Refactoring – Thinking and Practice

Start simple and evolve

Draft version 0.1

2001/10/14

Shiyiyang

CTO of Life High Tech Co. Ltd.

ZheJiang University.

All Rights Reserved by the author.

If you have any question or comments , please [mail to me](#) or discuss in erptao forum.

Goto <http://www.erptao.org>,you can get last update version of this document there.

目录

第一部分、WHAT IS REFACTORIZING?	1
介绍	1
什么是 REFACTORIZING?	1
REFACTORIZING 原则	2
<i>Two Hats (两顶帽子)</i>	2
<i>Unit Test</i>	2
<i>Small step</i>	4
<i>Code Review 和 Pair Programming</i>	6
<i>The Rule of Three</i>	6
好处	6
简化测试	6
更简单的设计	7
<i>Refactoring makes software easier to understand</i>	7
<i>Refactoring improves the design of software</i>	8
<i>Refactoring help you find Bugs</i>	9
<i>Refactoring Helps you program faster</i>	9
应用 REFACTORIZING 需要考虑的问题	9
数据库	10
接口改变和 <i>Published Interface</i>	10
用 <i>Refactoring</i> 思想武装自己的设计	11
编程语言	11
支持 <i>Refactoring</i> 的语言特点和编程风格	11
使 <i>Refactoring</i> 复杂化的语言特点和编程风格	12
不应 REFACTORIZING 的场合	13
程序原型	13
<i>Doesn't work</i>	14
<i>Close to deadline</i>	14
实施 REFACTORIZING 可能碰到的阻碍以及解决方案	14
学习	15
<i>Refactoring</i> 获得短期效益	15
削减 <i>Refactoring</i> 的额外开销	16
安全 <i>Refactory</i>	16
REFACTORIZING 和方法	16
增量迭代	16
<i>Moving Target</i>	18
拥抱变化— <i>Refactoring</i> 和 <i>XP</i>	19
REFACTORIZING 和设计	20
简化设计	20
Why should design be simple?	20
What is simple design?	21
How does refactoring support simple upfront design?	22
设计模式	23
简单设计的要求	23

Refactoring 的目标.....	23
何时进行 REFACTORING	24
<i>场合</i>	24
增加功能	24
重用	24
修正 bug	24
Code Review	25
<i>味道</i>	25
REFACTORING 和命名	27
<i>类命名</i>	28
简单超类名—传达设计目的.....	28
全称子类名—区分异同.....	28
<i>方法命名—揭示意图</i>	29
<i>实例（临时）变量命名—暗示角色</i>	29

第一部分、What is Refactoring?

介绍

代码太容易变坏。代码总是趋向于有更大的类、更长的方法、更多的开关语句和更深的条件嵌套。重复代码随处可见，特别是那些初看相似细看又不同的代码泛滥于整个系统：条件表达式，循环结构、集合枚举....信息被共享于系统一些关系甚少的组成部分之间，通常，这使得系统中几乎所有的重要信息都变成全局或者重复。你根本不能看到这种代码还有什么良好的设计。(如果有的话，也已经不可辨识了。)

这样的代码难以理解，更不要说对它加以修改。如果你关心系统体系结构、设计，或者是一个好程序，你的第一反应就是拒绝工作于这样的代码。你会说：“这么烂的代码，让我修改，还不如重写。”然而，你不大可能完全重写已经能够甚至是正在运作的系统，你不能保证新的系统能够实现全部的原有功能。更何况，你不是生活在真空，还有更多的投资、交付、竞争压力。

于是你使用一种 `quick-and-dirty` 的方法，如果系统有问题，那么就直接找到这个问题，便当地修改它。如果要增加一个新功能，你会从原来的系统中找到一块相近的代码，拷出来，作些修改。对于原来的系统，你想，既然我不能重头写过，而且它们已经在运作，让它去吧。然后，你增加的代码变成了下一个程序员咒骂的对象。系统越来越难以理解，维护越来越困难、越来越昂贵。系统变成了一个十足的大泥球。

这种情况是每一个人都不愿意碰到的，但是奇怪的是，这样的情景一次又一次出现在大多数人的编程生涯中。这是因为我们不知道该如何解决。

解决这个问题的最好办法当然是让它不要发生。然而，要阻止代码的腐化，你需要付出额外的代价。每次在修改或增加代码之前，你都要看一看手上的这些代码。如果它有很好的味道，那么你应该能够很方便地加入新的功能。如果你需要花很长的时间去理解原来的代码，花更长的时间去增加和修改代码。那么，先放下手里的活，让我们来做 Refactoring。

什么是 Refactoring?

每个人似乎都有自己的 Refactoring 的定义，尽管他们讲的就是同一件事情。在那么多的定义中，最先对 Refactoring 进行理论研究的 Raloh Johnson 的话显然更有说服力：

Refactoring is the process of taking an object design and rearranging it in various ways to make the design more flexible and/or reusable. There are several reasons you might want to do this, efficiency and maintainability being probaly the most important.

Martin Fowler[Fowler]把 Refactoring 定义为两部分，一部分为名词形式：

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.

另一部分则是动词形式:

Refactor (verb): to restruct software by applying a serials of refactoring without changing its observable behaviour.

Martin Fowler 的名词形式就是说 Refactoring 是对软件内部结构的改变, 这种改变的前提是不能改变程序的可观察的行为, 这种改变的也就是为了让他更容易理解, 更容易被修改。动词形式则突出 Refactor 是一种软件重构行为, 这种重构的方法就是应用一系列的 refactoring。

软件结构可以因为各种各样的原因而被改变, 如进行打印美化、性能优化等等, 但只有出于可理解性、可修改、可维护目的的改变才是 Refactoring。这种改变必须保持可观察的行为, 按照 Martin 的话来说, 就是 Refactoring 之前软件实现什么功能, 之后照样实现什么功能。任何用户, 不管是终端用户还是其他的程序员, 都不需要知道某些东西发生了变化。

Refactoring 原则

Two Hats (两顶帽子)

Kent Beck 提出这个比方。他说, 如果你在使用 Refactoring 开发软件, 你把开发时间分给两个不同的活动: 增加功能和 refactoring。增加功能时, 你不应该改变任何已经存在的代码, 你只是在增加新功能。这个时候, 你增加新的测试, 然后让这些新测试能够通过。当你换一顶帽子 refactoring 时, 你要记住你不应该增加任何新功能, 你只是在重构代码。你不会增加新的测试 (除非发现以前漏掉了一个)。只有当你的 Refactoring 改变了一个原先代码的接口时才改变某些测试。

在一个软件的开发过程中, 你可能频繁地交换这两顶帽子。你开始增加一个新功能, 这时你认识到, 如果原来的代码结构更好一点, 新功能就能够更方便地加入。因此, 你脱下增加功能的帽子, 换上 refactoring 的帽子。一会儿, 代码结构变好了, 你脱下 refactoring 的帽子, 戴上增加功能的帽子。增加了新功能以后, 你可能发现你的代码使得程序的结构难以理解, 这时你又交换帽子。

关于两顶帽子交换的故事不断地发生在你的日常开发中, 但是不管你带着哪一定帽子, 一定要记住带一定帽子只做一件事情。

Unit Test

保持代码的可观察行为不变称为 Refactoring 的安全性。Refactoring 工具用半形式化的理论证明来保证 Refactoring 的安全性。

但是, 要从理论上完全证明系统的可观察行为保持不变, 虽然不是说不可能, 也是十分困难的。工具也有自己的缺陷。首先, 目前对于 Refactoring 的理论研究并非十分成熟, 某些曾经被证明安全的 Refactoring 最近被发现在特定的场合下并不安全。其次, 目前的工具不能很好地支持“非正式”的 Refactoring 操作, 如果你发现一种新的 Refactoring 技巧, 工

具不能立即让这种 refactoring 为你所用。

自动化的测试是检验 Refactoring 安全性非常方便而且有效的方法。虽然我们不能穷尽整个系统中所有的测试，但如果在 Refactoring 之前成功的测试现在失败了，我们就会知道刚刚做的 Refactoring 破坏了系统的可观察行为。自动化测试能够在程序员不进行人工干预的情况下自动检测到这样的行为破坏。

自动化测试中最实用的工具是 XUnit 系列单元测试框架，该框架最初由 Kent Beck 和 Eric Gamma 为 Smalltalk 社团而开发。

Eric Gamma 对测试的重要性曾经有过这样的话：

The few tests you write, the less productive you are and the less stable your code becomes. The less productive and accurate you are, the more pressure you feel.... You would see the value of the immediate feedback you get from writing and saving and rerunning your own unit tests.

下面的片断来自 Javaworld, 两个 Sun 开发者展示了他们如何扩展单元测试来检查象 EJB 这样的分布式控件：

We can never over test software, but we seldom test it enough... Unit testing is a critical, but often misunderstood, part of the software development process. Unit testing involves individually testing each small unit of code to ensure that it works on its own, independent of the other units. In object-oriented languages, a unit often, but not always, equivocates to a class. If developers know for certain that each piece of the application works as it was designed to do, they would realize that problems with the assembled application must result from the way the components were put together. Unit testing tells developers that an application 'pieces are working as designed.

我曾经认为自己是很好的程序员。认为自己的代码几乎不可能出错。但事实上，我没有任何证据可以证明这一点，同样我也没有信心我的代码就一定会不会出错，或者当我增加一项新功能时，原先的行为一定没有遭到破坏。另一方面，我认为太多的测试于事无补，测试只能停留在理论之上，或只有那些实力强劲的大公司才能做到。

这个观点在 1999 年我看到 Kent Beck 和 Gamma 的 JUnit 测试框架之后被完全推翻了。JUnit 是 XP 的重要工具之一。XP 提倡一个规则叫做 test-first design。采用 Test First Design 方法，你在编写一个新功能前先写一个单元测试，用它来测试实现新功能需要但可能会出错的代码。这意味着，测试首先是失败的，写代码的目的就是为了让这些测试能够成功运行。

JUnit 的简单、易用和强大的功能几乎让我立刻接纳了单元测试的思想，不但因为它可以让我有证据表明我的代码是正确的，更重要的是在我每次对代码进行修改的同时，我有信心所有的变化都不会影响原有的功能。测试已经成为我所有代码的一部分。关于这一点，Kent Beck 在它的《Extreme Programming Explained》中指出：

Any program feature without an automated test simply doesn't exist. Programmers write unit tests so that their confidence in the operation of the program can become part of the program itself. Customer write functional test so that their confidence in the operation of the program can become part of the program, too. The result is a program that becomes more and more confidence over time – it becomes more capable of accepting change, not less.

