guisu，程序人生。 逆水行舟，不进则退。

能干的人解决问题。智慧的人绕开问题(A clever person solves a problem. A wise person avoids it)

重构与模式：改善代码三部曲中的第三部

标签： 设计模式 decorator command 算法 null 语言

2012-06-13 10:54 9106人阅读 评论(1) 收藏 举报

分类： 设计模式（25） 重构（10） 重构与模式

一、改善代码的三部曲

《设计模式》-> 《重构》-> 《重构与模式》。也就是设计->重构->重构曲新设计。

《设计模式》主要详细说明20几种模式，为我们带来了常见设计问题的经典解决方案，从而改变了整个面向对象开发的面貌。为设计而著。

《重构》改善既有代码的设计，总结了我们会用到的各种重构手法，为我们带来了一种改进代码的高效过程，从而彻底改变了面向对象设计的方式。侧重去除坏代码的味道。

《重构与模式》是设计模式相关的重构。模式不是设计出来的，是重构出来的。好的设计也不是设计出来的，是重构出来的。不要怕改变，只要改变得法，变就不再是灾难，而是进步的良机。侧重设计模式+重构手段。

在阅读重构与模式之前，最好熟读前面两本：《设计模式》和《重构》。

设计模式代表了传统的软件开发思想：好的设计会产生好的软件，因此在实际开发之前，值得花时间去做一个全面而细致的设计。重构代表了敏捷软件开发的浪潮：软件并不是在一开始就可以设计得完美无缺的，因此可以先进行实际开发，然后通过对代码不断的进行小幅度的修改来改善其设计。二者从不同角度阐述了设计的重要性。

有些人在编写任何代码之前，都要很早地为模式做计划，而有些人在编写了大量代码之后才开始添加模式。第二种使用模式的方式就是重构，因为是要在不增加系统特性或者不改变其外部行为的情况下改变系统的设计。有些人在程序中加入模式，只是因为觉得模式能够使程序更容易修改；更多人这样做只是为了简化目前的设计。如果代码已经编写，这两种情形都是重构，因为前者是通过重构使修改更容易，而后者则是通过重构在修改后进行整理。

虽然模式是在程序中能够看到的东西，但是模式也是一种程序转换。

重构是实现设计模式的一种手段，设计模式往往也是重构的目的。

二、重构与模式的缘由

应该通过重构实现模式、趋向模式和去除模式(refactoring to, towar
计中使用模式，也不再过早的在代码中加入模式。这技能避免过度设
1.过度设计：代码的灵活性和复杂性超出所需。有些开始设计的时候，
了某种设计模式预留扩展，但是后来却没怎么动，也就是导致了废
2.设计不足

产生设计不足的原因：

1）程序员没有时间，没有抽出时间，或者时间不允许进行重构

2）程序员在何为好的软件设计方面知识不足

3）程序员被要求在既有系统中快速的添加新功能

4）程序员被迫同时进行太多项目

长期的设计不足，会使软件开发节奏变成“快，慢，更慢”，可能的后果是：

1.0版本很快就交付了，但是代码质量很差

2.0版本也交付了，但质量低劣的代码使我们慢下来

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

到了4.0版本时或者之后，我们意识到这样肯定不行，开始考虑推倒重来

3.测试驱动开发和持续重构。

测试驱动开发和持续重构提供了一种精益、迭代和训练有素的编程风格，能够最大程度的有张有弛，“迅速而又从容不迫”

使用测试驱动开发和持续重构的益处：

1）保持较低的缺陷数量

2）大胆的进行重构

3）得到更加简单、更加优秀的代码

4）编程时没有压力

模式和重构之间存在着天然联系，模式是你想达到的目的地，而重构则是从其他地方抵达这个目的地的条条道路。

4.演进式设计

演进式设计即趋向性设计，主要是避免过度设计。

通过重构产生设计结构，也就是通过重构实现模式或者重构趋向模式。为设计而设计的思路并不适合大项目，循序渐进从重构到设计模式才是设计模式的王道。

敏捷开发中经常采用的演进式架构设计：

很多程序员可能都遇见过这种事：某块代码亟待修改，却没有人愿意接手。为什么会这样？这段代码正巧是两个组件间的接口，修改工作太过困难。而在演进式设计中，我们常常会做这种修改。代码应当是“活的”并且是“可生长”的，决不能无视强烈的变化需求 而保持一成不变。正因为如此，演进式设计可以提高设计质量，进而提高整个系统的质量。

