01

Powerpoint is a complete presentation graphic package. It gives you everything you need to produce a professional-looking presentation

重构-改善既有代码的设计

Refactoring-Improving the Design of Existing Code

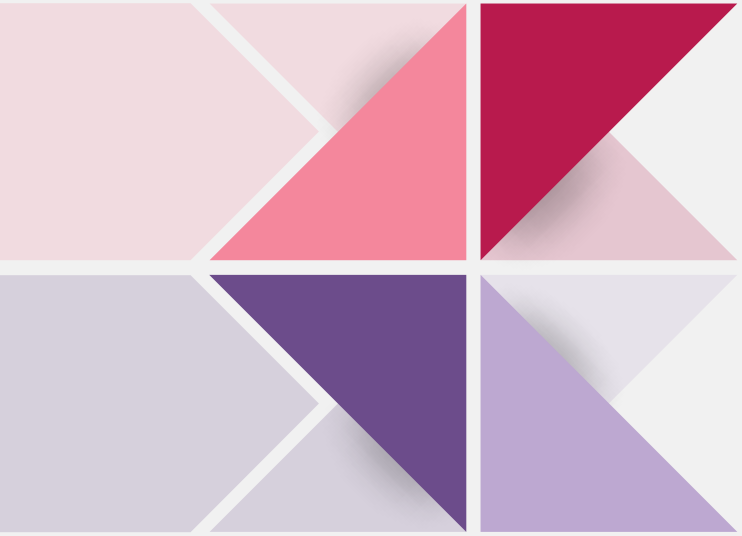
Bayescom

Mingrui Wei

02

Powerpoint is a complete presentation graphic package. It gives you everything you need to produce a professional-looking presentation

Index



01

本书作者

Martin Fowler

生于英国英格兰沃尔索尔，软件工程师，也是一个软件开发方面的著作者和国际知名演说家，专注于面向对象分析与设计，统一建模语言，领域建模，以及敏捷软件开发方法，包括极限编程。

作者生平与主要著作

马丁·福勒出生于英格兰沃尔索耳，在伦敦居住十年之后于1994年移居美国，居住在马萨诸塞州波士顿附近，梅尔萝丝市郊区。曾在玛丽女王文理学校接受中等教育。

1

马丁·福勒80年代初期开始从事软件工作，已写就五本软件开发方面的书籍（参见“主要著作”）。2000年3月，他成为ThoughtWorks（一个系统集成和顾问公司）的首席科学家。

福勒是敏捷联盟的成员，于2001年，同其他16名合著者一起协助创作了“敏捷软件开发宣言”。他负责维护一个bliki网站---一种blog和wiki的混合衍生物，他还使控制反转(Inversion of Control)之一的“依赖注入模式(Dependency Injection)”一词得到普及。

2

主要著作：《分析模式—可重用对象模型》、《规划极限编程》、《UML精粹—标准对象建模语言简明指南》(第三版)、《重构—改善既有代码的设计》、《企业应用架构模式》

重构的意义

在重构的时候，你会想到有句工程谚语吗：“如果它还可以运行，就不要动它。”

01. 重构的诞生

重构这个概念一开始来源于Smalltalk圈子，不久便进入其他语言的圈子中，由于重构但是框架开发中不可缺少的一部分，所以在框架开发人员在讨论自己的工作时，重构这个术语便诞生了

02. 重构的优点

作为程序员，我们知道代码被阅读和被修改的次数远远多于它被编写的次数。而重构的作用则是保持代码易读，易修改，它不仅适用于框架，也适用于一般软件。

03. 重构过程中问题（1）

软件开发过程中进行重构固然是极好，但是重构有风险。因为在重构过程中，必然会修改那些正在运作的程序，这可能会引入一些不易察觉的错误。如果重构方式错误，可能会毁掉你之前所有的成果。

03. 重构过程中问题（2）

而若是你在重构是不做好准备，不遵守规则，那么你在挖掘自己代码的时候会很快发现值得修改的地方，随着你挖掘的越深，找到重构的机会越多，你的修改也越多...最后你会发现你掉进自己挖的坑里爬不出来了，为了避免这种情况的出现，重构必须系统化进行。

重构的定义

总结

03

使用重构技术开发软件时，你把你的时间分配给两种截然不同的行为：添加新功能，以及重构。在添加新功能时你不应该修改既有代码，只管添加新功能，反之亦然。测试一般不应该添加，除非绝对必要。