单元测试的基本过程如下：

1. 设计一个应当失败的测试
2. 编译器应当立刻反映出失败。因为测试中需要使用的类和方法还没有实现。
3. 如果有编译错误，完成代码，只要让编译通过即可，这时的代码只反映了代码的意图而非实现。

4. 在 JUnit 中运行所有的测试，它应当指示测试失败
5. 编写实际代码，目的是为了测试能够成功。
6. 在 JUnit 中运行所有的测试，保证所有的测试全部通过，一旦所有的测试通过，停止编码。
7. 考虑一下是否有其他情况没有考虑到，编写测试，运行它，必要时修改代码，直至测试通过

在编写测试的时候，要注意对测试的内容加以考虑，并不是测试越多越好。Kent Beck 说：

You don't have to write a test for every single method you write, only production methods that could possibly break. Sometimes you just want to find out if something is possible. You go explore for half an hour. Yes, it is possible. Now you throw away your code and start over with tests.

另一位作者 Eric Gamma 说：

You can always write more tests. However, you will quickly find that a fraction of the tests you can imagine are actually useful. What you want is to write tests that fail even though you think they should work, or tests that succeed even though you think they would fail. Another way to think of it is in cost/benefit terms. You want to write tests that will pay you back with information.

你可能会认为单元测试虽然好，但是它会增加你的编程负担，而别人花钱是请你来写代码，而不是来写测试的。但是 WILLAM WAKE 说：

Unit Tests can be tedious to write, but they save you time in the future (by catching bugs after changes). Less obviously, but just as important. Is that they can save you time now: tests focus your design and implementation on simplicity, they support refactoring, and they validate feature as you develop.

你还会认为单元测试可能增加你的维护量，因为如果代码发生了改变，相应的测试也需要做出改变。事实上，测试只会让你的维护更快，因为它们让你对你所做出的改变更有信心，如果你做错了一件事，测试同时也会提醒你。如果接口发生了改变，你当然需要改变你的接口，但这一点并非太难。

单元测试是程序的一部分，而不是独立的测试部门所应完成的任务。这就是所谓的自测试代码。程序员可能花费一些时间在编写代码，花费一些时间在理解别人的代码，花费一些时间在做设计，但他们最多的时间是在做调试。任何一个人都有这样一种遭遇，一个小小的问题可能花费你一个下午、一天，甚至是几天的时间来调试。要改正一个 bug 往往很简单，但是要找到这样的 bug 却是一个大问题。如果你的代码能够带有自动化的自测试，那么一旦你加入一个新的功能，旧的测试会告诉你那些原来的代码存在着 bug，而新加入的测试则告诉哪些新加入的代码引入了 bug。

Small step

Refactoring 的另一个原则就是每一步总是做很少的工作，每做少量修改，就进行测试，保证 refactoring 的程序是安全的。

如果你一次做了太多的修改，那么就有可能介入很多的 bug，代码将难以调试。如果你发现修改并不正确，要想返回到原来的状态也十分困难。

这些细小的步骤包括：

- 寻找需要 refactoring 的地方。这些地方可能在理解代码、扩展系统的时候被发现。

或者是通过闻代码的味道而找到。或者通过某些代码分析工具。

- 如果对需要 Refactoring 的代码还没有单元测试，首先编写单元测试。如果已经有了单元测试，看一看该单元测试是否考虑到了你所面对的问题，不然就完善它。
- 运行单元测试，保证原先的代码是正确的。
- 根据代码所呈现的现象在头脑中反映该做什么样的 refactoring，或者找一本关于 Refactoring 分类目录的书放在面前，查找相似的情况，然后按照书中的指示一步一步进行 Refactoring。
- 每一步完成时，都进行单元测试，保证它的安全性，也就是可观察行为没有发生改变。
- 如果 Refactoring 改变了接口，你需要修改测试套件
- 全部做完后，进行全部的单元测试，功能测试，保证整个系统的可观察行为不受影响。

如果你按照这样的步骤去做 Refactoring，那么可能出错的机会就很小，正如 Kent Beck 所说：

“I’m not a great programmer; I’m just a good programmer with great habits”.

要求使用小步骤渐进地 Refactoring 并不完全出于对实践易行的考虑。

Ralph Johnson 在伊利诺斯州立大学领导的一个研究小组是 Refactoring 理论的引导者和最重要的理论研究团体。其中 William Opdyke 1992 年的博士论文《Refactoring Object-Oriented Framework》是公认的 Refactoring 第一位正式提出者。在那篇论文中，Opdyk 描述他对 refactoring 重构层次的看法：

People usually think about software changes either at a high level, in term of features to be added to a system, or at a low level, in terms of lines of code to be changed. Refactorings are reorganization plans that support change at an intermediate level. Consider , for example, the refactoring that moves a member function from one class to another...

为了实现这样的 intermediate level 操作，Opdyke 提出了原子 atomic refactoring 的概念，他指出：

All but the final category of supporting refactoring listed below are atomic; that is , they are the most primitive refactorings. The atomic refactoring create, delete change and move entities...

... the high-level refactoring are supported by a set of twenty six low-level(atomic) refactoring.

论文中，Opdyke 首先证明在一定的前提之下，这些原子 refactoring 将不会改变程序的 Observable behaviour。更高层的 refactoring 可以通过分解为这些原子的 refactoring 加以证明。Opdyke 也证明了他所提出的高层 refactoring 如何在每一步原子 atomic 之后都符合后续原子 atomic 所需要的前提。

小步前进使得对每一步进行证明成为可能，最终通过组合这些证明，可以从更高层次上来证明这些 refactoring 的安全性和正确性。

Refactoring 工具依赖于这些理论研究进行 Refactoring。如果每个人能够按照这样的一小步一小步进行 Refactoring，那么极有希望他的 refactoring 能够被正确地记录下来，为整个面向对象社团所用。同时，对他理论正确性地证明可以促使 refactoring 工具得到进一步的发展。

也许你会认为，随着工具的发展，程序员将变成 Refactoring 机器人。这样的看法是不正确的。

虽然使用一个 refactoring 工具能够避免介入使用手工方式可能产生的各种各样 bug，减少编译、测试和 code review。但是正如 Smalltalk Refactory Browser 的作者 Don Roberts 所说，Refactoring 工具不打算用来代替程序员，程序员需要自己来决定什么地方需要 refactoring，

做什么样的 refactoring。而在这一点上，经验是不可代替的。

Code Review 和 Pair Programming

要保证 refactoring 的正确性，还有一种很有用的方法就是进行 Code Review。

Code Review 原先一般都在一些大公司实行，他们可能聘请专家对项目进行 Code Review，以发现代码中存在的问题，改良系统的设计，提高程序员的水平。

同样在 refactoring 过程中，我们也可以使用 Code Review 的方法。问题是，我们是否有足够的精力和人员配备来进行这样的 Review 呢？

XP 成功经验表明，Code Review 不应当是只有大公司才能做的。甚之，XP 中的 Pair Programming 其实就是对 Code Review 的极端化，它也更加适合于表达 Code review 在 refactoring 过程中所能起到的作用。Kent Beck 说：

There are two roles in each pair. One partner, the one with the keyboard and the mouse, is thinking about the best way to implements this method right here. The other partner is thinking more strategically:

- Is this whole approach going to work?
- What are some other test cases that might not work yet?
- Is there some way to simply the whole system so the current problem just disappears?

使用这种方法进行 refactoring，可以在一个程序员没有想到一个应当有的单元测试时，当一个程序员无法找到合适的 Refactoring 方法或者当一个程序员没有按照正确的方法进行 refactoring 时，另外一个程序员可以提出自己的观点和建议。甚至在极端情况下，当拥有键盘的程序员对如何完成这个 refactoring 没有概念时，另外一个程序员可以接过键盘，直接往下做。

The Rule of Three

Don Roberts 提出的 The Rule of Three 好像和 Pattern 社团对模式的验证十分相似：

The first time you do something, you just do it. The second time you do something, you wince at the duplication, but you do the duplicate thing anyway. The Third time you do something similar, you refactor.

好处

简化测试

一个好的 Refactoring 实现能够减少对新设计的测试量.因为 Refactoring 的每一步都保持可观察的行为,也就是保持系统的所有单元测试都能顺利通过。所以只有发生改变的代码需要测试.这种增量测试使得所有的后续测试都建立在坚实的基础之上，整个系统测试的复杂性大

大降低。

更简单的设计

Refactoring 降低初始设计的复杂程度。Gamma 指出复杂设计模式的一个陷阱是过度狂热：“模式有其成本(间接性、复杂化)，因此设计应该达到需求所要求的灵活性，而不是越灵活越好”。如果设计试图介入太多以后可能需要的灵活性，就会产生不必要的复杂和错误。Refactoring 能够以多种方式扩展设计。他鼓励为手头的任务建立刚刚合适的解决方案，当新的需求来到时，可以通过 Refactoring 扩展设计。

Refactoring makes software easier to understand

程序的最终目的是为了指引计算机完成人们需要完成的事情。但是，要完成这个目标并非想象的那么容易。

程序编写是人的活动，人首先要理解才能行动。所以，源代码的另一个作用就是用于交流的工具。其他人可能会在几个月之后修改你的代码，如果连理解你的代码都做不到，他又如何完成所需的修改呢？我们通常会忘掉源代码的这种用处，尽管它可能是源代码更重要的用处。不然，我们为什么发展高级语言、面向对象语言，为什么我们不直接使用汇编语言甚至是机器语言来编写程序？难道我们真的在意计算机多花了几个 CPU 周期去完成一件事？

如果一个人能够理解我们的代码，他可能只需要一天的时间完成一个增加功能的任务，而如果他不理解我们的代码，可能需要花上一个礼拜或更长的时间。这里的问题是，我们在编写代码的时候不但需要考虑计算机 CPU 的想法，更要把以后的开发者放在心上。除非，你写代码的唯一目的就是把它丢掉。你不想让任何别的开发人员用到这段代码，包括你自己。因为你不可能记得所有你写过的代码，如果你经常回过头去看一下自己的代码，你就会体会到代码的可理解是如何重要。

Refactoring 可以使得你的代码更理解，Refactoring 支持更小的类、更短的方法、更少的局部变量、更小的系统耦合，Refactoring 要求你更加小心自己的命名机制，让名字反映出你的意图。如果哪一块代码太复杂以至于哪一理解，你都需要对他进行 Refactor。

你可能认为，反映代码的意图应当是注释和文档的责任。假设你写好了一段程序，给他加上注释，来说明你这段代码完成了什么。每次代码发生改变，你都需要修改注释。而其实代码本身应当足以说明这个问题。如果代码不能反映自己的意图，即使是再多的注释也不足以让你理解代码的所有机制，除非你把代码的每一句话都加上注释。甚之，这种重复的责任使得一旦注释和代码发生不一致，它反而会阻碍你对代码的理解。而任何一个程序员都不愿意写太多的注释和文档。所以 Martin fowler 说：

When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.

