7月读书笔记与小结

7月读书内容：

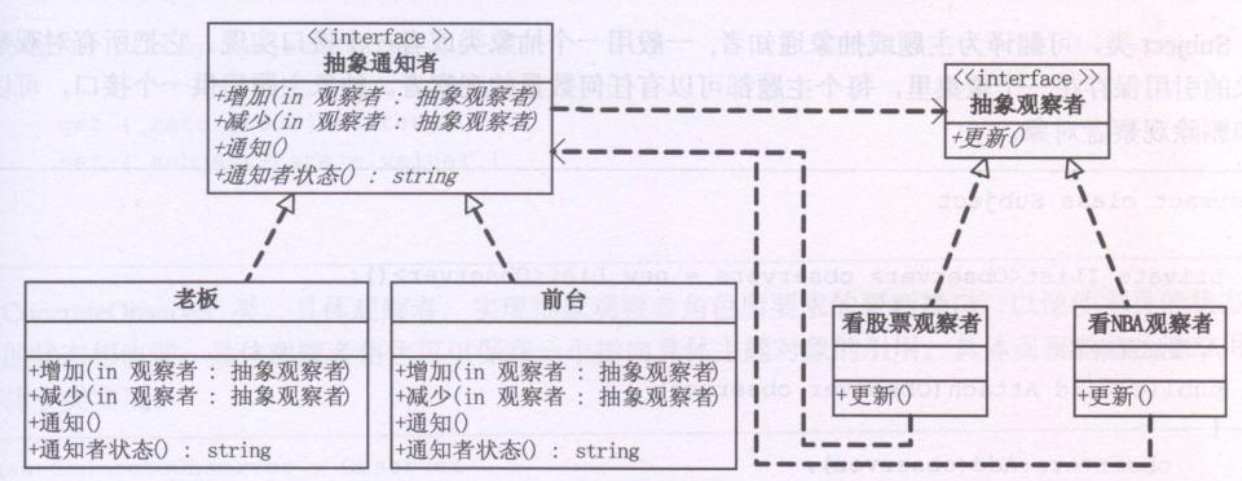
1. 《大话设计模式》第14，15，17章，掌握观察者，抽象工厂方法模式和适配器模式常用设计模式，通过示例理解实际的使用和结合项目中的应用，理解模式的优缺点，能在日后开发中灵活运用。
2. 《重构：改善既有代码的设计》第8，9，10章，了解重构得搬移特性，重现组织数据和简化条件逻辑等方法，结合实际工作，优化项目代码，提升性能和数据规范，并且能够在日后的开发中避免“坏味道”的代码。

