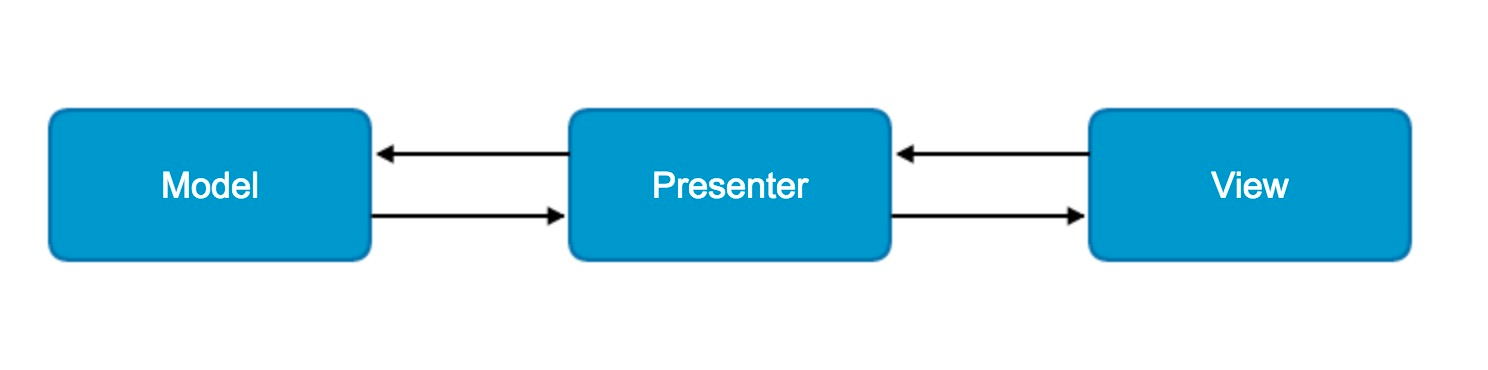
MVP结构图来看一下MVP中各个层次之间的关系



在MVP架构中将这三层分别抽象到各自的接口当中。通过接口将层次之间进行隔离，而Presenter对View和Model的相互依赖也是依赖于各自的接口。这点符合了接口隔离原则，也正是面向接口编程。在Presenter层中包含了一个View接口，并且依赖于Model接口，从而将Model层与View层联系在一起。而对于View层会持有一个Presenter成员变量并且只保留对Presenter接口的调用，具体业务逻辑全部交由Presenter接口实现类中处理

**MVP架构简介**

对于一个应用而言我们需要对它抽象出各个层面，而在MVP架构中它将UI界面和数据进行隔离，所以我们的应用也就分为三个层次。

* View: 对于View层也是视图层，在View层中只负责对数据的展示，提供友好的界面与用户进行交互。在Android开发中通常将Activity或者Fragment作为View层。
* Model: 对于Model层也是数据层。它区别于MVC架构中的Model，在这里不仅仅只是数据模型。在MVP架构中Model它负责对数据的存取操作，例如对数据库的读写，网络的数据的请求等。
* Presenter:对于Presenter层他是连接View层与Model层的桥梁并对业务逻辑进行处理。在MVP架构中Model与View无法直接进行交互。所以在Presenter层它会从Model层获得所需要的数据，进行一些适当的处理后交由View层进行显示。这样通过Presenter将View与Model进行隔离，使得View和Model之间不存在耦合，同时也将业务逻辑从View中抽离。

**为什么用MVP架构**

其实我们日常开发中的Activity，Fragment和XML界面就相当于是一个 MVC 的架构模式，Activity中不仅要处理各种 UI 操作还要请求数据以及解析。

这种开发方式的缺点就是业务量大的时候一个Activity 文件分分钟飙到上千行代码，想要改一处业务逻辑光是去找就要费半天劲，而且有点地方逻辑处理是一样的无奈是不同的 Activity 就没办法很好的写成通用方法。

**那 MVP 为啥好用呢？**

MVP 模式将Activity 中的业务逻辑全部分离出来，让Activity 只做 UI 逻辑的处理，所有跟Android API无关的业务逻辑由 Presenter 层来完成。

