arg) {
      case ENGINEER:
          _type = new Engineer();
          break;
      case SALESMAN:
          _type = new Salesman();
          break;
      case MANAGER:
          _type = new Manager();
          break;
      default:
          throw new IllegalArgumentException("Incorrect Employee Code");
      }
  }
```

这意味我将在这里拥有一个switch语句。完成重构之后，这将是代码中惟一的switch语句，并且只在对象型别发生改变时才会被执行。我也可以运用 *Replace Constructor with Factory Method*（304）针对不同的case子句建立相应的 ***factory methods***。我还可以立刻再使用 *Replace Conditional with Polymorphism*（255），从而将其他的case子句完全消除。

最后，我喜欢将所有关于type code和subclass的知识都移到新的class，并以此结束 *Replace Type Code with State/Strategy*（227）。首先我把type code的定义拷贝

到 EmployeeType class 去，在其中建立一个 ***factory method*** 以生成适当的 EmployeeType 对象，并调整 Employee class 中为 type code 赋值的函数：

```
class Employee...
  void setType(int arg) {
     _type = EmployeeType.newType(arg);
  }

class EmployeeType...
  static EmployeeType newType(int code) {
     switch (code) {
        case ENGINEER:
           return new Engineer();
        case SALESMAN:
           return new Salesman();
        case MANAGER:
           return new Manager();
        default:
           throw new IllegalArgumentException("Incorrect Employee Code");
     }
  }
  static final int ENGINEER = 0;
  static final int SALESMAN = 1;
  static final int MANAGER = 2;
```

然后，我删掉 Employee 中的 type code 定义，代之以一个「指向（代表、指涉）EmployeeType 对象」的 reference：

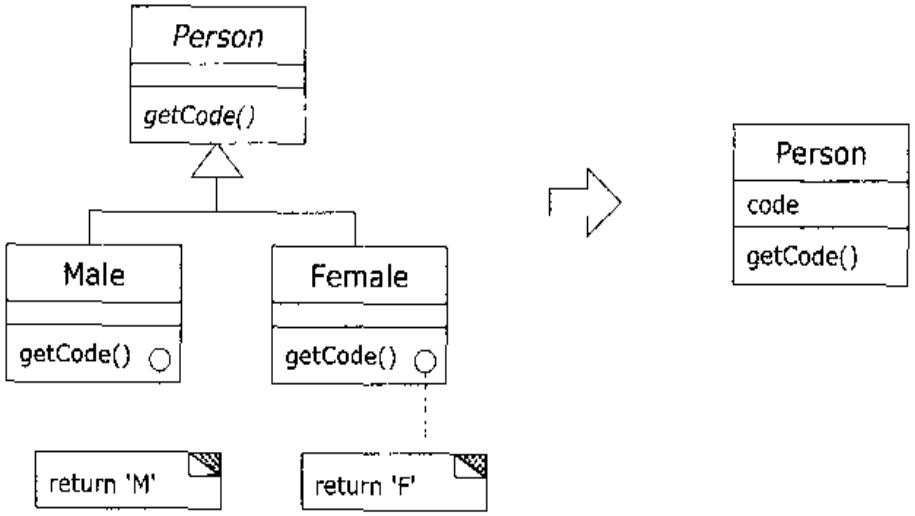
```
class Employee...
    int payAmount() {
      switch (getType()) {
          case EmployeeType.ENGINEER:
              return _monthlySalary;
          case EmployeeType.SALESMAN:
              return _monthlySalary + _commission;
          case EmployeeType.MANAGER:
              return _monthlySalary + _bonus;
          default:
              throw new RuntimeException("Incorrect Employee");
      }
    }
```

现在，万事俱备，我可以运用 *Replace Conditional with Polymorphism*（255）来处理 payAmount()函数了。

8.16 Replace Subclass with Fields（以值域取代子类）

你的各个 subclasses 的惟一差别只在「返回常量数据」的函数身上。

修改这些函数，使它们返回 superclass 中的某个（新增）值域，然后销毁 subclasses。



动机（Motivation）

建立 subclass 的目的，是为了增加新特性，或变化其行为。有一种变化行为（variant behavior）称为「常量函数」（constant method）[Beck]，它们会返回一个硬编码（hard-coded）值。这东西有其用途：你可以让不同的 subclasses 中的同一个访问函数（accessors）返回不同的值。你可以在 superclass 中将访问函数声明为抽象函数，并在不同的 subclass 中让它返回不同的值。

尽管常量函数有其用途，但若 subclass 中只有常量函数，实在没有足够的存在价值。你可以在 superclass 中设计一个与「常量函数返回值」相应的值域，从而完全去除这样的 subclass。如此一来就可以避免因 subclassing 而带来的额外复杂性。

作法（Mechanics）

- 对所有 subclasses 使用 *Replace Constructor with Factory Method*（304）。
- 如果有任何代码直接引用 subclass，令它改而引用 superclass。

- 针对每个常量函数，在 superclass 中声明一个 final 值域。
- 为 superclass 声明一个 protected 构造函数，用以初始化这些新增值域。
- 新建或修改 subclass 构造函数，使它调用 superclass 的新增构造函数。
- 编译，测试。
- 在 superclass 中实现所有常量函数，令它们返回相应值域值，然后将该函数从 subclass 中删掉。
- 每删除一个常量函数，编译并测试。
- subclass 中所有的常量函数都被删除后，使用 *Inline Method*（117）将 subclass 构造函数内联（*inlining*）到 superclass 的 ***factory method*** 中。
- 编译，测试。
- 将 subclass 删掉。
- 编译，测试。
- 重复「*inlining* 构造函数、删除 subclass」过程，直到所有 subclass 都被删除。

范例（Example）

本例之中，我以 `Person` 表示「人」，并针对每种性别建立一个 subclass：以 `Male` subclass 表示「男人」，以 `Female` subclass 表示「女人」：

```
abstract class Person {

    abstract boolean isMale();
    abstract char getCode();
...

class Male extends Person {
    boolean isMale() {
        return true;
    }
    char getCode() {
        return 'M';
    }
}
class Female extends Person {
    boolean isMale() {
        return false;
    }
    char getCode() {
        return 'F';
    }
}
```

在这里，两个 subclasses 之间惟一的区别就是：它们以不同的方式实现了 Person 所声明的抽象函数 getCode()，返回不同的硬编码常量（所以 getCode()是个常量函数 [Beck]）。我应该将这两个怠惰的 subclasses 去掉。

首先我需要使用 *Replace Constructor with Factory Method*（304）。在这里，我需要为每个 subclass 建立一个 ***factory method***：

```
class Person...
  static Person createMale(){
     return new Male();
  }
  static Person createFemale() {
     return new Female();
  }
```

然后我把对象创建过程从以下这样：

```
Person kent = new Male();
```

改为这样：

```
Person kent = Person.createMale();
```

将所有「构造函数调用动作」都替换为「***factory method*** 调用动作」后，我就不应该再有任何对 subclass 的直接引用了。一次全文搜索就可以帮助我证实这一点。然后，我可以把这两个 subclasses 都声明为 private，这样编译器就可以帮助我，保证至少 package 之外不会有任何代码取用它们。

现在，针对每个常量函数，在 superclass 中声明一个对应的值域：

```
class Person...
  private final boolean _isMale;
  private final char _code;
```

然后为 superclass 加上一个 protected 构造函数：

```
class Person...
  protected Person (boolean isMale, char code) {
     _isMale = isMale;
     _code = code;
  }
```

再为 subclass 加上新构造函数，令它调用 superclass 新增的构造函数：

```
class Male...
  Male() {
     super (true, 'M');
  }
```

```
class Female...
  Female() {
     super (false, 'F');
  }
```

完成这一步后，编译并测试。所有值域都被创建出来并被赋予初值，但到目前为止，我们还没有使用它们。现在我可以在 superclass 中加入访问这些值域的函数，并删掉 subclass 中的常量函数，从而让这些值域粉墨登场：

```
class Person...
  boolean isMale() {
     return _isMale;
  }
class Male...
  boolean isMale() {
     return true;
  }
```

我可以逐一对每个值域、每个 subclass 进行这一步骤的修改；如果我相信自己的运气，也可以采取一次性全部修改的手段。

所有值域都处理完毕后，所有 subclasses 也都空空如也了，于是我可以删除 Person 中那个抽象函数的 abstract 修饰符（使它不再成为抽象函数），并以 *Inline Method*（117）将 subclass 构造函数内联（*inlining*）到 superclass 的 ***factory method*** 中：

```
class Person...
  static Person createMale() {
     return new Person(true, 'M');
  }
```

编译、测试后，我就可以删掉 Male class，并对 Female class 重复上述过程。

9

简化条件表达式

Simplifying Conditional Expressions

条件逻辑（conditional logic）有可能十分复杂，因此本章提供一些重构手法，专门用来简化它们。其中一项核心重构就是 *Decompose Conditional*（238），可将一个复杂的条件逻辑分成若干小块。这项重构很重要，因为它使得「转辙逻辑」（switching logic）和「操作细节」（details）分离。

本章的其余重构手法可用以处理另一些重要问题：如果你发现代码中的多处测试有相同结果，应该实施 *Consolidate Conditional Expression*（240）；如果条件代码中有任何重复，可以运用 *Consolidate Duplicate Conditional Fragments*（243）将重复成分去掉。

如果程序开发者坚持「单一出口（one exit point）」原则，那么为让条件式也遵循这一原则，他往往会在其中加入控制标记（control flags）。我并不特别在意「一个函数一个出口」原则，所以我使用 *Replace Nested Conditional with Guard Clauses*（250）标示出那些特殊情况，并使用 *Remove Control Flag*（245）去除那些讨厌的控制标记。

较之于过程化（procedural）程序而言，面向对象（object oriented）程序的条件式通常比较少，这是因为很多条件行为都被多态机制（polymorphism）处理掉了。多态之所以更好，是因为调用者无需了解条件行为的细节，因此条件的扩展更为容易。所以面向对象程序中很少出现 `switch` 语句；一旦出现，就应该考虑运用 *Replace Conditional with Polymorphism*（255）将它替换为多态。

多态还有一种十分有用但鲜为人知的用途：通过 *Introduce Null Object*（260）去除对于 null value 的检验。

9.1 Decompose Conditional（分解条件式）

你有一个复杂的条件（if-then-else）语句。

从if、then、else三个段落中分别提炼出独立函数。

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else
    charge = quantity * _summerRate;
```

```
if (notSummer(date))
    charge = winterCharge(quantity);
else
    charge = summerCharge (quantity);
```

动机（Motivation）

程序之中，「复杂的条件逻辑」是最常导致复杂度上升的地点之一。你必须编写代码来检查不同的条件分支、根据不同的分支做不同的事，然后，你很快就会得到一个相当长的函数。大型函数自身就会使代码的可读性下降，而条件逻辑则会使代码更难阅读。在带有复杂条件逻辑的函数中，代码（包括检查条件分支的代码和真正实现功能的代码）会告诉你发生的事，但常常让你弄不清楚为什么会发生这样的事，这就说明代码的可读性的确大大降低了。

和任何大块头代码一样，你可以将它分解为多个独立函数，根据每个小块代码的用途，为分解而得的新函数命名，并将原函数中对应的代码替换成「对新建函数的调用」，从而更清楚地表达自己的意图。对于条件逻辑，「将每个分支条件分解，形成新函数」还可以给你带来更多好处：可以突出条件逻辑，更清楚地表明每个分支的作用，并且突出每个分支的原因。

作法（Mechanics）

- 将if段落提炼出来，构成一个独立函数。
- 将then段落和else段落都提炼出来，各自构成一个独立函数。

如果发现嵌套的（nested）条件逻辑，我通常会先观察是否可以使用 *Replace Nested Conditional with Guard Clauses*（250）。如果不行，才开始分解其中的每个条件。

范例（Example）

假设我要计算购买某样商品的总价（总价=数量*单价），而这个商品在冬季和夏季的单价是不同的：

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else
    charge = quantity * _summerRate;
```

我把每个分支的判断条件都提炼到一个独立函数中，如下所示：

```
if (notSummer(date))
    charge = winterCharge(quantity);
else
    charge = summerCharge (quantity);

private boolean notSummer(Date date) {
    return date.before (SUMMER_START) || date.after(SUMMER_END);
}
private double summerCharge(int quantity) {
    return quantity * _summerRate;
}
private double winterCharge(int quantity) {
    return quantity * _winterRate + _winterServiceCharge;
}
```

通过这段代码你可以看出，整个重构带来的清晰性。实际工作中，我会逐步进行每一次提炼，并在每次提炼之后编译并测试。

像这样的情况下，许多程序员都不会去提炼分支条件。因为这些分支条件往往非常短，看上去似乎没有提炼的必要。但是，尽管这些条件往往很短，在代码意图和代码自身之间往往存在不小的差距。哪怕在上面这样一个小小例子中，notSummer(date)这个语句也能够比原本的代码更好地表达自己的用途。对于原来的代码，我必须看着它，想一想，才能说出其作用。当然，在我们这个简单的例子中，这并不困难。不过，即使如此，提炼出来的函数可读性也更高一些——它看上去就像一段注释那样清楚而明白。

9.2 Consolidate Conditional Expression（合并条件式）

你有一系列条件测试，都得到相同结果。

将这些测试合并为一个条件式，并将这个条件式提炼成为一个独立函数。

```
double disabilityAmount() {
if (_seniority < 2) return 0;
if (_monthsDisabled > 12) return 0;
if (_isPartTime) return 0;
// compute the disability amount
```

```
double disabilityAmount() {
if (isNotEligableForDisability()) return 0;
// compute the disability amount
```

动机（Motivation）

有时你会发现这样一串条件检查：检查条件各不相同，最终行为却一致。如果发现这种情况，就应该使用 logical-AND 和 logical-OR 将它们合并为一个条件式。

之所以要合并条件代码，有两个重要原因。首先，合并后的条件代码会告诉你「实际上只有一次条件检查，只不过有数个并列条件需要检查而已」，从而使这一次检查的用意更清晰。当然，合并前和合并后的代码有着相同的效果，但原先代码传达出的信息却是「这里有一些独立条件测试，它们只是恰好同时发生」。其次，这项重构往往可以为你使用 *Extract Method*（110）做好准备。「将检查条件提炼成一个独立函数」对于厘清代码意义非常有用，因为它把描述「做什么」的语句换成了「为什么这样做」。

条件语句的「合并理由」也同时指出了「不要合并」的理由。如果你认为这些检查的确彼此独立，的确不应该被视为同一次检查，那么就不要使用本项重构。因为在这种情况下，你的代码已经清楚表达出自己的意义。

作法（Mechanics）

- □ 确定这些条件语句都没有副作用（连带影响）。

  ⇨ 如果条件式有副作用，你就不能使用本项重构。

- □ 使用适当的逻辑操作符（logical operators），将一系列相关条件式合并为一个。
- □ 编译，测试。
- □ 对合并后的条件式实施 *Extract Method*（110）。

范例：Ors

请看下列代码：

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // compute the disability amount
    ...
```

在这段代码中，我们看到一连串的条件检查，它们都做同一件事。对于这样的代码，上述条件检查等价于一个以 "logical-OR" 连接起来的语句：

```
double disabilityAmount() {
    if ((_seniority < 2) || (_monthsDisabled > 12) || (_isPartTime))
      return 0;
    // compute the disability amount
    ...
```

现在，我可以观察这个新的条件式，并运用 *Extract Method*（110）将它提炼成一个独立函数，以函数名称表达该语句所检查的条件：

```
double disabilityAmount() {
    if (isNotEligibleForDisability()) return 0;
    // compute the disability amount
    ...
}
boolean isNotEligibleForDisability() {
    return ((_seniority < 2) ||
            (_monthsDisabled > 12) || (_isPartTime));
}
```

范例：Ands

上述实例展示了 "logical Or" 的用法。下列代码展示 "logical-AND" 的用法：

```
if (onVacation())
    if (lengthOfService() > 10)
      return 1;
return 0.5;
```

这段代码可以变成：

```
if (onVacation() && lengthOfService() > 10) return 1;
else return 0.5;
```

你可能还会发现，某些情况下需要同时使用 logical-AND、logical-OR 和 logical_NOT，最终得到的条件式可能很复杂，所以我会先使用 *Extract Method*（110）将表达式的一部分提炼出来，从而使整个表达式变得简单一些。

如果我所观察的部分，只是对条件进行检查并返回一个值，我可以使用三元操作符将这一部分变成一条 return 语句。因此，下列代码：

```
if (onVacation() && lengthOfService() > 10) return 1;
else return 0.5;
```

就变成了：

```
return (onVacation() && lengthOfService() > 10) ? 1 : 0.5;
```

9.3 Consolidate Duplicate Conditional Fragments （合并重复的条件片段）

在条件式的每个分支上有着相同的一段代码。

将这段重复代码搬移到条件式之外。

```
if (isSpecialDeal()) {
   total = price * 0.95;
   send();
}
else {
   total = price * 0.98;
send();
}
```

```
if (isSpecialDeal())
   total = price * 0.95;
else
   total = price * 0.98;
send();
```

动机（Motivation）

有时你会发现，一组条件式的所有分支都执行了相同的某段代码。如果是这样，你就应该将这段代码搬移到条件式外面。这样，代码才能更清楚地表明哪些东西随条件的变化而变化、哪些东西保持不变。

作法（Mechanics）

- 鉴别出「执行方式不随条件变化而变化」的代码。
- 如果这些共通代码位于条件式起始处，就将它移到条件式之前。
- 如果这些共通代码位于条件式尾端，就将它移到条件式之后。

- 如果这些共通代码位于条件式中段，就需要观察共通代码之前或之后的代码是否改变了什么东西。如果的确有所改变，应该首先将共通代码向前或向后移动，移至条件式的起始处或尾端，再以前面所说的办法来处理。
- 如果共通代码不止一条语句，你应该首先使用 *Extract Method*（110）将共通代码提炼到一个独立函数中，再以前面所说的办法来处理。

范例（Example）

你可能遇到这样的代码：

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

由于条件式的两个分支都执行了 send()函数，所以我应该将 send()移到条件式的外围：

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```

我们也可以使用同样的手法来对待异常（exceptions）。如果在 try 区段内「可能引发异常」的语句之后，以及所有 catch 区段之内，都重复执行了同一段代码，我就可以将这段重复代码移到 final 区段。

9.4 Remove Control Flag（移除控制标记）

在一系列布尔表达式（boolean expressions）中，某个变量带有「控制标记」（control flag）的作用。

以 break 语句或 return 语句取代控制标记。

动机（Motivation）

在一系列条件表达式中，你常常会看到「用以判断何时停止条件检查」的控制标记（control flag）：

```
set done to false
while not done
    if (condition)
      do something
      set done to true
    next step of loop
```

这样的控制标记带来的麻烦超过了它所带来的便利。人们之所以会使用这样的控制标记，因为结构化编程原则告诉他们：每个子程序（routines）只能有一个入口（entry）和一个出口（exit）。我赞同「单一入口」原则（而且现代编程语言也强迫我们这样做），但是「单一出口」原则会让你在代码中加入讨厌的控制标记，大大降低条件表达式的可读性。这就是编程语言提供 break 语句和 continue 语句的原因：你可以用它们跳出复杂的条件语句。去掉控制标记所产生的效果往往让你大吃一惊：条件语句真正的用途会清晰得多。

作法（Mechanics）

对控制标记（control flags）的处理，最显而易见的办法就是使用 Java 提供的 break 语句或 continue 语句。

- 找出「让你得以跳出这段逻辑」的控制标记值。
- 找出「将可跳出条件式之值赋予标记变量」的那个语句，代以恰当的 break 语句或 continue 语句。
- 每次替换后，编译并测试。

在未能提供 break 和 continue 语句的编程语言中，我们可以使用另一种办法：

- 运用 *Extract Method*（110），将整段逻辑提炼到一个独立函数中。
- 找出「让你得以跳出这段逻辑」的那些控制标记值。
- 找出「将可跳出条件式之值赋予标记变量」的那个语句，代以恰当的 return 语句。
- 每次替换后，编译并测试。

即使在支持 break 和 continue 语句的编程语言中，我通常也优先考虑上述第二方案。因为 return 语句可以非常清楚地表示：不再执行该函数中的其他任何代码。如果还有这一类代码，你早晚需要将这段代码提炼出来。

请注意标记变量是否会影响这段逻辑的最后结果。如果有影响，使用 break 语句之后你还得保留控制标记值。如果你已经将这段逻辑提炼成一个独立函数，也可以将控制标记值放在 return 语句中返回。

范例：以 break 取代简单的控制标记

下列函数用来检查一系列人名之中是否包含两个可疑人物的名字（这两个人的名字硬编码于代码中）：

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
      if (! found) {
          if (people[i].equals ("Don")){
              sendAlert();
              found = true;
          }
          if (people[i].equals ("John")){
              sendAlert();
              found = true;
          }
      }
    }
}
```

这种情况下很容易找出控制标记：当变量 found 被赋予 true 时，搜索就结束。我可以逐一引入 break 语句：

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
      if (! found) {
          if (people[i].equals ("Don")){
              sendAlert();
              break;
          }
          if (people[i].equals ("John")){
              sendAlert();
              found = true;
          }
      }
    }
}
```

最后获得这样的成果：

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
      if (! found) {
          if (people[i].equals ("Don")){
              sendAlert();
              break;
          }
          if (people[i].equals ("John")){
              sendAlert();
              break;
          }
      }
    }
}
```

然后我就可以把对控制标记的所有引用都去掉：

```
void checkSecurity(String[] people) {
    for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don")){
          sendAlert();
          break;
      }
      if (people[i].equals ("John")){
          sendAlert();
          break;
      }
    }
}
```

范例：以return返回控制标记

本项重构的另一种形式将使用return语句。为了阐述这种用法，我把前面的例子稍加修改，以控制标记记录搜索结果：

```
void checkSecurity(String[] people) {
    String found = "";
    for (int i = 0; i < people.length; i++) {
      if (found.equals("")) {
          if (people[i].equals ("Don")){
              sendAlert();
              found = "Don";
          }
          if (people[i].equals ("John")){
              sendAlert();
              found = "John";
          }
      }
    }
    someLaterCode(found);
}
```

在这里，变量found做了两件事：它既是控制标记，也是运算结果。遇到这种情况，我喜欢先把计算found变量的代码提炼到一个独立函数中：

```
void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}

String foundMiscreant(String[] people){
    String found = "";
    for (int i = 0; i < people.length; i++) {
      if (found.equals("")) {
          if (people[i].equals ("Don")){
              sendAlert();
              found = "Don";
          }
          if (people[i].equals ("John")){
              sendAlert();
              found = "John";
          }
      }
    }
    return found;
}
```

然后以 return 语句取代控制标记：

```
String foundMiscreant(String[] people){
    String found = "";
    for (int i = 0; i < people.length; i++) {
      if (found.equals("")) {
          if (people[i].equals ("Don")){
              sendAlert();
              return "Don";
          }
          if (people[i].equals ("John")){
              sendAlert();
              found = "John";
          }
      }
    }
    return found;
}
```

最后完全去掉控制标记：

```
String foundMiscreant(String[] people){
    for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don")){
          sendAlert();
          return "Don";
      }
      if (people[i].equals ("John")){
          sendAlert();
          return "John";
      }
    }
    return "";
}
```

即使不需要返回某值，你也可以使用 return 语句来取代控制标记。这时候你只需要一个空的 return 语句就行了。

当然，如果以此办法去处理带有副作用（连带影响）的函数，会有一些问题。所以我需要先以 *Separate Query from Modifier*（279）将函数副作用分离出去。稍后你会看到这方面的例子。

9.5 Replace Nested Conditional with Guard Clauses

以卫语句取代嵌套条件式

函数中的条件逻辑（conditional logic）使人难以看清正常的执行路径。

使用卫语句（guard clauses）表现所有特殊情况。

```
double getPayAmount() {
    double result;
    if (_isDead) result = deadAmount();
    else {
        if (_isSeparated) result = separatedAmount();
        else {
          if (_isRetired) result = retiredAmount();
          else result = normalPayAmount();
        };
    }
    return result;
};
```

```
double getPayAmount() {
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) return retiredAmount();
    return normalPayAmount();
};
```

动机（Motivation）

根据我的经验，条件式通常有两种呈现形式。第一种形式是：所有分支都属于正常行为。第二种形式则是：条件式提供的答案中只有一种是正常行为，其他都是不常见的情况。

这两类条件式有不同的用途，这一点应该通过代码表现出来。如果两条分支都是正常行为，就应该使用形如「if…else…」的条件式；如果某个条件极其罕见，就应该单独检查该条件，并在该条件为真时立刻从函数中返回。这样的单独检查常常被称为「卫语句（guard clauses）」 [Beck]。

Replace Nested Conditional with Guard Clauses（250）的精髓就是：给某一条分支以特别的重视。如果使用 if-then-else 结构，你对 if 分支和 else 分支的重视是同等的。这样的代码结构传递给阅读者的消息就是：各个分支有同样的重要性。卫语句（guard clauses）就不同了，它告诉阅读者：『这种情况很罕见，如果它真的发生了，请做一些必要的整理工作，然后退出。』

「每个函数只能有一个入口和一个出口」的观念，根深蒂固于某些程序员的脑海里。我发现，当我处理他们编写的代码时，我经常需要使用 *Replace Nested Conditional with Guard Clauses*（250）。现今的编程语言都会强制保证每个函数只有一个入口，至于「单一出口」规则，其实不是那么有用。在我看来，保持代码清晰才是最关键的：如果「单一出口」能使这个函数更清楚易读，那么就使用单一出口；否则就不必这么做。

作法（Mechanics）

- 对于每个检查，放进一个卫语句（guard clauses）。
  - ⇨ 卫语句要不就从函数中返回，要不就抛出一个异常（exception）。
- 每次将「条件检查」替换成「卫语句」后，编译并测试。
  - ⇨ 如果所有卫语句都导致相同结果，请使用 *Consolidate Conditional Expressions*（240）。

范例（Example）

想像一个薪册系统，其中以特殊规则处理死亡员工、驻外员工、退休员工的薪资。这些情况不常有，但的确偶而会出现。

假设我在这个系统中看到下列代码：

```
double getPayAmount() {
    double result;
    if (_isDead) result = deadAmount();
    else {
       if (_isSeparated) result = separatedAmount();
       else {
           if (_isRetired) result = retiredAmount();
           else result = normalPayAmount();
       };
    }
return result;
};
```

在这段代码中，非正常情况的检查掩盖了正常情况的检查，所以我应该使用「卫语句」来取代这些检查，以提高程序清晰度。我可以逐一引入卫语句。让我们从最上面的条件检查动作开始：

```
double getPayAmount() {
    double result;
    if (_isDead) return deadAmount();
    if (_isSeparated) result = separatedAmount();
    else {
        if (_isRetired) result = retiredAmount();
        else result = normalPayAmount();
    };
    return result;
};
```

然后，继续下去，仍然一次替换一个检查动作：

```
double getPayAmount() {
    double result;
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) result = retiredAmount();
    else result = normalPayAmount();
    return result;
};
```

然后是最后一个：

```
double getPayAmount() {
    double result;
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) return retiredAmount();
    result = normalPayAmount();
    return result;
};
```

此时，result 变量已经没有价值了，所以我把它删掉：

```
double getPayAmount() {
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) return retiredAmount();
    return normalPayAmount();
};
```

嵌套（nested）条件代码往往由那些深信「每个函数只能有一个出口」的程序员写出。我发现那条规则（函数只能有一个出口）实在有点太简单化了。如果对函数剩余部分不再有兴趣，当然应该立刻退出。引导阅读者去看一个没有用的 else 区段，

只会妨碍他们的理解。

范例：将条件逆反（Reversing the Conditions）

审阅本书初稿时，Joshua Kerievsky 指出：你常常可以将条件表达式逆反，从而实现 *Replace Nested Conditional with Guard Clauses*（250）。为了拯救我可怜的想像力，他还好心帮我想了个例子：

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital > 0.0) {
      if (_intRate > 0.0 && _duration > 0.0) {
          result = (_income / _duration) * ADJ_FACTOR;
      }
    }
    return result;
}
```

同样地，我逐一进行替换。不过这次在插入卫语句（guard clause）时，我需要将相应的条件逆反过来：

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (_intRate > 0.0 && _duration > 0.0) {
      result = (_income / _duration) * ADJ_FACTOR;
    }
    return result;
}
```

下一个条件稍微复杂一点，所以我分两步进行逆反。首先加入一个 "logical-NOT" 操作：

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (!(_intRate > 0.0 && _duration > 0.0)) return result;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

但是在这样的条件式中留下一个"logical-NOT"，会把我的脑袋拧成一团乱麻，所以我把它简化成下面这样：

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (_intRate <= 0.0 || _duration <= 0.0) return result;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

这时候我比较喜欢在卫语句（guard clause）内返回一个明确值，因为这样我可以一目了然地看到卫语句返回的失败结果。此外，这种时候我也会考虑使用 *Replace Magic Number with Symbolic Constant*（204）。

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

完成替换之后，我同样可以将临时变量移除：

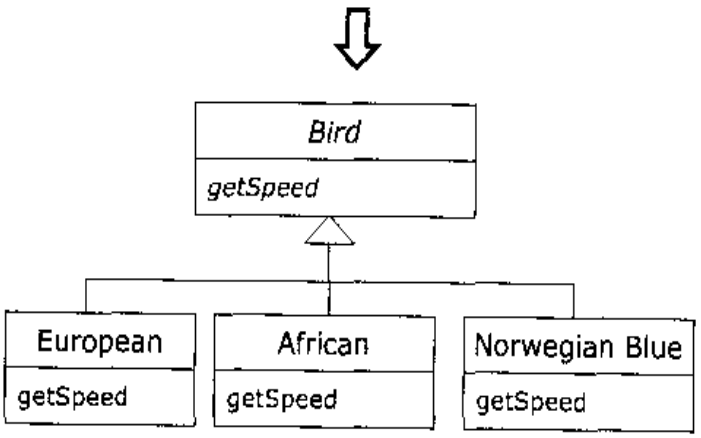
```
public double getAdjustedCapital() {
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    return (_income / _duration) * ADJ_FACTOR;
}
```

9.6 Replace Conditional with Polymorphism
以多态取代条件式

你手上有个条件式，它根据对象型别的不同而选择不同的行为。

将这个条件式的每个分支放进一个 subclass 内的覆写函数中，然后将原始函数声明为抽象函数（abstract method）。

```
double getSpeed() {
   switch (_type) {
     case EUROPEAN:
       return getBaseSpeed();
     case AFRICAN:
       return getBaseSpeed()-getLoadFactor()*_numberOfCoconuts;
     case NORWEGIAN_BLUE:
       return (_isNailed) ? 0 : getBaseSpeed(_voltage);
   }
   throw new RuntimeException ("Should be unreachable");
}
```



动机（Motivation）

在面向对象术语中，听上去最高贵的词非「多态」莫属。多态（polymorphism）最根本的好处就是：如果你需要根据对象的不同型别而采取不同的行为，多态使你不必编写明显的条件式（explicit conditional）。

正因为有了多态，所以你会发现：「针对 type code（型别码）而写的 switch 语句」以及「针对 type string（型别名称字符串）而写的 if-then-else 语句」在面向对象程序中很少出现。

多态（polymorphism）能够给你带来很多好处。如果同一组条件式在程序许多地点出现，那么使用多态的收益是最大的。使用条件式时，如果你想添加一种新型别，就必须查找并更新所有条件式。但如果改用多态，只需建立一个新的 subclass，并在其中提供适当的函数就行了。class 用户不需要了解这个 subclass，这就大大降低了系统各部分之间的相依程度，使系统升级更加容易。

作法（Mechanics）

使用 *Replace Conditional with Polymorphism*（255）之前，你首先必须有一个继承结构。你可能已经通过先前的重构得到了这一结构。如果还没有，现在就需要建立它。

要建立继承结构，你有两种选择：*Replace Type Code with Subclasses*（223）和 *Replace Type Code with State/Strategy*（227）。前一种作法比较简单，因此你应该尽可能使用它。但如果你需要在对象创建好之后修改 type code，就不能使用 subclassing 作法，只能使用 ***State/Strategy*** 模式。此外，如果由于其他原因你要重构的 class 已经有了 subclass，那么也得使用 ***State/Strategy***。记住，如果若干 switch 语句针对的是同一个 type code，你只需针对这个 type code 建立一个继承结构就行了。

现在，可以向条件式开战了。你的目标可能是 switch（case）语句，也可能是 if 语句。

- 如果要处理的条件式是一个更大函数中的一部分，首先对条件式进行分析，然后使用 *Extract Method*（110）将它提炼到一个独立函数去。
- 如果有必要，使用 *Move Method*（142）将条件式放置到继承结构的顶端。
- 任选一个 subclass，在其中建立一个函数，使之覆写 superclass 中容纳条件式的那个函数。将「与该 subclass 相关的条件式分支」拷贝到新建函数中，并对它进行适当调整。

  ⇨ 为了顺利进行这一步骤，你可能需要将 superclass 中的某些 private 值域声明为 protected。

- 编译，测试。
- 在 superclass 中删掉条件式内被拷贝出去的分支。
- 编译，测试。

- 针对条件式的每个分支，重复上述过程，直到所有分支都被移到 subclass 内的函数为止。
- 将 superclass 之中容纳条件式的函数声明为抽象函数（abstract method）。

范例（Example）

请允许我继续使用「员工与薪资」这个简单而又乏味的例子。我的 classes 是从 *Replace Type Code with State/Strategy*（227）那个例子中拿来的，因此示意图就如图 9.1 所示（如果想知道这个图是怎么得到的，请看第 8 章范例）。

```
class Employee...
    int payAmount() {
      switch (getType()) {
          case EmployeeType.ENGINEER:
              return _monthlySalary;
          case EmployeeType.SALESMAN:
              return _monthlySalary + _commission;
          case EmployeeType.MANAGER:
              return _monthlySalary + _bonus;
          default:
              throw new RuntimeException("Incorrect Employee");
      }
    }
```

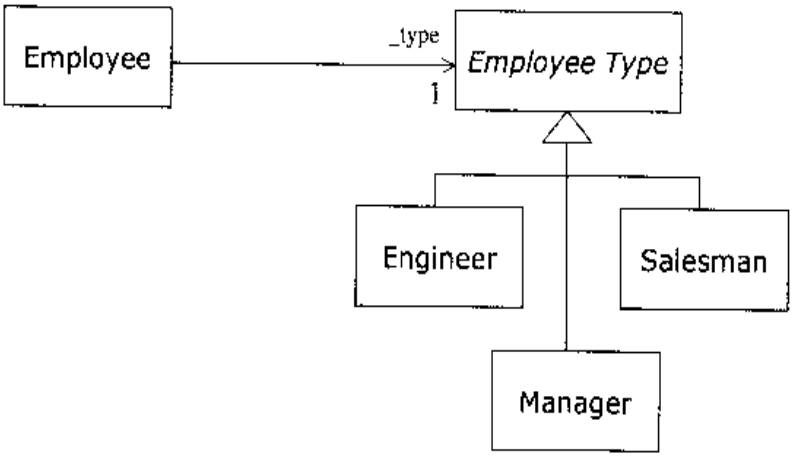


图 9.1 继承结构

```
    int getType() {
      return _type.getTypeCode();
    }
    private EmployeeType _type;

abstract class EmployeeType...
    abstract int getTypeCode();

class Engineer extends EmployeeType...
    int getTypeCode() {
      return Employee.ENGINEER;
    }

... and other subclasses
```

switch 语句已经被很好地提炼出来，因此我不必费劲再做一遍。不过我需要将它移到 EmployeeType class，因为 EmployeeType 才是被 subclassing 的 class。

```
class EmployeeType...
    int payAmount(Employee emp) {
      switch (getTypeCode()) {
          case ENGINEER:
              return emp.getMonthlySalary();
          case SALESMAN:
              return emp.getMonthlySalary() + emp.getCommission();
          case MANAGER:
              return emp.getMonthlySalary() + emp.getBonus();
          default:
              throw new RuntimeException("Incorrect Employee");
      }
    }
```

由于我需要 Employee class 的数据，所以我需要将 Employee 对象作为参数传递给 payAmount()。这些数据中的一部分也许可以移到 EmployeeType class 来，但那是另一项重构需要关心的问题了。

调整代码，使之通过编译，然后我修改 Employee 中的 payAmount()函数，令它委托（*delegate*，转调用）EmployeeType：

```
class Employee...
    int payAmount() {
      return _type.payAmount(this);
    }
```

现在，我可以处理 switch 语句了。这个过程有点像淘气小男孩折磨一只昆虫——每次掰掉它一条腿[6]。首先我把 switch 语句中的 "Engineer" 这一分支拷贝到 Engineer class：

```
class Engineer...
    int payAmount(Employee emp) {
      return emp.getMonthlySalary();
    }
```

这个新函数覆写了 superclass 中的 switch 语句之内那个专门处理 "Engineer" 的分支。我是个偏执狂，有时我会故意在 case 子句中放一个陷阱，检查 Engineer subclass 是否正常工作（是否被调用）：

```
class EmployeeType...
    int payAmount(Employee emp) {
      switch (getTypeCode()) {
          case ENGINEER:
              throw new RuntimeException (
                                  "Should be being overridden");
          case SALESMAN:
              return emp.getMonthlySalary()+ emp.getCommission();
          case MANAGER:
              return emp.getMonthlySalary() + emp.getBonus();
          default:
              throw new RuntimeException("Incorrect Employee");
      }
    }
```

接下来，我重复上述过程，直到所有分支都被去除为止：

```
class Salesman...
    int payAmount(Employee emp) {
      return emp.getMonthlySalary() + emp.getCommission();
    }
class Manager...
    int payAmount(Employee emp) {
      return emp.getMonthlySalary() + emp.getBonus();
    }
```

然后，将 superclass 的 payAmount() 函数声明为抽象函数：

