目錄

Introduction 1.1

序言 1.1.1

第一章 重构，第一个案例 1.2

起点 1.2.1

重构的第一步 1.2.2

分解并重组statement () 1.2.3

运用多态取代与价格相关的条件逻辑 1.2.4

结语 1.2.5

第二章 重构原则 1.3

何为重构 1.3.1

为何重构 1.3.2

何时重构 1.3.3

怎么对经理说 1.3.4

重构的难题 1.3.5

重构与设计 1.3.6

重构与性能 1.3.7

重构起源何处 1.3.8

第三章 代码的坏味道 1.4

重复代码 1.4.1

过长函数 1.4.2

过大的类 1.4.3

过长参数列 1.4.4

发散式变化 1.4.5

霰弹式修改 1.4.6

依恋情结 1.4.7

数据泥团 1.4.8

基本类型偏执 1.4.9

switch 惊悚现身 1.4.10

平行继承体系 1.4.11

冗赘类 1.4.12

夸夸其谈未来性 1.4.13
令人迷惑的暂时字段 1.4.14
过度耦合的消息链 1.4.15
中间人 1.4.16
狎昵关系 1.4.17
异曲同工的类 1.4.18
不完美的库类 1.4.19
纯稚的数据类 1.4.20
被拒绝的遗赠 1.4.21
过多的注释 1.4.22
第四章 构筑测试体系 1.5
自测试代码的价值 1.5.1
JUnit测试框架 1.5.2
添加更多测试 1.5.3
第五章 重构列表 1.6
重构的记录格式 1.6.1
寻找引用点 1.6.2
这些重构手法有多成熟 1.6.3
第六章 重新组织函数 1.7
Extract Method 1.7.1
Inline Method 1.7.2
Inline Temp 1.7.3
Replace Temp with Query 1.7.4
Introduce Explaining Variable 1.7.5
Split Temporary Variable 1.7.6
Remove Assignments to Parameters 1.7.7
Replace Method with Method Object 1.7.8
Substitute Algorithm 1.7.9
第七章 在对象之间搬移特性 1.8
Move Method 1.8.1
Move Field 1.8.2
Extract Class 1.8.3
Inline Class 1.8.4
Hide Delegate 1.8.5

Remove Middle Man 1.8.6
Introduce Foreign Method 1.8.7
Introduce Local Extension 1.8.8
第八章 重新组织数据 1.9
Self Encapsulate Field 1.9.1
Replace Data Value with Object 1.9.2
Change Value to Reference 1.9.3
Change Reference to Value 1.9.4
Replace Array with Object 1.9.5
Duplicate Observed Data 1.9.6
Change Unidirectional Association to Bidirectional 1.9.7
Change Bidirectional Association to Unidirectional 1.9.8
Replace Magic Number with Symbolic Constant 1.9.9
Encapsulate Field 1.9.10
Encapsulate Collection 1.9.11
Replace Record with Data Class 1.9.12
Replace Type Code with Class 1.9.13

Notes for Refactoring Improving the Design of Existing Code(Chinese Language)

This book mainly writes the notes about the book,Refactoring Improving the Design of Existing Code,and it's in Chinese.

《重构——改善既有代码的设计》笔记（中文）

重构的重新认识

基本定义：重构是在不改变软件可观察行为的前提下改善其内部结构。

重构的生活方式

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统一的代码，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，以XP那敏捷而又严谨的方法论演绎了重构的美。

序

“重构”这个概念来自Smalltalk圈子，没多久就进入了其他语言阵营之中。由于重构是框架开发中不可缺少的一部分，所以当框架开发人员讨论自己的工作时，这个属于就诞生了。当他们提炼自己的类继承体系时，但他们叫喊自己可以拿掉多少行代码时，它将随着设计者的经验成长而进化；他们也知道，代码被阅读和别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——对框架而言如此，对一般软件也如此。

好极了，还有什么问题吗？问题很显然：重构具有风险。它必须修改运作中的程序，这可能映入一些不易察觉的错误。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《设计模式》书中和另外三位作者曾经提过：设计模式为重构提供了目标。然而“确定目标”只是问题的一部分而已，改造程序已达到目标，是另一个难题。

前言

什么是重构

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，对代码作出修改，以改进程序的内部结构。重构是一种经千锤百炼形成的有条不紊的程序整理方法，可以最大限度地减少整理过程中引入错误的几率。本质上说，重构就是在代码写好之后改进它的设计。

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：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，编码工作从严谨的工程堕落为胡砍乱劈的随行行为。

“重构”正好与此相反。哪怕你手上有一个糟糕的设计，甚至是一堆混乱的代码，你也可以借由重构把它加工成设计良好的代码。重构的每个步骤都很简单，甚至显得有些过于简单：你只需要把某个字段从一个类移到另一个类，把某些代码冲一个函数拉出来构成另一个函数，或是在继承体系中把某些代码推上推下就行了。但是，聚沙成塔，这些小小的修改累计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。

本书有什么

第1章展示了一个小程序，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，以及几个很有用的重构手法。如果你想知道重构到底是怎么回事，这一章不可不读。

第2章讨论重构的一般性原则、定义，以及进行重构的原因，我也大致介绍了重构所存在的一些问题。

第3章由KentBeck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。测试在重构中扮演着非常重要的角色，第4章介绍如何运用一个简单而且开源的Java测试框架，在代码中构筑测试环境。

本书的核心部分—重构列表—从第5章延伸至第12章。它不能说是一份全面的列表，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手法。每当我想做点什么——例如Replace Conditional with Polymorphism (255)——的时候，这份列表就会提醒我如何一步一步安全前进。我希望这是值得你日后一再回顾的部分。本书介绍了其他人的许多研究成果，最后几章就是由他们之中的几位所客串写就的。

在Java中运用重构

为了很好地与读者交流我的想法，我没有使用Java语言中特别复杂的部分。所以我避免使用内嵌类、反射机制、线程以及很多强大的Java特性。这是因为我希望尽可能清楚地展现重构的核心。

我应该提醒你，这些重构手法并不针对并发或分布式编程。那些主题会引出更多的考虑，本书并未涉及。

谁该阅读本书

本书的目标读者是专业程序员，也就是那些以编写软件为生的人。书中的示例 和讨论，涉及大景需要详细阅读和理解的代码。

下面我要告诉你，如何能够在不通读全书的情况下充分用好它。

- 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的过程。
- 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”。
- 如果你想着手进行重构，请完整阅读前四章，然后选择性地阅读重构列表。一开始只需概略浏览列表，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，从中获取帮助。列表部分是供查阅的参考性内容，你不必一次就把它全部读完。此外你还应该读一读列表之后其他作者的“客串章节”，特别是第15章。

重构，第一个案例

起点

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达成目的，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。

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，因为尽管遵循重构手法可以使我避免绝大多数引入bug的情形，但我毕竟是人，毕竟有可能犯错。所以我需要可靠的测试。

进行重构的时候，我们需要依赖测试，让它告诉我们是否引入了bug。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。

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能力。

分解并重组statement ()

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心。如果只有一个变最会被修改，我可以把它当作返回值。

重构技术就是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。

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，那么就大胆去做吧。

运用多态取代与价格相关的条件逻辑

结语

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，将迥异于过程化风格——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，就很难再满足于结构化风格了。

这个例子给我们最大的启发是重构的节奏：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。

第二章

前面所举的例子应该已经让你对重构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。

何为重构

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。

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可观察行为的前提下，提高其可理解性，降低其修改成本。

重构（动词）：使用一系列重构手法，在不改变软件可观察行为的前提下，调整其结构。

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，但必须对软件可观察的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观察的行为——重构之后软件功能一如以往。任何用户，不论最终用户或其他程序员，都不知道已经有东西发生了变化。

两顶帽子

上述第二点引出了 KentBedc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，以及重构。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（除非发现有先前遗漏的东西），只在绝对必要（用以处理接口变化）时才修改测试。

为何重构

重构是个工具，它可以（并且应该）用于以下几个目的。

重构改进软件设计

重构很像是在整理代码，你所做的就是让所有东西回到应处的位置上。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图，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。

改进设计的一个重要方向就是消除重复代码。这个动作的重要性在于方便未来的修改。代码最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。然而代码量减少将使未来可能的程序修改动作容易得多。

重构使软件更容易理解

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“准确说出我所要的”。

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。每当看到不熟悉的代码，我必须试着理解其用途。

重构帮助找到bug

如果对代码进行重构，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。

重构提高编程速度

我绝对相信：良好的设计是快速开发的根本——事实上，拥有良好设计才可能做到快速开发。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，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。

何时重构

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应该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

DonRoberts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，但无论如何还是可以去做；第三次再做类似的事，你就应该重构。

事不过三，三则重构。

添加功能时重构

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，重构的直接原因 往往是为了帮助我理解需要修改的代码。

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。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，我用重构帮助加深自己的理解。我发现以这种程序来处理代码，常常能够帮助我找出bug。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——没有清晰到让你能一眼看出bug。

复审代码时重构

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。

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-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，而和单个复审者进行代码复审。

极限编程[Beck, XP] 中的“结对编程”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？——Kent Beck

是什么让程序如此难以相与？眼下我能想起下述四个原因，它们是：

- 难以阅读的程序，难以修改；
- 逻辑重复的程序，难以修改；
- 添加新行为时需要修改已有代码的程序，难以修改；
- 带复杂条件逻辑的程序，难以修改。

因此，我们希望程序：

1. 容易阅读；
2. 所有逻辑都只在唯一地点指定；
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。

重构是这样一个过程：它在一个目前可运行的程序上进行，在不改变程序行为的前提下使其具备上述美好性质，使我们能够继续保持高速开发，从而增加程序的价值。

怎么对经理说

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，我发现先重构再添加新功能会更快些。如果要修补错误，就得先理解软件的工作方式，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

“计算机科学是这样一门科学：它相信所有问题都可以通过增加一个间接层来解决。” —— Dennis DeBruler

重构往往把大型对象拆成多个小型对象，把大型函数拆成多个小型函数。 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两。如果某个对象委托另一对象，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。比如说一个子函数在两个不同的地点被调用，或超类中的某个函数被所有子类共享。
- 分开解释意图和实现。你可以选择每个类和函数的名字，这给了你一个解释自己意图的机会。类或函数内部则解释实现这个意图的做法。如果类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描述其结构中的大部分重要信息.
- 隔离变化。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。为此我做出一个子类，并在需要修改处引用这个子类。现在，我可以修改这个子类而不必承担无意中影响另一处的风险。
- 封装条件逻辑。对象有一种奇妙的机制：多态消息，可以灵活而清晰地表达条件逻辑。将条件逻辑转化为消息形式，往往能降低代码的重复、增加清晰度并提高弹性。

重构的难题

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，但与此同时，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

