《重构》

Refactoring - Improving the Design of Existing Code 中文版

■敬告读者

此处开放《重构》最后定稿之部分篇幅。此举得到 繁体版出版人碁峰图书公司 与简体版出版人中国电力出版社 之鼎力支持，十分感谢。

本次开放第 6 章（含）前所有内容及书后索引，达全书 1/3+ 篇幅。开放之 PDF 含标签（目录连结；只达「章」层级），请打开 PDF reader 之「导引框」，便可见到如下画面：

请注意，标签（目录）完整，但内容只有前 6 章及索引。

本次开放以 繁/简中文 读者为对象。由于我个人并不直接处理简体版最终版面工作，因此手上无简体版最终电子成品。我所开放的两份成品，都使用繁体字，但区分为「台湾术语版」和「大陆术语版」。

您目前所见到的这一份成品，是「大陆术语版」。enjoy it

侯捷.2003/07/12

软 件 工 程 系 列

重构——改善既有代码的设计（中文版）

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

写《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

重构——改善既有代码的设计
（中文版）
侯捷 熊节 译著
中国电力出版社

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。与此同时，软件系统专家发现，与这些设计下来的，质量不佳的程序共处，是多么艰难。对象专家运用许多（而且日益更多）技术来改善既有程序的结构完善性与性能，已有数年之久。如今这些技术被称为“重构”（refactoring）的实践技术。一直（只）流传于专家领域内，因为没有人愿意将坐那这些知识落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，揭去那些名家高数手法的神秘面纱，展示软件从业人员如何透过种种过程的重大意义。

只要受过适度训练，一位技巧纯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、容易维护的代码。本书之中，Martin Fowler 指引你面向优秀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——简单到了你乎不值得去做的程度。重构除及将值域（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），或若呈将某些代码上下移动于继承体系（hierarchy）之中。这些个别步骤虽简可能十分基本，有属下来的影响却能够彻底改善设计。重构已经被证明可以防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 个已被证用效果的重构手法，以及宜用和勿用的重点，都导你实施的时机，实际时的逐步指令，并各自携带一个例子，显示重构的运转。这份宝有价值的实例都以 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 10 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 10 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、各所技术领域之 Framework Libraries 有浓厚兴趣和钻研。

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，惟愿宁静淡泊终已，夜阑人静，一杯清水，几本闲书，神交千各方名士，献绩于天下同好，百愿足矣。

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

责任编辑：程 旗 任 昌
封面设计：王红卿

ISBN 7-5083-1554-5

For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macao SAR)
仅限于中华人民共和国境内（不包括中国香港、澳门特别行政区和中国台湾地区）销售发行。

www.PearsonEd.com

ISBN 7-5083-1554-5
定价：68.00 元

Refactorings（重构）列表

Add Parameter（添加参数） 275
Change Bidirectional Association to Unidirectional（将双向关联改为单向） 200
Change Reference to Value（将引用对象改为实值对象） 183
Change Unidirectional Association to Bidirectional（将单向关联改为双向） 197
Change Value to Reference（将实值对象改为引用对象） 179
Collapse Hierarchy（折迭继承体系） 344
Consolidate Conditional Expression（合并条件式） 240
Consolidate Duplicate Conditional Fragments（合并重复的条件片段） 243
Convert Procedural Design to Objects（将过程化设计转化为对象设计） 368

重构

— 改善既有代码的设计 —

Refactoring

Improving the Design of Existing Code

Martin Fowler 着

（以及 Kent Beck, John Brant , William Opdyke,
Don Roberts 对最后三 章的贡献）

侯捷 / 熊节 合译

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，藉由搬移、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。过去或许不得不暴虎凭河，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 其实不敢置信有谁能够乖乖地 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，你也可以藉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 。

Refactoring – Improving the Design of Existing Code

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。为求参考、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）。这么做的考虑是，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 的考虑。「*static* 值域与 *instance* 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，熊节居于首功 — 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 年的着译历程中，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（着译皆含）从早期盈盈满满的初阶语言用书，逐渐进化到中高级语言用书、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惘然。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。

目录

Contents

序言

by Erich Gamma

重构（**refactoring**）这个概念来自 Smalltalk 圈子，没多久就进入了其它语言阵营之中。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.