一．《大话设计模式》读书笔记：

1.观察者模式

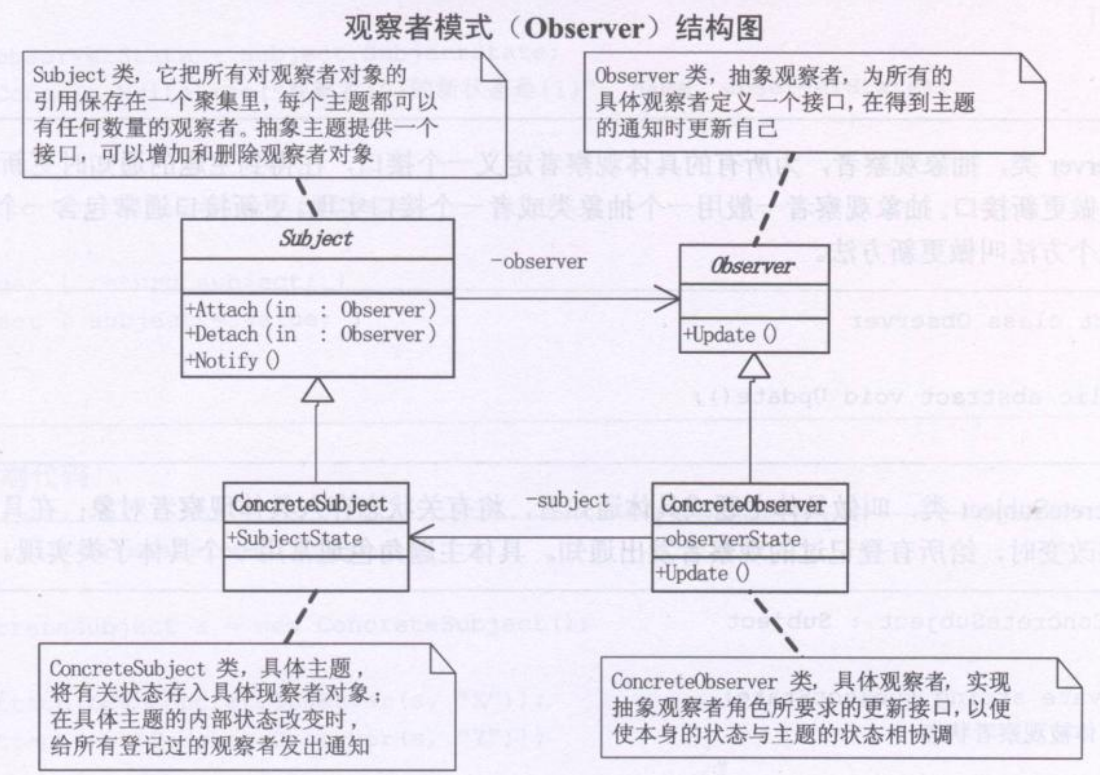
定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这个主题对象在状态发生变化时，会通知所有观察者对象，使他们能偶自动更新自己。

文中给出“老板回来了，我不知道”的示例，前台秘书就是一个观察者，老板回来之后，负责通知相关同事，这就是一个观察者所做的事，示例架构：



刚开始看到示例时，我的想法也是将秘书注册为观察者，用来观察老板的动向，直接定义一个观察者类，作为老板与同事之间的桥梁，后来发现，如果同事类中不一定都是看股票的，还有看球赛，F1赛车等等，这样，前台观察者类就不能仅仅发出老板回来这样的通知，还要发出球赛，赛车等信息，让同事类分别去做想做的事，这样，我之前的想法就欠耦合，需要将前台观察者类耦合出来，出一个抽象观察这类，因此有了上面的框架。

观察者模式架构图：



Subject类，为主题或抽象通知者，一般用一个抽象类或者一个接口实现，它把所有对观察者对象的引用保存在一个聚集里，每个主题都可以有任何数量的观察者，抽象主题提供一个接口，可以增加和删除观察者对象。

Observer类，抽象观察者，为所有的具体观察者定义一个接口，在得到主题的通知时更新自己，这个接口叫做更新接口，抽象观察者一般用一个抽象类或者一个接口实现，更新接口通常包含一个Update()方法，这个方法叫做更新方法。

ConcreteSubject类，叫做具体主题或者具体通知者，将有关状态存入具体观察者对象，在具体主题的内部状态改变时，给所有登记过的观察者发出通知，具体主题角色通常用一个具体子类实现。

ConcreteObserver类，具体观察者，实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调，具体观察者角色可以保存一个指向具体主题对象的引用，具体观察者角色通常用一个具体子类实现。

观察者优点：

1. 将一个系统分割成一系列相互协作的类有一个很不好的副作用，那就是需要维护相关对象间的一致性，我们不希望为了维持一致性而使各类紧密耦合，这样会给维护，扩展和重用带来不便。
2. 当一个对象的改变需要同时改变其他对象的时候
3. 不知道具体由多少对象有待改变时，应该考虑使用观察者模式
4. 一个抽象模型有两个方面，其中一个方面依赖于另一个方面，这时使用观察者模式可以将两者封装在独立的对象中使它们各自独立地改变和复用。
5. 观察者模式所做的工作就是解除耦合，让耦合的双方都依赖于抽象，而不是依赖于具体，从而使得各自得变化都不会影响另一边得变化。