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数据库结构紧密耦合在一起，这也是数据库结构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，将数据库结构和对象模型间的依赖降至最低，但数据库结构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。

在非对象数据库中，解决这个问题的办法之一就是：在对象模型和数据库模型之间插入一个分隔层，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， 但可以给你带来很大的灵活度。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平衡点。

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景，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类中的数据结构变化。你仍然可以放心将类的行为转移过去，但转移字段时就必须格外小心。数据尚未被转移前你就得先运用访问函数造成“数据已经转移”的假象。一旦你确定知道数据应该放在何处，就可以一次性地将数据迁移过去。这时唯一需要修改的只有访问函数，这也降低了错误风险。、

修改接口

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和接口。 你可以安全地修改对象内部实现而不影响他人，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。

简言之，如果重构手法改变了已发布接口，你必须同时维护新旧两个接口，直到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。千万不要复制函数实现, 那会让你陷入重复代码的泥淖中难以自拔。你还应该使用Java提供的**deprecation (**不建议使用）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。

“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构造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：不要发布接口。。发布接口很有用，但也有代价。所以除非真有必要，不要发布接口。这吋能意味需要改变你的代码所有权观念，让每个人都可以修改别人的代码，以适应接口的改动。以结对编程的方式完成这一切通常是个好主意。

不要过早发布接口。请修改你的代码所有权政策，使重构更顺畅。

JavaiE有一种特别的接口修改：在**throws**子句中增加一个异常。这并不是对函数签名的修改，所以你无法以委托的办法隐藏它；但如果用户代码不做出相应修改， 编译器不会让它通过。

难以通过重构手法完成的设计变动

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，哪怕它不能覆盖有冇潜在需求也没关系。。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，后一种情况很少出现。

何时不该重构

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。

一个折中办法就是：将“大块头软件”重构为封装良好的小型组件。然后你就可以逐一对组件做出“重构或重建”的决定。这是一个颇有希望的办法，但我还没有足够数据，所以也无法写出好的指导原则。对于一个重要的遗留系统，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，而那个时候已经为时晚矣。

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。

重构与设计

重构肩负一项特殊使命：它和设计彼此互补。

有一种观点认为：重构可以取代预先设计。这意思是你根本不必做任何设汁，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。事实上这种办法眞的可行。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[Beck,XP]的支持者极力提倡这种办法。

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，你甚至吋以不必再预先思考前述所谓的灵活方案——旦需要它，你总有足够的信心去重构。

重构与性能

虽然重构可能使软件运行更慢，但它也使软件的性能优化更容易。除了对性能有严格要求的实时系统，其他任何情况下“编写快速软件”的秘密就是：首先写出可调的软件，然后调整它以求获得足够速度。

我看过二种编写快速软件的方法。其中最严格的是时间预算法，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。每个组件绝对不能超出自己的预兑，就算拥有组件之间调度预配时间的机制也不行。

第二种方法是持续关注法。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。任何修改如果楚为了提高性能，通常会使程序难以维护，继而减缓开发速度。

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，就会发现它把大半时间都耗费在一小半代码身上。

第三种性能提升法就是利用上述的90%统计数据。采用这种方法时，你编写构造良好的程序，不对性能投以特别的关注，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，你首先应该用一个度最工具来监控程序的运行，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所在的一小段代码。然后你应该集中关注这些性能热点，并使用持续关注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，较少的工作量便吋显现较好的成果。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、再次度童。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。

重构起源何处

第三章 代码的坏味道

我们并不试图给你一个何时必须重构的精确衡量标准。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“这里有一个可以用重构解决的问题”。你必须培养出自己的判断力，学会判断一个类内有多少实例变量算是太大、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，请阅读本章内容和内封页表格来寻找灵感。你可以阅读本章（或快速浏览内封页表格）来判断自己已闻到的是什么味道，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“坏味道条款”和你所检测的不尽相符，但愿它们能够为你指引正确方向。

重复代码 Duplicate Code

坏味道行列中首当其冲的就是DuplicatedCode。如果你在一个以上的地点看到相同的程序结构，那么可以肯定：设法将它们合而为一，程序会变得更好。

最单纯的DuplicatedCode就是 “同一个类的两个函数含有相同的表达式”。这时候你需要做 的就是采用Extract Method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“两个互为兄弟的子类内含相同表达式”。要避免这种情况，只需对两个类都使用Extract Method，然后再对被提炼出来的代码使用Pull Up Method，将它推入超类内。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。然后你可能发现可以运用Form Template Method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，你可以选择其中较清晰的一个，并使用Substitute Algorithm将其他函数的算法替换掉。

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

过长函数 Long Method

拥有短函数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委托，根本没有进行仟何计算。

“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。

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。找到函数中适合集中在一起的部分，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最，它们会对你的函数提炼形成阻碍。如果 你尝试运用最终就会把许多参数和临时变量当作参数，传递给被提炼出来的新函数，导致可读性几乎没有任何提升。此时，你可以经常运用Replace Temp with Query来消除这些临时元素。Introduce Parameter Object和Preserve Whole Object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。

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。它们通常能指出代码用途和实现手法之间的语义距离。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。

条件表达式和循环常常也是提炼的信号。你可以使用Decompose Conditional 处理条件表达式。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。

过大的类

你可以运用Extract Class将几个变量一起提炼至新类内。提炼时应该选择类内彼此相关的变量，将它们放在一起。例如depositAmount和depositCurrency可能应该隶属同一个类。通常如果类内的数个变是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子类，你会发现Extract Subclass往往比较简单。

有时候类并非在所有时刻都使用所有实例变量。果眞如此，你或许可以多次使用Extract Class或Extract Subclass。

和“太多实例变量”一样，类内如果有太多代码，也是代码重复、混乱并最终走向死亡的源头。最简单的解决方案是把多余的东西消弭于类内部。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“拥有太多实例变量”一样,一个类如果拥有太多代码，往往也适合使用Extract Class和Extract Subclass。这里有个技巧：先确定客户端如何使用它们，然后运用为每一 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类。

如果你的Large Class是个GUI类，你可能需要把数据和行为移到一个独立的领域对象去。你可能需要两边各保留一些重复数据，并保持两边同步。Duplicate Observer Data告诉你该怎么做。这种情况下，特别是如果你使用旧式的AWT组件，你可以采用这种方式去掉GUI类并代以Swing组件。

过长参数列

如果向已有的对象发出一条请求就可以取代一个参数，那么你应该激活重构手法Replace Parameter With Method。在这里，“已有的对象”可能是函数所属类内的一个字段，也可能是另一个参数。你还可以运用Preserve Whole Object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为它们制造出一个“参数对象”。

这里有一个重要的例外：有时候你明显不希望造成“被调用对象”与“较大对象”间的某种依赖关系。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赖结构了。

发散式变化 Divergent Change

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

如果某个类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。当你看着一个类说：“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类中，而这个新类内的所有内容都应该反应此变化。为此，你应该找出某特定原因而造成的所有变化，然后运用Extract Class将它们提炼到另一个类中。

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类内做出许多小修改，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和Move Field把所有需要修改的代码放进同一个类。如果眼下没有合适的类可以安置这些代码，就创造一个。通常可以运用Inline class把一系列相关行为放进同一个类。这可能会造成少量Divergent Change，但你可以轻易处理它。

Divergent Change是指“一个类受多种变化的影响”，Shotgun Surgery则是指“一种变化引发多个类相应修改”。这两种情况下你都会望整理代码，使 “外界变化” 与 “需要修改的类”趋于一一对应。

依恋情结 Feature Envy

对象技术的全部要点在于：这是一种“将数据和对数据的操作行为包装在一起”的技术。有一种经典气味是：函数对某个类的兴趣高过对自己所处类的兴趣。这种仰慕之情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个值，从另一个对象那儿调用几乎半打的取值函数。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把这一部分提炼到独立函数中，再使用Move Method带它去它的梦中家园。

当然，并非所有情况都这么简单。一个函数往往会用到几个类的功能，那么它究竟该被置于何处呢?我们的原则是：判断哪个类拥有最多被此函数使用的数据，然后就把这个函数和那些数据摆在一起。如果先以Extract Method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。

有几个复杂精巧的模式破坏了这个规则。说起这个话题，GoF[GangofFour]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的行为隔离开来——当然也付出了“多一层间接性”的代价。

数据泥团 Data Clumps

你常常可以在很多地方看到相同的三四项数据：两个类中相同的字段、许多函数签名中相同的参数。这些总是绑在一起出现的数据眞应该拥有属于它们自己的对象。首先请找出这些数据以字段形式出现的地方，运用Extract Class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上，运用Introduce Parameter Object或Preserve Whole Object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。是的，不必在意Data Clumps只用上新对象的一部分字段，只要以新对象取代两个（或更多）字段，你就值回票价了。

一个好的评判办法是：删掉众多数据中的一项。这么做，其他数据有没有因而 失去意义?如果它们不再有意义，这就是个明确信号：你应该为它们产生一个新对象。减少字段和参数的个数，当然可以去除一些坏味道，但更重要的是：一互拥有新对象，你就有扒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, 这可以帮你指出能够移至新类中的种种程序行为。不必太久，所有的类都将在它们的小小社会中充分发挥价值。

基本类型偏执

大多数编程环境都有两种数据：结构类型允许你将数据组织成存意义的形式；基本类型则是构成结构类型的积木块。结构总是会带来一定的额外开销。它们可能代表着数据库中的表，如果只为做一两件事而创建结构类型也可能显得太麻烦。

对象的一个极大的价值在于：它们模糊（甚至打破）了横亘于基本数据和体积较大的类之间的界限。你可以轻松编写出一些与语言内置（基本）类型无异的小型类。例如Java就以基本类型表示数值，而以类表示字符串和日期一这两个类型在其他许多编程环境中都以基本类型表现。

对象技术的新手通常不愿意在小任务上运用小对象——像是结合数值和币种的money类、由一个起始值和一个结束值组成的range类、电话号码或邮政编码（ZIP）等等的特殊字符串。你可以运用Replace Data Value with Object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。如果想要替换的数据值是类型码，而它并不影响行为，则可以运用Replace Type Code with Class将它换掉。如果你有与类型码相关的条件表达式，可运用Replace Type Code with Subclass或Replace Type Code with State/Strategy加以处理。

如果你有一组应该总是被放在一起的字段，可运用Extract Class。如果你在参数列中看到基本型数据，不妨试试Introduce Parameter Object。如果你发现自己正从数组中挑选数据，可运用Replace Array With Object。

switch 惊悚现身 Switch Statements

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。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，就必须找到所有switch语句并修改它们。 面向对象中的多态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语句常常根据类型码进行选择，你要的是“与该类型码相关的函数或类”，所 以应该使用Extract Method将switch语句提炼到一个独立函数中，再以Move Method将它搬移到需要多态性的那个类里。此时你必须决定是否使用Replace Type Code with Subclasses或Replace Type Code with State/Strategy。一旦这样完成继承结构之后，你就可以运用Replace Conditional with Polymorphism了。

