细细品味C#

——重构的艺术

精

华

集

锦

csAxp

虾皮工作室

http://www.cnblogs.com/xia520pi/

2011年10月5日

目录

1、代码重构

1.1、版权声明

文章出处：http://kb.cnblogs.com/page/65289/
文章作者：暂无

1.2、内容详情

重构（Refactoring）就是在不改变软件现有功能的基础上，通过调整程序代码改善软件的质量、性能，使其程序的设计模式和架构更趋合理，提高软件的扩展性和维护性。

也许有人会问，为什么不在项目开始时多花些时间把设计做好，而要以后花时间来重构呢？要知道一个完美得可以预见未来任何变化的设计，或一个灵活得可以容纳任何扩展的设计是不存在的。系统设计人员对即将着手的项目往往只能从大方向予以把控，而无法知道每个细枝末节，其次永远不变的就是变化，提出需求的用户往往要在软件成型后，始才开始"品头论足"，系统设计人员毕竟不是先知先觉的神仙，功能的变化导致设计的调整再所难免。所以"测试为先，持续重构"作为良好开发习惯被越来越多的人所采纳，测试和重构像黄河的护堤，成为保证软件质量的法宝。

【为什么要重构（Refactoring）】

在不改变系统功能的情况下，改变系统的实现方式。为什么要这么做？投入精力不用来满足客户关心的需求，而是仅仅改变了软件的实现方式，这是否是在浪费客户的投资呢？

重构的重要性要从软件的生命周期说起。软件不同与普通的产品，他是一种智力产品，没有具体的物理形态。一个软件不可能发生物理损耗，界面上的按钮永远不会因为按动次数太多而发生接触不良。那么为什么一个软件制造出来以后，却不能永远使用下去呢？

对软件的生命造成威胁的因素只有一个：需求的变更。一个软件总是为解决某种特定的需求而产生，时代在发展，客户的业务也在发生变化。有的需求相对稳定一些，有的需求变化的比较剧烈，还有的需求已经消失了，或者转化成了别的需求。在这种情况下，软件必须相应的改变。

考虑到成本和时间等因素，当然不是所有的需求变化都要在软件系统中实现。但是总的说来，软件要适应需求的变化，以保持自己的生命力。

这就产生了一种糟糕的现象：软件产品最初制造出来，是经过精心的设计，具有良好架构的。但是随着时间的发展、需求的变化，必须不断的修改原有的功能、追加新的功能，还免不了有一些缺陷需要修改。为了实现变更，不可避免的要违反最初的设计构架。经过一段时间以后，软件的架构就千疮百孔了。bug 越来越多，越来越难维护，新的需求越来越难实现，软件的构架对新的需求渐渐的失去支持能力，而是成为一种制约。最后新需求的开发成本会超过开发一个新的软件的成本，这就是这个软件系统的生命走到尽头的时候。

重构就能够最大限度的避免这样一种现象。系统发展到一定阶段后，使用重构的方式，不改变系统的外部功能，只对内部的结构进行重新的整理。通过重构，不断的调整系统的结构，使系统对于需求的变更始终具有较强的适应能力。

通过重构可以达到以下的目标：

- 持续偏纠和改进软件设计

重构和设计是相辅相成的，它和设计彼此互补。有了重构，你仍然必须做预先的设计，但是不必是最优的设计，只需要一个合理的解决方案就够了，如果没有重构、程序设计会逐渐腐败变质，愈来愈像断线的风筝，脱缰的野马无法控制。重构其实就是整理代码，让所有带着发散倾向的代码回归本位。

- 使代码更易为人所理解

Martin Flower 在《重构》中有一句经典的话："任何一个傻瓜都能写出计算机可以理解的程序，只有写出人类容易理解的程序才是优秀的程序员。"对此，笔者感触很深，有些程序员总是能够快速编写出可运行的代码，但代码中晦涩的命名使人晕眩得需要紧握坐椅扶手，试想一个新兵到来接手这样的代码他会不会想当逃兵呢？

软件的生命周期往往需要多批程序员来维护，我们往往忽略了这些后来人。为了使代码容易被他人理解，需要在实现软件功能时做许多额外的事件，如清晰的排版布局，简明扼要的注释，其中命名也是一个重要的方面。一个很好的办法就是采用暗喻命名，即以对象实现的功能的依据，用形象化或拟人化的手法进行命名，一个很好的态度就是将每个代码元素像新生儿一样命名，也许笔者有点命名偏执狂的倾向，如能荣此雅号，将深以此为幸。

对于那些让人充满迷茫感甚至误导性的命名，需要果决地、大刀阔斧地整容，永远不要手下留情！

- 帮助发现隐藏的代码缺陷

孔子说过：温故而知新。重构代码时逼迫你加深理解原先所写的代码。笔者常有写下程序后，却发生对自己的程序逻辑不甚理解的情景，曾为此惊悚过，后来发现这种症状居然是许多程序员常患的"感冒"。当你也发生这样的情形时，通过重构代码可以加深对原设计的理解，发现其中的问题和隐患，构建出更好的代码。

- 从长远来看，有助于提高编程效率

当你发现解决一个问题变得异常复杂时，往往不是问题本身造成的，而是你用错了方法，拙劣的设计往往导致臃肿的编码。

改善设计、提高可读性、减少缺陷都是为了稳住阵脚。良好的设计是成功的一半，停下来通过重构改进设计，或许会在当前减缓速度，但它带来的后发优势却是不可低估的。

【何时着手重构（Refactoring）】

新官上任三把火，开始一个全新??、脚不停蹄、加班加点，一支声势浩大的千军万"码"夹裹着程序员激情和扣击键盘的鸣金奋力前行，势如破竹，攻城掠地，直指"黄龙府"。

开发经理是这支浩浩汤汤代码队伍的统帅，他负责这支队伍的命运，当齐恒公站在山顶上看到管仲训练的队伍整齐划一地前进时，他感叹说"我有这样一支军队哪里还怕没有胜利呢？"。但很遗憾，你手中的这支队伍原本只是散兵游勇，在前进中招兵买马，不断壮大，

所以队伍变形在所难免。当开发经理发觉队伍变形时，也许就是克制住攻克前方山头的诱惑，停下脚步整顿队伍的时候了。

Kent Beck 提出了"代码坏味道"的说法，和我们所提出的"队伍变形"是同样的意思，队伍变形的信号是什么呢？以下列述的代码症状就是"队伍变形"的强烈信号：

- 代码中存在重复的代码

中国有 118 家整车生产企业，数量几乎等于美、日、欧所有汽车厂家数之和，但是全国的年产量却不及一个外国大汽车公司的产量。重复建设只会导致效率的低效和资源的浪费。

程序代码更是不能搞重复建设，如果同一个类中有相同的代码块，请把它提炼成类的一个独立方法，如果不同类中具有相同的代码，请把它提炼成一个新类，永远不要重复代码。

- 过大的类和过长的方法

过大的类往往是类抽象不合理的结果，类抽象不合理将降低代码的复用率。方法是类王国中的诸侯国，诸侯国太大势必动摇中央集权。过长的方法由于包含的逻辑过于复杂，错误机率将直线上升，而可读性则直线下降，类的健壮性很容易被打破。当看到一个过长的方法时，需要想办法将其划分为多个小方法，以便于分而治之。

- 牵一毛而需要动全身的修改

当你发现修改一个小功能，或增加一个小功能时，就引发一次代码地震，也许是你的设计抽象度不够理想，功能代码太过分散所引起的。

- 类之间需要过多的通讯

A 类需要调用 B 类的过多方法访问 B 的内部数据，在关系上这两个类显得有点狎昵，可能这两个类本应该在一起，而不应该分家。

- 过度耦合的信息链

"计算机是这样一门科学，它相信可以通过添加一个中间层解决任何问题"，所以往往中间层会被过多地追加到程序中。如果你在代码中看到需要获取一个信息，需要一个类的方法调用另一个类的方法，层层挂接，就象输油管一样节节相连。这往往是因为衔接层太多造成的，需要查看就否有可移除的中间层，或是否可以提供更直接的调用方法。

- 各立山头干革命

如果你发现有两个类或两个方法虽然命名不同但却拥有相似或相同的功能，你会发现往往是因为开发团队协调不够造成的。笔者曾经写了一个颇好用的字符串处理类，但因为没有及时通告团队其他人员，后来发现项目中居然有三个字符串处理类。革命资源是珍贵的，我们不应各立山头干革命。

- 不完美的设计

在笔者刚完成的一个比对报警项目中，曾安排阿朱开发报警模块，即通过 Socket 向指定的短信平台、语音平台及客户端报警器插件发送报警报文信息，阿朱出色地完成了这项任务。后来用户又提出了实时比对的需求，即要求第三方系统以报文形式向比对报警系统发送请求，比对报警系统接收并响应这个请求。这又需要用到 Socket 报文通讯，由于原来的设

计没有将报文通讯模块独立出来，所以无法复用阿朱开发的代码。后来我及时调整了这个设计，新增了一个报文收发模块，使系统所有的对外通讯都复用这个模块，系统的整体设计也显得更加合理。

每个系统都或多或少存在不完美的设计，刚开始可能注意不到，到后来才会慢慢凸显出来，此时唯有勇于更改才是最好的出路。

- 缺少必要的注释

虽然许多软件工程的书籍常提醒程序员需要防止过多注释，但这个担心好象并没有什么必要。往往程序员更感兴趣的是功能实现而非代码注释，因为前者更能带来成就感，所以代码注释往往不是过多而是过少，过于简单。人的记忆曲线下降的坡度是陡得吓人的，当过了一段时间后再回头补注释时，很容易发生"提笔忘字，愈言且止"的情形。

曾在网上看到过微软的代码注释，其详尽程度让人叹为观止，也从中体悟到了微软成功的一个经验。

【重构（Refactoring）的难题】

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

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

这一小节比我希望的要短。暂且如此吧。你应该尝试一下重构，获得它所提供的利益，但在此同时，你也应该时时监控其过程，注意寻找重构可能引入的问题。请让我们知道你所遭遇的问题。随着对重构的了解日益增多，我们将找出更多解决办法，并清楚知道哪些问题是真正难以解决的。

- 数据库（Databases）

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

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

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

对开发者而言，对象数据库既有帮助也有妨碍。某些面向对象数据库提供不同版本的对象之间的自动迁移功能，这减少了数据迁移时的工作量，但还是会损失一定时间。如果各数

据库之间的数据迁移并非自动进行，你就必须自行完成迁移工作，这个工作量可是很大的。这种情况下你必须更加留神 classes 内的数据结构变化。你仍然可以放心将 classes 的行为转移过去，但转移值域（field）时就必须格外小心。数据尚未被转移前你就得先运用访问函数（accessors）造成「数据已经转移」的假象。一旦你确定知道「数据应该在何处」时，就可以一次性地将数据迁移过去。这时惟一需要修改的只有访问函数（accessors），这也降低了错误风险。

- 修改接口（Changing Interfaces）

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

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

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

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

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

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

「保留旧接口」的办法通常可行，但很烦人。起码在一段时间里你必须建造（build）并维护一些额外的函数。它们会使接口变得复杂，使接口难以使用。还好我们有另一个选择：不要发布（publish）接口。当然我不是说要完全禁止，因为很明显你必得发布一些接口。如果你正在建造供外部使用的 APIs，像 Sun 所做的那样，肯定你必得发布接口。我之所以说尽量不要发布，是因为我常常看到一些开发团队公开了太多接口。我曾经看到一支三人团队这么工作：每个人都向另外两人公开发布接口。这使他们不得不经常来回维护接口，而其实他们原本可以直接进入程序库，径行修改自己管理的那一部分，那会轻松许多。过度强调「代码拥有权」的团队常常会犯这种错误。发布接口很有用，但也有代价。所以除非真有必要，别发布接口。这可能意味需要改变你的代码拥有权观念，让每个人都可以修改别人的代码，以适应接口的改动。以搭档（成对）编程（Pair Programming）完成这一切通常是个好主意。

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

Java 之中还有一个特别关于「修改接口」的问题：在 throws 子句中增加一个异常。这

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

- 难以通过重构手法完成的设计改动

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

这种情况下我的办法就是「先想象重构的情况」。考虑候选设计方案时，我会问自己：将某个设计重构为另一个设计的难度有多大？如果看上去很简单，我就不必太担心选择是否得当，于是我就会选最简单的设计，哪怕它不能覆盖所有潜在需求也没关系。但如果预先看不到简单的重构办法，我就会在设计上投入更多力气。不过我发现，这种情况很少出现。

- 何时不该重构？

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

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

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

另外，如果项目已近最后期限，你也应该避免重构。在此时机，从重构过程赢得的生产力只有在最后期限过后才能体现出来，而那个时候已经时不我予。Ward Cunningham 对此有一个很好的看法。他把未完成的重构工作形容为「债务」。很多公司都需要借债来使自己更有效地运转。但是借债就得付利息，过于复杂的代码所造成的「维护和扩展的额外开销」就是利息。你可以承受一定程度的利息，但如果利息太高你就会被压垮。把债务管理好是很重要的，你应该随时通过重构来偿还一部分债务。

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

【重构（Refactoring）与设计】

「重构」肩负一项特别任务：它和设计彼此互补。初学编程的时候，我埋头就写程序，

浑浑噩噩地进行开发。然而很快我便发现，「事先设计」（upfront design）可以助我节省回头工的高昂成本。于是我很快加强这种「预先设计」风格。许多人都把设计看作软件开发的关键环节，而把编程（programming）看作只是机械式的低级劳动。他们认为设计就像画工程图而编码就像施工。但是你要知道，软件和真实器械有着很大的差异。软件的可塑性更强，而且完全是思想产品。正如 Alistair Cockburn 所说：『有了设计，我可以思考更快，但是其中充满小漏洞。』

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

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

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

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

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

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

劳而无获— Ron Jeffries

Chrysler Comprehensive Compensation（克莱斯勒综合薪资系统）的支付过程太慢了。虽

然我们的开发还没结束，这个问题却已经开始困扰我们，因为它已经拖累了测试速度。

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

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

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

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

处理日期区间时我们遵循这样一个规则：如果结束日期在起始日期之前，这个日期区间就该是空的。这是一条很好的规则，完全符合这个 class 的需要。采用此一规则后不久，我们意识到，创建一个「起始日期在结束日期之后」的日期区间，仍然不算是清晰的代码，于是我们把这个行为提炼到一个 factory method（译注：一个著名的设计模式，见《Design Patterns》），由它专门创建「空的日期区间」。

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

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

我们完全错了。除了一场很有趣的交谈，我们什么好事都没做。

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

【重构与性能（Performance）】

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

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

我看过三种「编写快速软件」的方法。其中最严格的是「时间预算法」（time budgeting），

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

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

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

第三种性能提升法系利用上述的 "90%" 统计数据。采用这种方法时，你以一种「良好的分解方式」（well-factored manner）来建造自己的程序，不对性能投以任何关切，直至进入性能优化阶段 — 那通常是在开发后期。一旦进入该阶段，你再按照某个特定程序来调整程序性能。

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

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

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

优化一个薪资系统— Rich Garzaniti

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

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

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

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

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

使用这些技术，我们稳步提高了系统性能。开发过程中原本似乎需要 1,000 小时以上才能完成的薪资计算，实际运作时只花 40 小时。一个月后我们把时间缩短到 18 小时。正式投入运转时只花 12 小时。经过一年的运行和改善后，全部计算只需 9 小时。

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

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

2、项目重构方案设计

2.1、版权声明

文章出处：http://www.cnblogs.com/zenghongliang/archive/2010/06/archive/2010/06/23/1763438.html
文章作者：圣殿骑士（Knights Warrior）

2.2、内容详情

最近接手到一个已经成型的项目，然后我们的任务就是对它进行重构，这个项目是一个功能很齐全的 WPF 视频播放器（附带很多其他功能），在仔细研究了项目的背景和架构以后，初步做出了一下的重构方案：

目前现状：

虽然整个系统做得很漂亮，代码也写得不错，但仍有以下不足：

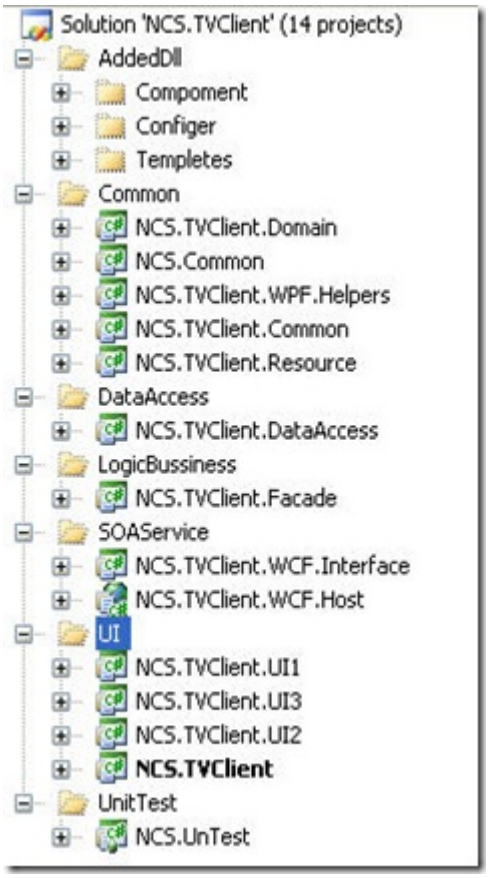
- 架构有待改善。虽然看似 MVC 架构，却没有遵循 MVC 的模式，里面逻辑和 UI 耦合很高，没有清晰的规律。
- 没有充分用到 WPF 的特性。WPF 除了给我们很多炫丽的效果外，还给我们提供了诸如 Binding,command 等特性，这些特性可以帮我们隔开耦合，同时减少代码量。
- 代码和文件没有组织。代码、dll、样式文件和资源文件等没有统一的组织，到处都有，这样看起来就会很混乱。
- 没有建立公用代码库。没有把公用的代码库独立出来，很多地方都是另外在写，这样既增加了代码量，同时维护和重构也带来了麻烦。
- 逻辑处理不应暴露在 Client 端。项目是一个 C/S 架构的系统，没有必要把所有的逻辑都暴露在 Client 端，应该用分布式把 Logic 放在服务器端，这样可以更安全同时使客户端变小。
- 没有单元测试。这样一个庞大的程序，没有单元测试是非常危险的，我们不可能做到 100%的覆盖率，但是我们可以对主要的逻辑和 Function 做单元测试，这样既减少了测试人员的工作量同时整个系统的安全、稳定和可维护性得到了大大的提高。
- 性能不够优化。启动项目,通过 WPF 性能工具 Perforator 和 Visual Profiler 分析得出，程序启动和界面操作都导致 CPU 很高，内存也消耗比较多。

解决方案：

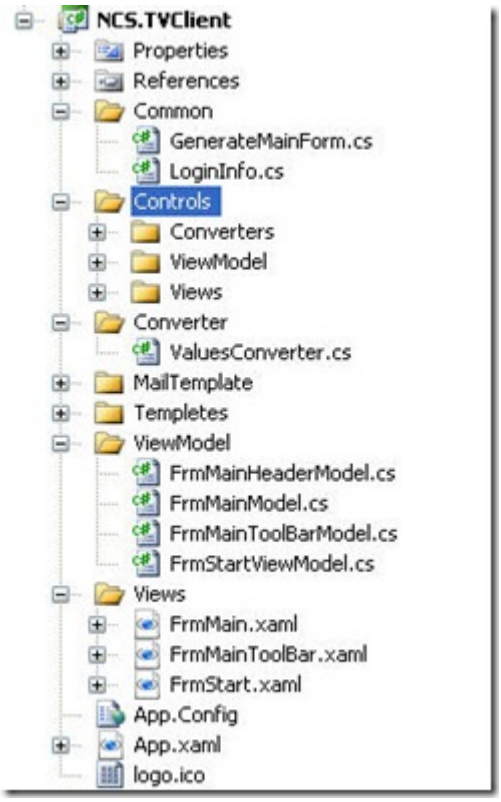
- 针对缺陷 1 的“架构问题”。做法是采用 MVP 或者 MVVM 模式，目前正在对比和考虑。
- 针对缺陷 2 的“WPF 特性”。做法是充分利用 Binding,command 等特性。
- 针对缺陷 3 的“代码和文件没有组织”。做法是建立一些单独的工程或者文件来分类和组织这些代码，并且充分隔离耦合。

