**一.建造者模式**

**建造者模式的理解要点就是零件生产和产品组装的过程,例如车有宝马和奔驰两种品牌，但它们都有轮子，发动机，都有这些东西，抽象出来就是我定义一个接口builder，接口规定了要造发动机，要造轮子，各个厂商必须按这个来，具体怎么造，各厂商自己实现。零件造完了还要组装成车，具体先组装什么再组装什么各不相同 ，这个本例中就是director，不同的组装顺序对应不同的组装director。**

**public class**Engine {

Double **power**;  
 String **brand**;  
 **public**Engine(Double power, String brand) {  
 **this**.**power**= power;  
 **this**.**brand**= brand;  
 }  
 **public**Double getPower() {  
 **return power**;  
 }  
 **public void**setPower(Double power) {  
 **this**.**power**= power;  
 }  
  
 **public**String getBrand() {  
 **return brand**;  
 }  
 **public void**setBrand(String brand) {  
 **this**.**brand**= brand;  
 }  
}  
  
**public class**Wheel {  
 Double **size**;  
 String **brand**;  
 **public**Wheel(Double size, String brand) {  
 **this**.**size**= size;  
 **this**.**brand**= brand;  
 }  
 **public**Double getSize() {  
 **return size**;  
 }  
 **public void**setSize(Double size) {  
 **this**.**size**= size;  
 }  
 **public**String getBrand() {  
 **return brand**;  
 }  
 **public void**setBrand(String brand) {  
 **this**.**brand**= brand;  
 }  
}  
  
**public class**Body {  
 Double **length**;  
 Double **width**;  
 **public**Body(Double length, Double width) {  
 **this**.**length**= length;  
 **this**.**width**= width;  
 }  
 **public**Double getLength() {  
 **return length**;  
 }  
 **public void**setLength(Double length) {  
 **this**.**length**= length;  
 }  
 **public**Double getWidth() {  
 **return width**;  
 }  
 **public void**setWidth(Double width) {  
 **this**.**width**= width;  
 }  
}  
  
**public class**Car {  
 Engine **engine**;  
 Body **body**;  
 Wheel **wheel**;  
 **public**Engine getEngine() {  
 **return engine**;  
 }  
 **public void**setEngine(Engine engine) {  
 **this**.**engine**= engine;  
 }  
 **public**Body getBody() {  
 **return body**;  
 }  
 **public void**setBody(Body body) {  
 **this**.**body**= body;  
 }  
 **public**Wheel getWheel() {  
 **return wheel**;  
 }  
 **public void**setWheel(Wheel wheel) {  
 **this**.**wheel**= wheel;  
 }  
}  
  
  
  
**public interface**Builder {  
 **public void**buildWheel();  
 **public void**buildEngine();  
 **public void**buildBody();  
 **public**Car getCar();  
}  
  
**public class**BenzBuilder **implements**Builder{  
 Car **car**=**new**Car();  
@Override  
 **public void**buildWheel() {  
 **car**.setWheel(**new**Wheel(10.0,**"奔驰"**));  
 }  
@Override  
 **public void**buildEngine() {  
 **car**.setBody(**new**Body(100.0,200.0));  
 }  
@Override  
 **public void**buildBody() {  
 **car**.setEngine(**new**Engine(2.5,**"奔驰"**));  
 }  
@Override  
 **public**Car getCar() {  
 **return car**;  
 }  
}   
  
  
**public class**Director {  
 **public void**construct(Builder builder){  
 builder.buildWheel( );  
 builder.buildBody();  
 builder.buildEngine();  
 }  
}  
  
  
**public class**Main {  
 **public static void**main(String[] args) {  
 Builder builder=**new**BenzBuilder();  
 Director director=**new**Director();  
 director.construct(builder);  
 Car car=builder.getCar();  
 System.***out***.println(**"body:"**+car.getBody().getLength()+**" "**+car.getBody().getWidth()+**" engine:"**+car.getEngine().getBrand()+**" "**+car.getEngine().getPower()+**" wheel:"**+car.getWheel().getBrand()+**" "**+car.getWheel().getSize());  
}  
}   
  