观察者的缺点：

  1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

使用场景：

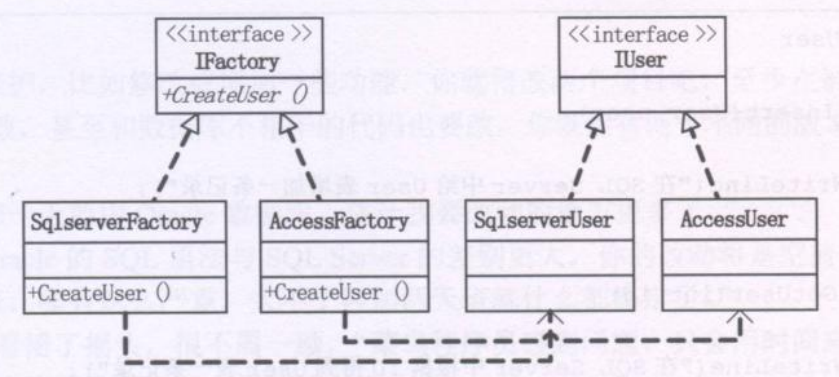
* 一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。
* 一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。
* 一个对象必须通知其他对象，而并不知道这些对象是谁。
* 需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制。

2.抽象工厂方法模式

工厂方法模式是定义一个用于创建对象的接口，让子类决定实例化哪一个类。

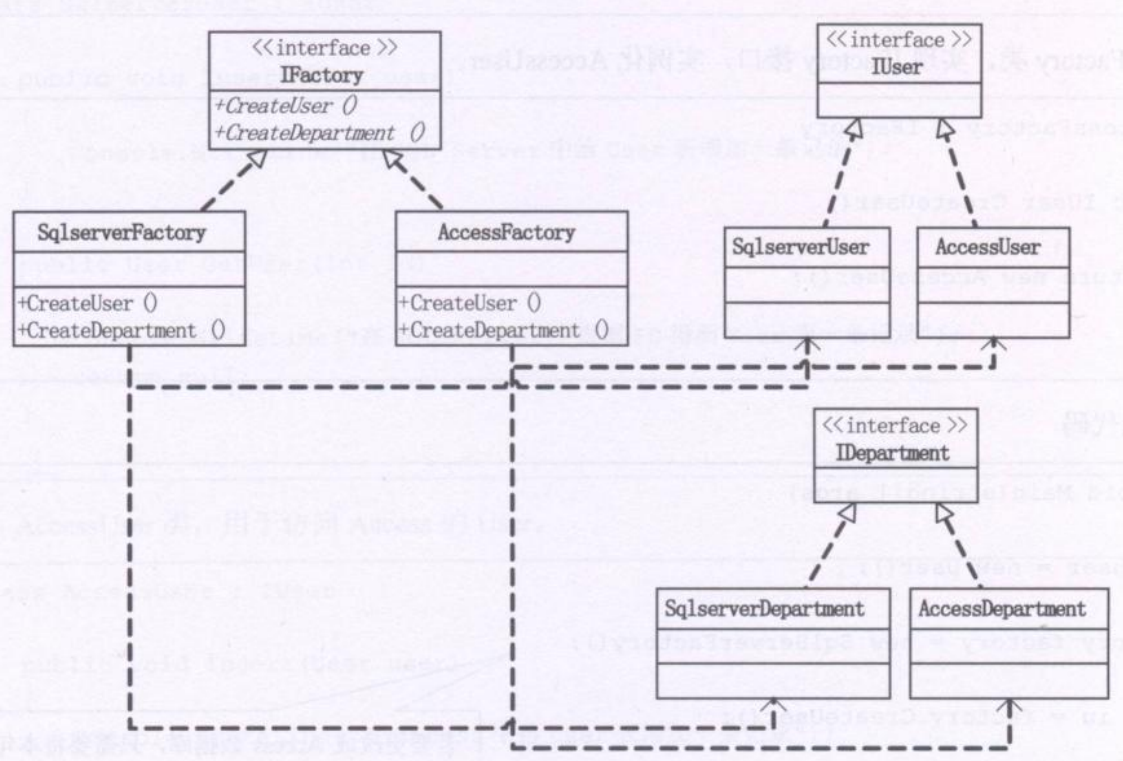
抽象工厂方法模式提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