Martin Fowler 同时指出，Refactoring 不但能够增加别人对你代码的理解，而且是一种非常好的理解别人代码，学习别人代码的方法。通常，当你拿到一大堆代码时，你可能会觉得一片茫然，不知道从何处开始。学习别人代码最好的方法就是对代码进行进行 Refactor。如果你发现自己不能理解一段代码，那么试图使用对自己说，哦，这段代码可能是在做什么事情。给他一个有意义的名字，refactor 它。Refactor 使得你对代码的理解不是仅仅停留在脑袋中，而是看到代码确实按照你的理解再发生变化。如果你的 refactor 改变了系统的行为，那么说明你的理解还有问题。你必须返回重来。

Refactoring improves the design of software

许多人把编程看作是一种低级活动。他们认为代码仅仅是设计的附属物。然而正是代码，而不是你脑袋里或纸上的设计，真正驱动计算机完成你想做的事情。而绝大多数的 bug 也产生于编码阶段。

很多方法学认为通过分析和设计的严格化就能产生更高质量的软件，这实际上是不可能的。正如 Brain Foote 和 Joseph Yoder 在《Big Ball of MUD》一文中指出，虽然许多作者提出了很多理想上非常完美的体系结构，如 Layer、PIPELINE 等等。但在实践中，几乎从来都不能看到这么结构清晰的系统。

这一方面是由于分析和设计一个领域的应用程序需要对该领域丰富的知识，而这种知识不是在一开始都能获得的。我们通常需要在实践反馈的过程中才能一步步加深自己对该领域的理解。因而，我们一开始的设计可能并不能正确反映系统的内在本质，所以也不可能在代码中得到很好的反映。

另一方面，即使一开始的设计是完好的，随着用户对系统使用的深入，新的需求可能会被加入，旧的需求会被修改、删除。一个最先的设计不可能完全预料到这些变化。

一旦实现开始偏离最初的设计，那么它的代码将不受控制，从而不可避免地开始腐蚀。代码加入越多，腐蚀的速度越快。如果没有办法让设计尽可能地与实现保持一致，那么这种腐蚀的最后结果就是代码不得被抛弃。

但是让代码实现和设计始终保持一致并非那么简单，在传统的软件方法中，一旦开发到了实现阶段，就很难对设计做出变化。所以最近发展的面向对象方法学把增量迭代(iterative)作为一个基本原则。

也许是题外话，在目前最为时行的一些“重型”软件方法学中，我们很难找到对迭代(包括设计的迭代)明确的支持、定义和操作过程，以及如何提高程序员适应这种动态开发能力的方法。在我看来，重型的软件方法学即使能够对软件开发起到一定的作用，他也不可能包罗万象，解决软件开发中的所有问题。开发方法学重视生命周期管理和控制，但软件的开发并不只有过程和生命周期，更重要的是，同时往往被“正规”的软件方法所忽略的是，软件开发是人的活动。不管你的过程控制是多么的严格，不管你的生命周期模型是多么的完美，如果不能提高人的生产力，提高产品的质量，那么一切都是毫无意义的。

在这里，要实现这样一种增量迭代的开发模型，你必须能够让分析人员、设计人员、程序人员、测试人员等等，所有参与开发活动的人有能力或者能够有切实可行的方法来实现迭代，实现增量。如果不是这样的话，你无法随时保持实现和设计之间的一致性，无法把编码实现中所发现的不合理设计反馈到初始设计，也无法在需求变化时对实现所产生的影响准确及时地得到反映。

如果没有切实可行的基本方法来支持迭代中所需要的改变，那么迭代将是非常困难的。我们可以设想，一旦新的需求到达，新一轮迭代开始，而原先的系统设计被证明无法适应现在的变化，这个时候你如何能够使得迭代顺利地以增量的方式进行下去？

Refactoring 提供了对 incremental iterative development 最好的支持。随着系统的演变，代码的结构越来越差，通常这意味着要完成一件事情需要更多的代码。很多情况，通常缘于在几个不同的地方都有类似的代码。因此 Refactoring 最重要的课题之一就是排除重复。我们通过排除重复力求能够达到 Once And Only Once 的境界。而它正是好设计的一个基本标准。Once and Only Once 虽然不能直接提高软件的效率，但却让代码的修改和理解变得更为容易、系统的结构更加清晰。它是消除代码腐蚀的最佳途径。

Refactoring help you find Bugs

帮助理解代码同时也可以帮助你发现 bug。很多代码处于系统中一些非常微妙的地方。如果你只是一遍又一遍地阅读代码，恐怕你永远也找不到 bug。有的时候，你使用 IDE 提供的工具也难以跟踪这样的 bug。因为如果你的代码没有很好的结构，譬如一个类具有太多的责任，那么它必然具有太多的状态，在调试程序的时候，你同样需要注意太多的现象。你可以用 refactoring 使得程序代码的结构更清晰，每一个时刻可以更集中关注专一的数据和行为，这会使你的工作量大大减少。同时，由于 refactoring 要求 small steps，并且对每一步都进行严格的测试，这样会使得 bug 很容易地浮现出来。

Refactoring Helps you program faster

在了解 refactoring 后，你可能承认 refactoring 能够改进软件的质量、设计、结构、可读性，减少 bug，但是你会认为运用 refactoring 技术进行软件开发有可能减慢你的开发速度。毕竟，你要编写额外的单元测试，你要去 refactor 的代码本来就能够工作的代码。

那么，为什么要如此麻烦？他不会拖我的进度吗？

事实上，Refactoring 确实能够加快软件的开发速度。一个好设计的基本前提就是允许更快的软件开发。说得更绝对一点，好设计的全部就是它是否能够更快、更灵活地支持软件的变化。如果没有一个好的设计，一开始你可能可以开发得很快，但是随着功能的增加，你的代码逐渐腐蚀，结构渐渐失去。每次当你需要加入新功能时，你必须花大量的时间去理解原来的代码，修改原来代码的 bug。改变一项功能需要花费越来越长的时间。

Refactoring 支持好的结构、设计和理解性，它让你更快地开发软件。因为它可以防止软件的侵蚀，他甚至用于改进设计。

应用 Refactoring 需要考虑的问题

任何一种技术都可能有它自己的麻烦。但是往往在我们使用一种新技术的时候，可能还不能深入到发现它带来的问题，正如 Martin Fowler 所说：

When you learn a new technique that greatly improves your productivity, it is hard to see when it does not apply.

他把 Refactoring 的情景和面向对象出现使得情景相比较：

Ten years ago it was like that with objects. It wasn't that I didn't think objects had limitations- I'm too cynical for that. It was just that I didn't know what those limitations were, although I know what the benefits were.

但是 Martin Fowler 和其他人确实观察到了 Refactoring 可能引发的某些问题，我们可以来看一下：

数据库

很多应用程序的代码可能与数据库结构绑定得非常严密。如果要修改这些代码，需要改变还有数据库结构和原先已经存在的数据。

O/R mapping 可以用来解决这个问题。使用专业的 O/R mapping 工具能够实现关系数据库的迁移。但是，就算这样，迁移也需要付出额外的代价。

如果你使用的并非关系数据库，而是直接采用 OO 数据库，这一点的影响可能会变得更小。

所以，我建议每一个使用数据库的应用程序都应该采用 O/R mapping 或者 OO 数据库。目前出现的各种企业级应用解决方案如 J2EE 本身就提供这样的构架。

如果你的代码没有这样一个隔离层，那么你必须手工或编写专用的代码来实现这些迁移功能。

接口改变和 **Published Interface**

有很多 Refactoring 操作（如 rename method name）确实改变了接口。面向对象承诺在接口不变的情况下给你以实现变化的自由。但如果接口发生改变，那么你就不得不非常小心了。

为了保证系统的可观察行为不变，你必须保证这些接口的改变不会影响到你无法取得的代码。如果你拥有了所有使用该接口的类的源代码，你只要把这些地方同时也改变即可。

但是，如果你没有办法得到所有这些使用的代码，那么你就不得不采取额外的途径。事实上，如果你的代码是一个代码库（如 Sun JDK 的集合框架）或者是一个 Framework，那么这一点几乎是不可避免的。

要使得这些依赖于你老接口的代码能够继续工作，你必须保留老接口。现在你有两套接口，一套是老的，一套是经过 Refactoring 的新接口。你必须把对老接口的调用分派到新接口。千万不要拷贝整个函数体，因为这会产生大量的重复代码。

这种方法虽然能够解决问题，但是却非常麻烦。由于 Refactoring 通常会涉及到状态、行为在不同类之间的转移，如果一个方法从一个类移动到另一个类，那么使用这种分派的方法可能需要一些不必要的中间状态或者参数。这会使你的代码显得难以理解和维护，在一定程度上削减了 Refactoring 所应起到的作用。

因此，这种方法只应该用于过渡时期。给用户一定的时间，允许用户代码能够逐渐转移到新接口，在超过一定的期限后，删除老方法，不再支持老接口。这也是 Java Deprecated API 的意义所在。

像这样保护接口虽然可能，却非常困难。你至少需要在一段时间内维护两套接口，以保证原来使用你老接口的客户代码还能继续使用你的新代码，Martin Fowler 把这些接口称之为 **Published Interface**。虽然你不可能避免公布你的一部分接口，不然谁也不能使用你的代码，但是过早公布不必要的接口会造成不必要的麻烦，就像 Martin Fowler 给我们的提示：

Don't publish interface prematurely.

用 Refactoring 思想武装自己的设计

如果你不理解 OO 的思想,那么你就不能真正地用好 OO 语言。同样,如果你没有把 Refactoring 的思想贯穿于你的开发过程,你也不可能用好 Refactoring。

Refactoring 包含两个方面的想法:它告诉你可以从简单的设计做起,因为即使代码已经实现,你还是可以用它来改进你的设计。然而,另一方面,它绝不是告诉你可以信手涂鸦。我给你的忠告是:

Started simple but not stupid.