  
  
运行结果：  
body:100.0 200.0 engine:奔驰 2.5 wheel:奔驰 10.0   
  
  
 如果有不同的建造工序，只需要新建一个不同的director   ,如果有不同的建造方法，只需要新建一个不同的builder实现类

**过滤器模式**

**public class**Request {  
String **value**;  
**public**String getValue() {  
**return value**;  
}  
**public void**setValue(String value) {  
**this**.**value**= value;  
}

}   
  
  
**public class**Response {  
    String **value**;  
**public**String getValue() {  
**return value**;  
}  
**public void**setValue(String value) {  
**this**.**value**= value;  
}  
}  
  
  
**public interface**Filter {

**public void**doFilter(Request request,Response response,FilterChain chain);  
}  
  
  
 **public class**FilterChain { List<Filter> **filterList**=**new**ArrayList<Filter>();  
**int index**=0;  
**public void**addFilter(Filter filter){  
**filterList**.add(filter);  
}  
**public void**doFilter(Request request,Response response){  
**if**(**index**<**filterList**.size()) {  
Filter filter = **filterList**.get(**index**);  
**index**++;  
filter.doFilter(request, response, **this**);  
}  
}  
}  
  
  
  
  
  
**public class**MyFilter1 **implements**Filter {  
Filter **filter**;  
  
  
@Override  
**public void**doFilter(Request request, Response response, FilterChain chain) {  
System.***out***.println(**"-------myFilter1---------"**);  
chain.doFilter(request,response);  
}  
}   
  
  
  
**public class**MyFilter2 **implements**Filter {  
@Override  
**public void**doFilter(Request request, Response response, FilterChain chain) {  
System.***out***.println(**"-----MyFilter2--------"**);  
chain.doFilter(request,response);  
}  
}   
  
**public class**Main {  
**public static void**main(String[] args) {  
Request request=**new**Request();  
request.setValue(**"request"**);  
Response response=**new**Response();  
response.setValue(**"response"**);  
Filter filter1=**new**MyFilter1();  
Filter filter2=**new**MyFilter2();  
FilterChain filterChain=**new**FilterChain();  
filterChain.addFilter(filter1);  
filterChain.addFilter(filter2);  
filterChain.doFilter(request,response);  
}  
}  
运行结果  
 -------myFilter1---------

-----MyFilter2--------

**职责链模式**

//处理者接口  
 **public interface**Handler {

**public void**handle(Request request);  
}  
  
//处理者实现类  
**public class**DepartmentLeaderHandler **implements**Handler {  
Handler **handler**;  
  
**public**DepartmentLeaderHandler(Handler handler) {  
**this**.**handler**= handler;  
}  
  
@Override  
**public void**handle(Request request) {  
  
System.***out***.println(**"部门领导处理"**);  
**if**(**handler**!=**null**){  
**handler**.handle(request);  
}  
}  
}  
  
  
//处理者实现类  
**public class**ViceGeneralHandler **implements**Handler {  
Handler **handler**;  
  
**public**ViceGeneralHandler(Handler handler) {  
**this**.**handler**= handler;  
}  
  
**public**Handler getHandler() {  
**return handler**;  
}  
  
**public void**setHandler(Handler handler) {  
**this**.**handler**= handler;  
}  
  
@Override  
**public void**handle(Request request) {  
System.***out***.println(**"副总审批"**);  
**if**(**handler**!=**null**){  
**handler**.handle(request);  
}  
}  
}  
  
//处理者实现类  
 **public class**GeneralHandler **implements**Handler { Handler **handler**;  
**public**GeneralHandler(Handler handler) {  
**this**.**handler**= handler;  
}  
  
@Override  
**public void**handle(Request request) {  
  
System.***out***.println(**"总经理审批"**);  
**if**(**handler**!=**null**){  
**handler**.handle(request);  
}  
}  
}  
  