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 对自己的运作方式做了一些假设，这些假设被嵌入并被继承下去。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 或使用其它 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它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，促进了我写这本书的动机，如此一来我就能够把 Kent 和其它一些人已经学会的「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，你也可以藉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。

本书介绍了其它人的许多研究成果，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它语言中实现，而且我也希望这本书能够给其它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，告诉读者如何在其它语言中进行重构，不过我真心希望看到其它人在本书基础上针对其它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它语言改写。

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 也在其它地方帮助我，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 编撰或合着了本书部分章节。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持和帮助。

谈到支持，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。感谢妳，当我埋首工作的时候，妳还是一样爱我。当我投入书中，总会不断想起妳。

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

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。如果我选择一个大型程序，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，一边想象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。

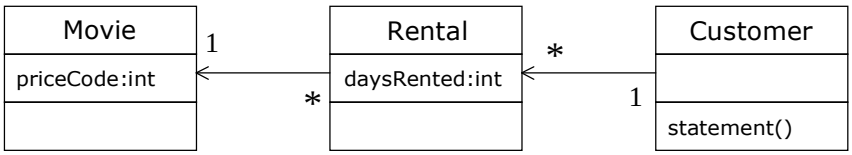


图 1.1 本例一开始的各个 classes。此图只显示最重要的特性。图中所用符号是 UML（Unified Modeling Language，统一建模语言，[Fowler, UML]）。

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。就像其它 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。

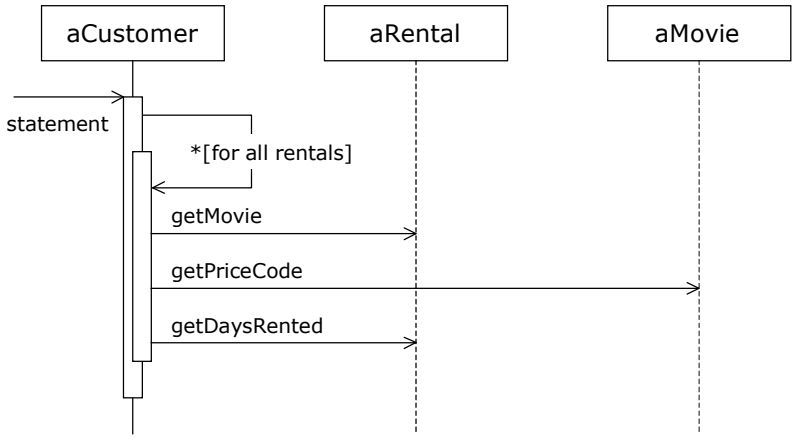


图 1.2 statement() 的交互过程（interactions）

```
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它 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，第三个变化来了：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。花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。藉着这个意外疏忽，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。

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;

    }
```

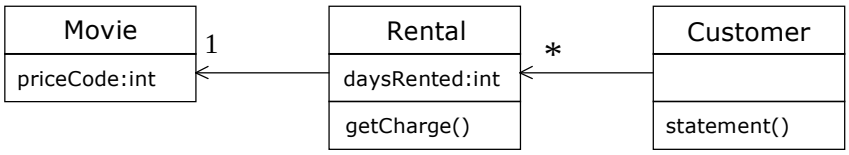


图 1.3　搬移「金额计算」函数后，所有 classes 的状态（*state*）

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，而我又不想修改其它 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，而它可以被当做参数传入新函数中。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。