- 针对缺陷 4 的“没有建立公用代码库”。做法是把一些公用的代码和常用的代码做成单独的 Dll,并且有完整的单元测试，这样才能提高效率。
- 针对缺陷 5 的“逻辑处理不应暴露在 Client 端”。做法是用 WCF 做为中间层，把业务逻辑全部进行封装，通过 WCF 提供统一的接口供项目调用。
- 针对缺陷 6 的“没有单元测试”。做法是不管用 MVP 还是 MVVM，我们起码保证对逻辑组件的代码有充分的单元测试覆盖，同时对一些公用的组件也要有单独的单元测试代码。
- 针对缺陷 7 的“性能不够优化”。这个我会单独做一个性能优化列表出来，针对耗资源的操作和其他有损害性能的操作，我们应该避免。

那么我们就可以结合实际情况搭建如下的结构 ——如图（左）
因为使用了 MVVM 模式，所以 UI 结构图就做如下调整——如图（右）

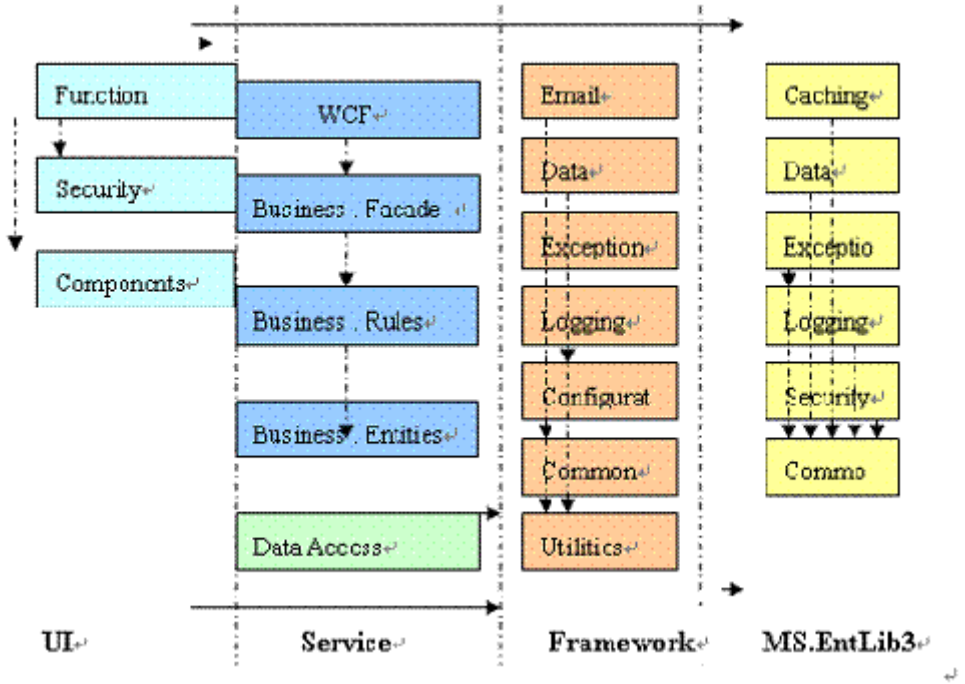


（左）图



（右）图

由于整个项目客户部希望我们引用第三方的组件或者工具，所以很多功能都只能通过企业库实现，比如 AOP 和 IOC,log 和 exception 对项目特征做了定制化，数据访问通过企业库重写实现局部 ORM,对性能要求比较高的应用仍然实现存储过程。对所有事务操作都转移到数据库，邮件使用 JOB 进行发送。大型数据和客户要求较高的实时操作，用 MSMQ 和 SSB 相结合的方式。层次依赖关系



UI: 功能模块使用时候,都会首先通过 UI 层次 Security 模块的安全验证(验证是通过 Components 模块里面的自定义的用于页面功能以及功能点验证的控件触发), Security 模块会通过服务层获取用户身份数据,用于页面验证.

功能模块的实际功能实现,如果需要数据库支持,那么依然会通过服务层进行数据操作.整个架构基于 MVVM 模式。

Service:通过 WCF 做中间服务，使应用隔离开来，这样有利于扩展和维护，同事提高了整个应用程序的伸缩性。

Business Logic: 服务层内部之间的组合关系,主要体现再依赖和调用,由上往下调用,逐级依赖,最后 Service 底层边界 Data Access 模块将调用 Framework 中的 Data 模块,Data 模块将调用 MS.EntLib3 中的 Data,向数据服务器发送数据操作命令和数据.

Framework: 该层次提供许多基础的功能模块（七大块）,分别提供给 UI,Service 层里面的模块直接或者间接的调用,同时也可以看到 Framework 层次内部各模块之间再运行时也有互相依赖调用的关系存在.该层次的部分模块会依赖和调用 Ms.EntLib3 中的模块,一般是按照两个层次里面的模块名称,产生关系的.

MS.EntLib3: 该层次的各个模块是整个系统框架中最底层的,只会在运行时被更高层次的模块依赖和调用,同时该层次内部各个模块之间也存在依赖和运行时调用关系.

整个架构采用迭代的方式进行开发，这样方便客户进行实时反馈，由于现在还没有开始，所以有很多时间进行准备，如果园子里有这方面经验的朋友，也可以畅所欲言，谢谢！

3、31 天重构学习笔记

3.1、版权声明

文章出处：http://www.cnblogs.com/KnightsWarrior/archive/2010/06/30/1767981.html
文章作者：圣殿骑士（Knights Warrior）

3.2、内容详情

3.2.1 封装集合

概念：本文所讲的封装集合就是把集合进行封装，只提供调用端需要的接口。

正文：在很多时候，我们都不希望把一些不必要的操作暴露给调用端，只需要给它所需要的操作或数据就行，那么做法就是封装。这个重构在微软的代码库也经常遇到。比如最经典的属性对字段的封装就是一个很好的例子，那么下面我们将看到对集合的封装，如下代码所示，调用端只需要一个集合的信息，而我们则提供了一个 IList 的集合，大家都知道 IList 具有对集合的所有操作，所以这会带来很多隐患，最好的做法就是对它进行重构。

```
using System.Collections.Generic;
namespace LosTechies.DaysOfRefactoring.EncapsulateCollection.Before
{
    public class Order
    {
        private List<OrderLine> _orderLines;
        private double _orderTotal;
        public IList<OrderLine> OrderLines
        {
            get { return _orderLines; }
        }
        public void AddOrderLine(OrderLine orderLine)
        {
            _orderTotal += orderLine.Total;
            _orderLines.Add(orderLine);
        }
        public void RemoveOrderLine(OrderLine orderLine)
        {
            orderLine = _orderLines.Find(o => o == orderLine);
            if (orderLine == null)
```

```
                return;
            _orderTotal -= orderLine.Total;
            _orderLines.Remove(orderLine);
        }
    }
    public class OrderLine
    {
        public double Total
        { get; private set; }
    }
}
```

那么重构之后，我们把 IList 换成了 IEnumerable，大家都知道只包括一个返回值为 IEnumerator 的 GetEnumerator()方法，所以这样只能遍历取出它的值，而不能对这个集合做出改变，这正是我们所需要的结果，具体代码如下：

```
using System.Collections.Generic;
namespace LosTechies.DaysOfRefactoring.EncapsulateCollection.After
{
    public class Order
    {
        private List<OrderLine> _orderLines;
        private double _orderTotal;
        public IEnumerable<OrderLine> OrderLines
        {
            get { return _orderLines; }
        }
        public void AddOrderLine(OrderLine orderLine)
        {
            _orderTotal += orderLine.Total;
            _orderLines.Add(orderLine);
        }
        public void RemoveOrderLine(OrderLine orderLine)
        {
            orderLine = _orderLines.Find(o => o == orderLine);
            if (orderLine == null)
                return;
            _orderTotal -= orderLine.Total;
            _orderLines.Remove(orderLine);
        }
    }
    public class OrderLine
    {
        public double Total { get; private set; }
```

```
    }
}
```

总结：这个例子很容易让我们想到以前系统间耦合常喜欢用数据库。每个系统都会操作数据库，并且有些系统还会对数据库的表结构或字段进行修改，那么这很容易就会造成维护的地狱，很明智的一个做法就是使用 SOA 来隔开这些耦合，让一些只需要数据展示的系统得到自己需要的数据即可。

3.2.2 移动方法

概念：本文所讲的移动方法就是方法放在合适的位置（通常指放在合适的类中）。

正文：移动方法是一个很简单也很常见的重构，只要是系统就会存在很多类，那么类里面包括很多方法，如果一个方法经常被另外一个类使用（比本身的类使用还多）或者这个方法本身就不应该放在这个类里面，那么这个适合应该考虑把它移到合适的类中。代码如下：

```
namespace LosTechies.DaysOfRefactoring.MoveMethod.Before
{
    public class BankAccount
    {
        public BankAccount(int accountAge, int creditScore, AccountInterest accountInterest)
        {
            AccountAge = accountAge;
            CreditScore = creditScore;
            AccountInterest = accountInterest;
        }

        public int AccountAge { get; private set; }
        public int CreditScore { get; private set; }
        public AccountInterest AccountInterest { get; private set; }

        public double CalculateInterestRate()
        {
            if (CreditScore > 800)
                return 0.02;
            if (AccountAge > 10)
                return 0.03;
            return 0.05;
        }
    }

    public class AccountInterest
```

```
    {
        public BankAccount Account { get; private set; }

        public AccountInterest(BankAccount account)
        {
            Account = account;
        }

        public double InterestRate
        {
            get { return Account.CalculateInterestRate(); }
        }

        public bool IntroductoryRate
        {
            get { return Account.CalculateInterestRate() < 0.05; }
        }
    }
}
```

移动以后大家可以看到 BankAccount 类的职责也单一，同时 CalculateInterestRate 也放到了经常使用且适合它的类中了，所以此重构是一个比较好的重构，能让整个代码变得更加合理。

```
namespace LosTechies.DaysOfRefactoring.MoveMethod.After
{
    public class AccountInterest
    {
        public BankAccount Account { get; private set; }

        public AccountInterest(BankAccount account)
        {
            Account = account;
        }

        public double InterestRate
        {
            get { return CalculateInterestRate(); }
        }

        public bool IntroductoryRate
        {
            get { return CalculateInterestRate() < 0.05; }
        }
```

```
        public double CalculateInterestRate()
        {
            if (Account.CreditScore > 800)
                return 0.02;
            if (Account.AccountAge > 10)
                return 0.03;
            return 0.05;
        }
    }
}
```

```
namespace LosTechies.DaysOfRefactoring.MoveMethod.After
{
    public class BankAccount
    {
        public BankAccount(int accountAge, int creditScore, AccountInterest accountInterest)
        {
            AccountAge = accountAge;
            CreditScore = creditScore;
            AccountInterest = accountInterest;
        }

        public int AccountAge { get; private set; }
        public int CreditScore { get; private set; }
        public AccountInterest AccountInterest { get; private set; }
    }
}
```

总结：这个重构法则在很多时候能让我们把代码组织的结构调整得更合理，同时也能给以后的维护带来方便。

3.2.3 提升方法

概念：提升方法是指将一个很多继承类都要用到的方法提升到基类中。

正文：提升方法是指将一个很多继承类都要用到的方法提升到基类中，这样就能减少代码量，同时让类的结构更清晰。如下代码所示，Turn 方法在子类 Car 和 Motorcycle 都会用到，因为 Vehicle 都会有这个方法，所以我们就会想到把它提到基类中。

```
namespace LosTechies.DaysOfRefactoring.PullUpMethod.Before
{
```

```
    public abstract class Vehicle
    {
        // other methods
    }

    public class Car : Vehicle
    {
        public void Turn(Direction direction)
        {
            // code here
        }
    }

    public class Motorcycle : Vehicle
    {
    }

    public enum Direction    {
        Left,
        Right
    }
}
```

重构后的代码如下，那么现在 Car 和 Motorcycle 都具有 Turn 这个方法，如果这个方法修改也只需要修改基类即可，所以给维护和以后的重构带来了方便。

```
namespace LosTechies.DaysOfRefactoring.PullUpMethod.After
{
    public abstract class Vehicle
    {
        public void Turn(Direction direction)
        {
            // code here
        }
    }

    public class Car : Vehicle
    {
    }

    public class Motorcycle : Vehicle
    {
    }
```

```
    public enum Direction
    {
        Left,
        Right
    }
}
```

总结：这个重构要根据具体情况使用，如果不是每个子类都有这个方法的话，可以考虑使用接口或者其他方式。

3.2.4 降低方法

概念：本文中的降低方法和前篇的提升方法整好相反，也就是把个别子类使用到的方法从基类移到子类里面去。

正文：如下代码所示，Animal 类中的方法 Bark 只有在其子类 Dog 中使用，所以最好的方案就是把这个方法移到子类 Dog 中。

```
namespace LosTechies.DaysOfRefactoring.PushDownMethod.Before
{
    public abstract class Animal
    {
        public void Bark()
        {
            // code to bark
        }
    }

    public class Dog : Animal
    {
    }

    public class Cat : Animal
    {
    }
}
```

重构后的代码如下，同时如果在父类 Animal 中如果没有其他的字段或者公用方法的话，可以考虑把 Bark 方法做成一个接口，从而去掉 Animal 类。

```
namespace LosTechies.DaysOfRefactoring.PushDownMethod.After
{
```

```
    public abstract class Animal
    {
    }

    public class Dog : Animal
    {
        public void Bark()
        {
            // code to bark
        }
    }

    public class Cat : Animal
    {
    }
}
```

总结：面向对象三大特征（继承、封装、多态）很多时候可以帮助我们，但同时也可能会造成使用过度或者使用不当，所以如何把握好设计，这个就变得至关重要。在什么时候使用继承的方式，在什么时候使用组合和聚合，接口和继承类的选择等久成了我们的重点。

3.2.5 提升字段

概念：本文中的提升字段和前面的提升方法颇为相似，就是把子类公用的字段提升到基类中，从而达到公用的目的。

正文：如下代码所示， Account 的两个子类 CheckingAccount 和 SavingsAccount 都有 minimumCheckingBalance 字段，所以可以考虑把这个字段提到基类中。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LosTechies.DaysOfRefactoring.PullUpField.Before
{
    public abstract class Account
    {
    }

    public class CheckingAccount : Account
    {
```

```
        private decimal _minimumCheckingBalance = 5m;
    }

    public class SavingsAccount : Account
    {
        private decimal _minimumSavingsBalance = 5m;
    }
}
```

重构后的代码如下，这样提的前提是这些子类有一个基类或者有很多相似的字段和方法，不然为了一个字段而单独建立一个抽象类是不可取的，所以这个就需要具体权衡。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LosTechies.DaysOfRefactoring.PullUpField.After
{
    public abstract class Account
    {
        protected decimal _minimumBalance = 5m;
    }

    public class CheckingAccount : Account
    {
    }

    public class SavingsAccount : Account
    {
    }
}
```

总结：这个重构的策略比较简单，同时也是比较常用的一些做法，最主要就是要注意权衡是否真的有这个必要，看这样做究竟有没有什么好处（比如只需要改一个地方，维护简便了，同时代码量也更少了等）。

3.2.6 降低字段

概念：本文中的降低字段和前篇的提升字段正好相反，就是把基类中只有某些少数类用到的字段降低到使用它们的子类中。

正文：如下代码所示，基类 Task 类中的_resolution 字段只会在子类 BugTask 中用到，所以就考虑把它放到 BugTask 类中。

```
namespace LosTechies.DaysOfRefactoring.PushDownField.Before
{
    public abstract class Task
    {
        protected string _resolution;
    }

    public class BugTask : Task
    {
    }

    public class FeatureTask : Task
    {
    }
}
```

重构后的代码如下所示，这样做的好处可以简化基类，同时让其他没有使用它的子类也变得更加简单，如果这样的字段比较多的话，使用此重构也能节约一部分内存。

```
namespace LosTechies.DaysOfRefactoring.PushDownField.After
{
    public abstract class Task
    {
    }

    public class BugTask : Task
    {
        private string _resolution;
    }

    public class FeatureTask : Task
    {
    }
}
```

总结：此重构也是一个非常简单的重构，在很多时候我们都会不自觉的使用它。

3.2.7 重命名（方法，类，参数）

概念：本文中的改名（方法，类，参数）是指在写代码的时候对类、方法、参数、委托、事件等等元素取一个有意义的名称。

正文：如下代码所示，加入一个公司建立一个员工的类，类中有一个员工名字的字段和一个按照小时计算员工收入的方法，那么下面代码的取名就显得很难理解了，所以我们会重构名称。

```
namespace LosTechies.DaysOfRefactoring.Rename.Before
{
    public class Person
    {
        public string FN { get; set; }
        public decimal ClcHrlyPR()
        {
            // code to calculate hourly payrate
            return 0m;
        }
    }
}
```

重构后代码如下所示，这样看起来就非常清晰，如果有新进项目组的成员，也会变得很乐意看这个代码。

```
namespace LosTechies.DaysOfRefactoring.Rename.After
{
    // Changed the class name to Employee
    public class Employee
    {
        public string FirstName { get; set; }

        public decimal CalculateHourlyPay()
        {
            // code to calculate hourly payrate
            return 0m;
        }
    }
}
```

总结：此重构经常被广大程序员所忽视，但是带来的隐患是不可估量的，也许老板要修改功能，那我们来看这段没有重构的代码（就算是自己写的，但由于时间和项目多等关系，我们也很难理解了），然后就会变得焦头烂额。相反重构后的代码就会觉得一目了然、赏心悦目。

3.2.8 使用委派代替继承

概念：本文中的“使用委派代替继承”是指在根本没有父子关系的类中使用继承是不合理的，可以用委派的方式来代替。

正文：如下代码所示，Child 和 Sanitation （公共设施）是没有逻辑上的父子关系，因为小孩不可能是一个公共设施吧！所以我们为了完成这个功能可以考虑使用委派的方式。

```
namespace LosTechies.DaysOfRefactoring.ReplaceInheritance.Before
{
    public class Sanitation
    {
        public string WashHands()
        {
            return "Cleaned!";
        }
    }

    public class Child : Sanitation
    {
    }
}
```

重构后的代码如下，把 Sanitation 委派到 Child 类中，从而可以使用 WashHands 这个方法，这种方式我们经常会用到，其实 IOC 也使用到了这个原理，可以通过构造注入和方法注入等。

```
namespace LosTechies.DaysOfRefactoring.ReplaceInheritance.After
{
    public class Sanitation
    {
        public string WashHands()
        {
            return "Cleaned!";
        }
    }

    public class Child
    {
        private Sanitation Sanitation { get; set; }
```

```
        public Child()
        {
            Sanitation = new Sanitation();
        }

        public string WashHands()
        {
            return Sanitation.WashHands();
        }
    }
}
```

总结：这个重构是一个很好的重构，在很大程度上解决了滥用继承的情况，很多设计模式也用到了这种思想（比如桥接模式、适配器模式、策略模式等）。

3.2.9 提取接口

概念：本文中的“提取接口” 是指超过一个的类要使用某一个类中部分方法时，我们应该解开它们之间的依赖，让调用者使用接口，这很容易实现也可以降低代码的耦合性。

正文：如下代码所示，RegistrationProcessor 类只使用到了 ClassRegistration 类中的 Create 方法和 Total 字段，所以可以考虑把他们做成接口给 RegistrationProcessor 调用。

```
namespace LosTechies.DaysOfRefactoring.ExtractInterface.Before
{
    public class ClassRegistration
    {
        public void Create()
        {
            // create registration code
        }

        public void Transfer()
        {
            // class transfer code
        }

        public decimal Total { get; private set; }
    }

    public class RegistrationProcessor
    {
```

```
        public decimal ProcessRegistration(ClassRegistration registration)
        {
            registration.Create();
            return registration.Total;
        }
    }
}
```

