一、设计模式原则

1、找出应用中可能需求变化的代码，把它们独立出来，不要和那些需求不变化的代码混在一起
2、针对接口编程，而不要针对实现类编程
3、多用组合，少用继承
4、为了交互对象之间的松耦合设计而努力
5、类应该对扩展开放，对修改关闭
6、依赖倒置，要依赖抽象，不要依赖具体类

名称	解释
0、单一职责原则（SRP）	就一个类而言，应该仅有一个引起它变化的原因。
一、"开放－封闭"原则(OCP)	在软件设计模式中，这种不能修改，但可以扩展的思想也是最重要的一种设计原则。即软件实体（类、模板、函数等等）应该可以扩展，但是不可修改。 【通俗】：设计的时候，时刻考虑，尽量让这个类是足够好，写好了就不要去修改了，如果新需求来，我们增加一些类就完事了，原来的代码能不动则不动。
二、里氏代换原则（LSP）	1.一个软件实体如果使用的是一个父类的话，那么一定适用于该子类，而且他觉察不出父类对象和子类对象的区别。也就是说，在软件里面，把父类都替换成它的子类，程序的行为没有变化。 【一句话】：**子类型必须能够替换掉他们的父类型。**
三、依赖倒置原则(DIP)	1.高层模块不应该依赖于底层模块。两个都应该依赖抽象。2.抽象不应该依赖于细节，细节依赖于抽象 【白话】：针对接口编程，不要针对实现编程。
四、接口隔离原则（ISP）	1.使用多个专门的接口比使用单一的总接口总要好。换而言之，从一个客户类的角度来讲：一个类对另外一个类的依赖性应当是建立在最小接口上的。 2、过于臃肿的接口是对接口的污染。不应该强迫客户依赖于它们不用的方法。
五、合成/聚合复用原则（CARP）	尽量使用合成/聚合，尽量不要使用类继承。 【聚合】：表示一种弱的拥有关系，体现的是 A 对象可以包含 B 对象，但 B 对象不是 A 对象的一部分。 【合成】：一种强的拥有关系，提现了严格的部分和整体的关系，部分和整体的生存周期一致。
六、迪米特法则（LoD） 最少知识原则	强调类之间的松耦合。即：如果两个类不必彼此直接通信，那么着两个类就不应当发送直接的相互作用。如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。

___大部分内容摘自《大话设计模式》

1.题目不是6个设计模式吗？怎么列举了7个？

答：不同的书上列举的不太一样，单一模式原则和接口隔离原则多数都提了一个。本文都列举上，待深入探究后给出详细分析。

2.接口隔离原则与单一职责原则不是相同的吗？

答：错，接口隔离原则与单一职责的审视角度是不相同的。

单一职责要求的是类和接口职责单一，注重的是职责，这是业务逻辑上的划分；

而接口隔离原则要求接口的方法尽量少。例如一个接口的职责可能包含10个方法，这10个方法都放在一个接口中，并且提供给多个模块访问，各个模块按照规定的权限来访问，在系统外通过文档约束“不使用的方法不要访问”，按照单一职责原则是允许的，按照接口隔离原则是不允许的，因为它要求“尽量使用多个专门的接口”,专门的接口指什么？就是指提供给每个模块都应该是单一接口，提供给几个模块就应该有几个接口，而不是建立一个庞大的臃肿的接口，容纳所有的客户端访问。

面向对象设计原则

在面向对象设计中，如何通过很小的设计改变就可以应对设计需求的变化，这是令设计者极为关注的问题。为此不少OO（Object Oriented 面向对象）先驱提出了很多有关 面向对象的设计原则用于指导 OO 的设计和开发.