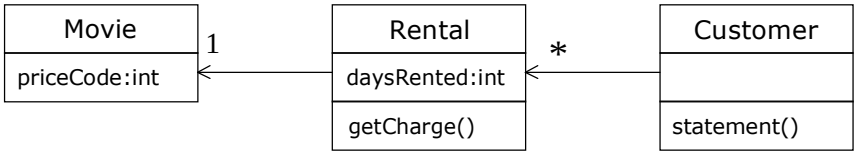


图 1.4 「常客积点计算」函数被提炼及搬移之前的 class diagrams

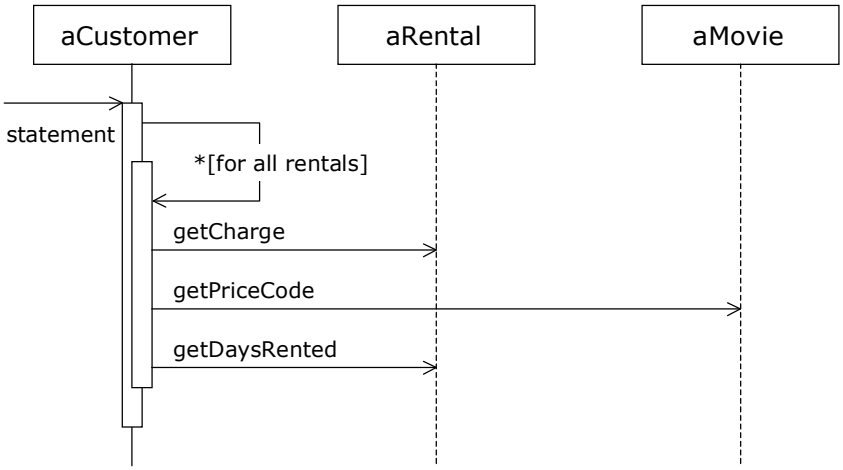


图 1.5 「常客积点计算」函数被提炼及搬移之前的 sequence diagrams

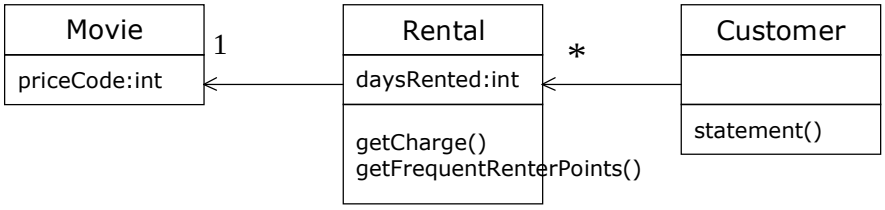


图 1.6 「常客积点计算」函数被提炼及搬移之后的 class diagrams

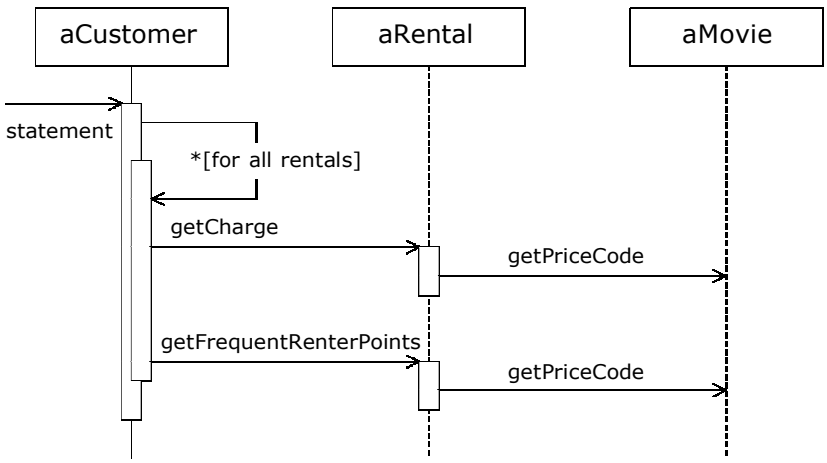


图 1.7 「常客积点计算」函数被提炼及搬移之后的 sequence diagrams

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 method***s，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
   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。