重构后的代码如下，我们提取了一个 IClassRegistration 接口，同时让 ClassRegistration 继承此接口，然后调用端 RegistrationProcessor 就可以直接通过 IClassRegistration 接口进行调用。

```
namespace LosTechies.DaysOfRefactoring.ExtractInterface.After
{
    public interface IClassRegistration
    {
        void Create();
        decimal Total { get; }
    }

    public class ClassRegistration : IClassRegistration
    {
        public void Create()
        {
            // create registration code
        }

        public void Transfer()
        {
            // class transfer code
        }

        public decimal Total { get; private set; }
    }

    public class RegistrationProcessor
    {
        public decimal ProcessRegistration(IClassRegistration registration)
        {
            registration.Create();
            return registration.Total;
        }
    }
}
```

总结：这个重构策略也是一个常见的运用，很多设计模式也会在其中运用此思想（如简单工程、抽象工厂等都会通过接口来解开依赖）。

3.2.10 提取方法

概念：本文中的把某些计算复杂的过程按照功能提取成各个小方法，这样就可以使代码的可读性、维护性得到提高。

正文：如下代码所示，CalculateGrandTotal 方法里面包含了多个逻辑，第一计算 subTotal 的总和，第二 subTotal 要循环减去 discount，也就是计算 Discounts，第三就是计算 Tax。所以我们可以根据功能把他们拆分成三个小方法。

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.ExtractMethod.Before
{
    public class Receipt
    {
        private IList<decimal> Discounts { get; set; }
        private IList<decimal> ItemTotals { get; set; }

        public decimal CalculateGrandTotal()
        {
            decimal subTotal = 0m;
            foreach (decimal itemTotal in ItemTotals)
                subTotal += itemTotal;

            if (Discounts.Count > 0)
            {
                foreach (decimal discount in Discounts)
                    subTotal -= discount;
            }

            decimal tax = subTotal * 0.065m;

            subTotal += tax;

            return subTotal;
        }
    }
}
```

重构后的代码如下，然后 CalculateGrandTotal 方法就直接调用 CalculateSubTotal、CalculateDiscounts、CalculateTax，从而是整个逻辑看起来更加清晰，并且可读性和维护性也得到了大大提高。

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.ExtractMethod.After
{
    public class Receipt
    {
        private IList<decimal> Discounts { get; set; }
        private IList<decimal> ItemTotals { get; set; }

        public decimal CalculateGrandTotal()
        {
            decimal subTotal = CalculateSubTotal();

            subTotal = CalculateDiscounts(subTotal);

            subTotal = CalculateTax(subTotal);

            return subTotal;
        }

        private decimal CalculateTax(decimal subTotal)
        {
            decimal tax = subTotal * 0.065m;

            subTotal += tax;
            return subTotal;
        }

        private decimal CalculateDiscounts(decimal subTotal)
        {
            if (Discounts.Count > 0)
            {
                foreach (decimal discount in Discounts)
                    subTotal -= discount;
            }
            return subTotal;
        }

        private decimal CalculateSubTotal()
```

```
        {
            decimal subTotal = 0m;
            foreach (decimal itemTotal in ItemTotals)
                subTotal += itemTotal;
            return subTotal;
        }
    }
}
```

总结：这个重构在很多公司都有一些的代码规范作为参考，比如一个类不能超过多少行代码，一个方法里面不能超过多少行代码，这在一定程度上也能使程序员把这些复杂的逻辑剥离成意义很清楚的小方法。

3.2.11 使用策略类

概念：本文中的“使用策略类” 是指用设计模式中的策略模式来替换原来的 switch case 和 if else 语句，这样可以解开耦合，同时也使维护性和系统的可扩展性大大增强。

正文：如下面代码所示，ClientCode 类会更加枚举 State 的值来调用 ShippingInfo 的不同方法，但是这样就会产生很多的判断语句，如果代码量加大，类变得很大了的话，维护中改动也会变得很大，每次改动一个地方，都要对整个结构进行编译（假如是多个工程），所以我们想到了对它进行重构，剥开耦合。

```
namespace LosTechies.DaysOfRefactoring.SwitchToStrategy.Before
{
    public class ClientCode
    {
        public decimal CalculateShipping()
        {
            ShippingInfo shippingInfo = new ShippingInfo();
            return shippingInfo.CalculateShippingAmount(State.Alaska);
        }
    }

    public enum State
    {
        Alaska,
        NewYork,
        Florida
    }

    public class ShippingInfo
```

```
    {
        public decimal CalculateShippingAmount(State shipToState)
        {
            switch (shipToState)
            {
                case State.Alaska:
                    return GetAlaskaShippingAmount();
                case State.NewYork:
                    return GetNewYorkShippingAmount();
                case State.Florida:
                    return GetFloridaShippingAmount();
                default:
                    return 0m;
            }
        }

        private decimal GetAlaskaShippingAmount()
        {
            return 15m;
        }

        private decimal GetNewYorkShippingAmount()
        {
            return 10m;
        }

        private decimal GetFloridaShippingAmount()
        {
            return 3m;
        }
    }
}
```

重构后的代码如下所示，抽象出一个 IShippingCalculation 接口，然后把 ShippingInfo 类里面的 GetAlaskaShippingAmount、GetNewYorkShippingAmount、GetFloridaShippingAmount 三个方法分别提炼成三个类，然后继承自 IShippingCalculation 接口，这样在调用的时候就可以通过 IEnumerable<IShippingCalculation> 来解除之前的 switch case 语句，这和 IOC 的做法颇为相似。

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LosTechies.DaysOfRefactoring.SwitchToStrategy.After_WithIoC
```

```
{
    public interface IShippingInfo
    {
        decimal CalculateShippingAmount(State state);
    }

    public class ClientCode
    {
        [Inject]
        public IShippingInfo ShippingInfo { get; set; }

        public decimal CalculateShipping()
        {
            return ShippingInfo.CalculateShippingAmount(State.Alaska);
        }
    }

    public enum State
    {
        Alaska,
        NewYork,
        Florida
    }

    public class ShippingInfo : IShippingInfo
    {
        private IDictionary<State, IShippingCalculation> ShippingCalculations { get; set; }

        public ShippingInfo(IEnumerable<IShippingCalculation> shippingCalculations)
        {
            ShippingCalculations = shippingCalculations.ToDictionary(calc => calc.State);
        }

        public decimal CalculateShippingAmount(State shipToState)
        {
            return ShippingCalculations[shipToState].Calculate();
        }
    }

    public interface IShippingCalculation
    {
        State State { get; }
        decimal Calculate();
    }
```

```
    public class AlaskShippingCalculation : IShippingCalculation
    {
        public State State { get { return State.Alaska; } }

        public decimal Calculate()
        {
            return 15m;
        }
    }

    public class NewYorkShippingCalculation : IShippingCalculation
    {
        public State State { get { return State.NewYork; } }

        public decimal Calculate()
        {
            return 10m;
        }
    }

    public class FloridaShippingCalculation : IShippingCalculation
    {
        public State State { get { return State.Florida; } }

        public decimal Calculate()
        {
            return 3m;
        }
    }
}
```

总结：这种重构在设计模式当中把它单独取了一个名字——策略模式，这样做的好处就是可以隔开耦合，以注入的形式实现功能，这使增加功能变得更加容易和简便，同样也增强了整个系统的稳定性和健壮性。

3.2.12 分解依赖

概念：本文中的“分解依赖” 是指对部分不满足我们要求的类和方法进行依赖分解，通过装饰器来达到我们需要的功能。

正文：正如下面代码所示，如果你要在你的代码中加入单元测试但有一部分代码是你不想测

试的，那么你应用使用这个的重构。下面的例子中我们应用静态类来完成某些工作，但问题是在单元测试时我们无法 mock 静态类，所以我们只能引入静态类的装饰接口来分解对静态类的依赖。从而我们使我们的调用类只需要依赖于装饰接口就能完成这个操作。

```
namespace LosTechies.DaysOfRefactoring.BreakDependencies.Before
{
    public class AnimalFeedingService
    {
        private bool FoodBowlEmpty { get; set; }

        public void Feed()
        {
            if (FoodBowlEmpty)
                Feeder.ReplenishFood();
            // more code to feed the animal
        }
    }

    public static class Feeder
    {
        public static void ReplenishFood()
        {
            // fill up bowl
        }
    }
}
```

重构后代码如下，我们添加一个接口和一个实现类，在实现类中调用静态类的方法，所以说具体做什么事情没有改变，改变的只是形式，但这样做的一个好处是增加了了代码的可测试性。在应用了分解依赖模式后，我们就可以在单元测试的时候 mock 一个 IFeederService 对象并通过 AnimalFeedingService 的构造函数传递给它。这样就可以完成我们需要的功能。

```
namespace LosTechies.DaysOfRefactoring.BreakDependencies.After
{
    public class AnimalFeedingService
    {
        public IFeederService FeederService { get; set; }

        public AnimalFeedingService(IFeederService feederService)
        {
            FeederService = feederService;
        }

        private bool FoodBowlEmpty { get; set; }
```

```
        public void Feed()
        {
            if (FoodBowlEmpty)
                FeederService.ReplenishFood();

            // more code to feed the animal
        }
    }

    public interface IFeederService
    {
        void ReplenishFood();
    }

    public class FeederService : IFeederService
    {
        public void ReplenishFood()
        {
            Feeder.ReplenishFood();
        }
    }

    public static class Feeder
    {
        public static void ReplenishFood()
        {
            // fill up bowl
        }
    }
}
```

总结：这个重构在很多时候和设计模式中的一些思想类似，使用中间的装饰接口来分解两个类之间的依赖，对类进行装饰，然后使它满足我们所需要的功能。

3.2.13 提取方法对象

概念：本文中的“提取方法对象”是指当你发现一个方法中存在过多的局部变量时，你可以通过使用“提取方法对象”重构来引入一些方法，每个方法完成任务的一个步骤，这样可以使得程序变得更具有可读性。

正文：如下代码所示，Order 类中的 Calculate 方法要完成很多功能，在之前我们用“提取

方法”来进行重构，现在我们采取“提取方法对象”来完成重构。

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.ExtractMethodObject.Before
{
    public class OrderLineItem
    {
        public decimal Price { get; private set; }
    }

    public class Order
    {
        private IList<OrderLineItem> OrderLineItems { get; set; }
        private IList<decimal> Discounts { get; set; }
        private decimal Tax { get; set; }

        public decimal Calculate()
        {
            decimal subTotal = 0m;

            // Total up line items
            foreach (OrderLineItem lineItem in OrderLineItems)
            {
                subTotal += lineItem.Price;
            }

            // Subtract Discounts
            foreach (decimal discount in Discounts)
                subTotal -= discount;

            // Calculate Tax
            decimal tax = subTotal * Tax;

            // Calculate GrandTotal
            decimal grandTotal = subTotal + tax;

            return grandTotal;
        }
    }
}
```

正如下代码所示，我们引入了 OrderCalculator 类，该类实现了所有的计算方法，Order 类将自身传递给 OrderCalculator 类并调用 Calculate 方法完成计算过程。

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.ExtractMethodObject.After
{
    public class OrderLineItem
    {
        public decimal Price { get; private set; }
    }

    public class Order
    {
        public IEnumerable<OrderLineItem> OrderLineItems { get; private set; }
        public IEnumerable<decimal> Discounts { get; private set; }
        public decimal Tax { get; private set; }

        public decimal Calculate()
        {
            return new OrderCalculator(this).Calculate();
        }
    }

    public class OrderCalculator
    {
        private decimal SubTotal { get; set; }
        private IEnumerable<OrderLineItem> OrderLineItems { get; set; }
        private IEnumerable<decimal> Discounts { get; set; }
        private decimal Tax { get; set; }

        public OrderCalculator(Order order)
        {
            OrderLineItems = order.OrderLineItems;
            Discounts = order.Discounts;
            Tax = order.Tax;
        }

        public decimal Calculate()
        {
            CalculateSubTotal();

            SubtractDiscounts();

            CalculateTax();
```

```
            return SubTotal;
        }

        private void CalculateSubTotal()
        {
            // Total up line items
            foreach (OrderLineItem lineItem in OrderLineItems)
                SubTotal += lineItem.Price;
        }

        private void SubtractDiscounts()
        {
            // Subtract Discounts
            foreach (decimal discount in Discounts)
                SubTotal -= discount;
        }

        private void CalculateTax()
        {
            // Calculate Tax
            SubTotal += SubTotal * Tax;
        }
    }
}
```

总结：本文的重构方法在有的时候还是比较有用，但这样会造成字段的增加，同时也会带来一些维护的不便，它和“提取方法”最大的区别就是一个通过方法返回需要的数据，另一个则是通过字段来存储方法的结果值，所以在很大程度上我们都会选择“提取方法”。

3.2.14 分离职责

概念：本文中的“分离职责”是指当一个类有许多职责时，将部分职责分离到独立的类中，这样也符合面向对象的五大特征之一的单一职责原则，同时也可以使代码的结构更加清晰，维护性更高。

正文：如下代码所示，Video 类有两个职责，一个是处理 video rental，另一个是计算每个客户的总租金。我们可以将这两个职责分离出来，因为计算每个客户的总租金可以在 Customer 计算，这也比较符合常理。

```
using System.Collections.Generic;
using System.Linq;
```

```
namespace LosTechies.DaysOfRefactoring.BreakResponsibilities.Before
{
    public class Video
    {
        public void PayFee(decimal fee)
        {
        }

        public void RentVideo(Video video, Customer customer)
        {
            customer.Videos.Add(video);
        }

        public decimal CalculateBalance(Customer customer)
        {
            returncustomer.LateFees.Sum();
        }
    }

    public class Customer
    {
        public IList<decimal> LateFees { get; set; }
        public IList<Video> Videos { get; set; }
    }
}
```

重构后的代码如下，这样 Video 的职责就变得很清晰，同时也使代码维护性更好。

```
using System.Collections.Generic;
using System.Linq;

namespace LosTechies.DaysOfRefactoring.BreakResponsibilities.After
{
    public class Video
    {
        public void RentVideo(Video video, Customer customer)
        {
            customer.Videos.Add(video);
        }
    }

    public class Customer
    {
        public IList<decimal> LateFees { get; set; }
```

```
        public IList<Video> Videos { get; set; }

        public void PayFee(decimal fee)
        {
        }

        public decimal CalculateBalance(Customer customer)
        {
            return customer.LateFees.Sum();
        }
    }
}
```

总结：这个重构经常会用到，它和之前的“移动方法”有几分相似之处，让方法放在合适的类中，并且简化类的职责，同时这也是面向对象五大原则之一和设计模式中的重要思想。

3.2.15 移除重复内容

概念：本文中的“移除重复内容”是指把一些很多地方都用到的逻辑提炼出来，然后提供给调用者统一调用。

正文：如下代码所示，ArchiveRecord 和 CloseRecord 都会用到 Archived = true; 和 DateArchived = DateTime.Now; 这两条语句，所以我们就可以对它进行重构。

```
using System;

namespace LosTechies.DaysOfRefactoring.RemoveDuplication.Before
{
    public class MedicalRecord
    {
        public DateTime DateArchived { get; private set; }
        public bool Archived { get; private set; }

        public void ArchiveRecord()
        {
            Archived = true;
            DateArchived = DateTime.Now;
        }

        public void CloseRecord()
        {
            Archived = true;
```

```
            DateArchived = DateTime.Now;
        }
    }
}
```

重构后的代码如下所示，我们提炼了 SwitchToArchived 方法来封装公用的操作，然后给 ArchiveRecord 和 CloseRecord 统一调用。

```
using System;

namespace LosTechies.DaysOfRefactoring.RemoveDuplication.After
{
    public class MedicalRecord
    {
        public DateTime DateArchived { get; private set; }
        public bool Archived { get; private set; }

        public void ArchiveRecord()
        {
            SwitchToArchived();
        }

        public void CloseRecord()
        {
            SwitchToArchived();
        }

        private void SwitchToArchived()
        {
            Archived = true;
            DateArchived = DateTime.Now;
        }
    }
}
```

总结：这个重构很简单，绝大多数程序员都会使用这种重构方法，但有时由于习惯、时间、赶进度等原因而忽略它，所以会使得整个系统杂乱无章，到处都是 Ctrl+C 和 Ctrl+V 的痕迹。

3.2.16 封装条件

概念：本文中的“封装条件”是指条件关系比较复杂时，代码的可读性会比较差，所以这时我们应当根据条件表达式是否需要参数将条件表达式提取成可读性更好的属性或者方法，如

果条件表达式不需要参数则可以提取成属性，如果条件表达式需要参数则可以提取成方法。

正文：如下代码所示，PerformCoolFunction 里面的 if 条件判断比较复杂，看起来有点杂乱，所以就把它提出来。

```
using System;namespace
LosTechies.DaysOfRefactoring.EncapsulateConditional.Before
{
    public class RemoteControl
    {
        private string[] Functions { get; set; }

        private string Name { get; set; }

        private int CreatedYear { get; set; }

        public string PerformCoolFunction(string buttonPressed)
        {
            // Determine if we are controlling some extra function
            // that requires special conditions
            if (Functions.Length > 1 && Name == "RCA" && CreatedYear > DateTime.Now.Year - 2)
                return "doSomething";
        }
    }
}
```

如下代码所示，我们把条件表达式封装成 HasExtraFunctions 属性，这样先前的条件判断就成了 if (HasExtraFunctions) ，所以这样就在很大程度上提高了可读性。

```
using System;
namespace LosTechies.DaysOfRefactoring.EncapsulateConditional.After
{
    public class RemoteControl
    {
        private string[] Functions { get; set; }

        private string Name { get; set; }

        private int CreatedYear { get; set; }

        private bool HasExtraFunctions
        {
            get { return Functions.Length > 1 && Name == "RCA" && CreatedYear > DateTime.Now.Year - 2; }
        }
```

```
        public string PerformCoolFunction(string buttonPressed)
        {
            // Determine if we are controlling some extra function
            // that requires special conditions
            if (HasExtraFunctions)
                return "doSomething";
        }
    }
}
```

总结：这个重构在很大程度上能改善代码的可读性，尤其是在一个逻辑很复杂的应用中，把这些条件判断封装成一个有意义的名字，这样很复杂的逻辑也会立刻变得简单起来。

3.2.17 提取父类

概念：本文中的“提取父类”是指类中有一些字段或方法，你想把它们提取到父类中以便同一继承层次的其它类也可以访问他们，这个和之前的很多重构有异曲同工之处。

正文：Dog 类中的 EatFood 和 Groom 有可能被其他类用到，因为他们都是动物的一些公有性质，所以这个时候我们就会考虑对它进行提炼。

```
namespace LosTechies.DaysOfRefactoring.ExtractSuperclass.Before
{
    public class Dog
    {
        public void EatFood()
        {
            // eat some food
        }

        public void Groom()
        {
            // perform grooming
        }
    }
}
```

代码如下所示，提取了 Animal 方法来封装公用的 EatFood 和 Groom 类，从而使其他继承了 Animal 类的子类都可以使用这两个方法了。