```
class EmployeeType...
    abstract int payAmount(Employee emp);
```

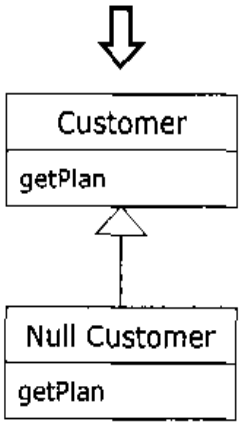
[6] 译注：「腿」和条件式「分支」的英文都是 "leg"。作者幽默地说「掰掉一条腿」，意思就是「去掉一个分支」。

9.7 Introduce Null Object（引入 Null 对象）

你需要再三检查「某物是否为 null value」。

将 null value（**无效值**）**替换为** null object（**无效物**）。

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```



动机（Motivation）

多态（polymorphism）的最根本好处在于：你不必再向对象询问「你是什么型别」而后根据得到的答案调用对象的某个行为——你只管调用该行为就是了，其他的一切多态机制会为你安排妥当。当你的某个值域内容是 null value 时，多态可扮演另一个较不直观（亦较不为人所知）的用途。让我们先听听 Ron Jeffries 的故事。

> Ron Jeffries
>
> 我们第一次使用 ***Null Object*** 模式，是因为 Rih Garzaniti 发现，系统在对对象发送一个消息之前，总要检查对象是否存在，这样的检查出现很多次。我们可能会向一个对象索求它所相关的 `Person` 对象，然后再问那个对象是否为 null。如果对象的确存在，我们才能调用它的 `rate()` 函数以查询这个人的薪资级别。我们在好些地方都是这样做的，造成的重复代码让我们很烦心。
>
> 所以，我们编写了一个 `MissingPerson` class，让它返回 '0' 薪资等级（我们把 null objects 称为 missing object（虚构对象）。很快地 `MissingPerson` 就有了很多函数，`rate()` 自然是其中之一。如今我们的系统有超过 80 个 null-object classes。
>
> 我们常常在显示信息的时候使用 null object。例如我们想要显示一个 `Person` 对象信息，它大约有 20 个 *instance* 变量。如果这些变量可被设为 null，那么打印一个 `Person` 对象的工作将非常复杂。所以我们不让 *instance* 变量被设为 null，而是插入各式各样

的 null objects——它们都知道如何正常（正确地）显示自己。这样，我们就可以摆脱大量代码。

我们对 null objects 的最聪明运用，就是拿它来表示不存在的 Gemstone session。我们使用 Gemstone 数据库来保存成品（程序代码），但我们更愿意在没有数据库的情况下进行开发，每过一周左右再把新码放进 Gemstone 数据库。然而在代码的某些地方，我们必须登录（log in）一个 Gemstone session。当我们没有 Gemstone 数据库时，我们就仅仅安插一个 missing Gemstone session，其接口和真正的 Gemstone session 一模一样，使我们无需判断数据库是否存在，就可以进行开发和测试。

null object 的另一个用途是表现出「虚构的箱仓」（missing bin）。所谓「箱仓」，这里是指群集（collection），用来保存某些薪资值，并常常需要对各个薪资值进行加和或遍历。如果某个箱仓不存在，我们就给出一个虚构的箱仓对象，其行为和一个空箱仓（empty bin）一样：这个虚构箱仓知道自己其实不带任何数据，总值为 0。通过这种作法，我们就不必为上千位员工每人产生数十来个空箱（empty bins）对象了。

使用 null objects 时有个非常有趣的性质：好事绝对不会因为 null objects 而「被破坏」。由于 null objects 对所有外界请求的响应，都像 real objects 的响应一样，所以系统行为总是正常的。但这并非总是好事，有时会造成问题的侦测和查找上的困难，因为从来没有任何东西被破坏。当然，只要认真检查一下，你就会发现 null objects 有时出现在不该出现的地方。

请记住：null objects 一定是常量，它们的任何成分都不会发生变化。因此我们可以使用 ***Singleton*** 模式 [Gang of Four] 来实现它们。例如不管任何时候，只要你索求一个 `MissingPerson` 对象，你得到的一定是 `MissingPerson` class 的惟一实体。

关于 ***Null Object*** 模式，你可以在 Woolf [Woolf] 中找到更详细的介绍。

作法（Mechanics）

- 为 source class 建立一个 subclass，使其行为像 source class 的 null 版本。在 source class 和 null class 中都加上 `isNull()` 函数，前者的 `isNull()` 应该返回 false，后者的 `isNull()` 应该返回 true。
  - ⇨ 下面这个办法也可能对你有所帮助：建立一个 nullable 接口，将 `isNull()` 函数放在其中，让 source class 实现这个接口。
  - ⇨ 另外，你也可以创建一个 testing 接口，专门用来检查对象是否为 null。
- 编译。
- 找出所有「索求 source object 却获得一个 null」的地方。修改这些地方，使它们改而获得一个 null object。

- 找出所有「将 source object 与 null 做比较」的地方。修改这些地方，使它们调用 isNull()函数。
  - ⇨ 你可以每次只处理一个 source object 及其客户程序，编译并测试后，再处理另一个 source object。
  - ⇨ 你可以在「不该再出现 null value」的地方放上一些 assertions（断言），确保 null 的确不再出现。这可能对你有所帮助。
- 编译，测试。
- 找出这样的程序点：如果对象不是 null，做 A 动作，否则做 B 动作。
- 对于每一个上述地点，在 null class 中覆写 A 动作，使其行为和 B 动作相同。
- 使用上述的被覆写动作（A），然后删除「对象是否等于 null」的条件测试。编译并测试。

范例（Example）

一家公用事业公司的系统以 Site 表示地点（场所）。庭院宅第（house）和集合公寓（apartment）都使用该公司的服务。任何时候每个地点都拥有（或说都对应于）一个顾客，顾客信息以 Customer 表示：

```
class Site...
   Customer getCustomer() {
      return _customer;
   }
   Customer _customer;
```

Customer 有很多特性，我们只看其中三项：

```
class Customer...
   public String getName() {...}
   public BillingPlan getPlan() {...}
   public PaymentHistory getHistory() {...}
```

本系统又以 PaymentHistory 表示顾客的付款记录，它也有它自己的特性：

```
public class PaymentHistory...
    int getWeeksDelinquentInLastYear() //译注：delinquent：拖欠。
```

上面的各种取值函数（getter）允许客户取得各种数据。但有时候一个地点的顾客搬走了，新顾客还没搬进来，此时这个地点就没有顾客。由于这种情况有可能发生，所以我们必须保证 Customer 的所有用户都能够处理「Customer 对象等于 null」的情况。下面是一些示例片段：

```
Customer customer = site.getCustomer();
BillingPlan plan;
```

```
    if (customer == null) plan = BillingPlan.basic();
    else plan = customer.getPlan();
...
    String customerName;
    if (customer == null) customerName = "occupant";
    else customerName = customer.getName();
...
    int weeksDelinquent;
    if (customer == null) weeksDelinquent = 0;
    else weeksDelinquent =
        customer.getHistory().getWeeksDelinquentInLastYear();
```

这个系统中可能使用许多 Site 和 Customer，它们都必须检查 Customer 对象是否等于 null，而这样的检查完全是重复的。看来是使用 null object 的时候了。

首先新建一个 NullCustomer，并修改 Customer，使其支持「对象是否为 null」的检查：

```
class NullCustomer extends Customer {
    public boolean isNull() {
      return true;
    }
}
class Customer...
    public boolean isNull() {
      return false;
    }
    protected Customer() {} //needed by the NullCustomer
```

如果你无法修改 Customer，你可以使用 p.266 的作法：建立一个新的 testing 接口。

如果你喜欢，也可以新建一个接口，昭告大家「这里使用了 null object」：

```
interface Nullable {
    boolean isNull();
}
class Customer implements Nullable
```

我还喜欢加入一个 ***factory method***，专门用来创建 NullCustomer 对象。这样一来，用户就不必知道 null class 的存在了：

```
class Customer...
    static Customer newNull() {
      return new NullCustomer();
    }
```

接下来的部分稍微有点麻烦。对于所有「返回 null」的地方，我都要将它改为「返回 null object」，此外我还要把 foo==null 这样的检查替换成 foo.isNull()。我发

现下列办法很有用：查找所有「索求 Customer 对象」的地方，将它们都加以修改，使它们不能返回 null，改而返回一个 NullCustomer 对象。

```
class Site...
    Customer getCustomer() {
       return (_customer == null) ?
                 Customer.newNull():
                 _customer;
    }
```

另外，我还要修改所有「使用 Customer 对象」的地方，让它们以 isNull()函数进行检查，不再使用 "== null" 检查方式。

```
    Customer customer = site.getCustomer();
    BillingPlan plan;
    if (customer.isNull()) plan = BillingPlan.basic();
    else plan = customer.getPlan();
...
    String customerName;
    if (customer.isNull()) customerName = "occupant";
    else customerName = customer.getName();
...
    int weeksDelinquent;
    if (customer.isNull()) weeksDelinquent = 0;
    else weeksDelinquent =
        customer.getHistory().getWeeksDelinquentInLastYear();
```

毫无疑问，这是本项重构中最需要技巧的部分。对于每一个需要替换的「可能等于 null」的对象，我都必须找到「它是否等于 null」的所有检查动作，并逐一替换。如果这个对象被传播到很多地方，追踪起来就很困难。上述范例中，我必须找出每一个型别为 Customer 的变量，以及它们被使用的地点。很难将这个过程分成更小的步骤。有时候我发现「可能等于 null」的对象只在某几处被用到，那么替换工作比较简单。但是大多数时候我必须做大量替换工作。还好，撤销这些替换并不困难，因为我可以不太困难地找出对 isNull()的调用动作，但这毕竟也是很零乱很恼人的。

这个步骤完成之后，如果编译和测试都顺利通过，我就可以宽心地露出笑容了。接下来的动作比较有趣。到目前为止，使用 isNull()函数尚未带来任何好处。只有当我把相关行为移到 NullCustomer class 中并去除条件式之后，我才能得到切实的利益。我可以逐一将各种行为（函数）移过去。首先从「取得顾客名称」这个函数开始。此时的客户端代码大约如下：

```
String customerName;
if (customer.isNull()) customerName = "occupant";
else customerName = customer.getName();
```

首先为 NullCustomer 加入一个合适的函数，通过这个函数来取得顾客名称：

```
class NullCustomer...
    public String getName(){
      return "occupant";
    }
```

现在，我可以去掉条件代码了：

```
String customerName = customer.getName();
```

接下来我以相同手法处理其他函数，使它们对相应查询做出合适的响应。此外我还可以对「修改函数」（modifiers）做适当的处理。于是下面这样的客户端程序：

```
if (! customer.isNull())
    customer.setPlan(BillingPlan.special());
```

就变成了这样：

```
customer.setPlan(BillingPlan.special());

class NullCustomer...
    public void setPlan (BillingPlan arg) {}
```

请记住：只有当大多数客户代码都要求 null object 做出相同响应时，这样的行为搬移才有意义。注意我说的是「大多数」而不是「所有」。任何用户如果需要 null object 作出不同响应，他仍然可以使用 isNull()函数来测试。只要大多数客户端都要求 null object 做出相同响应，他们就可以调用缺省的 null 行为，而你也就受益匪浅了。

上述范例略带差异的某种情况是，某些客户端使用 Customer 函数的运算结果：

```
if (customer.isNull()) weeksDelinquent = 0;
else weeksDelinquent =
    customer.getHistory().getWeeksDelinquentInLastYear();
```

我可以新建一个 NullPaymentHistory class，用以处理这种情况：

```
class NullPaymentHistory extends PaymentHistory...
    int getWeeksDelinquentInLastYear() {
      return 0;
   }
```

并修改 NullCustomer，让它返回一个 NullPaymentHistory 对象：

```
class NullCustomer...
    public PaymentHistory getHistory() {
      return PaymentHistory.newNull();
    }
```

然后，我同样可以删除这一行条件代码：

```
int weeksDelinquent =
    customer.getHistory().getWeeksDelinquentInLastYear();
```

你常常可以看到这样的情况：null objects 会返回其他 null objects。

范例：另一种作法，Testing Interface

除了定义 isNull()之外，你也可以建立一个用以检查「对象是否为 null」的接口。使用这种办法，必须新建一个 Null 接口，其中不定义任何函数：

```
interface Null {}
```

然后，让 null object 实现 Null 接口：

```
class NullCustomer extends Customer implements Null...
```

然后，我就可以用 instanceof 操作符检查对象是否为 null：

```
aCustomer instanceof Null
```

通常我尽量避免使用 instanceof 操作符，但在这种情况下，使用它是没问题的。而且这种作法还有另一个好处：不需要修改 Customer。这么一来即使无法修改 Customer 源码，我也可以使用 null object。

其他特殊情况

使用本项重构时，你可以有数种不同的 null objects，例如你可以说「没有顾客」（新建的房子和暂时没人住的房子）和「不知名顾客」（有人住，但我们不知道是谁）这两种情况是不同的。果真如此，你可以针对不同的情况建立不同的 null class。有时候 null objects 也可以携带数据，例如不知名顾客的使用记录等等，于是我们可以在查出顾客姓名之后将帐单寄给他。

本质上来说，这是一个比 ***Null Object*** 模式更大的模式：***Special Case*** 模式。所谓 special case class（特例类）是某个 class 的特殊情况，有着特殊的行为。因此表示「不知名顾客」的 UnknownCustomer 和表示「没有顾客」的 NoCustomer 都是 Customer 的特例。你经常可以在表示数量的 classes 中看到这样的「特例类」，例如 Java 浮点数有「正无穷大」、「负无穷大」和「非数量」（NaN）等特例。special case class（特例类）的价值是：它们可以降低你的「错误处理」开销。例如浮点运算决不会抛出异常。如果你对 NaN 做浮点运算，结果也会是个 NaN。这和「null object 的访问函数通常返回另一个 null object」是一样的道理。

9.8 Introduce Assertion（引入断言）

某一段代码需要对程序状态（state）做出某种假设。

以 assertion（断言）明确表现这种假设。

```
double getExpenseLimit() {
    // should have either expense limit or a primary project
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

```
double getExpenseLimit() {
    Assert.isTrue (_expenseLimit != NULL_EXPENSE ||
    _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

动机（Motivation）

常常会有这样一段代码：只有当某个条件为真时，该段代码才能正常运行。例如「平方根计算」只对正值才能进行（译注：这里没考虑复数与虚数），又例如某个对象可能假设其值域（fields）至少有一个不等于 null。

这样的假设通常并没有在代码中明确表现出来，你必须阅读整个算法才能看出。有时程序员会以注释写出这样的假设。而我要介绍的是一种更好的技术：使用 assertion（断言）明确标明这些假设。

assertion 是一个条件式，应该总是为真。如果它失败，表示程序员犯了错误。因此 assertion 的失败应该导致一个 *unchecked* exception[7]（不可控异常）。Assertions 绝对不能被系统的其他部分使用。实际上程序最后成品往往将 assertions 统统删除。因此，标记「某些东西是个 assertion」是很重要的。

[7] 译注：所谓 *unchecked* exception 是指「未曾于函数签名式（signature）中列出」的异常。

Assertions 可以作为交流与调试的辅助。在交流（沟通）的角度上，assertions 可以帮助程序阅读者理解代码所做的假设；在调试的角度上，assertions 可以在距离「臭虫」最近的地方抓住它们。当我编写自我测试代码的时候，我发现，assertions 在调试方面的帮助变得不那么重要了，但我仍然非常看重它们在交流方面的价值。

作法（Mechanics）

如果程序员不犯错，assertions 就应该不会对系统运行造成任何影响，所以加入 assertions 永远不会影响程序的行为。

- 如果你发现代码「假设某个条件始终（必须）为真」，就加入一个 assertion 明确说明这种情况。
  - ⇨ 你可以新建一个 `Assert` class，用于处理各种情况下的 assertions。

注意，不要滥用 assertions。请不要使用它来检查你「认为应该为真」的条件，请只使用它来检查「一定必须为真」的条件。滥用 assertions 可能会造成难以维护的重复逻辑。在一段逻辑中加入 assertions 是有好处的，因为它迫使你重新考虑这段代码的约束条件。如果「不满足这些约束条件，程序也可以正常运行」，assertions 就不会带给你任何帮助，只会把代码变得混乱，并且有可能妨碍以后的修改。

你应该常常问自己：如果 assertions 所指示的约束条件不能满足，代码是否仍能正常运行？如果可以，就把 assertions 拿掉。

另外，还需要注意 assertions 中的重复代码。它们和其他任何地方的重复代码一样不好闻。你可以大胆使用 *Extract Method*（110）去掉那些重复代码。

范例（Example）

下面是一个简单例子：开支（经费）限制。后勤部门的员工每个月有固定的开支限额；业务部门的员工则按照项目的开支限额来控制自己的开支。一个员工可能没有开支额度可用，也可能没有参与项目，但两者总得要有一个（否则就没有经费可用了）。在开支限额相关程序中，上述假设总是成立的，因此：

```
class Employee...
    private static final double NULL_EXPENSE = -1.0;
    private double _expenseLimit = NULL_EXPENSE;
    private Project _primaryProject;
```

```
    double getExpenseLimit() {
      return (_expenseLimit != NULL_EXPENSE) ?
          _expenseLimit:
          _primaryProject.getMemberExpenseLimit();
    }
    boolean withinLimit (double expenseAmount) {
      return (expenseAmount <= getExpenseLimit());
  }
```

这段代码包含了一个明显假设：任何员工要不就参与某个项目，要不就有个人开支限额。我们可以使用 assertion 在代码中更明确地指出这一点：

```
double getExpenseLimit() {
    Assert.isTrue (_expenseLimit != NULL_EXPENSE ||
                   _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
      _expenseLimit:
      _primaryProject.getMemberExpenseLimit();
}
```

这条 assertion 不会改变程序的任何行为。另一方面，如果 assertion 中的条件不为真，我就会收到一个运行期异常：也许是在 withinLimit()函数中抛出一个空指针（null pointer）异常，也许是在 Assert.isTrue()函数中抛出一个运行期异常。有时 assertion 可以帮助程序员找到臭虫，因为它离出错地点很近。但是，更多时候，assertion 的价值在于：帮助程序员理解代码正确运行的必要条件。

我常对 assertion 中的条件式使用 *Extract Method*（110），也许是为了将若干地方的重复码提炼到同一个函数中，也许只是为了更清楚说明条件式的用途。

在 Java 中使用 assertions 有点麻烦：没有一种简单机制可以协助我们插入这东西[8]。assertions 可被轻松拿掉，所以它们不可能影响最终成品的性能。编写一个辅助类（例如 **Assert** class）当然有所帮助，可惜的是 assertions 参数中的任何表达式不论什么情况都一定会被执行一遍。阻止它的惟一办法就是使用类似下面的手法：

```
double getExpenseLimit() {
    Assert.isTrue (Assert.ON &&
      (_expenseLimit != NULL_EXPENSE ||
       _primaryProject != null));
    return (_expenseLimit != NULL_EXPENSE) ?
      _expenseLimit:
      _primaryProject.getMemberExpenseLimit();
}
```

[8] 译注：J2SE 1.4 已经支持 assert 语句。

或者是这种手法：

```
double getExpenseLimit() {
    if (Assert.ON)
      Assert.isTrue (_expenseLimit != NULL_EXPENSE ||
                     _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
      _expenseLimit :
      _primaryProject.getMemberExpenseLimit();
}
```

如果 Assert.ON 是个常量，编译器（译注：而非运行期间）就会对它进行检查；如果它等于 false，就不再执行条件式后半段代码。但是，加上这条语句实在有点丑陋，所以很多程序员宁可仅仅使用 Assert.isTrue()函数，然后在项目结束前以过滤程序滤掉使用 assertions 的每一行代码（可以使用 Perl 之类的语言来编写这样的过滤程序）。

Assert class 应该有多个函数，函数名称应该帮助程序员理解其功用。除了 isTrue()之外，你还可以为它加上 equals()和 shouldNeverReachHere()等函数。

10

简化函数调用

Making Method Calls Simpler

在对象技术中，最重要的概念莫过于「接口」（interface）。容易被理解和被使用的接口，是开发良好面向对象软件的关键。本章将介绍「使接口变得更简洁易用」的重构手法。

最简单也最重要的一件事就是修改函数名称。「名称」是程序写作者与阅读者交流的关键工具。只要你能理解一段程序的功能，就应该大胆地使用 *Rename Method*（273）将你所知道的东西传达给其他人。另外，你也可以（并且应该）在适当时机修改变量名称和 class 名称。不过，总体来说，「修改名称」只是相对比较简单的文本替换功夫，所以我没有为它们提供单独的重构项目。

函数参数在「接口」之中扮演十分重要的角色。*Add Parameter*（275）和 *Remove Parameter*（277）都是很常见的重构手法。初始接触面向对象技术的程序员往往使用很长的参数列（parameter lists），这在其他开发环境中是很典型的方式。但是，使用对象技术，你可以保持参数列的简短，以下有一些相关的重构可以帮助你缩短参数列。如果来自同一对象的数个值被当作参数传递，你可以运用 *Preserve Whole Object*（288）将它们替换为单一对象，从而缩短参数列。如果此前并不存在这样一个对象，你可以运用 *Introduce Parameter Object*（295）将它创建出来。如果函数参数来自该函数可取用的一个对象，则可以使用 *Replace Parameter with Method*（292）避免传递参数。如果某些参数被用来在条件式中做选择依据，你可以实施 *Replace Parameter with Explicit Method*（285）。另外，你还可以使用 *Parameterize Method*（283）为数个相似函数添加参数，将它们合并到一起。

关于缩减参数列的重构手法，Doug Lea 对我提出了一个警告：并发编程（con-current programming）往往需要使用较长的参数列，因为这样你可以保证传递给函数的参数都是不可被修改的，就像内置型对象和 *value* object 一定地不可变。通常，你可

以使用不可变对象（immutable objects）取代这样的长参数列，但另一方面你也必须对此类重构保持谨慎。

多年来我一直坚守一个很有价值的习惯：明确地将「修改对象状态」的函数（修改函数，modifiers）和「查询对象状态」的函数（查询函数，queries）分开设计。不知道多少次，我因为将这两种函数混在一起而麻烦缠身；不知道多少次，我看到别人也因为同样的原因而遇到同样的麻烦。因此，如果我看到这两种函数混在一起，我就使用 *Separate Query from Modifier*（279）将它们分开。

良好的接口只向用户展现必须展现的东西。如果一个接口暴露了过多细节，你可以将不必要暴露的东西隐藏起来，从而改进接口的质量。毫无疑问，所有数据都应该隐藏起来（希望你不需要我来告诉你这一点），同时，所有可以隐藏的函数都应该被隐藏起来。进行重构时，你往往需要暂时暴露某些东西，最后再以 *Hide Method*（303）和 *Remove Setting Method*（300）将它们隐藏起来。

构造函数（constructors）是 Java 和 C++ 中特别麻烦的一个东西，因为它强迫你必须知道「待建对象」属于哪一个 class，而你往往并不需要知道这一点。你可以使用 *Replace Constructor with Factory Method*（304）避免了解这「被迫了解的一点」。

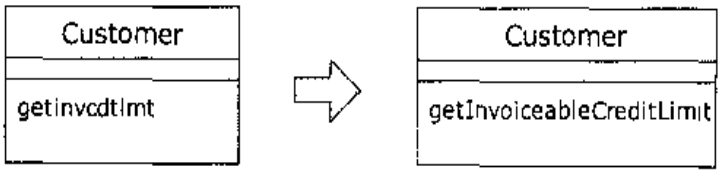
转型（casting）是 Java 程序员心中另一处永远的痛。你应该尽量使用 *Encapsulate Downcast*（308）将「向下转型动作」封装隐藏起来，避免让 class 用户做那种动作。

和许多现代编程语言一样，Java 也有异常处理（exception-handling）机制，这使得错误处理（error handling）相对容易一些。不习惯使用异常的程序员，往往会以错误代码（error codes）表示程序遇到的麻烦。你可以使用 *Replace Error Code with Exception*（310）来运用这些崭新的异常特性。但有时候异常也并不是最合适的选择，你应该实施 *Replace Exception with Test*（315）先测试一番。

10.1 Rename Method（重新命名函数）

函数的名称未能揭示函数的用途。

修改函数名称。



动机（Motivation）

我极力提倡的一种编程风格就是：将复杂的处理过程分解成小函数。但是，如果做得不好，这会使你费尽周折却弄不清楚这些小函数各自的用途。要避免这种麻烦，关键就在于给函数起一个好名称。函数的名称应该准确表达它的用途。给函数命名有一个好办法：首先考虑应该给这个函数写上一句怎样的注释，然后想办法将注释变成函数名称。

人生不如意，十之八九。你常常无法第一次就给函数起一个好名称。这时候你可能会想：就这样将就着吧——毕竟只是一个名称而已。当心！这是恶魔的召唤，是通向混乱之路，千万不要被它诱惑！如果你看到一个函数名称不能很好地表达它的用途，应该马上加以修改。记住，你的代码首先是为人写的，其次才是为计算器写的。而人需要良好名称的函数。想想过去曾经浪费的无数时间吧。如果给每个函数都起一个良好的名称，也许你可以节约好多时间。起一个好名称并不容易，需要经验；要想成为一个真正的编程高手，“起名称”的水平是至关重要的。当然，函数签名式（signature）中的其他部分也一样重要：如果重新安排参数顺序，能够帮助提高代码的清晰度，那就大胆地去做吧，你有 *Add Parameter*（275）和 *Remove Parameter*（277）这两项武器。

作法（Mechanics）

- 检查函数签名式（signature）是否被 superclass 或 subclass 实现过。如果是，则需要针对每份实现品分别进行下列步骤。
- 声明一个新函数，将它命名为你想要的新名称。将旧函数的代码拷贝到新函数中，并进行适当调整。
- 编译。

- 修改旧函数，令它将调用转发给新函数。
  - ⇨ 如果只有少数几个地方引用旧函数，你可以大胆地跳过这一步骤。
- 编译，测试。
- 找出旧函数的所有被引用点，修改它们，令它们改而引用新函数。每次修改后，编译并测试。
- 删除旧函数。
  - ⇨ 如果旧函数是class public接口的一部分，你可能无法安全地删除它。这种情况下，将它保留在原处，并将它标记为"deprecated"（不再被赞同）。
- 编译，测试。

范例（Example）

我以getTelephoneNumber()函数来取得某人的电话号码：

```
public String getTelephoneNumber() {
    return ("(" + _officeAreaCode + ") " - _officeNumber);
}
```

现在，我想把这个函数改名为getOfficeTelephoneNumber()。首先建立一个新函数，命名为getOfficeTelephoneNumber()，并将原函数getTelephoneNumber()的代码拷贝过来。然后，让旧函数直接调用新函数：

```
class Person...
    public String getTelephoneNumber(){
       return getOfficeTelephoneNumber();
    }
    public String getOfficeTelephoneNumber() {
       return ("(" + _officeAreaCode + ") " + _officeNumber);
    }
```

现在，我需要找到旧函数的所有调用者，将它们全部改为调用新函数。全部修改完后，就可以将旧函数删掉了。

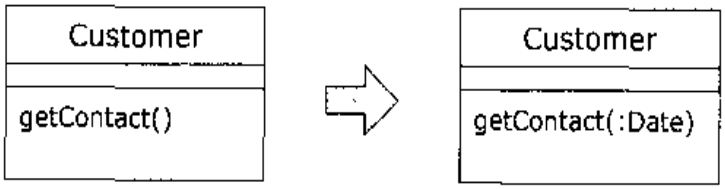
如果需要添加或去除某个参数，过程也大致相同。

如果旧函数的调用者并不多，我可以直接修改这些调用者，令它们调用新函数，不必让旧函数充当中介。如果测试出错，我可以回到起始处，并放慢前进速度。

10.2 Add Parameter（添加参数）

某个函数需要从调用端得到更多信息。

为此函数添加一个对象参数，让该对象带进函数所需信息。



动机（Motivation）

Add Parameter（275）是一个很常用的重构手法，我几乎可以肯定你已经用过它了。使用这项重构的动机很简单：你必须修改一个函数，而修改后的函数需要一些过去没有的信息，因此你需要给该函数添加一个参数。

实际上我比较需要说明的是：不使用本重构的时机。除了添加参数外，你常常还有其他选择。只要可能，其他选择都比本项「添加参数」要好，因为它们不会增加参数列的长度。过长的参数列是不好的味道，因为程序员很难记住那么多参数，而且长参数列往往伴随着坏味道 Data Clumps。

请看看现有的参数，然后问自己：你能从这些参数得到所需的信息吗？如果回答是否定的，有可能通过某个函数提供所需信息吗？你究竟把这些信息用于何处？这个函数是否应该属于拥有该信息的那个对象所有？看看现有参数，考虑一下，加入新参数是否合适？也许你应该考虑使用 *Introduce Parameter Object*（295）。

我并非要你绝对不要添加参数。事实上我自己经常添加参数，但是在添加参数之前你有必要了解其他选择。

作法（Mechanics）

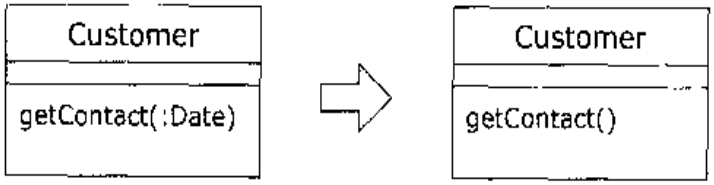
Add Parameter（275）的作法和 *Rename Method*（273）非常相似。

- 检查函数签名式（signature）是否被 superclass 或 subclass 实现过。如果是，则需要针对每份实现分别进行下列步骤。
- 声明一个新函数，名称与原函数同，只是加上新添参数。将旧函数的代码拷贝到新函数中。
  - ⇨ 如果需要添加的参数不止一个，将它们一次性添加进去比较容易。
- 编译。
- 修改旧函数，令它调用新函数。
  - ⇨ 如果只有少数几个地方引用旧函数，你大可放心地跳过这一步骤。
  - ⇨ 此时，你可以给参数提供任意值。但一般来说，我们会给对象参数提供 null，给内置型参数提供一个明显非正常值。对于数值型参数，我建议使用 0 以外的值，这样你比较容易将来认出它。
- 编译，测试。
- 找出旧函数的所有被引用点，将它们全部修改为对新函数的引用。每次修改后，编译并测试。
- 删除旧函数。
  - ⇨ 如果旧函数是 class public 接口的一部分，你可能无法安全地删除它。这种情况下，请将它保留在原地，并将它标示为 "deprecated"（不再被赞同）。
- 编译，测试。

10.3 Remove Parameter（移除参数）

函数本体（method body）不再需要某个参数。

将该参数去除。



动机（Motivation）

程序员可能经常添加参数，却往往不愿意去掉它们。他们打的如意算盘是，无论如何，多余的参数不会引起任何问题，而且以后还可能用上它。

这也是恶魔的诱惑，一定要把它从脑子里赶出去！参数指出函数所需信息，不同的参数值代表不同的意义。函数调用者必须为每一个参数操心该传什么东西进去。如果你不去掉多余参数，你就是让你的每一位用户多费一份心。这是很不划算的，尤其「去除参数」是非常简单的一项重构。

但是，对于多态函数（polymorphic method），情况有所不同。这种情况下，可能多态函数的另一份（或多份）实现码会使用这个参数，此时你就不能去除它。你可以添加一个独立函数，在这些情况下使用，不过你应该先检查调用者如何使用这个函数，以决定是否值得这么做。如果某些调用者已经知道他们正在处理的是一个特定的 subclass，并且已经做了额外工作找出自己需要的参数，或已经利用对 classes 体系的了解来避免取到 null，那么就值得你建立一个新函数，去除那多余参数。如果调用者不需要了解该函数所属的 class，你也可以保持调用者无知（而幸福）的状态。

作法（Mechanics）

Remove Parameter（277）的作法和 *Rename Method*（273）、*Add Parameter*（275）非常相似。

- 检查函数签名式（signature）是否被 superclass 或 subclass 实现过。如果是，则需要针对每份实现品分别进行下列步骤。
- 声明一个新函数，名称与原函数同，只是去除不必要的参数。将旧函数的代码拷贝到新函数中。
  - ⇨ 如果需要去除的参数不止一个，将它们一次性去除比较容易。
- 编译。
- 修改旧函数，令它调用新函数。
  - ⇨ 如果只有少数几个地方引用旧函数，你大可放心地跳过这一步骤。
- 编译，测试。
- 找出旧函数的所有被引用点，将它们全部修改为对新函数的引用。每次修改后，编译并测试。
- 删除旧函数。
  - ⇨ 如果旧函数是 class public 接口的一部分，你可能无法安全地删除它。这种情况下，将它保留在原处，并将它标记为 "deprecated"（不再被赞同）。
- 编译，测试。

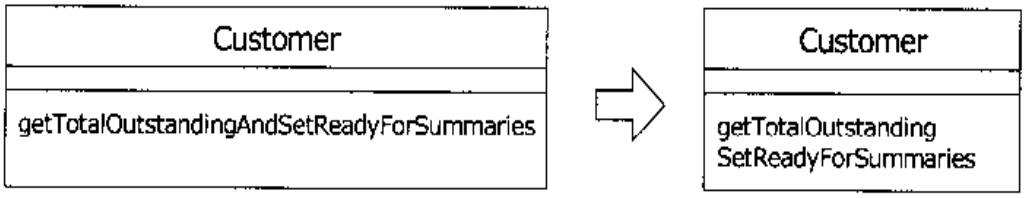
由于我可以轻松地添加、去除参数，所以我经常一次性地添加或去除必要的参数。

10.4 Separate Query from Modifier

将查询函数和修改函数分离

某个函数既返回对象状态值，又修改对象状态（state）。

建立两个不同的函数，其中一个负责查询，另一个负责修改。



动机（Motivation）

如果某个函数只是向你提供一个值，没有任何看得到的副作用（或说连带影响），那么这是个很有价值的东西。你可以任意调用这个函数，也可以把调用动作搬到函数的其他地方。简而言之，需要操心的事情少多了。

明确表现出「有副作用」与「无副作用」两种函数之间的差异，是个很好的想法。下面是一条好规则：任何有返回值的函数，都不应该有看得到的副作用。有些程序员甚至将此作为一条必须遵守的规则 [Meyer]。就像对待任何东西一样，我并不绝对遵守它，不过我总是尽量遵守，而它也回报我很好的效果。

如果你遇到一个「既有返回值又有副作用」的函数，就应该试着将查询动作从修改动作中分割出来。

你也许已经注意到了：我使用「看得到的副作用」这种说法。有一种常见的优化办法是：将查询所得结果高速缓存（cache）于某个值域中，这么一来后续的重复查询就可以大大加快速度。虽然这种作法改变了对象的状态，但这一修改是察觉不到的，因为不论如何查询，你总是获得相同结果 [Meyer]。

作法（Mechanics）

- 新建一个查询函数，令它返回的值与原函数相同。
  - ⇨ 观察原函数，看它返回什么东西。如果返回的是一个临时变量，找出临时变量的位置。
- 修改原函数，令它调用查询函数，并返回获得的结果。
  - ⇨ 原函数中的每个return句都应该像这样：return newQuery();，而不应该返回其他东西。
  - ⇨ 如果调用者将返回值赋给了一个临时变量，你应该能够去除这个临时变量。
- 编译，测试。
- 将「原函数的每一个被调用点」替换为「对查询函数的调用」。然后，在调用查询函数的那一行之前，加上对原函数的调用。每次修改后，编译并测试。
- 将原函数的返回值改为void，并删掉其中所有的return句。

范例（Example）

有这样一个函数：一旦有人入侵安全系统，它会告诉我入侵者的名字，并发送一个警报。如果入侵者不止一个，也只发送一条警报：

```
String foundMiscreant(String[] people){
    for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don")){
          sendAlert();
          return "Don";
      }
      if (people[i].equals ("John")){
          sendAlert();
          return "John";
      }
    }
    return "";
}
```

该函数被下列代码调用：

```
void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}
```

为了将查询动作和修改动作分开，我首先建立一个适当的查询函数，使其与修改函数返回相同的值，但不造成任何副作用：

```
String foundPerson(String[] people){
    for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don")){
          return "Don";
      }
      if (people[i].equals ("John")){
          return "John";
      }
    }
    return "";
}
```

然后，我要逐一替换原函数内所有的 return 句，改调用新建的查询函数。每次替换后，编译并测试。这一步完成之后，原函数如下所示：

```
String foundMiscreant(String[] people){
    for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don")){
          sendAlert();
          return foundPerson(people);
      }
      if (people[i].equals ("John")){
          sendAlert();
          return foundPerson(people);
      }
    }
    return foundPerson(people);
}
```

现在，我要修改调用者，将原本的单一调用动作替换为两个调用：先调用修改函数，然后调用查询函数：

```
void checkSecurity(String[] people) {
    foundMiscreant(people);
    String found = foundPerson(people);
    someLaterCode(found);
}
```

所有调用都替换完毕后，我就可以将修改函数的返回值改为 void：

```
void foundMiscreant (String[] people){
    for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don")){
          sendAlert();
          return;
      }
```

```
        if (people[i].equals ("John")){
            sendAlert();
            return;
        }
    }
}
```

现在，为原函数改个名称可能会更好一些：

```
void sendAlert (String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")){
            sendAlert();
            return;
        }
        if (people[i].equals ("John")){
            sendAlert();
            return;
        }
    }
}
```

当然，这种情况下，我得到了大量重复代码，因为修改函数之中使用了与查询函数相同的代码。现在我可以对修改函数实施 *Substitute Algorithm*（139），设法让它再简洁一些：

```
void sendAlert(String[] people){
    if (! foundPerson(people).equals(""))
        sendAlert();
}
```

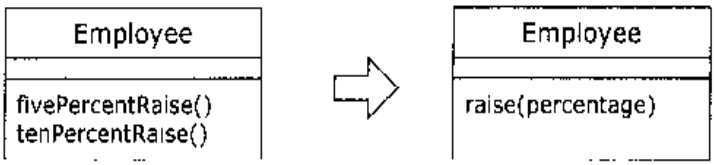
并发（Concurrency）问题

如果你在一个多线程系统中工作，肯定知道这样一个重要的惯用手法：在同一个动作中完成检查和赋值。这是否和 *Separate Query from Modifier*（279）互相矛盾呢？我曾经和 Doug Lea 讨论过这个问题，并得出结论：两者并不矛盾，但你需要做一些额外工作。将查询动作和修改动作分开来仍然是很有价值的。但你需要保留第三个函数来同时做这两件事。这个「查询-修改」函数将调用各自独立的查询函数和修改函数，并被声明为 synchronized。如果查询函数和修改函数未被声明为 synchronized，那么你还应该将它们的可见范围限制在 package 级别或 private 级别。这样，你就可以拥有一个安全、同步的操作，它由两个较易理解的函数组成。这两个较低层函数也可以用于其他场合。

10.5 Parameterize Method（令函数携带参数）

若干函数做了类似的工作，但在函数本体中却包含了不同的值。

建立单一函数，以参数表达那些不同的值。



动机（Motivation）

你可能会发现这样的两个函数：它们做着类似的工作，但因少数几个值致使动作略有不同。这种情况下，你可以将这些各自分离的函数替换为一个统一函数，并通过参数来处理那些变化情况，用以简化问题。这样的修改可以去除重复的代码，并提高灵活性，因为你可以用这个参数处理其他（更多种）变化情况。

作法（Mechanics）

- 新建一个带有参数的函数，使它可以替换先前所有的重复性函数（repetitive methods）。
- 编译。
- 将「对旧函数的调用动作」替换为「对新函数的调用动作」。
- 编译，测试。
- 对所有旧函数重复上述步骤，每次替换后，修改并测试。

也许你会发现，你无法用这种办法处理整个函数，但可以处理函数中的一部分代码。这种情况下，你应该首先将这部分代码提炼到一个独立函数中，然后再对那个提炼所得的函数使用 *Parameterize Method*（283）。

范例（Example）

下面是一个最简单的例子：

```
class Employee {
    void tenPercentRaise () {
      salary *= 1.1;
    }
    void fivePercentRaise () {
      salary *= 1.05;
    }
```

这段代码可以替换如下：

```
void raise (double factor) {
    salary *= (1 + factor);
}
```

当然，这个例子实在太简单了，所有人都能做到。

下面是一个稍微复杂的例子：

```
protected Dollars baseCharge() {
    double result = Math.min(lastUsage(),100) * 0.03;
    if (lastUsage() > 100) {
      result += (Math.min (lastUsage(),200) - 100) * 0.05;
    };
    if (lastUsage() > 200) {
      result += (lastUsage() - 200) * 0.07;
    };
    return new Dollars (result);
}
```

上述代码可以替换如下：

```
protected Dollars baseCharge() {
    double result = usageInRange(0, 100) * 0.03;
    result += usageInRange (100,200) * 0.05;
    result += usageInRange (200, Integer.MAX_VALUE) * 0.07;
    return new Dollars (result);
}
protected int usageInRange(int start, int end) {
    if (lastUsage() > start)
      return Math.min(lastUsage(),end) - start;
    else return 0;
}
```

本项重构的伎俩在于：以「可将少量数值视为参数」为依据，找出带有重复性的代码。

10.6 Replace Parameter with Explicit Methods
以明确函数取代参数

你有一个函数，其内完全取决于参数值而采取不同反应。

针对该参数的每一个可能值，建立一个独立函数。