第6章创建

6.1 用Creating Method替换构造函数

当类中有多个构造函数，因此很难决定在开发期间用哪一个时，可以用能够说明意图的返回对象实例的Creation Method替换构造函数

动机：

Creation Method——类中的一个静态或者非静态的负责实例化类的新实例方法。因Creating Method命名没有限制，所以可以取最能表达所创建对象的名字。

类中有太多构造函数→提炼类或者提炼子类 或者 用Creation Method替换构造函数来澄清构造函数的意图

优缺点：

+ 比构造函数能够更好的表达所创建的实例种类

+ 避免了构造函数的局限，比如两个构造函数的参数数目和类型不能相同

+ 更容易发现无用的创建代码

- 创建方式是非标准的，有的用new实例化，而有的用Creation Method实例化

变体：

不需要为每个对象的配置都设立一个Creation Method，非必要情况下可以添加参数来减少Creation Method的数量

当Creation Method过多的分散了类的主要职责是，应该考虑将相关的Creation Method重构为一个Factory

6.2 将创建知识搬移到Factory

当用来实例化一个类的数据和代码在多个类中到处都是时，可以讲有关创建的知识搬移到一个Factory中

动机：

创建蔓延——将创建的职责放在了不应该承担对象创建任务的类中，是解决方案蔓延中的一种，一般是之前的设计问题导致。

使用一个Factory类封装创建逻辑和客户代码的实例化选项，客户可以告诉Factory实例如何实例化一个对象，然后用同一个Factory实例在运行时执行实例化。

Factory不需要用具体类专门实现，可以使用一个接口定义Factory，然后让现有的类实现这个接口。

关闭

如果Factory中创建逻辑过于复杂，应将其重构为Abstract Factory，客户代码可以配置系统使用某个ConcreteFactory（AbstractFactory的一个具体实现）或者默认的ConcreteFactory。

只有确实改进了代码设计，或者无法直接进行实例化时才有足够的理由进行Factory重构

优缺点：

+ 合并创建逻辑和实例化选项
+ 将客户代码与创建逻辑解耦
- 如果可以直接实例化，会使设计复杂化

6.3 用Factory封装类

当直接实例化处在同一包结构中、实现统一接口的多个类。可以把类的构造函数声明为非公共的，并通过Factory来创建它们的实例

动机：

可以通过Factory将一组客户并不需关心的子类屏蔽到包内部。

如果类共享一个通用的公共接口、共享相同的超类、并且处在同一包结构中，该重构可能有用。

优缺点：

+ 通过意图导向的Creation Method简化了不同种类实例的创建
+ 通过隐藏不需要公开的类减少了包的“概念重量”
+ 帮助严格执行“面向接口编程，而不是面向实现”这一格言
- 当需要创建新种类的实例时，必须更新Creation Method
- 当客户只能获得Factory的二进制代码而无法获得源码时，对Factory的定制将受到限制

6.4 用Factory Method引入多态创建

当一个层次中的类都相似的实现一个方法，只是对象创建的步骤不同时，可以创建调用Factory Method来处理实例化方法的唯一超类版本

动机：

Factory Method是OOP中最常见的模式，因其提供了多台创建对象的方法

使用Factory Method后的代码往往比在类中赋值方法来创建自定义对象要简单

使用Factory Method的主要情况：

当兄弟子类实现了除对象创建步骤外都很相似的方法时

当超类和子类实现了除对象创建步骤外都很相似的方法时

优缺点：

+ 减少因创建自定义对象而产生的重复代码
+ 有效的表达了对象创建发生的位置，以及如何重写对象的创建
+ 强制Factory Method使用的类必须实现统一的类型
- 可能会向Factory Method的一些实现者传递不必要的参数