**public class**Main {  
**public static void**main(String[] args) {  
Request request=**new**Request(2);  
DepartmentLeaderHandler departmentLeaderHandler=**new**DepartmentLeaderHandler(**new**ViceGeneralHandler(**new**GeneralHandler(**null**)));  
departmentLeaderHandler.handle(request);  
}  
}

**策略模式**

//策略接口   
**public interface**TravelStrategy {

**public void**travel();  
}  
  
//不同的策略算法，如本例中是乘公交  
 **public class**BusStrategy **implements**TravelStrategy {  
@Override  
**public void**travel() {  
System.***out***.println(**"乘公交策略"**);  
}  
}  
//不同的策略算法，如本例中是乘汽车  
 **public class**CarStrategy **implements**TravelStrategy { @Override  
**public void**travel() {  
System.***out***.println(**"开汽车策略"**);  
}  
}  
  
//上下文，使用算法的对象  
 **public class**Context { TravelStrategy **strategy**;  
  
**public**Context(TravelStrategy strategy) {  
**this**.**strategy**= strategy;  
}  
  
**public**TravelStrategy getStrategy() {  
**return strategy**;  
}  
  
**public void**setStrategy(TravelStrategy strategy) {  
**this**.**strategy**= strategy;  
}  
  
**public void**travel(){  
**strategy**.travel();  
}  
}  
  
  
 **public class**Main { **public static void**main(String[] args) {  
TravelStrategy busStrategy=**new**BusStrategy();  
Context context=**new**Context(busStrategy);  
context.travel();  
TravelStrategy carStrategy=**new**CarStrategy();  
context.setStrategy(carStrategy );  
context.travel();  
}  
}  
  
  
策略模式的作用就是将不同算法封装起来，然后在需要使用不同算法的情况下，通过简单的构造方法参数传入，或set 方法传入就可以轻松实现使用不同算法做同一件事件，如本例中是travel

关于Strategy模式的故事讲到这里，应该基本OK啦！下面我们再聊些更高层次的东西。什么是更高层次的东西？嘿！当然是设计原则了！在前面总结Strategy模式的优点的时候我们提到过，Strategy模式不仅保留了继承的优点，而且还提供了更灵活的扩展能力。为什么会这样呢？Strategy模式是怎么做到这一点的呢？哈！这是因为它“上面有人”啊！谁啊？它就是我们下面要介绍的重量级设计原则：

Favor composition over inheritance.（优先使用对象组合，而非类继承）

关于组合和继承，我们只要这样来理解即可：组合是一种“HAS-A”关系，而继承是一种“IS-A”关系。很明显“HAS-A”要比“IS-A”更灵活一些。也就是说在创建系统的时候，我们应该优先使用对象组合，因为它不仅可以给你提供更多灵活性和扩展性，而且还使你可以在运行时改变行为(组合不同的对象)，这简直是酷毙了！但是也不是说继承就是不能用，只是说应该把继承应用在相对更稳定，几乎没有变化的地方，例如前面的Duck类里的Swim()方法，因为可以肯定所有鸭子一定都会游泳，所以就没有必要给这个行为提供基于Strategy模式的实现方式，因为那样做除了是程序更复杂以外，没有什么意义。

**模板模式**

**package**template;

**public abstract class**AbstractPersonTemplate {  
**public void**getup(){  
System.***out***.println(**"起床"**);  
}  
**public abstract void**work();   //实现各异的方法  
**public void**sleep(){  
System.***out***.println(**"睡觉"**);  
}  
  
**public void**oneDayWork(){  
getup();  
work();  
sleep();  
}  
  
}  
  
  
  
  
  
**public class**Student **extends**AbstractPersonTemplate {  
@Override  
**public void**work() {  
System.***out***.println(**"学习"**);  
}  
}  
  
  
**public class**Teacher **extends**AbstractPersonTemplate {  
@Override  
**public void**work() {  
System.***out***.println(**"教书"**);  
}  
}