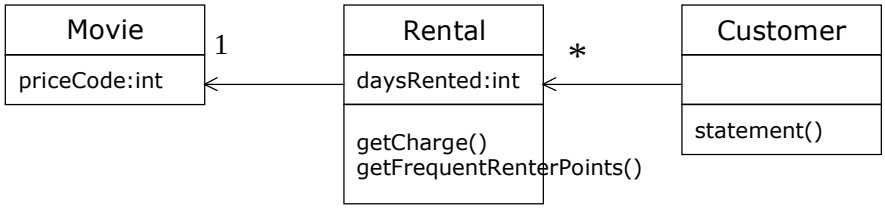


图 1.8 「总量计算」函数被提炼前的 class diagram

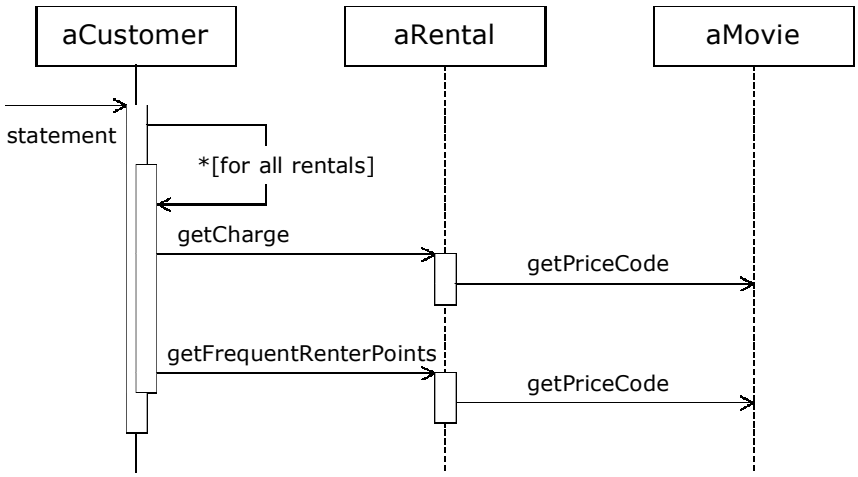


图 1.9 「总量计算」函数被提炼前的 sequence diagram

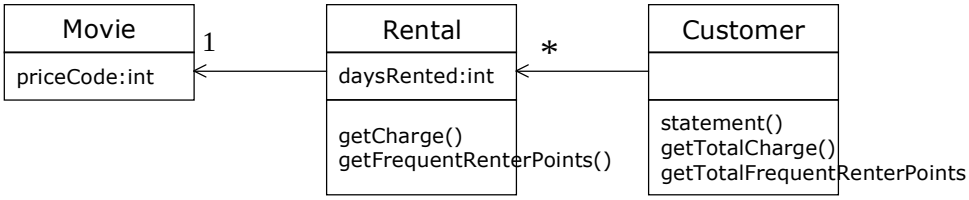


图 1.10 「总量计算」函数被提炼后的 class diagram

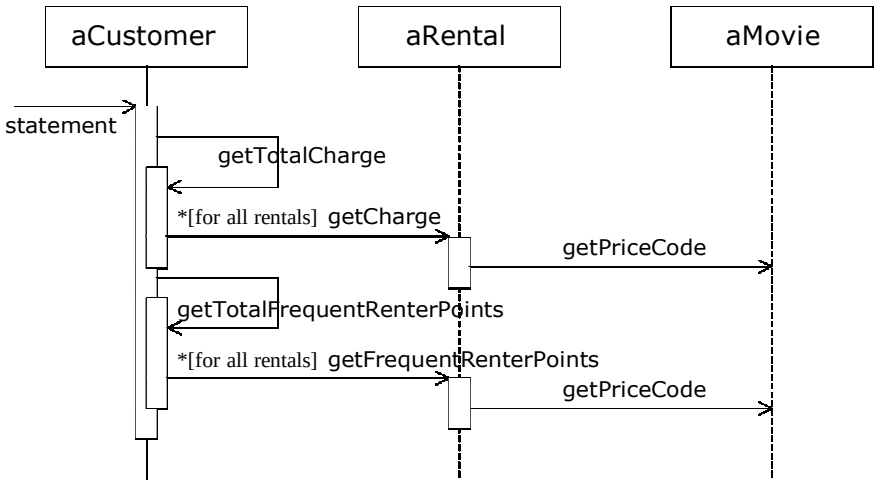


图 1.11 「总量计算」函数被提炼后的 sequence diagram

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，周边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***，其它函数就必须了解 `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。完成其它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;
 }
```

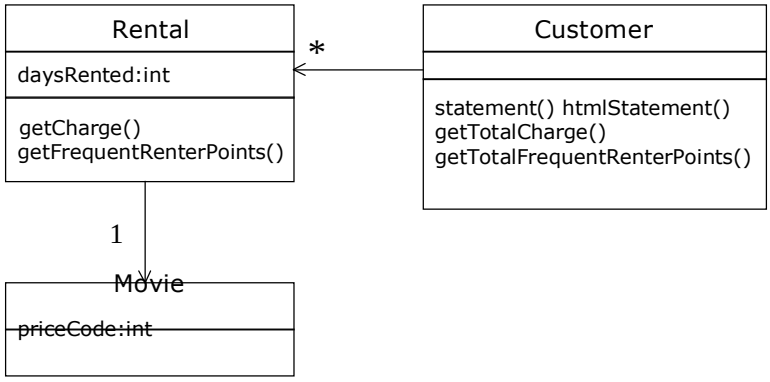


图 1.12 本节所讨论的两个函数被移到 `Movie` class 内之前系统的 class diagram。

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;
  }
```

