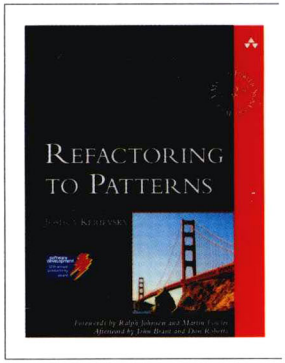
PEARSON

软件开发方法学精选系列



Refactoring to Patterns

software development 15th annual productivity award

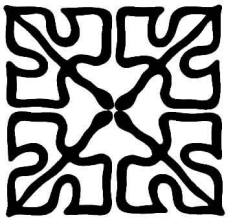
[美] Joshua Kerievsky 著

杨光 刘基诚 译

重构与模式

（修订版）

人民邮电出版社
POSTS & TELECOM PRESS



软件开发方法学精选系列

Refactoring to Patterns

[美] Joshua Kerievsky 著

杨光 刘基诚 译

重构与模式

（修订版）

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

重构与模式 /（美）科瑞福斯凯（Kerievsky,J.）著；杨光，刘基诚译. -- 修订本. -- 北京 : 人民邮电出版社，2013.1
（软件开发方法学精选系列）
书名原文：Refactoring to Patterns
ISBN 978-7-115-29725-9

Ⅰ. ①重… Ⅱ. ①科… ②杨… ③刘… Ⅲ. ①软件开发－研究 Ⅳ. ①TP311.52

中国版本图书馆CIP数据核字(2012)第242502号

内 容 提 要

本书开创性地深入揭示了重构与模式这两种软件开发关键技术之间的联系，说明了通过重构实现模式改善既有的设计，往往优于在新的设计早期使用模式。本书不仅展示了一种应用模式和重构的创新方法，而且有助于读者透过实战深入理解重构和模式。书中讲述了27种重构方式。

本书适于面向对象软件开发人员阅读，也可作为高等学校计算机专业、软件工程专业师生的参考读物。

软件开发方法学精选系列

重构与模式（修订版）

◆ 著 [美] Joshua Kerievsky
译 杨 光 刘基诚
责任编辑 杨海玲

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：20 2013年1月第1版
字数：444千字 2013年1月北京第1次印刷

著作权合同登记号 图字：01-2012-7106号

ISBN 978-7-115-29725-9

定价：55.00元

读者服务热线：(010)67132692 印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第0021号

目　录

第 1 章 本书的写作缘由

软件模式的伟大之处，就在于它们传达了许多有用的设计思想。所以，在学习了大量模式之后，就理应成为非常优秀的软件设计人员，不是吗？当学习、使用了几十个模式后，我也曾这样认为。模式帮助我开发灵活的框架，帮助我构建坚固、可扩展的软件系统。但是几年后，我却发现自己在模式方面的知识和使用模式的方式总是使我在工作中犯过度设计的错误。

设计技术进一步提高之后，我发现自己使用模式的方式逐渐发生了变化：我开始“通过重构实现模式、趋向模式和去除模式（refactoring to，towards，and away from pattern）”，而不再是在预先（up-front）设计中使用模式，也不再过早地在代码中加入模式。这种使用模式的新方式来自于我对极限编程（XP）设计实践的采用，它帮助我既避免了过度设计，又不至于设计不足。

1.1 过度设计

所谓**过度设计**（over-engineering），是指代码的灵活性和复杂性超出所需。有些程序员之所以这样做，是因为他们相信自己知晓系统未来的需求。他们推断，最好今天就把方案设计得更灵活、更复杂，以适应明天的需求。这听上去很合理，但是别忘了，这需要你未卜先知。

如果预计错误，浪费的将是宝贵的时间和金钱。花费几天甚至几星期对设计方案进行微调，仅仅为了增加过度的灵活性或者不必要的复杂性，这种情况并不罕见，而且这样只会减少用来添加新功能、排除系统缺陷的时间。

如果预期中的需求根本不会成为现实，那么按此编写的代码又将怎样呢？删除是不现实的。删除这些代码并不方便，何况我们还指望着有一天它们能派上用场呢。无论原因如何，随着过度灵活、过分复杂的代码的堆积，你和团队中的其他程序员，尤其是那些新成员，就得在毫无必要的更庞大、更复杂的代码基础上工作了。

