设计模式

## 单例模式

保证一个类只有一个实例，并且提供一个访问该实例的全局访问点。

### 饿汉式(线程安全，调用效率高，不能延迟加载！)

public class A{

private A(){};

private static A instance = new A();

public static A getInstace(){return instance;}

}

### 懒汉式(线程安全，调用效率不高，能延迟加载！)

public class A{

private A(){};

private static A instance = null;

public static synchronized A getInstace(){

if(instance==null)instance = new A();

return instance;

}

}

### 静态内部类式(线程安全，调用效率高，能延迟加载！)

public class A{

private A(){};

private static class B{

private static final A instance = new A();

}

public static A getInstace(){return B.instance;}

}

### 枚举式(线程安全，调用效率高，不能延迟加载！天然防止反射和反序列化漏洞)

public enum A{

INSTANCE;

}

## 工厂模式

实例化对象，用工厂方法代替new操作。

将选择实现类、创建对象统一管理和控制。从而将调用者跟我们的实现类解耦。

### 简单工厂模式(对于增加新的产品，需要修改已有代码)

public class CarFactory{

public static Car createCar(String type){

if(“奥迪”.equals(type))return new Audi(); //返回实现了Car接口的实体类

else if(“比亚迪”.equals(type))return new Byd();

else return null;

}

} *//小问题:违反开闭原则:如果有其他的类型Car就要修改代码*

public class CarFactory2{

public static Car createAudi(){return new Audi();} //返回实现了Car接口的实体类

public static Car createByd(){return new Byd();}

}

### 工厂方法模式(支持增加任意产品)

public interface CarFactory{

Car createCar(); //交给实现该接口的类去实现

}

public class AudiFactory implements CarFactory{

@Override

public Car createCar(){return new Audi();}

}

public class BydFactory implements CarFactory{

@Override

public Car createCar(){return new Byd();}

}

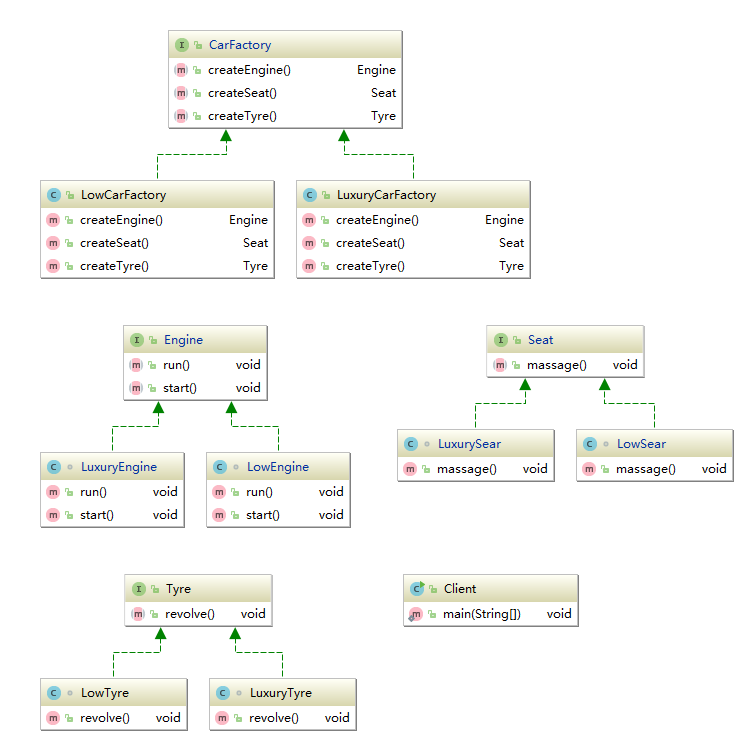
//有几个就扩展几个工厂

### 抽象工厂模式(对于增加新的产品无能为力；支持增加产品族)

**public interface** CarFactory {*//发动机、座椅、轮胎组成一个产品族* Engine createEngine();  
 Seat createSeat();  
 Tyre createTyre();  
}