```
namespace LosTechies.DaysOfRefactoring.ExtractSuperclass.After
```

```
{
    public class Animal
    {
        public void EatFood()
        {
            // eat some food
        }

        public void Groom()
        {
            // perform grooming
        }
    }

    public class Dog : Animal
    {
    }
}
```

总结：这个重构是典型的继承用法，很多程序员都会选择这样做，但是要注意正确的使用，不要造成过度使用了继承，如果过度使用了，请考虑用接口、组合和聚合来实现。

3.2.18 使用条件判断代替异常

概念：本文中的“使用条件判断代替异常”是指把没有必要使用异常做判断的条件尽量改为条件判断。

正文：如下代码所示，在日常的编码中我们经常需要用到异常来控制程序流，Start 方法里面用 try　catch 做条件判断，我们知道这里没有必要使用这种方式，因为你不需要做类型不可控的类型转换，也不需要处理异常行为，所以我们应该对它进行重构。

```
namespace LosTechies.DaysOfRefactoring.ReplaceException.Before
{
    public class Microwave
    {
        private IMicrowaveMotor Motor { get; set; }

        public bool Start(object food)
        {
            bool foodCooked = false;
            try
            {
```

```
                Motor.Cook(food);
                foodCooked = true;
            }
            catch (InUseException)
            {
                foodcooked = false;
            }
            return foodCooked;
        }
    }
}
```

重构后的代码如下所示，try　catch 做条件判断的语句改成了 if　return 的方式，这样在很多程度上统一了代码的书写，同时也提高了性能。

```
namespace LosTechies.DaysOfRefactoring.ReplaceException.After
{
    public class Microwave
    {
        private IMicrowaveMotor Motor { get; set; }

        public bool Start(object food)
        {
            if (Motor.IsInUse)
                return false;

            Motor.Cook(food);
            return true;
        }
    }
}
```

总结：　这个重构在项目代码中也经常用到，因为对于一部分程序员，是很难把握什么时候用 try　catch ，什么地方该用 try　catch 。记得之前大家还专门讨论过这些，比如如何用好以及在大中型项目中应该把它放在哪一个组件中等。

3.2.19 提取工厂类

概念：本文中的“提取工厂类”是指如果要创建的对象很多，则代码会变的很复杂。一种很好的方法就是提取工厂类。

正文：一般来说我们需要在代码中设置一些对象，以便获得它们的状态，从而使用对象，所

谓的设置通常来说就是创建对象的实例并调用对象的方法。有时如果要创建的对象很多，则代码会变的很复杂。这便是工厂模式发挥作用的情形。工厂模式的复杂应用是使用抽象工厂创建对象集，但我们在这里只是使用基本的工厂类创建对象的一个简单应用。

```
namespace LosTechies.DaysOfRefactoring.ExtractServiceClass.Before
{
    public class PoliceCarController
    {
        public PoliceCar New(int mileage, bool serviceRequired)
        {
            PoliceCar policeCar = new PoliceCar();
            policeCar.ServiceRequired = serviceRequired;
            policeCar.Mileage = mileage;

            return policeCar;
        }
    }
}
```

那么重构后的代码如下，New 方法变得很简单了，指需要调用实现接 IPoliceCarFactory 接口的 PoliceCarFactory 类就可以返回对象，这样就隔开了创建对象的逻辑，如果需求现在变为根据不同的条件创建不同的对象，什么时候创建对象等都变成了比较简单的事情，在后期可以把对象都配置在 XML 里面，使用反射的方式实现 IOC 注入创建。

```
namespace LosTechies.DaysOfRefactoring.ExtractServiceClass.After
{
    public interface IPoliceCarFactory
    {
        PoliceCar Create(int mileage, bool serviceRequired);
    }

    public class PoliceCarFactory : IPoliceCarFactory
    {
        public PoliceCar Create(int mileage, bool serviceRequired)
        {
            PoliceCar policeCar = new PoliceCar();
            policeCar.ReadForService = serviceRequired;
            policeCar.Mileage = mileage; return policeCar;
        }
    }

    public class PoliceCarController
    {
        public IPoliceCarFactory PoliceCarFactory { get; set; }
```

```
        public PoliceCarController(IPoliceCarFactory policeCarFactory)
        {
            PoliceCarFactory = policeCarFactory;
        }

        public PoliceCar New(int mileage, bool serviceRequired)
        {
            return PoliceCarFactory.Create(mileage, serviceRequired);
        }
    }
}
```

总结：这个重构经常会在项目中使用，如果要创建的对象是一个，你可以采用简单工厂，但是这种方式还是会存在很多依赖，维护起来也比较不方便。所以推荐使用工厂方法模式，把实例化延迟到子类。如果你要创建一系列的对象，那么就推荐你使用抽象工厂模式，但是要注意不要过度设计，只要能满足不断变化的需求和给以后的维护和重构带来方便即可。

3.2.20 提取子类

概念：本文中的”提取子类”是指把基类中的一些不是所有子类都需要访问的方法调整到子类中。

正文：当你的基类中存在一些方法不是所有的子类都需要访问，你想将它们调整到子类中时，这个重构会变得很有用了。如下代码所示，我们需要一个 Registration 类用来处理学生选课的信息。但是当 Registration 类开始工作后，我们意识到我们会在两种不同的上下文中使用 Registration 类，NonRegistrationAction 和 Notes 只有在我们处理未注册情况下才用到。

所以我们将 NonRegistration 和 Notes 提到单独的 NonRegistration 类中。

```
using System;
namespace LosTechies.DaysOfRefactoring.SampleCode.ExtractSubclass.Before
{
    public class Registration
    {
        public NonRegistrationAction Action { get; set; }

        public decimal RegistrationTotal { get; set; }

        public string Notes { get; set; }

        public string Description { get; set; }
```

```
        public DateTime RegistrationDate { get; set; }
    }
}
```

重构后的代码如下所示，这样也满足面向对象五大原则之一的单一职责。同时也让类的结构变得更加清晰，增强了可维护性。

```
using System;
namespace LosTechies.DaysOfRefactoring.SampleCode.ExtractSubclass.After
{
    public class Registration
    {
        public decimal RegistrationTotal { get; set; }

        public string Description { get; set; }

        public DateTime RegistrationDate { get; set; }

    }

    public class NonRegistration : Registration
    {
        public NonRegistrationAction Action { get; set; }

        public string Notes { get; set; }
    }
}
```

总结：这个重构方法经常用来规范类的职责，和之前的一些重构方法也有些类似。

3.2.21 合并继承

概念：本文中的”合并继承”是指如果子类的属性和方法也适合于基类，那么就可以移除子类，从而减少依赖关系。

正文：上一篇我们讲到“提取子类”重构是指当基类中的一个责任不被所有的子类所需要时，将这些责任提取到合适的子类中。而我们今天所要讲的的“合并继承”重构一般用在当我们觉得不需要子类的时候。

如下代码所示，StudentWebSite 子类除了有一个属性用来说明网站是否是活动的外没有别的责任，在这种情形下我们意识到 IsActive 属性可以应用到所有的网站，所以我们可以将

IsActive 属性上移到基类中，并去掉 StudentWebSite 类。

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.SampleCode.CollapseHierarchy.Before
{
    public class Website
    {
        public string Title { get; set; }
        public string Description { get; set; }
        public IEnumerable<Webpage> Pages { get; set; }
    }

    public class StudentWebsite : Website
    {
        public bool IsActive { get; set; }
    }
}
```

重构后的代码如下：

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.SampleCode.CollapseHierarchy.After
{
    public class Website
    {
        public string Title { get; set; }
        public string Description { get; set; }
        public IEnumerable<Webpage> Pages { get; set; }
        public bool IsActive { get; set; }
    }
}
```

总结： 这篇和上篇其实最主要论述了子类和父类的继承关系以及如何判断什么时候需要使用继承，一般我们都能处理好这些关系，所以相对比较简单。

3.2.22 分解方法

概念：本文中的”分解方法”是指把我们所做的这个功能不停的分解方法，直到将一个大方法分解为名字有意义且可读性更好的若干个小方法。

正文：如下代码所示，因为现实中 AcceptPayment 方法不会做这么多的事情。，所以我们通过几次分解将 AcceptPayment 拆分成若干个名字有意义且可读性更好的小方法。

```
using System.Collections.Generic;
using System.Linq;
namespace LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.Before
{
    public class CashRegister
    {
        public CashRegister()
        {
            Tax = 0.06m;
        }

        private decimal Tax { get; set; }

        public void AcceptPayment(Customer customer, IEnumerable<Product> products, decimal payment)
        {
            decimal subTotal = 0m;
            foreach (Product product in products)
            {
                subTotal += product.Price;
            }

            foreach (Product product in products)
            {
                subTotal -= product.AvailableDiscounts;
            }
            decimal grandTotal = subTotal * Tax;
            customer.DeductFromAccountBalance(grandTotal);
        }
    }

    public class Customer
    {
        public void DeductFromAccountBalance(decimal amount)
        {
            // deduct from balance
        }
    }
    public class Product
    {
        public decimal Price { get; set; }
```

```
        public decimal AvailableDiscounts { get; set; }
    }
}
```

重构后的代码如下，我们把 AcceptPayment 的内部逻辑拆分成了 CalculateSubtotal、SubtractDiscounts、AddTax、SubtractFromCustomerBalance 四个功能明确且可读性更好的小方法。

```
using System.Collections.Generic;
namespace LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After
{
    public class CashRegister
    {
        public CashRegister()
        {
            Tax = 0.06m;
        }

        private decimal Tax { get; set; }

        private IEnumerable<Product> Products { get; set; }

        public void AcceptPayment(Customer customer, IEnumerable<Product> products, decimal payment)
        {
            decimal subTotal = CalculateSubtotal();
            subTotal = SubtractDiscounts(subTotal);
            decimal grandTotal = AddTax(subTotal);
            SubtractFromCustomerBalance(customer, grandTotal);
        }

        private void SubtractFromCustomerBalance(Customer customer, decimal grandTotal)
        {
            customer.DeductFromAccountBalance(grandTotal);
        }

        private decimal AddTax(decimal subTotal)
        {
            return subTotal * Tax;
        }

        private decimal SubtractDiscounts(decimal subTotal)
        {
            foreach (Product product in Products)
            {
```

```
                subTotal -= product.AvailableDiscounts;
            }
            return subTotal;
        }

        private decimal CalculateSubtotal()
        {
            decimal subTotal = 0m;
            foreach (Product product in Products)
            {
                subTotal += product.Price;
            }

            return subTotal;
        }
    }

    public class Customer
    {
        public void DeductFromAccountBalance(decimal amount)
        {
            // deduct from balance
        }
    }

    public class Product
    {
        public decimal Price { get; set; }

        public decimal AvailableDiscounts { get; set; }
    }
}
```

总结：其实这个重构和我们前面讲的“提取方法”和“提取方法对象”如出一辙，尤其是“提取方法”，所以大家只要知道用这种思想重构就行。

3.2.23 引入参数对象

概念：本文中的“引入参数对象”是指当一个方法的参数过多或者过为复杂时，可以考虑把这些参数封装成一个单独的类。

正文：如果一个方法所需要的参数大于 5 个，理解该方法的签名就变得比较困难，因为这样

感觉参数很长、样式不好并且没有分类，所以我们有必要把参数进行封装。

```
namespace LosTechies.DaysOfRefactoring.SampleCode.ParameterObject.Before
{
    public class Registration
    {
        public void Create(decimal amount, Student student, IEnumerable<Course> courses, decimal credits)
        {
            // do work
        }
    }
}
```

通常这种情形下创建一个用户传递参数的类是很有帮助的，这会使得代码更容易明白也更灵活，因为当你需要增加参数时，只需要给参数类添加一个属性即可。请注意只有当你发现方法的参数比较多时才应该应用该重构，如果方法的参数比较少，就没有必要应用此重构，因为该重构会增加系统中类的数量，同时也会加大维护负担。所以要看参数情况而定。
重构后的代码如下：

```
using System.Collections.Generic;
namespace LosTechies.DaysOfRefactoring.SampleCode.ParameterObject.After
{
    public class RegistrationContext
    {
        public decimal Amount { get; set; }

        public Student Student { get; set; }

        public IEnumerable<Course> Courses { get; set; }

        public decimal Credits { get; set; }
    }

    public class Registration
    {
        public void Create(RegistrationContext registrationContext)
        {
            // do work
        }
    }
}
```

总结：这种重构很重要，尤其是当一个方法的参数比较多的时候，不管是大中型项目还是小型项目，都会遇到这种场景，所以建议大家多使用这个重构。这种封装的思想在 SOA 里面

也经常运用到，封装输入 Message，封装输出 Message，消息来和消息去以及消息间的交互就构成了整个应用体系。

3.2.24 分解复杂判断

概念：本文中的”分解复杂判断”是指把原来复杂的条件判断等语句用尽快返回等方式简化代码。

正文：简单的来说，当你的代码中有很深的嵌套条件时，花括号就会在代码中形成一个长长的箭头。我们经常在不同的代码中看到这种情况，并且这种情况也会扰乱代码的可读性。

如下代码所示，HasAccess 方法里面包含一些嵌套条件，如果再加一些条件或者增加复杂度，那么代码就很可能出现几个问题：1，可读性差。 2，很容易出现异常。 3，性能较差。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace LosTechies.DaysOfRefactoring.SampleCode.ArrowheadAntipattern.Before
{
    public class Security
    {
        public ISecurityChecker SecurityChecker { get; set; }

        public Security(ISecurityChecker securityChecker) { SecurityChecker = securityChecker; }

        public bool HasAccess(User user, Permission permission, IEnumerable<Permission> exemptions)
        {
            bool hasPermission = false;
            if (user != null)
            {
                if (permission != null)
                {
                    if (exemptions.Count() == 0)
                    {
                        if (SecurityChecker.CheckPermission(user, permission)
                            || exemptions.Contains(permission))
                        {
                            hasPermission = true;
                        }
                    }
```

```
                }
            }
            return hasPermission;
        }
    }
}
```

那么重构上面的代码也很简单，如果有可能的话，尽量将条件从方法中移除，我们让代码在做处理任务之前先检查条件，如果条件不满足就尽快返回，不继续执行。下面是重构后的代码：

```
using System.Collections.Generic;
using System.Linq;
namespace LosTechies.DaysOfRefactoring.SampleCode.ArrowheadAntipattern.After
{
    public class Security
    {
        public ISecurityChecker SecurityChecker { get; set; }

        public Security(ISecurityChecker securityChecker) { SecurityChecker = securityChecker; }

        public bool HasAccess(User user, Permission permission, IEnumerable<Permission> exemptions)
        {
            if (user == null || permission == null)
                return false;

            if (exemptions.Contains(permission))
                return true;

            return SecurityChecker.CheckPermission(user, permission);
        }
    }
}
```

总结：这个重构很重要，它和后面讲的”尽快返回“有些类似，我们在做复杂的处理过程时，要经常考虑这个重构，用好了它，会对我们的帮助很大。

3.2.25 引入契约式设计

概念：本文中的”引入契约式设计”是指我们应该对应该对输入和输出进行验证，以确保系统不会出现我们所想象不到的异常和得不到我们想要的结果。

正文：契约式设计规定方法应该对输入和输出进行验证，这样你便可以保证你得到的数据是可以工作的，一切都是按预期进行的，如果不是按预期进行，异常或是错误就应该被返回，下面我们举的例子中，我们方法中的参数可能会值为 null 的情况，在这种情况下由于我们没有验证，NullReferenceException 异常会报出。另外在方法的结尾处我们也没有保证会返回一个正确的 decimal 值给调用方法的对象。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LosTechies.DaysOfRefactoring.SampleCode.Day25_DesignByContract
{
    public class CashRegister
    {
        public decimal TotalOrder(IEnumerable<Product> products, Customer customer)
        {
            decimal orderTotal = products.Sum(product => product.Price);

            customer.Balance += orderTotal;

            return orderTotal;
        }
    }
}
```

对上面的代码重构是很简单的，首先我们处理不会有一个 null 值的 customer 对象，检查我们最少会有一个 product 对象。在返回订单总和之前先确保我们会返回一个有意义的值。如果上面说的检查有任何一个失败，我们就抛出对应的异常，并在异常里说明错误的详细信息，而不是直接抛出 NullReferenceException。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Contracts;

namespace LosTechies.DaysOfRefactoring.SampleCode.DesignByContract.After
{
    public class CashRegister
    {
        public decimal TotalOrder(IEnumerable<Product> products, Customer customer)
        {
            if (customer == null)
```

```
                throw new ArgumentNullException("customer", "Customer cannot be null");
            if (products.Count() == 0)
                throw new ArgumentException("Must have at least one product to total", "products");

            decimal orderTotal = products.Sum(product => product.Price);

            customer.Balance += orderTotal;

            if (orderTotal == 0)
                throw new ArgumentOutOfRangeException("orderTotal", "Order Total should not be zero");

            return orderTotal;
        }
    }
}
```

上面的代码中添加了额外的代码来进行验证，虽然看起来代码复杂度增加了，但我认为这是非常值得做的，因为当 NullReferenceException 发生时去追查异常的详细信息真是很令人讨厌的事情。

总结：微软在处理代码乃至产品的时候，很喜欢应用此重构，你如果认真看它的代码库，认真看一下 WCF 的设计，就不难发现了。这个重构建议大家经常使用，这会增强整个系统的稳定性和健壮性。

3.2.26 避免双重否定

概念：本文中的”避免双重否定”是指把代码中的双重否定语句修改成简单的肯定语句，这样即让代码可读，同时也给维护带来了方便。

正文：避免双重否定重构本身非常容易实现，但我们却在太多的代码中见过因为双重否定降低了代码的可读性以致于非常让人容易误解真正意图。存在双重否定的代码具有非常大的危害性，因为这种类型的代码容易引起错误的假设，错误的假设又会导致书写出错误的维护代码，最终会导致 bug 产生。具体可以看下面的代码：

```
using System.Collections.Generic;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.DoubleNegative.Before
{
    public class Order
    {
        public void Checkout(IEnumerable<Product> products, Customer customer)
```

```
        {
            if  (!customer.IsNotFlagged)
            {
                // the customer account is flagged
                // log some errors and return
                return;
            }

            // normal order processing

        }
    }

    public class  Customer
    {
        public decimal  Balance { get; private set; }

        public bool  IsNotFlagged
        {
            get  { return  Balance < 30m; }
        }
    }
}
```

如上代码中的双重否定可读性非常低，因为我们很难搞明白双重否定的正确值。要重构它也非常容易，如下是重构后的代码：

```
using System.Collections.Generic;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.DoubleNegative.After
{
    public class Order
    {
        public void Checkout(IEnumerable<Product> products, Customer customer)
        {
            if (customer.IsFlagged)
            {
                // the customer account is flagged
                // log some errors and return
                return;
            }
            // normal order processing
        }
    }
```

```
    public class Customer
    {
        public decimal Balance { get; private set; }
        public bool IsFlagged
        {
            get { return Balance >= 30m; }
        }
    }
}
```

总结：　”双重否定“很容易让人产生错误的判断，也很难让人理解你的代码，所以这个重构在我们的代码中是很重要的，尤其是在判断条件很多且业务复杂的时候。

3.2.27 去除上帝类

概念：本文中的”去除上帝类”是指把一个看似功能很强且很难维护的类，按照职责把自己的属性或方法分派到各自的类中或分解成功能明确的类，从而去掉上帝类。

正文：我们经常可以在一些原来的代码中见到一些类明确违反了 SRP 原则（单一原则），这些类通常以“Utils”或“Manager”后缀结尾，但有时这些类也没有这些特征，它仅仅是多个类多个方法的组合。另一个关于上帝类的特征是通常这些类中的方法被用注释分隔为不同的分组。那么久而久之，这些类被转换为那些没有人愿意进行归并到合适类的方法的聚集地，对这些类进行重构是将类中的代码按照职责分派到各自的类中，这样就解除了上帝类，也减轻了维护的负担。

```
using System.Collections.Generic;
using LosTechies.DaysOfRefactoring.EncapsulateCollection.After;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
using Customer = LosTechies.DaysOfRefactoring.BreakResponsibilities.After.Customer;
namespace LosTechies.DaysOfRefactoring.SampleCode.RemoveGodClasses.Before
{
    public class CustomerService
    {
        public decimal CalculateOrderDiscount(IEnumerable<Product> products, Customer customer)
        {
            // do work
        }

        public bool CustomerIsValid(Customer customer, Order order)
        {
            // do work
```

```
        }

        public IEnumerable<string> GatherOrderErrors(IEnumerable<Product> products, Customer customer)
        {
            // do work
        }

        public void Register(Customer customer)
        {
            // do work
        }

        public void ForgotPassword(Customer customer)
        {
            // do work
        }
    }
}
```