**public class**Main {  
**public static void**main(String[] args) {  
AbstractPersonTemplate student=**new**Student();  
student.oneDayWork();  
AbstractPersonTemplate teacher=**new**Teacher();  
teacher.oneDayWork();  
}  
}

# 模板方法模式中的方法

　　模板方法中的方法可以分为两大类：模板方法和基本方法。

## 模板方法

　　一个模板方法是定义在抽象类中的，把基本操作方法组合在一起形成一个总算法或一个总行为的方法。

　　一个抽象类可以有任意多个模板方法，而不限于一个。每一个模板方法都可以调用任意多个具体方法。

## 基本方法

　　基本方法又可以分为三种：抽象方法(Abstract Method)、具体方法(Concrete Method)和钩子方法(Hook Method)。

　　●　　**抽象方法：**一个抽象方法由抽象类声明，由具体子类实现。在Java语言里抽象方法以abstract关键字标示，需要变动的方法。

　　●　　**具体方法：**一个具体方法由抽象类声明并实现，而子类并不实现或置换，即不需要变动的方法。

　　●　　**钩子方法：**一个钩子方法由抽象类声明并实现，而子类会加以扩展。通常抽象类给出的实现是一个空实现，作为方法的默认实现。

　　在上面的例子中，AbstractTemplate是一个抽象类，它带有三个方法。其中abstractMethod()是一个抽象方法，它由抽象类声明为抽象方法，并由子类实现；hookMethod()是一个钩子方法，它由抽象类声明并提供默认实现，并且由子类置换掉。concreteMethod()是一个具体方法，它由抽象类声明并实现。

## 默认钩子方法

　　一个钩子方法常常由抽象类给出一个空实现作为此方法的默认实现。这种空的钩子方法叫做“Do Nothing Hook”。显然，这种默认钩子方法在缺省适配模式里面已经见过了，一个缺省适配模式讲的是一个类为一个接口提供一个默认的空实现，从而使得缺省适配类的子类不必像实现接口那样必须给出所有方法的实现，因为通常一个具体类并不需要所有的方法。

**命令模式**

这个很哈理解，命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉Struts的同学应该知道，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！  
  
 ◆很容易构造一个命令队列

　　◆记录相关的命令日志

　　◆增加命令的状态，实现命令的撤销和重做

　　◆允许接受请求的一方决定是否可做

　　◆新的命令轻而易举可以加入其中

　　缺点：可能会有过多的具体命令类存在

**public class**Recevier {

**public void**action(){  
     System.***out***.println(**"do action"**);  
}  
}  
  
  
**public interface**Command {  
**public void**exec();  
}  
  
**public class**MyCommand **implements**Command{  
  
    Recevier **recevier**;  
  
**public**MyCommand(Recevier recevier) {  
**this**.**recevier**= recevier;  
}  
  
@Override  
**public void**exec() {  
**recevier**.action();  
}  
}

**public class**Invoker {  
       Command **command**;  
  
**public**Invoker(Command command) {  
**this**. **command**= command;  
}  
  
**public void**invoke(){  
**command**.exec();  
}  
}

**public class**Main {  
**public static void**main(String[] args) {  
      Recevier recevier=**new**Recevier();  
      Command command=**new**MyCommand(recevier);  
      Invoker invoker=**new**Invoker(command);  
      invoker.invoke();  
}  
}

结果：

do action

**访问者模式**

访问者模式把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。访问者模式适用于数据结构相对稳定算法又易变化的系统。因为访问者模式使得算法操作增加变得容易。若系统数据结构对象易于变化，经常有新的数据对象增加进来，则不适合使用访问者模式。访问者模式的优点是增加操作很容易，因为增加操作意味着增加新的访问者。访问者模式将有关行为集中到一个访问者对象中，其改变不影响系统数据结构。其缺点就是增加新的数据结构很困难。—— From 百科

