/\*1.单一职责原则

定义：不要存在多余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

问题由来：类T负责两个不同的职责：职责p1，职责p2.当由于职责p1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责p2

功能发生故障。

解决方案：遵循单一职责原则。分别建立两个类T1，T2， 使T1 完成职责p1功能,T2完成职责p2的功能。\*/

//没有完善的类设计：

class People{

public void readBook(String person)

{

System.out.println(person+"正在阅读电子书");

}

}

public class Test

{

public static void main(String [] args)

{

People p=new People();

p.readBook("学生");

p.readBook("讲师");

p.readBook("教授");

}

}

//分析：这样写后，某一天发现讲师，学生，教授并不是每个人都在看相同的电子书，修改后如果满足单一原则，可以把人类

//改为讲师类，学生类，教授类

//改善后：

class Student

{

public void readBook(String person)

{

System.out.println(person+"正在看电子书");

}

}

class Professor{

public void readBook(String person)

{

System.out.println(person+"正在看科研类书");

}

}

class Teacher{

public void readBook(String person)

{

System.out.println(person+"正在看课件相关书");

}

}

public class Test

{

public static void main(String [] args)

{

People p=new People();

p.readBook("学生");

p.readBook("讲师");

p.readBook("教授");

Professor pro=new Professor();

pro.readBook("教授")

}

}

//这样改进后每个类都负责各自的功能，尽管拆分后代码量较大，但在大型项目中还是适合遵循单一职责原则

/\*2里氏替换原则

定义：所有的引用基类的地方必须透明地使用其子类的对象。

即只要父类出现的地方子类能够出现，而且替换为子类不会产生任何错误或异常。

问题由来：有一功能p1,有类a完成。现需要将功能p1进行扩展，扩展后的功能为p，其中p由原有功能p1和p2功能组成。新功能p由类

类a的子类b来完成，则子类b在完成新功能p2的同时，有可能会导致原有功能p1发生故障

解决方案:当使用继承时，遵循里氏转换原则。类b继承类a,除添加新的方法完成功能p2外，尽量不要重写父类的方法，也尽量不要重载

父类a的方法。\*/

public abstract class ViewPoint{

public abstract void where();

}

public class xiaoming extends ViewPoint{

public void where(){

System.out.pirntln("欢迎来到张家界");

}

}

public class xiaohong extends ViewPoint{

public void where(){

System.out.println("欢迎来到桂林");

}

}

public class Person{

private ViewPoint view; //定义要旅游的景点

public void setViewPoint(ViewPoint view) //引用基类

{

this.view=view;

}

public void travelTo()

{

view.where();

}

}

public class Test{

public static void main(Sring [] args)

{

Person per=new Person();

per.setViewPoint(new xiaoming()); //这里可以传子类对象

per.travelTo();

}

}

//父类出现的地方可以用子类替换，反之则不行，因为子类的范围更大

/\*3依赖倒置原则

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；抽象不应该细节；细节应该抽象

中心思想是：面向接口编程

问题由来：类a直接依赖类b，假如要将类a改为依赖类c，则必须通过修改类a的代码达成。这种场景下，类a一般高层模块，负责

复杂的业务逻辑；类b和类c是低层模块，负责基本的原子操作；假如修改a,会给程序带来风险

解决方案：将类a修改为依赖接口，类b和类c各自实现接口，类a通过接口间接与类b或者类c发生联系，则会大大降低修改类a的几率

说明面向接口编程比面向实现要好的例子\*/

class Book{

public String getContent(){

return "这是一本小说"

}

}

class Student{

public void getBook(Book book)

{

System.out.println(book.getContent);

}

}

public class Test{

public static void main(String [] args)

{

Student stu=new Student();

stu.getBook(new Book);

}

}

//改善后

//引入一个接口 降低学生与书本的耦合度 让书本和报纸都去继承接口

interface IReader{

public String getContent();

}

class Newspaper implements IReader{

public String getContent()

{

System.out.println("我是报纸")

}

}

class Book implements IReader{

public Sring getContent(){

System.out.println("我是读物中的书")

}

}

class Student{

public void getBook(IReader ir)

{

System.out.println(ir.getContent)

}

}

public class Test{

public static void main(String [] args)

{

Student stu=new Student();

stu.getBook(new Newspaper);

stu.getBook(new Book);