将业务处理分离出来后最明显的好处就是管理方便，但是缺点就是增加了代码量。

Demo1 代码 简单的使用mvp

代码。。。。。。。。。。。。。

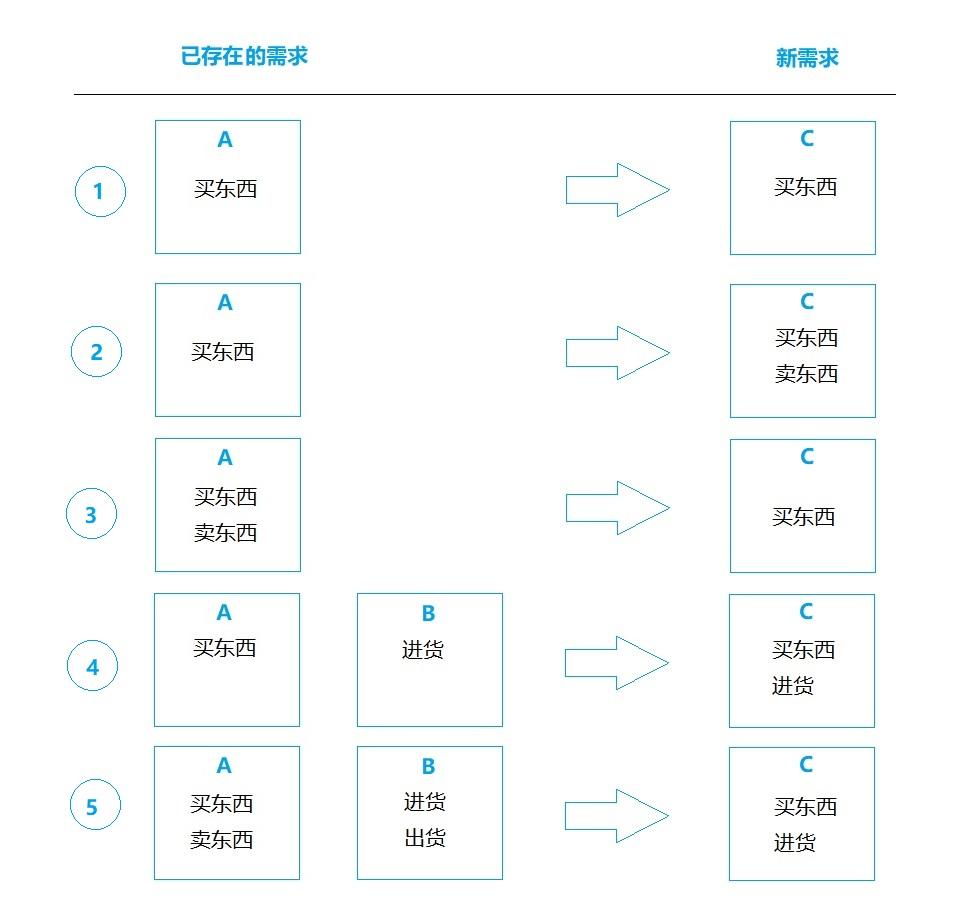
## MVP中的代码复用场景

因为上节中乞丐版MVP Demo的代码只实现了一个Activity的请求操作，容易出现一个概念的混淆：

**每个Activity都需要有与它对应的一套MVP（Model,View,Presenter）吗？**

答案肯定是否定的！

首先不需要数据请求的Activity当然就同样不需要MVP辅助。与其他Activity中存在相同逻辑的Activity，就不需要重复生成对应的MVP。但是这个存在相同逻辑的定义，不同的场景有不同的说法：



### 场景1：业务逻辑完全相同

场景1中Activity A和Activity C都只有一个“买东西”的逻辑，属于典型的逻辑相同，所以Activity C就可以直接用Activity A写好的MVP无需再做任何处理。

### 场景2、3：包含部分相同业务逻辑

场景2和场景3的逻辑类似，都属于一个业务逻辑中包含另外一个可以单独存在的业务逻辑，这种情况采用继承的方法即可：

### 场景4

场景4中Activity C想要同时调用独立服务于Activity A 和 Activity B的业务逻辑，只需要将两个业务逻辑对应的Presenter分别实例化并调用业务方法即可：

private PresenterA presenterA;

private PresenterB presenterB;

private void getData(){

presenterA.getData();

presenterB.getData();

}

不要忘了实现两个Presenter对应的View：

public class ActivityC extends Activity implements ViewA,ViewB{

}

### 场景5

场景5属于场景3与场景4的结合体，同样需要先把A和B的业务逻辑拆分开，然后同时调用，这里就不举例子了。

### 总结

通过上面一揽子场景的分析，得出的第一个结论就是MVP的结构太过于繁重，所以为了避免多写重复代码和日后需要进行无意义的修改，在开发前一定要设计好逻辑调用图，这样才能事半功倍。

对于上面经典的通过业务逻辑继承实现包含重复逻辑的方法，其实也可以在一个Presenter中写好完整的逻辑方法，对于不同的Activity需要哪个业务逻辑方法就调用哪个，这样岂不就简单多了。但是从架构设计角度看这种做法是不严谨的，可能存在漏洞，所以为保持软件架构的健壮还是不要偷懒的好。