如果你一开始就设计了愚蠢的接口,甚至是错误的接口。在程序演变的过程中,这一部分可能变成系统的核心。对之进行 Refactoring 可能需要花费大量的精力,而改变接口和类的操作可能会是这些 Refactoring 主要内容。对核心类接口的变化可能会迅速波及到系统的各个层面,如果你的总体结构是好的,那么这种涟漪可能会在某一个层次消失。(譬如环状和层次性的体系结构。)如果你没有这样的抽象机制和保护体系,那么对核心类的修改将会直接导致整个系统的变更,这是不能接受的。

所以,在设计一个类的时候,你需要问自己几个问题,如果事情发生了这种变化,我会如何修改来适应?如果发生了那种变化,我会怎样来适应?如果你能够想到可能的 Refactoring 方法,那么证明你的设计是可行的。这并不意味着你要去实现这样的设计,而是保证自己的设计不会把自己逼入到死角。如果你发现自己的代码几乎没有办法 Refactoring 来适应新的需求,那么你要仔细考虑考虑别的思路。

每次公司的程序员问我一个设计是否合理,我总是反问几个问题:你如何适应这种变化,适应那种可能的变化。我同时指出现在没有必要去实现这些变化。我很少直接回答他好坏或者给他一个答案,但在思考了我反问他们的问题以后,程序员总能对自己的设计做出好的评判,从而找到很好的解决方案。所以,使用 Refactoring 的思想考虑你的设计。

编程语言

虽然 Refactoring 是一种独立于编程语言的方法,但你所使用的编程语言往往会或多或少地影响到 Refactoring 的效率,从而影响你采用 Refactoring 的积极性。

Refactoring 最初的研究是从 Smalltalk 开始的。随着 Refactoring 在 Smalltalk 上的极端成功,更多的面向对象社团开始把 Refactoring 扩展到其他语言环境。但是不同语言的不同特点有时会对应用 Refactoring 提供便利,有时却会制造障碍。

支持 Refactoring 的语言特点和编程风格

静态类型检查和存取保护

静态类型检查可以缩小对你想要 refactoring 的程序部分的可能引用范围。举个例子,如果你想要改变一个类的成员函数名,那么你必须改变函数的声明和所有对该函数的引用。如果程序很大,那么查找这样和改变这样的引用就比较困难。

和 Smalltalk 这样的动态类型语言不同,对静态类型进行检查的语言(C++,Java,Delphi 等等)通常具有类继承和相关的存取保护(private,protected,public),这些特点使得寻找对某一个函数的引用变得相对简单。如果重命名的函数原先声明为 private,那么对该函数的引用只能是在他所在的类或者该类的友类(C++)等等。如果声明为 protected,那么只有本类,子类和友员类

(同包类)才能引用到该成员函数.如果声明为 `public`, 那么还只需要在本类、子类、友类和明确引入该类的其他类即可 (`include`, `import`)。

我想提起大家注意的另外一个问题.在软件的最初开发和整个开发流程中尽可能早地应用好的设计原则是一个软件项目成功的重要因素。不管是从封装的角度还是从 `Refactoring` 的角度来看, 定义成员变量和成员函数应当从最高的保护级别开始。除了非常明显的例子之外, 你最好首先把成员变量和函数定义为 `private`。随着软件开发的进一步深入, 当其他类对该类提出“额外”的请求, 你慢慢地放宽保护。原则是: 如果能够放在 `private`, 就不要放在 `protected`, 能够放在 `protected`, 就不要放在 `public`。

使 `Refactoring` 复杂化的语言特点和编程风格

预处理指令

某些语言环境通常提供预处理指令, 如 `C++`。因为预处理不是 `C++` 语言的一部分, 这通常使得 `Refactoring` 工具实现变得困难。有研究指出, 程序往往需要在预处理之后才能进行更好的结构分析, 而在这一点上预处理指令信息已经不存在。而 `refactoring` 一旦没有和源代码的直接联系, 程序员将不太可能对理解 `Refactoring` 的结果。

依赖对象尺寸和实现格式的代码

`C++` 继承自 `C`, 这使得 `C++` 很快流行起来, 程序员的学习难度也大大减小。但这是一把双面刃。`C++` 因此而支持很多编程风格, 而其中的某些违反了优雅设计的基本原则。

使用 `C++` 的指针、`cast` 操作和 `sizeof (Object)` 这些依赖对象尺寸和实现格式的代码很难 `refactor`。指针和 `cast` 介入别名的概念, 这使得你要查找所有对此 `Object` 有引用的代码变得非常困难。这些特征的一个共同特点就是它们暴露了对象的内部表达格式, 从而违反了抽象的基本原则。

举个例子, `C++` 使用 `V-table` 机制来表达可执行程序中的成员变量。继承得来的成员变量在前, 本类定义的在后。一个我们经常使用, 并且认为安全的 `refactoring` 是 `push up fields`, 也就是把子类中的一个成员变量移到父类。因为现在变量从父类继承而非本类定义, 经过 `refactoring` 后的可执行程序之中变量的实际位置已经发生了变化。

如果程序中所有的变量引用都是通过类接口来存取的, 那么这样的变化不会有问题。但是, 如果变量是通过指针运算 (譬如, 一个程序员有一个指向对象的指针, 知道变量在类的第 9 个字节, 然后使用指针运算给第 9 个字节赋值), 上面的 `refacoting` 过程就会改变程序的行为。类似情况, 如果程序员使用 `if (sizeof (object) == 15)` 这样的条件判断, `refactoring` 的结果很可能会对该对象的大小产生影响, 从而变得不再安全。

语言复杂度

语言越复杂, 对语言语义的形式化就更加困难。相对 `Smalltalk` 和稍微复杂的 `Java` 而言, `C++` 可称得上是一种非常复杂的语言, 这使得对 `C++` 程序 `refactoring` 工具的研究大大滞后于 `smalltalk` 和 `Java`。

解析引用的方式

由于 `C++` 绝大部分是在编译是解析引用, 所以在 `refactoring` 一个程序之后通常至少需要编译程序的一部分, 把可执行程序连接起来才能看到测试 `refactoring` 的影响。相反, `smalltalk` 和

CLOS 提供解释执行和增量编译的技术。Java 虽然没有解释执行，但它明确把一个公共类放在一个单元内的要求，使得执行一系列 refactoring 的成本减小。由于 refactoring 的基本方法就是每一步小小变化，每一步测试，对于 C++ 而言，每一个迭代的成本相对较高，从而程序员变得不太愿意做这些小变化。

反射、Meta 级程序分析和变更

这一点可能更让研究者关心而不是实践者的问题。C++ 并没有提供对 meta 级程序分析和变更的很好支持，你无法找到象 CLOS 这样的 metaobject 协议。这些协议有时对 refactoring 非常有用，譬如我们可以把一个类的选定实例改变为另一个类的实例，这时候可以利用这些反射协议实现把所有对旧对象的引用自动变更为指向新的实例。

Java 虽然还没有像 CLOS 这样强大的 meta 级功能，但是 JDK 的发展已经显示了 Java 在这方面非常强劲的实力。象上面的例子，我们也可以在 Java 上做到。

一个小结

基于上面的比较，我们认为 Java 是应用 Refactoring 的最佳语言。最近的观察也证实了这一点[Lance Tokuda]。

从实践者的角度来看，目前最流行的 refactoring 文献基本上都采用 Java 语言作为范例，其中包括 Martin 的《Refactoring》。目前市场上有数种支持 Java 和 Smalltalk 的 Refactoring 工具，而 C++ 的工具却几乎没有。这里面，语言本身的复杂性有很大的影响。

当然，这并不意味着 C++ 程序员就不应该使用 refactoring 技术，只不过需要更多的努力。Refactoring 技术已经证明自己是 OO 系统演化的最佳方法之一，不要放弃。

不应 Refactoring 的场合

程序原型

当你原型化一个系统时，你通常不在乎程序的灵活性和效率。原型的目的仅仅是为了证明一个概念、实现一种探索，加快加深与客户交流。一旦原型完成，这样的代码就应该丢掉，重新开始正确的编写。但是，一旦原型能够起到演示的作用，继续保留它并按照这样的方式一直实现下去的诱惑将很难抵挡。你会说：“程序运行的很好，就用这一个吧！”。从原型开始的目的来看，这样的策略最后是不会成功的。所以 Foote 甚至说：

One way to minimize the risk of a prototype being put into production is to write the prototype in using a language or tool that you couldn't possible use for a production version of your product.

由于原型从根本来说，不会有意考虑以后的变化和程序的结构，对原型做 Refactoring 是不可能也是毫无必要的。

Doesn't work

如果一个程序本身还不能工作，那么对他进行 Refactoring 就没有任何意义。Refactoring 保留程序的可观察行为，那么 Refactoring 的结果还是不能正常工作的。

这里并不排除使用 Refactoring 可以排除 bug，但那是在程序的绝大部分主要功能已经实现的情况下。

Close to deadline

如果你已经接近了最后提交日期，这个时候 Refactoring 可能产生的效率的提高将发生在底线之后。到那时候，一切都没有意义了。

但是这个问题可以有另外一种考虑的思路，为什么会出现这种情况？是因为估算严重失误还是因为效率太低？如果效率太低，你怎样来提高效率？Refactoring 是一种提高生产力极好的方法，因为它使设计更好，从而使得变化更快。所以，如果你发觉时间可能不够，往往就是需要 Refactoring 的一个信号

实施 Refactoring 可能碰到的阻碍以及解决方案

Refactoring 带来的长期利益是每个人都能够看到的，但是并不是每个人都愿意使用它，Opdyke 说：

Then, proponents of such techniques may be surprised (and discouraged) when world does not beat a path to their door.

接着，他指出即使已经意识到长期的利益，但人们还是不情愿使用这种方法的四个关键理由：

1. "I don't understand how to apply your approach."
2. "Is the benefits of your approach are only realized in the long-term, why exert the effort now? In the long term, I may no longer be with this project or with this organization"
3. "Your approach is an overhead activity; I'm paid to write NEW feature"
4. "Our existing implementation might change in unexpected way, or possibly break, when we apply your approach. Reliability and backward compatibility are import to us".

这些问题其实并不单单存在于 Refactoring 领域，在鼓励人们设计、编写更重用的系统时，我们也必须解决这些问题：

1. 技术人员可能不理解重用什么以及如何重用；
2. 如果不能看到短期效益，技术人员不会采用提倡重用的方法。
3. 我们必须明确支持重用的方法可能引发的额外学习曲线和其他成本。
4. 采用支持重用的方法不应该打断一个项目；新的实现应当和存在的系统向后兼容。

学习

要想知道如何 refactoring，你必须学习。

Refactoring 已经被有经验的 OO 程序员成功地使用了 10 多年。这些程序员也把他们的 Refactoring 的经验反馈给了 OO 社团。

本文旨在为你提供一个导引，在学习了这篇文章后，你可以沿着以下几个方向进行深入：

1. 要立刻开始动手进行 Refactoring，你的最佳选择就是 Martin Fowler 的《Refactoring: Improving the Design of Existing Code》，该书有一个 Refactoring 的分类目录。详细描述了 70 多种 Refactoring 方法。Martin Fowler 的主页 (<http://www.martinfowler.com>) 上还有最近扩展的一些 Refactoring 方法。
2. 你需要阅读各种好的程序代码和设计结构，特别是设计模式，因为设计模式可以称得上 Refactoring 的终极目标之一。
3. 你可以到网上搜索各种 Refactoring 工具，我将在后面介绍，学习使用 Refactoring 工具进行更安全的 Refactoring。
4. 如果你想学习 Refactoring 方面的理论，请参见我文后附上的各种参考文献，特别是 Ralph Johnson 教授的文章和他弟子们的博士论文。

另外，在我的主页 www.erptao.org 上，你可以：

1. 对本文提出评论和建议，以帮助我改进；
2. 在论坛中有 refactoring 方面的讨论，你可以在那里看到 refactoring 的一些实践，如果你有问题，可以在那里提出，我将尽量回答。