```
void setValue (String name, int value) {
    if (name.equals("height")){
        _height = value;
        return;
    }
    if (name.equals("width")){
        _width = value;
        return;
    }
    Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg) {
    _height = arg;
}
void setWidth (int arg) {
    _width = arg;
}
```

动机（Motivation）

Replace Parameter with Explicit Methods（285）恰恰相反于 *Parameterize Method*（283）。如果某个参数有离散取值，而函数内又以条件式检查这些参数值，并根据不同参数值做出不同的反应，那么就应该使用本项重构。调用者原本必须赋予参数适当的值，以决定该函数做出何种响应；现在，既然你提供了不同的函数给调用者使用，就可以避免出现条件式。此外你还可以获得「编译期代码检验」的好处，而且接口也更清楚。如果以参数值决定函数行为，那么函数用户不但需要观察该函数，而且还要判断参数值是否合法，而「合法的参数值」往往很少在文档中被清楚地提出。

就算不考虑「编译期检验」的好处，只是为了获得一个清晰的接口，也值得你执行本项重构。哪怕只是给一个内部的布尔(boolean)变量赋值，相较之下 `Switch.beOn()` 也比 `Switch.setState(true)` 要清楚得多。

但是，如果参数值不会对函数行为有太多影响，你就不应该使用 *Replace Parameter with Explicit Methods*（285）。如果情况真是这样，而你也只需要通过参数为一个值域赋值，那么直接使用设值函数（setter）就行了。如果你的确需要「条件判断」式的行为，可考虑使用 *Replace Conditional with Polymorphism*（255）。

作法（Mechanics）

- 针对参数的每一种可能值，新建一个明确函数。
- 修改条件式的每个分支，使其调用合适的新函数。
- 修改每个分支后，编译并测试。
- 修改原函数的每一个被调用点，改而调用上述的某个合适的新函数。
- 编译，测试。
- 所有调用端都修改完毕后，删除原（带有条件判断的）函数。

范例（Example）

下列代码中，我想根据不同的参数值，建立 Employee 之下不同的 subclass。以下代码往往是 *Replace Constructor with Factory Method*（304）的施行成果：

```
static final int ENGINEER = 0;
static final int SALESMAN = 1;
static final int MANAGER = 2;
static Employee create(int type) {
    switch (type) {
      case ENGINEER:
          return new Engineer();
      case SALESMAN:
          return new Salesman();
      case MANAGER:
          return new Manager();
      default:
          throw new IllegalArgumentException
                        ("Incorrect type code value");
    }
}
```

由于这是一个 ***factory method***，我不能实施 *Replace Conditional with Polymorphism*（255），因为使用该函数时我根本尚未创建出对象。我并不期待太多新的subclasses，所以一个明确的接口是合理的（译注：不甚理解作者文意）。首先，我要根据参数值建立相应的新函数：

```
static Employee createEngineer() {
    return new Engineer();
}
static Employee createSalesman() {
    return new Salesman();
}
static Employee createManager() {
    return new Manager();
}
```

然后把「switch 语句的各个分支」替换为「对新函数的调用」：

```
static Employee create(int type) {
    switch (type) {
      case ENGINEER:
          return Employee.createEngineer();
      case SALESMAN:
          return new Salesman();
      case MANAGER:
          return new Manager();
      default:
          throw new IllegalArgumentException
                                ("Incorrect type code value");
    }
}
```

每修改一个分支，都需要编译并测试，直到所有分支修改完毕为止：

```
static Employee create(int type) {
    switch (type) {
      case ENGINEER:
          return Employee.createEngineer();
      case SALESMAN:
          return Employee.createSalesman();
      case MANAGER:
          return Employee.createManager();
      default:
          throw new IllegalArgumentException
                                ("Incorrect type code value");
    }
}
```

接下来，我把注意力转移到旧函数的调用端。我把诸如下面这样的代码：

```
Employee kent = Employee.create(ENGINEER)
```

替换为：

```
Employee kent = Employee.createEngineer()
```

修改完 create() 函数的所有调用者之后，我就可以把 create() 函数删掉了。同时也可以把所有常量都删掉。

10.7 Preserve Whole Object（保持对象完整）

你从某个对象中取出若干值，将它们作为某一次函数调用时的参数。

改使用（传递）整个对象。

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

动机（Motivation）

有时候，你会将来自同一对象的若干项数据作为参数，传递给某个函数。这样做的问题在于：万一将来被调用函数需要新的数据项，你就必须查找并修改对此函数的所有调用。如果你把这些数据所属的整个对象传给函数，可以避免这种尴尬的处境，因为被调用函数可以向那个参数对象请求任何它想要的信息。

除了可以使参数列更稳固（不变动）之外，*Preserve Whole Object*（288）往往还能提高代码的可读性。过长的参数列很难使用，因为调用者和被调用者都必须记住这些参数的用途。此外，不使用完整对象也会造成重复代码，因为被调用函数无法利用完整对象中的函数来计算某些中间值。

「甘蔗不曾两头甜」！如果你传的是数值，被调用函数就只与这些数值有依存关系（dependency），与这些数值所属对象没有任何依存关系。但如果你传递的是整个对象，「参数对象」和「被调用函数所在对象」之间，就有了依存关系。如果这会使你的依存结构恶化，那么你就不该使用 *Preserve Whole Object*（288）。

我还听过另一种不使用 *Preserve Whole Object*（288）的理由：如果被调用函数只需要「参数对象」的其中一项数值，那么只传递那个数值会更好。我并不认同这种观点，因为传递一项数值和传递一个对象，至少在代码清晰度上是等价的（当然对于 *pass by value*（传值）参数来说，性能上可能有所差异）。更重要的考量应该放在「对象之间的依存关系」上。

如果被调用函数使用了「来自另一个对象的很多项数据」，这可能意味该函数实际上应该被定义在「那些数据所属的对象」中。所以，考虑 *Preserve Whole Object*（288）的同时，你也应该考虑 *Move Method*（142）。

运用本项重构之前，你可能还没有定义一个完整对象。那么你就应该先使用 *Introduce Parameter Object*（295）。

还有一种常见情况：调用者将自己的若干数据作为参数，传递给被调用函数。这种情况下，如果该对象有合适的取值函数（getter），你可以使用 `this` 取代这些参数值，并且无须操心对象依存问题。

作法（Mechanics）

- 对你的目标函数新添一个参数项，用以代表原数据所在的完整对象。
- 编译，测试。
- 判断哪些参数可被包含在新添的完整对象中。
- 选择上述参数之一，将「被调用函数」内对该参数的各个引用，替换为「对新添之参数对象的相应取值函数（getter）」的调用。
- 删除该项参数。
- 编译，测试。
- 针对所有「可从完整对象中获得」的参数，重复上述过程。
- 删除调用端中那些带有「被删除之参数」的所有代码。

  ⇨ 当然，如果调用端还在其他地方使用了这些参数，就不要删除它们。

- 编译，测试。

范例（Example）

以下范例，我以一个 Room 对象表示「房间」，它负责记录房间一天中的最高温度和最低温度。然后这个对象需要将「实际温度范围」与预先规定的「温度控制计划」相比较，告诉客户当天温度是否符合计划要求：

```
class Room...
    boolean withinPlan(HeatingPlan plan) {
      int low = daysTempRange().getLow();
      int high = daysTempRange().getHigh();
      return plan.withinRange(low, high);
    }
class HeatingPlan...
    boolean withinRange (int low, int high) {
      return (low >= _range.getLow() &&
              high <= _range.getHigh());
    }
    private TempRange _range;
```

其实我不必将 TempRange 对象的信息拆开来单独传递，只需将整个对象传递给 withinPlan()函数即可。在这个简单的例子中，我可以一次性完成修改。如果相关的参数更多些，我也可以进行小步重构。首先，我为参数列添加新的参数项，用以传递完整的 TempRange 对象：

```
class HeatingPlan...
    boolean withinRange (TempRange roomRange, int low, int high)
    {
      return (low >= _range.getLow() &&
              high <= _range.getHigh());
    }
class Room...
    boolean withinPlan(HeatingPlan plan) {
      int low = daysTempRange().getLow();
      int high = daysTempRange().getHigh();
      return plan.withinRange(daysTempRange(), low, high);
    }
```

然后，我以 TempRange 对象提供的函数来替换 low 参数：

```
class HeatingPlan...
    boolean withinRange (TempRange roomRange, int high) {
      return (roomRange.getLow() >= _range.getLow() &&
              high <= _range.getHigh());
    }
class Room...
    boolean withinPlan(HeatingPlan plan) {
      int low = daysTempRange().getLow();
```

```
        int high = daysTempRange().getHigh();
        return plan.withinRange(daysTempRange(), high);
    }
```

重复上述步骤，直到把所有待处理参数项都去除为止：

```
class HeatingPlan...
    boolean withinRange (TempRange roomRange) {
        return (roomRange.getLow() >= _range.getLow() &&
                roomRange.getHigh() <= _range.getHigh());
    }
class Room...
    boolean withinPlan(HeatingPlan plan) {
        int low = daysTempRange().getLow();
        int high = daysTempRange().getHigh();
        return plan.withinRange(daysTempRange());
    }
```

现在，我不再需要 low 和 high 这两个临时变量了：

```
class Room...
    boolean withinPlan(HeatingPlan plan) {
        int low = daysTempRange().getLow();
        int high = daysTempRange().getHigh();
        return plan.withinRange(daysTempRange());
    }
```

使用完整对象后不久，你就会发现，可以将某些函数移到 TempRange 对象中，使它更容易被使用，例如：

```
class HeatingPlan...
    boolean withinRange (TempRange roomRange) {
        return (_range.includes(roomRange));
    }
class TempRange...
    boolean includes (TempRange arg) {
        return (arg.getLow() >= this.getLow() &&
                arg.getHigh() <= this.getHigh());
    }
```

10.8 Replace Parameter with Methods（以函数取代参数）

对象调用某个函数，并将所得结果作为参数，传递给另一个函数。而接受该参数的函数也可以（也有能力）调用前一个函数。

让参数接受者去除该项参数，并直接调用前一个函数。

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

动机（Motivation）

如果函数可以通过其他途径（而非参数列）获得参数值，那么它就不应该通过参数取得该值。过长的参数列会增加程序阅读者的理解难度，因此我们应该尽可能缩短参数列的长度。

缩减参数列的办法之一就是，看看「参数接受端（receiver）」是否可以通过「与调用端相同的计算」来取得参数携带值。如果调用端通过「其所属对象内部的另一个函数」来计算参数，并在计算过程中「未曾引用调用端的其他参数」（译注：亦就是说没有太多与外界的相依关系），那么你就应该可以将这个计算过程转移到被调用端内，从而去除该项参数。如果你所调用的函数隶属另一对象，而该对象拥有一个 reference 指向调用端所属对象，前面所说的这些也同样适用。

但是，如果「参数计算过程」倚赖调用端的某个参数，那么你就无法去掉被调用端的那个参数，因为每一次调用动作中，该参数值都可能不同（当然，如果你能够运用 *Replace Parameter with Explicit Methods*（285）将该参数替换为一个函数，又另当别论）。另外，如果参数接受端（receiver）并没有一个 reference 指向参数发送端（sender），而你也不想加上这样一个 reference，那么也无法去除参数。

有时候，参数的存在是为了将来的弹性。这种情况下我仍然会把这种多余参数拿掉。是的，你应该只在必要关头才添加参数，预先添加的参数很可能并不是你所需要的。不过，对于这条规则，也有一个例外：如果修改接口会对整个程序造成非常痛苦的结果（例如需要很长时间来重建程序，或需要修改大量代码），那么

可以考虑保留前人预先加入的参数。如果真是这样，你应该首先判断修改接口究竟会造成多严重的后果，然后考虑是否「降低系统各部位之间的依存程度」以减少「修改接口所造成的影响」。稳定的接口确实很好，但是被冻结在一个不良接口上，也是有问题的。

作法（Mechanics）

- 如果有必要，将参数的计算过程提炼到一个独立函数中。
- 将函数本体内「对该参数的引用」替换为「对新建函数的调用」。
- 每次替换后，修改并测试。
- 全部替换完成后，使用 *Remove Parameter*（277）将该参数去掉。

范例（Example）

以下代码用于计算定单折扣价格。虽然这么低的折扣不大可能出现在现实生活中，不过作为一个范例，我们暂不考虑这一点：

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    int discountLevel;
    if (_quantity > 100) discountLevel = 2;
    else discountLevel = 1;
    double finalPrice =
      discountedPrice (basePrice, discountLevel);
    return finalPrice;
}
private double discountedPrice (int basePrice, int discountLevel) {
    if (discountLevel == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

首先，我把计算折扣等级（discountLevel）的代码提炼成为一个独立的getDiscountLevel()函数：

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    int discountLevel = getDiscountLevel();
    double finalPrice =
      discountedPrice (basePrice, discountLevel);
    return finalPrice;
}
private int getDiscountLevel() {
    if (_quantity > 100) return 2;
    else return 1;
}
```

然后把discountedPrice()函数中对discountLevel参数的所有引用点，替换为对getDiscountLevel()函数的调用：

```
private double discountedPrice (int basePrice, int discountLevel) {
    if (getDiscountLevel() == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

此时我就可以使用 *Remove Parameter*（277）去掉discountLevel参数了：

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    int discountLevel = getDiscountLevel();
    double finalPrice = discountedPrice (basePrice);
    return finalPrice;
}
private double discountedPrice (int basePrice) {
    if (getDiscountLevel() == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

接下来可以将discountLevel变量去除掉：

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double finalPrice = discountedPrice (basePrice);
    return finalPrice;
}
```

现在，可以去掉其他非必要的参数和相应的临时变量。最后获得以下代码：

```
public double getPrice() {
    return discountedPrice ();
}
private double discountedPrice () {
    if (getDiscountLevel() == 2) return getBasePrice() * 0.1;
    else return getBasePrice() * 0.05;
}
private double getBasePrice() {
    return _quantity * _itemPrice;
}
```

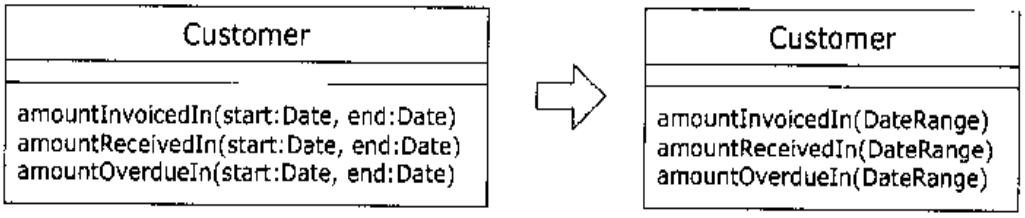
最后我还可以针对discountedPrice()函数使用 *Inline Method*（117）：

```
private double getPrice () {
    if (getDiscountLevel() == 2) return getBasePrice() * 0.1;
    else return getBasePrice() * 0.05;
}
```

10.9 Introduce Parameter Object（引入参数对象）

某些参数总是很自然地同时出现。

以一个对象取代这些参数。



动机（Motivation）

你常会看到特定的一组参数总是一起被传递。可能有好几个函数都使用这一组参数，这些函数可能隶属同一个class，也可能隶属不同的classes。这样一组参数就是所谓的 Data Clump（数据泥团），我们可以运用一个对象包装所有这些数据，再以该对象取代它们。哪怕只是为了把这些数据组织在一起，这样做也是值得的。本项重构的价值在于「缩短了参数列的长度」，而你知道，过长的参数列总是难以理解的。此外，新对象所定义的访问函数（accessors）还可以使代码更具一致性，这又进一步降低了代码的理解难度和修改难度。

本项重构还可以带给你更多好处。当你把这些参数组织到一起之后，往往很快可以发现一些「可被移至新建class」的行为。通常，原本使用那些参数的函数对那些参数会有一些共通措施，如果将这些共通行为移到新对象中，你可以减少很多重复代码。

作法（Mechanics）

- 新建一个class，用以表现你想替换的一组参数。将这个class设为不可变的（不可被修改的，immutable）。
- 编译。

- 针对使用该组参数的所有函数，实施 *Add Parameter*（275），以上述新建 class 之实体对象作为新添参数，并将此一参数值设为 null。
  - ⇨ 如果你所修改的函数被其他很多函数调用，那么你可以保留修改前的旧函数，并令它调用修改后的新函数。你可以先对旧函数进行重构，然后逐一令调用端转而调用新函数，最后再将旧函数删除。
- 对于 Data Clump（数据泥团）中的每一项（在此均为参数），从函数签名式（signature）中移除之，并修改调用端和函数本体，令它们都改而通过「新建的参数对象」取得该值。
- 每去除一个参数，编译并测试。
- 将原先的参数全部去除之后，观察有无适当函数可以运用 *Move Method*（142）搬移到参数对象之中。
  - ⇨ 被搬移的可能是整个函数，也可能是函数中的一个段落。如果是后者，首先使用 *Extract Method*（110）将该段落提炼为一个独立函数，再搬移这一新建函数。

范例（Example）

下面是一个「帐日和帐项」（account and entries）范例。表示「帐项」的 Entry 实际上只是个简单的数据容器：

```
class Entry...
    Entry (double value, Date chargeDate) {
      _value = value;
      _chargeDate = chargeDate;
    }
    Date getDate(){
      return _chargeDate;
    }
    double getValue(){
      return _value;
    }
    private Date _chargeDate;
    private double _value;
```

我关注的焦点是用以表示「帐日」的 Account，它保存了一组 Entry 对象，并有一个函数用来计算两日期间的帐项总量：

```
class Account...
    double getFlowBetween (Date start, Date end) {
      double result = 0;
      Enumeration e = _entries.elements();
      while (e.hasMoreElements()) {
          Entry each = (Entry) e.nextElement();
          if (each.getDate().equals(start) ||
```

```
                each.getDate().equals(end) ||
                (each.getDate().after(start) &&
                 each.getDate().before(end)))
            {
                result += each.getValue();
            }
        }
        return result;
    }
    private Vector _entries = new Vector();
client code...
    double flow = anAccount.getFlowBetween(startDate, endDate);
```

我已经记不清有多少次看见代码以「一对值」表示「一个范围」，例如表示日期范围的 start 和 end、表示数值范围的 upper 和 lower 等等。我知道为什么会发生这种情况，毕竟我自己也经常这样做。不过，自从我得知 ***Range*** 模式 [Fowler, AP] 之后，我就尽量以「范围对象」取而代之。我的第一个步骤是声明一个简单的数据容器，用以表示范围：

```
class DateRange {
    DateRange (Date start, Date end) {
        _start = start;
        _end = end;
    }
    Date getStart() {
        return _start;
    }
    Date getEnd() {
        return _end;
    }
    private final Date _start;
    private final Date _end;
}
```

我把 **DateRange** class 设为不可变，也就是说，其中所有值域都是 final，只能由构造函数来赋值，因此没有任何函数可以修改其中任何值域值。这是一个明智的决定，因为这样可以避免别名（aliasing）带来的困扰。Java 的函数参数都是 ***pass by value***（传值），不可变类（immutable class）正是能够模仿 Java 参数的工作方式，因此这种作法对于本项重构是最合适的。

接下来我把 **DateRange** 对象加到 getFlowBetween() 函数的参数列中：

```
class Account...
    double getFlowBetween (Date start, Date end, DateRange range)
    {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
```

```
            if (each.getDate().equals(start) ||
                each.getDate().equals(end) ||
                (each.getDate().after(start) &&
                 each.getDate().before(end)))
            {
                result += each.getValue();
            }
        }
        return result;
    }
client code...
    double flow =
        anAccount.getFlowBetween(startDate, endDate, null);
```

至此，我只需编译一下就行了，因为我尚未修改程序的任何行为。

下一个步骤是去除旧参数之一，以新建对象取而代之。首先我删除 start 参数，并修改 getFlowBetween() 函数及其调用者，让它们转而使用新对象：

```
class Account...
    double getFlowBetween (Date end, DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            if (each.getDate().equals(range.getStart()) ||
                each.getDate().equals(end) ||
                (each.getDate().after(range.getStart()) &&
                 each.getDate().before(end)))
            {
                result += each.getValue();
            }
        }
        return result;
    }
client code...
    double flow = anAccount.getFlowBetween
                        (endDate, new DateRange (startDate, null));
```

然后我将 end 参数也移除：

```
class Account...
    double getFlowBetween (DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            if (each.getDate().equals(range.getStart()) ||
                each.getDate().equals(range.getEnd()) ||
                (each.getDate().after(range.getStart()) &&
```

```
                each.getDate().before(range.getEnd())))
            {
                result -= each.getValue();
            }
        }
        return result;
    }

client code...
    double flow = anAccount.getFlowBetween
                            (new DateRange (startDate, endDate));
```

现在，我已经引入了「参数对象」。我还可以将适当的行为从其他函数移到这个新建对象中，进一步从本项重构获得更大利益。这里，我选定条件式中的代码，实施 *Extract Method*（110）和 *Move Method*（142），最后得到如下代码：

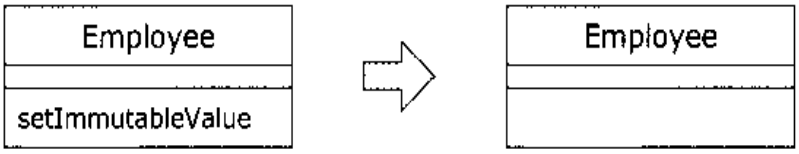
```
class Account...
    double getFlowBetween (DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            if (range.includes(each.getDate())) {
                result -= each.getValue();
            }
        }
        return result;
    }
class DateRange...
    boolean includes (Date arg) {
        return (arg.equals(_start) ||
                arg.equals(_end) ||
                (arg.after(_start) && arg.before(_end)));
    }
```

如此单纯的提炼和搬移动作，我通常一步完成。如果在这个过程中出错，我可以回到重构前的状态，然后分成两个较小步骤重新进行。

10.10 Remove Setting Method（移除设值函数）

你的class中的某个值域，应该在对象初创时被设值，然后就不再改变。

去掉该值域的所有设值函数（setter）。



动机（Motivation）

如果你为某个值域提供了设值函数（setter），这就暗示这个值域值可以被改变。如果你不希望在对象初创之后此值域还有机会被改变，那就不要为它提供设值函数（同时并将该值域设为final）。这样你的意图会更加清晰，并且往往可以排除其值被修改的可能性——这种可能性往往是非常大的。

如果你保留了间接访问变量的方法，就可能经常有程序员自目使用它们 [Beck]。这些人甚至会在构造函数中使用设值函数！我猜想他们或许是为了代码的一致性，但却忽视了设值函数往后可能带来的混淆。

作法（Mechanics）

- 检查设值函数（setter）被使用的情况，看它是否只被构造函数调用，或者被构造函数所调用的另一个函数调用。
- 修改构造函数，使其直接访问设值函数所针对的那个变量。
  - ⇨ 如果某个subclass通过设值函数给superclass的某个private值域设了值，那么你就不能这样修改。这种情况下你应该试着在superclass中提供一个protected函数（最好是构造函数）来给这些值域设值。不论你怎么做，都不要给superclass中的函数起一个与设值函数混淆的名字。
- 编译，测试。
- 移除这个设值函数，将它所针对的值域设为final。
- 编译，测试。

范例（Example）

译注：本书英文版网站上的勘误网页（www.refactoring.com/errata.html）显示，本页程序有些问题，惟因前后颇有牵连，故勘误表上并未明确条列代码之修改。请读者自行上网查阅理解。

下面是一个简单例子：

```
class Account {
    private String _id;
    Account (String id) {
      setId(id);
    }
    void setId (String arg) {
      _id = arg;
    }
```

以上代码可修改为：

```
class Account {
    private final String _id;
    Account (String id) {
      _id = id;
    }
```

问题可能以数种不同的形式出现。首先，你可能会在设值函数中对引数做运算：

```
class Account {
    private String _id;
    Account (String id) {
      setId(id);
    }
    void setId (String arg) {
      _id = "77" + arg;
    }
```

如果对引数的修改很简单（就像上面这样）而且又只有一个构造函数，我可以直接在构造函数中做相同的修改。如果修改很复杂，或者有一个以上的函数调用它，我就需要提供一个独立函数。我需要为新函数起个好名字，清楚表达该函数的用途：

```
class Account {
    private final String _id;
```

```
Account (String id) {
  initializeId(id);
}
void initializeId (String arg) {
  _id = "ZZ" + arg;
}
```

如果subclass需要对superclass的private变量赋初值，情况就比较麻烦一些：

```
class InterestAccount extends Account...
    private double _interestRate;

    InterestAccount (String id, double rate) {
      setId(id);
      _interestRate = rate;
    }
```

问题是我无法在InterestAccount()中直接访问id变量。最好的解决办法就是使用superclass构造函数：

```
class InterestAccount...
    InterestAccount (String id, double rate) {
      super(id);
      _interestRate = rate;
    }
```

如果不能那样做，那么使用一个命名良好的函数就是最好的选择：

```
class InterestAccount...
    InterestAccount (String id, double rate) {
      initializeId(id);
      _interestRate = rate;
    }
```

另一种需要考虑的情况就是对一个群集（collections）设值：

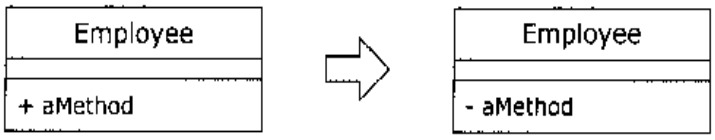
```
class Person {
    Vector getCourses() {
    return _courses;
}
void setCourses(Vector arg) {
    _courses = arg;
}
private Vector _courses;
```

在这里，我希望将设值函数替换为"add"操作加上"remove"操作。我已经在*Encapsulate Collection*（208）中谈到了这一点。

10.11 Hide Method（隐藏某个函数）

有一个函数，从来没有被其他任何 class 用到。

将这个函数修改为 private。



动机（Motivation）

重构往往促使你修改「函数的可见度」（visibility of methods）。提高函数可见度的情况很容易想像：另一个 class 需要用到某个函数，因此你必须提高该函数的可见度。但是要指出一个函数的可见度是否过高，就稍微困难一些。理想状况下你可以使用工具检查所有函数，指出可被隐藏起来的函数。即使没有这样的工具，你也应该时常进行这样的检查。

一种特别常见的情况是：当你面对一个过于丰富、提供了过多行为的接口时，就值得将非必要的取值函数（getter）和设值函数（setter）隐藏起来。尤其当你面对的是一个「只不过做了点简单封装」的数据容器（data holder）时，情况更是如此。随着愈来愈多行为被放入这个 class 之中，你会发现许多取值/设值函数不再需要为 public，因此可以把它们隐藏起来。如果你把取值/设值函数设为 private，并在他处直接访问变量，那就可以放心移除取值/设值函数了。

作法（Mechanics）

- □ 经常检查有没有可能降低某个函数的可见度（使它更私有化）。
  - ⇨ 使用 lint-style 工具，尽可能频繁地检查。当你在另一个 class 中移除对某个函数的调用时，也应该进行检查。
  - ⇨ 特别对设值函数（setter）进行上述的检查。
- □ 尽可能降低所有函数的可见度。
- □ 每完成一组函数的隐藏之后，编译并测试。
  - ⇨ 如果有不适当的隐藏，编译器很自然会检验出来，因此不必每次修改后都进行编译。如有任何错误出现，很容易被发现。

10.12 Replace Constructor with Factory Method
以「工厂函数」取代「构造函数」

你希望在创建对象时不仅仅是对它做简单的建构动作（simple construction）。

将 constructor（**构造函数**）**替换为** ***factory method***（**工厂函数**）。

```
Employee (int type) {
    _type = type;
}
```

```
static Employee create(int type) {
    return new Employee(type);
}
```

动机（Motivation）

使用 *Replace Constructor with Factory Method*（304）的最显而易见的动机就是在subclassing过程中以 ***factory method*** 取代type code。你可能常常需要根据type code创建相应的对象，现在，创建名单中还得加上subclasses，那些subclasses也是根据type code来创建。然而由于构造函数只能返回「被索求之对象」，因此你需要将构造函数替换为 ***Factory Method*** [Gang of Four]。

此外，如果构造函数的功能不能满足你的需要，也可以使用 ***factory method*** 来代替它。***Factory method*** 也是 *Change Value to Reference*（179）的基础。你也可以令你的 ***factory method*** 根据参数的个数和型别，选择不同的创建行为。

作法（Mechanics）

- 新建一个 ***factory method***，让它调用现有的构造函数。
- 将「对构造函数的调用」替换为「对 ***factory method*** 的调用」。

- 每次替换后，编译并测试。
- 将构造函数声明为 private。
- 编译。

范例：根据整数（实际为 type code）创建对象

又是那个单调乏味的例子：员工薪资系统。我以 Employee 表示「员工」：

```
class Employee {

    private int _type;
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;

    Employee (int type) {
       _type = type;
    }
```

我希望为 Employee 提供不同的 subclasses，并分别给予它们相应的 type code。因此，我需要建立一个 ***factory method***：

```
static Employee create(int type) {
    return new Employee(type);
}
```

然后，我要修改构造函数的所有调用点，让它们改用上述新建的 ***factory method***，并将构造函数声明为 private：

```
client code...
    Employee eng = Employee.create(Employee.ENGINEER);

class Employee...
    private Employee (int type) {
       _type = type;
   }
```

范例：根据字符串（String）创建 subclass 对象

迄今为止，我还没有获得什么实质收获。目前的好处在于：我把「对象创建之调用动作的接收者」和「被创建之对象所属的 class」分开了。如果我随后使用 *Replace Type Code with Subclasses*（223）把 type code 转换为 Employee 的 subclass，我就可以运用 ***factory method***，将这些 subclass 对用户隐藏起来：

```
static Employee create(int type) {
    switch (type) {
      case ENGINEER:
          return new Engineer();
      case SALESMAN:
          return new Salesman();
      case MANAGER:
          return new Manager();
      default:
          throw new IllegalArgumentException
                          ("Incorrect type code value");
    }
}
```

可惜的是，这里面有一个 switch 语句。如果我添加一个新的 subclass，就必须记得更新这里的 switch 语句，而我又偏偏很健忘。

绕过这个 switch 语句的一个好办法是使用 Class.forName()。第一件要做的事是修改参数型别，这从根本上说是 *Raname Method*（273）的一种变体。首先我得建立一个函数，让它接收一个字符串引数（string argument）：

```
static Employee create (String name) {
    try {
      return (Employee) Class.forName(name).newInstance();
    } catch (Exception e) {
      throw new IllegalArgumentException
                      ("Unable to instantiate" + name);
    }
}
```

然后让稍早那个「create()函数 int 版」调用新建的「create()函数 String 版」：

```
class Employee {
    static Employee create(int type) {
      switch (type) {
          case ENGINEER:
              return create("Engineer");
          case SALESMAN:
              return create("Salesman");
          case MANAGER:
              return create("Manager");
          default:
              throw new IllegalArgumentException
                              ("Incorrect type code value");
      }
    }
```

然后，我得修改 create()函数的调用者，将下列这样的语句：

```
Employee.create(ENGINEER)
```

修改为：

```
Employee.create("Engineer")
```

完成之后，我就可以将「create()函数，int版本」移除了。

现在，当我需要添加新的 Employee subclasses，就不再需要更新 create()函数了。但我却因此失去了编译期检验，使得一个小小的拼写错误就可能造成运行期错误。如果有必要防止运行期错误，我会使用明确函数来创建对象（见本页下）。但这样一来，每添加一个新的 subclass，我就必须添加一个新函数。这就是为了型别安全而牺牲掉的灵活性。还好，即使我做了错误选择，也可以使用 *Parameterize Method*（283）或 *Replace Parameter with Explicit Methods*（285）撤销决定。

另一个「必须谨慎使用 Class.forName()」的原因是：它向用户暴露了 subclass 名称。不过这并不是太糟糕，因为你可以使用其他字符串，并在 ***factory method*** 中执行其他行为。这也是「不使用 *Inline Method*（117）去除 ***factory method***」的一个好理由。

范例：以明确函数（Explicit Methods）创建 subclass

我可以通过另一条途径来隐藏 subclass——使用明确函数。如果你只有少数几个 subclasses，而且它们都不再变化，这条途径是很有用的。我可能有个抽象的 Person class，它有两个 subclass：Male 和 Female。首先我在 superclass 中为每个 subclass 定义一个 ***factory method***：

```
class Person...
    static Person createMale(){
      return new Male();
    }
    static Person createFemale() {
      return new Female();
    }
```

然后我可以把下面的调用：

```
Person kent = new Male();
```

替换成：

```
Person kent = Person.createMale();
```

但是这就使得 superclass 必须知晓 subclass。如果想避免这种情况，你需要一个更为复杂的设计，例如 ***Product Trader*** 模式 [Bäumer and Riehle]。绝大多数情况下你并不需要如此复杂的设计，上面介绍的作法已经绰绰有余。

10.13 Encapsulate Downcast（封装「向下转型」动作）

某个函数返回的对象，需要由函数调用者执行「向下转型」（downcast）动作。

将向下转型（downcast）动作移到函数中。

```
Object lastReading() {
    return readings.lastElement();
}
```

```
Reading lastReading() {
    return (Reading) readings.lastElement();
}
```

动机（Motivation）

在强型别（strongly typed）OO 语言中，向下转型是最烦人的事情之一。之所以很烦人，是因为从感觉上来说它完全没有必要：你竟然越俎代庖地告诉编译器某些应该由编译器自己计算出来的东西。但是，由于「计算对象型别」的动作往往比较麻烦，你还是常常需要亲自告诉编译器「对象的确切型别」，向下转型在 Java 特别盛行，因为 Java 没有 template（模板）机制，因此如果你想从群集（collection）之中取出一个对象，就必须进行向下转型。

向下转型也许是一种无法避免的罪恶，但你仍然应该尽可能少做。如果你的某个函数返回一个值，并且你知道「你所返回的对象」其型别比函数签名式（signature）所昭告的更特化（specialized；译注：意指返回的是原本声明之 return type 的 subtype），你便是在函数用户身上强加了非必要的工作。这种情况下你不应该要求用户承担向下转型的责任，应该尽量为他们提供准确的型别。

以上所说的情况，常会在「返回迭代器（iterator）或群集（collection）」的函数身上发生。此时你就应该观察人们拿这个迭代器干什么用，然后针对性地提供专用函数。

作法（Mechanics）

- 找出「必须对函数调用结果进行向下转型」的地方。
  - ⇨ 这种情况通常出现在「返回一个群集（collection）或迭代器（iterator）」的函数中。
- 将向下转型动作搬移到该函数中。
  - ⇨ 针对返回群集（collection）的函数，使用 *Encapsulate Collection*（208）。

范例（Example）

下面的例子中，我以 Reading 表示「书籍」。我还拥有一个名为 lastReading() 的函数，它从一个用以「保存 Reading 对象」的 vector 中返回其最后一个元素：

```
Object lastReading() {
    return readings.lastElement();
}
```

我应该将这个函数变成：

```
Reading lastReading() {
    return (Reading) readings.lastElement();
}
```

当我拥有一个群集时，上述那么做就很有意义。如果「保存 Reading 对象」的群集被放在 Site class 中，并且我看到了如下的代码（客户端）：

```
Reading lastReading = (Reading) theSite.readings().lastElement()
```

我就可以不再把「向下转型」工作推给用户，并得以向用户隐藏群集：

```
Reading lastReading = theSite.lastReading();

class Site...
   Reading lastReading() {
      return (Reading) readings().lastElement();
   }
```

如果你修改函数，将其「返回型别」（return type）改为原返回型别的 subclass，那就是改变了函数签名式（signature），但并不会破坏客户端代码，因为编译器知道它总是可以将一个 subclass 自动向上转型为 superclass。当然啦你必须确保这个 subclass 不会破坏 superclass 带来的任何契约（contract）。（译注：在 OO 设计中，继承关系代表 **is-a** 关系，因此 subclass **is-a** superclass，因此正确设计之 subclass 决不会破坏 superclass 带来的任何契约。）

10.14 Replace Error Code with Exception
以异常取代错误码

某个函数返回一个特定的代码（special code），用以表示某种错误情况。

改用异常（exception）。

```
int withdraw(int amount) {
    if (amount > _balance)
        return -1;
    else {
        _balance -= amount;
        return 0;
    }
}
```

```
void withdraw(int amount) throws BalanceException {
    if (amount > _balance) throw new BalanceException();
    _balance -= amount;
}
```

动机（Motivation）

和生活一样，计算器偶尔也会出错。一旦事情出错，你就需要有些对策。最简单的情况下，你可以停止程序运行，返回一个错误码。这就好像因为错过一班飞机而自杀一样（如果真那么做，哪怕我是只猫，我的九条命也早赔光了）。尽管我的油腔滑调企图带来一点幽默，但这种「软件自杀」选择的确是有好处的。如果程序崩溃代价很小，用户又足够宽容，那么就放心终止程序的运行好了。但如果你的程序比较重要，就需要以比较认真的方式来处理。

问题在于：程序中发现错误的地方，并不一定知道如何处理错误。当一段副程序（routine）发现错误时，它需要让它的调用者知道这个错误，而调用者也可能将这个错误继续沿着调用链（call chain）传递上去。许多程序都使用特殊输出来表示错误，Unix系统和C-based系统的传统方式就是「以返回值表示副程序的成功或失败」。

Java有一种更好的错误处理方式：异常（exceptions）。这种方式之所以更好，因为它清楚地将「普通程序」和「错误处理」分开了，这使得程序更容易理解——我希望你如今已经坚信：代码的可理解性应该是我们虔诚追求的目标。

作法（Mechanics）

- 决定待抛异常应该是 *checked* 还是 *unchecked*。
  - ⇨ 如果调用者有责任在调用前检查必要状态，就抛出 *unchecked* 异常。
  - ⇨ 如果想抛出 *checked* 异常，你可以新建一个 exception class，也可以使用现有的 exception classes。
- 找到该函数的所有调用者，对它们进行相应调整，让它们使用异常。
  - ⇨ 如果函数抛出 *unchecked* 异常，那么就调整调用者，使其在调用函数前做适当检查。每次修改后，编译并测试。
  - ⇨ 如果函数抛出 *checked* 异常，那么就调整调用者，使其在 try 区段中调用该函数。
- 修改该函数的签名式（signature），令它反映出新用法。

如果函数有许多调用者，上述修改过程可能跨度太大。你可以将它分成下列数个步骤：

- 决定待抛异常应该是 *checked* 还是 *unchecked*。
- 新建一个函数，使用异常来表示错误状况，将旧函数的代码拷贝到新函数中，并做适当调整。
- 修改旧函数的函数本体，让它调用上述新建函数。
- 编译，测试。
- 逐一修改旧函数的调用者，令其调用新函数。每次修改后，编译并测试。
- 移除旧函数。

范例（Example）

现实生活中你可以透支你的账户余额，计算器教科书却总是假设你不能这样做。这不是很奇怪吗？不过下面的例子仍然假设你不能这样做：

```
class Account...
    int withdraw(int amount) {
      if (amount > _balance)
          return -1;
      else {
          _balance -= amount;
          return 0;
```

```
        }
    }
    private int _balance;
```

为了让这段代码使用异常，我首先需要决定使用 *checked* 异常还是 *unchecked* 异常。决策关键在于：调用者是否有责任在取款之前检查存款余额，或者是否应该由 withdraw()函数负责检查。如果「检查余额」是调用者的责任，那么「取款金额大于存款余额」就是一个编程错误。由于这是一个编程错误（也就是一只「臭虫」），所以我应该使用 *unchecked* 异常。另一方面，如果「检查余额」是 withdraw()函数的责任，我就必须在函数接口中声明它可能抛出这个异常（译注：这是一个 *checked* 异常），那么也就提醒了调用者注意这个异常，并采取相应措施。

范例：*unchecked* 异常

首先考虑 *unchecked* 异常，使用这个东西就表示应该由调用者负责检查。首先我需要检查调用端的代码，它不应该使用 withdraw()函数的返回值，因为该返回值只用来指出程序员的错误。如果我看到下面这样的代码：

```
if (account.withdraw(amount) == -1)
    handleOverdrawn();
else doTheUsualThing();
```