如果你只是在单一函数中有些选择事例，且并不想改动它们，那么多态就有点 杀鸡用牛刀了。这种情况下Replace Paramerer with Explicit Methods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。

平行继承体系 Parallel Inheritance Hierarchies

Parallel Inheritance Hierarchies其实是Shotgun Surgery的特殊情况。在这种情况下，每当你为某个类增加一个子类，必须也为另一个类相应增加一个子类。如果你发现某个继承体系的类名称前缀和另一个继承体系的类名称前缀完全相同，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例引用另一个继承体系的实例。如果再接再励运用Move Method和Move Field，就可以将引用端的继承体系消弭于无形。

冗赘类 Lazy Class

你所创建的每一个类，都得有人去理解它、维护它，这些工作都是要花钱的。 如果一个类的所得不值其身价，它就应该消失。项目中经常会出现这样的情况：某个类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，并添加一个类来应付这些变化，但变化实际上没有发生。不论上述哪一种原因，请让这个类庄严赴义吧。如果某些子类没有做足够的工作，试试Collapse Hierarchy。对于几乎没用的组件，你应该以Inline Class对付它们。

夸夸其谈未来性 Speculative Generality

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钩子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。

如果你的某个抽象类其实没有太大作用，请运用Collapse Hierarchy 。不必要的委托可运用Inline Class除掉。如果函数的某些参数未被用上，可对它实施 Remove Parameter。如果函数名称带有多余的抽象意味，应该对它实施Rename Method,让它现实一些。

如果函数或类的唯一用户是测试用例，这就飘出了坏味道Speculative Generality。如果你发现这样的函数或类，请把它们连同其测试用例一并删掉。但如果它们的用途是帮助测试用例检测正当功能，当然必须刀下留人。

令人迷惑的暂时字段 Temporary Field

有时你会看到这样的对象：其内某个实例变量仅为某种特定情况而设。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最。在变量未被使用的情况下猜测当初其设置目的的，会让你发疯的。

请使用Extract Class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。

如果类中有个复杂算法，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字段中。但是这些字段只在使用该算法时才有效，其他情况下只会让入迷惑。这时候你可以利用Extract Class把这些变量和其相关函数提炼到一个独立类中。提炼后的新对象将是一个函数对象[Beck]。

过度耦合的消息链 Message Chains

如果你看到用户向一个对象请求另一个对象，然后再向后者请求另一个对象， 然后再请求另一个对象……这就是消息链。实际代码中你看到的可能是一长串getThis()或一长串临时变量。采取这种方式，意味客户代码将与查找过程中的导 航结构紧密耦合。一旦对象间的关系发生任何变化，客户端就不得不做出相应修改。

这时候你应该使用Hide Delegate。你可以在消息链的不同位置进行这种重构手法。理论上可以重构消息链上的任何一个对象，但这么做往往会把一系列对象(intermediate object)都变成Middle Man。通常更好的选择是：先观察消息链最终得到的对象是用来干什么的，看看能否以Extract Method把使用该对象的代码提炼到一个独立函数中，再运用Move Method把这个函数推入消息链。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都视为坏东西，我们不这样想。

中间人

对象的基本特征之一就是封装——对外部世界隐藏其内部细节。封装往往伴随委托。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車簿亦或秘书來记录自己的约会。

但是人们可能过度运用委托。你也许会看到某个类接口有一半的函数都委托给其他类，这样就是过度运用。这时应该使用Remove Middle Men,直接和眞正负责的对象打交道。如果这样“不干实事”的函数只有少数几个，可以运用Inline Method把它们放进调用端。如果这些Middle Man还有其他行为，可以运用replace Delegate with Inheritance把它变成实责对象的子类，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。

狎昵关系 Inappropriate Intimacy

有时你会看到两个类过于亲密，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，我们不必做卫道士；但对于类，我们希望它们严守清规。

就像古代恋人一样，过分狎昵的类必须拆散。你叮以采用Move Method和Move Field帮它们划清界线，从而减少狎昵行径。你也可以看看是否可以运用Change Bidirectional Association to Unidirectional让其中一个类对另一个斩断情丝。如果两个类实在是情投意合，可以运用Extract Class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类。或者也可以尝试运用Hide Delegate让另一个类来为它们传递相思情。

继承往往造成过度亲密，因为子类对超类的了解总是超过后者的主观愿望。如 果你觉得该让这个孩子独自生活了，请运用Replace Inheritance with Delegation让它离开继承体系。

异曲同工的类 Alternative Classes with Different Interfaces

如果两个函数做同一件事，却打着不同的签名，请运用Rename Method根据它们的用途重新命名。但这往往不够，请反复运用Move Method将某些行为 移入类，直到两者的协议一致为止。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为自己赎点罪。

不完美的库类

复用常被视为对象的终极目的。不过我们认为，复用的意义经常被高估——大多数对象只要够用就好。但是无可否认，许多编程技术都建立在程序库的基础上，没人敢说是不是我们都把排序算法忘得一干二净了。

库类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，所以库作者的任务真地很艰巨。麻烦的是库往往构造得不够好，而且往往不可能让我们修改其中的类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。如果你只想修改库类的一两个函数,可以运用Introduce Foreign Method；如果想要添加一大堆额外行为，就得运用Introduce Local Extension。

纯稚的数据类 Data Class

所谓DataClass是指：它们拥有一些字段，以及用于访问（读写）这些字段的函数，除此之外一无长物。这样的类只是一种不会说话的数据容器，它们几乎一定被其他类过份细琐地操控着。这些类早期可能拥有public字段，果真如此你应该在别人注意到它们之前，立刻运用以Encapsulate Field将它们封装起来。如果这些类内含容器类的字段，你应该检查它们不是得到了恰巧的封装；如果没有，就运用 Encapsulate Collection把它们封装起来。对于那些不该被其他类修改的字段，请运用Remove Setting Method。

然后，找出这些取值/设值函数被其他类运用的地点。尝试以Move Method把那些调用行为搬移到Data Class來 。如果无法搬移整个函数，就运用Extract Method产生一个可被搬移的函数。不久之后你就可以运用Hide Method把这些取值/设值函数隐藏起来了。

DataClass就像小孩子。作为一个起点很好，但若要让它们像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。

被拒绝的遗赠

子类应该继承超类的函数和数据。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，这就意味着继承体系设计错误。你需要为这个子类新建一个兄弟类，再运用Push Down Method和Push Down Field把所有用不到的函数下推给那个兄弟。这样一来，超类就只持有所有子类共享的东西。你常常会听到这样的建议：所有超类都应该是抽象（abstract) 的。

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。我们经常利用继承来复用一些行为，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。

如果子类复用了超类的行为（实现），却又不愿意支持超类的接口，Refused Bequest的坏味道就会变得浓烈。拒绝继承超类的实现，这一点我们不介意；但如果拒绝继承超类的接口，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，应该运用Replace Inheritance with Delegation来达到目的。

过多的注释 Comments

别担心，我们并不是说你不该写注释。从嗅觉上说，Comments不是种坏味道,事实上它们还是一种香味呢。我们之所以要在这里提到Comments, 是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;如果函数已经提炼出来，但还是需要注释来解释其行为，试试Rename Method;如果 你需要注释说明某些系统的需求规格，试试Introduce Assertion。

当你感觉需要撰写注释时，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。

第四章 构筑测试体系

如果你想进行重构，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。

自测试代码的价值

确保所有测试都完全自动化，让它们检查自己的测试结果。

一套测试就是一个强大的**bug**侦测器，能够大大缩减查找**bug**所需要的时间。

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（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[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、自由散漫的黑客，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，而且也知道测试能让他们尽可能快速地前进。

Java之中的测试惯用手法是testing main，意思是每个类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。这种做法的问题是很难轻松运行多个测试。另一种做法是：建立一个独立类用于测试， 并在一个框架中运行它，使测试工作更轻松。

JUnit测试框架

频繁地运行测试。每次编译请把测试也考虑进去—— 聲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。

单元测试和功能测试

JUnit框架的用途是单元测试，所以我应该讲讲单元测试（Unit Test) 和功能测试 （Functional Test) 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程序员的生产率。至于让QA部门开心，那只是附带效果而已。

单元测试是高度局部化的东西，每个测试类都隶属于单一包。它能够测试其他包的接口，除此之外它将假设其他包一切正常。

功能测试就完全不同。它们用来保证软件能够正常运作。它们从客户的角度保障质最，并不关心程序员的生产力。它们应该由一个喜欢寻找bug的独立团队来开发。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。

—般而言，功能测试尽可能把整个系统当作一个黑箱。面对一个拥有GUI的待测系统，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bug,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，用来暴露这个**bug**。

JUnit框架设计用来编写单元测试。功能测试往往以其他工具辅助进行，例如某些拥有GUI的测试工具，然而通常你还得撰写一些“专用于你的应用程序”的测试工具，它们能比通用的GUI脚本更好地达到测试效果。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。在进行重构时，我会更多地倚赖程序员的好朋友：单元测试。

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：观察类该做的所有事情，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的行为，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字段的访问函数，因为它们太简单了，不大可能出错。

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。

测试的一项重要技巧就是“寻找边界条件”。“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。

当事情被认为应该会出错时，别忘了检查是否抛出了预期的异常。

随着测试类愈来愈多，你可以生成另一个类，专门用来包含由其他测试类所组成的测试套件。这很容易做到，因为一个测试套件本来就可以包含其他测试套件。这样，你就可以拥有一个“主控的”测试类。

不要因为测试无法捕捉所有**bug**就不写测试，因为测试的确可以捕捉到大多数**bug**。

对象技术有个微妙处：继承和多态会让测试变得比较困难，因为将有许多种组合需要测试。

第五章 重构列表

第5~12章构成了一份重构列表草案，其中所列的重构手法来自我最近数年的心得。这份列表并非巨细靡遗，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。

重构的记录格式

介绍重构时，我采用一种标准格式。每个重构手法都有如下五个部分。

- 首先是名称（name)。建造一个重构词汇表，名称是很重要的。这个名称也就是我将在本书其他地方使用的名称。
- 名称之后是一个简短概要（summary)。简单介绍此一重构手法的适用情景，以及它所做的事情。这部分可以帮助你更快找到你所需要的重构手法。
- 动机（motivation) 为你介绍“为什么需要这个重构”和 “什么情况下不该使用这个重构”。
- 做法 （mechanics) 简明扼要地一步-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”包括三个部分：

1. 一句话介绍这个重构能够帮助解决的问题：
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UML图。总之，哪种形式能更好呈现该重构的本质，我就使用哪种形式。