6.5 用Builder封装Composite

当构造Composite是重复的、复杂的且容易出错的工作时，通过使用Builder处理构造细节来简化构造过程。

动机：

构造Composite是重复的、复杂的、容易出错的工作，通过使用Builder处理构造细节来简化构造过程

Builder模式很擅长处理繁重的、复杂的构造步骤。

Builder模式的意图：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

优缺点：

+ 简化了构造Composite的客户代码
+ 减少了创建Composite的重复和易出错的本性
+ 在客户代码和Composite之间实现了松耦合
+ 允许对已封装的Composite或复杂对象创建不同的表示
- 接口可能不会很清楚的表达其意图

6.6 内联Singleton

当代码需要访问一个对象，但是不需要对象的全局入口时，可以把Singleton的功能搬移到一个保存并提供对[illegible]问入口的类中。删除Singleton。

动机：

Singleton意图：确保一个类仅有一个实例，并提供一个访问它的全局访问点

保持暴露对象和保护对象之间的平衡对维护系统的灵活性是至关重要的

任何全局数据在被证明是无害之前都是有害的

如果遇到本不该实现为Singleton的Singleton，不要犹豫，内联它！

优缺点：

+ 使对象的协作变得更明显和明确
+ 保护了单一的实例，且不需要特殊的代码
- 当在许多层次间传递对象实例比较困难的时候，会使设计变得复杂

第7章 简化

我们所编写的绝大部分代码都不会从一开始就很简单。
算法经常会因为支持多种变化而变得复杂。
控制状态转换的转换的逻辑往往会变得越来越复杂。

7.1 细合方法

当你无法迅速的理解一个方法的逻辑时，把方法的逻辑转换成几个同一层面上的、能够说明意图的步骤。

动机：

Composed Method由对其他方法的调用组成，好的Composed Method的代码都在细节的同一层面上。
Composed Method一般不会引入性能问题

优缺点：

+ 清晰的描述了一个方法所实现的功能以及如何实现
+ 把方法分解成命名良好的、处在细节的同一层面上的行为模块，以此来简化方法
- 可能会产生过多的小方法
- 可能会使调试变得困难，因为程序的逻辑分散在许多小方法中

Composed Method指导原则：

Composed Method都很小。一般在5行左右，很少超过10行
删除重复代码和死代码。除去明显的和微妙的代码重复，除去没有被使用的代码，以减少方法的代码量
表达意图。清楚的命名程序中的变量、方法和参数，使它们明确表达意图。
简化。转换代码，使它尽可能简单。
使用细节的统一层面。当把一个方法分解成一组行为时，要保证这些行为在细节的相似层面上。

7.2 用Strategy替换条件逻辑

当方法中条件逻辑控制着应该执行计算的哪个变体时，为每个变体创建一个Strategy并使方法把计算委托到Strategy实例。

动机：

——为算法的各个变体生成一系列的类，并用Strategy的一个实例装配主类，主类在运行时委托到该Strategy实例
复杂的条件逻辑是最常导致复杂度上升的地点之一

优缺点：

+ 通过减少或去除条件逻辑使算法变得清晰易懂
+ 通过把算法的变体搬移到类层次中简化了类
+ 允许在运行时用一种算法替换另一种算法
- 当应用基于继承的解决方案或“*简化条件表达式*”中的重构更简单时，会增加设计的复杂度
- 增加了算法如何获取或接收上下文类数据的复杂度

7.3 将装饰功能搬移到Decorator

当代码向类和核心职责提供装饰功能时，可以考虑将装饰代码搬移到Decorator
无论多么喜欢一个模式，不要在不必要的时候使用它

优缺点：

+ 把装饰功能从类中移除，从而简化类
+ 有效的把类的核心职责和装饰功能区分开来
+ 可以去除几个相关类中重复的装饰逻辑
- 改变了被装饰对象的类型
- 会使代码变得更难理解和调试
- 当Decorator组合产生负面影响的时候，会增加设计的复杂度

7.4 用State替换状态改变条件语句

当控制一个对象状态转换的条件表达式过于复杂时，可以考虑用处理特殊状态转换的State类替换条件语句

优缺点：

+ 减少或去除状态改变条件逻辑
+ 简化了复杂的状态改变逻辑
+ 提供了观察状态改变逻辑的很好的鸟瞰图
- 当状态转换逻辑已经易于理解的时候，会增加设计的复杂度

7.5 用Composite替换隐含树

当用原生表示法隐含的形成了树结构时，可以考虑用Composite替换这个原生表示法

关闭

优缺点：
+ 封装重复的指令，如格式化、添加或删除结点
+ 提供了处理相似逻辑增长的一般性方法
+ 简化了客户代码的构造职责
- 当构造隐式树更简单的时候，会增加设计的复杂度