我应该将它替换为这样的代码：

```
if (!account.canWithdraw(amount))
    handleOverdrawn();
else {
    account.withdraw(amount);
    doTheUsualThing();
}
```

每次修改后，编译并测试。

现在，我需要移除错误码，并在程序出错时抛出异常。由于行为（根据其文本定义得知）是异常的、罕见的，所以我应该用一个卫语句（guard clause）检查这种情况：

```
void withdraw(int amount) {
    if (amount > _balance)
      throw new IllegalArgumentException ("Amount too large");
    _balance -= amount;
}
```

由于这是程序员所犯的错误，所以我应该使用 assertion 更清楚地指出这一点：

```
class Account...
    void withdraw(int amount) {
      Assert.isTrue ("sufficient funds", amount <= _balance);
      _balance -= amount;
    }

class Assert...
    static void isTrue (String comment, boolean test) {
      if (! test) {
          throw new RuntimeException
                                ("Assertion failed: " + comment);
      }
    }
```

范例：*checked* 异常

checked 异常的处理方式略有不同。首先我要建立（或使用）一个合适的异常：

```
class BalanceException extends Exception {}
```

然后，调整调用端如下：

```
try {
    account.withdraw(amount);
    doTheUsualThing();
} catch (BalanceException e) {
    handleOverdrawn();
}
```

接下来我要修改 withdraw() 函数，让它以异常表示错误状况：

```
void withdraw(int amount) throws BalanceException {
    if (amount > _balance) throw new BalanceException();
    _balance -= amount;
}
```

这个过程的麻烦在于：我必须一次性修改所有调用者和被它们调用的函数，否则编译器会报错。如果调用者很多，这个步骤就实在太大了，其中没有编译和测试的保障。

这种情况下，我可以借助一个临时中间函数。我仍然从先前相同的情况出发：

```
if (account.withdraw(amount) == -1)
    handleOverdrawn();
else doTheUsualThing();
class Account ...
    int withdraw(int amount) {
      if (amount > _balance)
          return -1;
      else {
          _balance -= amount;
          return 0;
      }
    }
```

首先，产生一个newWithdraw()函数，让它抛出异常：

```
void newWithdraw(int amount) throws BalanceException {
    if (amount > _balance) throw new BalanceException();
      _balance -= amount;
}
```

然后，调整现有的withdraw()函数，让它调用newWithdraw()：

```
int withdraw(int amount) {
    try {
      newWithdraw(amount);
      return 0;
    } catch (BalanceException e) {
      return -1;
    }
}
```

完成以后，编译并测试。现在我可以逐一将「对旧函数的调用」替换为「对新函数的调用」：

```
try {
    account.newWithdraw(amount);
    doTheUsualThing();
} catch (BalanceException e) {
    handleOverdrawn();
}
```

由于新旧两函数都存在，所以每次修改后我都可以编译、测试。所有调用者都被我修改完毕后，旧函数便可移除，并使用 *Rename Method*（273）修改新函数名称，使它与旧函数相同。

10.15 Replace Exception with Test
以测试取代异常

面对一个「调用者可预先加以检查」的条件，你抛出了一个异常。

修改调用者，使它在调用函数之前先做检查。

```
double getValueForPeriod (int periodNumber) {
    try {
        return _values[periodNumber];
    } catch (ArrayIndexOutOfBoundsException e) {
        return 0;
    }
}
```

```
double getValueForPeriod (int periodNumber) {
    if (periodNumber >= _values.length) return 0;
    return _values[periodNumber];
}
```

动机（Motivation）

异常（exception）的出现是程序语言的一大进步。运用 *Replace Error Code with Exception*（310），异常便可协助我们避免很多复杂的错误处理逻辑。但是，就像许多好东西一样，异常也会被滥用，从而变得不再让人愉快（就连味道极好的 Aventinus 啤酒，喝得太多也会让我厌烦 [Jackson]）。「异常」只应该被用于异常的、罕见的行为，也就是那些「产生意料外的错误」的行为，而不应该成为「条件检查」的替代品。如果你可以合理期望调用者在调用函数之前先检查某个条件，那么你就应该提供一个测试，而调用者应该使用它。

作法（Mechanics）

- 在函数调用点之前，放置一个测试句，将函数内的 `catch` 区段中的代码拷贝到测试句的适当 `if` 分支中。
- 在 `catch` 区段起始处加入一个 assertion，确保 `catch` 区段绝对不会被执行。
- 编译，测试。

- 移除所有 catch 区段，然后将 try 区段内的代码拷贝到 try 之外，然后移除 try 区段。
- 编译，测试。

范例（Example）

下面的例子中，我以一个 ResourcePool 对象管理「创建代价高昂、可复用」的资源（例如数据库连接，database connection）。这个对象带有两个「池」（pools），一个用以保存可用资源，一个用以保存已分配资源。当用户索求一份资源时，ResourcePool 对象从「可用资源池」中取出一份资源交出，并将这份资源转移到「已分配资源池」。当用户释放一份资源时，ResourcePool 对象就将该资源从「已分配资源池」放回「可用资源池」。如果「可用资源池」不能满足用户的索求，ResourcePool 对象就创建一份新资源。

资源供应函数可能如下所示：

```
class ResourcePool
    Resource getResource() {
      Resource result;
      try {
          result = (Resource) _available.pop();
          _allocated.push(result);
          return result;
      } catch (EmptyStackException e) {
          result = new Resource();
          _allocated.push(result);
          return result;
      }
    }
    Stack _available;
    Stack _allocated;
```

在这里，「可用资源用尽」并不是一种意料外的事件，因此我不该使用异常（exceptions）表示这种情况。

为了去掉这里的异常，我首先必须添加一个适当的提前测试，并在其中处理「可用资源池为空」的情况：

```
Resource getResource() {
    Resource result;
    if (_available.isEmpty()) {
      result = new Resource();
      _allocated.push(result);
      return result;
    }
```

```
        else {
          try {
              result = (Resource) _available.pop();
              _allocated.push(result);
              return result;
          } catch (EmptyStackException e) {
              result = new Resource();
              _allocated.push(result);
              return result;
          }
        }
    }
```

现在getResource()应该绝对不会抛出异常了。我可以添加 assertion 保证这一点：

```
Resource getResource() {
    Resource result;
    if (_available.isEmpty()) {
      result = new Resource();
      _allocated.push(result);
      return result;
    }
    else {
      try {
          result = (Resource) _available.pop();
          _allocated.push(result);
          return result;
      } catch (EmptyStackException e) {
          Assert.shouldNeverReachHere
              ("available was empty on pop");
          result = new Resource();
          _allocated.push(result);
          return result;
      }
    }
}
class Assert...
    static void shouldNeverReachHere(String message) {
      throw new RuntimeException (message);
    }
```

编译并测试。如果一切运转正常，就可以将try区段中的代码拷贝到try区段之外，然后将try区段全部移除：

```
Resource getResource() {
    Resource result;
    if (_available.isEmpty()) {
      result = new Resource();
      _allocated.push(result);
      return result;
```

```
        }
        else {
            result = (Resource) _available.pop();
            _allocated.push(result);
            return result;
        }
    }
```

在这之后我常常发现，我可以对条件代码（conditional code）进行整理。本例之中我可以使用 *Consolidate Duplicate Conditional Fragments*（243）：

```
Resource getResource() {
    Resource result;
    if (_available.isEmpty())
        result = new Resource();
    else
        result = (Resource) _available.pop();
    _allocated.push(result);
    return result;
}
```

11

处理概括关系

Dealing with Generalization

有一批重构手法专门用来处理「概括关系」（**generalization**；译注：这里指的是「class 继承」这档事），其中主要是将函数（methods）上下移动于继承体系之中。*Pull Up Field*（320）和 *Pull Up Method*（322）都用于将 class 特性向继承体系的上端移动，*Push Down Method*（328）和 *Push Down Field*（329）则将 class 特性向继承体系的下端移动。构造函数比较难以向上拉动，因此专门有一个 *Pull Up Constructor Body*（325）处理它。我们不会将构造函数往下推，因为 *Replace Constructor with Factory Method*（304）通常更管用。

如果有若干函数大体上相同，只在细节上有所差异，可以使用 *Form Template Method*（345）将它们的共同点和不同点分开。

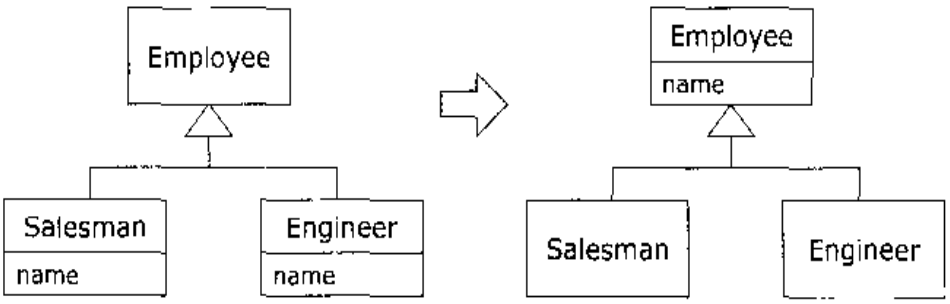
除了在继承体系中移动 class 特性之外，你还可以建立新 classes，改变整个继承体系。*Extract Subclass*（330）、*Extract Superclass*（336）和 *Extract Interface*（341）都是这样的重构手法，它们在继承体系的不同位置构造出新元素。如果你想在型别系统（type system）中标示（*mark*）一小部分函数，*Extract Interface*（341）特别有用。如果你发现继承体系中的某些 classes 没有存在必要，可以使用 *Collapse Hierarchy*（344）将它们移除。

有时候你会发现继承并非最佳选择，你真正需要的其实是委托（delegation），那么，*Replace Inheritance with Delegation*（352）可以帮助你把继承改为委托。有时候你又会想要做反向修改，此时就可使用 *Replace Delegation with Inheritance*（355）。

11.1 Pull Up Field（值域上移）

两个 subclasses 拥有相同的值域。

将此一值域移至 superclass。



动机（Motivation）

如果各个 subclass 是分别开发的，或者是在重构过程中组合起来的，你常会发现它们拥有重复特性，特别是值域更容易重复。这样的值域有时拥有近似的名字，但也并非绝对如此。判断若干值域是否重复，惟一的办法就是观察函数如何使用它们。如果它们被使用的方式很相似，你就可以将它们归纳到 superclass 去。

本项重构从两方面减少重复：首先它去除了「重复的数据声明」；其次它使你可以将使用该值域的行为从 subclass 移至 superclass，从而去除「重复的行为」。

作法（Mechanics）

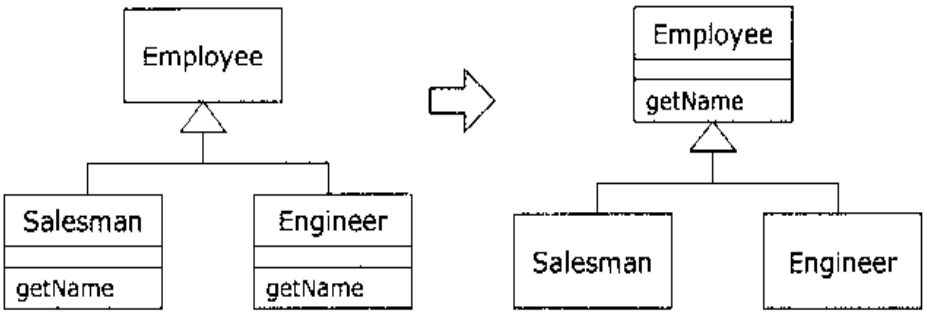
- 针对待提升之值域，检查它们的所有被使用点，确认它们以同样的方式被使用。
- 如果这些值域的名称不同，先将它们改名，使每一个名称都和你想为 superclass 值域取的名称相同。

- 编译，测试。
- 在 superclass 中新建一个值域。
  - ⇨ 如果这些值域是 private，你必须将 superclass 的值域声明为 protected，这样 subclasses 才能引用它。
- 移除 subclass 中的值域。
- 编译，测试。
- 考虑对 superclass 的新建值域使用 *Self Encapsulate Field*（171）。

11.2 Pull Up Method（函数上移）

有些函数，在各个 subclass 中产生完全相同的结果。

将该函数移至 superclass。



动机（Motivation）

避免「行为重复」是很重要的。尽管「重复的两个函数」也可以各自工作得很好，但「重复」自身会成为错误的滋生地，此外别无价值。无论何时，只要系统之内出现重复，你就会面临「修改其中一个却未能修改另一个」的风险。通常，找出重复也有一定困难。

如果某个函数在各 subclass 中的函数体都相同（它们很可能是通过「拷贝-粘贴」得到的），这就是最显而易见的 *Pull Up Method*（322）适用场合。当然，情况并不总是如此明显。你也可以只管放心地重构，再看看测试程序会不会发牢骚，但这就需要对你的测试有充分的信心。我发现，观察这些可疑（可能重复的）函数之间的差异往往大有收获：它们经常会向我展示那些我忘记测试的行为。

Pull Up Method（322）常常紧随其他重构而被使用。也许你能找出若干个「身处不同 subclasses 内的函数」而它们又可以「通过某种形式的参数调整」而后成为相同函数。这时候，最简单的办法就是首先分别调整这些函数的参数，然后再将它们概括（*generalize*）到 superclass 中。当然，如果你自信足够，也可以一次同时完成这两个步骤。

有一种特殊情况也需要使用 *Pull Up Method*（322）：subclass 的函数覆写（overrides）了 superclass 的函数，但却仍然做相同的工作。

Pull Up Method（322）过程中最麻烦的一点就是：被提升的函数可能会引用「只出现于 subclass 而不出现于 superclass」的特性。如果被引用的是个函数，你可以将该函数也一同提升到 superclass，或者在 superclass 中建立一个抽象函数。在此过程中，你可能需要修改某个函数的签名式（signature），或建立一个委托函数（delegating method）。

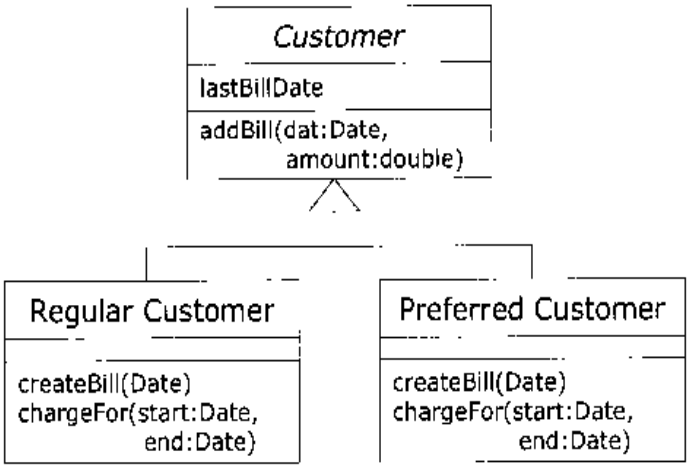
如果两个函数相似但不相同，你或许可以先以 *Form Template Method*（345）构造出相同的函数，然后再提升它们。

作法（Mechanics）

- 检查「待提升函数」，确定它们是完全一致的（identical）。
  - ⇨ 如果这些函数看上去做了相同的事，但并不完全一致，可使用 *Substitute Algorithm*（139）让它们变得完全一致。
- 如果「待提升函数」的签名式（signature）不同，将那些签名式都修改为你想要在 superclass 中使用的签名式。
- 在 superclass 中新建一个函数，将某一个「待提升函数」的代码拷贝到其中，做适当调整，然后编译。
  - ⇨ 如果你使用的是一种强型（strongly typed）语言，而「待提升函数」又调用了一个「只出现于 subclass，未出现于 superclass」的函数，你可以在 superclass 中为被调用函数声明一个抽象函数。
  - ⇨ 如果「待提升函数」使用了 subclass 的一个值域，你可以使用 *Pull Up Field*（320）将该值域也提升到 superclass；或者也可以先使用 *Self Encapsulate Field*（171），然后在 superclass 中把取值函数（getter）声明为抽象函数。
- 移除一个「待提升的 subclass 函数」。
- 编译，测试。
- 逐一移除「待提升的 subclass 函数」，直到只剩下 superclass 中的函数为止。每次移除之后都需要测试。
- 观察该函数的调用者，看看是否可以将它所索求的对象型别改为 superclass。

范例（Example）

我以 `Customer` 表示「顾客」，它有两个 subclass：表示「普通顾客」的 `RegularCustomer` 和表示「贵宾」的 `PreferredCustomer`。



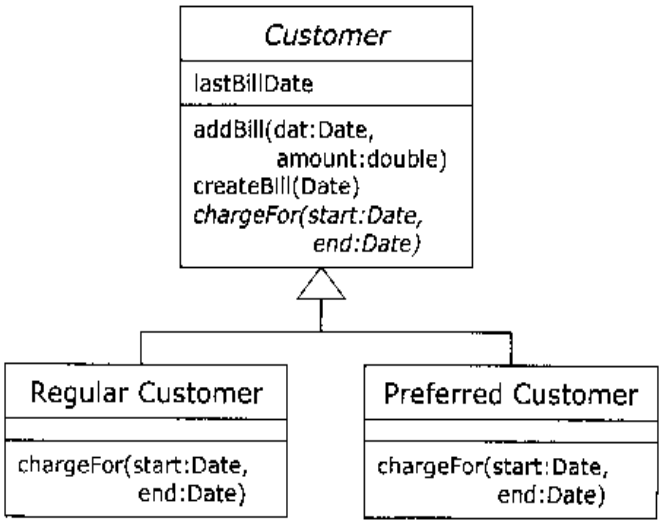
两个 subclasses 都有一个 `createBill()`函数，并且代码完全一样：

```
void createBill (date Date) {
    double chargeAmount = chargeFor (lastBillDate, date);
    addBill (date, charge);
}
```

但我不能直接把这个函数上移到 superclass，因为各个 subclass 的 `chargeFor()`函数并不相同。我必须先在 superclass 中声明 `chargeFor()`抽象函数：

```
class Customer...
    abstract double chargeFor(date start, date end)
```

然后，我就可以将 `createBill()`函数从其中一个 subclass 拷贝到 superclass。拷贝完之后应该编译，然后移除那个 subclass 的 `createBill()`函数，然后编译并测试。随后再移除另一个 subclass 的 `createBill()`函数，再次编译并测试：



11.3 Pull Up Constructor Body（构造函数本体上移）

你在各个 subclass 中拥有一些构造函数，它们的本体（代码）几乎完全一致。

在 superclass 中新建一个构造函数，并在 subclass 构造函数中调用它。

```
class Manager extends Employee...
  public Manager (String name, String id, int grade) {
     _name = name;
     _id = id;
     _grade = grade;
  }
```

```
  public Manager (String name, String id, int grade) {
     super (name, id);
     _grade = grade;
  }
```

动机（Motivation）

构造函数（constructors）是很奇妙的东西。它们不是普通函数，使用它们比使用普通函数受到更多的限制。

如果你看见各个 subclass 中的函数有共同行为，你的第一个念头应该是将共同行为提炼到一个独立函数中，然后将这个函数提升到 superclass。对构造函数而言，它们彼此的共同行为往往就是「对象的建构」。这时候你需要在 superclass 中提供一个构造函数，然后让 subclass 都来调用它。很多时候，「调用 superclass 构造函数」就是 subclass 构造函数的惟一动作。这里不能运用 *Pull Up Method*（322），因为你无法在 subclass 中继承 superclass 构造函数（你可曾痛恨过这个规定？）。

如果重构过程过于复杂，你可以考虑转而使用 *Replace Constructor with Factory Method*（304）。

作法（Mechanics）

- 在 superclass 中定义一个构造函数。
- 将 subclass 构造函数中的共同代码搬移到 superclass 构造函数中。
  - ⇨ 被搬移的可能是 subclass 构造函数的全部内容。
  - ⇨ 首先设法将共同代码搬移到 subclass 构造函数起始处，然后再拷贝到 superclass 构造函数中。
- 将 subclass 构造函数中的共同代码删掉，改而调用新建的 superclass 构造函数。
  - ⇨ 如果 subclass 构造函数中的所有代码都是共同码，那么对 superclass 构造函数的调用将是 subclass 构造函数的惟一动作。
- 编译，测试。
  - ⇨ 如果日后 subclass 构造函数再出现共同代码，你可以首先使用 *Extract Method*（110）将那一部分提炼到一个独立函数，然后使用 *Pull Up Method*（322）将该函数上移到 superclass。

范例（Example）

下面是一个表示「雇员」的 Employee class 和一个表示「经理」的 Manager class：

```
class Employee...
    protected String _name;
    protected String _id;

class Manager extends Employee...
    public Manager (String name, String id, int grade) {
      _name = name;
      _id = id;
      _grade = grade;
    }
    private int _grade;
```

Employee 的值域应该在 Employee 构造函数中被设妥初值。因此我定义了一个 Employee 构造函数，并将它声明为 protected，表示 subclass 应该调用它：

```
class Employee...
    protected Employee (String name, String id) {
      _name = name;
      _id = id;
```

```
    }
```

然后，我从 subclass 中调用它：

```
public Manager (String name, String id, int grade) {
    super (name, id);
    _grade = grade;
}
```

后来情况又有些变化，构造函数中出现了共同代码。假如我有以下代码：

```
class Employee...
    boolean isPriviliged() {..}
    void assignCar() {..}
class Manager...
    public Manager (String name, String id, int grade) {
      super (name, id);
      _grade = grade;
      if (isPriviliged()) assignCar();  //every subclass does this
    }
    boolean isPriviliged() {
      return _grade > 4;
    }
```

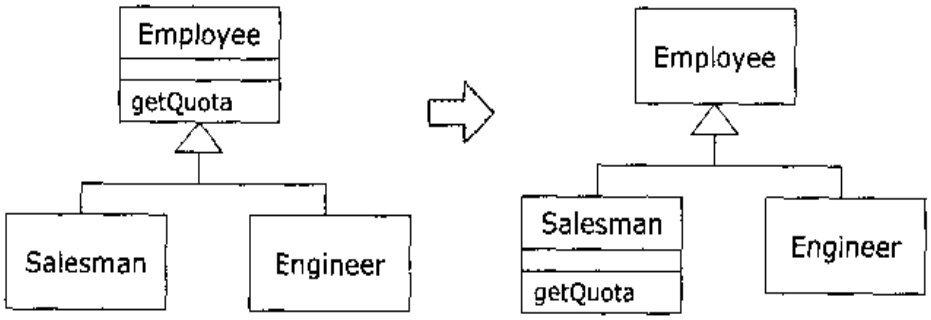
我不能把调用 assignCar() 的行为移到 superclass 构造函数中，因为惟有把合适的值赋给 _grade 值域后才能执行 assignCar()。此时我需要 *Extract Method*（110）和 *Pull Up Method*（322）。

```
class Employee...
    void initialize() {
      if (isPriviliged()) assignCar();
    }
class Manager...
    public Manager (String name, String id, int grade) {
      super (name, id);
      _grade = grade;
      initialize();
    }
```

11.4 Push Down Method（函数下移）

superclass 中的某个函数只与部分（而非全部）subclasses 有关。

将这个函数移到相关的那些 subclasses 去。



动机（Motivation）

Push Down Method（328）恰恰相反于 *Pull Up Method*（322）。当我有必要把某些行为从 superclass 移至特定的 subclass 时，我就使用 *Push Down Method*（328），它通常也只在这种时候有用。使用 *Extract Subclass*（330）之后你可能会需要它。

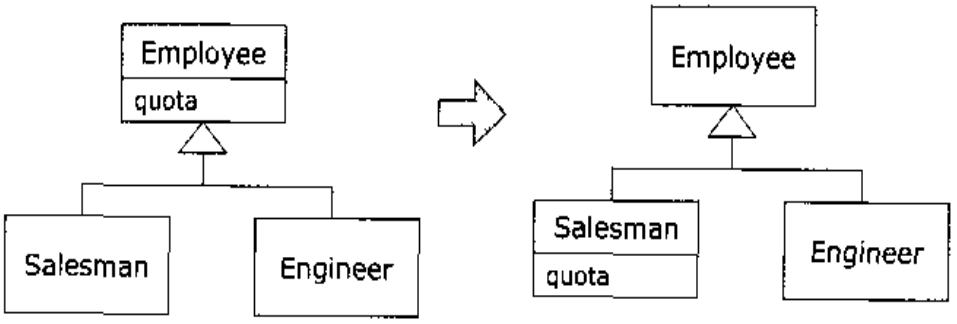
作法（Mechanics）

- 在所有 subclass 中声明该函数，将 superclass 中的函数本体拷贝到每一个 subclass 函数中。
  - ⇨ 你可能需要将 superclass 的某些值域声明为 protected，让 subclass 函数也能够访问它们。如果日后你也想把这些值域下移到 subclasses，通常就可以那么做；否则应该使用 superclass 提供的访问函数（accessors）。如果访问函数并非 public，你得将它声明为 protected。
- 删除 superclass 中的函数。
  - ⇨ 你可能必须修改调用端的某些变量声明或参数声明，以便能够使用 subclass。
  - ⇨ 如果有必要通过一个 superclass 对象访问该函数，或如果你不想把该函数从任何 subclass 中移除，或如果 superclass 是抽象类，那么你就可以在 superclass 中把该函数声明为抽象函数。
- 编译，测试。
- 将该函数从所有不需要它的那些 subclasses 中删掉。
- 编译，测试。

11.5 Push Down Field（值域下移）

superclass 中的某个值域只被部分（而非全部）subclasses 用到。

将这个值域移到需要它的那些 subclasses 去。



动机（Motivation）

Push Down Field（329）恰恰相反于 *Pull Up Field*（320）。如果只有某些（而非全部）subclasses 需要 superclass 内的一个值域，你可以使用本项重构。

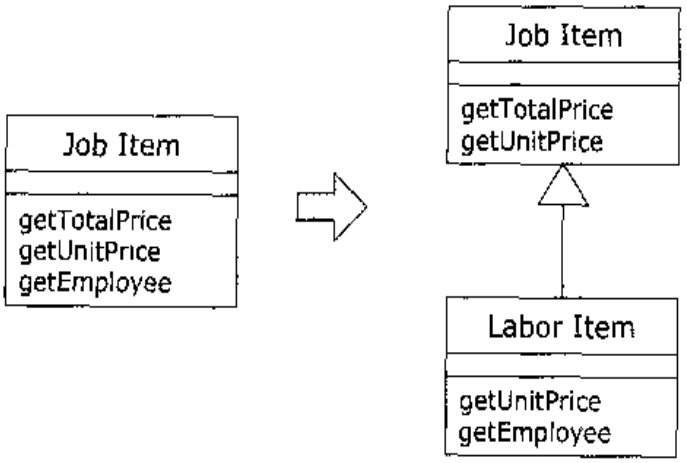
作法（Mechanics）

- 在所有 subclass 中声明该值域。
- 将该值域从 superclass 中移除。
- 编译，测试。
- 将该值域从所有不需要它的那些 subclasses 中删掉。
- 编译，测试。

11.6 Extract Subclass（提炼子类）

class 中的某些特性（features）只被某些（而非全部）实体（instances）用到。

新建一个 subclass，将上面所说的那一部分特性移到 subclass 中。



动机（Motivation）

使用 *Extract Subclass*（330）的主要动机是：你发现 class 中的某些行为只被一部分实体用到，其他实体不需要它们。有时候这种行为上的差异是通过 type code 区分的，此时你可以使用 *Replace Type Code with Subclasses*（223）或 *Replace Type Code with State/Strategy*（227）。但是，并非一定要出现了 type code 才表示需要考虑使用 subclass。

Extract Class（149）是 *Extract Subclass*（330）之外的另一种选择，两者之间的抉择其实就是委托（delegation）和继承（inheritance）之间的抉择。*Extract Subclass*（330）通常更容易进行，但它也有限制：一旦对象创建完成，你无法再改变「与型别相关的行为」（class-based behavior）。但如果使用 *Extract Class*（149），你只需插入另一个不同组件（plugging in different components）就可以改变对象的行为。此外，subclasses 只能用以表现一组变化（a set of variations）。如果你希望 class 以数种不同的方式变化，就必须使用委托（delegation）。

作法（Mechanics）

- 为 source class 定义一个新的 subclass。
- 为这个新的 subclass 提供构造函数。
  - ⇨ 简单的作法是：让 subclass 构造函数接受与 superclass 构造函数相同的参数，并通过 super 调用 superclass 构造函数。
  - ⇨ 如果你希望对用户隐藏 subclass 的存在，可使用 *Replace Constructor with Factory Method*（304）。
- 找出调用 superclass 构造函数的所有地点。如果它们需要的是新建的 subclass，令它们改而调用新构造函数。
  - ⇨ 如果 subclass 构造函数需要的参数和 superclass 构造函数的参数不同，可以使用 *Rename Method*（273）修改其参数列。如果 subclass 构造函数不需要 superclass 构造函数的某些参数，可以使用 *Rename Method*（273）将它们去除。
  - ⇨ 如果不再需要直接实体化（具现化，instantiated）superclass，就将它声明为抽象类。
- 逐一使用 *Push Down Method*（328）和 *Push Down Field*（329）将 source class 的特性移到 subclass 去。
  - ⇨ 和 *Extract Class*（149）不同的是，先处理函数再处理数据，通常会简单一些。
  - ⇨ 当一个 public 函数被下移到 subclass 后，你可能需要重新定义该函数的调用端的局部变量或参数型别，让它们改调用 subclass 中的新函数。如果忘记进行这一步骤，编译器会提醒你。
- 找到所有这样的值域：它们所传达的信息如今可由继承体系自身传达（这一类值域通常是 boolean 变量或 type code），以 *Self Encapsulate Field*（171）避免直接使用这些值域，然后将它们的取值函数（getter）替换为多态常量函数（polymorphic constant methods），所有使用这些值域的地方都应该以 *Replace Conditional with Polymorphism*（255）重构。
  - ⇨ 任何函数如果位于 source class 之外，而又使用了上述值域的访问函数（accessors），考虑以 *Move Method*（142）将它移到 source class 中，然后再使用 *Replace Conditional with Polymorphism*（255）。
- 每次下移之后，编译并测试。

范例（Example）

下面是 JobItem class，用来决定当地修车厂的工作报价：

```
class JobItem ...
    public JobItem (int unitPrice, int quantity,
                    boolean isLabor, Employee employee) {
      _unitPrice = unitPrice;
      _quantity = quantity;
      _isLabor = isLabor;
      _employee = employee;
    }
    public int getTotalPrice() {
      return getUnitPrice() * _quantity;
    }
    public int getUnitPrice(){
      return (_isLabor) ?
          _employee.getRate():
          _unitPrice;
    }
    public int getQuantity(){
      return _quantity;
    }
    public Employee getEmployee() {
      return _employee;
    }
    private int _unitPrice;
    private int _quantity;
    private Employee _employee;
    private boolean _isLabor;

class Employee...
    public Employee (int rate) {
      _rate = rate;
    }
    public int getRate() {
      return _rate;
    }
    private int _rate;
```

我要提炼出一个 LaborItem subclass，因为上述某些行为和数据只在 labor（劳工）情况下才需要。首先建立这样一个 class：

```
class LaborItem extends JobItem {}
```

我需要为 LaborItem 提供一个构造函数，因为 JobItem 没有「无引数构造函数」（no-argument constructor）。我把 superclass 构造函数的参数列拷贝过来：

```
public LaborItem (int unitPrice, int quantity,
                  boolean isLabor, Employee employee) {
    super (unitPrice, quantity, isLabor, employee);
}
```

这就足以让新的 subclass 通过编译了。但是这个构造函数会造成混淆：某些参数是 LaborItem 所需要的，另一些不是。稍后我再来解决这个问题。

下一步是要找出对 JobItem 构造函数的调用，并从中找出「可替换为 LaborItem 构造函数」者。因此，下列语句：

```
JobItem j1 = new JobItem (0, 5, true, kent); //译注：labor case
```

就被修改为：

```
JobItem j1 = new LaborItem (0, 5, true, kent);
```

此时我尚未修改变量型别，只是修改了构造函数所属的 class。之所以这样做，是因为我希望只在必要地点才使用新型别。到目前为止，subclass 还没有专属接口，因此我还不想宣布任何改变。

现在正是清理构造函数参数列的好时机。我将针对每个构造函数使用 *Rename Method*（273），首先处理 superclass 构造函数。我要新建一个构造函数，并把旧构造函数声明为 protected（不能直接声明为 private，因为 subclass 还需要它）：

```
class JobItem...
  protected JobItem (int unitPrice, int quantity,
                     boolean isLabor, Employee employee) {
    _unitPrice = unitPrice;
    _quantity = quantity;
    _isLabor = isLabor;
    _employee = employee;
  }
  public JobItem (int unitPrice, int quantity) {
    this (unitPrice, quantity, false, null)
  }
```

现在，外部调用应该使用新构造函数：

```
JobItem j2 = new JobItem (10, 15);
```

编译、测试都通过后，我再使用 *Rename Method*（273）修改 subclass 构造函数：

```
class LaborItem...
  public LaborItem (int quantity, Employee employee) {
    super (0, quantity, true, employee);
  }
```

此时的我仍然暂时使用 protected superclass 构造函数。

现在，我可以将 JobItem 的特性向下搬移。先从函数开始，我先运用 *Push Down Method*（328）对付 getEmployee()函数：

```
class LaborItem...
  public Employee getEmployee() {
    return _employee;
  }

class JobItem...
  protected Employee _employee;
```

因为 _employee 值域也将在稍后被下移到 LaborItem，所以我现在先将它声明为 protected。

将 _employee 值域声明为protected之后，我可以再次清理构造函数，让 _employee 只在「即将去达的 subclass 中」被初始化：

```
class JobItem...
  protected JobItem (int unitPrice, int quantity, boolean isLabor)
  {
      _unitPrice = unitPrice;
      _quantity = quantity;
      _isLabor = isLabor;
  }
class LaborItem ...
  public LaborItem (int quantity, Employee employee) {
    super (0, quantity, true);
    _employee = employee;
  }
```

_isLabor 值域所传达的信息，现在已经成为继承体系的内在信息，因此我可以移除这个值域了。最好的方式是：先使用 *Self Encapsulate Field*（171），然后再修改访问函数(accessors)，改用多态常量函数。所谓「多态常量函数」会在不同的 subclass 实现版本中返回不同的固定值：

```
class JobItem...
  protected boolean isLabor() {
    return false;
  }

class LaborItem...
  protected boolean isLabor() {
    return true;
  }
```

然后，我就可以摆脱 _isLabor 值域了。

现在，我可以观察 isLabor()函数的用户，并运用 *Replace Conditional with Polymorphism*（255）重构它们。我找到了下列这样的函数：

```
class JobItem...
  public int getUnitPrice(){
    return (isLabor()) ?
        _employee.getRate():
        _unitPrice;
  }
```

将它重构为：

```
class JobItem...
  public int getUnitPrice(){
    return _unitPrice;
  }

class LaborItem...
  public int getUnitPrice(){
    return _employee.getRate();
  }
```

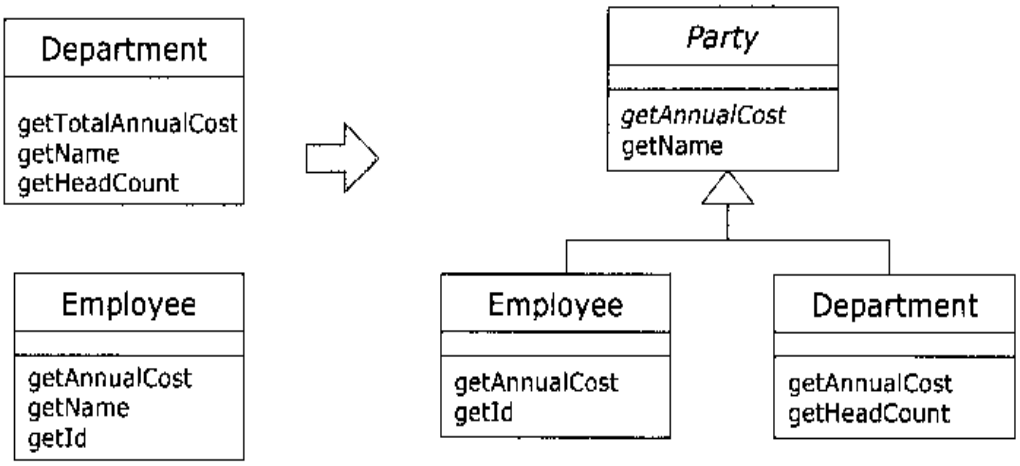
当使用某项值域的函数全被下移至subclass后，我就可以使用 *Push Down Field*（329）将值域也下移。如果尚还无法移动值域，那就表示，我需要对函数做更多处理，可能需要实施 *Push Down Method*（328）或 *Replace Conditional with Polymorphism*（255）。

由于 _unitPrice 值域只被 LaborItem 以外的对象（也就是 parts job items）所用，所以我可以再次运用 *Extract Subclass*（330）对 JobItem 提炼出一个 subclass：PartsItem。完成后，我可以将 JobItem 声明为抽象类。

11.7 Extract Superclass（提炼超类）

两个 classes 有相似特性（similar features）。

为这两个 classes 建立一个 superclass，将相同特性移至 superclass。



动机（Motivation）

重复代码是系统中最主要的一种糟糕东西。如果你在不同的地方进行相同一件事情，一旦需要修改那些动作时，你就得负担比你原本应该负担的更多事情。

重复代码的某种形式就是：两个 classes 以相同的方式做类似的事情，或者以不同的方式做类似的事情。对象提供了一种简化这种情况的机制，那就是继承机制。但是，在建立这些具有共通性的 classes 之前，你往往无法发现这样的共通性，因此你经常会在"具有共通性"的 classes 存在之后，再开始建立其间的继承结构。

另一种选择就是 *Extract Class*（149）。这两种方案之间的选择其实就是继承（inheritance）和委托（delegation）之间的选择。如果两个 classes 可以共享行为，也可以共享接口，那么继承是比较简单的作法。如果你选错了，也总有 *Replace Inheritance with Delegation*（352）这瓶后悔药可吃。

作法（Mechanics）

- 为原本的 classes 新建一个空白的 abstract superclass。
- 运用 *Pull Up Field*（320）、*Pull Up Method*（322）和 *Pull Up Constructor Body*（325）逐一将 subclass 的共同元素上移到 superclass。
  - ⇨ 先搬移值域，通常比较简单。
  - ⇨ 如果相应的 subclass 函数有不同的签名式（signature），但用途相同，可以先使用 *Rename Method*（273）将它们的签名式改为相同，然后再使用 *Pull Up Method*（322）。
  - ⇨ 如果相应的 subclass 函数有相同的签名式，但函数本体不同，可以在 superclass 中把它们的共同签名式声明为抽象函数。
  - ⇨ 如果相应的 subclass 函数有不同的函数本体，但用途相同，可试着使用 *Substitute Algorithm*（139）把其中一个函数的函数本体拷贝到另一个函数中。如果运转正常，你就可以使用 *Pull Up Method*（322）。
- 每次上移后，编译并测试。
- 检查留在 subclass 中的函数，看它们是否还有共通成分。如果有，可以使用 *Extract Method*（110）将共通部分再提炼出来，然后使用 *Pull Up Method*（322）将提炼出的函数上移到 superclass。如果各个 subclass 中某个函数的整体流程很相似，你也许可以使用 *Form Template Method*（345）。
- 将所有共通元素都上移到 superclass 之后，检查 subclass 的所有用户。如果它们只使用共同接口，你就可以把它们所索求的对象型别改为 superclass。