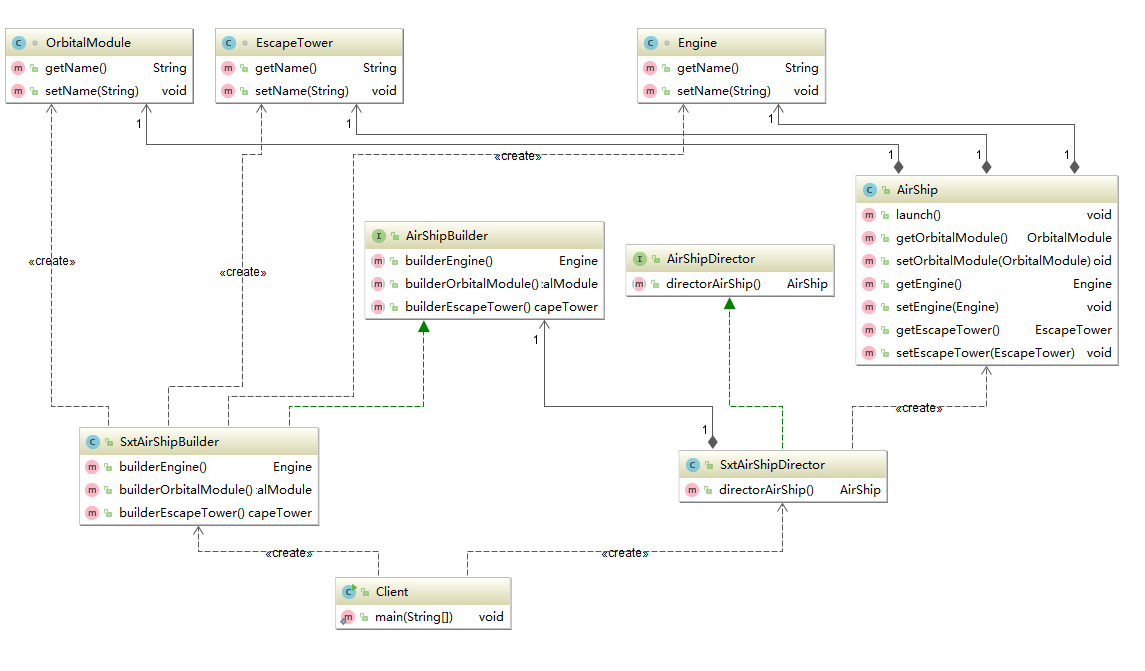
**public class** LowCarFactory **implements** CarFactory{  
 @Override  
 **public** Engine createEngine() {  
 **return new** LowEngine();  
 }  
 @Override  
 **public** Seat createSeat() {  
 **return new** LowSear();  
 }  
 @Override  
 **public** Tyre createTyre() {  
 **return new** LowTyre();  
 }  
}

**public class** LuxuryCarFactory **implements** CarFactory{  
 @Override  
 **public** Engine createEngine() {  
 **return new** LuxuryEngine();  
 }  
 @Override  
 **public** Seat createSeat() {  
 **return new** LuxurySear();  
 }  
 @Override  
 **public** Tyre createTyre() {  
 **return new** LuxuryTyre();  
 }  
}



## 建造者模式

* 分离了对象子组件的单独构造(由Builder来负责)和装配(由Director负责)。从而可以构造出复杂的对象。这个模式适用于:某个对象的构建过程复杂的情况下使用。
* 由于实现了构建和装配的解耦。不同的构建器，相同的装配，也可以做出不同的对象；相同的构建器，不同的装配顺序也可以做出不同的对象。也就是实现了构建算法、装配算法的解耦，实现了更好的复用。



## 原型模式

借助Cloneable接口实现

*浅克隆:在JAVA中，浅复制是指当对象的字段值被复制时,字段引用的对象不会被复制.例如,如果一个对象有一个指向字符串的字段,并且我们对该对象做了一个浅复制,那麽两个对象将引用同一个字符串*

*深克隆:克隆时把原型对象的属性对象也克隆一遍，使两个对象引用不同的对象*

**public class** Sheep2 **implements** Cloneable{  
 **private** String **sname**;  
 **private** Date **birthday**;  
   
 @Override  
 **protected** Object clone() **throws** CloneNotSupportedException {  
 Object obj = **super**.clone(); *//直接调用Object对象的clone()方法  
   
 //添加如下代码实现深克隆(deep clone)* Sheep2 s = (Sheep2)obj;  
 s.**birthday** = (Date) **this**.**birthday**.clone();*//把属性也进行克隆!!!* **return** obj;  
 }  
  
 **public** String getSname() {  
 **return sname**;  
 }  
  
 **public void** setSname(String sname) {  
 **this**.**sname** = sname;  
 }  
  
 **public** Date getBirthday() {  
 **return birthday**;  
 }  
  
 **public void** setBirthday(Date birthday) {  
 **this**.**birthday** = birthday;  
 }  
  
 **public** Sheep2(String sname, Date birthday) {  
 **super**();  
 **this**.**sname** = sname;  
 **this**.**birthday** = birthday;  
 }  
 **public** Sheep2() {  
   
 }  
}

也可以用序列化和反序列化实现深克隆(注意要实现Serializable接口)

**public class** Client3 {  
 **public static void** main(String[] args) **throws** Exception {  
 Date d = **new** Date(123123123222L);  
 Sheep s1 = **new** Sheep(**"少利"**,d);  
 System.***out***.println(s1); *//原型模式.Sheep@7852e922* System.***out***.println(s1.getSname()); *//少利* System.***out***.println(s1.getBirthday()); *//Mon Nov 26 08:52:03 CST 1973  
   
 //以s1为原型建新对象  
// Sheep s2 = (Sheep) s1.clone();  
 //使用序列化和反序列化实现深克隆* ByteArrayOutputStream bos = **new** ByteArrayOutputStream();  
 ObjectOutputStream oos = **new** ObjectOutputStream(bos);  
 oos.writeObject(s1);  
 **byte**[] bytes = bos.toByteArray(); *//序列化* ByteArrayInputStream bis = **new** ByteArrayInputStream(bytes);  
 ObjectInputStream ois = **new** ObjectInputStream(bis);  
 Sheep s2 = (Sheep) ois.readObject(); *//反序列化  
   
   
   
 //修改s1的属性* d.setTime(24124124124L);  
 System.***out***.println(s1.getBirthday()); *//Wed Oct 07 13:08:44 CST 1970* System.***out***.println(s2); *//原型模式.Sheep@4e25154f* System.***out***.println(s2.getSname()); *//少利* System.***out***.println(s2.getBirthday()); *//Mon Nov 26 08:52:03 CST 1973* s2.setSname(**"多利"**);  
 System.***out***.println(s2.getSname()); *//多利* }  
}