### 优化 构建base层

写到这里，相信大多数人都会惊讶于MVP模式代码量的巨大，冗余代码实在太多，所以接下需要为MVP中所有单元都设计一个顶级父类来减少重复的冗余代码。同样的道理，我们也为Activity设计一个父类方便与MVP架构更完美的结合。最后将所有父类单独分到一个base包中供外界继承调用。

Demo2

代码。。。。。

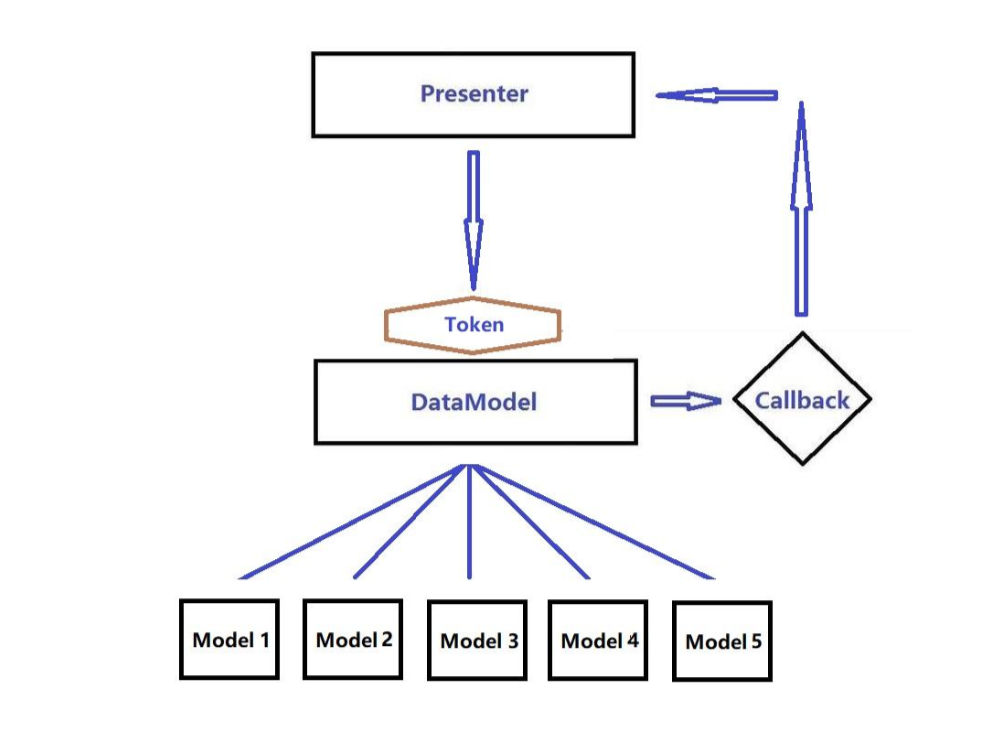
### 优化 M层

#### 单独封装，集中管理

Model层相比其他单元来说比较特殊，因为它们更像一个整体，只是单纯的帮上层拿数据而已。再就是MVP的理念是让业务逻辑互相独立，这就导致每个的网络请求也被独立成了单个Model，这种方式在实际开发中就会出现一些问题：

* 无法对所有Model统一管理。
* 每个Model对外提供的获取数据方法不一样，上层请求数据没有规范。
* 代码冗余高，网络数据请求除URL和参数外其他大概都一样的。
* 对已存在的Model管理困难，不能直观的统计已存在的Model。

所以我们更希望Model层是一个庞大且独立单一模块，请求方式规范化，管理Model更加直观。



**Presenter 请求数据不再直接调用具体的Model对象，统一以 DataModel 类作为数据请求层的入口，以常量类 Token 区别具体请求。** DataModel会根据Token的不同拉取底层对应的具体Model。

优化之后的Model层是一个庞大而且独立的模块，对外提供统一的请求数据方法与请求规则，这样做的好处有很多：

* 数据请求单独编写，无需配合上层界面测试。
* 统一管理，修改方便。
* 利用Token类可以直观的统计出已存在的请求接口。

Demo3

代码。。。。。。。。。。。。

通过MVP架构的使用可以看出对于各个层次之间的职责更加单一清晰，同时也很大程度上降低了代码的耦合度。

参考：

[https://github.com/googlesamples/android-architecture](https://link.jianshu.com/?t=https%3A%2F%2Fgithub.com%2Fgooglesamples%2Fandroid-architecture).

