# 一 类设计

1面向对象的基本概念：继承、封装、多态。类、抽象类、接口之间的关系

2面向对象的设计原则：单一职责、开-闭、里氏替换、依赖倒置、接口分离、迪米特原则等

3理解面向对象的设计思想（清楚表达）

4掌握面向对象的设计原则（清楚表达）

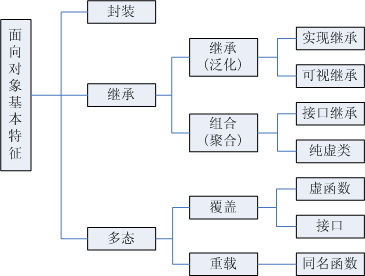
5独立完成3个以上模块级类设计，并能清楚的阐述设计思路以及遵循的设计原则

6能够结合面向对象的设计原则和设计模式描述1-2个开源组件或者公司技术框架的类设计

# 面向对象的三个基本特征 和 五种设计原则

## 一、三个基本特征

面向对象的三个基本特征是：封装、继承、多态。



### 封装

封装最好理解了。封装是面向对象的特征之一，是对象和类概念的主要特性。

封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

### 继承

面向对象编程 (OOP) 语言的一个主要功能就是“继承”。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

通过继承创建的新类称为“子类”或“派生类”。

被继承的类称为“基类”、“父类”或“超类”。

继承的过程，就是从一般到特殊的过程。

要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。

在某些 OOP 语言中，一个子类可以继承多个基类。但是一般情况下，一个子类只能有一个基类，要实现多重继承，可以通过多级继承来实现。

继承概念的实现方式有三类：实现继承、接口继承和可视继承。

\* 实现继承是指使用基类的属性和方法而无需额外编码的能力；

\* 接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力；

\* 可视继承是指子窗体（类）使用基窗体（类）的外观和实现代码的能力。

在考虑使用继承时，有一点需要注意，那就是两个类之间的关系应该是“属于”关系。例如，Employee 是一个人，Manager 也是一个人，因此这两个类都可以继承 Person 类。但是 Leg 类却不能继承 Person 类，因为腿并不是一个人。

抽象类仅定义将由子类创建的一般属性和方法，创建抽象类时，请使用关键字 Interface 而不是 Class。

OO开发范式大致为：划分对象→抽象类→将类组织成为层次化结构(继承和合成) →用类与实例进行设计和实现几个阶段。

### 多态

多态性（polymorphisn）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。

实现多态，有二种方式，覆盖，重载。

覆盖，是指子类重新定义父类的虚函数的做法。

重载，是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。

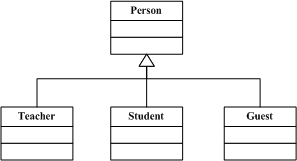
其实，重载的概念并不属于“面向对象编程”，重载的实现是：编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。如，有两个同名函数：function func(p:integer):integer;和function func(p:string):integer;。那么编译器做过修饰后的函数名称可能是这样的：int\_func、str\_func。对于这两个函数的调用，在编译器间就已经确定了，是静态的（记住：是静态）。也就是说，它们的地址在编译期就绑定了（早绑定），因此，重载和多态无关！真正和多态相关的是“覆盖”。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态（记住：是动态！）的调用属于子类的该函数，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。因此，这样的函数地址是在运行期绑定的（晚邦定）。结论就是：重载只是一种语言特性，与多态无关，与面向对象也无关！引用一句Bruce Eckel的话：“不要犯傻，如果它不是晚邦定，它就不是多态。”

那么，多态的作用是什么呢？

我们知道，封装可以隐藏实现细节，使得代码模块化；继承可以扩展已存在的代码模块（类）；它们的目的都是为了——代码重用。而多态则是为了实现另一个目的——接口重用！多态的作用，就是为了类在继承和派生的时候，保证使用“家谱”中任一类的实例的某一属性时的正确调用。

概念讲解

### 泛化（Generalization）



图表 1 泛化

在上图中，空心的三角表示继承关系（类继承），在UML的术语中，这种关系被称为泛化（Generalization）。Person(人)是基类，Teacher(教师)、Student(学生)、Guest(来宾)是子类。