7.6 用Command替换条件调度程序

当条件逻辑用来调度请求和执行操作时，为每个动作创建一个Command。把这些Command存储在一个集合中，并用获取及执行Command的代码替换条件逻辑。

为每个动作创建一个Command，把这些Command存储在一个集合中，并用获取及执行Command的代码替换条件逻辑

优缺点：
+ 提供了用统一方法执行不同行为的简单机制
+ 允许在运行时改变所处理的请求，以及如何处理请求
+ 仅仅需要很少的代码实现
- 当条件调度程序已经足够的时候，会增加设计的复杂度

第8章 泛化

泛化是把特殊代码转换成通用目的代码的过程。泛化代码的产生往往的重构的结果。

8.1 形成Template Method

当子类中的两个方法以相同的顺序执行相似的步骤，但是步骤并不完全相同。通过把这些步骤提取成具有相同签名的方法来泛化这两个方法，然后上移这些泛化方法，形成Template Method。

优缺点：
+ 通过把不变行为搬移到超类，去除子类中的重复代码
+ 简化并有效的表达了一个通用算法的步骤
+ 允许子类很容易的定制一个算法
- 当为了生成算法，子类必须实现很多方法的时候，会增加设计的复杂度

8.2 提取Composite

当一个类层次结构中的多个子类实现了同一个Composite时，可以提取一个实现该Composite的超类

优缺点：
+ 去除重复的类存储逻辑和类处理逻辑
+ 能够有效的表达类处理逻辑的可继承性

8.3 用Composite替换一/多之分

当类使用不同的代码处理单一对象与多个对象时，用Composite能够产生既可以处理单一对象又可以处理多个对象的代码

优缺点：
+ 去除与处理一个或多个对象相关联的重复代码
+ 提供处理一个或多个对象的统一方法
+ 支持处理多个对象的更丰富的方法
- 可能会在Composite的构造过程中要求类型安全的运行时检查

8.4 用Observer替换硬编码的通知

当子类通过硬编码来通知另一个类的实例时可以去除这些子类，并使其超类能够通知一个或多个实现了Observer接口的类

关闭

优缺点：
+ 使主题及其观察者访问松散耦合
+ 支持一个或多个观察者
- 当硬编码的通知已经足够的时候，会增加设计的复杂度
- 当出现串联通知的时候，会增加代码的复杂度
- 当观察者没有从它们的主题中被删除的时候，可能会造成资源泄漏

8.5 通过Adapter统一接口

当客户代码与两个类交互，其中的一个类具有首选接口，可以用一个Adapter统一接口

动机：

当下面条件都为真时，重构Adapter就是有用的：

两个类所做的事情相同或相似，但是具有不同的接口

如果类共享同一个接口，客户代码会更简单、更直接、更紧凑

无法轻易改变其中一个类的接口，因为它是第三方库中的一部分，或者它是一个已经被其他客户代码广泛使用的框架的一部分，或者无法获得源码

优缺点：

+ 使客户代码可以通过相同的接口与不同的类交互，从而去除或减少重复代码
+ 使客户代码可以通过公共的接口与多个对象交互，从而简化了客户代码
+ 统一了客户代码与不同类的交互方式
- 当类的接口可以改变的时候，会增加设计的复杂度

8.6 提取Adapter

当一个类适配了多个版本的组件、类库、API或其他实体时，可以为组件、类库、API或其他实体的一个Adapter

Adapter用来适配对象，Facade用来适配整个系统，Facade通常用来与遗留系统进行交互

优缺点：

+ 隔离了不同版本的组件、类库或API之间的不同之处
+ 使类只负责适配代码的一个版本
+ 避免频繁的修改代码
- 如果某个重要行为在Adapter中不可用的话，那么客户代码将无法执行这一重要行为