<https://www.jianshu.com/p/5eeda2ac1487>

<https://www.jianshu.com/p/5c3bc32afa36>

# 聊设计原则在mvp中的体现

本篇是对六大原则的整理。(最后一种是哈姆雷特)

# 1.开闭原则(Open Close Principle)

定义：一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

开放-封闭原则的意思就是说，你设计的时候，时刻要考虑，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。这个原则有两个特性，一个是说“对于扩展是开放的”，另一个是说“对于更改是封闭的”。面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。这就是“开放-封闭原则”的精神所在

比如，刚开始需求只是写加法程序，很快在client类中完成后，此时变化没有发生，需求让再添加一个减法功能，此时会发现增加功能需要修改原来这个类，这就违背了开放-封闭原则，于是你就应该考虑重构程序，增加一个抽象的运算类，通过一些面向对象的手段，如继承、动态等来隔离具体加法、减法与client耦合，需求依然可以满足，还能应对变化。此时需求要添加乘除法功能，就不需要再去更改client及加减法类，而是增加乘法和除法子类即可。

绝对的修改关闭是不可能的，无论模块是多么的‘封闭‘，都会存在一些无法对之封闭的变化，既然不可能完全封闭，设计人员必须对于他设计的模块应该对哪种变化封闭做出选择。他必须先猜测出最有可能发生的变化种类，然后构造抽象来隔离那些变化。在我们最初编写代码时，假设变化不会发生，当变化发生时，我们就创建抽象来隔离以后发生同类的变化。

我们希望的是在开发工作展开不久就知道可能发生的变化，查明可能发生的变化所等待的时候越长，要创建正确的抽象就越困难。开放-封闭原则是面向对象设计的核心所在，遵循这个原则可以带来面向对象技术所声称的巨大好处，也就是可维护、可扩展、可复用、灵活性好。开发人员应该仅对程序中呈现出现频繁变化的那些部分做出抽象，然而对于应用程序中的每个部分都刻意地进行抽象同样不是一个好主意，拒绝不成熟的抽象和抽象本身一样重要。开放-封闭原则，可以保证以前代码的正确性，因为没有修改以前代码，所以可以保证开发人员专注于将设计放在新扩展的代码上。

简单的用一句经典的话来说：过去的事已成历史，是不可修改的，因为时光不可倒流，但现在或明天计划做什么，是可以自己决定(即扩展)的。

# 2.里氏代换原则(Liskov Substitution Principle)

定义1：如果对每一个类型为 T1的对象 o1，都有类型为 T2 的对象o2，使得以 T1定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。

定义2：子类型必须能够替换掉它们的父类型。

描述：一个软件实体如果使用的是一个父类的话，那么一定适用于其子类，而且它察觉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

例子：在生物学分类上，企鹅是一种鸟，但在[编程](https://www.2cto.com/kf)世界里，企鹅却不能继承鸟。在面向对象设计时，子类拥有父类所有非private的行为和属性，鸟会飞，但企鹅不会飞，所以企鹅不能继承鸟类。

只有当子类可以替换掉父类，软件单位的功能不受影响时，父类才能真正被复用，而子类也能够在父类的基础上增加新的行为，正是有里氏代换原则，使得继承复用成为了可能。正是由于子类型的可替换性才使得使用父类类型的模块在无需修改的情况下就可以扩展，不然还谈什么扩展开放，修改关闭呢

里氏替换原则通俗的来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下4层含义：

1.子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。

2.子类中可以增加自己特有的方法。

3.当子类的方法重载父类的方法时，方法的前置条件(即方法的形参)要比父类方法的输入参数更宽松。

4.当子类的方法实现父类的抽象方法时，方法的后置条件(即方法的返回值)要比父类更严格。

看上去很不可思议，因为我们会发现在自己编程中常常会违反里氏替换原则，程序照样跑的好好的。所以大家都会产生这样的疑问，假如我非要不遵循里氏替换原则会有什么后果?