简单来说，访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。

来看看原码：一个Visitor类，存放要访问的对象，

访问者模式，  
**public interface**Visitor {

**public void**visit(Subject subject);  
}  
  
//定义算法  
 **public class**MyVisitor **implements**Visitor {   
@Override  
**public void**visit(Subject subject) {  
System.***out***.println(**"do some arithmetic to subject"**);  
}  
}  
  
**public interface**Subject {  
**public void**accept(Visitor visitor);  
}   
//定义数据结构  
**public class**MySubject **implements**Subject {  
@Override  
**public void**accept(Visitor visitor) {  
visitor.visit(**this**);  
}  
}   
  
  
**public class**Main {  
**public static void**main(String[] args) {  
Subject subject=**new**MySubject();  
Visitor visitor=**new**MyVisitor();  
subject.accept(visitor);  
}  
}

**状态模式**

**状态（state）模式**：状态模式的意图是，允许一个对象在其内部状改变时改变它的行为。看起来就像是改变了它的类一样。主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同的一系列类当中，可以把复杂的逻辑判断简单化。  
  
  
状态模式就是根据传入的不同的状态，做不同的事件   
**public interface**State {

**public void**doAction();  
}  
  
  
 **public class**StartState **implements**State {   
@Override  
**public void**doAction() {  
        System.***out***.println(**"do start action"**);  
}  
}  
  
 **public class**EndState **implements**State {   
@Override  
**public void**doAction() {  
        System.***out***.println(**"do end action"**);  
}  
}  
  
  
  
**public class**Context **implements**State {  
        State **state**=**null**;  
  
**public**State getState() {  
**return state**;  
}  
  
**public void**setState(State state) {  
**this**.**state**= state;  
}  
  
  
@Override  
**public void**doAction() {  
**state**.doAction();  
}  
}  
  
  
**public class**Main {  
**public static void**main(String[] args) {  
        State startState =**new**StartState();  
        State endState=**new**EndState();  
        Context context=**new**Context();  
        context.setState(startState);  
        context.doAction();  
  
        context.setState(endState);  
        context.doAction();  
  
}  
}    
  
结果

do  start     action

do end action

备忘录模式

//状态所有者，它需要两个方法，一：创建备忘录，二：恢复备忘录   
**public class**Originator {

**String state;  
  
Memento createMemento(){  
        return new Memento(state);  
}  
  
public void restoreMemento(Memento memento){  
        this.state=memento.getState();  
}  
  
  
public String getState() {  
        return state;  
}  
  
public void setState(String state) {  
        this.state = state;  
}  
  
public void printState(){  
        System.out.println(state);  
}  
}**

**备忘录模式**

//备忘录，备记录不提供set方法，不允许修改， 而且对客户端只不可见的  
**public class**Memento {  
**private**String **state**;  
  
**public**Memento(String state) {  
**this**.**state**= state;  
}  
**public**String getState() {  
**return state**;  
}  
}  
   
  
//管理者，负责保存备忘录  
**public class**CareTaker {  
**private**Memento **memento**;  
**public**Memento retrieveMemento(){  
**return memento**;  
}  
  
**public void**saveMemento(Memento memento){  
**this**.**memento**=memento;  
}  
  
}   
  
  
**public class**Main {  
**public static void**main(String[] args) {  
*//主体状态*Originator originator=**new**Originator();  
*//管理者*CareTaker careTaker=**new**CareTaker();  
        originator.setState(**"on"**);  
        originator.printState();  
*//保存状态*careTaker.saveMemento(originator.createMemento());  
        originator.setState(**"off"**);  
        originator.printState();  
*//恢复状态*originator.restoreMemento(careTaker.retrieveMemento());  
        originator.printState();  
}  
}   
  
运行结果：

on

off

on

**桥接模式**

当一个业务拥有多个维度时，且这些维度组合不固定时，就需要用桥接模式。 本例中的维度三拥有另外两个维度的引用，这样的话可以达到组合，试想如果不用桥接模式的话，每个维度有10种那三个维度组合就是1000种，实际程序中不可能写一千个这样的类吧。如果用桥接模式，当增加一个维度后，可以很轻松的完成程序的扩展  
  