1. 单一职责原则

所谓单一职责原则，就是对一个类而言，应该仅有一个引起它变化的原因。换句话说， 一个类的功能要单一，只做与它相关的事情。在类的设计过程中要按职责进行设计，彼此保持正交，互不干涉。

比如我们编写一个java web应用程序，处理请求的时候，一般会定义一个 Servlet 类，于是我们就把各种各样的代码，如商业运算的算法，数据库访问的 sql 语句等写到这 个类中，这就意味着，无论任何需求到来，我们都要更改这个窗体类，维护麻烦，复用几 乎不可能，灵活性也相当差。

如果一个类承担的职责过多，就等于将这些职责耦合在一起，一个职责的变化可能会 削弱或者影响这个类完成其他职责的能力。我

们完全可以找出那些代码是做商业逻辑计算的，哪些逻辑是数据库访问的，然后将它们进行分离。软件设计真正要做的内容就是发现职责，并将这些职责相互分离。

2. 开放封闭原则

软件实体（类，模块，函数等等）应该可以扩展，但是不可修改。这个原则有两个特征，第一对于扩展开放 (Open for extension)，第二对于更改封闭 (Closed for modification) (引致《面向对象软件结构》)

> 我们在做任何系统的时候，都不要指望系统一开始需求确定，就再也不会变化，这是不现实也不科学的想法，既然需求一定要变，那么如何在面向需求的变化时，设计的软件可以容易修改，不至于新需求来了就要把整个程序推倒重写，那么开放封闭原则给了我们答案。

开放封闭原则的意思是：我们在设计的时候，要尽量让这个类是足够好的，写好了就不要去修改了，如果新需求来了，我们增加一些类就完事了，原来的代码能不动就不动。

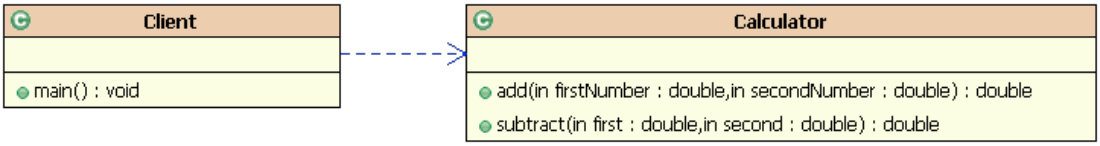
那么如何做才能满足这个原则？我们因该对程序中呈现出频繁变化的那一部分做出抽象，在 java 语言中我们可以使用抽象类或者接口作为抽象层，定义所有具有具体类必须实现的方法。抽象层应该预见所有可能的扩展，这样就能满足“对修改关闭”。同时根据 java 的多态性，抽象层可以有不同的实现类，每个实现类都可以改变系统的行为（通过重写抽象层中定义的方法），这就满足了“对扩展开放”。

2.1 举例说明

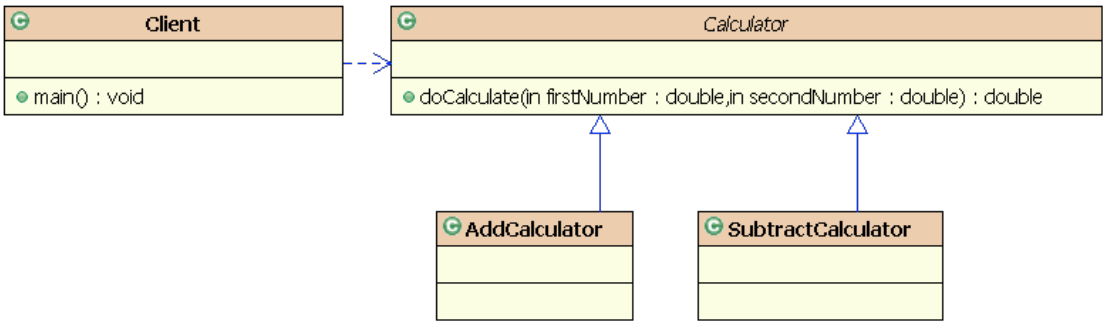
需求：制作一个能做加法和减法的计算器类，然后在客户端调用。

【注】：能够实现我们需求的功能性的类我称为服务类。使用这些服务的地方，我称为客户端

下面是设计的类图：



需求变化:为计算器增加乘法功能 如果按照上面的设计，要增加乘法运算，必须要修改 Calculator 类，此时违背了扩展开放原则。需求变化点是计算方式不确定，容易发生变化，于是我们将它进行抽象，得出如下的结构：