Refactoring 获得短期效益

如果没有短期效益，很难让技术人员直接看到 Refactoring 的效果。由于 Refactoring 开始于大学的研究机构和技术前沿的小公司，所以如何在工业主流软件中应用 Refactoring 成为 OO 社团实践的一个重要内容。

伊利诺斯大学研究小组的 CHOISE 操作系统框架是这些研究的一部分。其中，CHOICE 实现了对 System V, MS-DOS, BSD UNIX 等差别极大的不同文件系统的支持。在开发过程中应用 Refactoring 后，研究者指出 refactoring 确实能够获得不少短期和长期的利益。

对于近期而言，因为排除了重复代码，在通用代码测试中发现的错误只需要在一个地方进行修改。代码总量变小。与一个特定文件系统格式相关的代码与适合于两个或几个文件系统通用的代码隔离。

对于中期而言，来自于 refactoring 的抽象通常为以后其他的文件系统提供了方向。事实上，对两个现有文件系统的抽象不一定完全适合于第三个文件系统，但是已经存在的通用代码是一个有价值的起点。后续 refactoring 应用的结果将显示什么是文件系统真正的抽象。框架开发小组发现，随着实践的推移，加入一个文件系统所需要的努力越来越少。即使后来的文件系统更复杂，开发者更加缺乏经验。

削减 Refactoring 的额外开销

对于 Refactoring 所造成的额外开销，你可以这样来看待：

1. 工具和技术的发展使得 refactoring 更加自动化，能够减少大量的手工劳动和错误介入；
2. 有经验的 OO 程序员报告，refactoring 所造成的额外开销完全能够抵消采用 refactoring 对程序开发其他阶段的提升。
3. 刚刚开始使用 refactoring，你可能会觉得有点别扭。但一旦成为一种习惯，你将不会觉得这是一种额外工作，你会把它当作一个好程序员开发过程自然而然的一部分。你可以在 internet 上找到很多这样的报告，后面很多内容也都谈到了这个问题。

安全 Refactory

保持可观察的行为被称为 Refactoring 的安全性。要实现安全的 Refactoring，你通常有几种选择[William Opdyke]：

1. 相信你自己的编码能力；
2. 相信专家、文献提供的步骤能够帮助你减少错误。
3. 相信编译器能够捕获你没有发现的错误；
4. 相信你的测试套件能够捕获编译器没有发现的错误；
5. 相信 Code Review 会捕获你、你的编译器、你的测试套件漏掉的错误

各种各样的书籍、资料、论文、期刊可以增强你的编程能力，能够教给你更多的良好风格。Martin Fowler 的《Refactoring》就是这样一本书，他告诉我们 Refactoring 有非常细小的一步一步组成，每一步看起来都不起眼，但是一系列 Refactoring 的结果会对系统产生有力的影响。

编译器、好的风格、测试套件、Code Review 都是非常有价值的，但是所有这些方法方法都有他们的问题，编译器根本不知道程序的动态行为，好的风格、Code Review 完全依靠人来实现，而任何一个人都会犯错误。测试套件也没有办法覆盖所有的行为。

所以，面向软件社团对 Refactoring 的一个重要研究方向，就是定义 Refactoring 安全性方面的理论并实现这种理论支持下的工具。它们可以用来检查某一种 Refactoring 是否能够安全地应用到一个程序，如果安全的话，那么就有工具来完成 Refactoring。这样做也避免了手工完成可能引起的 bug。

Refactoring 和方法

增量迭代

自从有软件工程一说开始，大大小小出现了许多方法，其中一些还常常被我们挂在嘴边。这些软件方法中最著名的生命周期模型包括 WaterFall，它是由 Royce 于 1970 年首先提出来的。

一般来说,WaterFall 方法把软件开发的过程严格地分为几个阶段,包括:

Requirement phase、Specification phase、Design phase、Implementation phase、Integration phase, 要注意测试并非一个独立的阶段,几乎在每个阶段都需要检查和测试。

应当说, WaterFall 模型由它自己的很多优点。WaterFall 强制每一阶段都必须产生所有的产品,只有当这些产品通过相关的审核后才能开始下一阶段的工作,而这些产品中相当重要的一部分就是每一阶段的详细文档。Specification 文档、design 文档、code 文档和其他相关的文档,如数据库手册、用户指南等等是维护产品基本工具。有研究指出,大约 70% 的软件预算用于维护阶段。而 WaterFall 模型强制每一阶段必须有详细文档,所以,看起来 WaterFall 能使用这些文档大大削减这方面的开销。

但是, Waterfall 模型受文档驱使的方式也可能成为一个缺点。WaterFall 起始于需求阶段,一旦需求报告被用户承认,你就开始起草 Specification 文档,它指出了软件需要做什么东西。Specification 文档通常很长、很详细、很直白,因此很容易让人看着生厌。用户通常都很难完全看懂软件的 Specification 文档,他几乎不可能判断你讲得是否正确,但通常他还是会的签字的。

这里的问题是人和人之间的交流是否能够完全使用文档来实现,即使你能够使用更加高级的图形如 UML 图。对于这个问题,生物学家 Maturana 和 Varela 在生物系统中进行了研究,下面是他们在《The Tree of Life》一书中的答案:

。。。The phenomenon of communication does not depend on what is transmitted, but on what happens to the person who receives it. And this is a very different matter from ‘transmitting information’.

Alistair Cockburn 在他 2001 年的新书《Agile Software Development》解释说,你可以想象人被一层薄膜所包围,在接受到外部刺激,被重构翻译成内部信号,从而引发内部活动。所以人真正所接受的信息取决于内部而不是外部。更浅显的例子也有 cockburn 给出,他说:

To put this into a concrete example, consider that someone runs into the room and shouts “Fire!” in Japanese. A Japanese-speaking listener receive a lot of information, and immediately leaps out and runs to the exit. The Japanese person next to him, who happens to be asleep, receives no information at all. The external stimulus was never converted into an internal signal. A person who speaks no Japanese notices that someone came in and shouted something but received no particular information from the sounds uttered. What each person receives from the shout depends on her internal condition.

在对这一声大喊的翻译过程中,一个日本人能够接受到大量信息而非日本人却不知所云的重要原因是大喊的人和那个在房间里的日本人有一个共同的经验:日语。软件需求获取者、Specification 撰写者和用户之间往往不可能有这样的共同经验。

同样,如果光光依靠设计文档想使得设计者和开发人员之间能够达到很好的交流也是不可能的。文档有助于增进理解,但是它不可能达到完整的理解。Waterfall 试图依赖非常详细的文档来接近交流的完美。事实上,对文档的理解并不依赖于文档的详细程度。而是依赖于文档的编写者和文档阅读者之间的共同经验。

如果我和你之间都具有设计模式的背景,那么我说一句:“使用 visitor 模式“,可能比两个没有设计模式经验的人画上 100 个 UML 类图,写上 1000 句的详细文档更能说明问题。

这种共同经验的积累显然不可能一趋而就。当然,在一个软件公司内部,长期的合作可能使用设计者和开发者之间能够达到更高层的交流,譬如,在进行设计的时候,你说:“就像上次那个多语中问题一样处理“。那么程序员就根本不需要再多的设计信息也能够完全实现。

对于用户的需求而言,这种长期的合作关系是很少见的。因此,除非你对该用户的领域

非常熟悉，熟悉到你早已了解用户的一举一动。否则，你不可能一次就完成设计，你对用户的理解和用户对你的理解，也就是用户和你之间的交流需要多次的反复才能达到比较完善的程度。

交流的问题不仅仅出现在人和人之间，一个人需要和所实现的目标进行交流。这就是我们所说的理解。不管是在需求阶段、标书阶段、设计阶段或是实现阶段，问题的解决要依赖于你对这个问题的理解程度。理解越深刻，问题的解决就与完美。而这样的理解是不可能一次达到的。因此，对软件开发的任何一个阶段，你都需要进行迭代，从而能够更好地解决问题。

Waterfall 的另外一个问题就是用户在最终产品交付之前不能看到任何用户的東西。而用户对提交产品的看法最后既有可能是：“I know this is what I asked for, but it isn't really what I wanted.”

产生这种结果的原因是，因为用户通过纸上的文档理解到的东西远远不及真正能够使用的软件产品那么多。Waterfall 对需求和 Specification 如此的依赖，直接就可能产生并非用户真正需要的产品。

为了解决这个问题，我们需要使用增量模型。不同于 waterfall 和快速原型模型，增量模型并不试图一次性地发布完全符合用户需求的软件。相反，最终的产品被划分为几个 build，设计、开发、发布甚至是需求都按照一个一个 build 进行。任何一个阶段，用户都可以获得一个满足部分需求的实际可操作的软件。而不是象前面的模型一般，需要等待几个月、一年或几年才能等到一个它并不需要的产品。

增量模型同时能够适应用户的变化。变化是每一个成长中的用户组织固有的特点，改变的要求同时也是软件开发的一部分。这也是我们说“软件是软的”所包含的意义。从客户的财务观点来看，增量模型也使得他们不需要一次性花费大量的资金。如果在开发的过程中发现问题，用户也可以及时地测回资金。

增量模型同时也会出现问题。其中的一个难点就是每一个新发布的 build 必须能够平滑地集成到已经存在的系统中而不会破坏已经存在的東西。从这个角度来讲，一个增量模型下的设计可能需要比 Waterfall 更加复杂的设计。因为 Waterfall 一次性地考虑所有问题，它能够提前看到设计、实现中的所有问题（在他自己的范围之内），既然问题已经全部摆出，你就能够相对容易地解决问题。而增量模型则不同，在构建当前的 build 时，你不会去考虑以后的 build。所以，以后的 build 可能需要当前 build 不可预料的支持，它也可能以不可预料的方式集成到当前的系统之中。

面向对象的方法在这一点上助益甚多。面向对象的基本原则之一就是 Open-Closed 原则。一个模块可以在发布之后被 close，这样即使增加 build 也不会破坏原来的系统。同时，它通过各种面向对象特有的机制，多态和继承，使得系统能够以增量的方式平滑地集成。但是，要 close，你必须保证你的设计有足够的对以后的模块 open。而正象我们在最开始所说的一样，一次性的设计是不可能达到这样的目标。显然，你需要一种方法，能够在一个模块外部行为不变的情况下改变内部的结构，使得增量能够顺利地进行下去。Refactoring 是增量迭代模型的自然解决方案和核心技术。

Moving Target

增量迭代模型在处理 Moving Target 问题上更加显示出特有的优势。所谓的 Moving Target 就是可能出现随着开发的进展，软件的需求不断发生变化。

现今高速的信息流通和市场竞争压力是产生这种现象的最主要原因。一个最具竞争力的

企业就是最能快速相应变化的企业。适应变化是一个企业的核心竞争能力。而软件之所以能够在现今社会中占据如此重要的地位，就是它比其他人造物都更能适应变化。如果软件不能提供对变化最有力的支持，那么就没有什么其他的 artifact 能够做到了。

Moving Target 被认为是软件工程中最难以解决的问题之一。这里，随着软件构建的进程，用户的需求不断地发生变化。这不但会使开发小组产生受挫感，同时极易产生结构奇差的软件。同时这样的变化也会极大地增加软件开发的成本。

也许，这个问题可以通过构建一个快速原型来解决，这样的话，不管用户改变需求有多快，只要最后用户对 **specification** 满意了，产品马上就可以开始构建。但实际上，没有什么东西能够阻止客户在核准了 **specification** 以后再次改变需求。原型的好处只不过是能够让用户看到能工作的软件，从而减少变化的数量和频率。但是如果客户愿意花钱，它天天想改变需求都可以。

显然，过多的过程控制难以解决这样的问题。过程控制的复杂性会减低对需求变化的响应速度，那么，是不是象某些软件工程书所写的，我们对这样的状况无能为力呢？

拥抱变化—Refactoring 和 XP

随着 Kent Beck 的名著《Extreme Programming Explained》的诞生，XP 成为软件工程最热门的话题之一。同时，它也是各种争论的中心。

Kent Beck 在 XPE 的前言中的第二句话就对 XP 给出了定义：

XP is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements.