若在逻辑上B是A的“一种”，并且A的所有功能和属性对B而言都有意义，则允许B继承A的功能和属性。

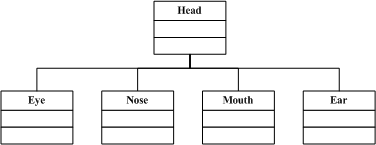
例如，教师是人，Teacher 是Person的“一种”（a kind of ）。那么类Teacher可以从类Person派生（继承）。

如果A是基类，B是A的派生类，那么B将继承A的数据和函数。

如果类A和类B毫不相关，不可以为了使B的功能更多些而让B继承A的功能和属性。

若在逻辑上B是A的“一种”（a kind of ），则允许B继承A的功能和属性。

### 聚合（组合）



图表 2 组合

若在逻辑上A是B的“一部分”（a part of），则不允许B从A派生，而是要用A和其它东西组合出B。

例如，眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类Head应该由类Eye、Nose、Mouth、Ear组合而成，不是派生（继承）而成。

聚合的类型分为无、共享(聚合)、复合(组合)三类。

聚合（aggregation）

http://www.cnitblog.com/images/cnitblog_com/lily/1972/o_aggregation.gif

图表 3 共享

上面图中，有一个菱形（空心）表示聚合（aggregation）（聚合类型为共享），聚合的意义表示has-a关系。聚合是一种相对松散的关系，聚合类B不需要对被聚合的类A负责。

组合（composition）

http://www.cnitblog.com/images/cnitblog_com/lily/1972/o_composition.gif

图表 4 复合

这幅图与上面的唯一区别是菱形为实心的，它代表了一种更为坚固的关系——组合（composition）（聚合类型为复合）。组合表示的关系也是has-a，不过在这里，A的生命期受B控制。即A会随着B的创建而创建，随B的消亡而消亡。

依赖(Dependency)

http://www.cnitblog.com/images/cnitblog_com/lily/1972/o_Dependency.gif

图表 5 依赖

这里B与A的关系只是一种依赖(Dependency)关系，这种关系表明，如果类A被修改，那么类B会受到影响。

## 二、五种设计原则

"面向对象设计五大原则"和良性依赖原则在应付变化方面的作用。

### 单一职责原则

(Single-Resposibility Principle)。"对一个类而言，应该仅有一个引起它变化的原因。"本原则是我们非常熟悉地"高内聚性原则"的引申，但是通过将"职责"极具创意地定义为"变化的原因"，使得本原则极具操作性，尽显大师风范。同时，本原则还揭示了内聚性和耦合生，基本途径就是提高内聚性；如果一个类承担的职责过多，那么这些职责就会相互依赖，一个职责的变化可能会影响另一个职责的履行。其实OOD的实质，就是合理地进行类的职责分配。

### 开放封闭原则

(Open-Closed principle)。"软件实体应该是可以扩展的，但是不可修改。"本原则紧紧围绕变化展开，变化来临时，如果不必改动软件实体裁的源代码，就能扩充它的行为，那么这个软件实体设计就是满足开放封闭原则的。如果说我们预测到某种变化，或者某种变化发生了，我们应当创建抽象类来隔离以后发生的同类变化。在Java中，这种抽象是指抽象基类或接口；在C++中，这各抽象是指抽象基类或纯抽象基类。当然，没有对所有情况都贴切的模型，我们必须对软件实体应该面对的变化做出选择。

### Liskov替换原则

(Liskov-Substituion Principle)。"子类型必须能够替换掉它们的基类型。"本原则和开放封闭原则关系密切，正是子类型的可替换性，才使得使用基类型模块无需修改就可扩充。Liskov替换原则从基于契约的设计演化而来，契约通过为每个方法声明"先验条件"和"后验条件"；定义子类时，必须遵守这些"先验条件"和"后验条件"。当前基于契的设计发展势头正劲，对实现"软件工厂"的"组装生产"梦想是一个有力的支持。