8.7 用Interpreter替换隐式语言

当类中的许多方法组合成了一种隐式语言的元素，可以为隐式语言的元素定义类，这样就可以通过类实例组合，形成易于理解的表达式

优缺点：

+ 比隐式语言更好的支持语言元素的组合
+ 不需要解析新的代码来支持语言元素的新组合
+ 允许行为的运行时配置
- 会产生定义语言和修改客户代码的开销
- 如果语言很复杂，则需要很多的编程工作
- 如果语言本身就很简单，则会增加设计的复杂度

第9章 保护

9.1 用类替换类型代码

字段的类型无法保护它免受不正确的复制和非法的等同性比较，可以把字段的类型声明为类，从而限制复制和等同性比较

优缺点：

+ 更好的避免非法赋值和比较
- 比使用不安全类型要求更多的代码

9.2 用Singleton限制实例化

代码创建了一个对象的多个实例，并导致内存使用过多和系统性能下降时，可以用Singleton替换多个实例

不要做不成熟的代码优化，经过不成熟优化的代码比未优化的代码更难于重构。在代码优化之前，你会发现更多可以改进的地方

关闭

优缺点：

+ 改进性能
- 在任何地方都可以很容易的访问。在很多情况下，这可能是设计的缺点
- 当对象含有不能共享的状态时，本重构无效

9.3 引入Null Object

当代码中到处都是处理null字段或变量的重复逻辑时，将null逻辑替换为一个Null Object，一个提供正确null行为的对象

优缺点：

+ 不需要重复的null逻辑就可以避免null错误

+ 通过最小化null测试简化了代码
- 当系统不太需要null测试的时候，会增加设计的复杂度
- 如果程序员不知道Null Object的存在，就会产生多余的null测试
- 使维护变得复杂，拥有超类的Null Object必须重写所有新继承到的公共方法

第10章 聚集操作

10.1 将聚集操作搬移到Collecting Parameter

有一个很大的方法将信息聚集到一个局部变量中时，可以把结果聚集到一个Collecting Parameter中，并将它传入被提炼出的方法中

优缺点：

+ 帮助我们把很大的方法转换成更小的，更简单的多个方法
- 使结果代码运行得更快

10.2 将聚集操作搬移到Visitor

有一个方法从不同的类中聚集信息，可以把聚集工作搬移到一个能够访问每个类以便采集信息的Vis

优缺点：

+ 调节多个算法，使其适用于不同的对象结构
+ 访问相同或不同继承结构中的类
+ 调用不同类上的类型特定方法，无需类型转换
- 当可以使用通用接口把互不相同的类变成相似类的时候，会增加代码的复杂度
- 新的可访问类需要新的接收方法，每个Visitor中需要新的访问方法
- 可能会破坏访问类的封装性

第11章 使用重构

11.1 链构造函数

有很多包含重复代码的构造函数时，可以把构造函数链接起来，从而获得最少的代码重复。

11.2 统一接口

当需要一个与其子类具有相同接口的超类或接口时，可以找到所有子类含有而超类没有的公共方法，把这些方法复制到超类中，并修改每个方法，使其执行空行为。

11.3 提取参数

当一个方法或构造函数将一个字段赋值为一个局部实例化的值时，可以把赋值声明的右侧提取到一个参数中，并通过客户代码提供的参数对字段进行赋值。

主题推荐 color 重构 rgb

猜你在找

PHP面向对象设计模式
C语言系列之 进程通讯与相关设计模式
C语言系列之 快速排序与全排列算法
C语言系列之 数组与算法实战
C语言系列之 字符串相关算法

游戏项目中运用到的设计模式二策略模式strategy《重
Convert jQuery RGB output to Hex Color
COLORREF和COLOR和RGB和CString的转化总结
GdiPlusGdi+ 的Color和Gdi 的RGB宏的区别
RGB "Bayer" Color and MicroLenses

查看评论

4楼 shiyuzh2007 2012-08-02 14:22发表

暂时还是看得不太懂，感觉挺好。

3楼 lixiao0320 2012-06-21 14:06发表

暂还不能完全看懂，但觉的不错。。。

2楼 dyx_crazy 2012-06-20 16:40发表

先顶下吧。。。这三部曲给我感觉是。。十一部曲。。看的眼花 菜鸟的我实在是看不太明白。。

1楼 猫儿爷爷 2012-06-19 18:37发表

太长了.. 懒得看下去了 帮你顶下吧.... 顺便收藏下 后期有需要在翻出来看看

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap

关闭