为了避免这一问题，人们决定分头负责系统的各个部分。这看似能使工作更容易，但是副作用又产生了。因为每个人都在自己的小天地里工作，很少看看别处的代码是否已经完成了自己需要的功能，最后生成大量重复的代码。

过度设计下的代码会影响生产率，因为当其他人接手一个过度设计的方案时，必须先花上一

些时间了解设计中的许多微妙之处，然后才能自如地扩展或者维护它。

过度设计总在不知不觉之中出现，许多架构师和程序员在进行过度设计时甚至自己都不曾意识到。而当公司发现团队的生产率下降时，又很少有人知道是过度设计在作怪。

程序员之所以会过度设计，也许是因为他们不想受不良设计的羁绊。不良的设计可能会深深地融入代码之中，对其进行改进不啻严峻的挑战。我遇到过这种情况，所以使用模式预先进行设计对我的吸引力才会如此之大。

1.2　模式万灵丹

最初学习模式时，它们代表的是我很想精通的一种灵活、精妙甚至非常优雅的面向对象设计方法。完整地学习了无数的模式和模式语言之后，我用它们改进以前开发的系统，用它们构思将要开发的系统。其效果非常可观，我知道，自己的路子走对了。

然而，随着时间的推移，模式的强大使我开始对更简单的代码编写方式视而不见。只要遇到某个可以使用两三种不同方法进行的计算，我就会很快想到实现 Strategy 模式，而事实上，使用简单的条件表达式编程更加容易，也更加快捷，完全足够。

有一次，我对模式的走火入魔可以说是暴露无遗。在结对编程中，我和搭档编写了一个类，它实现了 Java 的 TreeModel 接口，在树型窗口部件（widget）中显示 Spec 对象的图形。代码能够工作，但是树型窗口部件通过调用 Spec 对象的 toString()方法来显示它们，而该方法并不返回需要的 Spec 对象信息。我们不能修改 Spec 的 toString()方法，因为系统的其他部分还要用到这个方法。我们只好慎重思考如何继续。和往常一样，我开始考虑哪个模式能够助我们一臂之力。脑子里浮现出 Decorator 模式。我建议，按照这个模式用一个对象封装 Spec 对象，再重写（override）这个对象的 toString()方法。搭档对这条建议的反应使我大吃一惊："在这里用 Decorator 模式？那不等于大炮打蚊子吗？"他的解决方案是，创建一个名为 NodeDisplay 的很小的类，将 Spec 对象作为其构造函数的参数，它的一个公共方法 toString()包含了 Spec 对象的正确显示信息。NodeDisplay 类编写起来几乎花不了多少时间，因为它的代码不超过 10 行。而我使用 Decorator 模式的解决方案至少需要 50 行代码，需要多次反复委托调用 Spec 对象。

这样的经验使我意识到，再也不能过多地考虑模式了，应该重新把精力放在短小、简单和直截了当的代码上。我走到了一个十字路口：我努力学习模式，想成为更优秀的软件设计师，然而，现在为了真正更上一层楼，我需要放弃对它们的依赖。

1.3　设计不足

设计不足比过度设计要常见得多。所谓**设计不足**（under-engineering），是指所开发的软件设计不良。其产生原因有如下几种：

- 程序员没有时间，没有抽出时间，或者时间不允许进行重构；
- 程序员在何为好的软件设计方面知识不足；
- 程序员被要求在既有系统中快速地添加新功能；
- 程序员被迫同时进行太多项目。

随着时间的推移，设计不足的软件将变成昂贵、难以维护甚至无法维护的大麻烦。Brian Foote 和 Joseph Yoder 曾经创造了一种名为 Big Ball of Mud（大泥球）的模式语言，他们是这样描述类似软件的。

数据结构的构造非常随意，甚至近乎不存在。任何东西都要与其他东西通信。所有重要的状态数据都可能是全局的。在状态信息被隔开的地方，需要通过错综复杂的后端通道杂乱地传递，以绕开系统的原有结构。

变量名和函数名信息量不足，甚至会起误导作用。函数可能使用大量全局变量以及定义模糊的冗长的参数列表。函数本身冗长、费解，完成多项毫无关联的任务。代码重复很多。控制流很难看清，难以找到来龙去脉。程序员的意图几乎无法理解。代码完全不可读，近乎难于破译的天书。代码中有许多经过多个维护者之手不断修修补补留下的明显印记，这些维护者几乎都没有理解自己的修补会造成怎样的后果。[Foote and Yoder, 661]