### 依赖倒置原则

(Dependecy-Inversion Principle)。"抽象不应依赖于细节，细节应该依赖于抽象。"本原则几乎就是软件设计的正本清源之道。因为人解决问题的思考过程是先抽象后具体，从笼统到细节，所以我们先生产出的势必是抽象程度比较高的实体，而后才是更加细节化的实体。于是，"细节依赖于抽象"就意味着后来的依赖于先前的，这是自然而然的重用之道。而且，抽象的实体代表着笼而统之的认识，人们总是比较容易正确认识它们，而且本身也是不易变的，依赖于它们是安全的。依赖倒置原则适应了人类认识过程的规律，是面向对象设计的标志所在。

### 接口隔离原则

(Interface-Segregation Principle)。"多个专用接口优于一个单一的通用接口。"本原则是单一职责原则用于接口设计的自然结果。一个接口应该保证，实现该接口的实例对象可以只呈现为单一的角色；这样，当某个客户程序的要求发生变化，而迫使接口发生改变时，影响到其他客户程序的可能生性小。

良性依赖原则。"不会在实际中造成危害的依赖关系，都是良性依赖。"通过分析不难发现，本原则的核心思想是"务实"，很好地揭示了极限编程(Extreme Programming)中"简单设计"各"重构"的理论基础。本原则可以帮助我们抵御"面向对象设计五大原则"以及设计模式的诱惑，以免陷入过度设计(Over-engineering)的尴尬境地，带来不必要的复杂性。

# 七大面向对象设计原则

<http://blog.csdn.net/qiulongtianshi/article/details/7570021>

花絮

每天都在和面向对象打交道，但是我们在应用面向对象的时候感觉自己的面向对象技术应用的很合理？理解的很到位？应用的很到位？用的时候恰到好处？用的是否符合软件的发展趋势？

上面很多一连串的问题，没有把你搞晕吧！我想说的面向对象的应用也有面向对象的原则，一个程序员的编程素质从一个小例子几十行代码就可以展现出来。面向对象编程是是面向对象的基础中的基础，废话不多说，你要遵循下面向对象的原则编程，那你的编程水平才会大幅度提高！

思考

大家可以想象为什么你去面试的是时候一般会给你一小段程序让你去编写？这样的有什么意义？主考官的出题目的是什么？我们应该怎么显示自己代码素质？

我想这个问题我们在面试的时候应该好好想一下，因为这样我们才能更好的应对面试，更好的展现我们的才华，公司更喜欢录用你，而你要更高薪金是很轻而易举的事情。

答案是：一小段代码就能显示我们的代码素质，去面试实现简简单单的功能并不是很牛，而是把自己的编程思维展现给别人（编程思维培养请看：“菜鸟”程序员和“大神”程序员差距在哪里），而最重要的是展现出软件的功能的可维护性和复用性，因为这样编程的才是自己思维的结晶，编程的后期维护带来了极大的好处，所以说软件功能的可维护性和复用性是软件的发展潜力，软件的这个方面才是最重要的！





通过这两幅图片你就可以认识到他们的重要性了吧，因为他们是这个行业的顶级专家，他们的思想是这么高，我们不应该好好的学习一下巨人的思想吗？答案是肯定的。

个人理解观点：

软件的复用(Reuse)或重用拥有众多优点，如可以提高软件的开发效率，提高软件质量，节约开发成本，恰当的复用还可以改善系统的可维护性。

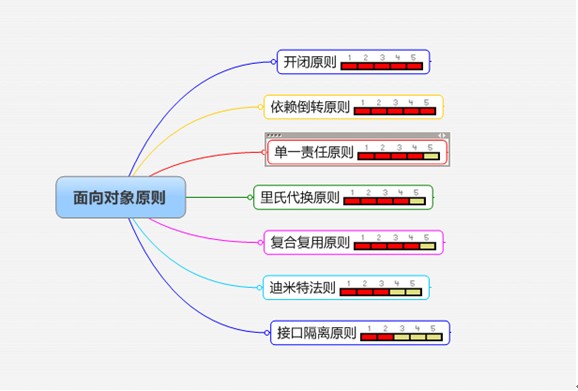
面向对象设计复用的目标在于实现支持可维护性的复用。

在面向对象的设计里面，可维护性复用都是以面向对象设计原则为基础的，这些设计原则首先都是复用的原则，遵循这些设计原则可以有效地提高系统的复用性，同时提高系统的可维护性。