这段话解释了 XP 的主题，这在后面的字里行间都可以看到：

。。。 It reliance on an evolutionary design process that lasts as long as the system lasts.

..... System goes sour – the software is successfully put into production, but after a couple of year the cost making changes or the defect rate rises so much the system must be replaced.

.....Business misunderstood。。。 Business changes。。。

XP 要解决的重要问题就是如何实现更好的交流，从而更加适应需求的变化。它允许用户的需求在任何时候、任何方向上发生变化。为了实现这一点，它没有和其他工程学方法一样，把重点放在过程上，相反更强调个人技巧和实践原则。

XP 的基本原则包括：

Rapid feedback

Assume simplicity

Incremental change

Embracing change

Quality work

为了实现这些原则，Kent Beck 提出了它的极端说：

1. If code reviews are good, we'll review code all the time(pair Programming)
2. If testing is good, every body will test all the time(unit testing),even the customer(functional testing)
3. If design is good, we'll make it part of everybody's daily business(refactoring)
4. If simplicity is good, we'll always leave the system with the simplest design that supports its current functionality(the simplest thing that could possibly work)

5. If architecture is important , everybody will work defining and refining the architecture all the time(metaphor)
6. If integration testing is important , then we'll integrate and test several times a day(continuous integration).
7. If short iterations are good, we'll make the iterations really, really short-seconds and minutes and hours, not weeks and months and years(the Planning Game)

我并不是想在这里讨论 XP 的好坏。但是，我觉得任何人，不管他是否觉得 XP 可行，有一点不可否认：XP 的这些实践原则和程序技巧，是软件界长期以来的积累。即使你不采用 XP 作为你的开发方法学，这些原则和技巧仍然是每一个想成为优秀的程序和设计人员不可欠缺的基本技能。这也是我对待 UML 的态度，你可以不使用 RUP 作为你的过程管理指导，但 UML 本身确实是软件行业对建模技术的一个总结。如果你觉得看图比看文字更清楚，那么 UML 还是有可取之处的（当然，有些人不喜欢看规规矩矩的图形，Kent Beck 就是其中的一个）。

上面可以看到，Refactoring 是 XP 实践的核心之一。不是那么显而易见，但也许更重要的是，Refactoring 和其他的原则和实践之间存在着不可分割的联系。

Kent Beck 指出，你不可能只为今天的代码做设计。不然，你的设计就会走入死胡同，系统就无法不断进化，从而无法适应需求的变化。但是，设计的复杂不但不能解决问题，而且增加了设计的成本，增加了理解的难度，更要命的是增加了扩展和修改的难度，同样使你的系统变得十分僵化。所以，简单的设计-Simple Design--是成功的关键。然而，简单的设计要能够适应未来的需要，必须有一种切实的方法能在以后克服现在的局限性。Refactoring 是其中的关键技术之一。然而，这一点并不是轻而易举能够实现的。要使 Refactoring 能够真正达到这样的目标，你需要能够控制代码的所有权，不然，你就很难修改其他人的代码对你代码作出的引用，因此需要共同的代码所有权 (collective ownership)。为了使 Refactoring 能够更快捷、方便地进行，你的代码必须有良好的风格和大家都同意的代码标准(coding standard)。如果你们成对编程 (pair Programming)，那么你们就能更好地处理复杂的 Refactoring，也使得 Refactoring 出错的可能性大大减小。更进一步，测试 (testing) 能够使得 Refactoring 更加安全。编码是一个团队的工作，所以你需要连续不断的集成 (continuous integration)，这样，当你的 refactoring 和别人的代码发生冲突的时候，你马上就可以知道，原先的代码和设计能够及时得到恢复。你还需要足够的休息 (rest)，这样你才会有更多的勇气，不怕犯错，把 Refactoring 进行到底。

Refactoring 和设计

简化设计

Why should design be simple?

传统的软件方法偏向于进行一次性的 Upfront 设计，我们知道这很难。

软件方法学的设计者通常喜欢用建筑打比方。他们说，如果你要建造一座大厦，那么在你画完所有的施工图和建筑规范之前，你从来不应该开始真正施工。

我喜欢这样的比喻，因为建筑学的发展经历了几千年甚至可能是上万年，而软件行业，最多也只有 5, 60 年的时间，我们需要从建筑中学习很多东西。就如模式语言的提出者，著名的建筑学家，Christopher Alexander，给了设计模式很大的启发。

但是，如果我们只是从建筑学到东西，恐怕还是远远不够。我还不知道哪一栋大厦的外形能够发生变化，或者哪一个大商场在发现自动扶梯位置不太合适时，能够自由移动到一个更加合适的地方（我附近一家大超市的自动扶梯真的让我想到了这一点）。也许要部分达到这样的目的还是有可能的，但肯定要花费极高的代价。但是如果我们去构造一个企业的应用系统，那么我们必须准备好企业发展的同时需要对我们的软件做出改变。

软件的核心就是可变。这种变化不仅仅在于需求的变化，还在于人们理解的变化，而软件这种活动的产物，必须要随之发生改变。

增量迭代的开发方法学是对软件变化的一种响应。但是正如我们前面所讲，增量迭代模型有可能对系统的设计提出比传统的 Waterfall 更复杂的要求。你需要预期所有可能的情形，然后才能保证你是可以增量的，也是能够迭代的。

这种两难的境地来源于一个基本的假设：

削减软件成本的最主要方法是减少返工的成本以及它的可能性。

传统的软件发法学认为，如果我能够尽量早地冻结需求，那么我修改设计的可能性和成本就越小，如果我的设计能够尽早地冻结，那么我修改实现的可能性和成本就越小。要想尽早冻结设计，它必须是非常详尽而复杂的。

我把这种思路称为“改变恐惧症”，不仅仅因为变化是最难研究的东西（制订出条条框框则简单的多），也因为我们没有足够的经验来适应变化，更重要的是，从来没有哪个时代比今天变化更快。

现在我们从另一个角度来想问题：

我们是否可能通过增加返工的方法来削减软件的成本？

这个方法初看来自相矛盾。但是 Kent Beck 说：

The key is that risk is money just as much as time is money。

既然系统在变，那么你不能肯定今天设计的东西一定能够被日后用到。而你需要等待设计完成的成本会很高，因为在你的设计完成之前，没有人可以开始工作。

设计不是独立的。别人要使用你的设计，他必须能够理解你的设计。如果你今天的设计十分复杂，那么你就增加了从今天开始的所有开销，更多的东西需要检查和测试，更多的东西需要理解，更多的东西需要解释。更重要的是，你不能估算出明日的成本。你必须估计日后将要发生什么事情，通常你不可能准确无误地做到这一点。

所以，让设计简单，我们可以通过不断地增量修改设计使得系统更加接近需求，就象我们开车，每次都做一些小小的调整，最后达到目的地。

What is simple design?

要让设计简单，是否意味着我们可以随意 fix and build，写一段，然后任意修改它。但是简单并不意味着随意，当然更不意味着愚蠢。

就象我在 Duplicate Code 里指出的一样，Once and Only Once 通常是代码最简单的形式，因为它没有重复，每一个类都有自己简单的责任，每一个方法都有自己简单的意图。

简单的设计最重要的特性就是容易适应变化。为了达到这样的目的，简单设计应当：

1. 能够简单地被理解。这依赖于你代码的可理解性。只有可理解，下面的简单性才能达到。
2. 能够简单地被修改和扩展。OO 系统往往通过增量的方法改变或增加系统行为，所以也就是要被简单地重用，这要求我们没有重复代码。
3. 有最少数目的类。要易于理解系统，那么系统中每一个类应当和需要解决的问题的每一

个重要概念相对应,如果人工加入许多毫无意义的类或者太多与问题概念无法对应的类,系统将无法理解

4.有最少数目的方法.每一个方法应该有他独立的意义,如果没有可以言述意图的名字,那么系统将难以理解.

如果你看过 Kent Beck 和其他 Agile 联盟作者的文章,或者你看过我的 Duplicate Code 和 Long Method,你可能会觉得 2 和 3,4 存在着矛盾.因为要达到 2 的目的,你必须有很多小类和精干的方法,而 3,4 则要求你具有最少的类和方法.要理解它们之间其实一致的关键在于,你如何决定需要一个新类或者需要一个新的方法.在进行 Refactoring 时,如果你觉得系统的某一部分具有独立的可对应问题领域的概念时,那么你应当毫不犹豫地使用 Extract Class 形成一个新类.反之,某一个类无法具有其独立的应用意义,或者该类数据太多而没有行为,这个时候你应当考虑这个类是不是应该被删除.当然,这里面有些例外(如 method object)同样道理,如果一个代码片断能够有独立意图的行为,那么不管它的大小,可能是一个简单的表达式,都应该有独立的方法,但如果没有这样清晰的意图,再多的代码都可以在一个方法里面.

显然,简单的设计并非我们想象的那么简单.除了我们可以不太考虑以后解决的问题外,简单,作为一种重要审美标准.不是轻易能够达到的.我们一开始的设计往往不能够达到这样的要求,可能是类的划分不合理,可能存在着重复的代码,可能行为的分配需要调整,要达到简单,你就必须 Refactoring 你的代码,使设计更加合理.

How does refactoring support simple upfront design?