示例中工厂模式数据访问程序实现结构图：



这样的设计缺陷是，数据库里不可能只有User表，如果增加其他部门表，就不能体现其灵活性。

抽象工厂方法模式实现结构图：



IDepartment接口，用于客户端访问，解除与具体数据库访问的耦合。

SqlserverDepartment类，用于访问SQL Server的Department

AccessDepartment类，用于访问Access的Department

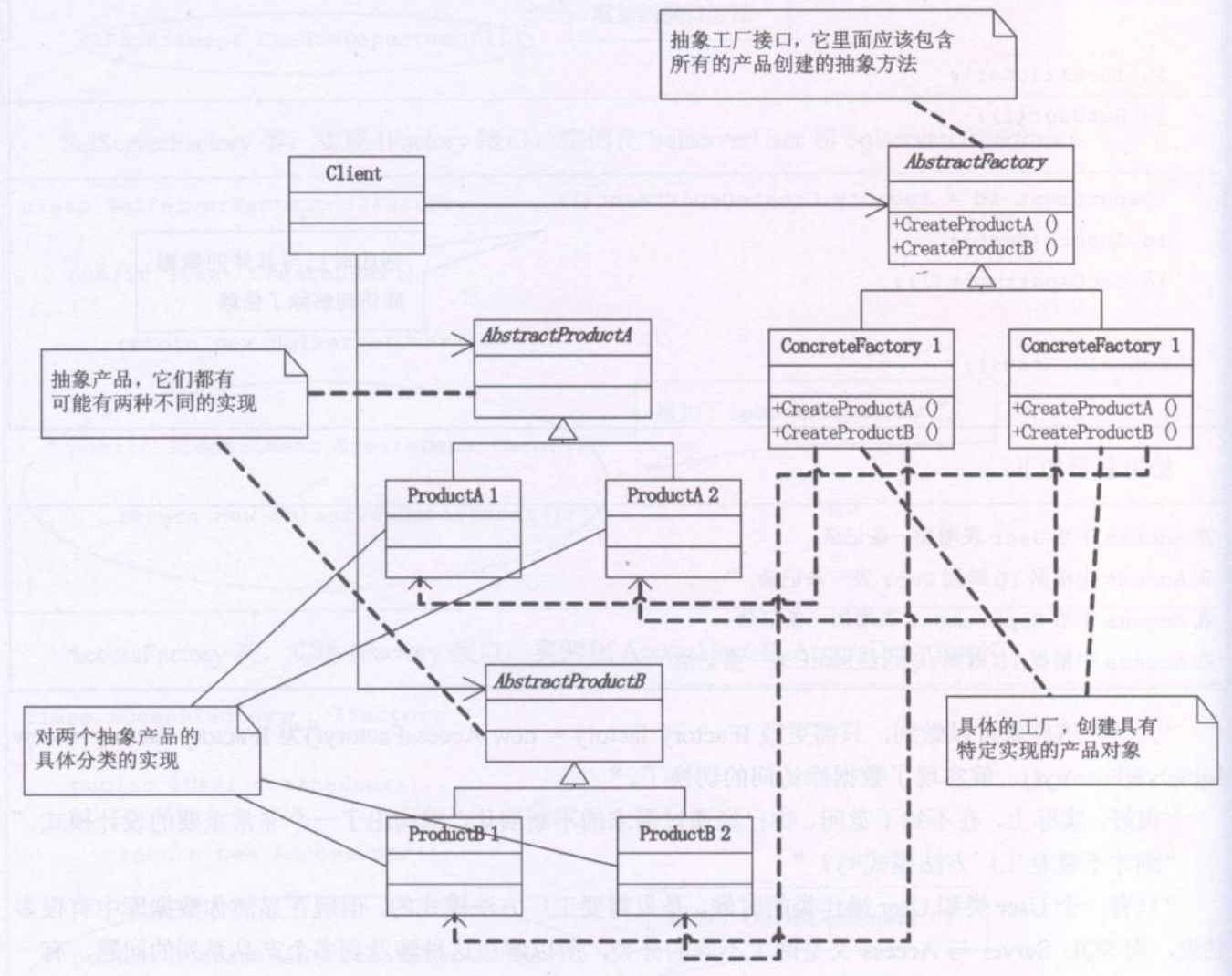
IFactory接口，定义一个创建访问User表对象的抽象的工厂接口

SqlServerFactory类，实现IFactory接口，实例化SqlserverUser和SqlserverDepartment

AccessFactory类，实现IFactory接口，实例化AccessUser和AccessDepartment

因此：只有一个User类和User操作类的时候，只需要工厂方法模式，但是数据库中有很多的表时，SQL Server与Access又是两大不同的分类，解决这样的涉及多个产品系列的问题，抽象工厂方法模式发挥重大作用。

抽象工厂方法模式结构图：



AbstractProductA 和 AbstractProductB是两个抽象产品，可以有不同的实现，ProductA1, ProductA2, ProductB1, ProductB2就是对两个抽象产品的具体分类实现