我们看到要重构上面的代码是很简单的，只要将相关的方法按职责分派到对应的类中即可，带来的好处就是这会降低代码的颗粒度并减少未来维护代码的成本。下面是重构后的代码，它将上面的代码按照职责分为了两个不同的类。

```
using System.Collections.Generic;
using LosTechies.DaysOfRefactoring.EncapsulateCollection.After;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
using Customer = LosTechies.DaysOfRefactoring.BreakResponsibilities.After.Customer;
namespace LosTechies.DaysOfRefactoring.SampleCode.RemoveGodClasses.After
{
    public class CustomerOrderService
    {
        public decimal CalculateOrderDiscount(IEnumerable<Product> products, Customer customer)
        {
            // do work
        }

        public bool CustomerIsValid(Customer customer, Order order)
        {
            // do work
        }

        public IEnumerable<string> GatherOrderErrors(IEnumerable<Product> products, Customer customer)
        {
            // do work
```

```
        }
    }

    public class CustomerRegistrationService
    {
        public void Register(Customer customer)
        {
            // do work
        }

        public void ForgotPassword(Customer customer)
        {
            // do work
        }
    }
}
```

总结：　“去除上帝类”是我们经常容易造成的，第一是因为简便，看到有一个现成的类，大家都会喜欢把代码往里面写，最后导致越写越大，并且声明功能都有，这样即降低了可读性，也造成了维护的负担。

3.2.28 为布尔方法命名

概念：本文中的”为布尔方法命名”是指如果一个方法带有大量的 bool 参数时，可以根据 bool 参数的数量，提取出若干个独立的方法来简化参数。

正文：我们现在要说的重构并不是普通字面意义上的重构，它有很多值得讨论的地方。当一个方法带有大量的 bool 参数时，会导致方法很容易被误解并产生非预期的行为，

根据布尔型参数的数量，我们可以决定提取出若干个独立的方法来。具体代码如下：

```
using LosTechies.DaysOfRefactoring.BreakResponsibilities.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.RenameBooleanMethod.Before
{
    public class BankAccount
    {
        public void CreateAccount(Customer customer, bool withChecking, bool withSavings, bool withStocks)
        {
            // do work
        }
    }
}
```

我们可以将上面的 bool 参数以独立方法的形式暴露给调用端以提高代码的可读性，同时我们还需要将原来的方法改为 private 以限制其可访问性。显然我们关于要提取的独立方法会有一个很大的排列组合，这是一大缺点，所以我们可以考虑引入”参数对象“重构。

```
using LosTechies.DaysOfRefactoring.BreakResponsibilities.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.RenameBooleanMethod.After
{
    public class BankAccount
    {
        public void CreateAccountWithChecking(Customer customer)
        {
            CreateAccount(customer, true, false);
        }

        public void CreateAccountWithCheckingAndSavings(Customer customer)
        {
            CreateAccount(customer, true, true);
        }

        private void CreateAccount(Customer customer, bool withChecking, bool withSavings)
        {
            // do work
        }
    }
}
```

总结：　“为布尔方法命名”这个重构在很多时候都不常用，如果用户的参数可枚举，我们一般会枚举它的值，不过使用这种重构也有好处，就是分解开来以后，方法多了，参数少了，代码维护起来方便了一些。

3.2.29 去除中间人对象

概念：本文中的”去除中间人对象”是指把 在中间关联而不起任何其他作用的类移除，让有关系的两个类直接进行交互。

正文：有些时候在我们的代码会存在一些”幽灵类“，设计模式大师 Fowler 称它们为“中间人”类，“中间人”类除了调用别的对象之外不做任何事情，所以“中间人”类没有存在的必要，我们可以将它们从代码中删除，从而让交互的两个类直接关联。

如下代码所示，Consumer 类要得到 AccountDataProvider 的数据，但中间介入了没起任何作用的 AccountManager 类来关联，所以我们应当移除。

```
using LosTechies.DaysOfRefactoring.PullUpField.After;

namespace LosTechies.DaysOfRefactoring.SampleCode.RemoveMiddleMan.Before
{
    public class Consumer
    {
        public AccountManager AccountManager { get; set; }

        public Consumer(AccountManager accountManager)
        {
            AccountManager = accountManager;
        }

        public void Get(int id)
        {
            Account account = AccountManager.GetAccount(id);
        }
    }

    public class AccountManager
    {
        public AccountDataProvider DataProvider { get; set; }

        public AccountManager(AccountDataProvider dataProvider)
        {
            DataProvider = dataProvider;
        }

        public Account GetAccount(int id)
        {
            return DataProvider.GetAccount(id);
        }
    }

    public class AccountDataProvider
    {
        public Account GetAccount(int id)
        {
            // get account
        }
    }
}
```

重构后的代码如下所示，Consumer 和 AccountDataProvider 直接进行关联，这样代码就简单了。

```
using LosTechies.DaysOfRefactoring.PullUpField.After;

namespace LosTechies.DaysOfRefactoring.SampleCode.RemoveMiddleMan.After
{
    public class Consumer
    {
        public AccountDataProvider AccountDataProvider { get; set; }

        public Consumer(AccountDataProvider dataProvider)
        {
            AccountDataProvider = dataProvider;
        }

        public void Get(int id)
        {
            Account account = AccountDataProvider.GetAccount(id);
        }
    }

    public class AccountDataProvider
    {
        public Account GetAccount(int id)
        {
            // get account
        }
    }
}
```

总结：　“去除中间人对象”很多时候都会很有作用，尤其是在误用设计模式的代码中最容易见到，设计模式中的适配器模式和代理模式等都用中间的类是两者进行关联，这是比较合理的，因为中间类做了很多事情，而对于没有任何作用的中间类应该移除。

3.2.30 尽快返回

概念：　本文中的”尽快返回”是指把原来复杂的条件判断等语句用尽快返回的方式简化代码。

正文：如首先声明的是前面讲的”分解复杂判断“，简单的来说，当你的代码中有很深的嵌套条件时，花括号就会在代码中形成一个长长的箭头。我们经常在不同的代码中看到这种情

况，并且这种情况也会扰乱代码的可读性。下代码所示，HasAccess 方法里面包含一些嵌套条件，如果再加一些条件或者增加复杂度，那么代码就很可能出现几个问题：1，可读性差 2，很容易出现异常 3，性能较差

```
using System.Collections.Generic;
using System.Linq;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
using Customer = LosTechies.DaysOfRefactoring.BreakResponsibilities.After.Customer;
namespace LosTechies.DaysOfRefactoring.SampleCode.ReturnASAP.Before
{
    public class Order
    {
        public Customer Customer { get; private set; }

        public decimal CalculateOrder(Customer customer, IEnumerable<Product> products, decimal discounts)
        {
            Customer = customer;
            decimal orderTotal = 0m;
            if (products.Count() > 0)
            {
                orderTotal = products.Sum(p => p.Price);
                if (discounts > 0)
                {
                    orderTotal -= discounts;
                }
            }
            return orderTotal;
        }
    }
}
```

那么重构上面的代码也很简单，如果有可能的话，尽量将条件判断从方法中移除，我们让代码在做处理任务之前先检查条件，如果条件不满足就尽快返回，不继续执行。下面是重构后的代码：

```
using System.Collections.Generic;
using System.Linq;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
using Customer = LosTechies.DaysOfRefactoring.BreakResponsibilities.After.Customer;
namespace LosTechies.DaysOfRefactoring.SampleCode.ReturnASAP.After
{
    public class Order
    {
        public Customer Customer { get; private set; }
```

```
        public decimal CalculateOrder(Customer customer, IEnumerable<Product> products, decimal discounts)
        {
            if (products.Count() == 0)
                return 0;
            Customer = customer;
            decimal orderTotal = products.Sum(p => p.Price);
            if (discounts == 0)
                return orderTotal;
            orderTotal -= discounts;
            return orderTotal;
        }
    }
}
```

总结：这个重构很重要，它和前面讲的”分解复杂判断“有些类似，我们在做复杂的处理过程时，要经常考虑这个重构，用好了它，会对我们的帮助很大。

3.2.31 使用多态代替条件判断

概念：本文中的”使用多态代替条件判断”是指如果你需要检查对象的类型或者根据类型执行一些操作时，一种很好的办法就是将算法封装到类中，并利用多态性进行抽象调用。

正文：本文展示了面向对象编程的基础之一“多态性”， 有时你需要检查对象的类型或者根据类型执行一些操作时，一种很好的办法就是将算法封装到类中，并利用多态性进行抽象调用。

如下代码所示，OrderProcessor 类的 ProcessOrder 方法根据 Customer 的类型分别执行一些操作，正如上面所讲的那样，我们最好将 OrderProcessor 类中这些算法（数据或操作）封装在特定的 Customer 子类中。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.ReplaceWithPolymorphism.Before
{
    public abstract class Customer
    {
    }
```

```
    public class Employee : Customer
    {
    }

    public class NonEmployee : Customer
    {
    }

    public class OrderProcessor
    {
        public decimal ProcessOrder(Customer customer, IEnumerable<Product> products)
        {
            // do some processing of order
            decimal orderTotal = products.Sum(p => p.Price);
            Type customerType = customer.GetType();
            if (customerType == typeof(Employee))
            {
                orderTotal -= orderTotal * 0.15m;
            }
            else if (customerType == typeof(NonEmployee))
            {
                orderTotal -= orderTotal * 0.05m;
            }
            return orderTotal;
        }
    }
}
```

重构后的代码如下，每个 Customer 子类都封装自己的算法，然后 OrderProcessor 类的 ProcessOrder 方法的逻辑也变得简单并且清晰了。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.ReplaceWithPolymorphism.After
{
    public abstract class Customer
    {
        public abstract decimal DiscountPercentage { get; }
    }

    public class Employee : Customer
```

```
    {
        public override decimal DiscountPercentage
        {
            get { return 0.15m; }
        }
    }

    public class NonEmployee : Customer
    {
        public override decimal DiscountPercentage
        {
            get { return 0.05m; }
        }
    }

    public class OrderProcessor
    {
        public decimal ProcessOrder(Customer customer, IEnumerable<Product> products)
        {
            // do some processing of order
            decimal orderTotal = products.Sum(p => p.Price);
            orderTotal -= orderTotal * customer.DiscountPercentage;
            return orderTotal;
        }
    }
}
```

总结：　“使用多态代替条件判断”这个重构在很多时候会出现设计模式中（常见的工厂家族、策略模式等都可以看到它的影子），因为运用它可以省去很多的条件判断，同时也能简化代码、规范类和对象之间的职责。

4、改善代码设计

4.1、版权声明

文章出处：http://www.cnblogs.com/technology/archive/2011/05/17/2048735.html
文章作者：陈　华

4.2、内容详情

4.2.1 总结篇

【找出需要重构的地方】

最明显可能需要重构的地方包括：注释，长的方法，长的类，长的参数列表．这些都是很快能看出来的．

注释 (Comments)

1．函数中有处注释在说明下面的一个代码块在做什么事情，通常采用 Extract Method 将这些代码放到一个单独的方法中．
2．如果某个函数上面的注释在解释这个函数是做什么用的，多数情况下是这个方法名的名字取得不是很好，通常使用 Rename Method 进行重命名．
3．如果注释在解释代码运行到这边需要满足的条件，考虑使用 Introduce Assertion．
另外，我觉得过分的追求代码中没有注释也是不对的，注释需要恰到好处．

长的方法 (Long Method)

1．寻找代码中的注释，使用 Extract Method 将函数分成一小块一小块的方法．
2．观察代码中有没有太多重复的代码，使用 Extract Method 把这些代码分离出去，在原来的代码中调用这些小方法．

长的类 (Long Class)

造成类的代码过长的原因可能有以下两点：

1．随着时间的推移，类中在不停的增加新的功能．可以使用 Extract Class，Extract Subclass，Extract Interface 解决这个问题．
2．类中有很多涉及界面的代码，比如更新某个控件．可以使用 Duplicate Observed

Data 来帮助提取一个用于更新界面的类，也就是所谓的 "界面与业务分离"，不过负责更新界面的这个类要写好更新的同步机制.

长的参数列表 (Long Parameter List)

1. 如果参数能通过某个函数直接获得，应该去除该参数项，使用 Replace Parameter with Method，在函数中直接调用.
2. 如果某个物件能够提供函数中所需的所有参数，则可以将整个物件作为参数传递给函数, Preserve Whole Object.
3. 如果有好几个函数都包含某几个参数，这几个参数很有可能就是所谓的数据泥团 (Data Clumps)，如果合理，尝试使用 Introduce Parameter Object，将数据泥团封装到一个物件中，让函数直接调用这个物件.

【重构，各抒己见】

一提到重构，不少人有一肚子的口水要喷. 一部分人心怀激动的感概重构所带来的诸多好处，一少些人愤懑它所带来的不属于自己的利益，甚至是某次不恰当的重构所带来的毁灭性结果. 还有很多人在讨论某项重构究竟值不值得去做 (这往往应该根据不同情况具体分析后做出不同的选择).

对于重构，就好比你想用积木搭一个建筑物，重构就像是把一块木头削成一个个小积木的过程，而如何去搭建，大部分是设计模式所要解决的问题. 重构所能带来的好处大多数的共识是：重构后代码能够更让别人读懂和理解，能发现代码隐藏的缺陷，帮助改善软件设计，新需求来临时能提高编程效率.

记得我第一个独立完成的程序是一个课程表软件 (左图)，用 VB.NET 写的，花了三天功夫写了 2000 多行，当时一共写了近 10 个界面用于包括设置一个星期中每天的课程还有其它一些杂七杂八的内容. 当时我也知道 "重复" 的代码很多，但只想着 "能运行就好了"，于是还光明正大的把一大块代码复制到另外几个模板里. 如果想对这样的代码做点优化的话，这时不应该是重构，而应该重写. 后来我也不用这么笨拙的工具来存储课程表了，直接用 HTML+CSS 写一个网页.

不应该先乱七八糟的写完一堆代码之后再思考怎么去改善，应该在写得过程中不断的尝试着对现有的代码作出一些优化，最起码别出现太让人费解的变量名和方法名.

重构是一项需要不少时间的工作，如果系统即将到了发布的 deadline，这时并不适合大范围的重构.

很多人觉得用不到重构，他们觉得重构得不恰当会导致原本好歹能运行的系统工作

不正常. 或者重构需要他们大把大把的精力和脑细胞, 而重构带来的利益很可能不属于他. 还有一些人不太喜欢频繁的函数调用, 认为重构会降低一些系统性能等等原因. 但不要因为不会使用到重构, 而不去学习它, 更不用去抨击重构带来实际好处 (往往因为你没体会到).

【参考】

这几篇都是我看书并联想自己写过的代码的总结, 废话较少, 特别感谢不少细心的园友帮我纠正一些错误, 还提出了一些很有价值的建议.

参考 <重构 —— 改善既有代码设计>,<重构手册>, 还有自己的一些理解和经验.

4.2.2 优化函数的构成

【1. Extract Method (提炼函数)】

解释:

如果发现一个函数的代码很长, 很可能的一种情况是这个函数做了很多事情, 找找看函数中有没有注释, 往往注释都是为了解释下面一块代码做的什么事情, 可以考虑将这块代码提炼(Extract)成一个独立的函数.

这样做的好处不言而喻, 是面向对象五大基本原则中的单一职责原则 (Single Responsibility Principle), 比较长的函数被拆分成一个个小函数, 将有利于代码被复用.

冲动前:

```
public void Print(Employee employee)
{
    //print employee's information
```

```
    Console.WriteLine("Name:" + employee.Name);
    Console.WriteLine("Sex:" + employee.Sex);
    Console.WriteLine("Age:" + employee.Age);

    //print employee's salary
    Console.WriteLine("Salary:" + employee.Salary);
    Console.WriteLine("Bonus:" + employee.Bonus);
}
```

冲动后:

```
public void Print(Employee employee)
{
    //print employee's information
    PrintInfo(employee);
    //print employee's salary
    PrintSalary(employee);
}

public void PrintInfo(Employee employee)
{
    Console.WriteLine("Name:" + employee.Name);
    Console.WriteLine("Sex:" + employee.Sex);
    Console.WriteLine("Age:" + employee.Age);
}

public void PrintSalary(Employee employee)
{
    Console.WriteLine("Salary:" + employee.Salary);
    Console.WriteLine("Bonus:" + employee.Bonus);
}
```

【2. Inline Method (将函数内联)】

解释:

有些函数很短，只有一两行，而且代码的意图也非常明显，这时可以考虑将这个函数干掉，直接使用函数中的代码.

物件中过多的方法会让人感到不舒服，干掉完全不必要的函数后代码会更简洁.

冲动前:

```
public bool IsDeserving(int score)
```

```
{
    return IsScoreMoreThanSixty(score);
}

public bool IsScoreMoreThanSixty(int score)
{
    return (score > 60);
}
```

冲动后:

```
public bool IsDeserving(int score)
{
    return (score > 60);
}
```

【3. Inline Temp (将临时变量内联)】

解释:

如果有一个临时变量 (Temp)用来表示某个函数的返回值，一般来说，这样的做法挺好的. 但如果这个临时变量实在多余，将这个临时变量内联之后毫不影响代码的阅读，甚至这个临时变量妨碍了其它重构工作，就应该将这个临时变量内联化.

把这个临时变量干掉的好处在于减少了函数的长度，有时可以让其它重构工作更顺利的进行.

冲动前:

```
int salary = employee.Salary;
return (salary > 10000);
```

冲动后:

```
return (employee.Salary > 10000);
```

【4. Replace Temp With Query (用查询式代替临时变量)】

解释:

程序中有一个临时变量(Temp)用来保存某个表达式的计算结果，将这个计算表达式提炼(Extract)到一个独立的函数(即查询式 Query)中，将这个临时变量所有被调用的地方换成对新函数(Query)的调用，新函数还可以被其它函数使用.

好处在于减少函数长度，增加代码复用率，有利于代码进一步的重构. 并且注意

Replace Temp With Query 往往是 Extract Method 之前必不可少的步骤，因为局部变量会使代码不太容易被提炼，所以在进行类似的重构前可以将它们替换成查询式.

下面的这个例子不是很有必要使用 Replace Temp With Query，主要展示如何 Replace Temp With Query. 试想"冲动前"函数中有很多个代码块都使用到 totalPrice，突然有一天我发现这个函数太长，我需要将这一块块的代码提炼成单独的函数，这样就需要将 totalPrice = price * num; 放到每一个提炼出来的函数中. 而如果原来函数中使用的是查询式，就不存在这个问题. 如果查询式中的计算量很大，也不建议使用 Replace Temp With Query.

冲动前:

```
public double FinalPrice(double price, int num)
{
    double totalPrice = price * num;
    if (totalPrice > 100)
        return totalPrice * 0.8;
    else
        return totalPrice * 0.9;
}
```

冲动后:

```
public double FinalPrice(double price, int num)
{
    if (TotalPrice(price, num) > 100)
        return TotalPrice(price, num) * 0.8;
    else
        return TotalPrice(price, num) * 0.9;
}

public double TotalPrice(double price, int num)
{
    return price * num;
}
```

【5. Introduce Explaining Variable (引入可以理解的变量)】

解释:

很多时候在条件逻辑表达式中，很多条件令人难以理解它的意义，为什么要满足这个条件? 不清楚. 可以使用 Introduce Explaining Variable 将每个条件子句提炼出来，分别用一个恰当的临时变量名表示条件子句的意义.

好处在于增加了程序的可读性.