## 适配器模式(adapter)

将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得用本由于接口不兼容而不能一起工作的那些类可以在一起工作。

public interface Target{ //目标接口：客户所期待的接口

void handleReq()；

}

public class Adaptee{ //需要适配的类

public void request(){System.out.println(“可以完成客户请求所需要的功能”);}

}

public class Adapter implements Target{ //适配器

private Adaptee adaptee;

@Override

public void handleReq(){adaptee.request();}

public Adapter(Adaptee adaptee){this.adaptee=adaptee;}

}

调用

public class Client{

public static void main(String[] args){

Client c = new Client();*//客户端类(相当于例子中的笔记本，只有USB接口)*

Adaptee a = new Adaptee();*//被适配的类(相当于例子中的，PS/2键盘)*

Target t = new Adapter(a); *//适配器(相当于usb和ps/2的转接器)*

t.handleRReq();//已将其他类转换为可以调用的类

}

}

## 代理模式

**-核心作用**:通过代理，控制对对象的访问！

可以详细控制访问某个(某类)对象的方法，在调用这个方法前做前置处理，调用这个方 法后做后置处理.(即:AOP的微观实现！)

**-AOP(Aspect Oriented Programming面向切面编程)的核心实现机制！**

### 静态代理

**public interface** Star {  
 **void** confer(); */\*\*面谈\*/* **void** signContract(); */\*\*签合同\*/* **void** bookTicket(); */\*\*订票\*/* **void** sing(); */\*\*唱歌\*/* **void** collectMoney(); */\*\*收尾款\*/*}

真实角色(以明星为例，明星虽然会所有环节，但没必要每个环节都参与，只负责唱歌环节)

**public class** RealStar **implements** Star{  
 @Override  
 **public void** confer() {  
 System.***out***.println(**"RealStar.confer()"**);  
 System.***out***.println(**"真实角色面谈"**);  
 }  
 @Override  
 **public void** signContract() {  
 System.***out***.println(**"RealStar.signContract()"**);  
 System.***out***.println(**"真实角色签合同"**);  
 }  
 @Override  
 **public void** bookTicket() {  
 System.***out***.println(**"RealStar.bookTicket()"**);  
 System.***out***.println(**"真实角色订票"**);  
 }  
 @Override  
 **public void** sing() {  
 System.***out***.println(**"RealStar.sing()"**);  
 System.***out***.println(**"真实角色唱歌"**);  
 }  
 @Override  
 **public void** collectMoney() {  
 System.***out***.println(**"RealStar.collectMoney()"**);  
 System.***out***.println(**"真实角色收尾款"**);  
 }  
}

代理角色(相当于经纪人,负责处理明星的其他环节)

**public class** ProxyStar **implements** Star{  
 **private** Star **star**;  
 **public** ProxyStar(Star star) {  
 **super**();  
 **this**.**star** = star;  
 }  
 @Override  
 **public void** confer() {  
 System.***out***.println(**"RealStar.confer()"**);  
 System.***out***.println(**"代理角色面谈"**);  
 }  
 @Override  
 **public void** signContract() {  
 System.***out***.println(**"RealStar.signContract()"**);  
 System.***out***.println(**"代理角色签合同"**);  
 }  
 @Override  
 **public void** bookTicket() {  
 System.***out***.println(**"RealStar.bookTicket()"**);  
 System.***out***.println(**"代理角色订票"**);  
 }  
 @Override  
 **public void** sing() {  
 **star**.sing();*//调用真实star的sing方法* }  
 @Override  
 **public void** collectMoney() {  
 System.***out***.println(**"RealStar.collectMoney()"**);  
 System.***out***.println(**"代理角色收尾款"**);  
 }  
}

调用

**public class** Client {  
 **public static void** main(String[] args) {  
 Star real = **new** RealStar();  
 Star proxy = **new** ProxyStar(real);  
   
 proxy.confer();  
 proxy.signContract();  
 proxy.bookTicket();  
 proxy.sing();*//只有sing是真实角色执行* proxy.collectMoney();  
 }  
}

### 动态代理

**JDK自带的动态代理**

* java.lang.reflect.Proxy
  + 作用:动态生成代理类和对象
* java.lang.reflect.InvocationHandler（处理器接口）
* 可以通过invoke方法实现对真实角色的代理访问
* 每次通过Proxy生成代理类对象时都要指定对应的处理器对象

**public interface** Star {  
 **void** confer(); */\*\*面谈\*/* **void** signContract(); */\*\*签合同\*/* **void** bookTicket(); */\*\*订票\*/* **void** sing(); */\*\*唱歌\*/* **void** collectMoney(); */\*\*收尾款\*/*}