虽然你开发的系统也许不会这么恐怖，但是很可能也曾经有过设计不足的时候。我知道自己肯定这样干过。迅速使代码运行起来是压倒一切的要求，而这往往伴随着巨大的压力，使我们无法改进既有代码的设计。有些情况下，我们会有意地不对代码进行改进，因为我们知道（或者自认为知道）软件的生命期不会太长。而另一些时候，是别人迫使我们不对代码进行改进，因为好心的经理会这样说："不出事的地方就不用改了，这样我们公司能够更有竞争力，更可能在市场竞争中取胜。"

长期的设计不足，会使软件开发节奏变成"快、慢、更慢"，可能造成这样的后果：

(1) 系统的 1.0 版很快就交付了，但是代码质量很差；

(2) 系统的 2.0 版也交付了，但质量低劣的代码使我们慢了下来；

(3) 在企图交付未来版本时，随着劣质代码的倍增，开发速度也越来越慢，最后人们对系统、程序员乃至使大家陷于这种境地的整个过程都失去了信心；

(4) 到了 4.0 版时或者之后，我们意识到这样肯定不行，开始考虑推倒重来。

这种事情在我们的行业里司空见惯。它的代价非常高昂，而且会极大地降低企业本应具备的竞争力。幸运的是，我们还有更光明的道路可走。

1.4　测试驱动开发和持续重构

测试驱动开发[Beck, TDD]和持续重构，是极限编程诸多优秀实践中的两个，它们彻底改进了我开发软件的方式。我发现，这两个实践能够帮助我和公司降低过度设计和设计不足的几率，

将时间用在按时地构造出高质量、功能丰富的代码上。

通过测试驱动开发（TDD）和持续重构，我们将编程变成一种对话[①]，从而高效地使可以工作的代码不断演变。

- 问：编写一个测试，向系统提问。
- 答：编写代码通过这个测试，回答这一提问。
- 提炼：通过合并概念、去芜存菁、消除歧义，提炼你的回答。
- 反复：提出下一个问题，继续进行对话。

这种编程节奏使我耳目一新。通过使用测试驱动开发，我们再也不用先花大量时间仔细考虑一个设计，就能够应付系统的每个细枝末节了。现在，我可以用几秒钟或者几分钟，先让原始的功能正确地工作起来，然后再重构，使它不断演进，达到必需的复杂程度。

Kent Beck 为测试驱动开发和持续重构创造了一句“咒语”：“红、绿、重构”。其中的“红”和“绿”是指在单元测试工具（比如 JUnit）中编写并运行一个测试时所看到的颜色。整个过程是下面这样的。

(1) 红：创建一个测试，表示代码所要完成的任务。在编写的代码能够通过测试之前，测试将失败（显示红色）。

(2) 绿：编写一些权宜代码，先通过测试（显示绿色）。这时，你用不着为难自己，非要给出没有重复、简单和清晰的设计。可以在测试通过、能够心安理得地尝试更好的设计之后，再逐步朝这个目标努力。

(3) 重构：对已经通过测试的代码，改进其设计。

听上去就这么简单，测试驱动开发和持续重构使编程领域面目一新。那些缺乏经验的程序员可能会这样想：“什么？为还不存在的代码编写测试？编写的代码通过测试之后，还需要立即进行重构？这不就是那种浪费很大、杂乱无章的软件开发方式吗？”

实际上，事情恰恰相反。测试驱动开发和持续重构提供了一种精益、迭代和训练有素的编程风格，能够最大程度地有张有弛，提高生产率。Martin Fowler 称之为“迅速而又从容不迫”[Beck, TDD]，而 Ward Cunningham 则解释说，这种说法主要指的是持续分析和设计，与测试关系不大。

5

程序员需要从实践中学习测试驱动开发和持续重构的正确节奏。我认识的一位程序员 Tony Mobley 曾称这种开发风格为一次范型转变，其影响之巨，不亚于结构化程序设计到面向对象程序设计的转变。无论你需要多长时间来适应这种开发风格，一旦习惯之后，你将发现，再用其他任何方式开发成品代码，都会感觉奇怪、不舒服甚至非常业余。许多使用测试驱动开发和持续重构编程的人，都发现这种方式有助于：