维度一

**public interface**Transport {

**public void**transport();  
}  
  
---------------------------------------------------------  
  
**public class**Passenger **implements**Transport{  
@Override  
**public void**transport() {  
            System.***out***.println(**"passenger "**);  
}  
}  
  
----------------------------------------------------------  
**public class**Goods **implements**Transport {  
@Override  
**public void**transport() {  
                System.***out***.println(**"goods "**);  
}  
}

维度二

**public interface**Road {  
**public void**run();  
}

----------------------------------------------------------

**public class**SpeedWay **implements**Road{  
@Override  
**public void**run() {  
                System.***out***.println(**"speedway"**);  
}  
}

----------------------------------------------------------

**public class**CommonWay **implements**Road {  
@Override  
**public void**run() {  
                System.***out***.println(**"commonway"**);  
}  
}

维度三  
**public class**Vihicle {  
//拥有维度一的引用  
**private**Transport **implementator**;  
//拥有维度二的引用  
**private**Road **road**;  
**public**Vihicle(Transport implementator,Road road) {  
**this**.**implementator**= implementator;  
**this**.**road**=road;  
}  
  
**public void**transport(){  
**implementator**.transport();  
**road**.run();  
}  
}

----------------------------------------------------------  
**public class**Jeep **extends**Vihicle {  
**public**Jeep(Transport implementator,Road road) {  
**super**(implementator,road);  
}  
  
@Override  
**public void**transport() {  
                System.***out***.println(**"jeep"**);  
**super**.transport();  
}

｝  
----------------------------------------------------------  
**public class**Car **extends**Vihicle {  
**public**Car(Transport implementator,Road road) {  
**super**(implementator,road);  
}  
  
@Override  
**public void**transport() {  
            System.***out***.println(**"car"**);  
**super**.transport();  
}  
}

**public class**Main {  
**public static void**main(String[] args) {  
        Passenger passenger=**new**Passenger();  
        SpeedWay speedWay=**new**SpeedWay();  
        Vihicle car=**new**Car(passenger,speedWay);  
        car.transport();  
  
        Goods goods=**new**Goods();  
        CommonWay commonWay=**new**CommonWay();  
        Vihicle jeep=**new**Jeep(goods,commonWay);  
        jeep.transport();  
}  
}

**抽象工厂模式**

抽像工厂模式的好处就是当添加一个新的品种时，不需要修改原来的任何代码，只需要增加一个新品种的工厂类就行了，在简单工厂模式在新增加一个品种时需要修改原来的工厂类这是主要区别，相比而言，抽像工厂模式更具拓展性.

**public interface**Traffic {

**public void**run();  
}  
  
**public class**Car **implements**abstractfactory.Traffic {  
@Override  
**public void**run() {  
System.***out***.println(**"car run"**);  
}  
}  
  
**public class**Jeep **implements**abstractfactory.Traffic {  
@Override  
**public void**run() {  
System.***out***.println(**"jeep run"**);  
}  
  
}  
  
  
  
**public interface**Provider {  
**public**Traffic getTraffic();  
}  
  
  
**public class**CarFactory **implements**Provider {  
@Override  
**public**Traffic getTraffic() {  
**return new**Car();  
}  
  
}

**public class**JeepFactory **implements**Provider {  
@Override  
**public**Traffic getTraffic() {  
**return new**Jeep();  
}  
}

**public class**Main {  
**public static void**main(String[] args) {  
Provider jeepFactory=**new**JeepFactory();  
Traffic jeep=jeepFactory.getTraffic();  
jeep.run();  
  
  
Provider carFactory=**new**CarFactory();  
Traffic car=carFactory.getTraffic();  
car.run();  
}  
}

**简单工厂模式**

简单工厂模式  
 **public interface**Traffic {

**public void**run();  
}  
  
  
  
**public class**Car **implements**Traffic {  
@Override  
**public void**run(){  
    System.***out***.println(**"car run"**);  
}  
}  
  