面向对象设计原则和设计模式也是对系统进行合理重构的指南针，重构(Refactoring)是在不改变软件现有功能的基础上，通过调整程序代码改善软件的质量、性能，使其程序的设计模式和架构更趋合理，提高软件的扩展性和维护性。

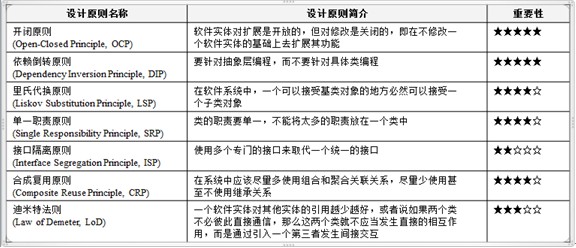
## 面向对象原则综述

七大原则总脉络图：



注：1，2，3，4，5显示的重要等级

常用的面向对象设计原则包括7个，这些原则并不是孤立存在的，它们相互依赖，相互补充。



下面就是面向对象七个原则的一一解析

## 一 开闭原则

1. 开闭原则定义 ：一个软件实体应当对扩展开放，对修改关闭。也就是说在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展，即实现在不修改源代码的情况下改变这个模块的行为。

2. 开闭原则分析：

（1）开闭原则由Bertrand Meyer于1988年提出，它是面向对象设计中最重要的原则之一。

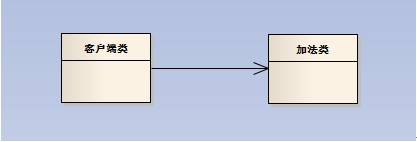
（2）在开闭原则的定义中，软件实体可以指一个软件模块、一个由多个类组成的局部结构或一个独立的类。

（3）抽象化是开闭原则的关键。

（4）开闭原则还可以通过一个更加具体的“对可变性封装原则”来描述，对可变性封装原则(Principle of Encapsulation of Variation,EVP)要求找到系统的可变因素并将其封装起来。

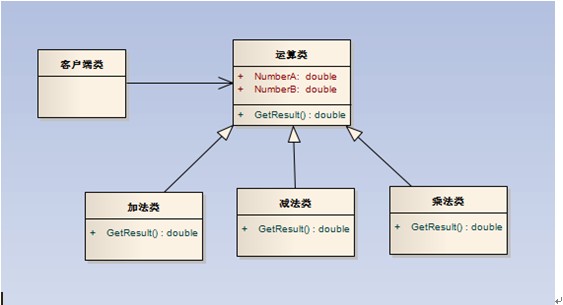
3. 开闭原则实例：

某图形界面系统提供了各种不同形状的按钮，客户端代码可针对这些按钮进行编程，用户可能会改变需求要求使用不同的按钮，原始设计方案如图所示：



图（1）

现对该系统进行重构，使之满足开闭原则的要求。

 图（2）

对比分析

图（1）：客户端的一个方法直接调用加法类，但是我想添加一个减法类，你就会发现添加减法类就得改变加法类中代码（用switch语句实现），这就违背了“开闭原则”，于是我们就应该重新重构。

如图（2）在这个图中我们添加了一个运算类的父类，这样我们再添加减法类的时候就不用修改客户端类。

开闭原则总结：面对需求，对程序的改动是通过增加新代码进行的，而不是改变原来的代码。

## 二、依赖倒转原则

1.依赖倒转原则定义

i. 高层模块不应该依赖低层模块，它们都应该依赖抽象。抽象不应该依赖于细节，细节应该依赖于抽象。

ii. 要针对接口编程，不要针对实现编程。

2. 依赖倒转原则分析

a) 简单来说，依赖倒转原则就是指：代码要依赖于抽象的类，而不要依赖于具体的类；要针对接口或抽象类编程，而不是针对具体类编程。

b) 实现开闭原则的关键是抽象化，并且从抽象化导出具体化实现，如果说开闭原则是面向对象设计的目标的话，那么依赖倒转原则就是面向对象设计的主要手段。

c) 依赖倒转原则的常用实现方式之一是在代码中使用抽象类，而将具体类放在配置文件中。

d) 类之间的耦合 1.零耦合关系 2.具体耦合关系 3.抽象耦合关系。

依赖倒转原则要求客户端依赖于抽象耦合，以抽象方式耦合是依赖倒转原则的关键。

e) 依赖注入:

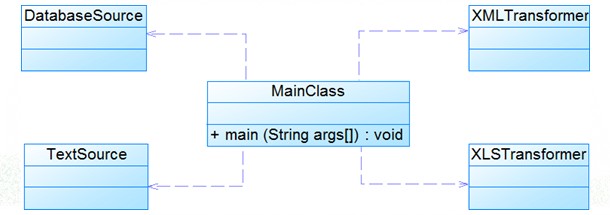
构造注入：通过构造函数注入实例变量

设值注入：通过Setter方法注入实例变量

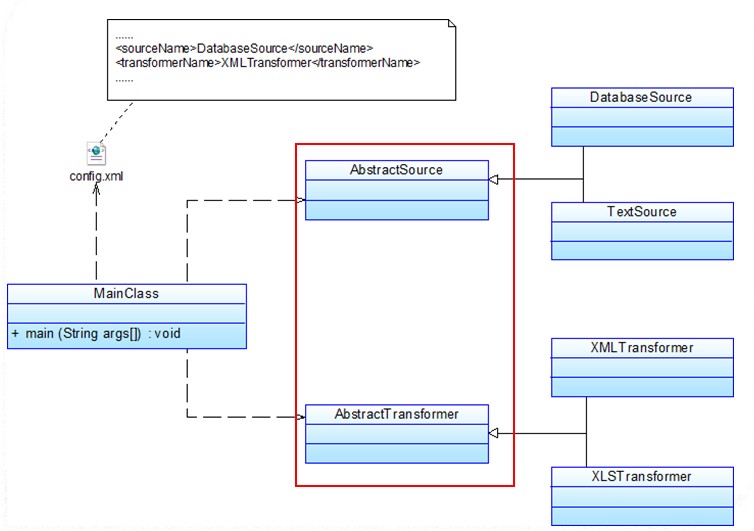
接口注入：通过接口方法注入实例变量

3. 依赖倒转原则实例

某系统提供一个数据转换模块，可以将来自不同数据源的数据转换成多种格式，如可以转换来自数据库的数据(DatabaseSource)、也可以转换来自文本文件的数据(TextSource)，转换后的格式可以是XML文件(XMLTransformer)、也可以是XLS文件(XLSTransformer



图（一）



图（二）

图（一）和图（二）分析：

图（一）为什么要到图（二）哪？因为该系统可能需要增加新的数据源或者新的文件格式，每增加一个新的类型的数据源或者新的类型的文件格式，客户类MainClass都需要修改源代码，以便使用新的类，但违背了开闭原则。现使用依赖倒转原则对其进行重构。

总结：高层模块不应该依赖底层模块，两个都应该依赖与抽象；抽象不应该依赖于细节，细节应该依赖于抽象。

## 三、 里氏代换原则

1. 定义

i. 如果对每一个类型为S的对象o1，都有类型为T的对象o2，使得以T定义的所有程序P在所有的对象o1都代换成o2时，程序P的行为没有变化，那么类型S是类型T的子类型。

ii. 所有引用基类（父类）的地方必须能透明地使用其子类的对象。

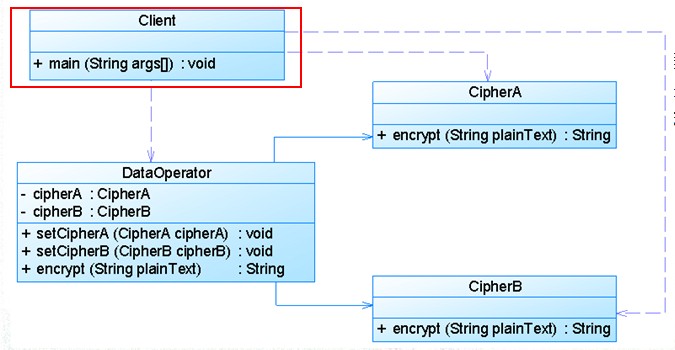
2.分析

i. 在软件中如果能够使用基类对象，那么一定能够使用其子类对象。把基类都替换成它的子类，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类的话，那么它不一定能够使用基类。