冲动前:

```
if ((operateSystem.Contains("Windows")) && (browser.Contatins("IE")))
{
    //do something
}
```

冲动后:

```
bool isWindowsOS = operateSystem.Contains("Windows");
bool isIEBrowser = browser.Contatins("IE");
if (isWindowsOS && isIEBrowser)
{
    //do something
}
```

【6. Split Temporary Variable (撇清临时变量)】

解释:

例如代码中有个临时变量在函数上面某处表示长方形周长，在函数下面被赋予面积，也就是这个临时变量被赋值超过一次，且表示的不是同一种量. 应该针对每次赋值，分配一个独立的临时变量.

一个变量只应表示一种量，否则会令代码阅读者感到迷惑.

冲动前:

```
double temp = (width + height) * 2;
//do something
temp = width * height;
//do something
```

冲动后:

```
double perimeter = (width + height) * 2;
//do something
double area = width * height;
//do something
```

【7. Remove Assignments to Parameters (消除对参数的赋值操作)】

解释:

传入参数分"传值"和"传址"两种，如果是"传址"，在函数中改变参数的值无可厚非，因为我们就是想改变原来的值．但如果是"传值"，在代码中为参数赋值，就会令人产生疑惑．所以在函数中应该用一个临时变量代替这个参数，然后对这个临时变量进行其它赋值操作．

冲动前:

```
public double FinalPrice(double price, int num)
{
    price = price * num;
    //other calculation with price
    return price;
}
```

冲动后:

```
public double FinalPrice(double price, int num)
{
    double finalPrice = price * num;
    //other calculation with finalPrice
    return finalPrice;
}
```

【8. Replace Method with Method Object (用函数物件代替函数)】

解释:

冲动的写下一行行代码后，突然发现这个函数变得非常大，而且由于这个函数包含了很多局部变量，使得无法使用 Extract Method，这时 Replace Method with Method Object 就起到了杀手锏的效果．做法是将这个函数放入一个单独的物件中，函数中的临时变量就变成了这个物件里的值域 (field).

冲动前:

```
class Bill
{
    public double FinalPrice()
    {
        double primaryPrice;
        double secondaryPrice;
        double teriaryPrice;
        //long computation
        ...
    }
```

```
}
```

冲动后:

```
class Bill
{
    public double FinalPrice()
    {
        return new PriceCalculator(this).compute();
    }
}

class PriceCalculator
{
    double primaryPrice;
    double secondaryPrice;
    double teriaryPrice;

    public PriceCalculator(Bill bill)
    {
        //initial
    }

    public double compute()
    {
        //computation
    }

}
```

【9. Substitute Algorithm (替换算法)】

解释:

有这么一个笑话:

某跨国日化公司，肥皂生产线存在包装时可能漏包肥皂的问题，肯定不能把空的肥皂盒卖给顾客，于是该公司总裁命令组成了以博士牵头的专家组对这个问题进行攻关，该研发团队使用了世界上最高精尖的技术（如红外探测，激光照射等），在花费了大量美金和半年的时间后终于完成了肥皂盒检测系统，探测到空的肥皂盒以后，机械手会将空盒推出去．这一办法将肥皂盒空填率有效降低至5%以内，问题基本解决．

而某乡镇肥皂企业也遇到类似问题，老板命令初中毕业的流水线工头想办法解决之，经过半天的思考，该工头拿了一台电扇到生产线的末端对着传送带猛吹，那些没有装填肥皂的肥皂盒由于重量轻就都被风吹下去了...

这个笑话可以很好的解释 Substitute Algorithm, 对于函数中复杂的算法, 尽量想办法将这个算法简单化, 从而达到与之前同样甚至更好的效果.

4.2.3 优化物件之间的特性

【1. Move Method (函数搬家)】

解释:

如果 ClassA 的某个函数对 ClassB 有过多的依赖, 可以考虑将这个函数搬到 ClassB 中, 在 ClassA 的这个函数中直接调用 ClassB 中这个函数的返回值.

这样做的好处是减少物件与物件之间的耦合度, 很多情况下这样做更利于进一步的重构.

冲动前:

```
class EmployeeSalary
{
    private double baseSalary = 15000.0;

    public double Salary(Employee employee)
    {
        return baseSalary + 10000 / employee.Level;
    }
    // other method with baseSalary
}

class Employee
{
    public int Level { get; set; }
}
```

冲动后:

```
class EmployeeSalary
{
    private double baseSalary = 15000.0;

    public double Salary(Employee employee)
    {
        return employee.Salary(baseSalary);
```

```
    }
    // other method with baseSalary
}

class Employee
{
    public int Level { get; set; }
    public double Salary(double baseSalary)
    {
        return baseSalary + 10000 / Level;
    }
}
```

【2. Move Field (值域搬家)】

解释:

有一天发现公司原来计算员工工资的方法不合适了，比如不是所有的员工起薪(baseSalary) 都是一万五，我想把 baseSalary 搬到 Employee 这个物件中作为员工的一个属性.

这样做可使程序扩展性变得更好，最明显的是我可以设置不同员工的起薪了.

冲动前:

```
class EmployeeSalary
{
    private double baseSalary = 15000.0;

    public double Salary()
    {
        double salary = baseSalary;
        //do some compution with salary
        return salary;
    }
}
```

冲动后:

```
class EmployeeSalary
{
    public double Salary(Employee employee)
    {
        double salary = employee.BaseSalary;
```

```
        //do some compution with salary
        return salary;
    }
}

class Employee
{
    public double BaseSalary { get; set; }
}
```

【3. Extract Class (提炼类)】

解释:

当某个物件做的事情过多，这样的物件往往含有大量的字段，属性和方法．应该由两个或更多个物件来分担这些责任，这时需要使用 Extract Class.

冲动前:

```
class Employee
{
    public double BaseSalary { get; set; }
    public double Level { get; set; }

    public double Salary()
    {
        double salary = BaseSalary;
        //do some complex compution with salary
        return salary;
    }
}
```

冲动后:

```
class EmployeeSalary
{
    public double Salary(Employee employee)
    {
        double salary = employee.BaseSalary;
        //do some complex compution with salary
        return salary;
    }
}
```

```
class Employee
{
    public double BaseSalary { get; set; }
    public double Level { get; set; }

    public double Salary()
    {
        EmployeeSalary salary = new EmployeeSalary();
        return salary.Salary(this);
    }
}
```

【4. Inline Class (将类内联)】

解释:

Inline Class 和 Extract Class 正好相反. 当一个物件没有做它应该做的事情, 还专门使用了另一个物件来协助它完成这个职责, 这时可以考虑使用 Inline Class.

如上面所示的例子, 如果我觉得 Employee 这个物件本身就应该实现 Salary 的计算工作, 而不是专门写一个 Salary 的计算物件来帮助它计算, 可以使用 Inline Class 将 Salary 内联到 Employee 中去, 也就是"冲动后"的代码重构成"冲动前"代码的样子.

【5. Hide Delegate (隐藏委托关系)】

解释:

试想这么一个情况: 有一个 Employee 类, 这个类中含有一个部门 (Department) 属性, 并且 Department 是一种类. 如果我想知道某职工所在部门的经理人是谁的时候, 我需要通过 xxEmployee.Department.Manger 来访问. 但这样做有个缺点是对于其它代码, Department 是 public 的, 其它代码能够访问到 Department 里的其它特性. 可以在 Employee 类中写一个 GetManger() 方法进行封装, 在调用的时候只需要 xxEmployee.GetManger() 就行了.

冲动前:

```
class Department
{
    public string Manger { get; set; }
}

class Employee
{
```

```
    public Department Department { get; set; }
}
```

冲动后:

```
class Department
{
    public string Manger { get; set; }
}

class Employee
{
    private Department Department;

    public string GetManger()
    {
        return Department.Manger;
    }
}
```

【6. Remove Middle Man (干掉中间人)】

解释:

这一条与上一条 Hide Delegate 是相反的. 当我们要访问 Department 的其它很多特性时, 我们用 Hide Delegate 写了一条条简单的委托访问函数, 当这些函数多到几乎访问遍了 Department 里的内容, 可以考虑使用 Remove Middle Man 方法将这些访问函数干掉.

如上面的例子, 就是将"冲动后"的代码重构成"冲动前"代码的样子.

【7. Introduce Foreign Method (引入外加函数)】

解释:

Introduce Foreign Method 有很深的 C#3.0 中扩展方法的味道, 但扩展方法比 Introduce Foreign Method 好在: 扩展方法就好象是被扩展的那个类型自己的方法一样, 而 Introduce Foreign Method 的函数还需要传递这个类型的参数, 但其实编译器编译扩展方法后还是会把参数传进去的, 扩展方法只是一种语法糖.

它的主要目的是实现被调用的类中没有实现的功能, 注意在进行本项重构时, 如果引入一个外加函数, 这说明这个函数本应该在被调用的类中实现. 下面举一个简单到不能再简单的例子, 这个例子只是说明怎么使用 Introduce Foreign Method, 我并不是说 Int32 类型就应该有一个 NextNum 的方法 , 并且实际中多数情况下这种重构用于引用类型.

冲动前:

```
int num = 1;
//I want to get num's next
int nextNum = num + 1;
```

冲动后:

```
int num = 1;
int nextNum = NextNum(num);

private static int NextNum(int arg)
{
    return arg + 1;
}
```

【8. Introduce Local Extension (引入本地扩展)】

解释:

如果我不想使用 Introduce Foreign Method, 我觉得它就本来应该实现某某功能, 如果被调用的类不是密封 (sealed) 的话, 可以自定义一个数据类型, 继承这个类, 在自己定义的数据类型中实现我想要它实现的功能, 这就是 Introduce Local Extension.

4.2.4 组织好你的数据

【1. Self Encapsulate Field (自封装值域)】

解释:

大部分类 (class) 中都会有一些值域 (field), 随之还会有一些方法使用到了这些值域. "如果调用这些值域"这个问题分为两种观点: 1. 应该直接调用它们 2. 应该通过访问函数调用它们.

我觉得大部分情况下直接调用比较方便, 过多的访问函数还会造成类中的函数过多, 当然将来如果我觉得直接调用带来了一些问题, 写一个一个的访问函数也并不是很困难.

下面的例子主要说明如何给值域写一个访问函数, 并通过访问函数调用值域的值.

冲动前:

```
private string _userName, _password;

public bool IsValid()
```

```
{
    bool isValid = !(String.IsNullOrEmpty(_userName) && String.IsNullOrEmpty(_password));
    return isValid;
}
```

冲动后:

```
private string _userName, _password;

public bool IsValid()
{
    bool isValid = !(String.IsNullOrEmpty(GetUserName()) && String.IsNullOrEmpty(GetPassword()));
    return isValid;
}

private string GetUserName()
{
    return _userName;
}

private string GetPassword()
{
    return _password;
}
```

【2. Replace Data Value with Object (以物件取代数据值)】

解释:

如果你发现代码中有很多字段或者值域似乎都在描述某一样东西，可以考虑将这些字段或者值域封装到一个类中，用这个物件代替原先代码中繁杂的字段和值域.

冲动前:

```
class Order
{
    public string CustomerName { get; set; }
    public string CustomerAddress { get; set; }
    public int CreditLevel { get; set; }
    public int CustomerTel { get; set; }
    //...
}
```

冲动后:

```
class Order
{
    public Customer Customer { get; set; }
}

class Customer
{
    public string CustomerName { get; set; }
    public string CustomerAddress { get; set; }
    public int CreditLevel { get; set; }
    public int CustomerTel { get; set; }
    //...
}
```

【3. Change Value to Reference (将值对象改为引用对象)】

解释:

一个类中有时包含值对象作为它的字段或者属性，如订单 Order 类中包含了一个客户 customerName 的字段. 如果同一个客户有好几份订单，那么每一份订单就都会保存一次客户的姓名. 如果这个客户的姓名改变了，那么就需要更改每份个订单中的 customerName.

如果将 customerName 提取出去，提炼一个 Customer 的类，订单使用的是 Customer 实例的引用，则只需更改实例中客户姓名的属性，因为所有的订单都是引用的这个客户实例，所以它们不需要作其它更改.

冲动前:

```
class Order
{
    private string _customerName;

    public Order(string customerName)
    {
        this._customerName = customerName;
    }
}
```

冲动后:

```
class Order
```

```
{
    private Customer _customer;

    public Order(Customer customer)
    {
        this._customer = customer;
    }
}

class Customer
{
    public string CustomerName { get; set; }
}
```

【4. Change Reference to Value (将引用对象改为值对象)】

解释:

如果给一个类传递引用类型变得不是你想要的效果，比如我曾经做过一个类似于 MSN 的软件，其中将人的头像，昵称，个性签名显示在一个自定义控件中，然后在使用了 ListBox.Add(myControl)，结果我惊奇的发现最后好友列表里居然都是同一个 (最后一个) 人. 原来 ListBox.Add() 传递进去的都是引用的同一个 myControl 实例，尽管代码貌似在动态的生成每一个人，但其实都只修改了一个 myControl，而 ListBox 仅仅了引用这一个对象. 后来我在代码中动态的生成不同对象的实例才解决了这个问题.

除了这样，还可以在你的类中定义一个值对象，每一次生成一个类的实例时，类中值对象的控制权总是属于自己.

【5. Replace Array with Object (用物件取代数组)】

解释:

一个类中可能会包含很多字段，你不能因为这些字段都是 string 类型，就通通把它们放到一个 string[] 里，在类的方法体中通过数组索引来获取你需要的值，这样很难让别人记得你 string[0] 表示的是姓名, string[1] 表示的是他的住址...

下面的将展示前后代码可读性的差距.

冲动前:

```
string[] _person = new string[5];

public string GetName()
{
    return _person[0];
```

```
}

public string GetAdd()
{
    return _person[1];
}
```

冲动后:

```
private Person _person;

public string GetName()
{
    return _person.Name;
}

public string GetAdd()
{
    return _person.Address;
}
```

【6. Duplicate Observed Data (复制被监视数据)】

解释:

一个良好的系统应该事先业务逻辑 (Business Logic) 与用户界面 (User Interface) 分离，如果没有这样做，常见的比如在 Program.cs 堆叠了大量的代码，业务逻辑与用户界面非常紧密的耦合在了一起. 实现业务逻辑与用户界面分离，最重要的实现是数据的同步机制, Duplicate Observed Data 重构手段用来完成这项工作. 由于例子较长，可以独立成篇，在以后介绍观察者模式 (Observer Pattern) 的时候会搬出来讨论.

【7. Change Unidirectional Association to Bidirectional (将单向关联改为双向)】

解释:

如果两个 Class 都需要使用对方的特性，但两者之间只有一条单向关联 (如只有调用方才能调用被调用的特性)，如果你想让被调用的那一方知道有哪些物件调用了自身，这时就需要使用 Change Unidirectional Association to Bidirectional.

通常的做法是在被调用方的代码里定义一个被调用方类型的集合 (Collection)，如 ArrayList<T>，List<T>(推荐使用 List<T>)，在[调用方] 调用 [被调用方] 的时候，在 [被调用方] 里的集合中也添加一下自己的引用.

【8. Change Bidirectional Association to Unidirectional (将双向关联改为单向)】

解释:

使用了 Change Unidirectional Association to Bidirectional 之后，如果需求变了，发现双向关联是没有必要的，这时就需要将双向关联改为单向，也就是 Change Bidirectional Association to Unidirectional，步骤与 Change Unidirectional Association to Bidirectional 相反.

【9. Replace Magic Number with Symbolic Constant (用符号常量取代魔法数)】

解释:

新手在写程序的时候，往往不注重命名 (name)，他们对程序的要求也就是能正确运行即可. 在 IDE 中随便拖几个 Button，也不重新命名，于是在代码中出现了类似 Button1, Button2, Button3...之类的，你无法光看代码就能想象得出 Button1 对应 UI 中的哪一个按钮. 同样的坏习惯出现在代码的字段，临时变量的命名，到处是 a, aa, aaa...

使用 Replace Magic Number with Symbolic Constant 吧！如果你的 Button 是用于"发送"，那么你可以给它一个名字 —— btnSend；如果你的一个 Int32 型 变量用于表示一个人的年龄，请给它一个名字 —— age.

【10. Encapsulate Field (封装值域)】

解释:

如果你的 class 中存在一个 public 字段，请将它设置为属性.

面向对象的第一条原则就是封装 (encapsulation)，或者称之为数据隐藏 (Data Hiding)，如果一个字段想被外界访问到，你应该将它设为属性，在 C# 中设置属性比在 Java 中要方便许多，因为 C# 编译器帮你做了一些额外工作.

冲动前:

```
public string _name;
```

冲动后:

```
public string Name { get; set; }
```

【11. Encapsulate Collection (封装集合)】

解释:

如果你的 class 中有一个方法需要返回多个值，在类中有一个集合 (Collection) 暂时保持这些返回值，有些情况下应当避免让 [调用端] 直接访问这个集合. 如果 [调用端] 修改了你集合的某一项，而 [被调用端] 不知道自己的集合被修改了，而另外的一些方法仍然

在调用被修改后的集合，这样可能会造成无法预料的后果.

Encapsulate Collection 建议你在 [被调用端] 将这个集合封装 (至少你将它的访问权限设置为 private) 起来，有需要的话，并提供修改这个集合的函数.

【12. Replace Record with Data Class (用数据取代记录)】

解释:

与第二条 Replace Data Value with Object 和第五条 Replace Array with Object 做法相似，目的是提取类中的数据到一个描述记录 (Record) 的类中，方便以后可以对这个类进行扩展.

【13. Replace Subclass with Fields (用值域取代子类)】

解释:

子类建立在父类之上再增加新的功能或者重载父类可能的变化行为，有一种变化行为 (variant behavior) 成为常量函数 (constant method)，他们会返回一个硬编码 (hard-coded) 值，这个值一般有这样的用途：让不同的父类中的同一个访问函数返回不同的值，你可以在父类中将访问函数声明为抽象函数，并在不同的不同的子类中让它返回不同的值. 但如果子类中只有返回常量的函数，没有其它的作用，往往子类并没有太大的存在价值.

你可以在父类中设计一些与子类返回值相应的值域，再对父类做一些其它修改，从而可以消除这些子类，好处是避免不必要的子类带来的复杂性，这就是 Replace Subclass with Fields.

冲动前:

```
class Person
{
    protected abstract bool isMale();
    protected abstract char Code();
    //...
}

class Male : Person
{
    protected override bool isMale()
    {
        return true;
    }

    protected override char Code()
    {
```

```
        return 'M';
    }
}

class Female : Person
{
    protected override bool isMale()
    {
        return false;
    }

    protected override char Code()
    {
        return 'F';
    }
}
```

冲动后:

```
class Person
{
    public bool IsMale { get; set; }
    public char Code { get; set; }

    public Person(bool isMale, char code)
    {
        this.IsMale = IsMale;
        this.Code = code;
    }

    public Person Male()
    {
        return new Person(true, 'M');
    }

    public Person Female()
    {
        return new Person(false, 'F');
    }
    //...
}
```

调用的时候这样: Person Create_Chen = Person.Male();

4.2.5 简化条件表达式

【1. Decompose Conditional (分解条件式)】

解释:

"复杂的条件逻辑" 是导致复杂性上升最常见的地方, "条件表达式中堆积的计算过程", "条件式表达得不简洁"等等都是造成复杂的原因. Decompose Conditional 用于将这些复杂的元素从条件表达式中分离出去, 仅在条件表达式中调用简洁的函数.

这样做带来的直接好处是减少重复, 而且代码的可读性提高了.

冲动前:

```
if (date.After(SUMMER_START) && date.Before(SUMMER_END))
    charge = days * _price + _summerServiceTip;
else
    charge = days * _price;
```

冲动后:

```
if (date.IsSummer())
    charge = SummerCharge(days);
else
    charge = WinterCharge(days);
```

【2. Consolidate Conditional Expression (合并条件式)】

解释:

如果代码中有一连串的 if 检查语句, 检查语句中的条件不相同, 但最终的行为都是一样的. 对于这样的情况, 应该使用 "逻辑与" 和 "逻辑或" 将它们合并成一个条件表达式, 如果嫌这个合并条件后的表达式太罗嗦, 你还可以将这个表达式提取成一个函数.