**真实角色**(以明星为例，明星虽然会所有环节，但没必要每个环节都参与，只负责唱歌环节)

**public class** RealStar **implements** Star{  
 @Override  
 **public void** confer() {  
 System.***out***.println(**"RealStar.confer()"**);  
 System.***out***.println(**"真实角色面谈"**);  
 }  
 @Override  
 **public void** signContract() {  
 System.***out***.println(**"RealStar.signContract()"**);  
 System.***out***.println(**"真实角色签合同"**);  
 }  
 @Override  
 **public void** bookTicket() {  
 System.***out***.println(**"RealStar.bookTicket()"**);  
 System.***out***.println(**"真实角色订票"**);  
 }  
 @Override  
 **public void** sing() {  
 System.***out***.println(**"RealStar.sing()"**);  
 System.***out***.println(**"真实角色唱歌"**);  
 }  
 @Override  
 **public void** collectMoney() {  
 System.***out***.println(**"RealStar.collectMoney()"**);  
 System.***out***.println(**"真实角色收尾款"**);  
 }  
}

**处理器**

**public class** StarHandler **implements** InvocationHandler{  
 Star **realStar**;  
   
 **public** StarHandler(Star realStar) {  
 **super**();  
 **this**.**realStar** = realStar;  
 }  
  
 @Override  
 **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  
 Object object = **null**;  
 System.***out***.println(**"真正的方法执行前:"**);  
 System.***out***.println(**"面谈，签合同，预付款，订机票"**);  
   
 **if**(method.getName().equals(**"sing"**)){  
 object = method.invoke(**realStar**,args);  
 }  
 System.***out***.println(**"真正的方法执行后:"**);  
 System.***out***.println(**"收尾款"**);  
 **return** object;  
 }  
}

**调用（**基本实现:动态生成代理类，不管调用代理类的哪个方法，都相当于调用handler的invoke方法**）**

**public class** Client {  
 **public static void** main(String[] args) {  
 Star realStar = **new** RealStar();  
 StarHandler handler = **new** StarHandler(realStar);  
 Star proxy = (Star) Proxy.*newProxyInstance*(ClassLoader.*getSystemClassLoader*(), **new** Class[] {Star.**class**}, handler);  
 proxy.sing();  
 */\*  
 结果：  
 真正的方法执行前:  
 面谈，签合同，预付款，订机票  
 RealStar.sing()  
 真实角色唱歌  
 真正的方法执行后:  
 收尾款  
 \*/* }  
}

## 桥接模式

* **核心要点:**
* 处理多层继承结构，处理多维度变化的场景，将各个维度设计成独立的继承结构，使各 个维度可以独立的扩展在抽象层建立关联。

以电脑为例

品牌:联想、戴尔、神舟...

机型:台式、笔记本...

**public class** Computer2 {  
 **protected** Brand **brand**;  
   
 **public** Computer2(Brand brand) {  
 **this**.**brand**=brand;  
 }  
   
 **public void** sale(){  
 **brand**.sale();  
 }  
}  
  
**class** Desktop2 **extends** Computer2{  
  
 **public** Desktop2(Brand brand) {  
 **super**(brand);  
 }  
   
 @Override  
 **public void** sale() {  
 **super**.sale();  
 System.***out***.println(**"销售台式机"**);  
 }  
}  
  
**class** Laptop2 **extends** Computer2{  
  
 **public** Laptop2(Brand brand) {  
 **super**(brand);  
 }  
   
 @Override  
 **public void** sale() {  
 **super**.sale();  
 System.***out***.println(**"销售笔记本"**);  
 }  
}

**品牌(**有新的品牌只需要自己写这个品牌类即可，不用修改其他代码！**)**

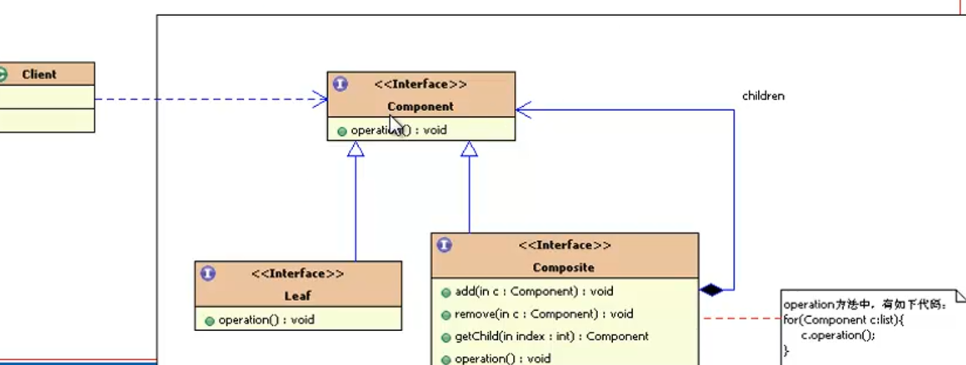
**public interface** Brand {  
 **void** sale();  
}  
**class** Lenovo **implements** Brand{  
 @Override  
 **public void** sale() {  
 System.***out***.println(**"销售联想电脑"**);  
 }  
}  
**class** Dell **implements** Brand{  
 @Override  
 **public void** sale() {  
 System.***out***.println(**"销售Dell电脑"**);  
 }  
}  
**class** Shenzhou **implements** Brand{  
 @Override  
 **public void** sale() {  
 System.***out***.println(**"销售神舟电脑"**);  
 }  
}