范例（Example）

下面例中，我以 `Employee` 表示「员工」，以 `Department` 表示「部门」：

```
class Employee...
  public Employee (String name, String id, int annualCost) {
    _name = name;
    _id = id;
    _annualCost = annualCost;
  }
  public int getAnnualCost() {
    return _annualCost;
  }
```

```
    public String getId(){
      return _id;
    }
    public String getName() {
      return _name;
    }
    private String _name;
    private int _annualCost;
    private String _id;

  public class Department...
    public Department (String name) {
      _name = name;
    }
    public int getTotalAnnualCost(){
      Enumeration e = getStaff();
      int result = 0;
      while (e.hasMoreElements()) {
          Employee each = (Employee) e.nextElement();
          result += each.getAnnualCost();
      }
      return result;
    }
    public int getHeadCount() {
      return _staff.size();
    }
    public Enumeration getStaff() {
      return _staff.elements();
    }
    public void addStaff(Employee arg) {
      _staff.addElement(arg);
    }
    public String getName() {
      return _name;
    }
    private String _name;
    private Vector _staff = new Vector();
```

这里有两处共同点。首先，员工和部门都有名称（names）；其次，它们都有年度成本（annual costs），只不过计算方式略有不同。我要提炼出一个 superclass，用以包容这些共通特性。第一步是新建这个 superclass，并将现有的两个 classes 定义为其 subclasses：

```
abstract class Party {}
class Employee extends Party...
class Department extends Party...
```

然后我开始把特性上移至 superclass。先实施 *Pull Up Field*（320）通常会比较简单：

```
class Party...
  protected String _name;
```

然后，我可以使用 *Pull Up Method*（322）把这个值域的取值函数（getter）也上移至 superclass：

```
class Party {
   public String getName() {
      return _name;
   }
```

我通常会把这个值域声明为 private。不过，在此之前，我需要先使用 *Pull Up Constructor Body*（325），这样才能对 _name 正确赋值：

```
class Party...
   protected Party (String name) {
      _name = name;
   }
   private String _name;

class Employee...
   public Employee (String name, String id, int annualCost) {
      super (name);
      _id = id;
      _annualCost = annualCost;
   }

class Department...
   public Department (String name) {
      super (name);
   }
```

Department.getTotalAnnualCost()和 Employee.getAnnualCost()两个函数的用途相同，因此它们应该有相同的名称。我先运用 *Rename Method*（273）把它们的名称改为相同：

```
class Department extends Party {
   public int getAnnualCost(){
      Enumeration e = getStaff();
      int result = 0;
      while (e.hasMoreElements()) {
          Employee each = (Employee) e.nextElement();
          result += each.getAnnualCost();
      }
      return result;
   }
```

它们的函数本体仍然不同，因此我目前还无法使用 *Pull Up Method*（322）。但是我可以在 superclass 中声明一个抽象函数：

```
abstract public int getAnnualCost()
```

这一步修改完成后，我需要观察两个 subclasses 的用户，看看是否可以改变它们转而使用新的 superclass。用户之一就是 Department 自身，它保存了一个 Employee 对象群集。Department.getAnnualCost()只调用群集内的元素（对象）的 getAnnualCost()函数，而该函数此刻乃是在 Party class 中声明的：

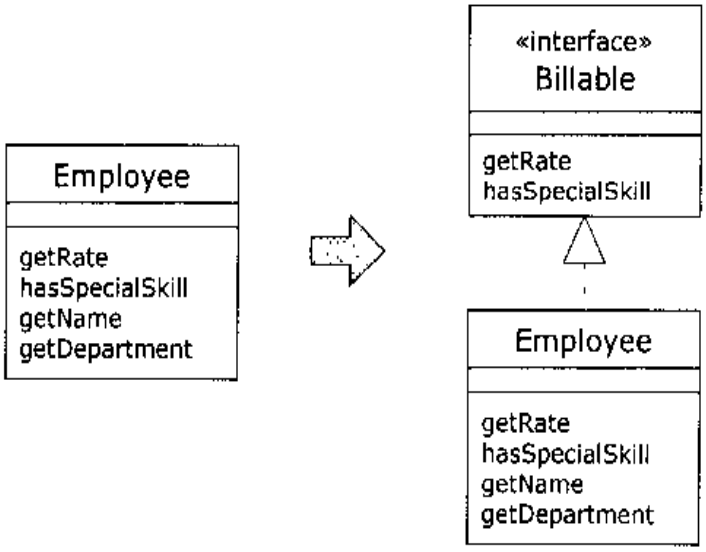
```
class Department...
  public int getAnnualCost(){
    Enumeration e = getStaff();
    int result = 0;
    while (e.hasMoreElements()) {
        Party each = (Party) e.nextElement();
        result += each.getAnnualCost();
    }
    return result;
  }
```

这一行为暗示一种新的可能性：我可以用 ***Composite*** 模式 [Gang of Four] 来对待 Department 和 Employee，这样就可以让一个 Department 对象包容另一个 Department 对象。这是一项新功能，所以这项修改严格来说不属于重构范围。如果用户恰好需要 ***Composite*** 模式，我可以修改 _staff 值域名字，使其更好地表现这一模式。这一修改还会带来其他相应修改：修改 addStaff()函数名称，并将该函数的参数型别改为 Party class。最后还需要把 headCount()函数变成一个递归调用。我的作法是在 Employee 中建立一个 headCount()函数，让它返回 1；再使用 *Substitute Algorithm*（139）修改 Department 的 headCount()函数，让它总和（add）各部门的 headCount()调用结果。

11.8 Extract Interface（提炼接口）

若干客户使用 class 接口中的同一子集；或者，两个 classes 的接口有部分相同。

将相同的子集提炼到一个独立接口中。



动机（Motivation）

classes 之间彼此互用的方式有若干种。「使用一个 class」通常意味覆盖该 class 的所有责任区（whole area of responsibilities）。另一种情况是，某一组客户只使用 class 责任区中的一个特定子集。再一种情况则是，class 需要与「所有可协助处理某些特定请求」的 classes 合作。

对于后两种情况，将「被使用之部分责任」分离出来通常很有意义，因为这样可以使系统的用法更清晰，同时也更容易看清系统的责任划分。如果新的 classes 需要支持上述子集，也比较能够看清子集内有些什么东西。

在许多面向对象语言中，这种「责任划分」能力是通过多重继承（multiple inheritance）支持的。你可以针对一段行为（each segment of behavior）建立一个 class，再将它们组合于一份实现品（implementation）中。Java 只提供单一继承（single inheritance），但你可以运用 interfaces（接口）来昭示并实现上述需求。interfaces 对于 Java 程序的设计方式有着巨大的影响，就连 Smalltalk 程序员都认为 interfaces（接口）是一大进步！

Extract Superclass（336）和 *Extract Interface*（341）之间有些相似之处。*Extract Interface*（341）只能提炼共通接口，不能提炼共通代码。使用 *Extract Interface*（341）可能造成难闻的「重复」臭味，幸而你可以运用 *Extract Class*（149）先把共通行为放进一个组件（component）中，然后将工作委托（*delegating*）该组件，从而解决这个问题。如果有不少共通行为，*Extract Superclass*（336）会比较简单，但是每个 class 只能有一个 superclass（译注：每个 class 却能有多个 interfaces）。

如果某个 class 在不同环境下扮演截然不同的角色，使用 interface（接口）就是个好主意。你可以针对每个角色以 *Extract Interface*（341）提炼出相应接口。另一种可以用上 *Extract Interface*（341）的情况是：你想要描述一个 class 的外驶接口（outbound interface，亦即这个 class 对其 server 所进行的操作）。如果你打算将来加入其他种类的 server，只需要求它们实现这个接口即可。

作法（Mechanics）

- 新建一个空接口（empty interface）。
- 在接口中声明「待提炼类」的共通操作。
- 让相关的 classes 实现上述接口。
- 调整客户端的型别声明，使得以运用该接口。

范例（Example）

`TimeSheet` class 表示「月报表」，其中将计算花在员工身上的费用。为了计算这笔费用，`TimeSheet` 需要知道员工级别，以及该员工是否有特殊技能：

```
double charge(Employee emp, int days) {
    int base = emp.getRate() * days;
    if (emp.hasSpecialSkill())
       return base * 1.05;
   else return base;
}
```

除了提供员工的索费级别和特殊技能信息外，`Employee` 还有很多其他方面的功能，但本应用程序只需这两项功能。我可以针对这两项功能定义一个接口，从而强调「我只需要这部分功能」的事实：

```
interface Billable {
    public int getRate();
    public boolean hasSpecialSkill();
}
```

然后，我声明让 Employee 实现这个接口：

```
class Employee implements Billable ...
```

完成以后，我可以修改 charge()函数声明，强调该函数只使用 Employee 的这部分行为：

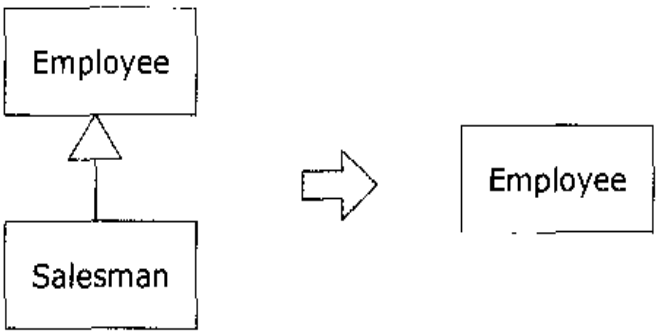
```
double charge(Billable emp, int days) {
    int base = emp.getRate() * days;
    if (emp.hasSpecialSkill())
      return base * 1.05;
    else return base;
}
```

此刻，我们只不过是在文档化（documentability）方面获得了一些适度收获。对函数，这样的收获并没有太大价值；但如果有若干 classes 都使用 Billable 接口，它就会很有用。如果我还想计算计算器租金，巨大的收获就显露出来了。为了让公司里的计算器都「能够被计费」（billable），我只需让 Computer class 实现 Billable 接口，然后就可以把计算器租金登记到月报表上了。

11.9 Collapse Hierarchy（折叠继承体系）

superclass 和 subclass 之间无太大区别。

将它们合为一体。



动机（Motivation）

如果你曾经编写过继承体系，你就会知道，继承体系很容易变得过分复杂。所谓重构继承体系，往往是将函数和值域在体系中上下移动。完成这些动作后，你很可能发现某个 subclass 并未带来该有的价值，因此需要把 classes 并合（折叠）起来。

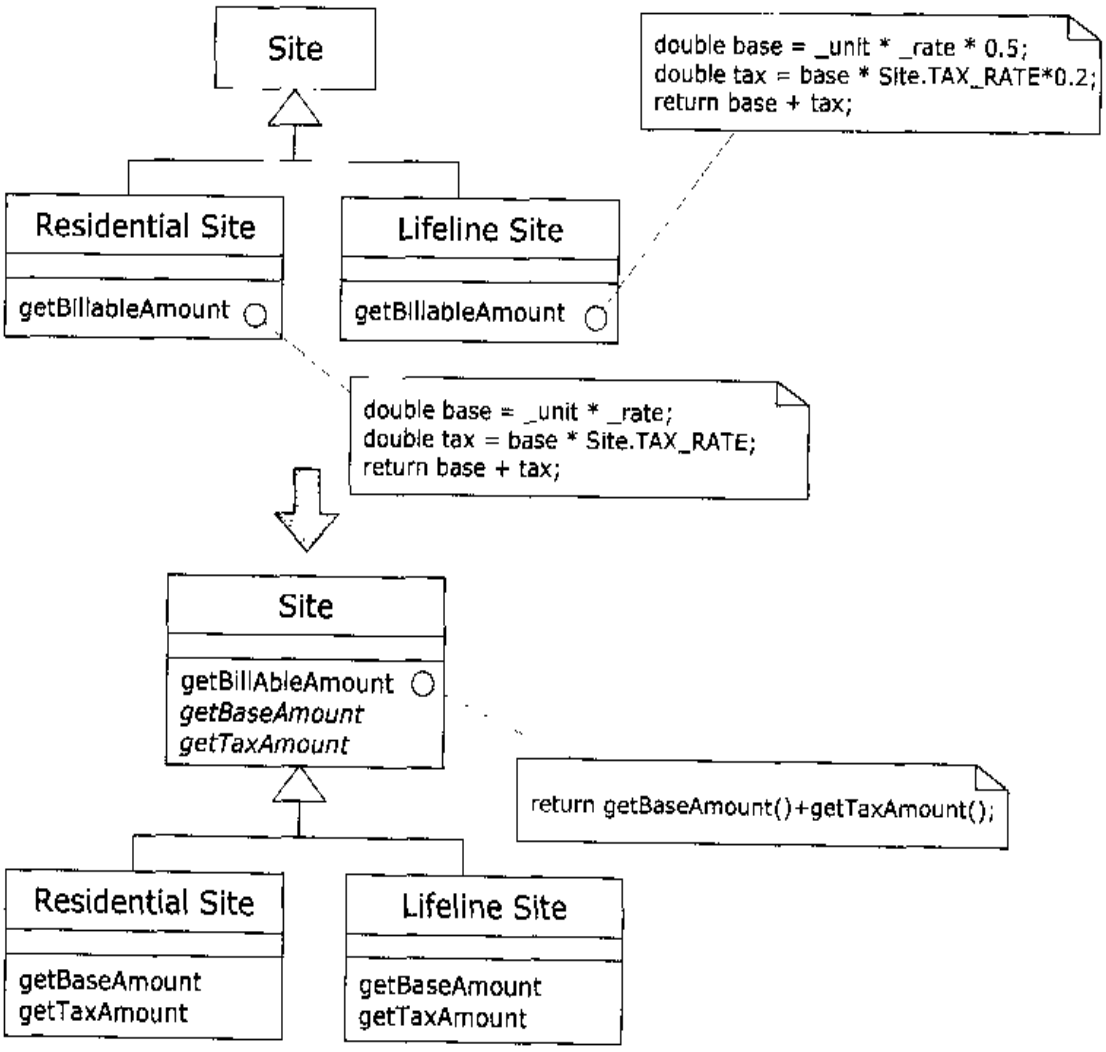
作法（Mechanics）

- 选择你想移除的 class：是 superclass 还是 subclass？
- 使用 *Pull Up Field*（320）和 *Pull Up Method*（322），或者 *Push Down Method*（328）和 *Push Down Field*（329），把想要移除的 class 内的所有行为和数据（值域）搬移到另一个 class。
- 每次移动后，编译并测试。
- 调整「即将被移除的那个 class」的所有引用点，令它们改而引用合并（折叠）后留下的 class。这个动作将会影响变量的声明、参数的型别以及构造函数。
- 移除我们的目标；此时的它应该已经成为一个空类（empty class）。
- 编译，测试。

11.10 Form Template Method（塑造模板函数）

你有一些 subclasses，其中相应的某些函数以相同顺序执行类似的措施，但各措施实际上有所不同。

将各个措施分别放进独立函数中，并保持它们都有相同的签名式（signature），于是原函数也就变得相同了。然后将原函数上移至 superclass。



动机（Motivation）

继承是「避免重复行为」的一个强大工具。无论何时，只要你看见两个 subclasses 之中有类似的函数，就可以把它们提升到 superclass。但是如果这些函数并不完全相同呢？此时的你应该怎么办？我们仍有必要尽量避免重复，但又必须保持这些函数之间的实质差异。

常见的一种情况是：两个函数以相同序列（sequence）执行大致相近的措施，但是各措施不完全相同。这种情况下我们可以将「执行各措施」的序列移至 superclass，并倚赖多态（polymorphism）保证各措施仍得以保持差异性。这样的函数被称为 ***Template Method***（模板函数）[Gang of Four]。

作法（Mechanics）

- 在各个 subclass 中分解目标函数，使分解后的各个函数要不完全相同，要不完全不同。
- 运用 *Pull Up Method*（322）将各 subclass 内完全相同的函数上移至 superclass。
- 对于那些（剩余的、存在于各 subclasses 内的）完全不同的函数，实施 *Rename Method*（273），使所有这些函数的签名式（signature）完全相同。

  ⇨ 这将使得原函数变为完全相同，因为它们都执行同样一组函数调用；但各 subclass 会以不同方式响应这些调用。

- 修改上述所有签名式后，编译并测试。
- 运用 *Pull Up Method*（322）将所有原函数一一上移至 superclass。在 superclass 中将那些「有所不同、代表各种不同措施」的函数定义为抽象函数。
- 编译，测试。
- 移除其他 subclass 中的原函数，每删除一个，编译并测试。

范例（Example）

现在我将完成第一章遗留的那个范例。在此范例中，我有一个 `Customer`，其中有两个用于打印的函数。`statement()`函数以 ASCII 码打印报表（statement）：

```
public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      //show figures for this rental
```

```
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }
        //add footer lines
        result += "Amount owed is " +
                  String.valueOf(getTotalCharge()) + "\n";
        result += "You earned " +
                  String.valueOf(getTotalFrequentRenterPoints()) +
                  " frequent renter points";
        return result;
    }
```

函数 htmlStatement() 则以 HTML 格式输出报表：

```
    public String htmlStatement() {
        Enumeration rentals = _rentals.elements();
        String result = "<H1>Rentals for <EM>" + getName() +
                        "</EM></H1><P>\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for each rental
            result += each.getMovie().getTitle()+ ": " +
                      String.valueOf(each.getCharge()) + "<BR>\n";
        }
        //add footer lines
        result += "<P>You owe <EM>" +
                  String.valueOf(getTotalCharge()) +
                  "</EM><P>\n";
        result += "On this rental you earned <EM>" +
                  String.valueOf(getTotalFrequentRenterPoints()) +
                  "</EM> frequent renter points<P>";
        return result;
    }
```

使用 *Form Template Method*（345）之前，我需要对上述两个函数做一些整理，使它们成为「某个共同 superclass」下的 subclass 函数。为了这一目的，我使用函数对象（method object）[Beck] 针对「报表打印工作」创建一个「独立的策略继承体系」（separate strategy hierarchy），如图 11.1。

```
class Statement {}
class TextStatement extends Statement {}
class HtmlStatement extends Statement {}
```

现在，通过 *Move Method*（142），我将两个负责输出报表的函数分别搬移到对应的 subclass 中：

```
class Customer...
    public String statement() {
        return new TextStatement().value(this);
    }
    public String htmlStatement() {
        return new HtmlStatement().value(this);
    }
```

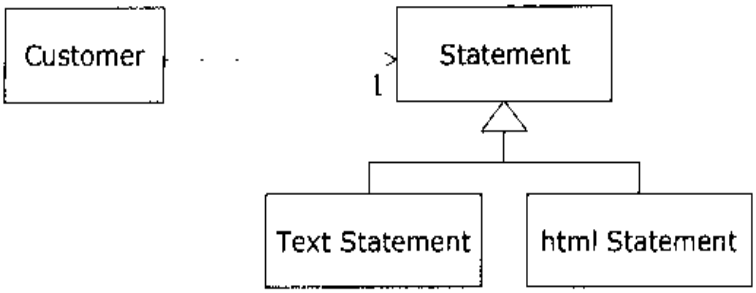


图 11.1 针对「报表输出」使用 ***Strategy*** 模式

```
class TextStatement {
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = "Rental Record for " + aCustomer.getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }
        //add footer lines
        result += "Amount owed is "+String.valueOf(aCustomer.getTotalCharge())+"\n";
        result += "You earned " +
                  String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
                  " frequent renter points";
        return result;
    }

class HtmlStatement {
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = "<H1>Rentals for <EM>" + aCustomer.getName() +
                        "</EM></H1><P>\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for each rental
            result += each.getMovie().getTitle()+ ": " +
                      String.valueOf(each.getCharge()) + "<BR>\n";
        }
        //add footer lines
        result += "<P>You owe <EM>" +
                  String.valueOf(aCustomer.getTotalCharge()) + "</EM><P>\n";
        result += "On this rental you earned <EM>"
                + String.valueOf(
            aCustomer.getTotalFrequentRenterPoints()) +
            "</EM> frequent renter points<P>";
        return result;
    }
```

搬移之后，我还对这两个函数的名称做了一些修改，使它们更好地适应 ***Strategy*** 模式的要求。我之所以为它们取相同名称，因为两者之间的差异不在于函数，而

在于函数所属的 class。如果你想试着编译这段代码，还必须在 Customer class 中添加一个 getRentals() 函数，并放宽 getTotalCharge() 函数和 getTotalFrequentRenterPoints()函数的可视性（visibility）。

面对两个 subclass 中的相似函数，我可以开始实施 *Form Template Method*（345）了。本重构的关键在于：运用 *Extract Method*（110）将两个函数的不同部分提炼出来，从而将相像的代码（similar code）和变动的代码（varying code）分开。每次提炼后，我就建立一个签名式（signature）相同但本体（bodies）不同的函数。

第一个例子就是打印报表表头（headers）。上述两个函数都通过 Customer 对象获取信息，但对运算结果（字符串）的格式化方式不同。我可以将「对字符串的格式化动作」提炼到独立函数中，并将提炼所得命以相同的签名式（signature）：

```
class TextStatement...
    String headerString(Customer aCustomer) {
       return "Rental Record for " + aCustomer.getName() + "\n";
    }
    public String value(Customer aCustomer) {
       Enumeration rentals = aCustomer.getRentals();
       String result =headerString(aCustomer);
       while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                        String.valueOf(each.getCharge()) + "\n";
       }
       //add footer lines
       result += "Amount owed is " +
                   String.valueOf(aCustomer.getTotalCharge()) + "\n";
       result += "You earned " +
                  String.valueOf(
                        aCustomer.getTotalFrequentRenterPoints()) +
                  " frequent renter points";
       return result;
    }

class HtmlStatement...
    String headerString(Customer aCustomer) {
       return "<H1>Rentals for <EM>" + aCustomer.getName() +
                "</EM></H1><P>\n";
    }
    public String value(Customer aCustomer) {
       Enumeration rentals = aCustomer.getRentals();
       String result = headerString(aCustomer);
       while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for each rental
            result += each.getMovie().getTitle()+ ": " +
                        String.valueOf(each.getCharge()) + "<BR>\n";
       }
```

```
        //add footer lines
        result += "<P>You owe <EM>" +
                  String.valueOf(aCustomer.getTotalCharge()) + "</EM><P>\n";
        result += "On this rental you earned <EM>" +
                  String.valueOf(
                        aCustomer.getTotalFrequentRenterPoints()) +
                  "</EM> frequent renter points<P>";
        return result;
    }
```

编译并测试，然后继续处理其他元素。我将逐一对各个元素进行上述过程。下面是整个重构完成后的结果：

```
class TextStatement...
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString(each);
        }
        result += footerString(aCustomer);
        return result;
    }
    String eachRentalString (Rental aRental) {
        return "\t" + aRental.getMovie().getTitle()+ "\t" +
                String.valueOf(aRental.getCharge()) + "\n";
    }
    String footerString (Customer aCustomer) {
        return "Amount owed is " +
                String.valueOf(aCustomer.getTotalCharge()) +
                "\n" + "You earned " +
                String.valueOf(
                    aCustomer.getTotalFrequentRenterPoints()) +
                " frequent renter points";
    }

class HtmlStatement...
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString(each);
        }
        result += footerString(aCustomer);
        return result;
    }
    String eachRentalString (Rental aRental) {
        return aRental.getMovie().getTitle()+ ": " +
                String.valueOf(aRental.getCharge()) + "<BR>\n";
    }
    String footerString (Customer aCustomer) {
        return "<P>You owe <EM>" +
```

```
                String.valueOf(aCustomer.getTotalCharge()) +
                "</EM><P>" +
                "On this rental you earned <EM>" +
                String.valueOf(
                    aCustomer.getTotalFrequentRenterPoints()) +
                "</EM> frequent renter points<P>";
    }
```

所有这些修改都完成后，两个 value() 函数看上去已经非常相似了，因此我可以使用 *Pull Up Method*（322）将它们提升到 superclass 中。提升完毕后，我需要在 superclass 中把 subclass 函数声明为抽象函数。

```
class Statement...
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString(each);
        }
        result += footerString(aCustomer);
            return result;
        }
    abstract String headerString(Customer aCustomer);
    abstract String eachRentalString (Rental aRental);
    abstract String footerString (Customer aCustomer);
```

然后我把 TextStatement.value() 函数拿掉，编译并测试。完成之后再把 HtmlStatement.value() 也删掉，再次编译并测试。最后结果如图 11.2。

完成本重构后，处理其他种类的报表就容易多了：你只需为 Statement 再建一个 subclass，并在其中覆写（overrides）三个抽象函数即可。

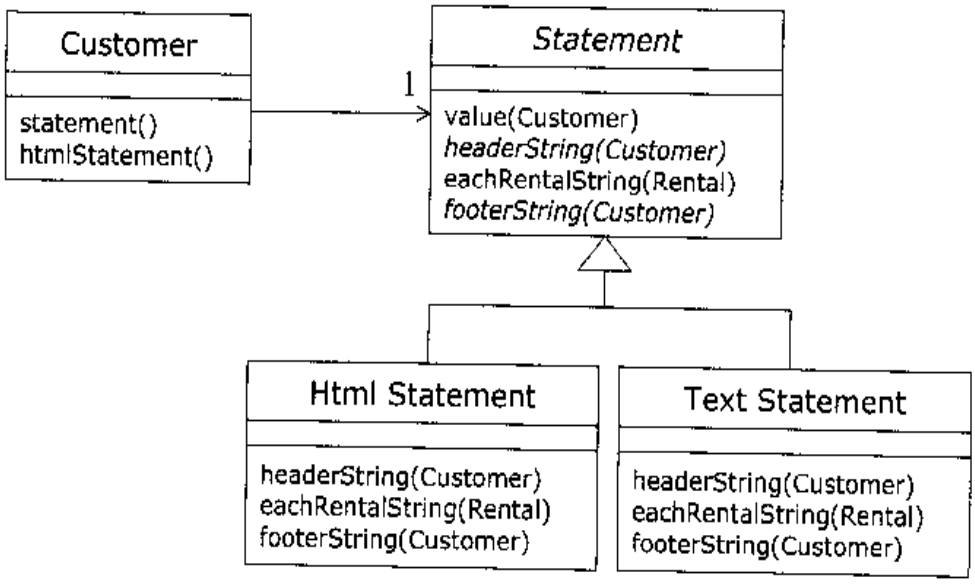
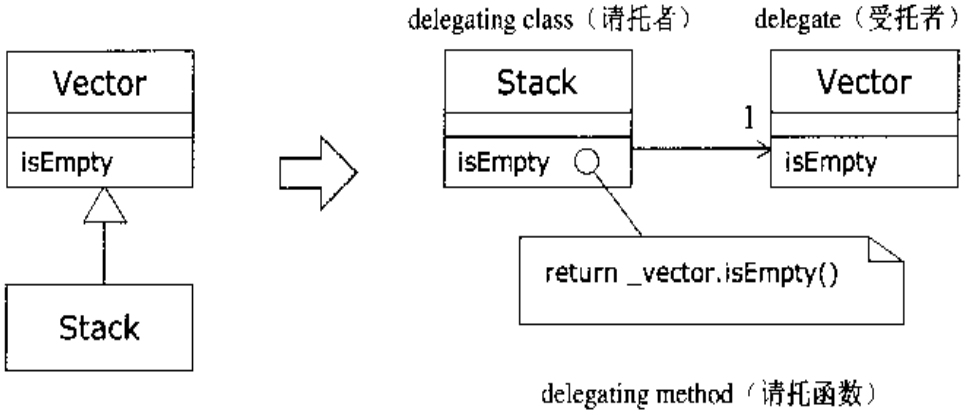


图 11.2 ***Template Method***（模板函数）塑造完毕后的 classes

11.11 Replace Inheritance with Delegation（以委托取代继承）

某个 subclass 只使用 superclass 接口中的一部分，或是根本不需要继承而来的数据。

在 subclass 中新建一个值域用以保存 superclass；调整 subclass 函数，令它改而委托 superclass；然后去掉两者之间的继承关系。



动机（Motivation）

继承（inheritance）是一件很棒的事，但有时候它并不是你要的。常常你会遇到这样的情况：一开始你继承了一个 class，随后发现 superclass 中的许多操作并不真正适用于 subclass。这种情况下你所拥有的接口并未真正反映出 class 的功能。或者，你可能发现你从 superclass 中继承了一大堆 subclass 并不需要的数据，抑或者你可能发现 superclass 中的某些 protected 函数对 subclass 并没有什么意义。

你可以选择容忍，并接受传统说法：subclass 可以只使用 superclass 功能的一部分。但这样做的结果是：代码传达的信息与你的意图南辕北辙——这是一种混淆，你应该将它去除。

如果以委托（delegation）取代继承（inheritance），你可以更清楚地表明：你只需要受托类（delegated class）的一部分功能。接口中的哪一部分应该被使用，哪一部分应该被忽略，完全由你主导控制。这样做的成本则是需要额外写出请托函数（delegating methods），但这些函数都非常简单，极少可能出错。

作法（Mechanics）

- 在 subclass 中新建一个值域，使其引用（指向、指涉、refers）superclass 的一个实体，并将它初始化为 this。
- 修改 subclass 内的每一个（可能）函数，让它们不再使用 superclass，转而使用上述那个「受托值域」（delegate field）。每次修改后，编译并测试。

  ⇨ 你不能如此这般地修改 subclass 中「通过 super 调用 superclass 函数」的函数，否则它们会陷入无限递归（infinite recurse）。这一类函数只有在继承关系被打破后才能修改。

- 去除两个 classes 之间的继承关系，将上述「受托值域」（delegate field）的赋值动作修改为「赋予一个新对象」。
- 针对客户端所用的每一个 superclass 函数，为它添加一个简单的请托函数（delegating method）。
- 编译，测试。

范例（Example）

「滥用继承」的一个经典范例就是让 Stack class 继承 Vector class。Java 1.1 的 utility library（java.util）恰好就是这样做的。（这些淘气的孩子啊！）不过，作为范例，我只给出一个比较简单的形式：

```
class MyStack extends Vector {

    public void push(Object element) {
       insertElementAt(element,0);
    }

    public Object pop() {
       Object result = firstElement();
       removeElementAt(0);
       return result;
    }
}
```

只要看看 MyStack 的用户，我就会发现，用户只要它做四件事：push()、pop()、size()和 isEmpty()。后两个函数是从 Vector 继承来的。

我要把这里的继承关系改为委托关系。首先，我要在 MyStack 中新建一个值域，用以保存「受托之 Vector 对象」。一开始我把这个值域初始化为 this，这样在重构进行过程中，我就可以同时使用继承和委托：

```
private Vector _vector = this;
```

现在，我开始修改 MyStack 的函数，让它们使用委托关系。首先从 push() 开始：

```
public void push(Object element) {
    _vector.insertElementAt(element,0);
}
```

此时我可以编译并测试，一切都将运转如常。现在轮到 pop()：

```
public Object pop() {
    Object result = _vector.firstElement();
    _vector.removeElementAt(0);
    return result;
}
```

修改完所有 subclass 函数后，我可以打破与 superclass 之间的联系了：

```
class MyStack ~~extends Vector~~
    private Vector _vector = new Vector();
```

然后，对于 Stack 客户端可能用到的每一个 Vector 函数（译注：这些函数原本是继承而来的），我都必须在 MyStack 中添加一个简单的请托函数（delegating method）：

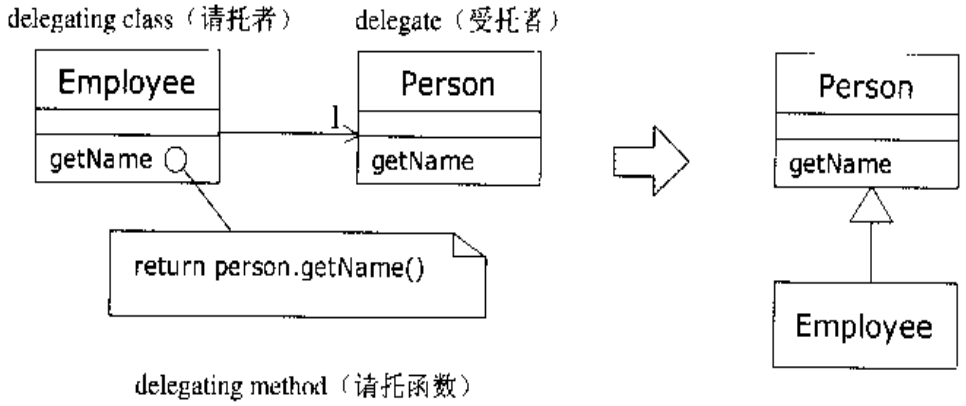
```
public int size() {
    return _vector.size();
}
public boolean isEmpty() {
    return _vector.isEmpty();
}
```

现在我可以编译并测试。如果我忘记加入某个请托函数，编译器会告诉我。

11.12 Replace Delegation with Inheritance（以继承取代委托）

你在两个 classes 之间使用委托关系（delegation），并经常为整个接口编写许多极简单的请托函数（delegating methods）。

让「请托（delegating）class」继承「受托 class（delegate）」。



动机（Motivation）

本重构与 *Replace Inheritance with Delegation*（352）恰恰相反。如果你发现自己需要使用「受托 class」中的所有函数，并且费了很大力气编写所有极简的请托函数（delegating methods），本重构可以帮助你轻松回头使用「继承」。

两条告诫需牢记于心。首先，如果你并没有使用「受托 class」的所有函数（而非只是部分函数），那么就不应该使用 *Replace Delegation with Inheritance*（355），因为 subclass 应该总是遵循（奉行）superclass 的接口。如果过多的请托函数让你烦心，你有别的选择：你可以通过 *Remove Middle Man*（160）让客户端自己调用受托函数，也可以使用 *Extract Superclass* 将两个 classes 接口相同的部分提炼到 superclass 中，然后让两个 classes 都继承这个新的 superclass；你还可以以类似手法使用 *Extract Interface*（341）。

另一种需要当心的情况是：受托对象被不止一个其他对象共享，而且受托对象是可变的（mutable）。在这种情况下，你就不能将「委托关系」替换为「继承关系」，因为这样就无法再共享数据了。数据共享是必须由「委托关系」承担的一种责任，你无法把它转给「继承关系」。如果受托对象是不可变的（immutable），数据共享就不成问题，因为你大可放心地拷贝对象，谁都不会知道。

作法（Mechanics）

- 让「请托端」成为「受托端」的一个 subclass。
- 编译。
  - ⇨ 此时，某些函数可能会发生冲突：它们可能有相同的名称，但在返回型别（return type）、异常指定（exceptions）或可视性（visibility）方面有所差异。你可以使用 *Rename Method*（273）解决此类问题。
- 将「受托值域」（delegate field）设为「该值域所处之对象自身」。
- 去掉简单的诸托函数（delegating methods）。
- 编译并测试。
- 将所有其他「涉及委托关系」的动作，改为「调用对象自身（继承而来的函数）」。
- 移除「受托值域」（delegate field）。

范例（Example）

下面是一个简单的 `Employee`，将一些函数委托给另一个同样简单的 `Person`：

```
class Employee {
    Person _person = new Person();
    public String getName() {
      return _person.getName();
    }
    public void setName(String arg) {
      _person.setName(arg);
    }
    public String toString () {
      return "Emp: " + _person.getLastName();
    }
}
class Person {
    String _name;
    public String getName() {
      return _name;
    }
    public void setName(String arg) {
      _name = arg;
    }
    public String getLastName() {
      return _name.substring(_name.lastIndexOf(' ')+1);
    }
}
```

第一步，只需声明两者之间的继承关系：

```
class Employee extends Person
```

此时，如果有任何函数发生冲突，编译器会提醒我。如果某几个函数的名称相同、但返回型别不同，或抛出不同的异常，它们之间就会出现冲突。所有此类问题都可以通过 *Rename Method*（273）加以解决。为求简化，我没有在范例中列出这些麻烦情况。

下一步要将「受托值域」（delegate field）设值为「该值域所处之对象自身」。同时，我必须先删掉所有简单的请托函数（例如 `getName()`和 `setName()`）。如果留下这种函数，就会因为无限递归而引起系统的 call stack 满溢（overflow）。在此范例中，我应该把 **Employee** 的 `getName()`和 `setName()`拿掉。

一旦 **Employee** 可以正常工作了，我就修改其中「使用了请托函数（译注：或受托值域）」的函数，让它们直接调用「从 superclass 继承而来的函数」：

```
public String toString () {
    return "Emp: " + getLastName();
}
```

摆脱所有涉及委托关系的函数后，我也就可以摆脱 `_person` 这个（受托）值域了。

12

大型重构

Big Refactoring

by Kent Beck and Martin Fowler

前面的章节已经向读者展示了各个单项重构步骤。目前还缺乏的是对整个「游戏」的完整概念。你之所以进行重构，必定是为了达到某个目的，而不仅仅是为了看起来有所动作（起码大多数时候你的重构是为了达到某个目的）。那么，整个「游戏」看起来又是怎样的呢？

这场游戏的本质（nature）

以下面介绍的重构手法中，你肯定会注意到一件事：重构步骤的描述，不再如前面那么仔细。这是因为在大型重构中，情况有很多变化，我们无法告诉你准确的重构步骤；如果没有看到实际情况，任谁都无法确切知道该怎么做。当你为某个函数添加参数时，作法可以很仔细而清楚，因为重构范围（作用域）很清楚。但是当你分解一个继承体系时，由于每个继承体系都是不同的，所以我们无法告诉你确切的重构步骤。

另外，对于这些大型重构，还有一件事需要注意：它们会耗费相当长的时间。第 6 章至第 11 章所介绍的重构手法，都可以在数分钟（至多一个小时）内完成，但是我们曾经进行过的一些大型重构，却需要数月甚至数年的时间。如果你需要给一个运行中的系统添加功能，你不可能说服经理把系统停止运行两个月让你进行重构；你只能一点一点地做你的工作，今天一点点，明天一点点。

在这个过程中，你应该根据需要安排自己的工作，只在需要添加新功能或修补错误时才进行重构。你不必一开始就完成整个系统的重构；重构程度只要能满足其他任务的需要就行了。反正明天你还可以回来重构。

本章范例也反映出这样的哲学。如果要向你展示本书中所有的重构，轻易就能耗去上百页篇幅。我们很清楚这一点，因为Martin的确尝试过。所以，我们把范例压缩至「数张概略图」的尺度。

由于大型重构可能需要花费相当长的时间，因此它们并不像其他章节介绍的重构那样，能够立刻让人满意。你必须有那么一点小小的信仰：你每天都在使你自己的程序世界更安全。

进行大规模重构时，有必要为整个开发团队建立共识；这是小型重构所不需要的。大型重构为许许多多的修改指定了方向。整个团队都必须意识到：有一个大型重构正在进行，每个人都应该相应地安排自己的行动。说到这里，我想给大家讲个故事。两个家伙的车子在山顶附近抛锚了，于是他俩走下车，一人走到车的一头，开始推车。经过毫无成果的半小时之后，车头那家伙开口说道：『我从来不知道把车推下山这么难！』另一个家伙答道：『嘿，你说「推下山」是什么意思？难道我们不是想把车推上山吗？』我猜你一定不想让这个故事在你的开发团队中重演，对吧！

大型重构的重要性

我们已经看到，使那些小型重构突显价值的质量（可预测的结果、可观察的过程、立竿见影的满足等等），在大型重构中往往并不存在。既然如此，为什么大型重构还那么重要，以至于我们想要把它们放进本书？那是因为如果没有它们，我们就可能面临这样的风险：投入了大把时间学习重构，在实际工作中却无法获得实在的利益。这对我们来说是非常糟糕的，我们不能容忍这种事情发生。