重构（名词）

01

对软件内部结构的一种调整，目的是在不改变软件可观察行为前提下，提高其可理解性，降低其修改成本

重构（动词）

02

使用一系列重构手法，在不改变软件可观察行为前提下，调整其结构

重构可以达成的目的

重构不是万能的，但可以帮助你始终良好的控制好自己的代码，它可以用于达到以下几个目的

01

改进软件设计

如果没有重构，程序设计会逐渐腐败变质。随着需求的改动，我们只为短期目的，或在未完全整体设计时就去修改代码，程序会逐渐失去自己的结构，代码也越加难以理解。重构很像在整理代码，让代码回到它应该在的位置，维持其该有的形态

02

使得代码易于阅读理解

所谓程序设计，就是通过编写代码使得机器按照响应执行指令。而阅读你的代码的除了计算机之外，还有其他读者，他们才是重要的，易于理解的代码使得你和他们修改代码更加容易，同时也也会助于你理解那些不熟悉的代码，看到以前设计层面看不到的东西；

03

帮助找到bug

对代码重构，使得我们深入理解代码的行为，并且恰到好处的把新的理解反馈回去，在理解程序结构之时，也明白了自己所做的一些假设，以这样的思路去编码更容易有效的写出强健的代码

04

提高编码速度

重构一听起来很容易看出它能够提高编码质量，但是速度好像并不能体现出来。其实不然，拥有良好的设计才能做到快速开发。若没有，或许你在某段时间内进展迅速，但是之后你会发现再去修改代码时，需要花的时间会更多。

重构的时机

重构应该是任何时候都可以进行的不必特地拨出一些时间进行。

01

三次法则—-事不过三，三则重构

第一次去做某件事时只管去做；第二次去做类似的事或许有些反感，但无论如何还是可以去做；第三次再做类似的事，你就应该去重构了。

02

添加新功能时重构

这是最常见重构时机，重构的直接原因主要是为了帮助理解代码，另外一个原因就是原本的代码设计已无法帮助自己添加所需的特性。

03

修补错误时重构

调试过程中运用重构，多半是为了让代码更具可读性。在阅读理解代码的时，用重构帮助加深自己理解，这常常可以帮助我们找出bug。可以这样想：程序出现bug，这是程序代码还没清晰到让你一眼看出bug。

04

复审代码时重构

代码复审这个活动有助于在开发团队转播知识和帮助更多人理解软件整体系统，从而改善开发状况。自己写的代码或许自己清晰，而他人则不然，我们在一段时间里能想到的好的想法有限，但在代码复审时则更易得到别的好的想法

重构的难题

学习一种可以大幅提高生产力的新技术时，你总是难以察觉其不适应的场合。

01

数据库

绝大多数商用程序与它们背后的数据库结构紧密耦合在一起，这是数据库结构难以修改的原因之一。另一个原因就是数据迁移。

02

修改接口

关于对象的一件重要事情是：它们允许你修改软件模块的实现和接口。你可以安全的修改其内部实现而不影响他人，但是你修改了接口，任何事都有可能发生。而很多重构手法都会修改接口。

03

难以通过重构手法完成的设计改动

当由于需求原因使得重构难以完成设计改动，例如将不考虑安全性时构造的系统重构成具备良好安全性系统。这种情况下的办法可以是想象重构的情况，如果预想不到简单的重构方法就在设计上投入更多精力。

04

何时不该重构

有的时候，重构还不如重写所有代码。重构一般是代码在大部分情况都能正常运作的情况下进行的，反之则重写。另外，若是项目已近最后期限，也应该避免重构，因为已经没有时间啦。

重构与设计、性能

1

重构与设计

重构和设计时彼此互补的。

如果没有重构则必须保证设计正确无误，这个太难了，而若没有设计这也不是有效的途径，连极限编程的爱好者们也会进行预先设计，故将设计和重构结合在一起才是一个良好的软件编程途径，这样既能保证有一个可接受的设计方案，又能保证后续修改成本不在高昂。

2

重构与性能

为了让软件易于理解，常常将代码修改使得程序运行变慢

编写快速软件的思路：首先，写出可调的软件，然后调整它使得获得足够快的速度。编写快速软件的三种方法：时间预估法；持续关注法，性能提升法。

代码的坏味道

-1

决定何时重构、何时停止和知道重构机制如何运转一样重要。

1

重复代码(Duplicated Code)

如果在一处以上的地方看到相同的程序结构，那么可以肯定：设法将他们合而为一，程序会变得更好。

2

过长函数(Long Method)

程序越长越难理解，早期由于子程序调用需要额外开销，所以人们不乐意使用小函数。现代的OO语言几乎已经完全免除了进程内的函数调用的开销，所以现在拥有短函数的对象活的比较滋润。