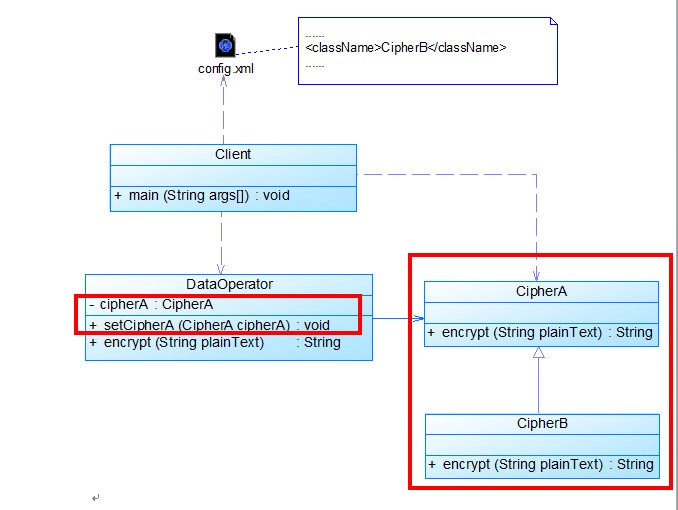
ii. 里氏代换原则是实现开闭原则的重要方式之一，由于使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

3. 实例

某系统需要实现对重要数据（如用户密码）的加密处理，在数据操作类(DataOperator)中需要调用加密类中定义的加密算法，系统提供了两个不同的加密类，CipherA和CipherB，它们实现不同的加密方法，在DataOperator中可以选择其中的一个实现加密操作。如图所示：



图（一）



图（二）

图（一）和图（二）分析：

图（一）为什到图（二）哪？因为如果需要更换一个加密算法类或者增加并使用一个新的加密算法类，如将CipherA改为CipherB，则需要修改客户类Client和数据操作类DataOperator的源代码，违背了开闭原则。现使用里氏代换原则对其进行重构，使得系统可以灵活扩展，符合开闭原则。

总结：子类型必须能够替换掉它们的父类型。

## 四、单一职责原则

1. 定义

i. 一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中。

ii. 就一个类而言，应该仅有一个引起它变化的原因。

2. 分析

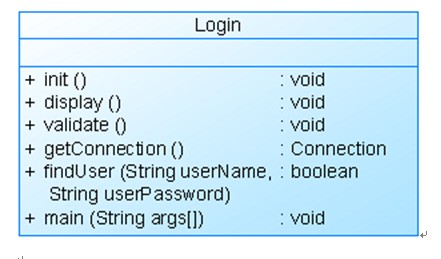
i. 一个类（或者大到模块，小到方法）承担的职责越多，它被复用的可能性越小，而且如果一个类承担的职责过多，就相当于将这些职责耦合在一起，当其中一个职责变化时，可能会影响其他职责的运作。

ii.类的职责主要包括两个方面：数据职责和行为职责，数据职责通过其属性来体现，而行为职责通过其方法来体现。

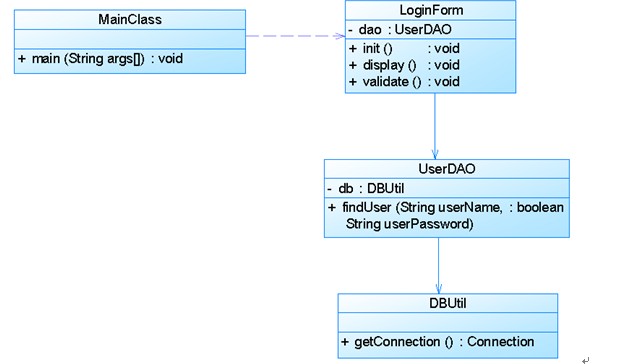
iii.单一职责原则是实现高内聚、低耦合的指导方针，在很多代码重构手法中都能找到它的存在，它是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关重构经验。