后果就是：你写的代码出问题的几率将会大大增加。

# 3.依赖倒转原则(Dependence Inversion Principle)

定义：高层模块不应该依赖低层模块，二者都应该依赖其抽象;抽象不应该依赖细节;细节应该依赖抽象。即针对接口编程，不要针对实现编程

依赖倒转其实就是谁也不要依靠谁，除了约定的接口，大家都可以灵活自如。依赖倒转可以说是面向对象设计的标志，用哪种语言来编写程序不重要，如果编写时考虑的都是如何针对抽象编程而不是针对细节编程，即程序中所有的依赖关系都是终止于抽象类或者接口，那就是面向对象的设计，反之那就是过程化的设计了。如果设计的各个部件或类相互依赖，这样就是耦合度高，难以维护和扩展，这也就体现不出面向对象的好处了。

依赖倒转原则，好比一个团队，有需求组，开发组，测试组，开发组和测试组都是面对同样的需求后，做自己相应的工作，而不应该是测试组按照开发组理解的需求去做测试用例，也就是说开发组和测试组都是直接面向需求组工作，大家的目的是一样的，保证产品按时上线，需求是不依赖于开发和测试的。

依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在java中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

依赖倒置原则的中心思想是面向接口编程，传递依赖关系有三种方式，以上的说的是是接口传递，另外还有两种传递方式：构造方法传递和setter方法传递，相信用过Spring框架的，对依赖的传递方式一定不会陌生。

在实际编程中，我们一般需要做到如下3点：

低层模块尽量都要有抽象类或接口，或者两者都有。

变量的声明类型尽量是抽象类或接口。

使用继承时遵循里氏替换原则。

总之，依赖倒置原则就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

# 4.接口隔离原则(Interface Segregation Principle)

接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高[系统](https://www.2cto.com/os/)的灵活性和可维护性。

说到这里，很多人会觉的接口隔离原则跟单一职责原则很相似，其实不然。其一，单一职责原则原注重的是职责;而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节;而接口隔离原则主要约束接口接口，主要针对抽象，针对程序整体框架的构建。

采用接口隔离原则对接口进行约束时，要注意以下几点：

1. 接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不挣的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。

2. 为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。

3. 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。

运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地实践这一原则。

5.迪米特法则(Law Of Demeter)

迪米特法则其根本思想，是强调了类之间的松耦合，类之间的耦合越弱,越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成影响，也就是说，信息的隐藏促进了软件的复用。

自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢?那正是迪米特法则要去完成的。

迪米特法则又叫最少知道原则，最早是在1987年由美国Northeastern University的Ian Holland提出。通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息。迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

一句话总结就是：一个对象应该对其他对象保持最少的了解。

# 6.单一职责原则(Single Responsibility Principle)

定义：不要存在多于一个导致类变更的原因。通俗的说，即一个类只负责一项职责，应该仅有一个引起它变化的原因

说到单一职责原则，很多人都会不屑一顾。因为它太简单了。稍有经验的程序员即使从来没有读过设计模式、从来没有听说过单一职责原则，在设计软件时也会自觉的遵守这一重要原则，因为这是常识。在软件编程中，谁也不希望因为修改了一个功能导致其他的功能发生故障。而避免出现这一问题的方法便是遵循单一职责原则。虽然单一职责原则如此简单，并且被认为是常识，但是即便是经验丰富的程序员写出的程序，也会有违背这一原则的代码存在。为什么会出现这种现象呢?因为有职责扩散。所谓职责扩散，就是因为某种原因，职责P被分化为粒度更细的职责P1和P2。

遵循单一职责原的优点有：

1.可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多;

2.提高类的可读性，提高系统的可维护性;

3.变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可以显著降低对其他功能的影响。

需要说明的一点是单一职责原则不只是面向对象编程思想所特有的，只要是模块化的程序设计，都需要遵循这一重要原则。

7.组合/聚合复用原则

就是说要尽量的使用合成和聚合，而不是继承关系达到复用的目的

该原则就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分：新的对象通过向这些对象的委派达到复用已有功能的目的。

其实这里最终要的地方就是区分“has-a”和“is-a”的区别。相对于合成和聚合，

继承的缺点在于：父类的方法全部暴露给子类。父类如果发生变化，子类也得发生变化。聚合的复用的时候就对另外的类依赖的比较的少。。

## 合成/聚合复用

① 优点：

新对象存取成分对象的唯一方法是通过成分对象的接口;

这种复用是黑箱复用，因为成分对象的内部细节是新对象所看不见的;

这种复用支持包装;

这种复用所需的依赖较少;

每一个新的类可以将焦点集中在一个任务上;

这种复用可以在运行时动态进行，新对象可以使用合成/聚合关系将新的责任委派到合适的对象。

② 缺点：

通过这种方式复用建造的系统会有较多的对象需要管理。

继承复用

① 优点：

新的实现较为容易，因为基类的大部分功能可以通过继承关系自动进入派生类;

修改或扩展继承而来的实现较为容易。

② 缺点：

继承复用破坏包装，因为继承将基类的实现细节暴露给派生类，这种复用也称为白箱复用;如果基类的实现发生改变，那么派生类的实现也不得不发生改变;从基类继承而来的实现是静态的，不可能在运行时发生改变，不够灵活。