3

过大的类(Large Class)

如果你想用单个类做太多事，其内往往就会出现太多的实例化变量。一旦如此，重复代码就会接踵而来。你可以运用Extract Class将几个变量一起提炼到新类中。

4

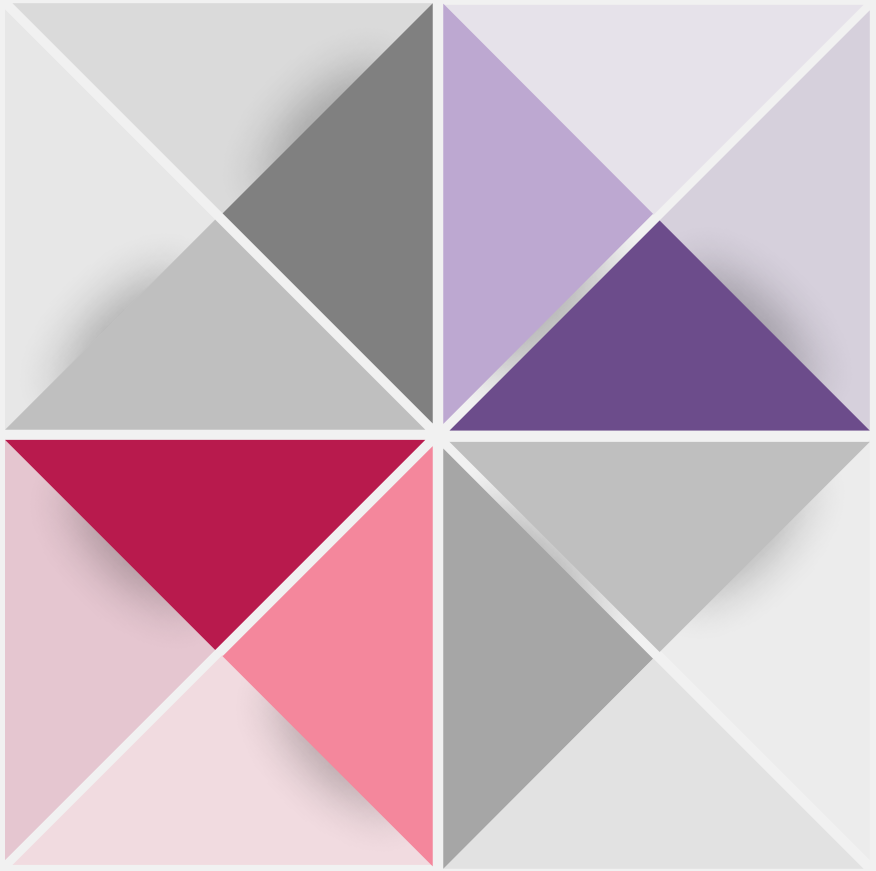
过长的参数列

因为过长的参数列难以理解，太多的参数会造成前后不一致、不易使用，一旦你需要更多的参数就不得不修改它。

代码的坏味道
-2

05
发散式变化(Divergent Change)
我们希望软件能够易于被修改，在需修改时，我们希望能够跳到系统的某一点，只在此处做修改，若不能做到，这时你就应该去重构了。

06
霰弹式修改(Shotgun Surgery)
与发散式变化相反，遇到某个变化，你必须在多各类里做多个小修改，这就是散弹式修改。这时你可以运用移动函数和移动文件来重构它



07
依恋情结(Feature Envy)
函数对某各类的兴趣高于对自己所处的类的兴趣。例如：调用了另一个类的大半的值或函数。

08
数据泥团(Data Clumps)
若你在很多地方看到相同的几项数据常常在一起，这时候你就可以将它们提炼到一个独立的对象中去。

代码的坏味道
-3

09

基本类型偏执

不愿意在小任务上运用小对象，常常直接使用基本类型数据。例如：由一个起始值和结束值组成的range类。

10

Switch Statements

面对对象程序的最明显的一个特征就是：少用switch语句。从本质上说，switch语句的问题在于重复。你常常会发现同样的swift语句散布于不同地方。你可以考虑使用多态来替换它。

11

平行继承体系(Parallel Inheritance Hierarchies)

Parallel Inheritance Hierarchies其实是Shotgun Surgery的特殊情况。在这种情况下，每当你为某个类增加一个子类，必须也为另一个类相应增加一个子类。

12

冗赘类 (Lazy Class)

如果一个类的所得不值其身价，它就应该消失。

代码的坏味道
-3

13

夸夸其谈未来性 (Speculative Generality)