  
**public class**Jeep **implements**Traffic {  
@Override  
**public void**run() {  
    System.***out***.println(**"jeep run"**);  
}  
}    
  
  
**public class**TrafficFactory {  
**public static**Traffic getTraffic(String type){  
**if**(**"car"**.equals(type)){  
**return new**Car();  
    }**else if**(**"jeep"**.equals(type)){  
**return new**Jeep();  
    }  
**return null**;  
    }  
}   
  
  
  
**public class**Main {  
**public static void**main(String[] args) {  
    Traffic car=TrafficFactory.*getTraffic*(**"car"**);  
    Traffic jeep=TrafficFactory.*getTraffic*(**"jeep"**);  
    car.run();  
    jeep.run();  
}  
}

**门面模式**

门面模式相当简单，思路就是将多个对象的功能集中在一个对象中，向外不过多暴露太多内部方法，  
**public class**A {

**public void**doSomethingA(){  
    System.***out***.println(**" do something A"**);  
}  
}  
  
  
  
**public class**B {  
**public void**doSomethingB(){  
System.***out***.println(**"do something B"**);  
}  
}   
  
  
  
**public class**C {  
**public void**doSomethingC(){  
System.***out***.println(**"do something C"**);  
}  
}   
  
  
  
  
**public class**Facade {  
A **a**=**new**A();  
B **b**=**new**B();  
C **c**=**new**C();  
**public void**doSomethingA(){  
**a**.doSomethingA();  
}  
**public void**doSomethingB(){  
**b**.doSomethingB();  
}  
**public void**doSomethingC(){  
**c**.doSomethingC();  
}  
}

**观察者模式**

观察者模式可以实现当某一个对象的状态发生变化， 可以将这种变化通知到其它的对象，当然前提其它对象要注册在该对象上，最典型的应用 就是点击一个按钮触发一个或多个监听器  
  
  
**public interface**Subject {

       //这里要将观察者注册到目标对象

**void**registerObserver(Observer observer);

     //将观察者从目标对象中移除  
**void**detachObserver(Observer o);

     //通知所有观察者  
**void**notifyObservers();  
  
}  
  
  
  
**public interface**Observer {  
**public void**update();  
}  
  
  
**public class**Hunter **implements**Subject {  
        List<Observer> **observerList**=**new**ArrayList<Observer>();  
        @Override  
**public void**registerObserver(Observer observer) {  
**observerList**.add(observer);  
        }  
  
        @Override  
**public void**detachObserver(Observer o) {  
**observerList**.remove(o);  
        }  
  
        @Override  
**public void**notifyObservers() {  
**for**(Observer observer:**observerList**){  
                        observer.update();  
                }  
            }  
}  
  
  
  
**public class**ObserverImpl **implements**Observer {  
        String **name**;  
  
**public**ObserverImpl(String name) {  
**this**.**name**= name;  
}  
  
@Override  
**public void**update() {  
        System.***out***.println(**name**+**" is notified"**);  
}  
}

**装饰者模式**

装饰者模式是在某一个目标方法前后装饰一层或多层其它业务逻辑，具体的代码如下：  
最典型的应用就是Java IO中的应用   
**public class**Employee **implements**Project {

@Override  
**public void**doCoding() {  
        System.***out***.println(**"代码工人写代码"**);  
}  
}  
  
  
**public class**Manager **implements**Project {  
        Project **project**;  
  
**public**Manager(Project project) {  
**this**.**project**= project;  
}  
  
@Override  
**public void**doCoding() {  
        System.***out***.println(**"项目经理启动项目"**);  
**project**.doCoding();  
        System.***out***.println(**"项目经理验收项目"**);  
}  
}  
  
**public class**ProductManager **implements**Project {  
        Project **project**;  
  
**public**ProductManager(Project project) {  
**this**.**project**= project;  
}  
  
  
@Override  
**public void**doCoding() {  
        System.***out***.println(**"产品经理进行需要分析"**);  
**project**.doCoding();  
        System.***out***.println(**"产品经理验证需求是否满足要求"**);  
  