① 对话这个隐喻出自 Kent Beck，借用了大哲学家苏格拉底的对话教学方式。编写测试代码就好像是向系统提问题，编写系统代码是为了回答问题，这样的对话不断反复，最后生成的就是我们所需要的系统。——译者注

- 保持较低的缺陷数量；
- 大胆地进行重构；
- 得到更简单、更优秀的代码；
- 编程时没有压力。

要了解测试驱动开发的细节，请研读 *Test-Driven Development* [Beck, TDD]或者 *Test-Driven Development: A Practical Guide* [Astels]两部著作。要对测试驱动开发有感性认识，可以参见本书的 7.5.3 节和 6.5.2 节。要了解如何持续重构，请研读《重构》[F]一书（尤其是第 1 章）以及本书中的重构内容。

1.5　重构与模式

我观察了自己和同事们在许多项目中重构的对象和方式。在使用《重构》[F]一书中描述的许多重构方法时，我们还发现模式有助于改进设计。很多次我们都是通过重构实现模式，或者通过趋向模式进行重构，小心翼翼地避免产生过分灵活或者过度复杂的方案。

深入研究了应用“模式导向的重构”的动机之后，我发现它和“实现低层次重构”的一般动机是一样的：减少或去除重复的地方，简化复杂之处，使代码更好地表达其意图。

但是，如果只学习某个设计模式的一部分，很容易忽视这种动机。例如，《设计模式》[DP] 中的所有模式都包含一个名为“意图”的部分。《设计模式》的作者们是这样描述意图的：“意图是回答下列问题的简单陈述：设计模式是做什么的？它的基本原理和意图是什么？它解决的是什么样的特定设计问题？”[DP, 6]话虽如此，但是许多设计模式的“意图”部分只是在说明模式解决的主要问题。相反，更多的注意力放在了“模式是做什么的”之上。

我们来看两个例子。

Template Method（模板方法）的意图

定义一个操作中算法的骨架，而将一些步骤延迟到子类中。Template Method 使得子类可以在不改变算法结构的情况下，重定义该算法的某些特定步骤。[DP, 325]

State（状态）的意图

允许一个对象在其内部状态改变时改变自己的行为。对象看起来似乎修改了自己的类。[DP, 315]

这些意图描述并没有说明 Template Method 有助于减少或者去掉类层次中各个子类里相似方法中的重复代码，也没有说明 State 模式有助于简化复杂的有条件的状态改变逻辑。如果程序员学习了一个设计模式的所有部分，尤其是“适用性”部分，他们将了解到该模式所要解决的问题是什么。

但是，在设计中使用《设计模式》一书时，许多程序员，包括我自己，都是通过阅读模式的“意图”部分，确定这个模式是否适合当前的情况。这种选择模式的方法的有效性不如将设计问题与模式能够解决的问题进行比对。为什么呢？因为模式之所以存在，就是为了解决问题，所以要了解在某种情况下模式是否真的有所帮助，必须理解它们有助于解决什么问题。

重构方面的文献似乎比模式方面的文献更关注具体的设计问题。开始学习某个重构时，你会在书的第一页看到重构有助于解决何种问题。本书给出的“模式导向的重构”目录直接延续了《重构》一书中所开创的工作，其目的是帮助读者了解模式有助于解决哪些具体的问题。

本书架设了模式和重构之间的桥梁，但是，其实《设计模式》一书的作者们在其皇皇巨著的“结论”一章已经提到了这两者之间的联系：

我们的设计模式记录了许多重构产生的设计结构。……设计模式为你的重构提供了目标。[DP, 354]

Martin Fowler 在《重构》一书的开始也有类似的说明：

模式和重构之间存在着天然联系。模式是你想到达的目的地，而重构则是从其他地方抵达这
7 *个目的地的条条道路。*[F, 107]

1.6　演进式设计

今天，在对模式——这种“重构产生设计结构”已经非常熟悉之后，我了解到充分理解为什么要“通过重构实现模式或者重构趋向模式”，比理解应用模式的结果或者结果的实现细节更有价值。

如果想成为一名更优秀的软件设计师，了解优秀软件设计的演变过程比学习优秀设计本身更有价值，因为设计的演变过程中隐藏着真正的大智慧。演变所得到的设计结构当然也有帮助，但是不知道设计是怎么发展而来的，在下一个项目中你就很可能错误地应用，或者陷入过度设计的误区。