3. 实例

i. 某基于Java的C/S系统的“登录功能”通过如下登录类(Login)实现：



图（一）



图（二）

图一和图二有什么区别哪？

图（一）功能太过于集成，严重违反类的单一原则。

总结：就一个类而言，应该仅有一个引起它变化的原因。

## 五、接口隔离原则

1. 定义

i. 客户端不应该依赖那些它不需要的接口。

ii. 一旦一个接口太大，则需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。

2.分析

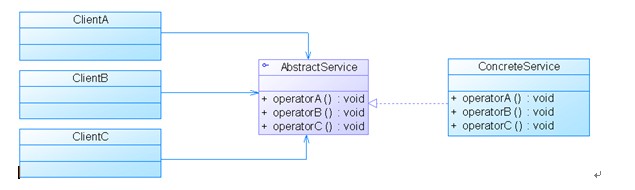
i. 接口隔离原则是指使用多个专门的接口，而不使用单一的总接口。每一个接口应该承担一种相对独立的角色，不多不少，不干不该干的事，该干的事都要干。

ii. 使用接口隔离原则拆分接口时，首先必须满足单一职责原则，将一组相关的操作定义在一个接口中，且在满足高内聚的前提下，接口中的方法越少越好。

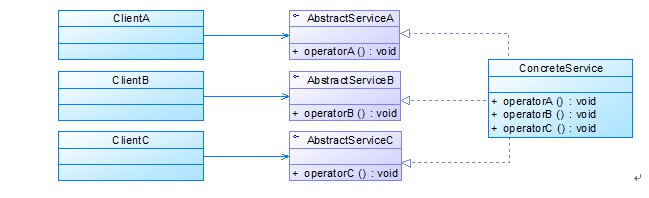
iii. 可以在进行系统设计时采用定制服务的方式，即为不同的客户端提供宽窄不同的接口，只提供用户需要的行为，而隐藏用户不需要的行为

3.实例

i. 下图展示了一个拥有多个客户类的系统，在系统中定义了一个巨大的接口（胖接口）AbstractService来服务所有的客户类。如图所示：



图（一）



图（二）

图（一）和图（二）分析：

图（一）为什么到图（二）哪？因为这样做既满足了接口隔离原则，又满足了单一原则，何乐而不为呢，但是也带来了很多的不便，类增多了。

总结：类应该完全依赖相应的专门的接口

六、合成复用原则

1.定义

i. 尽量使用对象组合，而不是继承来达到复用的目的。

2.分析

i. 合成复用原则就是指在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的。简言之：要尽量使用组合/聚合关系，少用继承。

ii. 在面向对象设计中，可以通过两种基本方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承。

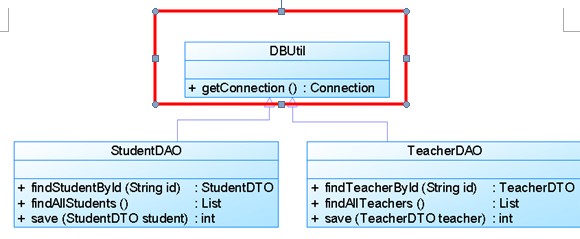
a) 继承复用：实现简单，易于扩展。破坏系统的封装性；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；只能在有限的环境中使用。（“白箱”复用 ）

b) 组合/聚合复用：耦合度相对较低，选择性地调用成员对象的操作；可以在运行时动态进行。（“黑箱”复用 ）

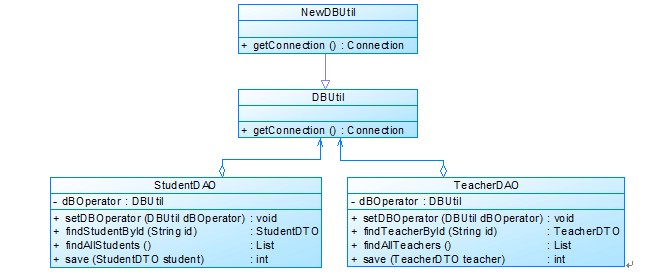
iii. 组合/聚合可以使系统更加灵活，类与类之间的耦合度降低，一个类的变化对其他类造成的影响相对较少，因此一般首选使用组合/聚合来实现复用；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。