**调用**

**public class** Client {  
 **public static void** main(String[] args) {  
 *//销售联想的笔记本电脑* Computer2 c = **new** Laptop2(**new** Lenovo());  
 c.sale();  
   
 c = **new** Desktop2(**new** Shenzhou());  
 c.sale();  
 }  
}

## 组合模式(composite)

* **使用组合模式的场景：**
* **把部分和整体的关系用树形结构来表示，从而使客户端可以使用统一的方式处理部分对象和整体对象**
* **组合模式核心**
  + **抽象构件(Component)角色:定义了叶子和容器构件的共同点**
  + **叶子(Leaf)构件角色:无子节点**
  + **容器(Composite)构件角色:有容器特征，可以包含子节点**



以杀毒为例(容器：文件夹Folder 叶子：XxFile文件 抽象构件:AbstractFile-有X功能)

**public interface** AbstractFile {  
 **void** killVirus(); *//杀毒*}  
  
**class** ImageFile **implements** AbstractFile{ *//相当于Leaf（叶子组件）* **private** String **name**;  
 **public** ImageFile(String name){  
 **super**();  
 **this**.**name**=name;  
 }  
  
 @Override  
 **public void** killVirus() { *//相当于Leaf（叶子组件）* System.***out***.println(**"----图像文件:"**+**name**+**",进行查杀！"**);  
 }  
}  
**class** TextFile **implements** AbstractFile{  
 **private** String **name**;  
 **public** TextFile(String name){  
 **super**();  
 **this**.**name**=name;  
 }  
  
 @Override  
 **public void** killVirus() { *//相当于Leaf（叶子组件）* System.***out***.println(**"----文本文件:"**+**name**+**",进行查杀！"**);  
 }  
}  
**class** VideoFile **implements** AbstractFile{  
 **private** String **name**;  
 **public** VideoFile(String name){  
 **super**();  
 **this**.**name**=name;  
 }  
  
 @Override  
 **public void** killVirus() {  
 System.***out***.println(**"----视频文件:"**+**name**+**",进行查杀！"**);  
 }  
}  
  
**class** Folder **implements** AbstractFile{ *//相当于容器组件* **private** String **name**;  
 *//定义容器，用来存放本容器构建下的子节点* **private** List<AbstractFile> **list** = **new** ArrayList<>();  
  
 **public** Folder(String name) {  
 **this**.**name** = name;  
 }  
 **public void** add(AbstractFile file){  
 **list**.add(file);  
 }  
 **public void** remove(AbstractFile file){  
 **list**.remove(file);  
 }  
 **public** AbstractFile getChild(**int** index){  
 **return list**.get(index);  
 }  
  
 @Override  
 **public void** killVirus() {  
 System.***out***.println(**"----文件夹:"**+**name**+**"，进行查杀!"**);  
 **list**.forEach(AbstractFile::killVirus);*//递归* }  
}

调用

**public class** Client {  
 **public static void** main(String[] args) {  
 AbstractFile f2,f3,f4,f5,f6;  
 Folder f1 = **new** Folder(**"我的收藏"**);  
 f2 = **new** ImageFile(**"头像.jpg"**);  
 f3 = **new** TextFile(**"Hello.txt"**);  
 f1.add(f2);  
 f1.add(f3);  
*// f2.killVirus();  
// f1.killVirus();* Folder f11 = **new** Folder(**"电影"**);  
 f4 = **new** VideoFile(**"笑傲江湖.avi"**);  
 f5 = **new** VideoFile(**"神雕侠侣.avi"**);  
 f11.add(f4);  
 f11.add(f5);  
 f1.add(f11);  
 f1.killVirus();  
 }  
}

结果：

----文件夹:我的收藏，进行查杀!

----图像文件:头像.jpg,进行查杀！

----文本文件:Hello.txt,进行查杀！

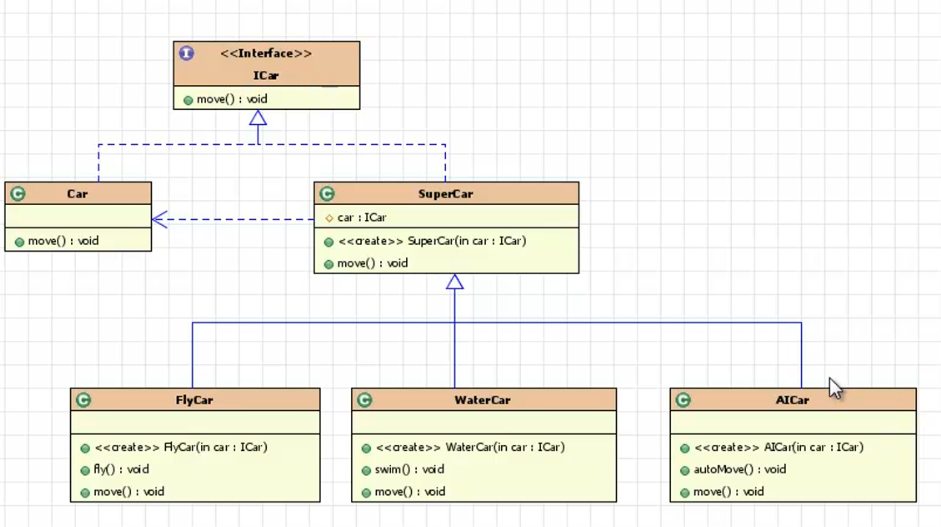
----文件夹:电影，进行查杀!