更重要的是，你之所以需要重构，决不会是因为它很好玩，而是因为你希望它能对你的程序有所帮助，让你能够做一些重构之前无法做的事情。

正如水草会堵塞河道一样，在一知半解的情况下做出的设计决策，一旦堆积起来，也会使你的程序陷于瘫痪。通过重构，你可以保证随时在程序中反映出自己对于「应该如何设计程序」的完整理解。正如水草会迅速蔓延一样，对系统理解不够完整的设计决策，也会很快地将它们的影响蔓延到整个程序中。要根除这种错误，一个、两个、甚至十个单独的行为都是不够的，只有持续而无处不在的重构才有可能竟其功。

四个大型重构

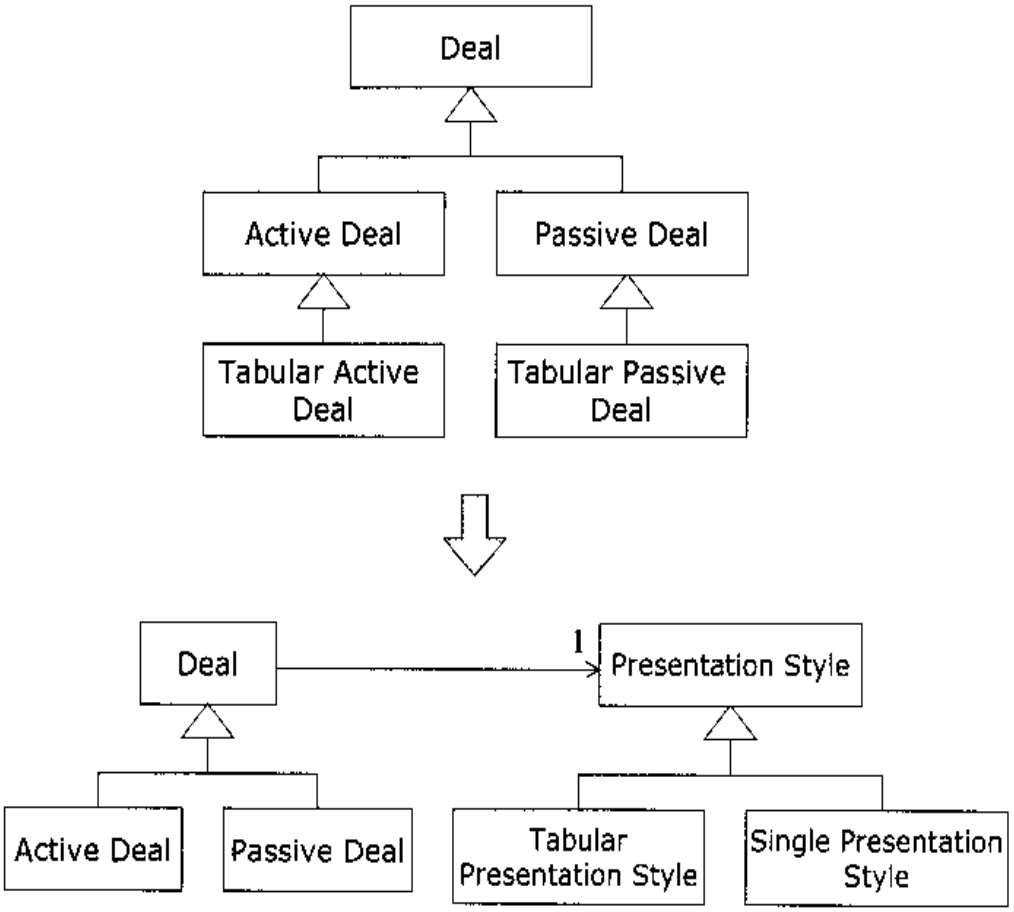
本章之中，我们将介绍四个大型重构实例。这些仅仅是例子，我们并没有打算覆盖所有领域。迄今为止，绝大多数关于重构的研究和实践都集中于比较小的重构手法上，以这种方式谈论大型重构，是一种非常新鲜的作法，这主要来自于 Kent 的经验。在大规模重构方面，Kent 的经验比其他所有人都要丰富。

Tease Apart Inheritance（362）用于处理混乱的继承体系——这种继承体系往往以一种令人迷惑的方式组合了数个不同方面的变化（variations）。*Convert Procedural Design to Objects*（368）可以帮助你解决一个「古典」问题：如何处理程序性代码（procedural code）？许多使用面向对象语言的程序员，其实并没有真正理解面向对象技术，因此你常会需要使用这项重构。如果你看到以传统的双层结构（two-tier，用户界面和数据库）方式编写的代码，你可能需要使用 *Separate Domain from Presentation*（370）将业务逻辑（business logic）与用户界面（user interface）隔离开来。经验丰富的面向对象开发人员发现：对于一个长时间、大负荷运转的系统来说，这样的分离是至关重要的。*Extract Hierarchy*（375）则可以将过于复杂的 class 转变为一群 subclasses，从而简化系统。

12.1 Tease Apart Inheritance（梳理并分解继承体系）

某个继承体系（inheritance hierarchy）同时承担两项责任。

建立两个继承体系，并通过委托关系（delegation）让其中一个可以调用另一个。



动机（Motivation）

继承是个好东西，它使你得以在 subclass 中写出明显「压缩过」（compressed）的代码。函数的重要性可能并不和它的大小成比例——在继承体系之中尤然。

不过，先别急着为这个强大的工具欢呼雀跃，因为继承也很容易被误用，并且这种误用还很容易在开发人员之间蔓延。今天你为了一项小小任务而加入一个小小的subclass，明天又为同样任务在继承体系的另一个地方加入另一个 subclass。一个星期（或者一个月或者一年）之后，你就会发现自己身陷泥淖，而且连一根拐杖都没有。

混乱的继承体系是一个严重的问题，因为它会导致重复代码，而后者正是程序员生涯的致命毒药。它还会使修改变得困难，因为「特定种类」的问题的解决策略被分散到了整个继承体系。最终，你的代码将非常难以理解。你无法简单地说：『这就是我的继承体系，它能计算结果』，而必须说：『它会计算出结果……呃，这些是用以表现不同表格形式的 subclasses，每个 subclass 又有一些 subclasses 针对不同的国家。』

要指出「某个继承体系承担了两项不同的责任」并不困难：如果继承体系中的某一特定层级上的所有 classes，其 subclass 名称都以相同的形容词开始，那么这个体系很可能就是承担着两项不同的责任。

作法（Mechanics）

- 首先识别出继承体系所承担的不同责任，然后建立一个二维表格（或者三维乃至四维表格，如果你的继承体系够混乱而你的绘图工具够酷的话），并以坐标轴标示出不同的任务。我们将重复运用本重构，处理两个或两个以上的维度（当然，每次只处理一个维度）。
- 判断哪一项责任更重要些，并准备将它留在当前的继承体系中。准备将另一项责任移到另一个继承体系中。
- 使用 *Extract Class*（149）从当前的 superclass 提炼出一个新 class，用以表示重要性稍低的责任，并在原 superclass 中添加一个 *instance* 变量（不是 *static* 变量），用以保存新建 class 的实体。
- 对应于原继承体系中的每个 subclass，创建上述新 class 的一个个 subclasses。在原继承体系的 subclasses 中，将前一步骤所添加的 *instance* 变量初始化为新建 subclass 的实体。
- 针对原继承体系中的每个 subclass，使用 *Move Method*（142）将其中的行为搬移到与之对应的新建 subclass 中。
- 当原继承体系中的某个 subclass 不再有任何代码时，就将它去除。
- 重复以上步骤，直到原继承体系中的所有 subclass 都被处理过为止。观察新继承体系，看看是否有可能对它实施其他重构手法，例如 *Pull Up Method*（322）或 *Pull Up Field*（320）。

范例（Examples）

让我们来看一个混乱的继承体系（如图 12.1）。

这个继承体系之所以混乱，因为一开始 Deal class 只被用来显示单笔交易。后来，某个人突发奇想地用它来显示一张交易表格。只需飞快建立一个 ActiveDeal subclass 再加上一点点经验，不必做太多工作就可以显示一张表格了。哦，还要「被动交易（PassiveDeal）」表格是吗？没问题，再加一个 subclass 就行了。

两个月过去，表格相关代码变得愈来愈复杂，你却没有一个好地方可以放它们，因为时间太紧了。咳，老戏码！现在你将很难向系统加入新种交易，因为「交易处理逻辑」与「数据显示逻辑」已经「你中有我，我中有你」了。

按照本重构提出的处方笺，第一步工作是识别出这个继承体系所承担的各项责任。这个继承体系的职责之一是捕捉不同交易种类间的变化（差异），职责之二是捕捉不同显示风格之间的变化（差异）。因此，我们可以得到下列表格：

Deal	Active Deal	Passive Deal
Tabular Deal		

下一步要判断哪一项职责更重要。很明显「交易种类」比「显示风格」重要，因此我们把「交易种类」留在原地，把「显示风格」提炼到另一个继承体系中。不过，实际工作中，我们可能需要将「代码较多」的职责留在原地，这样一来需要搬移的代码数量会比较少。

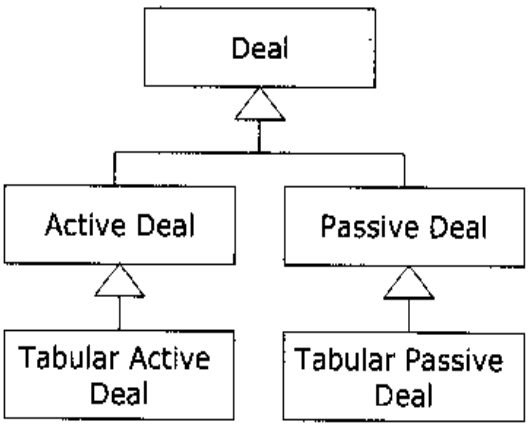


图 12.1 一个混乱的继承体系

然后，我们应该使用 *Extract Class*（149）提炼出一个单独的 `PresentationStyle` class，用以表示「显示风格」（如图 12.2）。

接下来我们需要针对原继承体系中的每个 subclass，建立 `PresentationStyle` 的一个个 subclasses（如图 12.3），并将 `Deal` class 之中用来保存 `PresentationStyle` 实体的那个 *instance* 变量初始化为适当的 subclass 实体：

```
ActiveDeal constructor
...presentation_ new SingleActivePresentationStyle();...
```

你可能会说：「这不是比原先的 classes 数量还多了吗？难道这还能让我的生活更舒服？」生活往往如此：以退为进，走得更远。对一个纠结成团的继承体系来说，被提炼出来的另一个继承体系几乎总是可以再戏剧性地大量简化。不过，比较安全的态度是一次一小步，不要过于躁进。

现在，我们要使用 *Move Method*（142）和 *Move Field*（146），将 `Deal` subclasses 中「与显示逻辑相关」的函数和变量搬移到 `PresentationStyle` 相应的 subclasses 去。我们想不出什么好办法来模拟这个过程，只好请你自己想像。总之，这个步骤完成后，`TabularActiveDeal` 和 `TabularPassiveDeal` 不再有任何代码，因此我们将它们移除（如图 12.4）。

两项职责被分割之后，我们可以分别简化两个继承体系。一旦本重构完成，我们总是能够大大简化被提炼出来的新继承体系，而且通常还可以简化原继承体系。

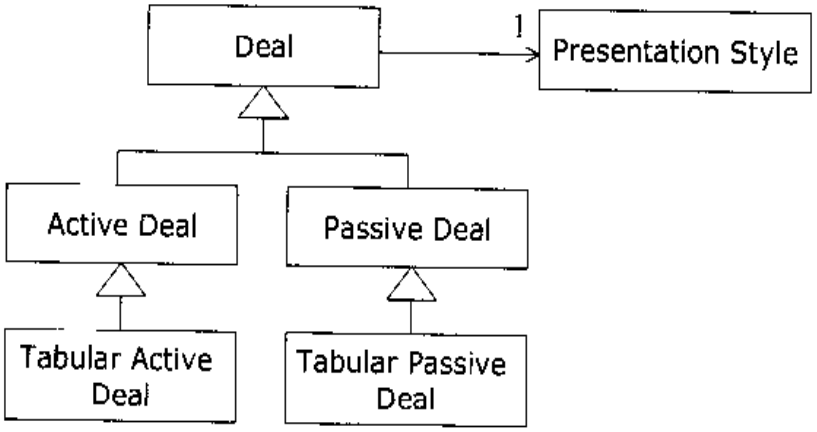


图 12.2 添加 `PresentationStyle`，用以表示「显示风格」

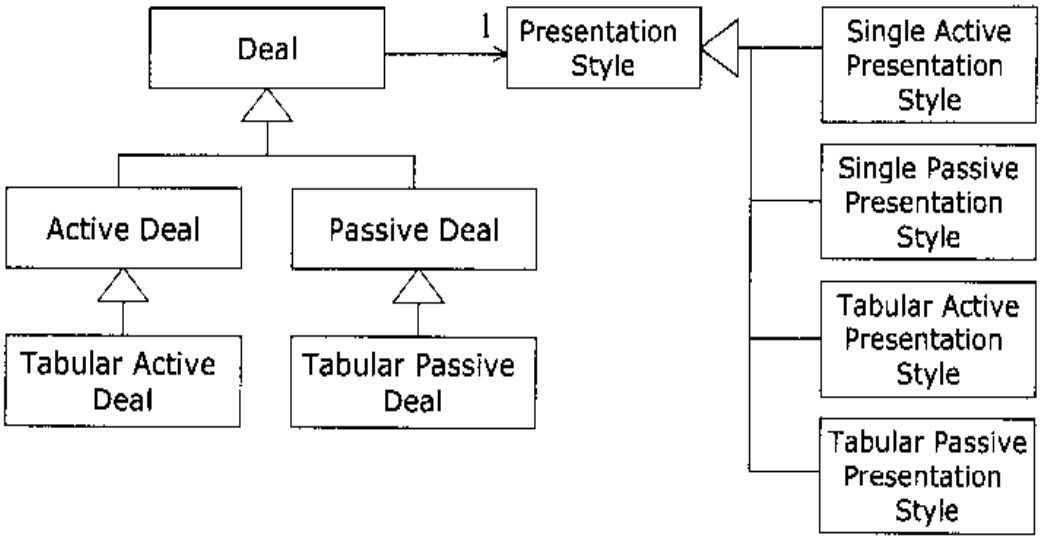


图 12.3 为 PresentationStyle 添加 subclasses

下一步，我们将摆脱「显示风格」中的主动（active）与被动（passive）区别，如图 12.5。

就连「单一显示」和「表格显示」之间的区别，都可以运用若干变量值来捕捉，根本不需要为它们建立 subclasses（如图 12.6）。

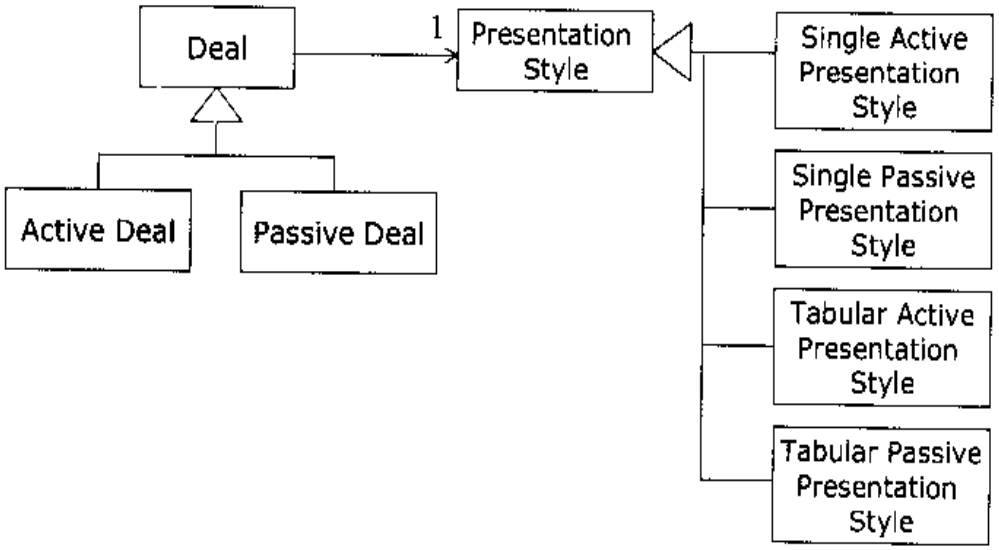


图 12.4 与表格相关的 Deal subclasses 都被移除了

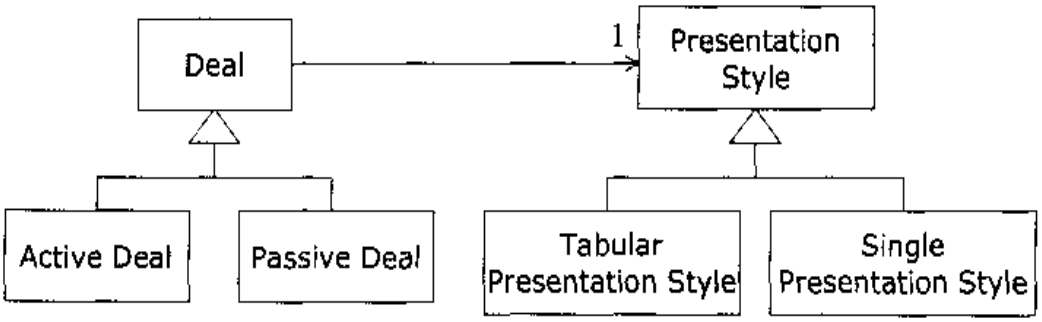


图 12.5 继承体系被分割了

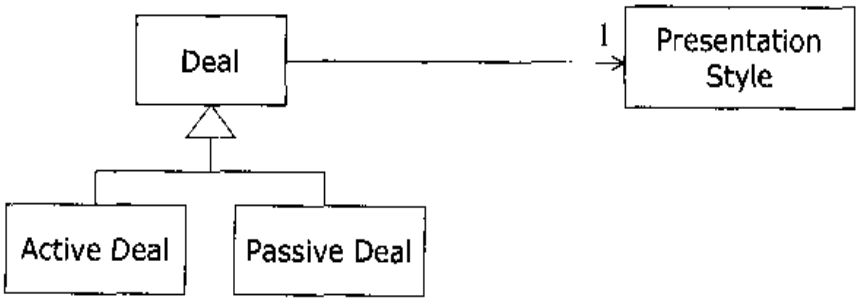
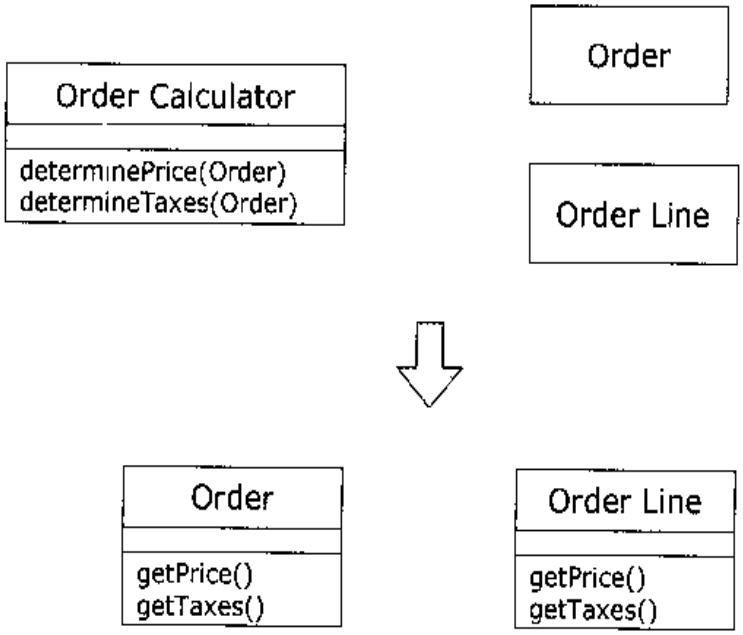


图 12.6 「显示风格」（presentation style）之间的差异可以使用一些变量来表现

12.2 Convert Procedural Design to Objects
将过程化设计转化为对象设计

你手上有一些代码，以传统的过程化风格（procedural style）写就。

将数据记录（data records）[9]变成对象，将行为分开，并将行为移入相关对象之中。



动机（Motivation）

有一次，我们的一位客户，在项目开始时，给开发者提出了两条必须遵守的条件：（1）必须使用Java；（2）不能使用对象。

这让我们忍俊不禁。尽管Java是面向对象语言，「使用对象」可远不仅仅是「调用构造函数」而已。对象的使用也需要花时间去学习。往往你会面对一些过程化风格（procedural style）的代码所带来的问题，并因而希望它们变得更面向对象一些。典型的情况是：class中有着长长的过程化函数和极少的数据，以及所谓的「哑数据对象」——除了数据访问函数（accessors）外没有其他任何函数。如果你要转换的是一个纯粹的过程化程序（procedural program），可能连这些东西都没有。

我们并不是说绝对不应该出现「只有行为而几乎没有数据」的对象。在 ***Strategy***

[9] 译注：这里所谓的记录（record），是指像C `struct` 那样的结构。

模式中，我们常常使用一些小型的 strategy 对象来改变宿主对象的行为，这些小型的 strategy 对象就是「只有行为而没有数据」。但是这样的对象通常比较小，而且只有在我们特别需要灵活性的时候，才会使用它们。

作法（Mechanics）

- 针对每一个记录型别（record type），将它转变为「只含访问函数」的「哑数据对象」（dumb data object）。
  - ⇨ 如果你的数据来自关系式数据库（relational database），就把数据库中的每个表（table）变成一个「哑数据对象」。
- 针对每一处过程化风格，将该处的代码提炼到一个独立 class 中。
  - ⇨ 你可以把提炼所得的 class 做成一个 ***Singleton***（单件；为了方便重新初始化），或是把提炼所得的函数声明为 static。
- 针对每一个长长的程序（procedure），实施 *Extract Method*（110）及其他相关重构，将它分解。再以 *Move Method*（142）将分解后的函数分别移到它所相关的哑数据类（dumb data class）中。
- 重复上述步骤，直到原始 class 中的所有函数都被移除。如果原始 class 是一个完全过程化（purely procedural）的 class，将它拿掉将大快人心。

范例（Example）

第 1 章的范例很好地展示了 *Convert Procedural Design to Objects*（368），尤其是第一阶段（对 `statement()` 函数的分解和安置）。完成这项重构之后，你就拥有了一个「聪明的」数据对象，可以对它进行其他种重构了。

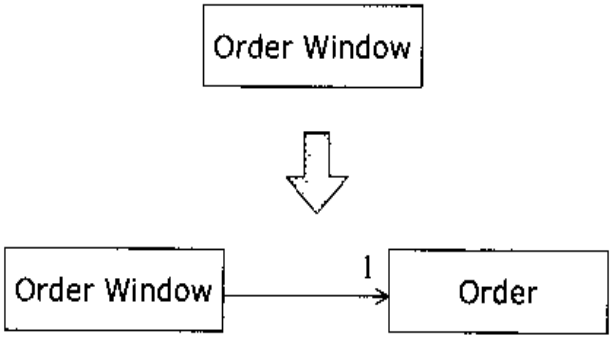
12.3 Separate Domain from Presentation

将领域和表述/显示分离

（译注：本节保留 domain-，presentation-logic，UI，class，object 等英文词）

某些 GUI classes 之中包含了 domain logic（领域逻辑）。

将 domain logic（领域逻辑）分离出来，为它们建立独立的 domain classes。



动机（Motivation）

提到面向对象，就不能不提 ***Model-View-Controller***（***MVC***，模型-视图-控制器）模式。在 Smalltalk-80 环境中，人们以此模式维护 GUI（图形用户界面）和 domain object（领域对象）间的关系。

MVC 模式的最核心价值在于：它将用户界面代码（即所谓 **view**，视图；亦即现今常说的 **presentation**，表述）和领域逻辑（即所谓 **model**，模型）分离了。presentation class 只含用以处理用户界面的逻辑；domain class 不含任何与程序外观相关的代码，只含业务逻辑（business logic）相关代码。将程序中这两块复杂的部分加以分离，程序未来的修改将变得更加容易，同时也使同一业务逻辑（business logic）的多种表述（显示）方式成为可能。那些熟稔面向对象技术的程序员会毫不犹豫地在他们的程序中进行这种分离，并且这种作法也的确证实了它自身的价值。

但是，大多数人并没有在设计中采用这种方式来处理 GUI。大多数带有 client-server GUIs 的环境都采用双层（two-tier）逻辑设计：数据保存在数据库中，业务逻辑（business logic）放在 presentation class 中。这样的环境往往迫使你也倾向这种风格的设计，使你很难把业务逻辑放在其他地方。

Java 是一个真正意义上的面向对象环境，因此你可以创建内含业务逻辑的非视觉性领域对象（nonvisual domain objects）。但你却还是会经常遇到上述双层风格写就的程序。

作法（Mechanics）

- 为每个窗口（window）建立一个 domain class。
- 如果窗口内有一张表格（grid），新建一个 class 来表示其中的行（rows），再以窗口所对应之 domain class 中的一个群集（collection）来容纳所有的 row domain objects。
- 检查窗口中的数据。如果数据只被用于 UI，就把它留着；如果数据被 domain logic 使用，而且不显示于窗口上，我们就以 *Move Field*（146）将它搬移到 domain class 中；如果数据同时被 UI 和 domain logic 使用，就对它实施 *Duplicate Observed Data*（189），使它同时存在于两处，并保持两处之间的同步。
- 检查 presentation class 中的逻辑。实施 *Extract Method*（110）将 presentation logic 从 domain logic 中分开。一旦隔离了 domain logic，再运用 *Move Method*（142）将它移到 domain class。
- 以上步骤完成后，你就拥有了两组彼此分离的 classes：presentation classes 用以处理 GUI，domain classes 内含所有业务逻辑（business logic）。此时的 domain classes 组织可能还不够严谨，更进一步的重构将解决这些问题。

范例（Example）

下面是一个商品订购程序。其 GUI 如图 12.7 所示，其 presentation class 与图 12.8 所示的关系式数据库（relational database）互动。

所有行为（包括 GUI 和定单处理）都由 `Orderwindow` class 处理。

首先建立一个 `Order` class 表示「定单」。然后把 `Order` 和 `Orderwindow` 联系起来，如图 12.9。由于窗口中有一个用以显示定单的表格（grid），所以我们还得建立一个 `OrderLine`，用以表示表格中的每一行（rows）。

我们将从窗口这边而不是从数据库那边开始重构。当然，一开始就把 domain model 建立在数据库基础上，也是一种合理策略，但我们最大的风险源于 presentation logic 和 domain logic 之间的混淆，因此我们首先基于窗口将这些分离出来，然后再考虑对其他地方进行重构。

面对这一类程序，在窗口中寻找内嵌的 SQL（结构化查询语言）语句，会对你有所帮助，因为 SQL 语句获取的数据一定是 domain data。

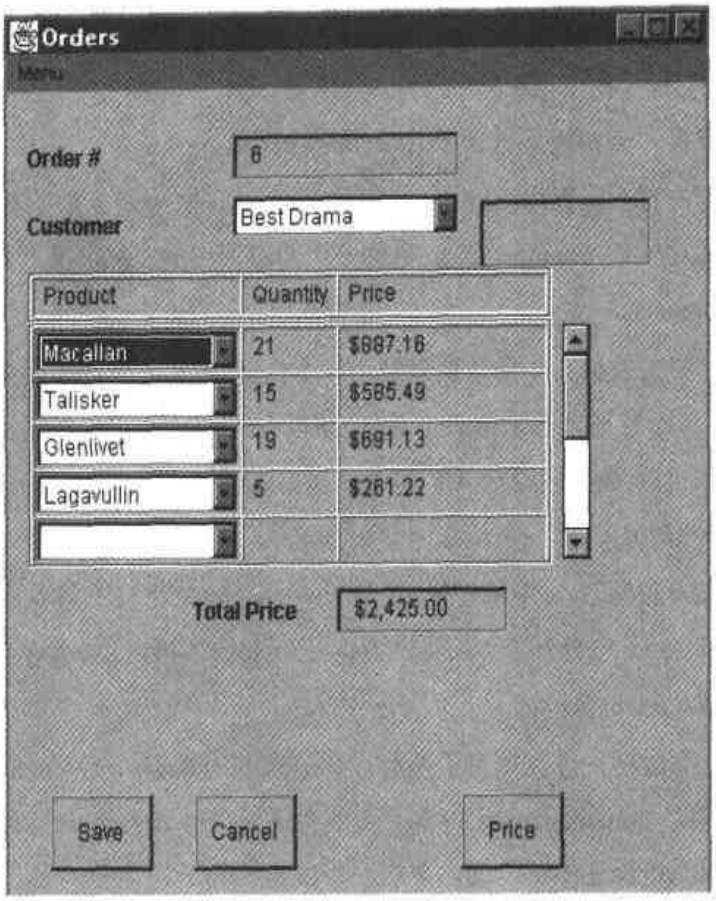


图 12.7 启动程序的用户界面

最容易处理的 domain data 就是那些不直接显示于 GUI 者。本例数据库的 **Customers** table 中有一个 **Codes** 值域，它并不直接显示于 GUI，而是被转换为一个更容易被人理解的短语之后再显示。程序中以简单型别（而非 AWT 组件）如 `String` 保存这个值域值。我们可以安全地使用 *Move Field*（146）将这个值域移到 domain class。

对于其他值域，我们就没有这么幸运了，因为它们内含 AWT 组件，既显示于窗口，也被 domain objects 使用。面对这些值域，我们需要使用 *Duplicate Observed Data*（189），把一个 domain field 放进 `Order` class，又把一个相应的 AWT field 放进 `Orderwindow` class。

这是一个缓慢的过程，但最终我们还是可以把所有 domain logic fields 都搬到 domain class。进行这一步骤时，你可以试着把所有 SQL calls 都移到 domain class，这样你就是同时移动了 database logic 和 domain data。最后，你可以在 `Orderwindow` 中移除 `import java.sql` 之类的语句，这就表示我们的重构告一段落了。在此阶段中你可能需要大量运用 *Extract Method*（110）和 *Move Method*（142）。

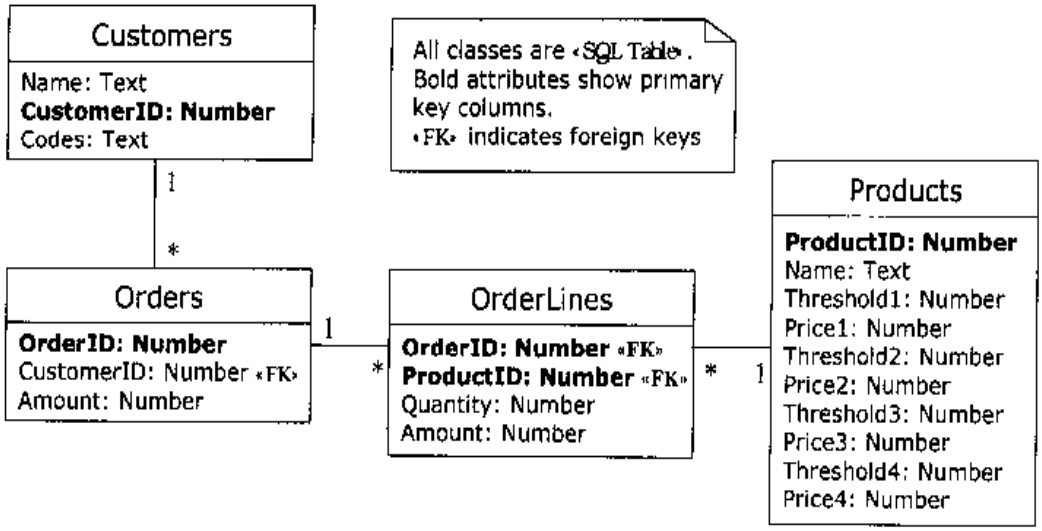


图 12.8 订单程序所用的数据库

* 所有 classes 都是«SQL Table»
粗体字表示主键（primary key），«FK»表示外键（foreign keys）

现在，我们拥有的二个 classes，如图 12.10 所示，它们离「组织良好」还有很大的距离。不过这个模型的确已经很好地分离了 presentation logic 和 domain logic（business logic）。本项重构的进行过程中，你必须时刻留心你的风险来自何方。如果「presentation logic 和 domain logic 混淆」是最大风险，那么就先把它们完全分开，然后才做其他工作；如果其他方面的事情（例如产品定价策略）更重要，那么就先把那一部分的 logic 从窗口提炼出来，并围绕着这个高风险部分进行重构，为它建立合适的结构。反正 domain logic 早晚都必须从窗口移出，如果你在处理高风险部分的重构时会遗留某些 logic 于窗口之中，没关系，就放手去做吧。

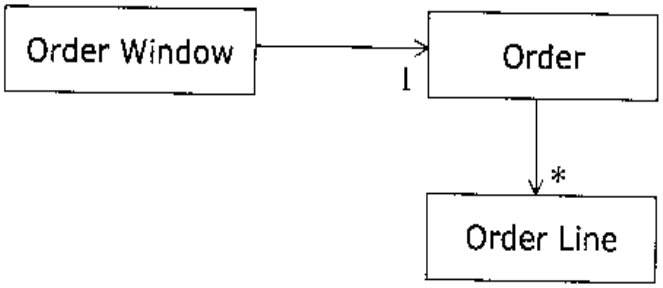


图 12.9 `OrderWindow` class 和 `Order` class

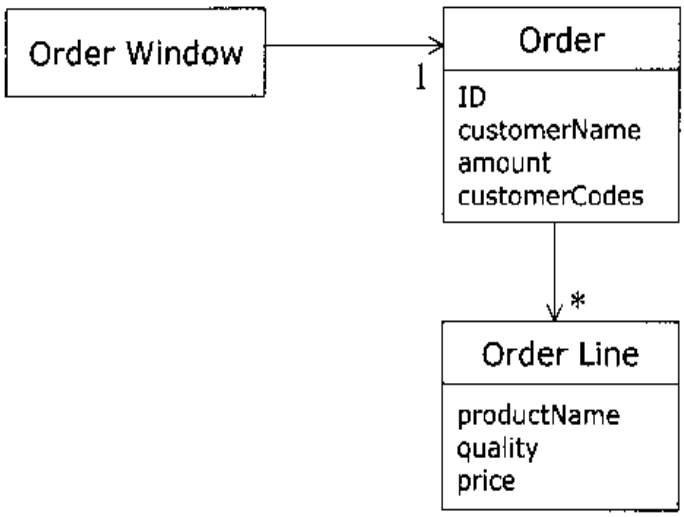
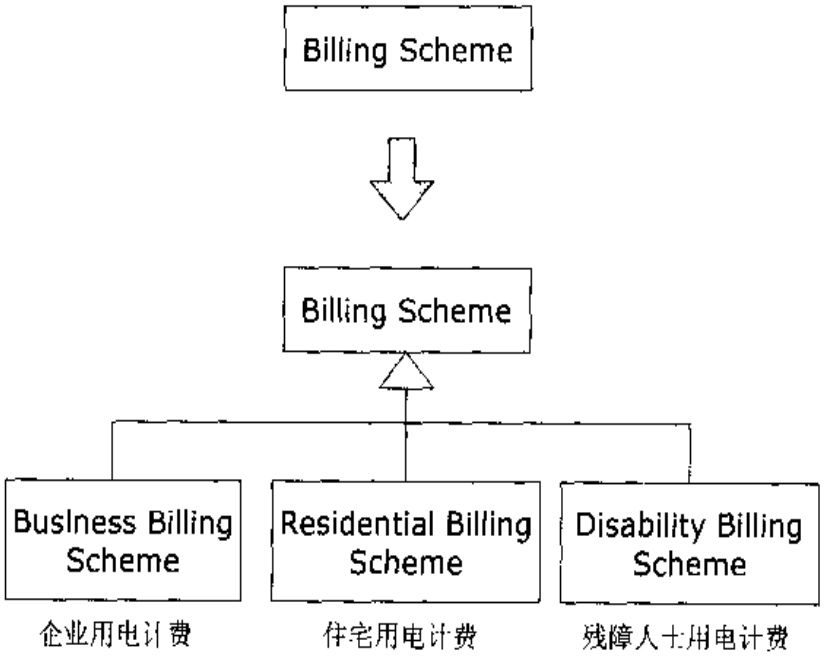


图 12.10 将数据安置（分散）于 domain classes 中

12.4 Extract Hierarchy（提炼继承体系）

你有某个 class 做了太多（过多）工作，其中一部分工作是以大量条件式完成的。

建立继承体系，以一个 subclass 表示一种特殊情况。



动机（Motivation）

在渐进式设计过程中，常常会有这样的情况：一开始设计者只想「以一个 class 实现一个概念」；但随着设计方案的演化，最后却可能「一个 class 实现了两个、三个乃至十个不同的概念」。一开始，你建立了这个简单的 class；数天或数周之后，你可能发现：只要加入一个标记（flag）和一两个测试，就可以在另一个环境下使用这个 class；一个月之后你又发现了另一个这样的机会；一年之后，这个 class 就完全一团糟了：标记变量和条件式遍布各处。

当你遇到这种「瑞士小刀般」的 class，不但能够开瓶开罐、砍小树枝、在演示文稿会上打出激光强调重点……（简直无所不能），你就需要一个好策略（亦即本项重构），将它的各个功能梳理并分开。不过，请注意，只有当条件逻辑（conditional logic）在对象的整个生命期间保持不变，本重构所导入的策略才适用。否则你可能必须在分离各种状况之前先使用 *Extract Class*（149）。

Extract Hierarchy（375）是一项大型重构，如果你一天之内不足以完成它，不要因此失去勇气。将一个极度混乱的设计方案梳理出来，可能需要数周甚至数月的时间。你可以先进行本重构中的一些简易步骤，稍微休息一下，再花数天编写一些明显有生产力的代码。当你领悟到更多东西，再回来继续本项重构的其他步骤——这些步骤将因为你的领悟而显得更加简单明了。

作法（Mechanics）

我们为你准备了两组重构作法。第一种情况是：你无法确定变异（variations）应该是些什么（也就是说你无法确定原始 class 中该有哪些条件逻辑）。这时候你希望每次一小步地前进：

- 鉴别出一种变异（variation）。
  - ⇨ 如果这种变异可能在对象生命期内发生变化，就运用 *Extract Class*（149）将它提炼为一个独立的 class。
- 针对这种变异，新建一个 subclass，并对原始 class 实施 *Replace Constructor with Factory Method*（304）。再修改 ***factory method***，令它返回适当的（相应的）subclass 实体。
- 将含有条件逻辑的函数，一次一个，逐一拷贝到 subclass，然后在明确情况下（注：对 subclass 明确，对 superclass 不明确！），简化这些函数。
  - ⇨ 如有必要隔离函数中的「条件逻辑」和「非条件逻辑」，可对 superclass 实施 *Extract Method*（110）。
- 重复上述过程，将所有变异（variations；特殊情况）都分离出来，直到可以将 superclass 声明为抽象类（abstract class）为止。
- 删除 superclass 中的那些「被所有 subclasses 覆写」的函数（的本体），并将它声明为抽象函数（abstract method）。

如果你非常清楚原始 class 会有哪些变异（variations），可以使用另一种作法：

- 针对原始 class 的每一种变异（variation），建立一个 subclass。
- 使用 *Replace Constructor with Factory Method*（304）将原始 class 的构造函数转变成 ***factory method***，并令它针对每一种变异返回适当的 subclass 实体。
  - ⇨ 如果原始 class 中的各种变异是以 type code 标示，先使用 *Replace Type Code with Subclasses*（223）；如果那些变异在对象生命期间会改变，请使用 *Replace Type Code with State/Strategy*（227）。

- 针对带有条件逻辑的函数，实施 *Replace Conditional with Polymorphism*（255）。如果并非整个函数的行为有所变化，而只是函数一部分有所变化，请先运用 *Extract Method*（110）将变化部分和不变部分隔开来。