冲动前:

```
if (computer.CPU != "T6670")
    return false;

if (computer.RAM != "1.00GB")
    return false;

if (computer.SytemType != "32-bit Operating System")
```

```
    return false;

//other compution
```

冲动后:

```
if ((computer.CPU != "T6670") || (computer.RAM != "1.00GB") || (computer.SytemType != "32-bit Operating System"))
 return false;

//other compution
```

你还可以将 if 里长长的条件表达式提取成一个方法，如 bool IsStandard(Computer computer)，这样在原来的 if 语句中只需要调用这个方法即可.

【3. Consolidate Duplicate Conditional Fragments (合并重复的条件片段)】

解释:

如果条件式的每个分支上都有同样一段代码，如果这段代码对条件分支在执行这段代码后执行后面的代码没有影响， 请将这段代码移到条件式的外面.

冲动前:

```
if (date.IsSummer())
{
    charge = days * _price + _summerServiceTip;
    PrintDetail();
}
else
{
    charge = days * _price;
    PrintDetail();
}

//other compution
```

冲动后:

```
charge = days * _price;

if (date.IsSummer())
    charge += _summerServiceTip;
```

```
PrintDetail();

//other compution
```

【4. Remove Control Flag (移除控制标志)】

解释:

很多代码里执行一个 for 或者 while 循环用于寻找一个数组里特点的元素，很多时候在循环开头就执行控制标志的检查，满足检查条件就继续执行循环查找元素. 如果这一次查找到了想要的元素，就更改控制标志的值，让它下次被检查出不符合条件，从而循环结束.

这并不是一个很好的做法，使用诸如 break，continue，return 语句会让你的代码意图更加直接，更加明显.

冲动前:

```
for (int i = 0; i < suspects.Length; i++)
{
    if (!found)
    {
        if (suspects[i].Name == guessName)
        {
            sendAlert();
            found = true;
        }
    }
}
```

冲动后:

```
for (int i = 0; i < suspects.Length; i++)
{
    if (suspects[i].Name == guessName)
    {
        sendAlert();
        break;
    }
}
```

【5. Replace Nested Conditional with Guard Clauses (以卫语句取代嵌套条件式)】

解释:

许多程序员觉得函数应该只有一个出口 (return), 结果导致函数中的条件逻辑 (Conditional Logic) 本来完全可以终止下面的代码继续执行 (因为没有必要), 结果却只在函数最后 return, 使人难以看清程序的执行路径.

Replace Nested Conditional with Guard Clauses 用来解决这个问题, 它能带给代码可读性的提高, 还有性能上一点点的优化.

冲动前:

```
double charge;

if (IsSummer(date))
{
    //...
    SummerCharge(charge);
}
else
{
    //...
    WinterCharge(charge);
}

return charge;
```

冲动后:

```
double charge;

if (IsSummer(date))
{
    //...
    SummerCharge(charge);
    return charge;
}
else
{
    //...
    WinterCharge(charge);
    return charge;
}
```

【6. Replace Conditional with Polymorphism (以多态取代条件)】

解释:

这条重构手法常常用于消除函数中长长的 switch-case 语句. 虽然写一个个的子类比较繁琐, 但随着项目的进行, 好处会体现出来的.

冲动前:

```
public double Salary(Employee employee)
{
    switch (employee.Type)
    {
        case Employee.Engineer:
            {
                //...
            }
        case Employee.Salesman:
            {
                //...
            }
        //...
        default:
            {
                //...
            }
    }
}
```

冲动后:

```
public abstract double Salary(Employee employee);

class Engineer : Employee
{
    public override double Salary(Employee employee)
    {
        //...
    }
}

class Salesman : Employee
{
    public override double Salary(Employee employee)
    {
        //...
```

```
    }
}
```

【7. Introduce Null Object (引入 Null 对象)】

解释:

如果代码中出现很多判断某值是不是为 null，诸如 if (XXX != null) {//...} else {//...} 这样的情况，可以考虑使用 Introduce Null Object 重构手段. 这个手段其实并不难以理解，可以简单理解成为某一个物件在为空状态下设定默认的值域和行为，可以建立一个子类，继承父类中需要对 "为空" 情况下做出响应的虚函数或者值域. 它是 Null Object 设计模式里的最基础最常见的手段.

【8. Introduce Assertion (引入断言)】

解释:

严格上说，引入断言并不是为了简化条件表达式，它主要是为了代替条件表达式上面的注释，通常这样的注释用来解释下面的条件表达式是基于什么样的假设之上的. 通常经过一系列的测试，发现所写的断言在任何情况下都是正确的，在系统发布的时候可以把它们全部删除掉.

在 C# 中引入断言使用 Debug.Assert() 方法，如果一切假设都是正确的，则代码会顺利的进行.

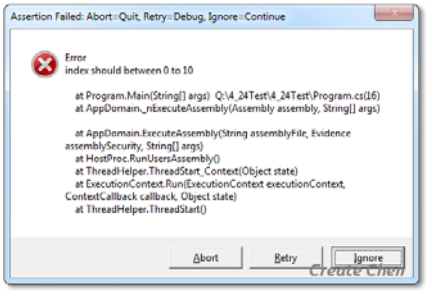
冲动前:

```
//index should between 0 to 10
return (customers[index] == "James") ? true : false;
```

冲动后:

```
Debug.Assert((index >= 0) && (index <= 10), "Error", "index should between 0 to 10");
return (customers[index] == "James") ? true : false;
```

如果断言错误，在运行的时候会有一个消息框给予错误信息的提示.



4.2.6 简化函数调用

【1. Parameterize Method (令函数携带参数)】

解释:

"令函数携带参数" 并不是简单的让你在函数里加上参数，如果函数里需要某个参数，我们谁都会加上它. 你可能发现这样的几个函数: 它们做着类似的事情，只是因为极少的几个值导致函数的策略不同，这时可以使用 Parameterize Method 消除函数中那些重复的代码了，而且可以用这个参数处理其它更多变化的情况.

下面有一个非常简单的例子.

冲动前:

```
public double FivePercentRaise()
{
    _salary *= 1.05;
    return _salary;
}

public double TenPercentRaise()
{
    _salary *= 1.10;
    return _salary;
}
```

冲动后:

```
public double Raise(int percent)
{
    _salary *= (1 + percent / 100);
    return _salary;
}
```

【2. Replace Parameter with Explicit Methods (用明确的函数取代参数)】

解释:

Replace Parameter with Explicit Methods 与 Parameterize Method 是相反的操作. 如果函数内根据不同的参数作出了不同的行为，那么可以考虑使用明确的函数取代参数. 不过这一条很少用到，很少出现类似下面例子的垃圾代码，建议使用 C# 里的属性，即使是"冲动后"的代码，在实际中也不是很多见.

冲动前:

```
public void Set(string name, int value)
{
    if (name.Equals("height"))
    {
        _height = value;
        return;
    }

    if (name.Equals("width"))
    {
        _width = value;
        return;
    }
}
```

冲动后:

```
public void SetHeight(int height)
{
    _height = height;
}

public void SetWidth(int width)
{
    _width = width;
}
```

【3. Preserve Whole Object (保持对象完整)】

解释:

你从某个物件中取出若干值，将这些值作为参数传递给某个函数，为何不考虑下把整个物件作为传递进去呢？当然具体情况具体分析，有时你确实需要按你原来做的那样做.

冲动前:

```
int height = rectangle.Height;
int width = rectangle.Width;

int area = CalculateArea(height, width);
```

冲动后:

```
int area = CalculateArea(rectangle);
```

【4. Replace Parameter with Methods (以函数取代参数)】

解释:

如果函数的某个参数值可以通过方法直接获得，这种情况下，很多时候函数不应该通过参数来获取该值，应该在函数里直接通过方法获取需要的值.

冲动前:

```
double basePrice = basePrice * num;
double discount = GetDiscount();
double charge = DiscountPrice(basePrice, discount);
```

冲动后:

```
double basePrice = basePrice * num;
double charge = DiscountPrice(basePrice);
```

【5. Introduce Parameter Object (引入参数对象)】

解释:

你是否发现有几个参数总是同时的出现在一些函数的参数列表里？这几个参数有一个响亮的臭名 —— 数据泥团(Data Clump)，Introduce Parameter Object 让你将数据泥团封装成一个对象，从而在原先的函数直接传入整个对象即可，以后还可以对这个对象进行扩展.

我不主张对于所有的数据泥团都这么做，哪怕这个数据泥团出现过几十次上百次，如果数据泥团的各个数据直接的联系并不是那么紧密，它们不能用一个物件笼统的包含它们，这样的情况下"引入参数对象"可能并不是很适合.

【6. Remove Setting Method (移除设值函数)】

解释:

这一条没什么好说的，如果你的类中某个值域只应该在对象初始化时被设值，那么类中就不应该再有关于这个值域的设值函数，应该通通把它们删掉.

【7. Hide Method (隐藏某个函数)】

解释:

这一条也没什么好说的，如果你的类中某个函数只可能用于类本身的函数调用，其它类从来没有用到过，最好将这个函数的访问权限设置为 private. 当然这一步要谨慎，现在没有被其它类调用过，不代表以后不会!

【8. Replace Constructor with Factory Method (用工厂函数取代构造函数)】

解释:

"以工厂函数取代构造函数" 让我想起了 "以多态取代条件"，如果你在创建对象的时候不是仅仅对它做简单的构造，例如你构造一个 Employee 对象，Employee 可能是工程师，销售员，也有可能是设计师，应该根据不同的类别的职工创建 Employee 对象.

【9. Encapsulate Downcast (封装向下转型动作)】

解释:

要找到需要进行 Encapsulate Downcast 重构的函数并不困难，如果你的函数返回的是 object 类型，在别的地方调用这个函数获取一个 object 之后还要进行一下转型(如将这个 object 转成 int 类型)，你应该在函数里封装一下转型的操作.

冲动前:

```
public object LastPerson()
{
    return _persons.LastElement();
}
```

冲动后:

```
public Person LastPerson()
{
    return (Person)_persons.LastElement();
}
```

【10. Replace Error Code with Exception (用异常取代错误码)】

解释:

你的函数返回的是一个 int 型，你约定返回 1 代表这个函数运行正常，返回 -1

代表某个地方出错了，错误码就是指这里的 "1" 和 "-1". 你想想你的函数返回的是错误码，在调用这个函数的时候极有可能可能还需要判断你返回的是什么错误码，如果你收到的是 -1 就停止程序的运行，书中形容这样的做法很有趣：就好像因为错过一班飞机而自杀一样，如果真那么做，哪怕我是猫，我的九条命 (注：猫生命力比较强) 也早就赔光了.

别忘了 C# 里的异常处理，这是一个很重要的概念，是时候使用它们了.

【11. Replace Exception with Test (用测试取代异常)】

解释:

"异常处理" 是一个很不错的功能，但不要滥用，对于肯定不会出现异常的代码块就不要将它们放入 try...catch 中.

对于在一定条件下代码会抛出异常的某些情况，建议使用 Replace Exception with Test，你可以在可能抛出异常的代码前先测试一下运行正常的条件是否满足，满足之后再运行，不满足的话则是另一种运行方案，通过这样可以代替 try...catch 的使用.

冲动前:

```
try
{
    return _persons[index];
}
catch (IndexOutOfRangeException e)
{
    //...
}
//...
```

冲动后:

```
if (index < _persons.Length)
    return _persons[index];
//...
```

4.2.7 处理概括关系

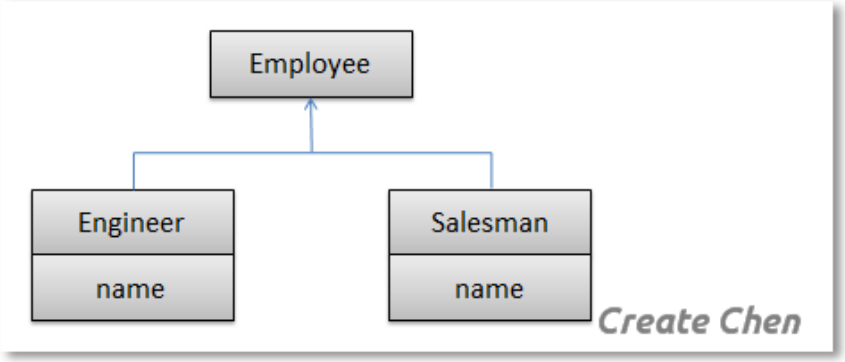
【1. Pull Up Field (提升值域)】

解释:

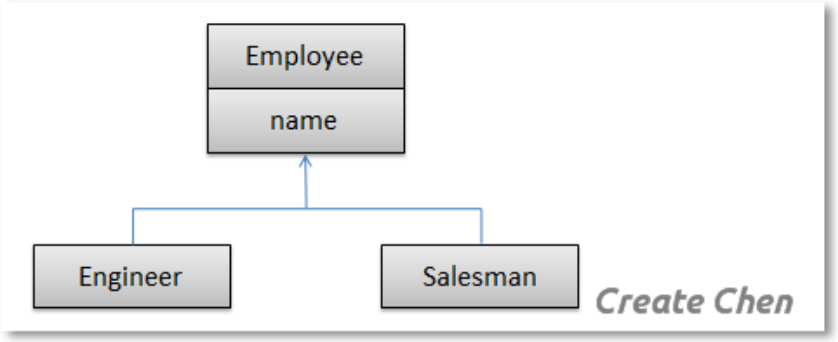
如果发现每个子类都拥有相同的某个值域，那么使用 Pull Up Field 将这个值域提升

到父类中去.

冲动前:



冲动后:

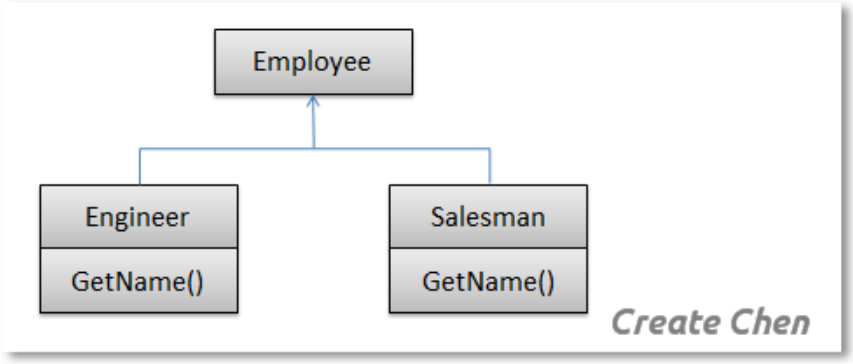


【2. Pull Up Method (提升函数)】

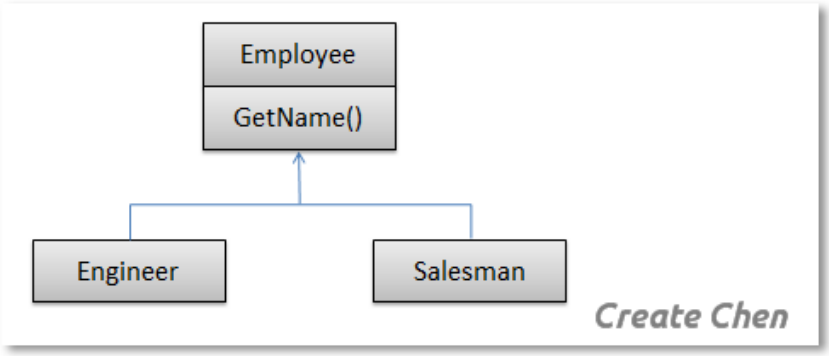
解释:

如果每个子类都有相同的某个函数，这个函数做同样的事情，而且结果也相同，那么使用 Pull Up Method 将这个函数提升到父类中去.

冲动前:



冲动后:



【3. Pull Up Constructor Body (提升构造函数)】

解释:

特别要注意每个子类中重复的代码，如果可能的话尽量将它们提炼成方法并搬到父类中去. 对于子类的构造函数，我们需要找出相同的部分，用这些相同的部分组成父类的构造函数.

如下面的例子，如果不光 Salesman，还有 Engineer 等等类别的员工在构造他们的时候都需要 name 和 level 属性，可以考虑使用 Pull Up Constructor Body 将设置这两个属性提升到父类的构造函数中去.

冲动前:

```
class Employee
{
    public string Name { get; set; }
    public int Level { get; set; }
    //...
}

class Salesman : Employee
{
    public string Hobby { get; set; }

    public Salesman(string name, int level, string hobby)
    {
        this.Name = name;
        this.Level = level;
        this.Hobby = hobby;
    }
    //...
```

```
}
//...
```

冲动后:

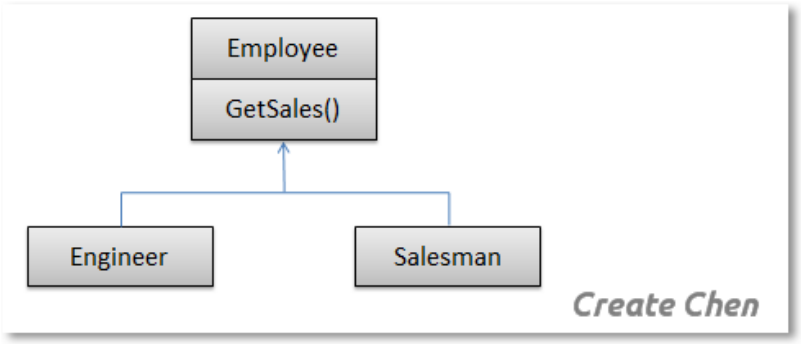
```
class Employee
{
    public string Name { get; set; }
    public int Level { get; set; }

    public Employee(string name, int level)
    {
        this.Name = name;
        this.Level = level;
    }
    //...
}

class Salesman : Employee
{
    public string Hobby { get; set; }

    public Salesman(string name, int level, string hobby)
        : base(name, level)
    {
        this.Hobby = hobby;
    }
    //...
}
//...
```

【4. Push Down Method (降低函数)】

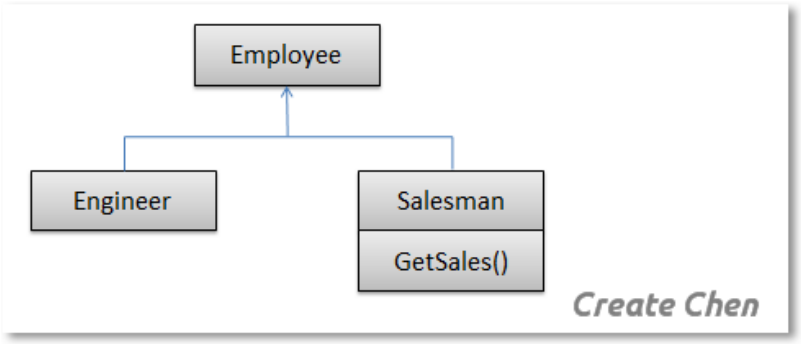
解释:

父类里有某个函数只与一部分子类有关，并不是与所有的子类都有关，使用 Push Down Method 重构手段将这些函数放到使用它们的子类中去，而不要放到父类中.

冲动前:



冲动后:



【5. Push Down Field (降低值域)】

解释:

> 与 Push Down Method 描述的问题类似，父类中如果某个值域并不是对于每个子类都有用的，应该把它放到需要它的子类中去.

【6. Extract Subclass (提炼子类)】

解释:

> 我们产生了类的一些实例，但并不是每个实例都用得到类中所有的特性，往往这是类中功能设计过多的原因造成的. 尝试从这个类中提炼出一些子类，子类中的功能应该划分得很明确.