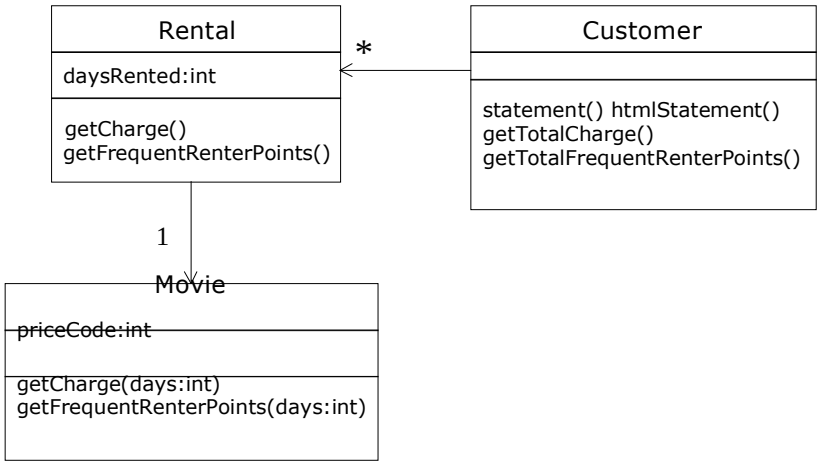


图 1.13　本节所讨论的两个函数被移到 `Movie` class 内之后系统的 class diagram。

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）。

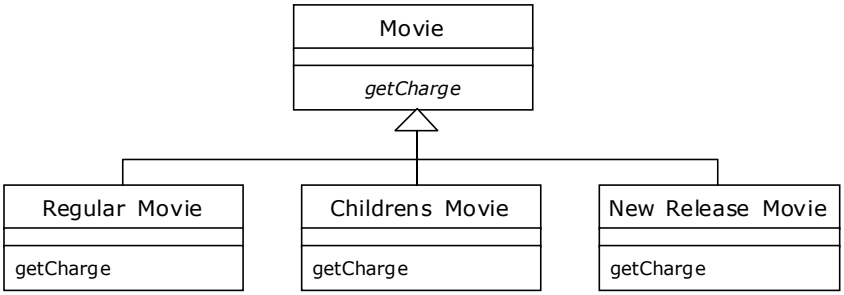


图 1.14 以继承机制表现不同的影片类型

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。

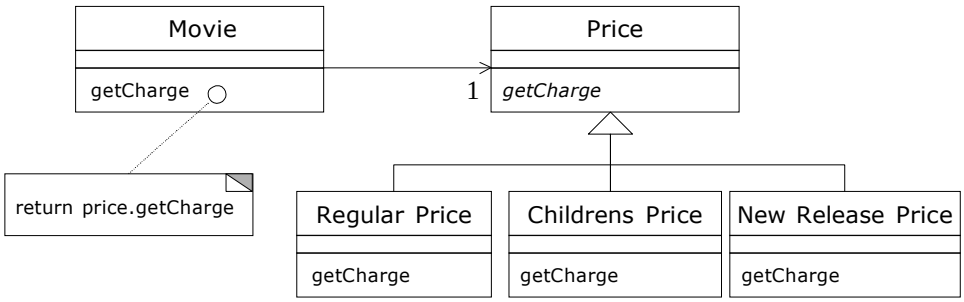


图 1.15 运用 ***State*** pattern（模式）表现不同的影片

加入这一层间接性，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它 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，或是加入其它取决于价格的行为，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。