}

}

//这样改后有更好的扩展性，可以通过接口IReader扩展类

/\*4接口隔离原则

定义：客户端不应该依赖它不需要的接口：一个类对另一个类的依赖建立在最小的接口上

问题由来：类a通过接口1依赖类b，类c通过接口1依赖类d,如果接口1对于类a和类b来说不是最小接口，则类b和类d必须去实现他们不需要的

方法

解决方案：将臃肿的接口1拆分为独立的几个接口，类a和类c分别与他们 需要的接口建立依赖关系，也就是采用接口隔离原则\*/

interface I{

public void method1();

public void method2();

public void method3();

public void method4();

public void method5();

}

class A{

public void depend1(I i)

{

i.method1();

}

public void depend2(I i)

{

i.method2();

}

public void depend3(I i)

{

i.method3();

}

}

class B implements I{

//对A类来说，method4.method5是不必须的.但这里b要实现所有的方法

}

class C {

public void depend1(I i)

{

i.method1();

}

public void depend2(I i)

{

i.method4();

}

public void depend3(I i)

{

i.method5();

}

}

class D implements I{

//同样的对c来说方法2和3不是必须的,但这里仍然要实现

}

public class Test{

public static void main(Sring [] args)

{

A a=new A();

a.depend1(new B()); //a通过接口i依赖b

........

}

}

//可以看出接口过于臃肿，只要接口出现的方法，不管对依赖于它的类是否有用都要实现，这样的设计不好

//改善后：将原有接口拆分

interface I1{

public void method1();

}

interface I2{

public void method2();

public void method3();

}

interface I3{

public void method4();

public void method5();

}

class A{

public void depend1(I1 i){

i.method1();

}

public void depend2(I2 i){

i.method2();

}

public void depend3(I2 i){

i.method3();

}

}

class B implements I1,I2{

//只要实现a所需的三个方法就行

}

class D implements I1,I3{

}

/\*5迪米特法则

定义：一个对象应该对其他对象保存最少的了解

问题由来：类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也更大

解决方案：尽量降低类与类之间的耦合\*/

public class Someone {

public void operation1(Friend friend) {

Stranger stranger = friend.provideStranger();

stranger.operation3();

}

}

public class Friend {

private Stranger stranger = new Stranger();

public void operation2() {

}

public Stranger provideStranger() {

return stranger;

}

}

改善后：

public class Someone {

public void operation1(Friend friend) {

friend.forward();

}

}

public class Friend {

private Stranger stranger = new Stranger();

public void operation2() {

}

public void forward {

stranger.operation3();

}

}

/\*6开闭原则

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

问题由来：在软件的生命周期内，因为变化，升级和维护等原因需要对软件原有代码进行修改时，可能会给旧代码中引入错误，也可能使我们

不得不对整个功能重构，并且原有代码经过重新测试

解决方案：当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是修改；已有的代码来实现变化。

**代码1. 这个类负责了太多的事**

public class OrderProcessingModule {  
  public void Process(OrderStatusMessage orderStatusMessage) {  
    // 从配置文件中读取连接字符串  
    string connectionString =   
      ConfigurationManager.ConnectionStrings["Main"].ConnectionString;  
  
    Order order = null;  
  
    using (SqlConnection connection =   
      new SqlConnection(connectionString)) {  
      // 从数据库中获取一些数据  
      order = fetchData(orderStatusMessage, connection);  
    }  
  
    // 向来自于OrderStatusMessage的订单提交变更  
    updateTheOrder(order);  
  
    // 国际订单有一些特定的规则  
    if (order.IsInternational) {  
      processInternationalOrder(order);  
    }  
  
    // 对于大批量订单我们需要特别处理  
    else if (order.LineItems.Count > 10) {  
      processLargeDomesticOrder(order);  
    }  
    // 小的国内订单也需要区别处理  
    else {  
      processRegularDomesticOrder(order);  
    }  
  
    // 如果订单准备好了就发货  
    if (order.IsReadyToShip()) {  
      ShippingGateway gateway = new ShippingGateway();  
  
      // 将订单对象提交运送  
      ShipmentMessage message = createShipmentMessageForOrder(order);  
      gateway.SendShipment(message);  
  }  
}

OrderProcessingModule真是太忙了。它要进行数据访问、获取配置文件信息、为订单处理执行业务规则(可能本身就非常复杂)，并且将完成的订单转移出货。通常的情况是，如果你通过这种方式创建了OrderProcessingModule，你将会经常深入到这段代码中进行修改。而许多系统需求的变化也会造成OrderProcessingModule的代码产生非常多的变更，让系统变得岌岌可危并使变更花费很大代价。