迄今为止，我们关于软件设计的文献更多地集中在讲授优秀的解决方案上，对这些解决方案的演变过程则重视不够。这种情况需要改变。正如伟大的诗人歌德说过的：“那些父辈们传下来的东西，如果你能拥有它，你就能重新得到它们。”重构方面的文献通过揭示优秀设计方案的演化过程，帮助我们更好地重新理解这些方案。

如果想发挥模式的最大效用，也必须这样做：将模式放到重构的背景中领会，而不是仅仅将模式视为与重构无关的可复用的要素。这恐怕就是我编写“模式导向的重构”目录的主要原因。

通过学习不断改进设计，你就能够成为一名更优秀的软件设计师，并且减少工作中过度设计和设计不足的情况。测试驱动开发和持续重构是演进式设计的关键实践。将“模式导向的重构”
8 的概念注入如何重构的知识中，你会发现自己如有神助，能够不断地改进并得到优秀的设计。

第2章 重 构

本章中我将就“何谓重构”和“需要怎样做才能善于重构”提出一些看法。本章最好和《重构》一书 [F]中的“重构原则”（Principles in Refactoring）[①]一起阅读。

2.1 何谓重构

重构就是一种“保持行为的转换”，或者如 Martin Fowler 定义的那样：“（重构）是一种对软件内部结构的改善，目的是在不改变软件的可见行为的情况下，使其更易理解，修改成本更低。”[F, 53]

重构过程包括去除重复、简化复杂逻辑和澄清模糊的代码。重构时，需要对代码无情地针砭，以改进其设计。这种改进可能很小，小到只是改变一个变量名；也可能很大，大到合并两个类层次。

要保证重构的安全性，确保所做的修改不会产生任何破坏则必须手工测试或者运行自动测试。如果能够快速地运行自动测试，确保（修改后）代码仍能工作，你就能更大胆地进行重构，更乐于尝试试验性的设计。

循序渐进地进行重构有助于防止增加缺陷。大多数重构过程都需花费一些时间。有些大型重构可能需要持续数天、数周甚至数月，才能完成转换。但是即便这样的大型重构也是循序渐进地实现的。

重构最好是持续而不是分阶段地进行。只要看到代码需要改善，就改善它。但是，如果你的经理要求你完成某项功能，以备明天进行演示之用，那么当然应该先完成这项功能，以后再进行重构。持续重构能够很好地满足业务需求，但是重构实践必须和谐地适应业务上的轻重缓急。

① 即该书的第 2 章。——译者注

2.2　重构的动机

虽然我们对代码进行重构的原因很多，但是以下这些动机是最具普遍性的。

使新代码的增加更容易

在系统中增加新功能时，可以选择快速编写出这个功能，而不考虑它是否能很好地适应原有设计，也可以选择对原有设计进行修改，从而能够容易和从容地接纳新功能。如果选择前者，就会导致所谓的设计欠账（参见 2.7 节），迟早还是要通过重构来偿还；如果选择后者，则需先分析为了更好地接纳新功能，需进行哪些修改，然后再进行必要的修改。这两种选择无所谓好坏。如果时间紧张，可能应该先快速地添加新功能，以后再进行重构；如果时间充裕，或者你认为在编写新功能之前先为它做好准备更快，那么就尽管在增加新功能之前进行重构。

改善既有代码的设计

通过持续改善代码的设计，代码将越来越容易处理。这与通常所见情况——重构很少，大量精力都花在快速而短视地增加新功能上，形成鲜明的对比。持续重构包括不断地嗅探代码的坏味（参见第 4 章），一旦发现坏味就立即（或者很快）去除。如果能够养成持续重构的“卫生”习惯，你将发现，代码的扩展和维护更加容易，从而更加享受工作。

对代码理解更透彻

有时，我们读代码时会对它的功能和机理毫无头绪。就算现在有人能够站在旁边进行解说，以后别人再来阅读这段代码时，还是会一头雾水。对这种代码是否该加一些注释呢？非也。如果代码本身不清晰，就说明存在坏味，需要通过重构清除，而不是用注释来掩饰。

在重构这种代码时，如果有完全理解代码的人在场是最好不过了。如果无法到场，看看他能否通过电子邮件、即时通信或者电话进行解释。如果这也不行，那就先重构你能够理解的部分。
10 最终，随着重构的进行，这段代码会越来越容易理解。