----视频文件:笑傲江湖.avi,进行查杀！

----视频文件:神雕侠侣.avi,进行查杀！

## 装饰模式

* 职责：
  + 动态的为一个对象增加新的功能。
  + 装饰模式是一种用于代替继承的技术，无须通过继承增加子类就能扩展对象的新功能。使用对象的关联关系，更加灵活，同时避免类型体系的快速膨胀。
* 实现细节：
* Component抽象构件角色：
* 真实对象和装饰对象有相同的接口。这样，客户端对象就能够以与真实对象相同的方式同装饰对象交互。
* ConcreteComponent具体构件角色(真实对象):
* 比如：io流中的FileInputStream、FileOutputStream
* Decorator装饰角色：
* 持有一个抽象构件的引用。装饰对象接受所有客户端的请求，并把这些请求转发给真实的对象。这样，就能在真实对象调用前后增加新的功能。
* ConcreteDecorator具体装饰角色:
* 负责给构件对象增加新 的责任。



**public interface** ICar {*//抽象构件* **void** move();  
}  
**class** Car **implements** ICar{ *//真实对象:具体构件* @Override  
 **public void** move() {  
 System.***out***.println(**"陆地上跑！"**);  
 }  
}  
  
**class** SuperCar **implements** ICar{*//Decorator装饰角色* **protected** ICar **car**;  
  
 **public** SuperCar(ICar car) {  
 **this**.**car** = car;  
 }  
 @Override  
 **public void** move() {  
 **car**.move();  
 }  
}  
**class** FlyCar **extends** SuperCar{ *//ConcreteDecorator具体装饰角色* **public** FlyCar(ICar car) {  
 **super**(car);  
 }  
 **public void** fly(){  
 System.***out***.println(**"天上飞!"**);  
 }  
 @Override  
 **public void** move(){  
 **super**.move();  
 fly();  
 }  
}  
**class** WaterCar **extends** SuperCar{ *//ConcreteDecorator具体装饰角色* **public** WaterCar(ICar car) {  
 **super**(car);  
 }  
 **public void** swim(){  
 System.***out***.println(**"水上游!"**);  
 }  
 @Override  
 **public void** move(){  
 **super**.move();  
 swim();  
 }  
}  
**class** AICar **extends** SuperCar{ *//ConcreteDecorator具体装饰角色* **public** AICar(ICar car) {  
 **super**(car);  
 }  
 **public void** autoMove(){  
 System.***out***.println(**"自动跑!"**);  
 }  
 @Override  
 **public void** move(){  
 **super**.move();  
 autoMove();  
 }  
}

调用：

**public class** Client {  
 **public static void** main(String[] args) {  
 Car car = **new** Car();  
 car.move();  
 System.***out***.println(**"增加新的功能，飞行----------"**);  
 FlyCar flyCar = **new** FlyCar(car);  
 flyCar.move();  
 System.***out***.println(**"再增加新的功能,水里游---------"**);  
 WaterCar waterCar = **new** WaterCar(flyCar);  
 waterCar.move();  
 }  
}

结果:

陆地上跑！

增加新的功能，飞行----------

陆地上跑！

天上飞!

再增加新的功能,水里游---------

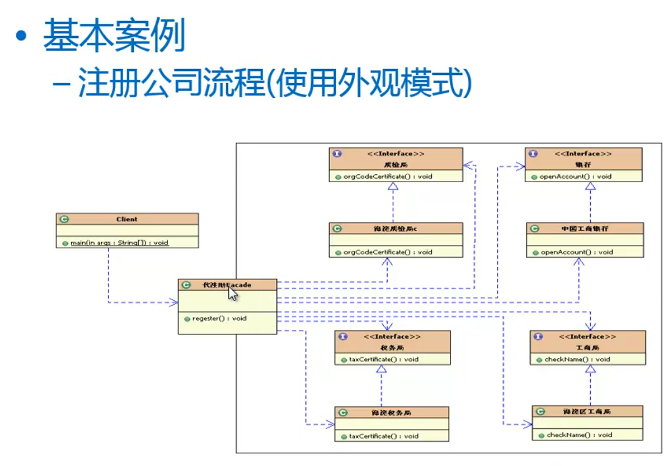
陆地上跑！

天上飞!

水上游!

## 外观模式

核心：为子系统提供统一的入口。封装子系统的复杂性，便于客户端调用。



## 享元模式

* 享元模式以共享的方式高效地支持大量细粒对象的重用。
* 享元模式能做到共享的关键是区分了内部状态和外部状态。
  + 内部状态：可以共享，不会随环境变化而改变
  + 外部状态：不可以共享，会随环境变化而改变

比如围棋：棋子的颜色、大小、形状是可以共享的（内部），而位置是不可以共享的（外部）

* 享元模式的实现：
  + FlyweightFactory享元工厂类
* 创建并管理享元对象，享元池一般设计成键值对
  + Flyweight抽象享元类
* 通常是一个接口或抽象类，声明公共方法，这些方法可以向外界提供对象的内部状态，设置外部状态。
  + ConcreteFlyweight具体享元类
* 为内部状态提供成员变量进行存储
  + UnsharedConcreteFlyWeight非共享享元类
* 不能被共享的子类可以设计为非共享享元类

**抽象享元--围棋棋子**

**public interface** ChessFlyWeight { *//享元类 围棋棋子* **void** setColor(String color);  
 String getColor();  
 **void** display(Coordinate c);  
}