范例（Example）

这里所举的例子是「变异并不明朗」的情况。你可以在 *Replace Type Code with Subclasses*（223）、*Replace Type Code with State/Strategy*（227）和 *Replace Conditional with Polymorphism*（255）等重构结果之上，验证「变异已经明朗」的情况下如何使用本项重构。

我们以一个电费计算程序为例。这个程序有两个 classes：表示「消费者」的 `Customer` 和表示「计费方案」的 `BillingScheme`，如图 12.11。

`BillingScheme` 使用大量条件逻辑来计算不同情况（变异）下的费用：冬季和夏季的电价不同，私宅用电、小型企业用电、社会救济（包括残障人士）用电的价格也不同。这些复杂的逻辑导致 `BillingScheme` 变得复杂。

第一个步骤是，提炼出条件逻辑中经常出现的某种变异性。本例之中可能是「视用户是否为残障人士」而发生的变化，用于标示这种情况的可能是 `Customer`、`BillingScheme` 或其他地方的一个标记变量（flag）。

我们针对这种变异建立一个 subclass。为了使用这个 subclass，我们需要确保它被建立并且被使用。因此我们需要处理 `BillingScheme` 构造函数：首先对它实施 *Replace Constructor with Factory Method*（304），然后在所得的 ***factory method*** 中为残障人士增加一个条件子句，使它在适当时候返回一个 `DisabilityBillingScheme` 对象。

然后，我们需要观察 `BillingScheme` 的其他函数，寻找那些随着「用户是否为残障人士」而变化的行为。`CreateBill()` 就是这样一个函数，因此我们将它拷贝到 subclass（图 12.12）。

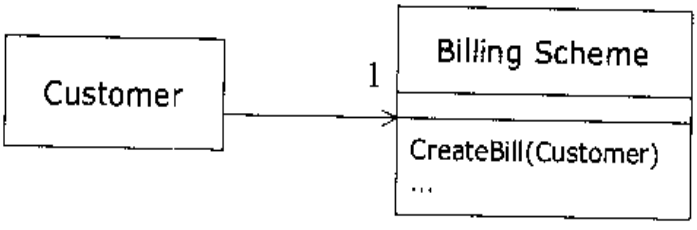


图 12.11 `Customer` 和 `BillingScheme`

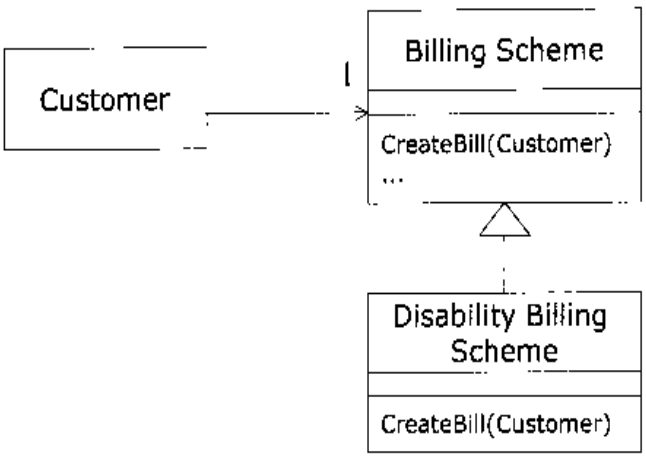


图 12.12 为「残障人士」添加一个 subclass

现在，我们需要检查 subclass 中的 createBill()函数。由于现在我们可以肯定该消费者是残障人士，因此可以简化这个函数。所以下列代码：

```
if (disabilityScheme()) doSomething
```

可以变成：

```
doSomething
```

如果规定在「残障人士用电」和「企业用电」之间只能择一，那么我们的方案就可以避免在 BusinessBillingScheme 中出现任何条件代码。

实施本项重构时，我们希望将「可能变化」和「始终不变」的部分分开，为此我们可以使用 *Extract Method*（110）和 *Decompose Conditional*（238）。本例将对 BillingScheme 各函数实施这两项重构，直到「是否为残障人士」的所有判断都得到了适当处理。然后我们再以相同过程处理他种变异（例如「社会救济用电」）。

然而，当我们处理第二种变异时，我们应该观察「社会救济用电」与「残障人士用电」有何不同。我们希望能够为不同的变异（特殊情况）建立起这般函数：有着相同意图，但针对不同的变异性（特殊情况）采取不同的实际作为（译注：这就是 ***Template Method***）。例如上述两种变异情况下的税额计算可能不同。我们希望确保两个 subclasses 中的相应函数有相同的签名式（signature）。这可能意味我们必须修改 DisabilityBillingScheme，使得以将 subclasses 整理一番。通常我们发现，面对更多变异时，这种「相仿之中略带变化」的函数（***similar and varying methods*** patterns）会使整个系统结构趋于稳定，使我们更容易添加后续更多变异。

13

重构，复用与现实

Refactoring, Reuse, and Reality

by William Opdyke

我和 Martin Fowler 第一次见面，是在温哥华（Vancouver）举行的 OOPSLA 92 大会上。在那之前数个月，我才刚在伊利诺斯大学（University of Illinois）完成关于「面向对象框架之重构」（refactoring object-oriented frameworks）博士论文[1]。当时，我一边考虑继续研究重构，一边也在寻找其他方向，例如医学信息学（medical informatics）。那时 Martin 恰好正在开发一个医学信息应用程序，这便成了我们在温哥华共进早餐时的话题。Martin 在本书最前面也说过，我们用了数分钟时间讨论我对重构的研究。当时他对这个题目的兴趣有限。但是正如你现在看到的，他的兴趣已经大大增加了。

乍见之下，重构很像是从理论研究实验室中诞生的。事实上它最初出现于软件开发者阵营之中。在那儿，面向对象程序员以及 Smalltalk 用户，迫切需要一种技术能够更好地「支持框架开发过程」（或更一般性地「支持变化过程」）。如今重构的相关衍生研究已经成熟，我们感觉它已经进入了「黄金时期」——更多软件从业人员可以体验重构带来的利益。

当 Martin 给我机会，让我为本书写一章的时候，数种想法就出现在我的脑海中。我可以记述早期的重构研究，当时我和 Ralph Johnson 有着迥然不同的技术背景，但我们走到一起，致力研究「如何支持面向对象软件的变化」。我也可以讨论如何为重构提供自动化支持能力，这也是我的研究领域之一，但是与本书关注焦点相去甚远。我还可以与读者分享自己获得的经验：如何把重构和软件业者（特别是那些开发大型项目的软件业者）的日常关心事务结合起来。

在许多领域，我从重构研究之中获得的许多领悟都很有用，这些领域包括软件技术评估、产品发展策略规划、为电信业开发原型和产品、为产品开发团队提供培训和顾问等等。

最终，我决定把以上许多问题都简单讲一讲。正如本章标题所暗示，许多关于重构的认识都适用于更具普遍意义的问题，例如软件复用、产品开发、平台选择等等。尽管本章的某些部分涉及重构中颇为有趣的理论，但本章关注的焦点，主要还是实际的、现实世界的问题，及其解决方案。

如果你想对重构做更深入的研究，请看本章最后所列的重构相关资源和参考文献。

13.1 现实的检验

决定追求我的博士学位之前，我在贝尔实验室（Bell Labs）工作了一些年头。那几年我主要是在公司的一个电子交换系统开发部门里工作。那些产品用来处理电话呼叫，对可靠性和速度的要求都非常高。公司已经投资数千个人年（staff-year）到这些系统的开发和持续发展上，产品生命周期长达数十年。在这些系统的开发中，大部分成本并不是花在最初版本，而是花在其后对系统不断的修改和调整上。如果能找到一种方法，使这些修改更容易、成本更低，那么公司将从中大大受益。

由于贝尔实验室出资让我攻读博士，所以我希望我的研究领域不仅能满足自己技术上的兴趣，也能与贝尔实验室的实际业务需求有关。20世纪80年代后期，面向对象技术刚刚诞生于研究性实验室里头。当Ralph Johnson提出一个既关注面向对象技术、又关注「变化过程（process of change）和软件进化（software evolution）」之支持技术的研究题目时，我立刻接受了它，以之作为我的博士研究题目。

曾经有人告诉我，很少人能够在完成了自己的博士学业后平静地看待自己的题目。有些人对自己的题目感到极其厌倦，很快转向其他研究；另一些人则保持对原先主题的高度热情。我就属于后面这种人。

当我拿到学位回到贝尔实验室，发生了一件奇怪的事：我周遭几乎没有人像我一样为重构激动不已。

我还清楚记得我在1993年初所做的一个演讲，那是在AT&T贝尔实验室和NCR（那时我们是同一家公司的两个部门）的员工技术交流论坛上。我做了一个45分钟的演讲，主题就是重构。一开始，演讲似乎进行得很顺利，我对这个主题的激情感染了听众。但是演讲结束时，几乎没有人提问。有一位与会者从后排走过来企图学习更多一些，他正要开始做毕业设计，正四处查找研究课题。我很希望看到一些项目开发人员能够表现出「想在工作中应用重构技术」的热情。如果他们真有热情，至少那天他们并没有表现出来。

看起来，人们根本不打算接受它。

关于研究，Ralph Johnson给我上了重要的一课：如果有人（文章读者或是演讲会听众）说「我不懂」或者不打算接受它，那就是我们的失败。我们有责任努力发展自己的思想，并将它清楚表达出来。

其后的两年中，在AT&T贝尔实验室的内部论坛上，在外面的研讨会上，我得到了无数次谈论重构的机会。随着与一线开发人员的交谈愈来愈多，我开始明白为什么以前的演讲不能感染别人。我与听众的距离有一部分是因为面向对象技术自身就很新。那些使用它工作的人多半都还没有完成第一个版本的开发，所以还没有遇到「演化（evolution）」这个大问题，而这个问题是重构能够帮忙解决的。这是研究人员的典型尴尬处境——技术的发展超前于实践。但是，造成这种距离，还有另一个讨厌的原因。有一些常识性原因影响了开发者，所以即使他们了解重构的好处，也不情愿对自己的程序进行重构。如果要让重构得到开发者的拥抱，首先必须解决这些问题。

13.2 为什么开发者不愿意重构他们的程序？

假设你是一位软件开发者。如果你的项目刚刚开始（没有向下兼容的问题），如果你知道系统想要解决的问题，如果你的投资方愿意一直付钱直到你对结果满意，你真够幸运。虽然这样的情景适用面向对象技术，但对我们大多数人来说，这是梦中才会出现的情景。

更多时候，你需要对既有软件进行扩展，你对自己所做的事情没有完整的了解，你受到生产进度的压力。这种情况下你该怎么办？

你可以重写整个程序。你可以倚赖自己的设计经验来纠正程序中存在的错误，这是创造性的工作，也很有趣。但谁来付钱呢？你又如何保证新的系统能够完成旧系统所做的每一件事呢？

你可以拷贝、修改现有系统的一部分，以扩展它的功能。这看上去也许很好，甚至可能被看做一种复用（reuse）方式：你甚至不必理解自己复用的东西。但是，随着时间流逝，错误会不断地被复制、被传播，程序变得臃肿，程序的当初设计开始腐败变质，修改的整体成本逐渐上升。

重构是上述两个极端的中庸之道。通过「重新组织软件结构」，重构使得设计思路更详尽明确。重构被用于开发框架、抽取可复用组件、使软件架构（architecture）更清晰、使新功能的增加更容易。重构可以帮助你充分利用以前的投资，减少重复劳动、使程序更简化更有性能。

假设你是一位开发者，你也想获得这些好处。你同意 Fred Brooks 所说的「应对并处理变化，是软件开发的根本复杂性之一」[2]。你也同意，就理论而言，重构能够提供上面所说的各种好处。

为什么还不肯重构你的程序呢？有几个可能的原因：

1. 你不知道如何重构。
2. 如果这些利益是长远（才展现）的，何必现在付出这些努力呢？长远看来，说不定当项目收获这些利益时，你已经不在职位上了。
3. 代码重构是一项额外工作，老板付钱给你，主要是让你编写新功能。
4. 重构可能破坏现有程序。

这些担忧都很正常，我经常听到电信公司和其他高科技公司的员工那么说。这其中有一些技术问题，以及一些管理问题。首先必须解决所有这些问题，然后开发者才会考虑在他们的软件中使用重构技术。现在让我们逐一解决这些问题。

如何重构，在哪里重构

如何才能学会重构呢？有什么工具？有什么技术？如何把这些工具和技术组合起来做出有用的事？应该何时使用它们？本书定义了好几十条重构作法，这些都是

Martin 在自己的工作经验中发掘的有用手法。重构如何被用以支持程序重大修改？本书提供了很好的例子。

在伊利诺斯大学的软件重构项目中，我们选择了一条「极简抽象派艺术家」（minimalist）路线。我们定义了较少的一组重构[1],[3]，展示它们的使用方法。我们对重构的收集系建立于自己的编程经验上。我们评估好几个面向对象框架（多数以 C++ 开发完成）的结构演化（structural evolution），和数字经验丰富的 Smalltalk 开发者交谈，并阅读他们的回顾记录。我们收集的重构手法大多很低层，例如建立或删除一个 class、一个变量或一个函数，修改变量和函数的属性，如访问权限（public 或 protected），修改函数参数等等，或者在 classes 之间移动变量和函数。我们以另一组数量较少的高级重构手法来处理较为复杂的情况，例如建立 abstract superclass、通过 subclassing 和「简化条件」等方式来简化一个 class、从现有的 class 中分解一部分，创建一个崭新而可复用的组件 class 等等（经常会在继承（inheritance）、委托（delegation）、聚合（aggregation）之间转换）。这些较复杂的重构手法是以低层重构手法定义出来的。之所以采用这种方法，乃是为了「自动化支持」和「安全」两方面考量，我将于稍后讨论。

面对一个既有程序，我们该使用哪些重构呢？当然，这取决于你的目标。一个常见的重构原因，同时也是本书关注焦点，是「调整程序结构以使（短期内）添加新功能更容易」。我将在下一节讨论这一点。除此之外，还有其他理由让你使用重构。

有经验的面向对象程序员和那些受过设计模式（design patterns）和优秀设计技巧训练的人都知道，目前已经出现数种令人满意的程序结构性质量和特征（structural qualities and characteristics），可以支持扩展性和复用性[4],[5],[6]。诸如 CRC[7] 之类的面向对象设计技术也关注定义 classes 和 classes 之间的协议（protocols）。虽然它们关注的焦点是前期设计，但也可以用这些指导方针来评价一个现有程序。

自动化工具可用来识别程序中的结构缺陷，例如函数参数过多、函数过长等等。这些都应该考虑成为重构的对象。自动化工具还可以识别出结构上的相似，这样的相似很可能代表着冗余代码的存在。比如说，如果两个函数几乎相同（这经常是「拷贝/修改」第一个函数以获得第二个函数时造成的），自动化工具就会检测到这样相似性，并建议你使用一些重构手法，将相同代码搬到同一个地方去。如果程序中不同位置的两个变量有相同名称，有时你可以使用一个变量替代它们，并在两处继承之。这些都是非常简单的例子。有了自动化工具，其他很多更复杂的情况都可以被

检测出来并被纠正。这些结构上的畸形或结构上的相似并非总是暗示你必须重构，但很多时候它们的确就是这个意思。

对设计模式（design patterns）的很多研究，都集中于良好编程风格以及程序各部位之间有用的交互模式（patterns of interactions），而这些都可以映像为结构特征和重构手法。例如 ***Template Method*** 模式[8] 的「适用性」（applicability）一节就参考了我们的 abstract superclass 重构手法[9]。

我列出了一些试探法则[1]，可以帮助你识别C++ 程序中需要重构的地方。John Brant 和 Don Roberts [10],[11] 开发出一个工具，使用更大范围的试探来自动分析 Smalltalk 程序。这个工具会向开发者建议「可用以改进程序」的重构方法，以及适合使用这些重构方法的地点。

运用这样一个工具来分析你的程序，有点像运用 lint 来改善 C/C++ 程序。这个工具尚未聪明到能够理解程序意图，它在程序结构分析基础上提出的建议，或许只有一部分是你真正想要做出的修改。作为程序员，决定权在你手上。由你决定把哪些建议用于自己的程序上。这些修改应该改进程序的结构，应该为日后的修改提供更好的支撑。

在程序员说服自己「我应该重构我的代码」之前，他们需要先了解如何重构、在哪里重构。经验是无可替代的。研究过程中，我们得益于经验丰富的面向对象开发者的经验，得到了一些有用的重构作法，以及「该在哪里使用这些重构」的认识。自动化工具可以分析程序结构，建议可能改进程序结构的重构作法。和其他大多数学科一样，工具和技术会带来帮助，但前提是你打算使用它们。重构过程中，程序员自己对重构的理解也会逐渐加深。

重构 C++程序

Bill Opdyke

1989 年，我和 Ralph Johnson 刚开始研究重构的时候，C++ 正在飞快发展，并日渐在面向对象开发圈中流行起来。Smalltalk 用户是最先认识重构重要性的一群人，而我们认为，如果能够证明重构对 C++ 程序也同样可用，就会使更多面向对象开发者对重构产生兴趣。

C++ 的某些语言特性（特别是静态型别检查）简化了一部分程序分析和重构工作。但是另一方面，C++ 语言很复杂也很庞大，这很大程度是由于其历史而造成（C++ 是从 C 语言演化而来的）。C++ 允许的某些编程风格，使程序的重构和发展变得困难。

■ **对重构有支持能力的语言特性和编程风格**

重构时，你必须找出待重构的这一部分程序被什么地方引用（指涉）。C++ 静态型别特性让你可以比较容易地缩小搜索范围。举个简单但常见的例子，假设你想要给 C++ class 的一个成员函数改名，为正确完成这个动作，你必须修改函数声明以及对这个函数的所有引用点。如果程序很大，搜索、修改这些引用点会很困难。

和 Smalltalk 相比，C++ 的 classes 继承和保护访问级别（public、protected 和 private）特性，使你更容易判断哪些地方引用了这个「待易名函数」。如果这个函数被其所属 class 声明为 private，那么这个函数的「被引用点」就只可能出现在这个 class 内部以及被这个 class 声明为 friend 的地方；如果这个函数被声明为 protected，那么引用点只可能出现在它所属的 class 内、它的 subclasses（及更底层的 subclasses）内以及它的 friends 中；如果这个函数被声明为 public（限制最少的一种访问级别），引用点被限制在上述 protected 所列情况，以及对某些特定 class 实体（对象）的操作之上——该特定 class 可以是内含「待易名函数」者，或其 subclasses，或其更底层的 subclasses。

在十分庞大的程序中，不同地点有可能声明一些同名函数。有时候，两个或多个同名函数以同一个函数取代可能更好，某些重构手法可用来做这种修改；有时候则应该给两个同名函数中的一个改名，让另一个保持原来名称。如果项目开发成员不只一人，不同的程序员可能给风马牛不相及的函数取相同的名称。在 C++ 中当你对两个同名函数中的一个改名之后，几乎总是很容易找到哪些引用点针对的是这个被易名函数，哪些引用点针对的是另一个函数。这种分析在 Smalltalk 中要困难得多。

由于 C++ 以 subclassing 实现 subtyping，所以通常可以通过「将变量或函数在继承体系中移上移下」来扩大（普通化）或缩小（特殊化）其作用域（scope）。对程序做这一类分析并进行相应重构，都是很简单的。

如果在最初开发和整个开发过程中一直遵循一些良好的设计原则，那么重构过程会更轻松，软件的进化会更容易。「将所有成员变量和大多数成员函数定义为 private 或 protected」是一个抽象技术，常常使 class 的内部重构更简单，因为对程序其他地方造成的影响被减至最低。以继承机制表现「普通化和特殊化」体系（这在 C++ 中很自然），也使日后「泛化或特化成员变量或成员函数」的重构动作更容易进行，你只需在继承体系内上下移动这些成员即可。

C++ 环境中的很多特性都支持重构。如果程序员在重构时引入错误，C++编译器通常都会指出这个错误。许多 C++ 软件开发环境都提供了强大的交叉参考和代码浏览功能。

■ 增加重构复杂度的语言特性和编程风格

众所周知，C++对 C 的兼容性是一柄双刃剑。许多程序以 C 写成，许多程序员受的训练是 C 风格，所以（至少从表面看来）转移到 C++ 比转移到其他面向对象语言容易些。但是，C++ 支持许多编程风格，其中某些违反了合理健全的设计原则。

程序如果使用诸如指针、转型操作（cast operation）和 `sizeof(object)` 之类的 C++ 特性，将难以重构。指针和转型操作会造成别名（alias），使你很难找到待重构对象的所有被引用点。上述这些特性暴露了对象的内部表现形式，违反了抽象原则。

举个例子，在可执行程序中，C++以 V-table 机制表现成员变量。从 superclass 继承而来的成员变量首先出现，而后才是自身（locally）定义的成员变量。「将某个变量移往 superclass」通常是很安全的重构手法，但如果变量是由 superclass 继承而来，不是 subclass 自身定义出来，它在可执行文件中的物理（实际）位置便有可能因这样的重构而发生改变。当然啦，如果程序中对变量的所有引用（指涉）都是通过 class interface 进行，变量的物理位置调整，并不会改变程序行为。

此外，如果程序通过指针算术运算来引用这个变量（例如程序员拥有一个对象指针，而且他知道他想赋值的变量保存于第 5 个 byte，于是他就使用指针算术，直接把一个值赋进对象的第 5 个 byte 去），那么「将变量移到 superclass」的重构手法就有可能改变程序行为。同样地，如果程序员写下 `if(sizeof(object)==15)`这样的条件式，然后又对程序进行重构，删除 class 之中未用到的变量，那么这个 class 的实体大小就会发生改变，导致先前判断为真的条件式，如今有可能判断为伪。

可曾有人根据对象大小做条件判断？C++ 提供远为清楚的接口用以访问成员变量，还会有人以指针运算进行访问吗？这样写程序实在太荒唐了不是吗？我的观点是：C++提供了这些特性（以及其他倚赖对象物理布局的特性），而某些经验丰富的程序员的确使用了它们。毕竟，从 C 到 C++ 的移植不可能由面向对象程序员或设计师来进行（只能由 C 程序员来做）。

由于 C++ 是一个如此复杂的语言（和 Smalltalk 以及 Java 相比），意图建立某种程序结构，使之得以「协助自动检查某一重构是否安全，并于安全情况下自动执行该重构」，就困难得多。

C++ 在编译期对大多数 references 进行决议（resolves），所以对一个 C++ 程序进行重构，通常需要至少重新编译程序的某一部分，重新连接并生成可执行文件，然后才能测

试修改效果。与之形成鲜明对比的是，Smalltalk 和 CLOS（Common Lisp Object System）提供解译(interpretation)和增量编译(incremental compilation)环境。因此尽管在 Smalltalk 和 CLOS 中进行一系列渐进式重构是很自然的事，对 C++ 程序来说，每次迭代（重新编译 + 测试）的成本却太高了，所以 C++ 程序员往往不太乐意经常做这种小改动。

许多应用程序都用到数据库。如果在 C++ 程序中改变对象结构，可能会需要对 database schema（数据库表格的结构、架构、定义）作相应修改。（我在重构工作中应用的许多思想都来自对面向对象数据库模型演化的研究。）

C++ 的另一个局限性（这对软件研究者的吸引力可能大于软件开发者）就是：它没有支持 meta-level 的程序分析和修改。C++ 缺乏任何类似 CLOS metaobject 协议的东西。举个例子，CLOS 的 metaobject 协议支持一个时而很有用的重构手法：将选定的对象变成另一个 class 的实体，并让所有指向旧对象的 references 自动指向新对象。幸运的是只有在极少数情况下才会需要这种特性。

■ 结语

很多时候，重构技术可以（并且已经）应用于 C++ 程序了。C++ 程序员通常希望自己的程序能在未来数年中不断演化进步，而软件演化过程正是最能凸显重构的好处。C++ 语言提供的某些特性可以简化重构，但另一些特性会使重构变得困难。幸运的是，程序员已经公认：使用诸如「指针运算」之类的语言特性并不是好主意。大多数优秀的面向对象程序员都会避免使用它们。

非常感谢 Ralph Johnson、Mick Murphy、James Roskind 以及其他一些人，向我介绍了 C++ 之于重构的威力和复杂性。

重构以求短期利益

要说明「重构有哪些中长期好处」是比较容易的。但许多公司受到来自投资方日益沉重的压力，不得不追求短期成绩。重构可以在短期之内带来惊喜吗？

那些经验丰富的面向对象开发者，成功运用重构已经有超过十年的历史了。在强调代码简洁明了、复用性高的 Smalltalk 文化中，许多程序员都变得成熟了。在这样的文化中，程序员会投入时间去进行重构，因为他应该这样做。Smalltalk 语言和实现品使得重构成为可能，这是过去绝大多数语言和开发环境都没有能够做到的。

许多早期的Smalltalk程序设计都是在Xerox、PARC这样的研究机构或技术尖端的小型开发团队和顾问公司中进行的。这些团体的价值观和许多产业化软件团队的价值观是有所差异的。Martin和我都知道：如果要让主流软件开发者接受重构思想，重构带来的利益起码有一部分必须能够在短期内体现出来。

我们的研究团队[3],[9],[12]~[15]记录了数个例子，描述重构如何和程序功能的扩展交错进行，最终同时获得短期利益和长期利益。我们的一个例子是Choices文件系统框架。最初这个框架实现了BSD（Berkeley Software Distribution）Unix文件系统格式。后来它又被扩展支持Unix System V、MS-DOS、永续性（persistent）和分布式（distributed）文件系统。框架开发者采用的办法是：先把实现BSD Linux的部分原样复制一份过来，然后修改它，使它支持System V。系统最终可以有效运作，但充斥大量重复的代码。加入新代码后，框架开发者重构了这些代码，建立abstract superclasses容纳两个Unix文件系统的共通行为。相同的变量和函数被移到superclasses中。当两个对应函数几乎相同、但不完全相同时，他们就在subclass中定义新函数来包容两者不同之处，然后在原先函数里头把这些代码换成对新函数的调用。这样一来，两个subclass的代码就逐渐变得愈来愈相似了。一旦两个函数变得完全相同，就可以将它们搬移到共同的superclass去。

这些重构手法为开发者提供了多方面好处，既有短期利益，也有长期利益。短期来看，如果在测试阶段发现共同的代码有错误，只需在一个地方修改就行了。代码总量变少了。「特定于某一文件系统的行为」与「两种文件系统的共同行为」清晰地分开了，这使得追踪、修补「特定于某种文件系统的行为」更加容易。中期来看，重构得到的抽象层对于定义后续文件系统常常很有帮助。当然，现有的两种文件系统的共通行为未必就完全适用于第三种文件格式，但现有的共享基础是一个很有价值的起点。后继的重构动作可以澄清究竟哪些东西真正是所有文件系统共有的。框架开发团队发现：随着时间流逝，「增加新文件系统的支持」愈来愈省劲。就算新的格式更复杂、开发团队经验更浅，情况也一样。

我还可以找出其他例子来证明重构能够带来短期和长期利益，但是Martin早已做了此事，我不想再延长他的列表。还是拿我们都非常熟悉的一件事来做个比喻吧：我们的身体健康状况。

从很多角度来说，重构就好像运动、吃适当的食物。许多人都知道：我们应该多锻炼身体，应该注意均衡饮食。有些人的生活文化中非常鼓励这些习惯，有些人没有这些好习惯也可以混过一段时间，甚至看不出有什么影响。我们可以找各种借口，但如果一直忽视这些好习惯，那么我们只是在欺骗自己。

有些人之运动和均衡饮食，动机着眼于短期利益（例如精力更充沛、身体更灵活、自尊心增强……等等）。几乎所有人都知道这些短期利益非常真实。许多人（但不是所有人）都时断时续做过一些努力，另一些人则是不见棺材不掉泪，不到关键时刻不会有足够动力去做点什么事。

没错，做事应该谨慎。在着手干一件事之前，应该先向专家咨询一下。在开始运动和均衡饮食之前，应该先问问自己的保健医生。在开始重构之前，应该先查找相关资源——你手上这本书和本章引用的其他数据都很好。对重构有丰富经验的人可以向你提供更到位的帮助。

我见过的一些人正是「健康与重构」的典范。我羡慕他们旺盛的精力和超人的工作性能。反面典型则是明显的粗心大意爱忘事，他们的未来和他们开发的软件产品的未来，恐怕都不会很光明。

重构可以带来短期利益，让软件更易修改、更易维护。重构只是一种手段，不是目的。它是「程序员或程序开发团队如何开发并维护自己的软件」这一更宽广场景的一部分[3]。

降低重构带来的额外开销（Reducing the Overhead of Refactoring）

「重构是一种需要额外开销的活动。我付钱是为了让程序员写出新的、能带来收益的软件功能」。对于这种声音，我的回复总结如下：

- 目前已有一些工具和技术，可以使重构「快速」而「相对无痛苦」地完成。
- 一些面向对象程序员的经验显示，重构虽然需要额外开销，但可以从它「在程序开发的其他阶段协助降低所需心力及滞怠时间」而获得补偿。
- 尽管乍见之下重构可能有点笨拙、开销太大，但是当它成为软件开发规则的一部分，人们就不会再觉得它费事，反而开始觉得它是必不可少的。

伊利诺斯大学的软件重构团队开发的 Smalltalk 自动化重构工具也许是目前最成熟的自动化重构工具（参见第 14 章）。你可以从他们的网站（http://st-www.cs.uiuc.edu）自由下载这个工具。尽管其他语言的重构工具还没能这么方便，但是我们的论文和本书介绍的许多技术，都可以相对简单地套用，只要有一个文本编辑器或一个浏览器就足够了。软件开发环境和浏览器技术已经在最近数年获得了长足发展。我们希望将来能看到更多重构工具投入使用。

Kent Beck 和 Ward Cunningham 都是经验丰富的 Smalltalk 程序员，他们已经在 OOPSLA 和其他论坛上提出报告：重构使他们能够更快开发证券交易之类的软件。从 C++ 和 CLOS 开发者那里，我也听到了同样的消息。本书之中 Martin 介绍了重构对于 Java 程序的好处。我们希望读过本书、使用书中介绍的重构原则的人们，能够给我们带来更多好消息。

从我的经验看来，只要重构成为日常事务的一部分，人们就不会觉得它需要多么高昂的代价。说来容易做来难。对于那些怀疑论者，我的建议就是：只管去做，然后自己决定。但是，请给它一点时间证明它自己。

安全地进行重构

安全性（safety）是令人关心的议题，特别对于那些开发、维护大型系统的组织更是如此。许多应用程序背负着财政、法律和道德伦理方面的压力，必须提供不间断的、可靠的、不出错的服务。有许多组织提供大量培训和努力，力图以严谨的开发过程来帮助他们保证产品的安全性。

但是，对很多程序员来说，安全性的问题往往没那么严重。我们总是向孩子们灌输「安全第一」的思想，自己却扮演渴望自由的程序员、西部牛仔和血气方刚的驾驶员的角色，这实在是个莫大讽刺。给我们自由，给我们资源，看我们飞吧。不管怎么说，难道我们真的希望公司放弃我们的创造性果实，就为了获得可重复性和一致性吗？

这一节我将讨论安全重构（safe refactoring）的方法。和 Martin 在本书先前章节介绍过的方法相比，我关注的方法其结构比较更组织化、更严格，可因此排除重构可能引入的很多错误。

安全性（safety）是一个很难定义的概念。直观的定义是：所谓「安全重构」（safe refactoring）就是不会对程序造成破坏的重构。由于重构的意图就是在不改变程序

行为的前提下修改程序结构，所以重构后的程序行为应该与重构前完全相同。

如何进行安全重构呢？你有以下数种选择：

- 相信你自己的编码功力。
- 相信你的编译器能捕捉你遗漏的错误。
- 相信你的测试套件（test suite）能捕捉你和编译器都遗漏的错误。
- 相信代码复审（code review）能捕捉你、编译器和测试套件（test suite）都遗漏的错误。

Martin 在他的重构原则中比较关注前三个选项。大中型公司则常常以代码复审作为前三个步骤的补充。

尽管编译器、测试套件、代码复审、严守纪律的编码风格都很有价值，但所有这些方法还是有下列局限性：

- 程序员是可能犯错的，你也一样（我也一样）。
- 有一些微妙和不那么微妙的错误，编译器无法捕捉，特别是那些与继承相关的作用域错误（scoping errors）[1]。
- Perry、Kaiser[16] 和其他人已经指出，尽管「将继承作为一种实现技术」的作法让测试工作简单了不少，但由于先前「向 class 的某个实体发出请求」的很多操作如今「转而向 subclasses 发出请求」，我们仍然需要大量测试来覆盖这种情况。除非你的测试设计者是全知全能的上帝，或除非他对细节非常谨慎，否则就有可能出现测试套件覆盖不到的情况。「是否测试了所有可能的执行路径」？这是一个无法以计算判定的问题。换句话说，你无法保证测试套件覆盖所有可能情况。
- 和程序员一样，代码复审人员也是可能犯错的。而且复审人员可能因为忙于自己的主要工作，无法彻底检查别人的代码。

我在研究工作中使用的另一种方法是：定义并快速实现一个重构工具的原型，用以检查某项重构是否可以安全地施加于程序身上。如果可以，就重构之。这避免了大量可能因为人为错误而引入的臭虫。

在这里，我将概括介绍我的安全重构（safe refactoring）法。这可能是本章最具价

值的一部分了。如果你想获得更详细的信息，请看我的论文[1]和本章末尾所列的参考文献，也可以参考本书第14章。如果你觉得这一部分有点过分偏重技术，不妨跳过本节余下的数小段。

我的重构工具的一部分是程序分析器（program analyzer），这是一个用来分析程序结构的程序（被分析的对象是将来打算施加某项重构的一个C++程序）。这个工具可以解答一系列问题，内容涉及作用域（scoping）、型别（typing）和程序语义（程序的意图或用途）等方面。作用域的问题与继承有关，所以这一分析过程比起很多「非面向对象程序分析」要复杂；但C++的某些语言特性（例如静态型别，static typing）又使得这一分析过程比起「对Smalltalk等动态型别（dynamic typing）程序的分析」要简单。

举个例子，假设我们的重构是要删除程序中的某个变量。我的工具可以判断程序其他部分（如有的话）是否引用了这个变量。如果有，径自删除这一变量将会造成dangling references，那么这项重构就是不安全的。于是工具用户就会收到一个错误标记（error flag）。用户可能因此决定放弃进行这次重构，也可能修改程序中对此变量的引用点，使它们不再引用它，然后才进行重构，删除该变量。这个工具还可以进行其他许多检查，其中大多数都和上述检查一样简单，有些稍微复杂。

在我的研究中，我把安全（safety）定义为：「程序属性（包括作用域和型别等等）在重构之后仍然保持不变」。很多程序属性很像数据库中的完整性约束（integrity constraints）——修改database schema（数据库表格的结构、架构、定义）时，完整性约束必须保持不变[17]。每个重构都伴随一组必要前提，如果这些前提得到满足，该重构就能保证程序属性获得维持。一旦确定某次重构的全部过程都安全，我的工具才会执行该次重构。

幸运的是，对于「重构是否安全」进行的检查（尤其是对于数量占绝对优势的低层重构）往往是琐屑而平淡无奇的。为了保证较高层重构、较复杂重构的安全性，我们以低层重构来定义它们。例如「建立一个abstract superclass」的复杂重构手法就被定义为数个较小步骤，每个步骤都以较简单的重构完成，像是创建和搬移变量或函数等等。只要证明复杂重构的每一个步骤是安全的，我们就可以确定整个复杂重构也是安全的。

在某些十分罕见的情况下，重构其实可以在「工具无法确认」时仍然安全施加于程序身上。在那种情况下，工具会选择较安全的方式：禁止重构。拿先前例子来说，

你想删除程序中的某个变量，但程序其他地方对该变量有引用动作。然而或许这个引用动作所处段落永远不会被执行到，例如它也许出现于条件式（如 if-then）中，而它所处分支永远不为真。如果肯定这个分支永远不为真，你可以移除它，连同那个影响你重构的引用点一并移除。然后你就可以安全地进行重构，删除你想删除的变量或函数了。只不过，一般情况下你无法肯定分支永远为假（如果你继承了别人开发的代码，你有多大把握安全删掉其中某段代码？）

重构工具可以标记出这种「可能不安全」的引用关系，并向用户提出警告。用户可以先把这段代码放在一旁。一旦用户能够肯定引用点永远不会被执行到时，他就可以把这段多余代码移除，而后进行重构。这个工具让用户知道存在这么一个隐藏的引用关系，而不是盲目地进行修改。

这听起来好像有点复杂，作为博士论文的主题倒是不错（博士论文的主要读者——论文评议委员会——比较喜欢理论性题目），但是对于实际重构有用吗？

所有这些安全性检查都可以在重构工具中实现。如果程序员想要重构一个程序，只需以这个工具检查其代码。如果检查结果为「安全」，就执行重构。我的工具只是个研究雏型。Don Roberts、John Brant、Ralph Johnson 和我[10]后来实现了一个体质更健壮、功能更齐备的工具（参见第 14 章），这是我们对于「Smalltalk 程序重构」研究的一部分。

安全性可分很多不同级别施行于重构身上。其中某些级别容易实施，但不保证高级安全性。使用重构工具有很多好处。它可以在高级问题中做许多简单而乏味的检查和标记。如果不做这些检查，重构动作有可能导致程序完全崩溃。

编译、测试和代码复审可以指出很多错误，但也会遗漏一些错误，重构工具则可以帮助你抓住漏网之鱼。尽管如此，编译、测试和代码复审仍然是很有价值的，在实时（real-time）系统的开发和维护中更是如此。这些系统中的程序往往不是孤立运行的，它们是大型通信系统网络中的一部分。有些重构不但把代码清扫干净，而且会让程序跑得更快。然而提升某个程序的速度，可能会在另一个地方造成性能瓶颈。这就好像你升级 CPU 进而提升了部分系统性能，你需要以类似方法来调整、测试系统整体性能。另一方面，有些重构也可能略微降低系统整体性能。一般说来，重构对性能的影响是微不足道的。

「安全性作法」用来保证重构不会向程序引入新错误。这些作法并不能检查或修复程序重构前就存在的错误。但重构可以使你更容易找到并修复这些错误。

13.3 现实的检验（再论）

如果要让软件开发者接受重构，首先必须解决一些非常实际的问题。下面列出四个最常见的问题：

- 程序员不知道如何重构。
- 如果重构利益是长远的，何必现在付出这些努力呢？长远看来，说不定当项目收获这些利益时，你已经不在职位上了。
- 代码重构是一项额外工作，老板付钱给程序员，主要是为了编写新功能。
- 重构可能破坏现有程序。

本章中我简单回答了这些问题，并为那些希望更深入钻研的人指出方向。

对于某些项目，以下问题也是需要关心的：

- 如果代码由多位程序员共同拥有，怎么办？一方面，许多传统的变更管理机制都可以解决这个问题；另一方面，如果软件设计良好，又经过重构，子系统之间就会有效分离，于是很多重构手法都只会影响代码的一小部分。
- 如果你的 code base（代码材料库）中有多重版本的代码，怎么办？有些时候，重构和每一个版本相关，这种情况下我们必须在重构前先对所有版本进行安全测试。另一些时候，重构可能只与某些版本相关，那么，检查过程和重构过程就简单多了。如果打算同时管理多个版本变化，通常需要使用许多传统的版本管理技术。如果想将多个版本并入一个新的 code base（代码材料库）中，重构也会有所帮助，因为它可以顺畅地简化版本控制工作。

总而言之，「让软件开发者相信重构的实际价值」和「让博士论文评议委员会相信重构研究够得上博士水平」是完全不同的两码事。在写完毕业论文以后，我又花了相当长的时间才对这种差异有了足够充分的认识。

13.4 重构的资源和参考资料

本书至此，我希望你已经开始计划在自己的工作中使用重构技术，并鼓励公司里的其他人也这样做。如果你还犹豫不决，也许你愿意参考以下列出的数据，或是和 Martin（Fowler@acm.org）、我或其他有重构经验的人联系。

如果你打算深入研究重构，下列一些参考资料你可能会想看看。正如 Martin 所说，本书不是重构的第一份书面数据，但是（我希望）它能让更多人关注重构概念和它带来的利益。我的博士论文是这个主题的第一份正式书面数据，但如果读者有兴趣探索重构早期的基础研究，应该先看这几篇文章：[3],[9],[12],[13]。在 OOPSLA 95 和 OOPSLA 96 大会上，重构都是一个教学性主题[14],[15]。至于那些同时对设计模式（design patterns）和重构（refactoring）感兴趣的读者，Brian Foote 和我在 PLoP 94 上发表并于日后被收入 Addison-Wesley 出版社之 "Pattern Languages of Program Design" 丛书第一卷的 "*Lifecycle and Refactoring Patterns That Support Evolution and Reuse*" 是个不错的起点。此外，我对重构的研究很大程度建立在 Ralph Johnson 和 Brian 关于「面向对象应用程序框架和可复用 classes 的设计」[4] 研究基础上。John Brant、Don Roberts 和 Ralph Johnson 在伊利诺斯大学对重构的研究的主要关注点是 Smalltalk 程序重构[10],[11]。他们的网站（http://st-www.cs.uiuc.edu）上有他们的最新研究成果。最近，面向对象研究社群对重构的兴趣与日俱增。OOPSAL 96 会议之中一个主题为「重构与复用」的分会场议上也发表了数篇相关文章[18]。

13.5 从重构联想到软件复用和技术传播

前面所提的现实世界问题，并不仅仅存在于重构中。它们广泛存在于软件的演化（evolution）和复用（reuse）中。

过去数年，我用了很多时间来关注软件复用性、平台、框架、模式、遗留系统（往往涉及「非面向对象」软件）的发展相关问题。除了在朗讯（Lucent）和贝尔实验室（Bell Labs）开发项目，我还参加了其他公司的员工讨论会——他们也曾经与类似问题搏斗过。[19]~[22]

「可复用法」的现实问题，和重构的相关问题很类似：

- 技术人员可能不知道「该复用什么」或「如何复用」。
- 技术人员可能对于采用「可复用法」（reuse approach）缺乏动力，除非他们能够获得短期利益。

- 如果要成功适应「可复用法」（reuse approach），额外开销（overhead）、学习曲线（learning curve）和发现成本（discovery cost）都必须考虑。
- 采用「可复用法」（reuse approach）不该引起项目混乱。项目中可能有很大压力：尽管面对遗留系统的束缚，仍应让现有资产或实现品获得杠杆作用。新的实现品应该与现有系统协同工作，或至少向下兼容于现有系统。

Geoffrey Moore[23]把「技术的接纳过程」描述为一条钟型（bell-shaped）曲线：前段包括先行者（innovator）和早期接受者（early adopter），中部急剧增加的人群包括早期消费群体（early majority）和晚期消费群体（late majority），后段则是那些行动缓慢者（laggard）。一个思想或产品如果要成功，必须得到早期消费者和晚期消费者的广泛支持。另一方面，许多对于先行者和早期接受者很有吸引力的想法，最终彻底失败，因为它们没能跨越鸿沟，让早期消费者和晚期消费者接纳它们。之所以有这样的鸿沟是因为，不同的消费人群有着不同的消费动机。先行者和早期接受者感兴趣的是新技术、「范式移转和突破性思想」的愿景（visions of paradigm shifts and breakthroughs）。早期和晚期消费群则主要关心成熟度、成本、支持，以及这种新思想或新产品是否被「与他们有着相似需求」的其他人成功套用。

要打动并说服软件开发者，所需方式和打动并说服软件研究者是完全不同的。软件研究者通常是Moore所说的「先行者」，软件开发者（尤其是软件经理）则往往属于早期或晚期消费者。如果想要让你的思想深入所有人心，了解这一差异是非常重要的。是的，无论软件复用或重构，要想打动软件开发者，这一点都至关重要。

在朗讯（Lucent）和贝尔实验室（Bell Labs）中我发现，鼓励「复用性」应用及运行其必要平台，得冒一点风险。这需要主管人员精心制定策略、在中阶经理层组织领导会议、与项目开发组协商、通过研讨会和出版物向广大研究人员和开发人员宣扬这些技术的好处。在这整个过程中，很重要的几件事是：对员工进行培训、尽量获取短期利益、减少额外开销、安全引入新技术。这些见识，都是从我对重构的研究中得来的。

我的论文指导教授Ralph Johnson审查本章草稿时指出：这些原则不仅可应用于重构和软件复用性，同时也是技术传播时的常见问题。如果你正试图说服别人重构（或采用其他某种技术或实践），请注意保证自己随时关注这些问题，这样才能深入人心。技术的传播是很困难的，但不是做不到。

13.6 结语（A Final Note）

非常感谢你花时间阅读本章。我尝试解决你可能会有的关于重构的一些问题，并尝试让你了解重构的一些现实问题，这些问题亦存在于更广泛的领域中，例如软件演化和复用。希望你阅读本章之后，生出「在自己的工作中也使用这些想法」的热情。最后，祝你在软件开发之路一帆顺风。

13.7 参考文献（References）

1. Opdyke, William F. "Refactoring Object-Oriented Frameworks." Ph.D. diss., University of Illinois at Urbana-Champaign. Also available as Technical Report UIUCDCS-R-92-1759, Department of Computer Science, University of Illinois at Urbana-Champaign.