开始,你可以作简单的设计,也许是一个类,2、3 个方法,你针对这个类写出 test case, 实现代码,让 test case 通过.在实现这个类的过程中,你发现 2、3 个方法太长,它们之间会有很多重复代码,于是你进行 refactoring,你使用 extract method 让方法更能揭示意图.然后,你解决另外一个问题,你写出另外一个 Test case,可能和上面一个类没有多大的关系,也可能你发现这个类在以前的类上附加了一些功能.这时,你会发现直接对前面一个类进行修改比较困难,你对前面一个类进行 refactoring,让它更容易加入.如果随着新功能的加入,发现前面一个类其实包含了两个应用概念,你可能需要对这个类进行重构,把它拆分为两个类.你进行 refactoring,让所有的 Test cast 都能通过测试.你接着又选取一个需求,增加新的 test case,然后实现它.这时候你发现这个类和原来的某个类具有很多类似的地方,你可能需要把这两个类进行进一步的抽象,通过 refactoring 形成一个新的超类。。

随着时间的推移,越来越多的需求被实现,系统变得越来越大,某一天,你发觉系统有些变样,你觉得有必要修改系统的某一部分结构.这时候,你和你的同僚可能需要暂时摘下增加功能的帽子.你们可能需要更多的讨论,交流,使用任何有意义的方法为你们的讨论增强交流的效果,白板、CRC,然后进行 Refactoring.

Kent Beck 指出,某些大的 Refactoring 可能并不是一天就能完成的.它可能要花费几天甚至一个月的时间.但是,你还需要继续完成用户的需求.这时采取的方法就是小步的增量改变.在实现新需求,写下新的 test case 时,你可能看到一个机会能够让你的代码朝着大目标前进一步.使用这样的方法,每一次你的工作可能是移动一个变量或一个方法.但是随着新功能进一步的加入,这样的机会不断出现.最终,一个大的目标会变成很小的工作.这时候,你已经水到渠成,花上几分钟就完成了.

Refactoring 的这种工作方式可以大大减少 UpFront 的设计量,它同时使你的设计变为一

种必要和需求的产物,这种为了更好地加入新需求所做的设计,更准确地反映了问题的本身。同时,它使得设计随着你对问题的进一步深入而逐渐变得更合理,随着你对新技术的掌握而变得聪明,这是一种进化的设计方法。

设计模式

简单设计的要求

Refactoring 对简单设计的支持好像使设计模式变得有些过时。如果我一开始只需要写下一个简单的 test cast、实现、refactoring, 增加 test cast、实现、refactoring。这是否意味着设计模式就毫无意义了呢?

对设计模式和 Refactoring, 以及与之相关的 Agile 联盟方法学的研究给我一个很深刻的印象。一个方面, 就像 Martin Fowler 指出, Agile 方法学的提倡者往往也是模式社团的领导者。另一方面, 如果你仔细体会一下 Agile 宣言, 你会发觉它们非常重视人的能力。在我自己组织小组试用 XP 的过程中也发现, XP 对它的参与者有很高的要求。

我们已经看到, 简单的设计并不是愚蠢的设计, 相反, 它们是极端聪明的设计。要实现简单的设计, 你必须有丰富的知识。这里在于, 简单设计提倡你不要太多考虑以后的具体需求本身, 但它确实需要你考虑代码和设计的结构。在用 Refactoring 武装你的思想中, 我已经讲到, 如果你的设计结构完全不考虑以后的扩展, 那么 Refactoring 也无能为力。

设计模式是面向对象社团对设计的一个重要总结。其实, 它们也是对简单设计的最好诠释。因为, 好的设计模式, 就是对于解决问题所能发现的最简单、最易重用的设计结构。另一方面, 使用广为人知的设计模式也使得你的代码更容易为人理解、接受, 大大提高了交流的效率、宽度和深度。

从这种意义上讲, Refactoring 不但没有排除设计模式, 它更促进了设计模式的学习、发现和验证, 以及更广泛的应用。Agile 对程序员素质的要求, 在这一点上也得到了充分的体现。一旦小组成员具备熟练运用设计模式的能力, 你的设计可能就是一句话: 用 XXX 模式。(在我领导的 EOSP2P 项目中, 这个模式就是半 State/半策略模式。当我说出这个模式时, 我们发觉整个小组不再需要 upfront 设计, 我们构建, 然后 Refactoring。当然, 象 Factory method 模式几乎使每个系统都应当使用的。还有后续的 proxy 模式, 都让设计变得如此轻松)。

但是, 在应用设计模式的过程中, 你必须时时记住简单性这个原则。设计模式最好的应用方法就是从简单开始, 只要能够解决问题, 抓住这种模式的核心, 那么模式良好的结构保证你的 Refactoring 建立在一个坚实的基础之上。

所以, 为了使 Refactoring 能够更好地进行, 你需要更多的学习模式, 因为设计模式不但是良好设计的开始, 同时也是 Refactoring 的目标。

Refactoring 的目标

Refactoring 有时会失去目标, 你可能会产生循环的 Refactoring, 或者你无法判断 Refactoring 的结果到底有没有让程序的结构更好。

作为 OO 社团最重要贡献之一的设计模式能够为 Refactoring 提供一个明确的目标。在 Ralph Johnson 多篇关于 Refactoring 的论文中都提到了这个问题。Martin Fowler 的《Refactoring》中明确指出的就有 state、strategy、visitor 等模式。更深入一步, 设计模式不但是 Refactoring 的目标, 同时也为看起来有些零散的行为提供了一个全局性的指引。

Christopher Alexander 在《The Timeless way of Building》一书中对模式的定义如下:

Each pattern is a three-part rule, which expression a relation between a certain context, a

problem, and a solution.

很多人在软件领域对他的定义做了修改和扩展，但这三部分内容是不变的。把这个定义放到我们软件上下文来看，设计模式有两个重要特性：

1. 设计模式高于代码层，他不是描述一个好的编码风格，或者某种编程习惯用语；
2. 设计模式不是纯粹理论上的体系结构或者分析方法，它是一种可实际操作的东西。

模式界提出的 Three Rules 就说，你必须提出三个实际的应用例子才能够得上设计模式的条件。

正因如此，很多人都把设计模式称之为 micro-architecture。显然，由于它的可操作性，它可以通过 Refactoring 的方法达到。同时由于它一定层次理论和总结性，他可以用于引导 Refactoring 并作为 Refactoring 的目标。

何时进行 Refactoring

场合

增加功能

如果你的代码不需要增加新的功能，那么你几乎没有必要对他进行 Refactoring。增加功能是进行 Refactoring 最常见的起因。通常，refactoring 的第一个目标就是为了让代码更容易理解。Refactoring 不仅仅可以使难以理解的代码更加清晰，他也可以使你对它的理解呈现一种交互状态，你和代码的交互。理解了，新的功能就很容易加入。

另一个原因是原来代码的结构很难直接扩展，可能是因为一个类和另外一个类的绑定太紧，或者是他使用了大量的 case 语句，或者是有一些重复的代码让你觉得加入你的代码会使结构更差。总而言之，直接加入新功能不是太方便。这时候，你就需要 Refactoring，使得旧的代码在能让旧的 test case 通过的情况下，让新的代码加入更加方便、快捷。

重用

重用可以说是增加功能的一个特例。因为这是 OO 增加功能最主要的方法。但这里有一些差别。

如果你的代码是一个框架，那么当把代码交给其他人使用时，你必须重点考虑框架的重用方法。譬如，你想在框架之上增加一层 facade，这时候，你可能需要 refactoring 一些内部类，让这个 Facade 的结构更容易加入。

对框架 Refactoring 可能需要是一个反馈的过程。因为框架一旦稳定就可能长期存在，一般你不会直接对这个框架增加功能。这时，不同应用程序对框架使用经验的积累可能使得你的 Refactoring 具有更加明确的目标，更加深层次的抽象。

修正 bug

bug 产生的原因有很多。可能是你对解决的问题理解不够，也可能是程序的代码过于纷乱。当你的代码结构不是很清晰时，bug 往往隐藏在代码之中，使你很难发现。应用 Refactoring 让代码结构更清晰的同时，也使得 bug 自动浮出水面。

Refactoring 所追求的目标和过程往往使得代码的 Bug 无处遁形。Eric E. Allen 在 IBM 中

developerWorks 的 bug pattern 中的一个模式就是 copy-and-paste 方法造成代码重复，从而产生 bug 的一个例子。这个 Rogue Title Pattern 是一个非常常见的 bug 例子，当你修正了程序中一个 bug 时，你发现程序运行的结果是你明明已经修正的 bug 还在作怪。究其原因，就是程序员在实现一个功能的时候把一段代码复制过去，稍加修改，就成为一个新的函数。你修改了一个地方，但却忘了另外一个地方。

Refactoring 力求排除重复代码，如果有这样的 bug，在 refactoring 以后，一次修改就能纠正所有的问题。

Code Review

Review 可以使得专家知识传播到整个开发队伍。他们能够帮助更多的人理解一个大系统更多方面的问题。它们对于编写清晰的代码也有很好的帮助。俗话说得好，三个臭皮匠顶一个诸葛亮，Review 也使得你能够接受更多其他人的好意见和想法，从而使你的设计和代码更加合理。

按照 Martin Fowler 的建议，我使用 refactoring 作为我的主要 review 方法。我有比较多的机会做这样的 Code Review，在程序员发现一个难以解决的 bug，或者当他们觉得自己的思路无法进行下去的时候，他们通常会要求我的帮助。

我的方法通常是问几个简单的问题，然后直接找到他们的代码，进行 refactoring，在这个过程中，我总是一边 refactoring，一边和程序员讲解我这样做的意图。令人惊奇的是，一旦我这样做，程序员往往能够在我 refactoring 的过程中自己发现问题。由于 Refactoring 的结果非常具体，我很容易对程序员讲述一些本来就可以直接这样实现的例子，这对提高程序员的编程能力有极大的帮助。

味道

Kent Beck 的“Code Smell”或许是 OO 社团最有人情味的名言了。Martin Fowler 和 Kent Beck 专门合著了《Refactoring》其中的一章就叫《Bad Smells in Code》。

虽然对 Refactoring 的研究涉及到了形式化地探测代码的结构，譬如专门的重复代码监测工具。但这些研究尚处于幼年期。更何况，很多代码的结构并不是能够用简单的数字统计和抽象语法树所能解决的。

Code Smell 是高水平的程序员对代码的一种感觉，当你能够闻到这样的味道时，你就可以在不涉及到程序所要解决的具体问题时，就“闻到”代码结构的好坏。

Kent Beck 和 Martin Fowler 列出的代码味道有：

1. Duplicated Code

代码重复几乎是最常见的异味了。他也是 Refactoring 的主要目标之一。代码重复往往来自于 copy-and-paste 的编程风格。与他相对应 OAOO 是一个好系统的重要标志（请参见我的 duplicated code 一文）。

2. Long method

它是传统结构化的“遗毒”。一个方法应当具有自我独立的意图，不要把几个意图放在一起，我的《大类和长方法》一文中有详细描述。

3. Large Class

大类就是你把太多的责任交给了一个类。这里的规则是 One Class One

Responsibility。参见我的《大类和长方法》。

4. Divergent Change

一个类里面的内容变化率不同。某些状态一个小时变一次，某些则几个月一年才变一次；某些状态因为这方面的原因发生变化，而另一些则因为其他方面的原因变一次。

面向对象的抽象就是把相对不变的和相对变化相隔离。把问题变化的一方面和另一方面相隔离。这使得这些相对不变的可以重用。问题变化的每个方面都可以单独重用。这种相异变化的共存使得重用非常困难。