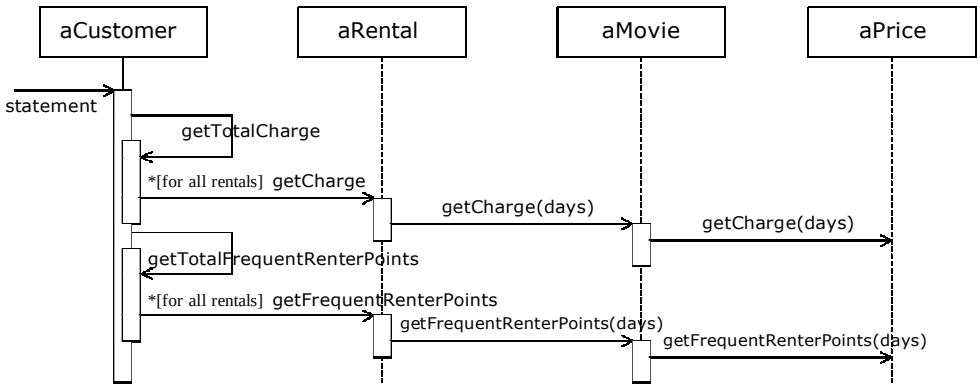


图 1.16　运用 ***State*** pattern（模式）当时的 interaction diagram

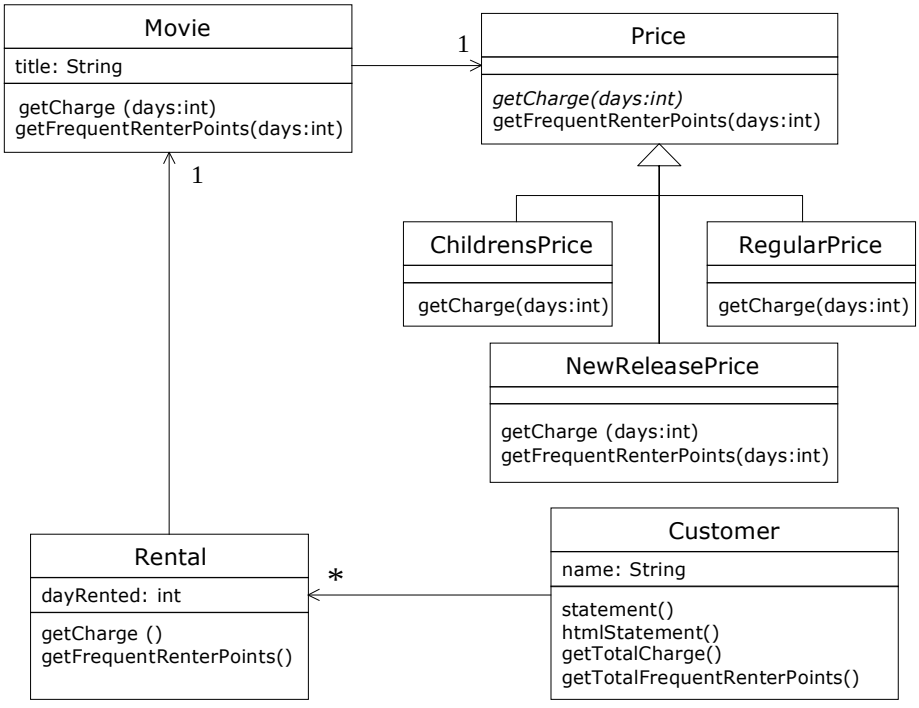


图 1.17　加入 ***State*** pattern（模式）之后的 class diagram

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 团队和其它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，便是与计算器交谈。你编写代码告诉计算器做什么事，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」。除了计算器外，你的源码还有其它读者：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。计算器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象出来。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。我不必想象代码「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*

『计算器科学是这样一 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
>
> *Refactoring – Improving the Design of Existing Code*

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（published）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「先想象重构的情况」。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。做出这种决定很困难，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作出「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一部分债务。

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 都把设计看作软件开发的关键环节，而把编程（programming）看作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（克赖斯勒综合薪资系统）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

Refactoring – Improving the Design of Existing Code

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：我们了如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，你常会作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，其它任何情况下「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。

第三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。本书中我把重构看作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/连接/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 之外的其它语言的可能性。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（克赖斯勒综合薪资系统）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）计算器上，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）将其它函数的算法替换掉。

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它任何地方出现。

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做参数，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一些。

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，那么你应该启动重构准则 *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，只在该处作修改。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。

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 内作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。这种孺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儿。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。其它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 — 这两个型别在其它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，其它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。实行这种方式，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，客户端就不得不作出相应修改。

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它 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 还有其它行为，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 对另一个斩断情思。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」，它们几乎一定被其它 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。对于那些不该被其它 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）被其它 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，我大可让计算器来帮我做这件事。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。

和本书其它内容一样，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 文檔。

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），或其它 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。