如果要增加乘法功能，只需要添加一个乘法类，然后在客户端使用即可。整个过程中， 我们对修改（服务端的类）没有做任何修改，而

只是扩展了服务端的类。当然客户端的调 用是需要修改的。我们说封闭原则是针对服务端代码，而客户端的修改关闭是不太现实的。

JDK 中的很多类如 List 集合，ArrrayList，LinkedList 的实现就遵循了开放封闭原则。

3. 依赖倒转原则

我们将电脑理解成一个大的软件系统，任何一个部件比如CPU，内存，硬盘，显卡等理解成程序中定义的类。由于电脑的易插拔方式，不管哪一个部件出了问题，都可以在 不影响别的部件的前提下进行修改或者替换，电脑的维护和更新升级变得很容易了。

我们电脑上的 CPU 全世界就只有几家在生产，我们都在使用它，但是不知道这些精密的CPU 是如何做出来的。这是因为CPU 内部“聚合”的非常的紧密，也就是具备“强内聚”，但是它又独自成为了产品，插入到适当的任何主板上都可以使用，这是什么原因？这是因为 CPU 对外的针脚都是同一个标准接口。CPU只需要按照标准接口去做，内部再 复杂也不让外界知道，而主板只需要按照标准接口预留CPU针脚的插槽就可以了。这种现象在面向对象中叫做“强内聚，低耦合”

什么是依赖倒转？ 在面向过程的开发中，为了使得一些代码可以复用，一般会将这些代码写成很多函数放到库中，就是函数库。比如我们很多项目都要访问数据库，所以我们把数据库的代码写成了函数，每次做新项目的时候就去调用这些函数，这就叫做“高层模块依赖于底层模块” 这种“高层模块依赖于底层模块”不是不可以，只是如果客户要求用不同的方式来存储数据，也就是更换数据库，就比较麻烦了。比如:如果电脑中的CPU，内存，硬盘都需要依赖具体的主板（比如所有的部件都直接集成到主板上。），那么当主板坏掉了，所有的部件就无法使用了。反过来，如果内存或者CPU 坏掉了，那么主板不能使用，其它的部件如集成显卡，声卡，网卡都无法使用了。如果不管是高层模块还是底层模块，它们都依赖于抽象，也就是接口或者抽象类，那么任何一个更改都不用担心其它受到影响，这就使得无论高层模块还是底层模块都可以很容易被复用。 所以依赖倒转其实就是谁也不要依赖谁，除了约定的接口，大家都可以灵活自如。 那么如何做才能符合这个原则？ 要针对接口（抽象层）编程，而不是实现类。比如定义集合因该使用：

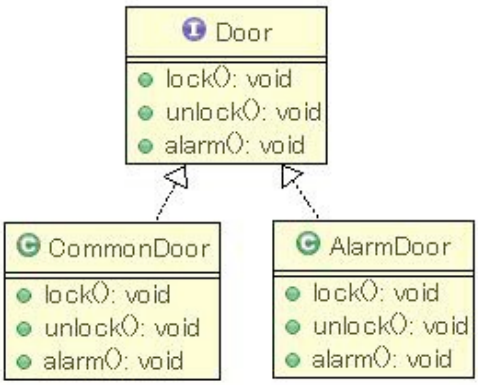
List list=new ArrayList() 而不是 ArrayList list=new ArrayList();

又比如 java.util.Collections 中提供的那些静态方法，这些静态方法中定义的参数类型不是接口就是抽象类，符合依赖倒转的原则。

4. 接口分离原则

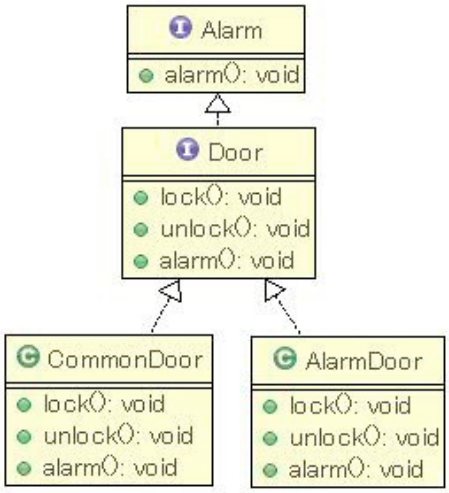
假如有一个 Door（门），有 lock(加锁)，unlock（解锁）功能，另外，可以在 Door上安装一个 Alarm(报警器)而使其具有报警功能。用户可以选择一般的 Door，也可以选择具有报警功能的 Door。