IFactory是一个抽象工厂接口，它里面应该包含所有的产品创建的抽象方法，而ConcreteFactory1和ConcreteFactory2就是具体的工厂，通常在运行时刻再创建一个ConcreteFactory类的实例，这个具体的工厂再创建具有特定实现的产品对象，为创建不同的产品对象，客户端使用不同的具体工厂。

抽象工厂方法模式优点：

1. 便于交换产品系列，在一个应用中只需要在初始化的时候出现一次，使得改变一个应用的具体工厂变的非常容易，它只需要改变具体工厂即可使用不同的产品配置。
2. 让具体的创建实例过程与客户端分离，客户端是通过它们的抽象接口操纵实例，产品的具体类名被具体工厂的实现分离，不会出现在客户代码中。

抽象工厂方法模式缺点：

产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码.

使用场景：

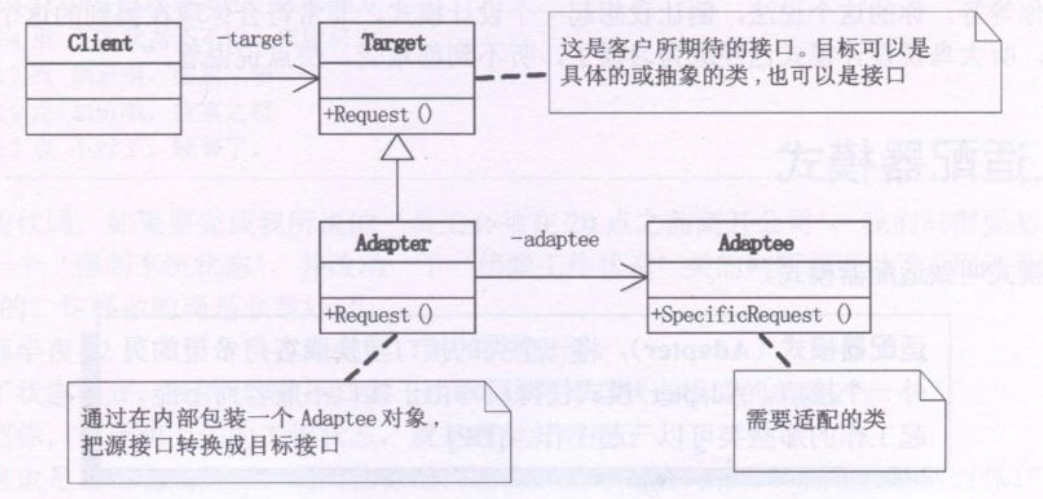
1、QQ 换皮肤，一整套一起换。

2、生成不同操作系统的程序。

1. 适配器模式

将一个类的接口转换成客户希望的另外一个接口，Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

适配器模式框架图：



适配器模式使用时机：

1. 使用一个已经存在的类，如果它的接口即它的方法和你的要求不同时时，可以考虑使用适配器模式
2. 两个类所做的事情相同或者相似，但是具有不同的接口时要使用适配器模式