撰写 “做法”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。

“范例”像是简单而有趣的教科书。我使用这些范例是为了帮助解释重构的基本要素，最大限度地避免其他枝节，所以我希望你能原谅其中的简化下作（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。

寻找引用点

在强类型语言中，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複写多次的函数时，情况更是如此。所以如果你在一个继承体系中工作，请先利用文本查找工具，检查是否有其他类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效率。如果眞是这样，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（而不是依赖赖编译器）。

第三个问题是：编译器无法找到通过反射机制而得到的引用点。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，所有这些便利都没有了，你必须为许多编译搭配测试。

这些重构手法有多成熟

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。

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。

第六章 重新组织函数

对付过长函数，一项重要的重构手法就是Extract Method，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。

Extract Method最大的困难就是处理局部变量，而临时变最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Spilt Temporary Variable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。它让我可以分解哪怕最混乱的函数，代价则是引入一个新类。

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。

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引入更清晰的算法。

Extract Method 提炼函数

动机

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。

有几个原因造成我喜欢简短而命名良好的函数。首先，如果每个函数的粒度都很小，那么函数被复用的机会就更大；其次，这会使高层函数读起来就像一系列注释：再次，如果函数都是细粒度，那么函数的覆写也会更容易些。

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，它们才能眞正起作用，所以你需要在函数名称上下点功夫。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。如果提炼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。

做法

- 创造一个新函数，根据这个函数的意图来对它命名（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复制到新建的目标函数中。
- 仔细检查提炼出的代码，看看其中是否引用了“作用域限于源函数”的变量(包括局部变量和源函数参数）。
- 检查是否有“仅用于被提炼代码段”的临时变量。如果有，在目标函数中将它们声明为临时变量。
- 检查被提炼代码段，看看是否有任何局部变量的值被它改变。如果一个临时变量值被修改了，看看是否可以将被提炼代码段处理为一个查询，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Spilt Temporary Variable，然后再尝试提炼。也可以使用Replace Temp with Query将临时变量消灭掉。
- 将被提炼代码段中需要读取的局部变最，当作参数传给目标函数。

- 处理完所有局部变量之后，进行编译。
- 在源函数中，将被提炼代码段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代码段的外围。如果是，现在你可以删除这些声明式了。

- 编译，测试。

Inline Method 内联函数

动机

本书经常以简短的函数表现动作意阁，这样会使代码更清晰易读。恒有时候你会遇到某些函数，其内部代码和函数名称间样清晰易读。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。果眞如此，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。你可以将它们都内联到一个大型函数中，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之前先这么做，往往可以获得不错的效果。你可以把所要的函数（有着你要的行为）的所有调用对象的函数内容都内联到函数对象中。比起既要移动一个函数、又要移动它所调用的其他所有函数，将整个大型函数作为整体来移动会比较简单。

做法

- 检查函数，确定它不具多态性。

如果子类继承了这个函数，就不要将此函数内联，因为子类无法覆写一个 根本不存在的函数.

- 找出这个函数的所有被调用点。
- 将这个函数的所冇被调用点都替换为函数本体。
- 编译，测试。
- 删除该函数的定义。

被我这样一写，Inline Method似乎很简单。但情况往往并非如此。对于递归调用、多返回点、内联至另一个对象中而该对象并无提供访问函数.....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。

Inline Temp 内联临时变量

动机

Inline Temp多半是作为Replace Temp with Query的一部分使用的，所以真正的动机出现在后者那儿。唯一单独使用Inline Temp的情况是：你发现某个临时变景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，可以放心地把它留在那儿。但如果这个临时变量妨碍了其他的重构手法，例如Extract Method你就应该将它内联化。

做法

- 检查给临时变量赋值的语句，确保等号右边的表达式没有副作用。
- 如果这个临时变景并未被声明为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”的表达式。
- 每次修改后，编译并测试。
- 修改完所行引用点之后，删除该临时变量的声明和赋值语句。
- 编译，测试。

Replace Temp with Query 以查询取代临时变量

你的程序以一个临时变量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中。将这个临时变量的所有引用点替换为对新函数的调用。此后，新函数就可被其他函数使用。

动机

临时变景的问题在于：它们是暂时的，而且只能在所属函数内使用。由于临时变量只在所属函数内可见，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需要的临时变量。如果把临时变量替换为一个查询，那么同一个类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，使你能够为这个类编写更清晰的代码。

Replace Temp with Query往往是你运用Extract Method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。

这个重构手法较为简单的情况是：临时变量只被赋值一次，或者赋值给临时变量的表达式不受其他条件影响。其他情况比较棘手，但也有可能发生。你可能需要先运用Spilt Temporary Variable或Seperate Query from Modifier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)，就需要将某些程序逻辑（例如循环）复制到查询函数去。

做法

首先是简单情况：

- 找出只被赋值一次的临变量。
  - 如果某个临时变量被赋值超过一次，考虑使用Spilt Temporary Variable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类需要使用它，那时放松对它的保护也很容易。
  - 确保提炼出来的函数无任何副作用，也就是说该函数并不修改任何对象内容。如果它有副作用，就对它进行Seperate Query from Modifier。

- 编译，测试。
- 在该临时变量身上实施Inline Temp。

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逻辑。

运用此手法，你可能会担心性能问题。和其他性能问题一样，我们现在不管它，因为它十有八九根本不会造成任何影响。若是性能真的出了问题，你也可以在优化时期解决它。

Introduce Explaining Variable 引入解释性变量

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。

动机

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。

在条件逻辑中，Introduce Explaining Variable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来解释一段代码的意义。毕竟临时变量只在它所处的那个函数中才有意义，局限性较大，函数则可以在对象的整个生命中都有用，并且可被其他对象使用。但有时候，当局部变量使 Extract Method难以进行时，我就使用Introduce Explaining Variable。

做法

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

应该在什么时候使用Introduce Explaining Variable呢？答案是：在Extract Method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绝非易事。这种情况下就会使用Introduce Explaining Variable来理清代码，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把中间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，那么中间引入的那些解释性临时变量也有其价值。

Split Temporary Variable 分解临时变量

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,也不被用于收集计算结果。针对每次赋值，创造一个独立、对应的临时变量。

动机

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“结果收集变量”就是两个典型例子：循环变景（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); 结果收集变量(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。如果临时变最承担多个责任，它就应该被替换(分解）为多个临时变量，每个变量只承担一个责任。同一个临时变量承担两件不同的事情，会令代码阅读者糊涂。

做法

- 在待分解临时变量的声明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，就意味这是个结果收集变量，那么就不要分解它。结果收集变量的作用通常是累加、字符串接合、写入流或者向集合添加元素。
- 将新的临时变量声明为final。
- 以该临时变量的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改名，并修改下次赋值之前的引用点。

Remove Assignments to Parameters 移除对参数的赋值)

代码对一个参数进行赋值。以一个临时变量取代该参数的位置。

动机

在按值传递的情况下，对参数的任何修改，都不会对调用端造成任何影响。那些用过按引用传递方式的人可能会在这一点上犯糊涂。

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

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

当然，面对那些使用“出参数”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出参数。

做法

- 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”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”。
- 修改赋值语句，使其改为对新建之临时变量赋值。
- 编译，测试。
    - 如果代码的语义是按引用传递的，请在调用端检查调用后是否还使用了这个参数，也要检查有多少个按引用传递的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止一个，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。

Java的按值传递

从本质上说，对象的引用是按值传递的。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是没有意义的。

Java 1.1及其后版本允许将参数标示为final, 从而避免函数中对参数赋值。即使某个参数被标示为final,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，我很少在参数列表中这样标示它们。

Replace Method with Method Object 以函数对象取代函数

你有一个大型函数，其中对局部变量的使用使你无法采用Extract Method。将这个函数放进一个单独对象中，如此一来局部变置就成了对象内的字段。然后你可以在同一个对象中将这个大型函数分解为多个小型函数。

动机

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, 你应该把手伸进工具箱的深处，祭出函数对象(method object)[Beck]这件法宝。

Replace Method with Method Object会将所有局部变量都变成函数对象的字段。然后你就可以对这个新对象使用Extract Method创造出新函数，从而将原本的大型函数拆解变短。

做法

- 建立一个新类，根据待处理函数的用途，为这个类命名。
- 在新类中建立一个final字段，用以保存原先大型函数所在的对象。我们将这个字段称为“源对象”。同时，针对原函数的每个临时变量和每个参数，在新类中建立一个对应的字段保存之。
- 在新类中建立一个构造函数，接收源对象及原函数的所有参数作为参数。
- 在新类中建立一个compute ()函数。
- 将原函数的代码复制到compute()函数中。如果需要调用源对象的任何函数，请通过源对象字段调用。
- 编译。
- 将旧函数的函数本体替换为这样一条语句：“创建上述新类的一个新对象，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字段，所以你可以任意分解这个大型函数，不必传递任何参数。

Substitute Algorithm 替换算法

你想要把某个算法替换为另一个更清晰的算法。将函数本体替换为另一个算法。

动机

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中单的小型函数，然后你才能很有把握地进行算法替换工作。

做法

- 准备好另一个（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问，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测试用例出现麻烦，以及出现了怎样的麻烦。

第七章 在对象之间搬移特性

在对象的设计过程中，“决定把责任放在哪儿”即使不是最重要的事，也是最重要的事之一。我使用对象技术已经十多年了，但还是不能一开始就保证做对。这曾经让我很烦恼，但现在我知道，在这种情况下，可以运用重构，改变自己原先的设计。

常常我只需要使用Move Method和Move Field简单地移动对象行为，就可以解决这些问题。如果这两个重构手法都需要用到，我会首先使用Move Field,再使用Move Method。

类往往会因为承担过多责任而变得臃肿不堪。这种情况下，我会使用Extract Class将一部分责任分离出去。如果一个类变得太“不负责任”，我就会使用Inline Class将它融入另一个类。如果一个类使用了另一个类类，运用Hide Delegate将这种关系隐藏起来通常是有帮助的。有时候隐藏委托类会导致拥有者的接口经常变化，此时需要使用Remove Middle Man。

本章的最后两项重构——Introduce Foreign Method和Introduce Local Extension比较特殊。只有当我不能访问某个类的源码，却又想把其他责任移进这个不可修改的类时，我才会使用这两个重构手法。如果我想加入的只是一或两个函数，就会使用Introduce Foreign Method;如果不止一两个函数，就使用Introduce Local Extension。

Move Method 搬移函数

你的程序中，有个函数与其所驻类之外的另一个类进行更多交流：调用后者，或被后者调用。在该函数最常引用的类中建立一个有着类似行为的新函数。将旧函数变成一个单纯的委托函数，或是将旧函数完全移除。