2. Brooks, Fred. "No Silver Bullet: Essence and Accidents of Software Engineering." In *Information Processing 1986: Proceedings of the IFIP Tenth World Computing Conference*, edited by H.-L.Kugler. Amsterdam: Elsevier, 1986.

3. Foote, Brian and William F. Opdyke. "Lifecycle and Refactoring Patterns That Support Evolution and Reuse." *In Pattern Languages of Program Design*, edited by J. Coplien and D. Schmidt. Reading, Mass.: Addison-Wesley, 1995.

4. Johnson, Ralph E. and Brian Foote. "Designing Reusable Classes." *Journal of Object-Oriented Programming* 1(1988): 22-35.

5. Rochat, Roxanna. "In Search of Good Smalltalk Programming Style." Technical report CR-86-19, Tektronix, 1986.

6. Lieberherr, Karl J. and Ian M. Holland. "Assuring Good Style For Object-Oriented Programs." *IEEE Software* (September 1989) 38-48.

7. Wirfs-Brock, Rebecca, Brian Wilkerson and Luaren Wiener. *Design Object-Oriented Software*. Upper Saddle River, N.J.: Prentice Hall, 1990.

8. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1985.

9. Opdyke, William F. and Ralph E. Johnson."Creating Abstract Superclasses by Refactoring." In *Proceedings of CSC '93: The ACM 1993 Computer Science Conference*. 1993.

10. Roberts, Don, John Brant, Ralph Johnson, and William Opdyke."An Automated Refactoring Tool." In *Proceedings of ICAST 96: 12th International Conference on Advanced Science and Technology*. 1996.

11. Roberts, Don, John Brant, and Ralph E. Johnson. "A Refactoring Tool for Smalltalk." *TAPOS* 3(1997) 39-42.

12. Opdyke, William F., and Ralph E. Johnson. "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems." In *Proceedings of SOOPPA '90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*. 1990.

13. Johnson, Ralph E. and William F. Opdyke. "Refactoring and Aggregation." In *Proceedings of ISOTAS '93: International Symposium on Object Technologies for Advanced Software*. 1993.

14. Opdyke, William and Don Roberts. "Refactoring." Tutorial presented at OOPSLA 95: 10th Annual Conference on Object-Oriented Program Systems, Languages and Applications, Austin, Texas, October 1995.

15. Opdyke, William and Don Roberts. "Refactoring Object-Oriented Software to Support Evolution and Reuse." Tutorial presented at OOPSLA 96: 11th Annual Conference on Object-Oriented Program Systems, Languages and Applications, San Jose, California, October 1996.

16. Perry, Dewayne E., and Gail E. Kaiser. "Adequate Testing and Object-Oriented Programming." *Journal of Object-Oriented Programming* (1990).

17. Banerjee, Jay, and Won Kim. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases." In *Proceedings of the ACM SIGMOD Conference*, 1987.

18. Proceedings of OOPSLA 96: Conference on Object-Oriented Programming Systems, Languages and Applications, San Jose, California, October 1996.

19. Report on WISR '97: Eighth Annual Workshop on Software Reuse, Columbus, Ohio, March 1997. *ACM Software Engineering Notes*. (1997).

20. Beck, Kent, Grady Booch. Jim Coplien, Ralph Johnson, and Bill Opdyke. "Beyond the Hype: Do Patterns and Frameworks Reduce Discovery Costs?" Panel session at OOPSLA 97: 12th Annual Conference on Object-Oriented Program Systems, Languages and Applications, Atlanta, Georgia, October 1997.

21. Kane, David, William Opdyke, and David Dikel."Managing Change to Reusable

Software." Paper presented at PLoP 97: 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, September 1997.

22. Davis, Maggie, Martin L. Griss, Luke Hohmann, Ian Hopper, Rebecca Joos and William F. Opdyke. "Software Reuse: Nemesis or Nirvana?" Panel session at OOPSLA 98: 13th Annual Conference on Object-Oriented Program Systems, Languages and Applications, Vancouver, British Columbia, Canada, October 1998.

23. Moore,Geoffrey A. *Cross the Chasm: Marketing and Selling Technology Products to Mainstream Customers*. New York: HarperBusiness, 1991.

14

重构工具
Refactoring Tools

by Don Roberts and John Brant

重构的最大障碍之一就是：几乎没有工具对它提供支持。那些把重构作为文化成分之一的语言（例如 Smalltalk）通常都提供了强人的开发环境，其中对代码重构的众多必要特性都提供了支持。但即使是这样的环境，到目前为止，也只是对重构过程提供了部分支持，绝大部分工作仍然得靠手工完成。

14.1 使用工具进行重构

和手工重构相比，自动化工具所支持的重构，给人一种完全不同的感觉。即使有测试套件（test suite）织成的安全网，手工重构仍然是很耗时的工作。正是这个简单的事实造成很多程序员不愿进行重构，尽管他们知道自己应该重构，但毕竟重构的成本太大了。如果能够把重构变得像调整代码格式那么简单，程序员自然也会乐意像整理代码外观那样去整理系统的设计。而这样的整理对代码的可维护性、可复用性和可理解性，都能够带来深远的正面影响。Kent Beck 如是说：

> *Kent Beck*
>
> **Refactoring Browser** 将会完全改变你的编程思路。以前你可能会想：『呃，我应该修改这个名字，但……』，现在，所有这些让你烦心的事情都烟消云散了，因为 [Refactoring Browser] 里头有个菜单选项（menu item）就是专门用来易名的，你只管放心用它就是了。
>
> 刚开始使用这个工具时，我按照以前的节奏，走了大概两个小时：我打算进行一项重构，于是抬头望着天空五分钟，然后手工完成重构，然后再一次抬头望天。很快我就发现：我必须学会以更大的范围、更快的节奏来考虑重构。现在，开发过程中我大约以一半时间进行重构，另一半时间输入新代码，两者的进行速度几乎完全相同。

由于有了这种级别的工具支持，重构和编程之间的差异愈来愈小了。我们几乎不会再说『我正在编程』或『我正在重构』，我们说得更多的是『把这个函数的这一部分提炼出来，把它推到 superclass 去，然后添加一行语句，调用新 subclass 中的新函数——我正在开发的那个函数。』由于自动化重构之后无须测试，因此编程与重构之间的差异、「更换帽子」的过程等等尽管仍然存在，但都远不如以前那样明显了。

以 *Extract Method*（110）这一重要的重构手法为例。如果你要手工进行此一重构，需要检查的东西相当多。如果使用 Refactoring Browser，你只需简单地圈选出你要提炼的段落，然后点选菜单选项（menu item）"Extract Method" 就行了。Refactoring Browser 会自动检查被圈选的代码段落是否可以提炼。代码无法提炼的原因可能有以下几点：它可能只包含部分标识符声明，或者可能对某个变量赋值而该变量又被其他代码用到。所有这些情况，你都完全不必担心，因为重构工具会帮助你处理这一切。然后，Refactoring Browser 会计算出新函数所需的参数，并要求你为新函数取一个名称。你还可以决定新函数参数的排列顺序。所有的准备工作都做完以后，Refactoring Browser 会把你圈选的代码从源函数中提炼出来，并在源函数中加上对新函数的调用。随后它会在源函数所属的 class 中建立新函数，并以用户指定的名称为新函数命名。整个过程只需 15 秒种。你可以拿这个时间长短和手工执行 *Extract Method*（110）各步骤所需时间做个比较，看看自动化重构工具的威力。

随着重构成本的降低，设计错误也不再像从前那样带来昂贵代价了。由于弥补设计错误所需的成本降低了，需要预先做的设计也就更少了。预先设计是一项带有预测性质的工作，因为项目激活之时，需求往往还不明朗。由于设计时尚未编写代码，所以正确的设计方式应该是：尽量简化「需求尚未明朗」的那一部分代码。过去，无论最初的设计方案水平如何，我们都不得不忍受，因为修改设计的代价实在太高了。有了自动化重构工具的帮助，我们可以让设计更具可变性，因为修改设计不再需要付出那么高的代价了。如今，我们可以只对当前完全了解的问题进行设计，因为我们知道以后可以很方便地扩展设计方案以加入额外的灵活性。我们不再需要预测系统未来所有可能的修改。如果发现当前的设计给编程带来麻烦，造成第 3 章所说的臭味，我们可以很快修改设计，使代码更干净、更可维护。

工具辅助下的重构工作，也影响了测试。拥有自动化重构工具的辅助之后，所需测试少多了，因为很多重构都可以自动进行，无需再做测试。当然，总有一些重构是无法自动进行的，因此测试步骤永远都不可能被完全忽略。经验显示：在自动化重构工具的协助下，我们每天所需运行的测试数量，和在「无自动化重构工具」环境中大致相当，但完成的重构数量则大大增加。

正如Martin指出，Java也需要这样的自动化重构工具。以下我们将提出一些准则——只有满足这些准则的自动化重构工具，才是成功的工具。尽管也提到了技术方面的准则，但我们相信，实用性方面的准则更重要得多。

14.2 重构工具的技术标准（Technical Criteria）

重构工具最主要的用途就是让程序员可以不必重新测试，便能对代码进行重构。即使有了自动化测试工具，测试仍然是很费时间的，如果能完全避免测试，将可极大加快重构过程。本小节简短讨论重构工具的技术标准。惟有满足这些标准，重构工具才能在「保持程序行为」的前提下，对程序进行改造。

程序数据库（Program Database）

对于重构工具，最早被人们所认识的需求就是「贯穿整个程序，搜索各种程序元素」的能力。例如，对于某个特定函数，找出其所有可能被调用点；对于某个特定的*instance*变量（译注：non-static变量），找到读/写该变量的所有函数。在Smalltalk这样紧密集成的环境中，这类信息总是被维护为一种便于搜索的格式。这不是传统意义上的数据库，但的确是一个可搜索的数据库。程序员只需执行一次搜索动作，就可以找到任何程序元素的交叉引用（cross references）。这种能力主要源自代码的动态编译机制：当任何一个class被修改，就立刻被编译为bytescodes，而上述的「数据库」则同时得到更新。在较为静态的开发环境如Java中，程序员是把代码输入到文本文件中；这种环境下如果要更新程序数据库，就必须运行一个程序来处理这些文本文件，从中提炼相关信息。这样的更新过程和Java代码自身的编译过程很相似。一些比较先进的开发环境（例如IBM VisualAge for Java）则模仿了Smalltalk的程序数据库动态更新机制。

有一种原始（粗糙）的作法是：以诸如 grep 之类的文本处理工具来进行搜索。这种办法很快就归于失败，因为它无法区分名为 foo 的变量和名为 zoo 的函数。要建立程序数据库，就必须借助语义分析来判断程序中每个语汇单元（token）在语句中的地位。而且这种分析在 class 定义和函数定义两层面上都不可少：在 class 定义层面上，需要以语义分析（semantic analysis）来区分 *instance* 变量和函数；在函数定义层面上，需要以语义分析来区分 *instance* 变量和函数引用（method references）。

解析树（Parse Trees）

绝大多数重构都必须处理函数层面下的一部分系统，通常是对「被修改之程序元素」的引用。举个例子，如果某个 *instance* 变量被改名，那么该 class 及其 subclass 中对于该 *instance* 变量的所有引用都必须更新。有些重构手法则整个运作于函数层面下，例如将某个函数的一部分提炼为一个独立函数。由于对函数的任何修改都必须能够处理函数结构，因此我们需要 parse trees（解析树）的帮助。这是一种数据结构，可用以表现函数的内部结构。下面是个简单例子：

```
public void hello( ){
    System.out.println("Hello World");
}
```

这个函数相应的 parse trees（解析树）如图 14.1。

准确性（Accuracy）

由工具实现的重构，必须合理保持程序原有行为。当然，完全的行为保持是不可能达到的，重构总是会给程序带来一些细微改变。例如重构可能会对程序的运行速度带来数个微秒的变化，这算是「完全的行为保持」吗？通常这般微小差异不会对程序造成影响；但如果程序有严格的实时性要求（real-time constraints），这一点点差异就可能导致整个程序出错。

即使是传统程序（而非实时系统）也可能被重构破坏。假设你的程序建构了一个字符串，然后使用 Java reflection API 执行以这个字符串命名的函数，那么如果日后你修改这个函数的名称，程序就会抛出一个异常；重构前的程序不会这样做。

然而，对绝大多数程序来说，重构可以相当准确。只要「可能破坏重构准确性」的因素都被识别出来，重构技术员就可以避免在不适当时候进行重构，也可以避免对于「重构工具无法修补的程序」错误地进行手工修补。

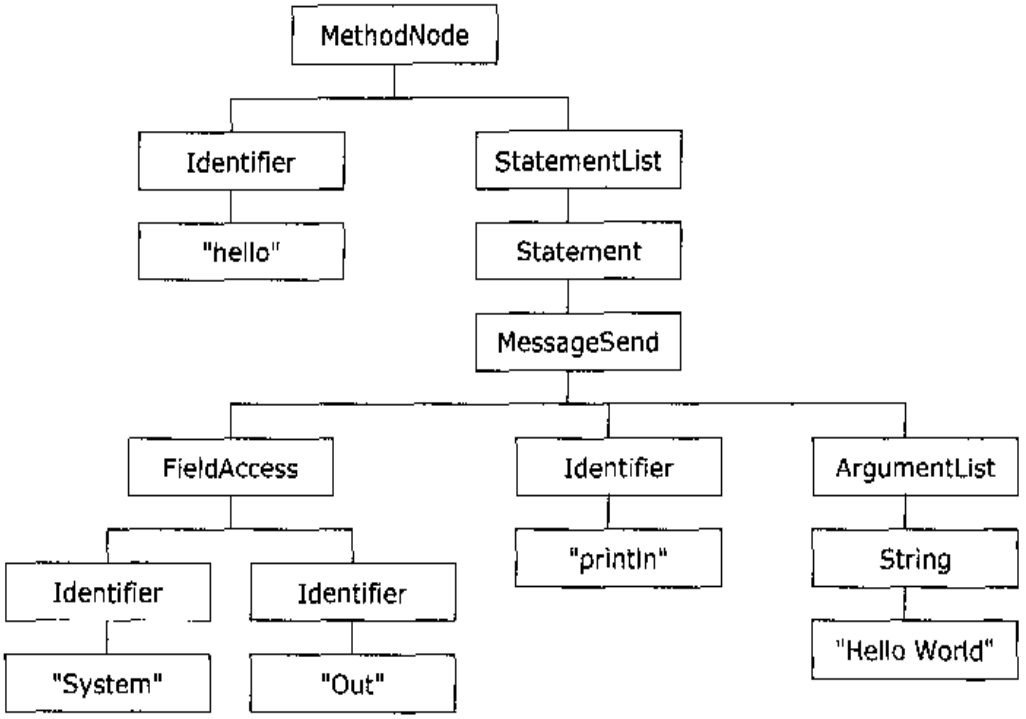


图 14.1 `hello()`函数（p.404）的解析树（parse trees）

14.3 重构工具的实用标准（Practical Criteria）

工具之所以被创造出来，是为了帮助人们完成工作。如果工具不能适应人们的工作方式，人们就不会使用它。重构工具的最重要标准（要求）就是：和其他工具共同集成出重构过程（refactoring process）。

速度（Speed）

重构前的分析和必要调整，可能会耗费较多时间，因为它们有可能非常复杂。工具设计者必须考虑这些前期工作对时间和准确性（accuracy）的影响。如果重构前需要大量准备工作，程序员就不会使用自动化重构工具，他们宁可手工进行重构。是的，开发速度总是很重要的。在开发 Refactoring Browser 的过程中，有数个重构手法并没有被我们实现出来，正是因为我们无法在可被接受的时间内安全实现它们。但是我们的工作仍然颇有成绩，绝大多数重构都可以在极短时间内以极高的准确度完成。计算器科学家总是但愿能够覆盖「特定函数无法处理之（所有）边界情况」，但事实上绝大多数程序并不涉及那些边界情况。因此，简单而快速的作法便可美好地胜任这些工作。

如果重构前的分析需要花费太长时间，一个简单的解决办法就是：直接询问程序员你所需要的信息。这种办法把「保证准确性（accuracy）」的责任交给了程序员，于是分析过程可以进行得更快一些。很多时候程序员其实都知道必要信息。尽管这种办法可能不够安全（因为程序员有可能犯错），但「错误」的责任也落了一部分在程序员肩上。讽刺的是，这竟然使程序员更有可能使用这些工具，因为他们无需倚赖程序的探询（错误尝试，heuristic）来收集信息。

Undo（撤销）

「自动化重构」令开发者得以采用探索方式（exploratory approach）进行设计。你可以试着把代码移到他处，观察新设计方案是否有效。由于我们假设重构都能够保持程序的原本行为，所以反向重构（亦即对原重构的撤销）也应该不影响程序的原本行为。Refactoring Browser 的早期版本并没有 Undo（撤销）功能，这使用户无法对重构充满信心。重构的撤销相当困难。但是很多时候我们偏偏必须找出程序重构前的版本，重新开始，这可真够讨厌的。于是我们后来为 Refactoring Browser 加上了 Undo 功能，又克服了一个障碍。现在，我们可以放心尝试，不会遭受任何惩罚，因为我们总是可以回到原先的任何一个版本。我们可以建立新的 class、搬移函数、观察代码的行为，而后又改变想法，走另一个完全不同的方向。这一切都可以非常快速地完成。

与其他工具集成（Integrated with Tools）

过去十年以来，集成开发环境（IDE）已经成为绝大多数开发项目的核心工具。IDE 将编辑器、编译器、连接器、调试器以及程序开发所需的其他所有工具，都集成于一起，开发者可以在同一个环境中极方便地使用所有这些工具。Refactoring Browser for Smalltalk 早期版本是一个独立于标准 Smalltalk 开发工具之外的独立工具，我们发现根本没人使用这样的产品，就连我们自己都不用。但是把重构功能直接集成到 Smalltalk Browser 之后，我们就开始经常使用它。「是否就在吾人指尖」造成了这一切的不同。

小结

我们开发并使用 Refactoring Browser 已经有好多年了，我们已经习惯使用它来重构它自身的代码。Refactoring Browser 之所以获得成功，原因之一在于：我们都是程序员，并且我们一直力图让它满足我们自己的需求。如果我们遇上一个手工执行的重构项，而又觉得它具有普遍意义，我们就在 Refactoring Browser 中实现它。如果哪里运行太慢，我们就把它调快一点；如果哪里的准确性还不够，我们也会改进它。

我们相信：自动化重构工具是控制「随软件项目演化而产生之复杂度」的最好办法。如果没有合适工具协助我们解决那些复杂度，软件就会变得臃肿不堪、错漏百出、不堪一击。由于 Java 比那些与它语法相近的语言简单得多，因此开发 Java 重构工具也容易得多。我们希望这种工具早日出现，我们希望能避免发生于 C++ 身上的缺陷。

15

集成

Put It All Together

by Kent Beck

现在，你已经拥有了七巧板的每一块：你已经了解了重构的基础，知道了重构的分类，还实践了所有这些重构。同时，你已经很擅长测试了，所以你不再畏首畏尾。于是你可能想：「我已经知道如何重构了。」不，还没有。

前面列出的技术仅仅是一个起点，是你登堂入室之前的大门。如果没有这些技术，你根本无法对运行中的程序进行任何设计上的改动。有了这些技术，你仍然做不到，但起码可以开始尝试了。

这些技术如此精彩，可它们却仅仅是个开始，这是为什么？答案很简单：因为你还不知道何时应该使用它们、何时不应该使用、何时开始、何时停止、何时前进、何时等待。使重构能够成功的，不是前面各自独立的技术，而是这种节奏。

当你真正懂得这一切时，你又是怎么知道的呢？当你开始冷静下来，你就会知道，自己已经真正「得道」了。那时候你将有绝对的自信：不论别人留下的代码多么杂乱无章，你都可以将它变好，好到足以进行后续的开发。

不过，大多数时候，「得道」的标志是：你可以自信地停止重构。在重构者的整场表演中，「停止」正是压轴大戏。一开始你为自己选择一个大目标，例如「去掉一堆不必要的 subclass」。然后你开始向着这个目标前进，每一步都走得小而坚定，每一步都有备份，保证能够回头。好的，你离目标愈来愈近，愈来愈近，现在只剩两个函数需要合并，然后就将大功告成。

就在此时，意想不到的事情发生了：你再也无法前进一步。也许是因为时间太晚，你太疲倦；也许是因为一开始你的判断就出错，实际上不可能去掉所有 subclass；也许是因为没有足够的测试来支持你。总而言之，你的自信灰飞烟灭了，你无法再自信满满地跨出下一步。你认为自己应该没把任何东西搞乱，但也无法确定。

这是该停下来的时候了。如果代码已经比重构之前好，那么就把它集成到系统中，发布你的成果。如果代码并没有变好，就果断放弃这些无用的工作，回到起始点。然后，为自己学到一课而高兴，为这次重构没能成功而抱憾。那么，明天怎么办？

明天，或者后天，或者下个月，甚至可能明年，灵感总会来的。为了等待进行一项重构的后一半所需的灵感，我最多曾经等过九个月。你可能会明白自己错在哪里，也可能明白自己对在哪里，总之都能使你想清楚下一个步骤如何进行。然后，你就可以像最初一样自信地跨出这一步。也许你羞愧地想：「我太笨了，竟然这么久都没想到这一步。」大可不必，每个人都是这样的。

这有点像在悬崖峭壁上的小径行走：只要有光，你就可以前进，虽然谨慎却仍然自信；但是一旦太阳下山，你就应该停止前进；夜晚你应该睡觉，并且相信明天早晨太阳仍旧升起。

这听起来似乎有点神秘而模糊，近乎清谈玄想。从感觉上来说，的确如此，因为这是一种全新的编程方式。当你真正理解重构之后，系统的整个设计对你来说，就像源码文件中的字符那样可以随心所欲地操控。你可以直接感受到整个设计，可以清楚看到如何将设计变得更灵活，也可以看到如何修改它：这里修改一点，于是这样表现；那里修改一点，于是那样表现。

但是，从另一个角度来说，这也并非那么地神秘而模糊。重构是一种可以学习的技术，你可以从本书读得并学习它的各个组成。然后，只要把这些技术集成在一起并使之完善，就可以从一个全新角度看待软件开发。

正如我所说，这是一种可以学习的技术。那么，应该如何学习呢？

- **随时挑一个目标。**某个地方的代码开始发臭了，你就应该将问题解决掉。你应该朝目标前进，达成目标后就停止。你之所以重构，不是为了探索真善美（至少不全是），而是为了让你的系统更容易被人理解，为了防止程序变得散乱。

- **没把握就停下来。**朝目标前进的过程中，可能会有这样的时候：你无法证明自己所做的一切能够保持程序原本的语义。此时你就应该停下来。如果代码已经改善了一些，就发布你的成果；如果没有，就撤销所有修改。

- **学习原路返回。**重构的原则不好学，而且很容易遗失准头。就连我自己，也经常忘记这些原则。我有时会连续做两、三项甚至四项重构，而没有每次执行测试用例（test cases）。当然那是因为我完全相信，即使没有测试的帮助，我也不会出错。于是我就放手干了。然后，「砰」的一声，某个测试失败，我却无法找到究竟哪一次修改造成了这个问题。

这时候你一定很愿意就地调试，试图从麻烦中脱身。毕竟，不管怎么说，一开始所有测试都能够正常运行，现在要让它们再次正常运行，会困难到哪里去？停！你的重构已经失控了，如果继续向前走，你根本不可能知道如何夺回控制权。你应该回到最近一个没有出错的状态，然后逐一重复刚才做过的重构项，每次重构之后一定要运行所有测试。

站着说话不腰疼，以上一切听起来似乎显而易见。当你出错的时候，使系统极大简化的一个方案也许已经近在咫尺，这时候要你停下来回到起点，不啻是最痛苦的事情。但是，现在，趁你头脑还清楚的时候，请想一想：如果你第一次重构用了一小时，重复它只需十分钟就够了，所以如果你退回原点，十分钟之内一定能够再次达到现在的进度。但如果你继续前进，调试所需时间也许是五秒种，也许是两小时。

当然，我现在说这些，也是看人挑担不吃力，实际做起来困难得多。我个人曾经因为没有遵循这条建议，花了四个小时进行三次尝试。我失控、放弃、慢慢前进、再次失控、再重复……真是痛苦的四个小时。这不是有趣的事，所以你需要帮助。

■ **二重奏**。和别人一起重构，可以收到更好的效果。两人结伴，对于任何一种软件开发都有很多好处，对于重构也不例外。重构时，小心谨慎按部就班的态度是有好处的。如果两人结伴，你的搭档能够帮助你一步一步前进，你也能够帮助他（她）。重构时，时刻留意远景目标是有好处的。如果两人结伴，你的搭档可能看到你没看到的东西，能想到你没想到的事情。重构时，明智结束是有好处的。如果你的搭档不知道你在干什么，那就意味你肯定也不知道自己在干什么，此时你就应该结束重构。最重要的是，重构时，拥有绝对自信是绝对有好处的。如果两人结伴，你的搭档能够给你温柔的鼓励，让你不致于灰心丧气。

与搭档协同工作的另一方面就是交谈。你必须讲出你所想做的事，这样你们两个才能朝着同一个方向努力。你得把你正在做的事情讲出来，这样你的搭档才有可能指出你的错误。你得把刚才做过的事情讲出来，这样下次遇到同样情况时你才能做得更好。所有这些交谈都有助于你更清楚了解如何让个别的重构项适应整个重构节奏。

即使你已经在你的重构目标（代码）中工作了好几年，一丝一缕了然于胸，但只要发现其中臭味，以及消除臭味的重构手法，你就有可能看到程序的另一种可能。你也许会想立刻挽起袖子，把你看到的所有问题都解决掉。不，不要这么莽撞。没有一位经理愿意听到他的开发成员说「我们要停上三个月来清理以前的代码」。而且

开发人员本来也就不应该这样做。大规模的重构只会带来灾难。

你面前的代码也许看起来混乱极了，不要着急，一点一点慢慢地解决这些问题。当你想要添加新功能时，用上几分钟时间把代码整理一下。如果首先添加一些测试能使你对整理工作更有信心，那就那么做，它们会回报你的努力。如果在添加新代码之前进行重构，那么添加新代码的风险将大大降低。重构可以使你更好理解代码的作用和工作方式，这使得新功能的添加更容易。而且重构之后代码的质量也会大大提高，下次你再有机会处理它们的时候，肯定会对目前所做的重构感到非常满意。

永远不要忘记「两顶帽子」。重构时你总会发现某些代码并不正确。你绝对相信自己的判断，因此想马上把它们改正过来。啊，顶住诱惑，别那么做。重构时你的目标之一就是保持代码的功能完全不变，既不多也不少。对于那些需要修改的东西，列个清单把它们记录下来（通常我在计算器旁边放一张索引卡），需要添加或修改的测试用例（test cases）、需要进行的其他重构、需要撰写的文档、需要画的图……都暂时记在卡上。这样就不会忘掉这些需要完成的工作。千万别让这些工作打乱你手上的工作。重构完成之后，再去做这些事情不迟。

参考书目
References

[Auer]

Ken. Auer "Reusability through Self-Encapsulation." In *Pattern Languages of Program Design* 1, Coplien J.O. Schmidt.D.C. Reading, Mass.: Addison-Wesley, 1995.

一篇有关于「自我封装」（self-encapsulation）概念的模式论文（patterns paper）。

[Bäumer and Riehle]

Bäumer, Riehle and Riehle. Dirk "Product Trader." In *Pattern Languages of Program Design 3*. R. Martin F. Buschmann D. Riehle. Reading, Mass.: Addison-Wesley, 1998.

一个模式（patterns），用来灵活创建对象而不需要知道对象隶属哪个 class。

[Beck]

Kent.Beck *Smalltalk Best Practice Patterns*. Upper Saddle River. N.J.: Prentice Hall. 1997a.

一本适合任何 Smalltalker 的基本书籍，也是一本对任何面向对象开发者很有用的书籍。谣传有 Java 版本。

[Beck, hanoi]

Kent.Beck "Make it Run, Make it Right: Design Through Refactoring."*The Smalltalk Report*, 6: (1997b): 19-24.

第一本真正领悟「重构过程如何运作」的出版品，也是《*Refactoring*》第一章许多构想的源头。

[Beck, XP]

Kent.Beck *eXtreme Programming eXplained: Embrace Change. Reading*, Mass.: Addison-Wesley, 2000.

[Fowler, UML]

FowlerM.Scott.K. *UML Distilled, Second Edition: A Brief Guide to the Standard Object Modeling Language*. Reading, Mass.: Addison-Wesley, 2000.

一本简明扼要的导引，助你了解《*Refactoring*》书中各式各样的 Unified Modeling Language（UML，统一建模语言）图片。

[Fowler, AP]

M.Fowler *Analysis Patterns: Reusable Object Models*. Reading, Mass.: Addison-Wesley, 1997.

一本 domain model patterns 专着。包括对 range pattern 的一份讨论。

[Gang of Four]

E. Gamma. R. Helm, R. Johnsonand J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.

或许是面向对象设计（object-oriented design）领域中最有价值的一本书。现今几乎任何人都必须语带智慧地谈点 strategy、singleton 和 chain of responsibility，才敢说自己懂得对象（技术）。

[Jackson, 1993]

Jackson, Michael. *Michael Jackson's Beer Guide*, Mitchell Beazley, 1993.

一本有用的导引，提供大量实践（实用性）研究。

[Java Spec]

Gosling, James, Bill Joy and Guy Steele. *The Java Language Specification, Second Edition*. Boston, Mass.: Addison-Wesley. 2000.

所有 Java 问题的官方答案。

[JUnit]

Beck, Kent, and Erich Gamma. JUnit Open-Source Testing Framework. Available on the Web (http://www.junit.org).

撰写 Java 程序的基本应用工具。是个简单框架（framework），帮助你撰写、组织、运行单元测试（unit tests）。类似的框架也存在于 Smalltalk 和 C++ 中。

[Lea]

Doug.Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Reading, Mass.: Addison-Wesley, 1997.

编译器应该阻止任何人实现 Runnable 接口　　如果他没有读过这本书。

[McConnell]

Steve. McConnell, *Code Complete: A Practical Handbook of Software Construction*. Redmond, Wash.: Microsoft Press, 1993.

一本对于编程风格和软件建构的卓越导引。写于 Java 诞生之前，但几乎书中的所有忠告都适用于 Java。

[Meyer]

Bertrand. Meyer, *Object Oriented Software Construction*. 2 ed. Upper Saddle River, N.J.: Prentice Hall, 1997.

面向对象设计（object-oriented design）领域中一本很好（也很庞大）的书籍，其中包括对于契约式设计（design by contract）的一份彻底讨论。

[Opdyke]

William F. Opdyke. Ph.D. diss., "Refactoring Object-Oriented Frameworks. " University of Illinois at Urbana-Champaign, 1992.

参见 ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z。是关于重构的第一份体面长度的著作。多少带点教育和工具导向的角度（毕竟这是一篇博士论文），对于想更多了解重构理论的人，是很有价值的读物。

[Refactoring Browser]

Brant, John, and Don Roberts. Refactoring Browser Tool.

http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser。未来的软件开发工具。

[Woolf]

Bobby. Woolf, "Null Object." In *Pattern Languages of Program Design* 3. Martin, R. Riehle. D. Buschmann F. Reading, Mass.: Addison-Wesley, 1998.

针对 null object pattern 的一份讨论。

原音重现

List of Soundbites

p.7 *When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.*

如果你发现自己需要为程序添加一个特性，而代码结构使你无法很方便地那么做，那就先重构那个程序，使特性的添加比较容易进行，然后再添加特性。

p.8 *Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking.*

重构前，先检查自己是否有一套可靠的测试机制。这些测试必须有自我检验能力。

p.13 *Refactoring changes the programs in small steps. If you make a mistake, it is easy to find the bug.*

重构技术系以微小的步伐修改程序。如果你犯下错误，很容易便可发现它。

p.15 *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

任何一个傻瓜都能写出计算器可以理解的代码。惟有写出人类容易理解的代码，才是优秀的程序员。

p.53 ***Refactoring** (noun) : a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior of the software.*

重构（名词）：对软件内部结构的一种调整，目的是在不改变「软件之可察行为」前提下，提高其可理解性，降低其修改成本。

p.54 ***Refactor** (verb) : to restructure software by applying a series of refactorings without changing the observable behavior of the software.*

重构（动词）：使用一系列重构准则（手法），在不改变「软件之可察行为」前提下，调整其结构。

p.58 *Three strikes and you refactor.*

事不过三，三则重构。

p.65 *Don't publish interfaces prematurely. Modify your code ownership policies to smooth refactoring.*

不要过早发布接口。请修改你的代码拥有权政策，使重构更顺畅。

p.88 *When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.*

当你感觉需要撰写注释，请先尝试重构，试着让所有注释都变得多余。

p.90 *Make sure all tests are fully automatic and that they check their own results.*

确保所有测试都完全自动化，让它们检查自己的测试结果。

p.90 *A suite of tests is a powerful bug detector that decapitates the time it takes to find bugs.*

一整组测试就是一个强大的臭虫侦测器，能够大大缩减查找臭虫所需要的时间。

p.94 *Run your tests frequently. Localize tests whenever you compile - every test at least every day.*

频繁地运行测试。每次编译请把测试也考虑进去——每天至少执行每个测试一次。

p.97 *When you get a bug report, start by writing a unit test that exposes the bug.*

每当你接获臭虫提报，请先撰写一个单元测试来揭发这只臭虫。

p.98 *It is better to write and run incomplete tests than not to run complete tests.*

编写未臻完善的测试并实际运行，好过对完美测试的无尽等待。

p.99 *Think of the boundary conditions under which things might go wrong and concentrate your tests there.*

考虑可能出错的边界条件，把测试火力集中在那儿。

p.100 *Don't forget to test that exceptions are raised when things are expected to go wrong.*

当事情被大家认为应该会出错时，别忘了检查彼时是否有异常被如期抛出。

p.101 *Don't let the fear that testing can't catch all bugs stop you from writing the tests that will catch most bugs.*

不要因为「测试无法捕捉所有臭虫」，就不撰写测试代码，因为测试的确可以捕捉到大多数臭虫。

索引

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W