5. Shotgun Surgery

这正好和上面相反。对系统一个地方的改变涉及到其他许多地方的相关改变。这些变化率和变化内容相似的状态和行为通常应当放在同一个类中。

6. Feature Envy

对象的目的是封装状态以及与这些状态紧密相关的行为。如果一个类的方法频繁用 `get` 方法存取其他类的状态进行计算，那么你要考虑把行为移到涉及状态数目最多的那个类。

7. Data Clumps

某些数据通常像孩子一样成群玩耍：一起出现在很多类的成员变量中，一起出现在许多方法的参数中.....，这些数据或许应该自己独立形成对象。

8. Primitive Obsession

面向对象的新手通常习惯使用几个原始类型的数据来表示一个概念。譬如对于范围，他们会使用两个数字。对于 `Money`，他们会用一个浮点数来表示。因为你没有使用对象来表达问题中存在的概念，这使得代码变的难以理解，解决问题的难度大大增加。

好的习惯是扩充语言所能提供原始类型，用小对象来表示范围、金额、转化率、邮政编码等等。

9. Switch Statement

基于常量的开关语句是 OO 的大敌，你应当把他变为子类、`state` 或 `strategy`。

10. Parallel Inheritance Hierarchies

并行的继承层次是 `shotgun surgery` 的特殊情况。因为当你改变一个层次中的某一个类时，你必须同时改变另外一个层次的并行子类。

11. Lazy Class

一个干活不多的类。类的维护需要额外的开销，如果一个类承担了太少的责任，应当消除它。

12. Speculative Generality

一个类实现了从未用到的功能和通用性。通常这样的类或方法唯一的用户是 `test case`。不要犹豫，删除它。

13. Temporary Field

一个对象的属性可能只在某些情况下才有意义。这样的代码将难以理解。专门建立一个对象来持有这样的孤儿属性，把只和他相关的行为移到该类。最常见的是一个特定的算法需要某些只有该算法才有用的变量。

14. Message Chain

消息链发生于当一个客户向一个对象要求另一个对象，然后客户又向这另一对象要求另一个对象，再向这另一个对象要求另一个对象，如此如此。这时，你需要隐藏分派。

15. Middle Man

对象的基本特性之一就是封装，而你经常会通过分派去实现封装。但是这一步不能

走得太远，如果你发现一个类接口的一大半方法都在做分派，你可能需要移去这个中间人。

16. Inappropriate Intimacy

某些类相互之间太亲密，它们花费了太多的时间去钻研别人的私有部分。对人类而言，我们也许不应该太假正经，但我们应当让自己的类严格遵守禁欲主义。

17. Alternative Classes with Different Interfaces

做相同事情的方法有不同的函数 `signature`，一致把它们往类层次上移，直至协议一致。

18. Incomplete Library Class

要建立一个好的类库非常困难。我们大量的程序工作都基于类库实现。然而，如此广泛而又相异的目标对库构建者提出了苛刻的要求。库构建者也不是万能的。有时候我们会发现库类无法实现我们需要的功能。而直接对库类的修改有非常困难。这时候就需要用各种手段进行 Refactoring。

19. Data Class

对象包括状态和行为。如果一个类只有状态没有行为，那么肯定有什么地方出问题了。

20. Refused Bequest

超类传下来很多行为和状态，而子类只是用了其中的很小一部分。这通常意味着你的类层次有问题。

21. Comments

经常觉得要写很多注释表示你的代码难以理解。如果这种感觉太多，表示你需要 Refactoring。

Refactoring 和命名

Refactoring 使代码更容易理解的最有效和常用的手段之一就是给类、方法和属性一个恰如其分的名字。

很多公司和个人都编出了大量的代码命名规范。如果这些规范太繁琐，那就无法执行。

有好的命名规则是好事，但并不总是如此。最重要的是，在整个项目开发团队内，必须保持一致的命名规则，也许不是世界上最好的，但却是对该项目最好的。

下面一些是 Refactoring 中一些有效的命名原则：

类命名

Kent Beck 提出其中大多数的规则。Kent Beck 的《Smalltalk Best Practice Pattern》是我至今为止读过的所有书里面最符合我审美观点的书籍——渊博的知识和春秋的笔法。

简单超类名—传达设计目的

命名总要受各种因素的影响。你想把名字取得尽量短，易于输入、格式化、容易说出口。同时，你也想让名字尽可能包含更多的信息，这样读者就能够更快地理解名字所包含的实际内容。你可能取一些尽量为人所熟悉的名字，这样在名字中就可以传递更多的共同经验。同时，你也想让名字尽量唯一，别人的代码命名就不会和你重复。

Kent Beck 给出的第一个规则就是不要缩写。缩写对输入（20 年内 10-100 次）的考虑多余对理解（20 年内 1000-10000 次）的考虑。理解缩写词需要两步，第一步理解这些缩写代表的词语，第二步采取理解这些词语所代表的意义。

对一个高层次的根类命名是一个重大的决策。在未来的 20 年中，人们可能不断地使用你所给出的名字。你必须不犯错误。

人们通常在命名超类时加上各种各样的修饰，富有计算机科学意义、给人深刻印象但最终却没有意义的单词，如 Object, Thing, Component, Part, Manager, Entity 或 Item。

你在创建一个词汇表，而不是写一个程序。暂时做一回诗人。简单、生气勃勃、容易记忆远远比说出全部内容更有效。规则：

Name a superclass with a single word that conveys its purpose in the design.

（用一个单词命名超类，传达它的设计目的），好的例子包括：

Number 、Collection、Magnitude、Model

全称子类名—区分异同

命名类的一种方法是给一个唯一的名字。唯一的名字可以让你用最短的信息表达最多的信息。

这对于通用术语来说是对的。Array 是 Collection 的一个子类，因为绝大多数人都知道“Array”意味着什么。

但在绝大多数情况下，类继承的层次结构对于理解你的代码十分重要，特别是一个子类概念上是超类的变种同时又和超类共享实现的情况。你需要传递两部分信息：

1. 新类如何相同，以及
2. 新类如何不同

要表达相同，你可以借用超类的名字。这不一定是直接的子类 and 父类关系，层次上有一定距离也无妨。要表达不同，你需要一个单词确切地强调新类为什么不是超类的理由。所以，有规则：

Name subclasses in your hierarchies by prepending on an adjective to the superclass name.

（在超类名字前加上内容命名你层次中的子类。）

例子：

BigInteger 是一个可以表达很大很大数字的整数。

方法命名—揭示意图

为什么好的代码总有很多简单的方法，代码可能只有一行，譬如：

```
class ParagraphEditor...
    public void highlight (aRectangle:Rectangle) {
        reverse(aRectangle);
    }
```

这个 `highlight` 只是分派到 `reverse`，为什么还需要？

关键在于 **Communication**。因为有了这个方法，以后的代码就可以用 `highlight` 来说话。我要加亮一个区域，我就 `highlight` 它。

你当然可以直接使用 `reverse`，结果是一样的。但所有调用代码都揭示了实现——“我通过反转一个矩形来加亮它”。

代码应当揭示意图，它另外的好处是可以更方便通过继承修整。如果你想要一个 `ParagraphEditor` 用颜色加亮，那么只需继承 `ParagraphEditor` 并覆盖 `highlight` 即可。所以：

communicate what is to be done rather than how it is to be done.

传达你要做什么（接口）而不是你如何做（实现）。

实例（临时）变量命名—暗示角色

任何实例变量传达的信息包括两部分：

1. 什么是它的目的？
2. 它如何被使用？

一个变量的目的或者说它担当的角色对读者非常重要，因为它能够正确地引导读者的注意力。一般，你在阅读代码时脑袋里总有一个目的。如果你理解变量的角色，而这个角色和你的目的无关，那么你可以直接跳过使用该变量的无关代码。如果发现该变量的角色和你的目的有关，那么你就能马上缩小你的阅读范围——那些引用此变量的相关代码。

一个变量如何被使用和发送给它什么消息通常是它的“类型”。理解类型并非不重要。但是，对于实例变量来说，你能够了解这个变量所担当角色的唯一地方就是它的名字。如果一个 `Point` 中有两个实例变量叫做 `int1`、`int2` 而并非 `x`、`y`，那么在你理解哪一个是横向坐标，哪一个是纵向坐标前，你可能要都上一堆代码。更何况你还要阅读与之相关的很多代码。而变量的类型很容易从它声明、传递给它的消息看出来。所以：

Name instance variable for the role they play in the computation. Make the name plural if the variable will hold a Collection.

（用计算中实例变量所承担的角色对它命名。如果变量持有一个集合，使用复数。）

参考书目

1. Martin Fowler : 《Refactoring: Improving the design of existing Code》
2. Kent Beck: 《Extreme Programming Explained》
3. Kent Beck: 《Smalltalk Best Practice Pattern》
4. William C. Wake: 《Extreme Programming Explored》
5. Jeff Langr: 《Evolution of Test and Code Via Test-First Design》
6. Robert C. Martin: 《An Extreme Programming Episode》
7. Don Robert, John Brant and Ralph Johnson: 《A Refactoring Tool for Smalltalk》
8. Jrefactory: jrefactory.Sourceforge.net
9. Gamma , Junit: <http://members.pingnet.ch/gamma/junit.htm>
10. Refactoring Browser: <http://chip.cs.uiuc.edu/users/brant/Refactory/>
11. GOF: 《Design Patterns: Elements of Reusable Object Oriented Software》
12. MartinFowler : 《UML distilled》
13. Braint Foote and Joseph Yoder: 《Big Ball of Mud》
14. William Opdyke Ph.D. Thesis: 《Refactoring Object-Oriented Frameworks》
15. Braint Foote and William Opdyke: 《Life Cycle and Refactoring Patterns that Support Evolution and Reuse》
16. Mel O Cinneide: Ph.D. : 《Automated Application of Design Patterns: a Refactoring Approach》
17. Donald Bradley Roberts: Ph.D.: 《Practical Analysis for Refactoring》
18. C2: www.c2.com
19. Eric Gamma, Kent Beck: Junit: <http://www.junit.org>
20. Lance Tokuda: 《Evolving Object-Oriented Designs with Refactoring》
21. Michael Hunger: 《Refactoring: Benefits and Disadvantages》
22. TOMCAT BOOK
23. Shiyiying: Code Smell serials
24. William F. Opdyke, Object-Oriented Refactoring, Legacy Constraints and Reuse
25. Stephen R.Schach: 《Classical and Object-Oriented Software Engineering》
26. Christopher Alexander: 《The Timeless Way of Building》