**具体享元类--**可以向外界提供内部状态，也可以设置外部状态  
**class** ConcreteChess **implements** ChessFlyWeight{*//具体的享元类* **private** String **color**;  
  
 **public** ConcreteChess(String color) {  
 **this**.**color** = color;  
 }  
  
 @Override  
 **public void** setColor(String color) {  
 **this**.**color**=color;  
 }  
  
 @Override  
 **public** String getColor() {  
 **return color**;  
 }  
  
 @Override  
 **public void** display(Coordinate c) {  
 System.***out***.println(**"棋子颜色:"**+**color**+**"\t棋子位置:("**+c.getX()+**","**+c.getY()+**")"**);  
 }  
  
}

**UnsharedConcreteFlyWeight非共享享元类--**棋子的坐标

**public class** Coordinate {*//棋子的坐标类 外部状态UnSharedConcreteFlyWeight* **private int x**,**y**;  
  
 **public** Coordinate(**int** x, **int** y) {  
 **this**.**x** = x;  
 **this**.**y** = y;  
 }  
  
 **public int** getX() {  
 **return x**;  
 }  
  
 **public void** setX(**int** x) {  
 **this**.**x** = x;  
 }  
  
 **public int** getY() {  
 **return y**;  
 }  
  
 **public void** setY(**int** y) {  
 **this**.**y** = y;  
 }  
}

**FlyweightFactory享元工厂类**

**public class** ChessFlyWeightFactory {*//享元工厂类* **private static** Map<String,ChessFlyWeight> *map* = **new** HashMap<>(); *//享元池* **public static** ChessFlyWeight getChess(String color){  
 **if**(*map*.get(color)!=**null**){  
 **return** *map*.get(color);  
 }**else**{  
 ChessFlyWeight cfw = **new** ConcreteChess(color);  
 *map*.put(color,cfw);  
 **return** cfw;  
 }  
 }  
}

**调用**

**public class** Client {  
 **public static void** main(String[] args) {  
 ChessFlyWeight chess1 = ChessFlyWeightFactory.*getChess*(**"黑色"**);  
 ChessFlyWeight chess2 = ChessFlyWeightFactory.*getChess*(**"黑色"**);  
 System.***out***.println(chess1==chess2); *//true* System.***out***.println(**"===========增加外部状态的处理============="**);  
 chess1.display(**new** Coordinate(10,10));  
 chess1.display(**new** Coordinate(20,20));  
 }  
}

**结果：**

true

===========增加外部状态的处理=============

棋子颜色:黑色 棋子位置:(10,10)

棋子颜色:黑色 棋子位置:(20,20)

## 责任链模式

将能够处理同一类请求的对象连成一条链，所提交的请求沿着链传递，链上的对象逐个判断是否有能力处理该请求，如果能则处理，否则传递给链上的下一个对象。

***以请假审批为案例***

**首先封装请假的基本信息**

**public class** LeaveRequest {*//封装请假的基本信息* **private** String **empName**;  
 **private int leaveDay**;  
 **private** String **reason**;  
  
 **public** LeaveRequest(String empName, **int** leaveDay, String reason) {  
 **this**.**empName** = empName;  
 **this**.**leaveDay** = leaveDay;  
 **this**.**reason** = reason;  
 }

get…set…

**然后定义抽象类Leader(负责审批)**

**public abstract class** Leader {  
 **protected** String **name**;  
 **protected** Leader **nextLeader**;*//责任链上的后继对象* **public void** setNextLeader(Leader nextLeader) { *//设置责任链上的后继对象* **this**.**nextLeader** = nextLeader;  
 }  
 **public** Leader(String name) {  
 **this**.**name** = name;  
 }  
 */\*\*处理请求的核心业务方法\*/* **public abstract void** handleRequest(LeaveRequest request);  
}

**如果请假天数少于3天，主任(Leader)审批**

**public class** Director **extends** Leader{ *//主任审批* **public** Director(String name) {  
 **super**(name);  
 }  
 @Override  
 **public void** handleRequest(LeaveRequest request) {  
 **if**(request.getLeaveDay()<3){  
 System.***out***.println(**"员工："**+request.getEmpName()+**"请假"**+request.getLeaveDay()+**"天，理由："**+request.getReason());  
 System.***out***.println(**"主任："**+**this**.**name**+**"审批通过"**);  
 }**else**{  
 **if**(**this**.**nextLeader**!=**null**){  
 **this**.**nextLeader**.handleRequest(request);*//交给后继对象处理* }  
 }  
 }  
}

**如果请假天数少于10天，经理(Leader)审批**

**public class** Manager **extends** Leader{ *//经理审批* **public** Manager(String name) {  
 **super**(name);  
 }  
 @Override  
 **public void** handleRequest(LeaveRequest request) {  
 **if**(request.getLeaveDay()<10){  
 System.***out***.println(**"员工："**+request.getEmpName()+**"请假"**+request.getLeaveDay()+**"天，理由："**+request.getReason());  
 System.***out***.println(**"经理："**+**this**.**name**+**"审批通过"**);  
 }**else**{  
 **if**(**this**.**nextLeader**!=**null**){  
 **this**.**nextLeader**.handleRequest(request);*//交给后继对象处理* }  
 }  
 }  
}