适配器模式优点：

1. 客户代码可以统一调用同一个接口
2. 可以更简单，更直接，更紧凑
3. 可以让任何两个没有关联的类一起运行。
4. 提高了类的复用。
5. 增加了类的透明度。
6. 灵活性好。

适配器模式缺点：

1、过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

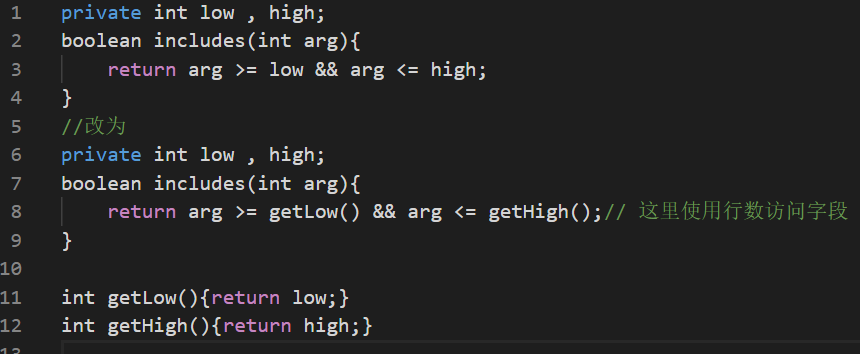
2.由于 JAVA 至多继承一个类，所以至多只能适配一个适配者类，而且目标类必须是抽象类。

二．《重构：改善既有代码的设计》读书笔记

1.重新组织数据

a.自封装字段

直接访问一个字段，但与字段之间 的耦合关系逐渐变得笨拙。为这个字段建立取值和设值函数，并且通过函数来访问字段,即get/set方法访问字段。



动机：

在“字段访问方式”上，有两种截然不同的观点：

* 第一种：在该变量的所在的类中，你可以自由访问它，直接访问。
* 第二种：即使在这个类中你也应该使用访问函数间接访问，间接访问。

两种访问各自的好处：

* 第一种：直接访问，代码比较容易阅读，不需要停下来说：这是一个取值函数。
* 第二种：间接访问，子类可以通过覆写一个函数而改变获取数据的途径；还支持更加灵活的数据管理方式，例如延迟初始化（意思是：只有在需要用到某值时，才对它初始化）。

做法：

比较简单，就是为字段建立取值和设值函数，然后检查引用点。

b.封装字段

你的类中有一个public字段，将它申明为private ，并提供访问函数（对应的get和set），即使用定义一个私有字段，生成get/set方法

1. 以对象取代数组

一个数组，其中的各个元素各自代表不同的东西。以对象取代数组，对于数组中的每个元素，以一个元素来表示。

动机：

数组是一种常见的用以组织数据的结构，它们应该用于“以某种顺序容纳一组相似对象”。有时候会发现一个数组容纳了多种不同的对象。如人们很难记住数组的第一个是人名这样的约定。