动机

“搬移函数”是重构理论的支柱。如果一个类有太多行为，或如果一个类与另一个类有太多合作而形成高度耦合，我就会搬移函数。通过这种手段，可以使系统中的类更简单，这些类最终也将更干净利落地实现系统交付的任务。

我常常会浏览类的所有函数，从中寻找这样的函数：使用另一个对象的次数比使用自己所驻对象的次数还多。一旦我移动了一些字段，就该做这样的检查。一旦发现有可能搬移的函数，我就会观察调用它的那一端、它调用的那一端，以及继承体系中它的任何一个重定义函数。然后，会根据“这个函数与哪个对象的交流比较多”，决定其移动路径。

这往往不是容易做出的决定。如果不能肯定是否应该移动一个函数，我就会继续观察其他函数。移动其他函数往往会让这项决定变得容易一些。有时候，即使你移动了其他函数，还是很难对眼下这个函数做出决定。其实这也没什么大不了的。如果真地很难做出决定，那么或许“移动这个函数与否”并不那么重要。所以，我会凭本能去做，反正以后总是可以修改的。

做法

- 检查源类中被源函数所使用的一切特性（包括字段和函数），考虑它们是否也该被搬移。
  - 如果某个特性只被你打算搬移的那个函数用到，就应该将它一并搬移。如果另有其他函数使用了这个特性，你可以考虑将使用该特性的所有函数全都一并搬移。有时候，搬移一组函数比逐一搬移简单些.
- 检查源类的子类和超类，看看是否有该函数的其他声明。
  - 如果出现其他声明，你或许无法进行搬移，除非目标类也同样表现出多态性。
- 在目标类中声明这个函数。
  - 你可以为此函数选择一个新名称——对目标类更有意义的名称。
- 将源函数的代码复制到目标函数中。调整后者，使其能在新家中正常运行。
  - 如果目标函数使用了源类中的特性，你得决定如何从目标函数引用源对象。如果目标类中没有相应的引用机制，就把源对象的引用当作参数，传给新建立的目标函数。
  - 如果源函数包含异常处理，你得判断逻辑上应该由哪个类来处理这一异常。如果应

该由源类来负责，就把异常处理留在原地。

- 编译目标类。
- 决定如何从源函数正确引用目标对象。
  - 可能会有一个现成的字段或函数帮助你取得目标对象。如果没有，就看能否轻松建立一个这样的函数。如果还是不行，就得在源类中新建一个字段来保存目标对象。这可能是一个永久性修改，但你也可以让它是暂时的，因为后继的其他重构项目可能会把这个新建字段去掉。
- 修改源函数，使之成为一个纯委托函数。
- 编译，测试。
- 决定是否删除源函数，或将它当作一个委托函数保留下来。
  - 如果你经常要在源对象中引用目标函数，那么将源函数作为委托函数保留下来会比较简单。
- 如果要移除源函数，请将源类中对源函数的所有调用，替换为对目标函数的调用。
  - 你可以每修改一个引用点就编译并测试一次。也可以通过一次“查找/替换”改掉所有引用点，这通常简单一些。
- 编译，测试。

Move Field 搬移字段

你的程序中，某个字段被其所驻类之外的另一个类更多地用到。在目标类新建一个字段，修改源字段的所有用户，令它们改用新字段。

动机

在类之间移动状态和行为，是重构过程中必不可少的措施。随着系统发展，你会发现自己需要新的类，并需要将现有的工作责任拖到新的类中。在这个星期看似合理而正确的设计决策，到了下个星期可能不再正确。这没问题。如果你从来没遇到这种情况，那才有问题。

如果我发现，对于一个字段，在其所驻类之外的另一个类中有更多函数使用了它，我就会考虑搬移这个字段。上述所谓“使用”可能是通过设值/ 取值函数间接进行的。我也可能移动该字段的用户（某个函数），这取决于是否需要保持接口不受变化。如果这些函数看上去很适合待在原地，我就选择搬移字段。

使用Extract Class时，我也可能需要搬移字段。此时我会先搬移字段，然后再搬移函数。

做法

- 如果字段的访问级是public, 使用Encapsulate Field将它封装起来。
    - 如果你有可能移动那些频繁访问该字段的函数，或如果有许多函数访问某个字段，先使 用Self Encapsulate Field也许会有帮助。
- 编译，测试。
- 在目标类中建立与源字段相同的字段，并同时建立相应的设值/取值函数。
- 编译目标类。
- 决定如何在源对象中引用目标对象。
    - 首先看是否有一个现成的字段或函数可以助你得到目标对象。如果没有，就看能否轻易建立这样一个函数。如果还不行，就得在源类中新建一个字段来存放目标对象。这可能是个永久性修改，但你也可以让它是暂时的, 因为后续重构可能会把这个新建字段除掉•
- 删除源字段。
- 将所有对源字段的引用替换为对某个目标函数的调用。
    - 如果需要读取该变量，就把对源字段的引用替换为对目标取值函数的调用；如果要对该变量赋值，就把对源字段的引用替换成对设值函数的调用。
    - 如果源字段不是private的，就必须在源类的所有子类中查找源字段的引用点，并进行相应替换。
- 编译，测试。

Extract Class 提炼类

某个类做了应该由两个类做的事。建立一个新类，将相关的字段和函数从旧类搬移到新类。

动机

你也许听过类似这样的教诲：一个类应该是一个清楚的抽象，处理一些明确的责任。但是在实际工作中，类会不断成长扩展。你会在这儿加入一些功能，在那儿加入一些数据。给某个类添加一项新责任时，你会觉得不值得为这项责任分离出一个单独的类。于是，随若责任不断增加，这个类会变得过分复杂。很快，你的类就 会变成一团乱麻。

这样的类往往含有大量函数和数据。这样的类往往太大而不易理解。此时你需要考虑哪些部分可以分离出去，并将它们分离到一个单独的类中。如果某些数据和某些函数总是一起出现，某些数据经常同时变化甚至彼此相依，这就表示你应该将它们分离出去。一个有用的测试就是问你自己，如果你搬移了某些字段和函数，会发生什么事？其他字段和函数是否因此变得无意义？

另一个往往在开发后期出现的信号是类的子类化方式。如果你发现子类化只影响类的部分特性，或如果你发现某些特性需要以一种方式来子类化，某些特性则需要以另一种方式子类化，这就意味你需要分解原来的类。

做法

- 决定如何分解类所负的责任。
- 建立一个新类，用以表现从旧类中分离出来的责任。
  - 如果旧类剩下的责任与旧类名称不符，为旧类更名。
- 建立“从旧类访问新类”的连接关系。
  - 有可能需要一个双向连接。但是在眞正需要它之前，不要建立“从新类通往旧类”的连接•
- 对于你想搬移的每一个字段，运用Move Field搬移之。
- 每次搬移后，编译、测试。
- 使用Move Method将必要函数搬移到新类 。先搬移较低层函数（也就是“被其他函数调用”多 于“调用其他函数”者），再搬移较高层函数。
- 每次搬移之后，编译、测试。
- 检查，精简每个类的接口。
  - 如果你建立起双向连接，检查是否可以将它改为单向连接。
- 决定是否公开新类。如果你的确需要公开它，就要决定让它成为引用对象还是不可变的值对象。

Inline Class 将类内联化

某个类没有做太多事情。将这个类的所有特性搬移到另一个类中，然后移除原类。

动机

Inline Class正好与Extract Class相反。如果一个类不再承担足够责任、不再有单独存在的理由（这通常是因为此前的重构动作移走了这个类的责任），我就会挑选这一“萎缩类”的最频繁用户（也是个类），以Inline Class手法将“萎缩类”塞进另一个类中。

做法

- 在目标类身上声明源类的public协议，并将其中所有函数委托至源类。
  - 如果“以一个独立接口表示源类函数”更合适的话，就应该在内联之前先使用Extract Interface。
- 修改所有源类引用点，改而引用目标类。
  - 将源类声明为private,以斩断包之外的所有引用可能。同时修改源类的名称，这便可使编译器帮助你捕捉到所有对于源类的隐藏引用点。
- 编译，测试。
- 运用Move Method和Move Field,将源类的特性全部搬移到目标类。
- 为源类举行一个简单的“丧礼”

Hide Delegate 隐藏委托关系

客户通过一个委托类来调用另一个对象。在服务类上建立客户所需的所有函数，用以隐藏委托关系。

动机

“封装”即使不是对象的最关键特征，也是最关键特征之一。“封装”意味每个对象都应该尽可能少了解系统的其他部分。如此一来，一旦发生变化，需要了解这一变化的对象就会比较少——这会使变化比较容易进行。

任何学过对象技术的人都知道：虽然Java允许将字段声明为public,但你还是应该隐藏对象的字段。随着经验日渐丰富，你会发现，有更多可以（而且值得）封装的东西。

如果某个客户先通过服务对象的字段得到另一个对象，然后调用后者的函数，那么客户就必须知晓这一层委托关系。万一委托关系发生变化，客户也得相应变化。你可以在服务对象上放置一个简单的委托函数，将委托关系隐藏起来，从而去除这种依赖。这么一来，即便将来发生委托关系上的变化，变化也将被限制在服务对象中，不会波及客户。

对于某些或全部客户，你可能会发现，有必要先使用Extract Class。一旦你对所有客户都隐藏了委托关系，就不再需要在服务对象的接口中公开被委托对象了。

做法

- 对于每一个委托关系中的函数，在服务对象端建立一个简单的委托函数。
- 调整客户，令它只调用服务对象提供的函数。
    - 如果使用者和服务提供者不在同一个包，考虑修改委托函数的访问权限，让客户得以在包之外调用它•
- 每次调整后，编译并测试。
- 如果将来不再有任何客户需要取用Delegate(受托类），便可移除服务对象中的相关访问函数。
- 编译，测试。

Remove Middle Man 移除中间人

动机

在Hide Delegate的“动机”一节中，我谈到了“封装受托对象”的好处。何是这层封装也是要付出代价的，它的代价就是：每当客户要使用受托类的新特性时，你就必须在服务端添加一个简单委托函数。随着受托类的特性（功能）越来越多，这一过程会让你痛苦不已。服务类完全变成了一个“中间人”，此时你就应该让客户直接调用受托类。

很难说什么程度的隐藏才是合适的。还好，有了Hide Delegate和Remove Middle Man,你大可不必操心这个问题，因为你可以在系统运行过程中不断进行调整。随着系统的变化，“合适的隐藏程度”这个尺度也相应改变。6个月前恰如其分的封装，现今可能就显得笨拙。重构的意义就在于：你永远不必说对不起——只要把出问题的地方修补好就行了。

做法

- 建立一个函数，用以获得受托对象。
- 对于每个委托函数，在服务类中删除该函数，并让需要调用该函数的客户转为调用受托对象。
- 处理每个委托函数后，编译、测试。