如果函数或类的唯一用户是测试用例，这就是Speculative Generality。如果你发现这样的函数或类，请把它们连同其测试用例一并删掉。但如果它们的用途是帮助测试用例检测正当功能，当然就没必要删除了。

14

令人迷惑的暂时字段 (Temporary Field)

有时你会看到这样的对象：其内某个实例变量仅为某种特定情况而设。这样的 代码让人不易理解，因为你通常认为对象在所有时候都需要它的所有变最。在变量未被使用的情况下猜测当初其设置目的的，会让你迷惑的。

15

过度耦合的消息链 (Message Chains)

代码中你看到的可能是一长串getThis()或一长串临时变量。采取这种方式，意味客户代码将与查找过程中的导航结构紧密耦合。一旦对象间的关系发生任何变化，客户端就不得不做出相应修改。

16

中间人 (Middle Man)

你也许会看到某个类接口有一半的函数都委托给其他类，这样就是过度运用。这时应该使用Remove Middle Men,直接和真正负责的对象打交道。

代码的坏味道
-3

17

狎(xiá)昵关系(Inappropriate Intimacy)
有时你会看到两个类过于亲密，花费太多时间去探究彼此的private成分。继承往往造成过度亲密，因为子类对超类的了解总是超过后者的主观愿望。若你觉得该让这个类单独出来，请运用Replace Inheritance with Delegation让它离开继承体系。

18

异曲同工的类 Alternative Classes with Different Interfaces
如果两个函数做同一件事，却打着不同的签名，请运用Rename Method根据它们的用途重新命名。但这往往不够，请反复运用Move Method将某些行为 移入类，直到两者的协议一致为止。如果你必须重复的移入代码才能完成这些，或许可运用Extract Superclass解决。

19

不完美的库类(Incomplete Library Class)
库类构筑者没有未卜先知的能力，库往往构造得不够好，而且往往不可能让我们修改其中的类使它完成我们希望完成的工作。此时你只想修改库类的一两个函数,可以运用Introduce Foreign Method；如果想要添加一大堆额外行为，就得运用Introduce Local Extension。

20

纯稚的数据类 (Data Class)
所谓DataClass是指：它们拥有一些字段，以及用于访问（读写）这些字段的函数，除此之外一无长物,作为一个起点很好，但若要让它们像成熟的对象那样参与整个系统的工作，它们就必须承担一定责任。其对他们进行归类封装

代码的坏味道
-3

21

Resfused Bequest (被拒绝的遗赠)

子类应该继承超类的函数和数据。但如果它们不想或不需要继承，又该怎么办 呢？它们得到所有礼物，却只从中挑选几样来玩！可以利用继承来复用一些行为，并发现这可以很好地应用于日常工作。

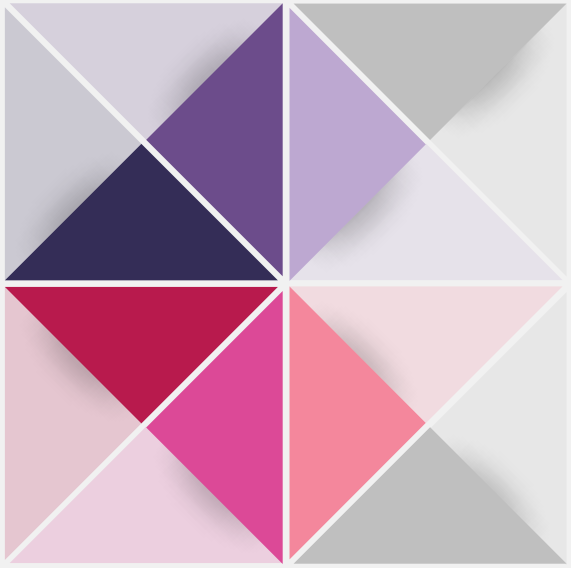
22

过多的注释 Comments

当你感觉需要撰写注释时，请先尝试重构，试着让所有注释都变得多余。注释除了用来记述将来的打算之外，注释还可以用来标记你并无十足把握的区域。

总结

结合个人实际中编码常遇到的情况的感觉：编写代码的时候，要定义好对像的特性及功能，不要让它做多余的事；如果发现自己想在程序中添加一个特性的时候，而代码结构使你无法方便的达成目的，那就先重构那个程序，使添加特性更加容易进行，然后在再添加特性；重构技术就是以微小的步伐进行修改程序，如果犯下错误也能轻松发现；命名很重要，毕竟代码只是在机器上运行而已，更主要的还是让人类阅读的，所以写出人类容易阅读理解的代码才是优秀的程序员。



Thank you