3. 实例

i. 某教学管理系统部分数据库访问类设计如图所示：



图（一）



图（二）

图（一）和图（二）分析：

图（一）为什么到图（二）哪？因为如果需要更换数据库连接方式，如原来采用JDBC连接数据库，现在采用数据库连接池连接，则需要修改DBUtil类源代码。如果StudentDAO采用JDBC连接，但是TeacherDAO采用连接池连接，则需要增加一个新的DBUtil类，并修改StudentDAO或TeacherDAO的源代码，使之继承新的数据库连接类，这将违背开闭原则，系统扩展性较差。

总结：类中应用，尽量使用对象组合而不是用继承来达到复用的目的。

## 七、迪米特法则

1. 定义

i. 每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位。

2. 分析

i.迪米特法则就是指一个软件实体应当尽可能少的与其他实体发生相互作用。这样，当一个模块修改时，就会尽量少的影响其他的模块，扩展会相对容易，这是对软件实体之间通信的限制，它要求限制软件实体之间通信的宽度和深度。

ii. 狭义的迪米特法则：可以降低类之间的耦合，但是会在系统中增加大量的小方法并散落在系统的各个角落，它可以使一个系统的局部设计简化，因为每一个局部都不会和远距离的对象有直接的关联，但是也会造成系统的不同模块之间的通信效率降低，使得系统的不同模块之间不容易协调。

iii. 广义的迪米特法则：指对对象之间的信息流量、流向以及信息的影响的控制，主要是对信息隐藏的控制。信息的隐藏可以使各个子系统之间脱耦，从而允许它们独立地被开发、优化、使用和修改，同时可以促进软件的复用，由于每一个模块都不依赖于其他模块而存在，因此每一个模块都可以独立地在其他的地方使用。一个系统的规模越大，信息的隐藏就越重要，而信息隐藏的重要性也就越明显。

iv. 迪米特法则的主要用途在于控制信息的过载。

1.在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及

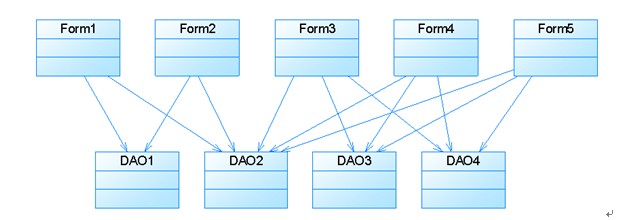
2.在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限

3.在类的设计上，只要有可能，一个类型应当设计成不变类

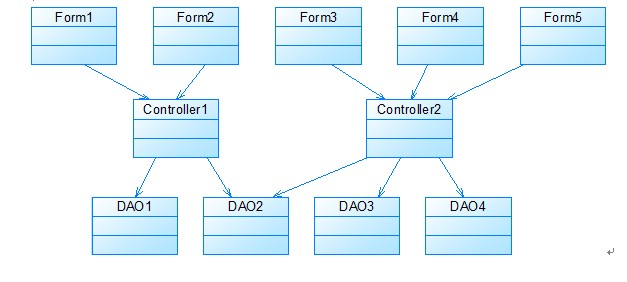
4.在对其他类的引用上，一个对象对其他对象的引用应当降到最低。

3. 实例

i.某系统界面类(如Form1、Form2等类)与数据访问类(如DAO1、DAO2等类)之间的调用关系较为复杂。如图所示：



图（一）



图（二）

图（一）和图（二）分析：

图（一）为什么到图（二）哪？因为这样就可以降低类的耦合性，是类中功能更加单一，相当于外观模式。

总结：一个软件实体应当尽可能少的与其他实体发生相互作用

综述：以上就是其中程序员必备的七大面向对象设计原则，原则犹如国家的法规，这就是面向对象的准则，要使我们的编程更加灵活，封装性更好，复用性更高，易于维护我们需要我们的这些原则建设我们规范我们的程序。这样我们编写的才是高质量的程序。我们每个人都这样去做，我们的编程水平一定会进步一大截！

# End