做法：

新建一个类，定义和数组中每个元素类型相当的字段。 最后去掉该数组。

d．以字段常量去掉魔法数

创建一个常量，根据其意义为它命名，将字面数值替换为这个常量。

动机：

魔法数是历史最悠久的不良现象。 根据名称魔法二字就可以知道，这是有魔力的，说明这个魔法数有着特殊的含义，却又不能明确表现这种意义。

做法：

声明一个常量，令其值为原本的魔法数值，然后找到这个魔法数的所有引用点，替换！

这个是重构的手法，更是能反应写程序这个人的水平。

2.简化条件表达式

a.Decompode Conditional 分解条件表达式  
你有一个复杂的条件语句（if-then-else）。  
从if、then、else三个段落中分别提炼出独立函数。

b.Consolidation Conditional Expression 合并条件表达式

你有一系列条件测试，都得到相同的结果。  
将这些测试合并为一个条件表达式，并将这个条件表达式提炼成为一个独立函数。

c.Consolidate Duplicate Conditional Fragments 合并重复的条件片段

在条件表达式的每个分支上有着相同的一段代码。  
将这段重复代码移动到条件表达式之外。

c.Remove Control Flag 移除控制标记

在一系列布尔表达式中，某个变量带有“控制标记”的作用。  
以break语句或return语句取代控制标记。

d.Replace Nested Conditional with Guard Clauses 以哨兵模式取代嵌套条件表达式

函数中的条件逻辑使人难以看清正常的执行路径。  
使用哨兵模式表现所有特殊情况。

e.Replace Conditional with Polymorphism 以多态取代条件表达式

你手上有个条件表达式，它根据对象类型的不同而选择不同的行为。  
将这个条件表达式的每一个分支放进一个子类内的覆写函数中，然后将原始函数声明为抽象函数。

f.Introduce Null Object 引用Null对象

你需要再三检查某对象是否为Null。  
将null值替换为null对象。

g.Introduce Assertion 引入断言

某一段代码需要对程序状态做出某种假设。  
以断言明确表示这种假设。

3.简化函数调用

a.Rename Method 函数改名

函数的名称未能揭示函数的用途。  
修改函数名称。

b.Add Parameter 添加参数

某个函数需要从调用端得到更多信息。  
为此函数添加一个对象参数，让该对象带进函数所需信息。

c.Remove Parameter 移除函数

函数本体不再需要某个参数。  
将该参数去除。

d.Separate Query from Modifier 将查询函数和修改函数分离

某个函数既返回对象状态值，又修改对象状态。  
建立两个不同的函数，其中一个负责查询，另一个负责修改。

e.Parameterize Method 令函数携带参数

若干函数做了类似的工作，但在函数本体中却包含了不同的值。  
建立单一函数，以参数表达那些不同的值。

f.Replace Parameter with Explicit Methods 已明确函数取代参数

你有一个函数，其中完全取决于参数值而采取不同行为。  
针对该参数的每一个可能值，建立一个独立函数。

g.Preserve Whole Object 保持对象完整

你从某个对象中取出若干值，将它们作为某一次函数调用时的参数。  
改为传递整个对象。

h.Replace Parameter with Methods 以函数取代参数

对象调用某个函数，并将所得结果作为参数，调用另外一个函数。而接受该参数的函数本身也能够调用前一个函数。  
让参数接受者去除该项参数，并直接调用前一个函数。

i.Introduce Parameter Object 引用参数对象

某些参数总是很自然地同时出现。  
以一个对象取代这些参数。

g.Remove Setting Method 移除设置函数

类中的某个字段应该在对象创建时被设置，然后不再修改。  
去掉该字段的所有设置函数。

k.Hide Method 隐藏函数

有一个函数，从来没有被其他任何类用到。  
将这个函数修改为private。

l.Replace Constructor with Factory Method 以工厂函数取代构造函数

你希望在创建对象时，不仅仅是做简单的建构动作。  
将构造函数替换为工厂函数。

m.Encapsulate Downcast 封装向下转型

某个函数返回的对象，需要由函数调用者执行向下转型。  
将向下转型动作移到函数中。

n.Replace Error Code with Exception 以异常取代错误码

某个函数返回一个特定的代码，用以表示某种错误情况。  
改用异常。

o.Replace Exception with Test 以测试取代异常

面对一个调用者可以预先检查的条件，你抛出了一个异常。  
修改调用者，使它在调用函数之前先做检查。