Introduce Foreign Method 引入外加函数

你需要为提供服务的类增加一个函数，但你无法修改这个类。在客户类中建立一个函数，并以第一参数形式传入一个服务类实例。

动机

这种事情发生过太多次了：你正在使用一个类，它真的很好，为你提供了需要的所有服务。而后，你又需要一项新服务，这个类却无法供应。于是你开始咒骂："为什么不能做这件事？"如果可以修改源码，你便可以自行添加一个新函数：如果不能，你就得在客户端编码，补足你要的那个函数。

如果客户类只使用这项功能一次，那么额外编码工作没什么大不了，甚至可能根本不需要原本提供服务的那个类。然而，如果你需要多次使用这个函数，就得不断重复这些代码。还记得吗，重复代码是软件万恶之源。这些重复代码应该被抽出来放进同一个函数中。进行本项重构时，如果你以外加函数实现一项功能，那就是一个明确信号：这个函数原本应该在提供服务的类中实现。

如果你发现自己为一个服务类建立了大量外加函数，或者发现有许多类都需要同样的外加函数，就不应该再使用本项重构，而应该使用Introduce Local Extension。

但是不要忘记：外加函数终归是权宜之计。如果有可能，你仍然应该将这些函数搬移到它们的理想家园。如果由于代码所有权的原因使你无法做这样的搬移，就把外加函数交给服务类的拥有者，请他帮你在服务类中实现这个函数。

做法

- 在客户类中建立一个函数，用来提供你需要的功能。
  - 这个函数不应该调用客户类的任何特性。如果它需要一个值，把该值当作参数传给它。
- 以服务类实例作为该函数的第一个参数。
- 将该函数注释为："外加函数（foreign method)，应在服务类实现。"
  - 这么一来，如果将来有机会将外加函数搬移到服务类中时，你便可以轻松找出这些外加函数.

Introduce Local Extension 引入本地拓展

你需要为服务类提供一些额外函数，但你无法修改这个类。建立一个新类，使它包含这些额外函数。让这个扩展品成为源类的子类或包装类。

动机

很遗憾，类的作者无法预知未来，他们常常没能为你预先准备一些有用的函数。如果你可以修改源码，最好的办法就是直接加入自己需要的函数。但你经常无法修改源码。如果只需要一两个函数，你可以使用Introduce Foreign Method。但如果你需要的额外函数超过两个，外加函数就很难控制它们了。所以，你需要将这些函数组织在一起，放到一个恰当地方去。要达到这一目的，两种标准对象技术——子类化（subclassing)和包装（wrapping)——是显而易见的办法。这种情况下，我把子类或包装类统称为本地扩展（local extension)。

所谓本地扩展是一个独立的类，但也是被扩展类的子类型：它提供源类的一切特性，同时额外添加新特性。在任何使用源类的地方，你都可以使用本地扩展取而代之。

使用本地扩展使你得以坚持“函数和数据应该被统一封装”的原则。如果你一直把本该放在扩展类中的代码零散地放置于其他类中，最终只会让其他这些类变得过分复杂，并使得其中函数难以被复用。

在子类和包装类之间做选择时，我通常首选子类，因为这样的工作量比较少。 制作子类的最大障碍在于，它必须在对象创建期实施。如果我可以接管对象创建过程，那当然没问题：但如果你想在对象创建之后再使用本地扩展，就有问题了。此外，子类化方案还必须产生一个子类对象，这种情况下，如果有其他对象引用了旧对象，我们就同时有两个对象保存了原数据！如果原数据是不可修改的，那也没问题，我可以放心进行复制：但如果原数据允许被修改，问题就来了，因为一个修改动作无法同时改变两份副本。这时候我就必须改用包装类。使用包装类时，对本地扩展的修改会波及原对象，反之亦然。

做法

- 建立一个扩展类，将它作为原始类的子类或包装类。
- 在扩展类中加入转型构造函数。
    - 所谓“转型构造函数”是指“接受原对象作为参数”的构造函数。如果采用子类化方案，那么转型构造函数应该调用适当的超类构造函数；如果采用包装类方案，那么转型构造函数应该将它得到的传入参数以实例变量的 形式保存起来，用作接受委托的原对象。
- 在扩展类中加入新特性。

- 根据需要，将原对象替换为扩展对象。
- 将针对原始类定义的所有外加函数搬移到扩展类中。

第八章 重新组织数据

本章将介绍几个能让你更轻松处理数据的重构手法。很多人或许会认为Self Encapsulate Field有点多余，但是关于“对象应该直接访问其中的数据，抑或应该通过访问函数来访问”这一问题，争论的声音从来不曾停止。有时候你确实需要访问函数，此时就可以通过Self Encapsulate Field得到它们。通常我会选择“直接访问”方式，因为我发现，只要我想做，任何时候进行这项重构都是很简单的。

面向对象语言有一个很有用的特征：除了允许使用传统语言提供的简单数据类型，它们还允许你定义新类型。不过人们往往需要一段时间才能习惯这种编程方式。一开始你常会使用一个简单数值来表示某个概念。随着对系统的深入了解，你可能会明白，以对象表示这个概念，可能更合适。Replace Value With Object让你可 以将“哑”数据变成善表达的对象。如果你发现程序中有太多地方需要这一类对象，也可以使用Change Value to Reference将它们变成引用对象。

魔法数——也就是带有特殊含义的数字——从来都是个问题。我还清楚记得，一开始学习编程的时候，老师就告诉我不要使用魔法数。但它们还是不时出现。因此，只要弄清楚魔法数的用途，我就运用Replace Magic Number with Symbolic Constant将它们除掉，以绝后患。

对象之间的关联可以是单向的，也可以是双向的。单向关联比较简单，但有时为了支持一项新功能，你需要使用Change Unidirectional Association to Bidirectional将它变成双向关联。Change Bidirectional Association to Unidirectional则恰恰相反：如果你发现不再需要双向关联，可以使用这项重构将它变成单向关联。

我常常遇到这样的情况：GUI类竟然去处理不该它们处理的业务逻辑。为了把这些处理业务逻辑的行为移到合适的领域类去，你需要在领域类中保存这些逻辑的相关数据，并运用Duplicate Observed Data提供对GUI的支持。一般来说，我不喜欢重复的数据，但这是一个例外，因为这里的重复数据通常是不可避免的。

面向对象编程的关键原则之一就是封装。如果一个类公开了任何public数据，你就应该使用Encapsulate Collection将它郑重地包装起来。如果被公开的数据是个集合，就应该使用Encapsulate Collection,因为集合有其特殊协议。如果一整条记录都被裸露在外，就应该使用Replace Record with Data Class。

需要特别对待的一种数据是类型码（type code):这是一种特殊数值，用来指出“与实例所属之类型相关的某些东西”。类型码通常以枚举形式出现，并且通常以static final整数实现。如果这些类型码用来表现某种信息，并且不会改变所属类型的行为，你可以运用Replace Type Code With Class将它们替换掉，这项重构会为你提供更好的类型检查，以及一个更好的平台，使你可以在未来更方便地将相关行为添加进去。另一方面，如果当前类型的行为受到类型码的影响，你就应该尽可能使用Replace Type Code With SubClasses。如果做不到，就只好使用更复杂（同时也更灵活）的Replace Type Code with State/Strategy。

Self Encapsulate Field 自封装字段

你直接访问一个字段，但与字段之间的耦合关系逐渐变得笨拙。为这个字段建立取值/设值函数，并且只以这些函数来访问字段。

动机

在“字段访问方式”这个问题上，存在两种截然不同的观点：其中一派认为，在该变量定义所在的类中，你可以自由访问它；另一派认为，即使在这个类中你也应该只使用访问函数间接访问。两派之间的争论可以说是如火如荼。（参见Auer在[Auer]p.413和Beck在[Beck]上的讨论。）

归根结底，间接访问变董的好处是，子类可以通过覆写一个函数而改变获取数据的途径：它还支持更灵活的数据管理方式，例如延迟初始化（意思是：只有在需要用到某值时，才对它初始化）。

直接访时变量的好处则是：代码比较容易阅读。阅读代码的时候，你不需要停下来说：“啊，这只是个取值函数。”

面临选择时，我总是做两手准备。通常情况下我会很乐意按照团队中其他人的意愿来做。就我自己而言，我比较喜欢先使用直接访问方式，直到这种方式给我带来麻烦为止，此时我就会转而使用间接访问方式。重构给了我改变主意的自由。

如果你想访问超类中的一个字段，却又想在子类中将对这个变量的访问改为一个计算后的值，这就是最该使用Self Encapsulate Field的时候。“字段自我封装”只是第一步。完成自我封装之后，你可以在子类中根据自己的需要随意覆写取值/设值函数。

做法

- 为待封装字段建立取值/设值函数。
- 找出该字段的所有引用点，将它们全部改为调用取值/设值函数。
    - 如果引用点要读取字段值，就将它替换为调用取值函数；如果引用点要给字段赋值，就将它替换为调用设值函数。
    - 你可以暂时将该字段改名，让编译器帮助你查找引用点。
- 将该字段声明为private。
- 复查，确保找出所有引用点。
- 编译，测试。

Replace Data Value with Object 以对象取代数据值

你有一个数据项，需要与其他数据和行为一起使用才有意义。将数据项变成对象。

动机

开发初期，你往往决定以简单的数据项表示简单的情况。但是，随着开发的进行，你可能会发现，这些简单数据项不再那么简单了。比如说，一开始你可能会用一个字符串来表不“电话号码”概念，但是随后你就会发现，电话号码需要“格式化”、“抽取区号”之类的特殊行为。如果这样的数据项只有一两个，你还可以把相关函数放进数据项所属的对象里；但是Duplicate Code坏味道和Feature Envy坏味道很快就会从代码中散发出来。当这些坏味道开始出现，你就应该将数据值变成对象。

做法

- 为待替换数值新建一个类，在其中声明一个final字段，其类型和源类中的待替换数值类型一样。然后在新类中加入这个字段的取值函数，再加上一个接受此字段为参数的构造函数。
- 编译。
- 将源类中的待替换数值字段的类型改为前面新建的类。
- 修改源类中该字段的取值函数，令它调用新类的取值函数。
- 如果源类构造函数中用到这个待替换字段（多半是赋值动作），我们就修改构造函数，令它改用新类的构造函数来对字段进行赋值动作。
- 修改源类中待替换字段的设值函数，令它为新类创建一个实例。
- 编译，测试。
- 现在，你有可能需要对新类使用Change Value to Reference。