方案一：在 Door 接口里定义所有的方法



缺陷:依赖 Door 接口的 CommonDoor(普通门)却不得不实现未使用的 alarm()方法

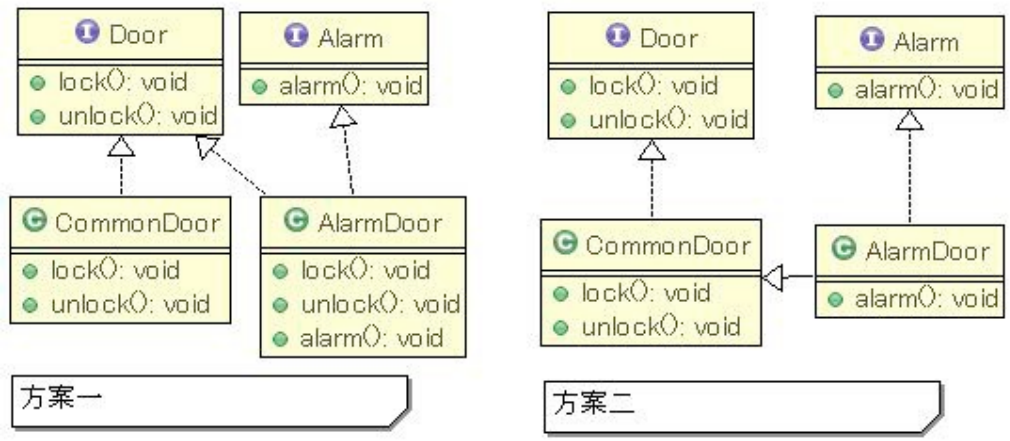
方案二：在 Alarm 接口定义 alarm 方法，在 Door 接口定义 lock，unlock 方法，Door接口继承 Alarm 接口。



缺陷：跟方案一一样，依赖 Door 接口的 CommonDoor 却不得不实现未使用的 alarm()方法，没有什么实质的改变 方案三：通过多重继承实现，在 Alarm 接口定义 alarm 方法，在 Door 接口定义 lock，unlock 方法。接口之间无继承关系。CommonDoor 实现 Door 接口，AlarmDoor 有 2 种实现方案：

1），同时实现 Door 和 Alarm 接口。

2），继承 CommonDoor，并实现 Alarm 接口。我们称第三种方案是符合接口分离原则的。



4.1 接口分离原则的含义

两层意思：

A)接口的设计应该遵循最小接口原则，不要把用户不使用的方法塞进同一个接口里。如果一个接口的方法没有被使用到，则说明该接口过胖，应该将其分割成几个功能专一的 接口。

B)接口的依赖（继承）原则：如果一个接口 a 依赖（继承）另一个接口 b，则接口 a 相当于继承了接口 b 的方法，那么继承了接口 b 后的接口 a 也应该遵循上述原则：不应该包含用户不使用的方法。反之，则说明接口 a 被 b 给污染了，应该重新设计它们的关系

如果用户被迫依赖他们不使用的接口，当接口发生改变时，他们也不得不跟着改变。换而言之，一个用户依赖了未使用但被其他用户使用的接口，当其他用户修改该接口时，依赖该接口的所有用户都将受到影响。这显然违反了开闭原则，也不是我们所期望的。

5. 最少知识原则

也称迪米特法则。指如果两个类不必彼此直接通信，那么这两个类就不要应当发生直 接的相互作用。如果其中一个类需要调用另外一个类的某一个方法的话，可以通过第三者 转发这个调用。

比如:产品销售部的小王要去与客户进行谈判，但是产品设计的一些技术问题它不是太清楚，为了谈判能取得成功，小王需要一个技术部的同事陪同前往。但小王刚来公司， 不认识技术部的人，但现在需要用到技术部的人，那小王该怎么办？小王可以找到技术部主管，请求技术部安排人与小王一起前往。至于技术部主管怎么安排小王不用关心，只需 要等待结果就可以了。