【7. Extract Superclass (提炼父类)】

解释:

> 如果你发现有两个类，他们有很多相同的特性，尝试找出两个类中相同的特性，如果你能找到一个合适的理由让这两个类继承自一个父类，从而你可以提炼出这个父类，父类中

包含那两个类中相同的部分.

【8. Extract Interface (提炼接口)】

解释:

类与类之间经常会相互调用，比如 ClassA 的某个函数里需要 ClassB 里的某个值域或者某个函数的返回值，因此我将整个 ClassB 作为参数传递给 ClassA 的这个函数，这意味着 ClassA 的这个函数能够调用 ClassB 里所有的功能，可不可以给 ClassA 的这个函数划定一个特定的职能呢？让它只能做某些事情，而避免其它 "越权行为". 有句话常被人说起 —— 使用接口来降低耦合性，这就是 Extract Interface 的功劳.

这条重构手段经常被使用到，主要解决类对另一个类的依赖问题，降低了耦合性.

冲动前:

```
class Xml
{
    public void Read()
    {
        //...
    }

    public void Translate()
    {
        //...
    }
    //some other methods
}

class WeatherService
{
    public string GetWeather(Xml xml)
    {
        xml.Read();
        xml.Translate();
        //other code, but without xml object
    }
    //some other methods
}
```

冲动后:

```
interface IOperation
```

```
{
    void Read();
    void Translate();
}

class Xml : IOperation
{
    public void Read()
    {
        //...
    }

    public void Translate()
    {
        //...
    }
    //some other methods
}

class WeatherService
{
    public string GetWeather(IOperation operation)
    {
        operation.Read();
        operation.Translate();
        //other code, but without xml object
    }
    //some other methods
}
```

注意代码高亮的那行已经变成调用接口，而不是仅依赖调用 Xml 类的实例，对于单个类实现这样的接口并不是很有价值，如果很多类都实现了同样的接口，这将是很有用的事情.

【9. Collapse Hierarchy (去掉不必要的继承关系)】

解释:

庞大的继承体系很容易变得复杂，理清父类与子类它们各自的职能是非常重要的，你很有可能会发现不必要的子类，那么使用 Pull Up Field 和 Pull Up Method 将它干掉.

【10. Replace Inheritance with Delegation (用委派取代继承)】

解释:

如果你想让 ClassA 使用某个类 (如 ClassB) 的某个函数，就让 ClassA 继承自 ClassB，这将是一个多么糟糕的设计！你可以在 ClassA 中包含一个 ClassB 的值域，通过这个值域调用你需要的函数，这个值域就是"委派"，而你这时可以去掉 ClassA 和 ClassB 之间不该存在的继承关系了.

【11. Replace Delegation with Inheritance (用继承代替委派)】

解释:

这一条与 Replace Inheritance with Delegation 正好相反，如果你需要使用委派中的所有函数，这时你就应该想想它们之间是不是存在继承关系.

5、重构（Refactoring）技巧读书笔记

5.1、版权声明

文章出处：http://www.cnblogs.com/rickie/archive/2004/12/14/76663.html
文章作者：Rickie

5.2、内容详情

5.2.1 读书笔记之一

本文简要整理重构方法的读书笔记及个人在做 Code Review 过程中，对程序代码常用的一些重构策略。通过适当的重构代码，的确可以显著提高代码的质量，令人赏心悦目。毫无疑问，这些重构策略均来自于 Martin Fowler 的《重构－改善既有代码的设计》，只是如何在实际项目中灵活运用而已。（注：本文重构策略的名称及其大部分内容来自《重构－改善既有代码的设计》一书，Martin Fowler 著，侯捷等译）。

先看看重构的定义吧：

（1）Refactoring means rewriting existing source code with the intent of improving its design rather than changing its external behavior. The focus of refactoring is on the structure of the source code, changing the design to make the code easier to understand, maintain, and modify. － 来自 Borland Together 提供的文档，觉得这个定义很清晰明了。

（2）重构是这样一个过程：在不改变代码外在行为的前提下，对代码做出修改，已改进程序的内部结构。－来自 Martin Fowler 的定义。

不过，我一般使用一些重构的工具，如 ReSharper for VS.Net v1.0 和 Borland Together for VS.Net v2.0，这些重构工具可以帮助你做很多事情，可以简化你许多工作，同时也可以避免出现一些错误。关于 ReSharper for VS.Net v1.0 的简单介绍，可以参考本人以前的一篇 Posting【体验 ReSharper V1.0 for VS.Net 2003 - Part I，体验 ReSharper V1.0 for VS.Net 2003 - Part II】。另外 VS.Net 2005 已经内置了重构功能。不过，目前这些重构工具还远远不能涵盖各种重构方法，有总比没有好了。

因此，掌握必要的重构技巧逐步成为对程序员基本的要求，重要的是在掌握这些技巧后，也有助于类库初期设计的质量，避免或减少代码的坏味道(bad smell)。

代码坏味道（Bad Smell in Codes）及其重构策略

1．尽量消除重复的代码，将它们合而为一

根据重复的代码出现在不同的地方，分别采取不同的重构的策略：

- 在同一个 Class 的不同地方：通过采用重构工具提供的 Extract Method 功能提炼出重复的代码， 然后在这些地方调用上述提炼出方法。
- 在不同 Subclasses 中：通过 Extract Method 提炼出重复的代码，然后通过 Pull Up Method 将该方法移动到上级的 Super class 内。
- 在没有关系的 Classes 中：通过对其中一个使用 Extract Class 将重复的代码提炼到一个新类中，然后在另一个 Class 中调用生成的新类，消除重复的代码。

2．拆解过长的函数

过长的函数在我们的日常代码中经常可见，在 C#中常通过#region #endregion 区隔为不同的功能区域。

重构策略：通过 Extract Method 将过长的函数按照功能的不同进行适当拆解为小的函数，并且给这些小函数一个好名字。通过名字来了解函数提供的功能，提高代码的理解性。

3．拆解过大的类

过大的类也经常见到，特别是类中含有大量的成员变量。

重构策略：通过 Extract Class 将一些相关成员变量移植到新的 Class 中，如 Employee 类，一般会包含有联系方式的相关属性（电话， Mobile，地址，Zip 等等），则可以将这些移植到新的 EmployeeContact 类中。

4．过长的参数列

过长的参数列的主要问题是难以理解，并且难以维护。如果要增加新的参数或者删除某一参数，易造成参数前后不一致。

重构策略：如果可以通过向已存在的对象查询获取参数，则可通过 Replace Parameter with Method，移除参数列，通过在函数内部向上述已存在的对象查询来获取参数。

如果参数列中若干参数是已存在对象的属性，则可通过 Preserve Whole Object 将这些参赛替换为一个完整对象，这样不仅提高代码的可读性，同时已易于代码今后的维护。

另外，还可以将若干不相关的参数，使用 Introduce Parameter Object 来创建一个新的参数类。不过，我个人觉得如果这些情况过多的话，会产生很多莫名其妙的参数类了，反而降低代码的可读性。

个人觉得前面 4 种坏味道比较显而易见，也比较容易处理。

5.2.2 读书笔记之二

重构的确是未来软件工程师需要掌握的一项技能。目前一些支持.Net 的重构工具，如 ReSharper for VS.Net v1.0、Borland Together for VS.Net v2.0 和 VS.Net 2005 等，只能支持一些有限的、比较简单的重构策略。大量的重构策略需要软件工程师清晰的了解，人工为主，运用重构工具辅助进行。（注：本文重构策略的名称及其大部分内容来自《重构－改善既有代码的设计》一书，Martin Fowler 著，侯捷等译）。

下面的内容延续上一节的内容，其中提及的一些代码坏味道（Bad Smell in Codes）及其重构策略相对而言要比较麻烦一些。

代码坏味道（Bad Smell in Codes）及其重构策略

5．Divergent Change（发散式变化）

现象：当某个 Class 因为外部条件的变化或者客户提出新的功能要求等时，每次修改要求我们更新 Class 中不同的方法。不过这种情况只有在事后才能觉察到，因为修改都是在事后发生的么（废话）。

重构策略：将每次因同一条件变化，而需要同时修改的若干方法通过 Extract Class 将它们提炼到一个新 Class 中。实现目标是：每次变化需要修改的方法都在单一的 Class 中，并且这个新的 Class 内所有的方法都应该与这个变化相关。

6．Shotgun Surgery（霰弹式修改）

现象：当外部条件发生变化时，每次需要修改多个 Class 来适应这些变化，影响到很多地方。就像霰弹一样，发散到多个地方。

重构策略：使用 Move Method 和 Move Field 将 Class 中需要修改的方法及成员变量移植到同一个 Class 中。如果没有合适的 Class，则创建一个新 Class。实现目标是，将需要修改的地方集中到一个 Class 中进行处理。

比较 Divergent Change（发散式变化）和 Shotgun Surgery（霰弹式修改）：

前者指一个 Class 受到多种外部变化的影响。而后者指一种变化需要影响到多个 Class 需要修改。都是需要修理的对象。

7．Feature Envy（依恋情结）

现象：Class 中某些方法“身在曹营心在汉”，没有安心使用 Class 中的成员变量，而需要大量访问另外 Class 中的成员变量。这样就违反了对象技术的基本定义：将数据和操作行为（方法）包装在一起。

重构策略：使用 Move Method 将这些方法移动到对应的 Class 中，以化解其“相思之苦”，让其牵手。

8．Data Clumps（数据泥团）

现象：指一些相同数据项目（Data Items），如 Class 成员变量和方法中参数列表等，在多个 Class 中多次出现，并且这些数据项目有其内在的联系。

重构策略：通过使用 Introduce Parameter Object（创建新的参数对象取代这些参数）或 Preserve Whole Object（使用已存在的对象取代这些参数），实现使用对象代替 Class 成员变量和方法中参数列表，清除数据泥团，使代码简洁，也提高维护性和易读性。

9．Primitive Obsession（基本型偏执狂）

现象：在 Class 中看到大量的基本型数据项目（Data Item），如 Employee 类中有大量的数据成员，Employee#, FirstName, MiddleName, LastName, Address, State, City, Street, Zip, OfficePhone, CellPhone, Email……等等。

重构策略：使用 Extract Class（提炼新类）或 Preserve Whole Object（使用已存在的对象取代这些参数），实现使用对象代替基本型数据项目（Data Item）。如上述 Employee 类中就可分别提炼出 EmployeeName 和 EmployeeContact 两个新类。

10．Switch Statements（Switch 语句）

现象：同样的 Switch 语句出现在不同的方法或不同的 Class 中，这样当需要增加新的 CASE 分支或者修改 CASE 分支内语句时，就必须找到所有的地方，然后进行修改。这样，就比较麻烦了。

重构策略：

(1)首先采用 Extract Method 将 Switch 语句提炼到一个独立的函数。

(2)然后以 Move Method 搬移到需要多态性（Polymorphism）的 Superclass 里面或者是构建一个新的 Superclass。

(3)进一步使用 Replace Type Code with Subclasses 或者 Replace Type Code with State/Strategy。这步就比较麻烦些，不过记住如下基本规则：这里一般有 3 个 Class 分别为 Source Class、Superclass 和 Subclass。

Source Class：

- 使用 Self Encapsulate Field，将 Type Code 成员变量封装起来，也就是建立对应的 Setter/Getter 函数。
- 在 Source Class 中增加一个 Superclass 类型的成员变量，用来存放 Subclass 实例对象。
- 在 Source Class 中的 Getter 函数，通过调用 Superclass 的 Abstract Query 函数来完成。
- 在 Source Class 中的 Setter 函数，通过调用 Superclass 中的 Static 工厂化方法来获取合适的 Subclass 实例对象。

Superclass：

新建的一个 Class（注：就是上面通过 Move Method 搬移生成的 Superclass），根据 Type Code 的用途命名该 Class，作为 Superclass。

- 在 Superclass 中建立一个 Abstract Query 函数，用来获取 Subclass 的 Type Code。
- 在 Superclass 中创建 Static 工厂化方法生产对应的 Subclass 对象，这里会存在一个 Switch 语句（不要再动脑筋来重构这个 Switch 语句了，这个 Switch 语句不会在多处重复存在，并且这里用于决定创建何种 Subclass 对象，这是完全可以接受的）。

Subclass：

- 根据每一个 Switch/Type 分支，建立对应的 Subclass，并且 Subclass 的命名可以参考 Switch/Type 分支的命名。
- 在每一个 Subclass 中重载 Superclass 的 Abstract Query 函数，返回特定的 Type Code。

(4)现在 Superclass 仍然存在 Switch 分支，是时候轮到 Replace Conditional with Polymorphism 上场了。具体而言，就是在每一个 Subclass 中创建重载方法（注：该方法是 Superclass 中含有 Switch 语句的方法），并将 Superclass 中 Switch 语句对应的 Case 分支剪切过来。最后将 Superclass 中该方法初象化 Abstract，并清除 Switch 语句及其所有的 Case 分支。

这样就完成了整个重构过程，这个比较麻烦。

注：并不是一看到 Switch 语句及 CASE 分支，就马上/偏执狂采用上述重构策略进行重构，画蛇添足或吃亏不讨好（个人观点）。一般而言，只有看到多处出现相同的 Switch 语句时，才应该考虑进行重构。

5.2.3 读书笔记之三

重构虽然是对现有的代码进行设计，以提高代码的质量和灵活性，但实际上，如果软件工程师掌握重构技术，对其初期的软件设计也有很好的指导，减少不当设计或设计不足，减少代码坏味道（Bad Smell in Codes），构建良好的系统。

注：本文重构策略的名称及其大部分内容来自《重构－改善既有代码的设计》一书，Martin Fowler 著，侯捷等译。

代码坏味道（Bad Smell in Codes）及其重构策略

11．Parallel Inheritance Hierarchies（平行继承体系）

现象：为某个 class 增加一个 subclass 时，也必须为另一个 class 相应增加一个 subclass。

重构策略：　在一个 class 继承体系的对象中引用（refer to）另一个 class 继承体系的对象，然后运用 Move Method 和 Move Field 将被引用 class 中的一些方法和成员变量迁移宿主 class 中，消除被引用 class 的继承体系（注：这种平行继承体系好象比较少见也）。

12．Lazy Class（冗赘类）

现象：某一些 class 由于种种原因，现在已经不再承担足够责任，有些多余了。如同国有企业冗余人员一样，需要下岗了。

重构策略：通过 Collapse Hierarchy，将这些冗余的 class 合并到 superclass 或 subclass 中，或者通过 Inline Class（与 Extract Class 相反），将这些冗余 class 中的所有 Method/Field 迁移到其他相关的 class 中。

13．Speculative Generality（夸夸其谈未来性）

现象：系统中出现一些无用的 abstract class，或者非必要的 delegation（委托），或者多余的参数等等。

重构策略：分别使用 Collapse Hierarchy 合并 abstract class，使用 Inline Class 移除非必要的 delegation，使用 Remove Parameter 删除多余的参数。

14．Temporary Field（令人迷惑的暂时值域）

现象：class 中存在一些 Field，这些 Field 只在某种非常特定的情况下需要。

重构策略：通过 Extract Class 将这些孤独的 Field 及其相关的 Method 移植的一些新的 Class 中。提炼出来的新 Class 可能没有任何抽象意义，只是提供 Method 的调用，这些新 Class 一般称为 Method Object。

15．Message Chains（过度耦合的消息链）

现象：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，……，这就是 Message Chain，意味着 Message Chain 中任何改变，将导致 Client 端不得不修改。

重构策略：通过 Hide Delegate（隐藏委托关系）消除 Message Chain，具体做法是在 Message Chain 的任何地方通过 Extract Method 建立一个简单委托（Delegation）函数，来减少耦合（Coupling）。

16．Middle Man（中间转手人）

现象：过度运用 delegation，某个/某些 Class 接口有一半的函数都委托给其他 class，这样就是过度 delegation。

重构策略：运用 Remove Middle Man，移除简单的委托动作（也就是移除委托函数），

让 client 直接调用 delegate 受托对象。和上面的 Hide Delegate（隐藏委托关系）刚好相反的过程。

由于系统在不断的变化和调整，因此[合适的隐藏程度]这个尺度也在相应的变化，Hide Delegate 和 Remove Middle Man 重构策略可以系统适应这种变化。

另外，可保留一部分委托关系（delegation），同时也让 Client 也直接使用 delegate 受托对象。

17．Inappropriate Intimacy（狎昵关系）

现象：两个 Class 过分亲密，彼此总是希望了解对方的 private 成分。

重构策略：可以采用 Move Method 和 Move Field 来帮助他们划清界限，减少他们之间亲密行为。或者运用 Change Bidirectional Association to Unidirectional，将双向关联改为单向，降低 Class 之间过多的依存性（inter-dependencies）。或者通过 Extract Class 将两个 Class 之间的共同点移植到一个新的 Class 中。

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

现象：两个函数做相同的事情，却有不同的 signature。

重构策略：使用 Rename Method，根据他们的用途来重命名。另外，可以适当运用 Move Method 迁移某些行为，使 Classes 的接口保持一致。

19．Incomplete Library Class（不完美的程序库类）

现象：Library Class（类库）设计不是很完美，我们需要添加额外的方法。

重构策略：如果可以修改 Library Class 的 Source Code，直接修改最好。如果无法直接修改 Library Class，并且只想修改 Library Class 内的一两个函数，可以采用 Introduce Foreign Method 策略：在 Client Class 中建立一个函数，以外加函数的方式来实现一项新功能（一般而言，以 server class 实例作为该函数的第一个参数）。

如果需要建立大量的额外函数，可应该采用 Introduce Local Extension：建立一个新 class，使它包含额外函数，并且这个 class 或者继承或者 wrap（包装）source class。

20．Data Class（纯稚的数据类）

现象：Data Class 指：一些 Class 拥有 Fields，以及用来访问 Fields 的 getter/setter 函数，但是没有其他的功能函数。（感觉这些 Data Class 如同 Entity Class 或 Parameter Class，用来传递参数，我认为这种情况下没有必要重构。）

重构策略：找出其他 class 中访问 Data Class 中的 getter/setter 的函数，尝试以 Move Method

将这些函数移植到 Data Class 中，实现将数据和操作行为（方法）包装在一起，也让 Data Class 承担一定的责任（方法）。

21．Refused Bequest（被拒绝的遗赠）

现象：Subclass 不想或不需要继承 superclass 的部分函数和 Field。

重构策略：为 subclass 新建一个兄弟（sibling class），再运用 Push Down Method 和 Push Down Field 将 superclass 中的相应函数和 Field 下推到兄弟 class，这样 superclass 就只包含 subclass 共享的东西了。其实，也就是将 superclass 中一些与特定的函数和 Field 放到特定的 subclass 中，superclass 中仅包含 subclass 共享的函数和 Field。

如果不想修改 superclass，还可以运用 Replace Inheritance with Delegation 来达到目的。也就是以委托取代继承，在 subclass 中新建一个 Field 来保存 superclass 对象，去除 subclass 对 superclass 的继承关系，委托或调用 superclass 的方法来完成目的。

22．Comments（过多的注释）

现象：（晕倒，这个也要重构，Remove 掉所有的 Comments 吗？不是。）当代码中出现一段长长的注释，一般是由于代码比较糟糕，需要进行重构，除去代码的坏味道。

重构策略：通过上面提及的各种重构策略，将代码的坏味道去除，使注释变成多余。

如果需要注释/解释一段代码做了什么，则可以试试 Extract Method，提取出一个独立的函数，让函数名称解释该函数的用途/功能。另外，如果觉得需要注释来说明系统的某些假设条件，

也可尝试使用 Introduce Assertion（引入断言），来明确标明这些假设。

当你感觉需要撰写注释时，请先尝试重构，试着让所有的注释都变得多余。

代码的坏味道（Bad Smell in Codes）学习完了，走完了第一步。后面的内容也翻过多次，感觉还是蛮好懂的。嗯，今天掌握了挑别人代码毛病的精锐武器。郁闷的是，自己写的代码也需要重构。