Change Value to Reference 将值对象改为引用对象

你从一个类衍生出许多彼此相等的实例，希望将它们替换为同一个对象。将这个值对象变成引用对象。

动机

在许多系统中，你都可以对对象做一个有用的分类：引用对象和值对象。前者就像“客户”、“账户”这样的东西，每个对象都代表真实世界中的一个实物，你可以直接以相等操作符（==,用来检验对象同一性）检查两个对象是否相等。后者则是像“日期”、“钱”这样的东西，它们完全由其所含的数据值来定义，你并不在意 副本的存在，系统中或许存在成百上千个内容为“1/1/2000”的“日期”对象。当然，你也需要知道两个值对象是否相等，所以你需要覆写equals()(以及hashCode())。

要在引用对象和值对象之间做选择有时并不容易。有时候，你会从一个简单的值对象开始，在其中保存少童不可修改的数据。而后，你可能会希望给这个对象加入一些可修改数据，并确保对任何一个对象的修改都能影响到所有引用此一对象的地方。这时候你就需要将这个对象变成一个引用对象。

做法

- 使用Replace Constructor with Factory Method。
- 编译，测试。
- 决定由什么对象负责提供访问新对象的途径。
  - 可能是一个静态字典或一个注册表对象，
  - 你也可以使用多个对象作为新对象的访问点。
- 决定这些引用对象应该预先创建好，或是应该动态创建。
  - 如果这些引用对象是预先创建好的，而你必须从内存中将它们读取出来，那么就得确保它们在被需要的时候能够被及时加载。
- 修改工厂函数，令它返回引用对象。
  - 如果对象是预先创建好的，你就需要考虑：万一有人索求一个其实并不存在的对象，要如何处理措误？
  - 你可能希望对工厂函数使用Rename Method，使其传达这样的信息：它返回的是一个既存对象。
- 编译，测试。

要把一个引用对象变成值对象，关键动作是：检查它是否不可变。如果不是，我就不能使用本项重构，因为可变的值对象会造成烦人的别名问题。

Change Reference to Value 将引用对象改为值对象

你有一个引用对象，很小且不可变，而且不易符理。将它变成一个值对象。

动机

正如我在Change Value to Reference中所说，要在引用对象和值对象之间做选择，有时并不容易。作出选择后，你常会需要一条回头路。

如果引用对象开始变得难以使用，也许就应该将它改为值对象。引用对象必须被某种方式控制，你总是必须向其控制者请求适当的引用对象。它们可能造成内存区域之间错综复杂的关联。在分布系统和并发系统中，不可变的值对象特别有用，因为你无需考虑它们的同步问题。

值对象有一个非常重要的特性：它们应该是不可变的。无论何时，只要你调用同一对象的同一个查询函数，都应该得到同样结果。如果保证了这一点，就可以放心地以多个对象表示同一个事物。如果值对象是可变的，你就必须确保对某一对象的修改会自动更新其他“代表相同事物”的对象。这太痛苦了，与其如此还不如把 它变成引用对象。

这里有必要澄清一下“不可变”（immutable)的意思。如果你以Money类表示 “钱”的概念，其中有“币种”和“金额”两条信息，那么Money对象通常是一个可变的值对象。这并非意味你的薪资不能改变，而是意味：如果要改变你的薪资，就需要使用另一个Money对象来取代现有的Money对象，而不是在现有的Money对象上修改。你和Money对象之间的关系可以改变，但Money对象自身不能改变。

做法

- 检查重构目标是否为不可变对象，或是否可修改为不可变对象。
  - 如果该对象目前还不是不可变的，就使用Remove Setting Method，直到它成为不可变的为止。
  - 如果无法将该对象修改为不可变的，就放弃使用本项重构。
- 建立equals()和hashCode()。
- 编译，测试。
- 考虑是否可以删除工厂函数，并将构造函数声明为public。

Replace Array with Object 以对象取代数组

你有一个数组，其中的元素各自代表不同的东西。以对象替换数组。对于数组中的每个元素，以一个字段来表示。

动机

数组是一种常见的用以组织数据的结构。不过，它们应该只用于“以某种顺序容纳一组相似对象”。有时候你会发现，一个数组容纳了多种不同对象，这会给用户带来麻烦，因为他们很难记住像“数组的第一个元素是人名”这样的约定。对象就不同了，你可以运用字段名称和函数名称来传达这样的信息，因此你无需死记它，也无需依赖注释。而且如果使用对象，你还可以将信息封装起来，并使用Move Method为它加上相关行为。

做法

- 新建一个类表示数组所拥有的信息，并在其中以一个public字段保存原先的数组。
- 修改数组的所有用户，让它们改用新类的实例。
- 编译，测试。
- 逐一为数组元素添加取值/设值函数。根据元素的用途，为这些访问函数命名。修改客户端代码，让它们通过访问函数取用数组内的元素。每次修改后,编译并测试。
- 当所有对数组的直接访问都转而调用访问函数后，将新类中保存该数组的字段声明为private。
- 编译。
- 对于数组内的每一个元素，在新类中创建一个类型相当的字段。修改该元素的访问函数，令它改用上述的新建字段。
- 每修改一个元素，编译并测试。
- 数组的所有元素都有了相应字段之后，删除该数组。

Duplicate Observed Data 复制被监视的数据

你有一些领域数据置身于GUI控件中，而领域函数需要访问这些数据。将该数据复制到一个领域对象中。建立一个Observer模式，用以同步领域对象和GUI对象内的重复数据。

动机

一个分层良好的系统，应该将处理用户界面和处理业务逻辑的代码分开。之所以这样做，原因有以下几点：（1)你可能需要使用不同的用户界面来表现相同的业务逻辑，如果同时承担两种责任，用户界面会变得过分复杂；（2)与GUI隔离之后，领域对象的维护和演化都会更容易，你甚至可以让不同的开发者负责不同部分的开发。

尽管吋以轻松地将“行为”划分到不同部位，“数据”却往往不能如此。同一项数据有可能既需要内嵌于GUI控件，也需要保存于领域模型里。自从MVC(Model-View-Controller,模型-视图-控制器）模式出现后，用户界面框架都使用多层系统来提供某种机制，使你不但可以提供这类数据，并保持它们同步。

如果你遇到的代码是以两层方式开发，业务逻辑被内嵌于用户界面之中，你就有必要将行为分离出来。其中的主要工作就是函数的分解和搬移。但数据就不同了：你不能仅仅只是移动数据，必须将它复制到新的对象中，并提供相应的同步机制。

做法

- 修改展现类，使其成为领域类的Observer[GoF]。
  - 如果尚未有领域类，就建立一个。
  - 如果没有“从展现类到领域类”的关联，就将领域类保存于展现类的一个字段中。
- 针对GUI类中的领域数据，使用Self Encapsulate Field。
- 编译，测试。
- 在事件处理函数中调用设值函数，直接更新GUI组件。
  - 在事件处理函数中放一个设值函数，利用它将GUI组件更新为领域数据的当前值。当然这其实没有必要，你只不过是拿它的值设定它自己。但是这样使用设值函数，便是允许其中的任何动作得以于日后被执行起来，这是 这一步骤的意义所在。
  - 进行这个改变时，对于组件，不要使用取值函数，应该直接取用，因为稍后我们将修改取值函数，使其从领域对象（而非GUI组件）取值。设值函数也将做类似修改。
  - 确保测试代码能够触发新添加的事件处理机制。
- 编译，测试。
- 在领域类中定义数据及其相关访问函数。
  - 确保领域类中的设值函数能够触发Observer模式的通报机制。

    - 对于被观察的数据，在领域类中使用与展现类所用的相同类型（通常是字符串）来保存。后续重构中你可以自由改变这个数据类型。
- 修改展现类中的访问函数，将它们的操作对象改为领域对象（而非GUI组件）。
- 修改Observer的update()，使其从相应的领域对象中将所需数据复制给GUI组件。
- 编译，测试。

Change Unidirectional Association to Bidirectional 将单向关联改为双向关联

两个类都需要使用对方特性，但其间只有一条单向连接。添加一个反向指针，并使修改函数能够同时更新两条连接。

动机

开发初期，你可能会在两个类之间建立一条单向连接，使其中一个类可以引用另一个类。随着时间推移，你可能发现被引用类需要得到其引用者以便进行某些处理。也就是说它需要一个反向指针。但指针是一种单向连接，你不可能反向操作它。通常你可以绕道而行，虽然会耗费一些计算时间，成本还算合理，然后你可以在被引用类中建立一个函数专门负责此一行为。但是，有时候想绕过这个问题并不容易，此时就需要建立双向引用关系，或称为反向指针。如果使用不当，反向指针很容易造成混乱；但只要你习惯了这种手法，它们其实并不是太复杂。

“反向指针”手法有点棘手，所以在你能够自如运用之前，应该有相应的测试。通常我不花心思去测试访问函数，因为普通访问函数的风险没有高到需要测试的地步，但本重构要求测试访问函数，所以它是极少数需要添加测试的重构手法之一。

本重构运用反向指针实现双向关联。其他技术（例如连接对象）需要其他重构手法。

做法

- 在被引用类中增加一个字段，用以保存反向指针。
- 决定由哪个类——引用端还是被引用端——控制关联关系。
- 在被控端建立一个辅助函数，其命名应该清楚指出它的有限用途。
- 如果既有的修改函数在控制端，让它负责更新反向指针。
- 如果既有的修改函数在被控端，就在控制端建立一个控制函数，并让既有的修改函数调用这个新建的控制函数。

Change Bidirectional Association to Unidirectional 将双向关联改为单向关联

两个类之间有双向关联，但其中一个类如今不再需要另一个类的特性。去除不必要的关联。

动机

双向关联很有用，但你也必须为它付出代价，那就是维护双向连接、确保对象被正确创建和删除而增加的复杂度。而且，由于很多程序员并不习惯使用双向关联，它往往成为错误之源。

大量的双向连接也很容易造成“僵尸对象”：某个对象本来已经该死亡了，却仍然保留在系统中，因为对它的引用还没有完全清除。

此外，双向关联也迫使两个类之间有了依赖：对其中任一个类的任何修改，都可能引发另一个类的变化。如果这两个类位于不同的包，这种依赖就是包与包之间的相依。过多的跨包依赖会造就紧耦合系统，使得任何一点小小改动都可能造成许多无法预知的后果。

只有在真正需要双向关联的时候，才应该使用它。如果发现双向关联不再有存在价值，就应该去掉其中不必要的一条关联。

做法

- 找出保存“你想去除的指针”的字段，检查它的每一个用户，判断是否可以去除该指针。
  - 不但要检查直接访问点，也要检查调用这些直接访问点的函数。
  - 考虑有无可能不通过指针取得被引用对象。如果有可能，你就可以对取值函数使用Subsitute Algorithm，从而让客户在没有指针的情况下也可以使用该取值函数。
- 对于使用该字段的所有函数，考虑将被引用对象作为参数传进去。
- 如果客户使用了取值函数，先运用Self Encapsulate Field将待删除字段自我封装起来，然后使用Subsitute Algorithm对付取值函数，令它不再使用该字段。然后编译、测试。
- 如果客户并未使用取值函数，那就直接修改待删除字段的所有被引用点：改以其他途径获得该字段所保存的对象。每次修改后，编译并测试。
- 如果已经没有任何函数使用待删除字段，移除所有对该字段的更新逻辑，然后移除该字段。’
  - 如果有许多地方对此字段赋值，先运用Self Encapsulate Field使这些地点改用同一个设值函数。编译、测试。而后将这个设值函数的本体清空。 再编译、再测试。如果这些都可行，就可以将此字段和其设值函数，连同对设值函数的所有调用，全部移除。