提高编程的趣味性

我经常反躬自问是什么促使自己重构代码。当然，我会说重构能够拨冗删繁，能够简化或者澄清代码。可真正促使我进行重构的是什么呢？情绪。我之所以经常重构，只是要使编写的代码不那么讨厌！

我曾经参与了一个存在很多重大设计欠账的项目。尤其是其中有一个职责过多的巨类。因为我们的工作很大程度就是在修改这个巨类，所以每次签入（check in）代码（这是经常的事情，因为我们采用了持续集成），都不得不处理涉及这个巨类的复杂合并。结果，每个人都不得不花更多的时间集成代码。这真让人讨厌！因此我和另一位程序员花了 3 个星期，将这个巨类拆分为多个小类。这是一项艰苦的工作，但是不得不做。大功告成之后，代码的集成时间大大缩短，整个团队的编程体验也大大改善。

2.3 众目睽睽

当《独立宣言》还在起草时，本杰明·富兰克林坐在托马斯·杰弗逊的身边，把杰弗逊关于“我们认为这些真理是神圣的，毋庸置疑”的措词修改为现在非常著名的句子“我们认为这些真理是不言而喻的”。根据传记作家 Walter Isaacson 的说法，杰弗逊对富兰克林的改动暴怒不已。富兰克林意识到朋友的情绪很激动，所以给他讲述了另一位朋友约翰·汤普森（John Thompson）的故事。

约翰·汤普森刚刚开始从事制帽业，想为自己的公司设计一个标记。他设计出如下图所示的标记：

John Thompson，帽商，
制作和销售帽子，现金支付

在启用新标记前，约翰决定给几个朋友看看，听听他们的意见。第一个朋友觉得“帽商”这个词有些重复，没有必要，因为后面的话“制作……帽子”，已经说明了约翰是一个帽商。于是“帽商”这个词被删除了。第二个朋友认为，“制作”一词可以不要，因为顾客不会关心到底是谁制作了帽子。于是“制作”一词也被删去。第三个朋友说，他认为“现金支付”毫无用处，因为不会有顾客赊账买帽子，一般人们都会用现金来买。所以这些词也被删去。

现在标记变成了：“John Thompson 销售帽子。”

“销售帽子！”他的另一个朋友说，“哎呀，没人认为你会给他发帽子的。这个词有什么用处呢？”于是“销售”被删去了。这时“帽子”这个词显然也没什么用处了，因为标记里已经有了帽子的图形。所以标记最终被简化成：

John Thompson

在 *Simple and Direct* 一书中，Jacques Barzun 阐释到，所有优秀的著述，都是不断修改而成的[Barzun, 227]。他指出，修改意味着重新审视。John Thompson 的标记在他的朋友们不断修改之下，去除了重复的文字，简化了语言，澄清了意图。富兰克林之所以要修改杰弗逊的句子，是因为他看到有更简明、更佳的方式来表达杰弗逊的意图。某个人的工作如果能经过多个人进行修改，将得到显著的改进。

这对代码而言亦然。要得到最佳的重构结果，需要多人的帮助。这正是极限编程建议采用结对编程和代码集体所有这两种实践的原因之一[Beck, XP]。

2.4　可读性好的代码

我常常遇到一些给我留下深刻印象的代码，以至于我会在数月乃至数年中不断地向人们讲述。我研究 Ward Cunningham[①]所写的一段代码时，就碰到这种情况。也许有读者尚不知道 Ward 是何许人也，但是应该知道他诸多杰出的创新。Ward 创造了 CRC（Class-Responsibility-Collaboration，类-职责-协作）卡、Wiki Web（维基网站，一种简单快速的可读写网站）、极限编程和 FIT 测试框架（http://fit.c2.com）。

我所研究的代码来自某个重构研讨班上使用的虚构的工资系统。作为这个研讨班的教师之一，我需要在教学之前先对代码研究一番。我先浏览了测试代码。研究的第一个测试方法是根据日期检查工资额。立即映入眼帘的是日期。代码是这样写的：

12

```
november(20, 2005)
```

这行代码调用了以下方法：

```
public void Date november(int day, int year)
```

我既惊又喜。即使是在测试代码中，Ward 也尽心竭力地编写了可读性极佳的方法。如果他不这么费心写出如此简单、容易理解的代码，完全可以写成这样：

```
java.util.Calendar c = java.util.Calendar.getInstance();
c.set(2005, java.util.Calendar.NOVEMBER, 20);
c.getTime();
```