}  
}   
  
  
  
**public class**Main {  
  
**public static void**main(String[] args) {  
        //是不是很像Java的IO流  
        ProductManager projectManager=**new**ProductManager(**new**Manager(**new**Employee()));  
        projectManager.doCoding();  
}  
}

**适配器模式**

# java适配器模式

Adapter模式也叫适配器模式，是由GoF提出的23种设计模式的一种。Adapter模式是构造型模式之一，通过Adapter模式，可以改变已有类(或外部类)的接口形式。

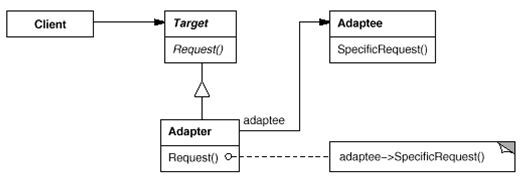
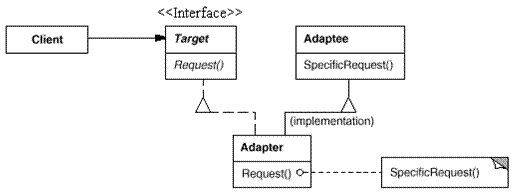
　　在大规模的系统开发过程中，我们常常碰到诸如以下这些情况：

　　我们需要实现某些功能，这些功能已有还不太成熟的一个或多个外部组件，如果我们自己重新开发这些功能会花费大量时间;所以很多情况下会选择先暂时使用外部组件，以后再考虑随时替换。但这样一来，会带来一个问题，随着对外部组件库的替换，可能需要对引用该外部组件的源代码进行大面积的修改，因此也极可能引入新的问题等等。***如何最大限度的降低修改面呢***？

　　Adapter模式就是针对这种类似需求而提出来的。

# 

定义：  
属于结构型模式，其主要作用是将一个类的接口转换成客户希望的另外一个接口。适配器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。  
比如你手机只有2.5mm接口(貌似就Nokia干的出来)，但你只能买到3.5mm的，这时就需要买个适配器了(─.─|||  
意图宗旨：  
保留现有类所提供的服务，向客户提供接口，以满足客户的期望  
什么时候可以使用它：  
1你想使用一个已经存在的类，而它的接口不符合你的需求。没有2.5mm  
2你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。  
3你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口——仅适用于对象Adapter。



如何实现：  
1类适配器

public class Adapter extends Adaptee implements Target{  
  @Override   
 public void getHeadset2() {  
  this.getHeadset3();   
 }   
 public static void main(String args[]){  
   Target target = new Adapter();   
   target.getHeadset2();   
 }  
  }  
    
interface Target{  
  void getHeadset2();  
  }  
  
  class Adaptee{  
  public void getHeadset3(){  
  System.out.println("我是3.5mm的耳机哦");  
  }  
  }

对象适配器

public class Adapter implements Target{   
 private Adaptee adaptee ;  
   public Adapter() {  
   super();   
 this.adaptee = new Adaptee();   
}   
 @Override   
 public void getHeadset2() {   
 adaptee.getHeadset3();  
  }  
  public static void main(String args[]){  
  Target target = new Adapter();  
 target.getHeadset2();  
  }  
 }  
  
  
  interface Target{  
  void getHeadset2();  
 }  
  
  
  class Adaptee{  
  public void getHeadset3(){  
  System.out.println("我是3.5mm的耳机哦");  
  }  
  }

对象的适配器模式的不同之处在于Adapter角色封装了Adaptee角色，而不像类的适配器模式所采取的继承方式。其原理基本上是相似的。  
  
  
  
 对于适配器模式，当adaptee是一个不成熟组件，而现在我们又要使用它的某些功能，将来很有可能把它替换掉，那适配器模式是最佳选择，我们发现如果我们将来要替换掉Adaptee，我们不用修改很多代码，只需要修改adapter中的getHeadset3方法就可以了，这就是适配器模式的使用场景