- 编译，测试。

Replace Magic Number with Symbolic Constant 以字面常量取代魔法数

你有一个字面数值，带有特别含义。创造一个常量，根据其意义为它命名，并将上述的字面数值替换为这个常量。

动机

在计算科学中，魔法数（magic number)是历史最悠久的不良现象之一。所谓魔法数是指拥有特殊意义，却又不能明确表现出这种意义的数字。如果你需要在不同的地点引用同一个逻辑数，魔法数会让你烦恼不已,因为一旦这些数发生改变， 你就必须在程序中找到所有魔法数，并将它们全部修改一遍，这简直就是一场噩梦。就算你不需要修改，要准确指出每个魔法数的用途，也会让你颇费脑筋。

许多语言都允许你声明常量。常量不会造成任何性能开销，却可以大大提高代码的可读性。

进行本项重构之前，你应该先寻找其他替换方案。你应该观察魔法数如何被使用，而后你往往会发现一种更好的使用方式。如果这个魔法数是个类型码，请考虑使用Replace Type Code with Class;如果这个魔法数代表一个数组的长度，请在遍历该数组的时候，改用Array.length()。

做法

- 声明一个常量，令其值为原本的魔法数值。
- 找出这个魔法数的所有引用点。
- 检查是否可以使用这个新声明的常量来替换该魔法数。如果可以，便以此常量替换之。
- 编译。
- 所有魔法数都被替换完毕后，编译并测试。此时整个程序应该运转如常，就像没有做任何修改一样。
    - 有个不错的测试办法：检查现在的程序是否可以被你轻松地修改常量值 (这可能意味某些预期结果将有所改变，以配合这一新值。实际工作中并非总是可以进行这样的测试）。如果可行，这就是一个不错的手法。

Encapsulate Field 封装字段

你的类中存在一个public字段。将它声明为private,并提供相应的访问函数。

动机

面向对象的首要原则之一就是封装，或者称为“数据隐藏”。按此原则，你绝不应该将数据声明为public,否则其他对象就有可能访问甚至修改这项数据，而拥有该数据的对象却毫无察觉。于是，数据和行为就被分开了——这可不是件好事。

数据声明为public被看做是一种不好的做法，因为这样会降低程序的模块化程度。数据和使用该数据的行为如果集中在一起，一旦情况发生变化，代码的修改就会比较简单，因为需要修改的代码都集中于同一块地方，而小不是星罗棋布地散落在整个程序中。

Encapsulate Field是封装过程的第一步。通过这项重构手法，你可以将数据隐藏起来，并提供相应的访问函数。但它毕竟只是第一步。如果一个类除了访问函数外不能提供其他行为，它终究只是一个哑巴类。这样的类并不能享受对象技术带来的好处。而你知道，浪费任何一个对象都是很不好的。实施Encapsulate Field之后，我会尝试寻找用到新建访问函数的代码，看看是否可以通过简单的Move Method轻快地将它们移到新对象上。

做法

- 为public字段提供取值/设值函数。
- 找到这个类以外使用该字段的所有地点。如果客户只是读取该字段，就把引用替换为对取值函数的调用：如果客户修改了该字段值，就将此引用点替换为对设值函数的调用。
    - 如果这个字段是个对象，而客户只不过是调用该对象的某个函数，那么无论该函数是否改变对象状态，都只能算是读取该字段.只有当客户为该字段赋值时，才能将其替换为设值函数。
- 每次修改之后，编译并测试。
- 将字段的所有用户修改完毕后，把字段声明为private。
- 编译，测试。

Encapsulate Collection 封装集合

有个函数返回一个集合。让这个函数返回该集合的一个只读副本，并在这个类中提供添加/移除集合元素的函数。

动机

我们常常会在一个类中使用集合（collection,可能是array、list、set或vector）来保存一组实例。这样的类通常也会提供针对该集合的取值/设值函数。

但是，集合的处理方式应该和其他种类的数据略有不同。取值函数不该返回集合自身，因为这会让用户得以修改集合内容而集合拥有者却一无所悉。这也会对用户暴露过多对象内部数据结构的信息。如果一个取值函数确实需要返回多个值，它应该避免用户直接操作对象内所保存的集合，并隐藏对象内与用户无关的数据结构。至于如何做到这一点，视你使用的Java版本不同而有所不同。

另外，不应该为这整个集合提供一个设值函数，但应该提供用以为集合添加/移除元素的函数。这样，集合拥有者（对象）就可以控制集合元素的添加和移除。

如果你做到以上几点，集合就被很好地封装起来了，这便可以降低集合拥有者和用户之间的耦合度。

做法

- 加入为集合添加/移除元素的函数。
- 将保存集合的字段初始化为一个空集合。
- 编译。
- 找出集合设值函数的所有调用者。你可以修改那个设值函数，让它使用上述新建立的“添加/移除元素”函数；也可以直接修改调用端，改让它们调用上述新建立的“添加/移除元素”函数。
  - 两种情况下需要用到集合设值函数：
    1. 集合为空时；
    2. 准备将原有集合替换为另一个集合时。
  - 你或许会想运用Rename Method为集合设值函数改名：从setXxx()改为initializeXxx()或replaceXxx ()。
- 编译，测试。
- 找出所有“通过取值函数获得集合并修改其内容”的函数。逐一修改这些函数，让它们改用添加/移除函数。每次修改后，编译并测试。

- 修改完上述所有“通过取值函数获得集合并修改集合内容”的函数后，修改取值函数自身，使它返回该集合的一个只读副本。
  - 在Java2中，你可以使用Collection.unmodifiableXxx()得到该集合的只读副本。
  - 在Java1.1中，你应该返回集合的一份副本。
- 编译，测试。
- 找出取值函数的所有用户，从中找出应该存在于集合所属对象内的代码。运用Extract Method和Move Method将这些代码移到宿主对象去。
  - 如果你使用Java2，那么本项重构到此为止。如果你使用Java1.1,那么用户也许会喜欢使用枚举。为了提供这个枚举，你应该像如下这样做。
- 修改现有取值函数的名字，然后添加一个新取值函数，使其返回一个枚举。找出旧取值函数的所有被使用点，将它们都改为使用新取值函数。
- 如果这一步跨度太大，你可以先使用Rename Method修改原取值函数的名称；再建立一个新取值函数用以返回枚举；最后再修改所有调用者，使其调用新取值函数。
- 编译，测试。

Replace Record with Data Class 以数据类取代记录

你需要面对传统编程环境中的记录结构。为该记录创建一个“哑”数据对象。

动机

记录型结构是许多编程环境的共同性质。有一些理由使它们被带进面向对象程序之中：你可能面对的是一个遗留程序，也可能需要通过一个传统API来与记录结构交流，或是处理从数据库读出的记录。这些时候你就有必要创建一个接口类，用以处理这些外来数据。最简单的做法就是先建立一个看起来类似外部记录的类，以便日后将某些字段和函数搬移到这个类之中。一个不太常见但非常令人注目的情况是：数组中的每个位置上的元素都有特定含义，这种情况下应该使用Replace Array with Object。

做法

- 新建一个类，表示这个记录。
- 对于记录中的每一项数据，在新建的类中建立对应的一个private字段，并提供相应的取值/设值函数。

现在，你拥有了一个“哑”数据对象。这个对象现在还没有任何有用的行为，但是更进一步的重构会解决这个问题。

Replace Type Code with Class 以类取代类型码

类之中有一个数值类型码，但它并不影响类的行为。以一个新的类替换该数值类型码。

动机

在以C为基础的编程语言中，类型码或枚举值很常见。如果带着一个有意义的符号名，类型码的可读性还是不错的。问题在于，符号名终究只是个别名，编译器看见的、进行类型检验的，还是背后那个数值。任何接受类型码作为参数的函数，所期望的实际上是一个数值，无法强制使用符号名。这会大大降低代码的可读性，从而成为bug之源。

如果把那样的数值换成一个类，编译器就可以对这个类进行类型检验。只要为这个类提供工厂函数，你就可以始终保证只有合法的实例才会被创建出来，而且它们都会被传递给正确的宿主对象。

但是，在使用Replace Type Code with Class之前，你应该先考虑类型码的其他替换方式。只有当类型码是纯粹数据时（也就是类型码不会在**switch**语句中引起行为变化时），你才能以类来取代它。Java只能以整数作为switch语句的判断依据，不能使用任意类，因此那种情况下不能够以类替换类型码。更重要的是：任何switch语句都应该运用Replace Conditional with Polymorphism去掉。为了进行那样的重构，你首先必须运用Replace Type Code with Subclasses或Replace Type Code with State/Strageygy，把类型码处理掉。

即使一个类型码不会因其数值的不同而引起行为上的差异，宿主类中的某些行为还是有可能更适合置放于类型码类中，因此你还应该留意是否有必要使用Move Method将一两个函数搬过去。

做法

- 为类型码建立一个类。
  - 这个类需要一个用以记录类型码的字段，其类型应该和类型码相同，并应该有对应的取值函数。此外还应该用一组静态变量保存允许被创建的实例，并以一个静态函数根据原本的类型码返回合适的实例。
- 修改源类实现，让它使用上述新建的类。
  - 维持原先以类型码为基础的函数接口，但改变静态字段，以新建的类产生代码。然后，修改类型码相关函数，让它们也从新建的类中获取类型码。
- 编译，测试。
  - 此时，新建的类可以对类型码进行运行期检查。

- 对于源类中每一个使用类型码的函数，相应建立一个函数，让新函数使用新建的类。
  - 你需要建立“以新类实例为自变量”的函数，用以替换原先“直接以类型码为参数”的函数。你还需要建立一个“返回新类实例”的函数，用以替换原先“直接返回类型码”的函数。建立新函数前，你可以使用Rename Method修改原函数名称，明确指出哪些函数仍然使用旧式的类型码，这往往是个明智之举。
- 逐一修改源类用户，让它们使用新接口。
- 每修改一个用户，编译并测试。
  - 你也可能需要一次性修改多个彼此相关的函数，才能保持这些函数之间的一致性，才能顺利地编译、测试。
- 删除使用类型码的旧接口，并删除保存旧类型码的静态变量。
- 编译，测试。