除了这种一大块代码的方式，你应该遵循单一责任原则，将整个OrderProcessingModule分成一系列相关类的子系统，每一个类完成它自己特定的职责。举个例子，你可以将所有数据访问的功能放到一个新类中，管它叫OrderDataService，而把Order的业务逻辑放到另一个类中(我会在下一节进行更详细的讲述)。

根据开闭原则，通过将业务逻辑和数据访问的职责划分到不同的类中，你将可以独立地改变它们中的一个而不会影响到另一个。数据库物理部署的变化可能将使你把数据访问部分完全更换掉(对扩展开放)，然而订单逻辑类依然没有任何改动(对变更关闭)。

单一责任原则的要点不仅仅是写一些更小的类和方法。它的要点是每一个类应该实现一系列紧密相关的功能。遵循单一责任原则的最简单办法就是不断地问自己是不是这个类的每一个方法和操作都与这个类的名称直接相关。如果你找到了一些方法与这个类的名称不相称，你可以考虑将这些方法移到另一个类中。

/\*7合成复用原则

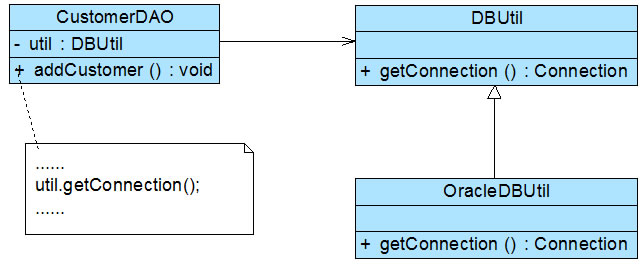
定义：尽量使用对象组合，而不是继承来达到复用的目的。

  合成复用原则就是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功能的目的。简言之：复用时要尽量使用组合/聚合关系（关联关系），少用继承。

      在面向对象设计中，可以通过两种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，但首先应该考虑使用组合/聚合，组合/聚合可以使系统更加灵活，降低类与类之间的耦合度，一个类的变化对其他类造成的影响相对较少；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。

|  |
| --- |
| Sunny软件公司开发人员在初期的CRM系统设计中，考虑到客户数量不多，系统采用**[MySQL](http://lib.csdn.net/base/14" \o "MySQL知识库" \t "_blank)**作为**[数据库](http://lib.csdn.net/base/14" \o "MySQL知识库" \t "_blank)**，与数据库操作有关的类如CustomerDAO类等都需要连接数据库，连接数据库的方法getConnection()封装在DBUtil类中，由于需要重用DBUtil类的getConnection()方法，设计人员将CustomerDAO作为DBUtil类的子类，初始设计方案结构如图1所示：  http://my.csdn.net/uploads/201205/14/1336930023_1487.jpg  **图1  初始设计方案结构图**        随着客户数量的增加，系统决定升级为Oracle数据库，因此需要增加一个新的OracleDBUtil类来连接Oracle数据库，由于在初始设计方案中CustomerDAO和DBUtil之间是继承关系，因此在更换数据库连接方式时需要修改CustomerDAO类的源代码，将CustomerDAO作为OracleDBUtil的子类，这将违反开闭原则。【当然也可以修改DBUtil类的源代码，同样会违反开闭原则。】        现使用合成复用原则对其进行重构。 |

      根据合成复用原则，我们在实现复用时应该多用关联，少用继承。因此在本实例中我们可以使用关联复用来取代继承复用，重构后的结构如图2所示：



**图2  重构后的结构图**

      在图2中，CustomerDAO和DBUtil之间的关系由继承关系变为关联关系，采用依赖注入的方式将DBUtil对象注入到CustomerDAO中，可以使用构造注入，也可以使用Setter注入。如果需要对DBUtil的功能进行扩展，可以通过其子类来实现，如通过子类OracleDBUtil来连接Oracle数据库。由于CustomerDAO针对DBUtil编程，根据里氏代换原则，DBUtil子类的对象可以覆盖DBUtil对象，只需在CustomerDAO中注入子类对象即可使用子类所扩展的方法。例如在CustomerDAO中注入OracleDBUtil对象，即可实现Oracle数据库连接，原有代码无须进行修改，而且还可以很灵活地增加新的数据库连接方式。