这就是两个类 Sales (销售员)，Engineer(技术员)，他们之间不需要直接联系,但现在 Sales 要用到 Engineer，要调用它的方法，如获取产品的技术特点，那么此时需要通 过第三个类 EngineerManager 类，让 Sales 通过 EnginnerManager 取得产品技术特点， 至于 EnginnerManager 让那个 Engineer 来解答产品问题，那就是 EnginnerManager 内部的事情了，与 Sales 无关

可以看到，最少知识原则实际上是强调了类之间的松耦合，类之间的耦合越弱，越有利于重复利用，一个处在弱耦合的类被修改，不会波及到有关系的类。我们在设计类的时 候，尽量的封装内部的工作细节，实现信息的隐藏，从而减少各个类之间的耦合。

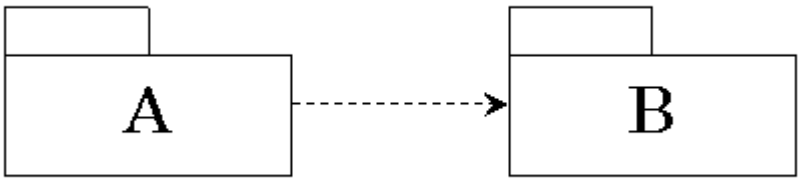
6. 包的设计原则

- 通用闭包原则(Common Closure Principle)

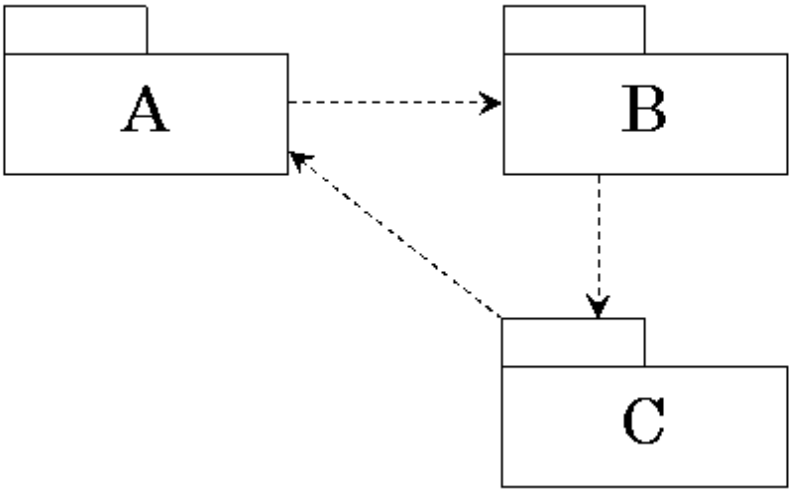
  通用闭包原则强调了包的内聚性，当一个类的改变可能引起对另一个类的改变的时候，最好将这两个类放在同一个包中

- 无循环依赖原则

  如果一个包 A 中的类引用了包 B 中的类，我们称包 A 依赖包 B。



如果存在 2 个或 2 个以上的包，它们之间的依赖关系图出现了环状，我们就称包之间
存在循环依赖关系。也就是说它们的依赖结构图根据箭头的方向形成了一个环状的闭合图形。如图：



如果多个包之间形成了循环依赖，比如上面的图，A 依赖 B，B 依赖 C，C 依赖 A，我们修改了 B 并需要发布 B 的一个新的版本，因为 B 依赖 C，所以发布时应该包含 C，但 C 同时又依赖 A，所以又应该把 A 也包含进发布版本里。也就是说，依赖结构中，出现在环内的所有包都不得不一起发布。它们形成了一个高耦合体，当项目的规模大到一定程度，包的数目变多时，包与包之间的关系便变得错综复杂，各种测试也将变得非常困难，常常会因为某个不相关的包中的错误而使得测试无法继续。而发布也变得复杂，需要把所有的包一起发布，无疑增加了发布后的验证难度。

如何打破这种循环依赖呢？

包 C 要依赖包 A，必定 A 中包含有 A，C 共通使用的类，把这些共同类抽出来放在一个新的包 D 里。这样就把 C 依赖 A 变成了 C

依赖 D 以及 A 依赖 D，从而打破了循环依赖关系