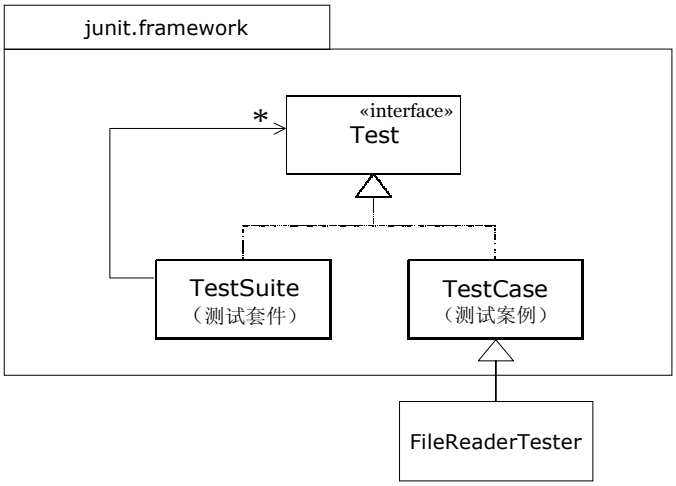


图 4.1 测试框架的 *Composite* 结构

```
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('2' == 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。

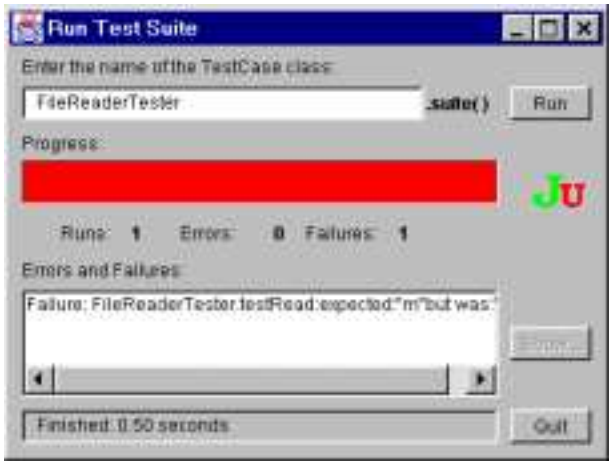


图 4.2 JUnit 的图形用户界面

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。它能够测试其它 packages 的接口，除此之外它将假设其它 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，或如果出现其它相关失败（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。功能测试往往以其它工具辅助进行，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，俾能够比单纯使用 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：原出现于 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()throwsIOException
    { 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「由其它 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它测试套件。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。这份名录并非巨细靡遗，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它地方使用的名称。

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**）包括三个部分：(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，最大限度地避免其它枝节，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它问题，但修正那些问题需要用到其它重构手法。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，记得寻求计算器的帮助。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器代码当做文本文件来处理，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，检查是否有其它 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记录。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它人。

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「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，当做参数传给目标函数。

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做参数传给目标函数。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，你才必须采取其它措施。

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它处理，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它所有函数，「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，而它妨碍了其它重构手法。

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它的重构手法 — 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」。新函数可被其它函数使用。

```
   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它条件影响。其它情况比较棘手，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。和其它性能问题一 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，并且可被其它对象使用。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，处理表达式的其它部分。

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；如果其它对象也需要它们，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，会令代码阅读者胡涂。

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它重构手法。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胡涂。

另一个让人胡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 final 关键词，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胡涂。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。

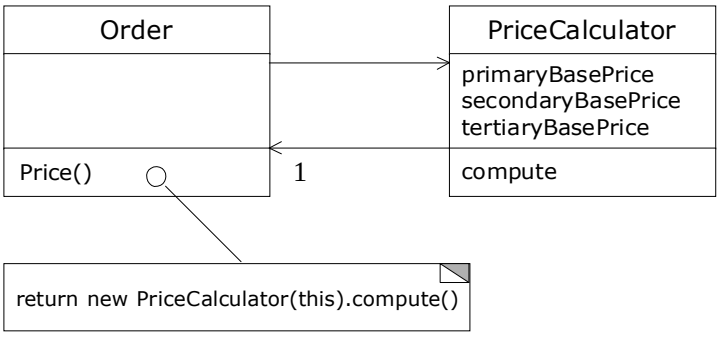
Java 1.1 及其后版本，允许你将参数标示为 `final`，从而避免函数中对参数赋值。即使某个参数被标示为 final，你仍然可以修改它所指向的对象。我总是把参数视为 final，但是我得承认，我很少在参数列（parameter list）中这样标示它们。

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，使你无法实行 *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（好酒沉瓮底呢），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），不必担心自变量（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。

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 专著。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

N

O

P

Q

R

S

T

U

V

W

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*
Inappropriate Intimacy, p85	*Move Method* (142), *Move Field* (146), *Change Bidirectional Association to Unidirectional* (200), *Replace Inheritance*
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),
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*
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