虽然上面的代码也能产生同样的日期，但是它无法完全像 Ward 的 november()方法那样做到以下两点：

- 读起来像自然语言；
- 将重要代码与分散注意力的代码分离开来。

我再来讲一个与此大相径庭的故事。有一个名叫 w44()的方法。我是在为一家大型华尔街银行开发的贷款风险估算程序（一堆 Turbo Pascal 大杂烩代码）中发现这个 w44()方法的。当时，我刚刚开始自己的职业程序员生涯，最初的三个星期都花在研究这个代码沼泽上。终于，我弄明白这里的“44”是逗号的 ASCII 码，而“w”则代表“with”。所以这位程序员的 w44()是指其例程返回的是一个数字，格式化为一个带[②]逗号的字符串。这可真够直观的！我怀疑这位程序员如果不是确实想不出其他好名字，就肯定是想让别人无法接手，以保住自己的铁饭碗。

Martin Fowler 说得好：

任何傻瓜都会编写计算机能理解的代码。好的程序员能够编写人能够理解的代码。[F, 15]

① Ward Cunningham 曾与 Kent Beck 共事多年。除了上述贡献之外，他还是将模式思想引入软件开发的先驱之一。——译者注

② 即 with。——译者注

2.5 保持清晰

保持代码清晰类似于保持房间整洁。一旦你的房间变得乱七八糟，就更难清理了。变得越乱，就越不想清理。

假定你对房间进行了大扫除。接下来该怎么办呢？如果想保持房间整洁，就不能把东西（比如那些臭袜子）扔在地板上，或者让书、杂志、眼镜和玩具堆在桌面上。必须保持卫生。

你经历过这些吗？我经历过。如果几个星期之内都能保持房间整洁，那么保持卫生就开始成为一种习惯，以后就不会再为应该把臭袜子扔在地板上还是收到洗衣篮里进行思想斗争了。我的习惯会驱使着我将袜子收进洗衣篮。

糟糕的是，新习惯往往有让步于旧习惯的危险。比如，哪天你累得已经顾不上收拾地板上的衣服，然后又有几本书被一个蹒跚学步的孩子从书架上碰到桌上。在你意识到之前，你的房间又归于一片狼藉。

要保持代码清晰，必须持续地去除重复，简化和澄清代码。决不能容忍代码中的脏乱，决不能倒退到坏习惯中去。清晰的代码能产生更好的设计，而更好的设计将使开发过程更加迅速，从而会使客户和程序员皆大欢喜。请保持代码的清晰吧！

2.6 循序渐进

从前，有一位年轻而又聪明的程序员参加了我教授的一个测试和重构强化培训班。这个培训班中的每个人都要参与编程练习，重构一些本书和《重构》[F]一书中所叙述的几乎所有坏味（参见第4章）的代码。在练习过程中，结对的程序员们必须发现一个坏味，找到去除这个坏味的重构办法，然后在班上其他人注视下，接上投影仪编程演示这个重构。

正午前大约5 min时，培训班进行了将近一小时的重构。因为午餐已经送到教室，我询问是否有人有一个比较小的重构，能够在午餐休息之前完成。这时那位年轻的程序员举起手，说他想到一个小的重构。他并没有说到具体的坏味或者相关的重构，而是描述了代码中的一个很大的问题，并且解释了他的解决设想。另外的几个学员提出疑问，说这样的问题不可能在5 min内解决。但是，年轻的程序员还是坚持自己能够完成这项任务，于是我们同意让他和结对搭档试一试。

如果要重构复杂的代码，5 min很快就会过去。

年轻的程序员和他的搭档发现，在搬移和修改了一些代码之后，许多单元测试都无法通过。在单元测试工具中，失败的单元测试显示为一个大大的红条，被投影仪投射到大屏幕上时，看上去更是又大又红。两位程序员全力修改失败的单元测试时，大家陆续离座，到旁边的桌子边吃午餐。15 min后，我也开始午餐休息了。在排队拿午餐时，我仍然注视着大屏幕上的编程进展。

20 min过去了，大大的红条仍然没有变绿（表示所有测试都已经通过）。这时，那位年轻程