**如果请假天数少于30天，总经理(Leader)审批**

**public class** GeneralManager **extends** Leader{ *//总经理审批* **public** GeneralManager(String name) {  
 **super**(name);  
 }  
 @Override  
 **public void** handleRequest(LeaveRequest request) {  
 **if**(request.getLeaveDay()<30){  
 System.***out***.println(**"员工："**+request.getEmpName()+**"请假"**+request.getLeaveDay()+**"天，理由："**+request.getReason());  
 System.***out***.println(**"总经理："**+**this**.**name**+**"审批通过"**);  
 }**else**{  
 System.***out***.println(**"请假不通过!"**);  
 }  
 }  
}

**调用：**

**public class** Client {  
 **public static void** main(String[] args) {  
 Leader a = **new** Director(**"张三"**);  
 Leader b = **new** Manager(**"李四"**);  
 Leader c = **new** GeneralManager(**"王五"**);  
 *//组织责任链对象的关系* a.setNextLeader(b);  
 b.setNextLeader(c);  
 *//开启请假请求* LeaveRequest req1 = **new** LeaveRequest(**"TOM"**,10,**"世界那么大。。。"**);  
 a.handleRequest(req1);  
 }  
}

**结果：**

员工：TOM请假10天，理由：世界那么大。。。

总经理：王五审批通过

## 迭代器模式

* 场景：
  + 提供一种可以遍历聚合对象的方式。又称为:游标cursor模式
  + 聚合对象：存储数据
  + 迭代器：遍历数据

**自定义迭代器接口**

**public interface** MyIterator {  
 **void** first(); *//将游标指向第一个元素* **void** next(); *//将游标指向下一个元素* **boolean** hasNext();  
 **boolean** isFirst();  
 **boolean** isLast();  
 Object getCurrentObj();*//获取当前游标指定的对象*}

**自定义聚合类（内部类实现迭代器）**

**public class** ConcreteMyAggregate {  
 **private** List<Object> **list** = **new** ArrayList<>();  
  
 **public void** addObject(Object obj){  
 **this**.**list**.add(obj);  
 }  
 **public void** removeObject(Object obj){  
 **this**.**list**.remove(obj);  
 }  
 **public** List<Object> getList() {  
 **return list**;  
 }  
 **public void** setList(List<Object> list) {  
 **this**.**list** = list;  
 }  
 **public** MyIterator createIterator(){*//获得迭代器* **return new** ConcreteIterator();  
 }  
 *//使用内部类定义迭代器，可以直接使用外部类的属性* **private class** ConcreteIterator **implements** MyIterator{  
 **private int cursor**;*//定义游标，记录遍历时的位置* @Override  
 **public void** first() {  
 **cursor**=0;  
 }  
 @Override  
 **public void** next() {  
 **if**(hasNext())  
 **cursor**++;  
 }  
  
 @Override  
 **public boolean** hasNext() {  
 **return cursor**!=**list**.size();  
 }  
  
 @Override  
 **public boolean** isFirst() {  
 **return cursor**==0;  
 }  
  
 @Override  
 **public boolean** isLast() {  
 **return cursor**==**list**.size()-1;  
 }  
  
 @Override  
 **public** Object getCurrentObj() {  
 **return list**.get(**cursor**);  
 }  
 }  
}

**调用：**

**public class** Client {  
 **public static void** main(String[] args) {  
 ConcreteMyAggregate cma = **new** ConcreteMyAggregate();  
 cma.addObject(**"aa"**);  
 cma.addObject(**"bb"**);  
 cma.addObject(**"cc"**);  
 cma.addObject(**"dd"**);  
 MyIterator iterator = cma.createIterator();  
 **while**(iterator.hasNext()){  
 System.***out***.println(iterator.getCurrentObj());  
 iterator.next();  
 }  
 }  
}

## 中介者模式

将多对多的关系转换为一对多的关系

* **本质：**
* 解耦多个同事对象之间的交互关系。每个对象都持有中介对象的引用，只跟中介者对象打交道。我们通过中介者对象统一管理这些交互关系。

**中介者接口：**

**public interface** Mediator {  
 **void** register(String dname,Department department);  
 **void** command(String dname);  
}

**中介者类**（总经理）

**public class** President **implements** Mediator{  
 **private** Map<String,Department> **map** = **new** HashMap<>();  
 @Override  
 **public void** register(String dname, Department department) {  
 **map**.put(dname,department);  
 }  
 @Override  
 **public void** command(String dname) {  
 **map**.get(dname).selfAction();  
 }  
}

**同事类的接口**

**public interface** Department {*//同事类的接口* **void** selfAction();*//做本部门的事情* **void** outAction();*//向总经理(中介者)发出申请*}

**具体的同事类**(财务部、市场部、研发部 )

**public class** Development **implements** Department {  
 **private** Mediator **m**;*//持有中介者(总经理)的引用* **public** Development(Mediator m) {  
 **this**.**m** = m;  
 m.register(**"development"**,**this**);  
 }  
 @Override  
 **public void** selfAction() {  
 System.***out***.println(**"研发部：汇报工作！申请资金！"**);  
 }  
 @Override  
 **public void** outAction() {  
 System.***out***.println(**"研发部：专心科研，开发项目"**);  
 }  
}

**public class** Finacial **implements** Department {  
 **private** Mediator **m**;*//持有中介者(总经理)的引用* **public** Finacial(Mediator m) {  
 **this**.**m** = m;  
 m.register(**"finacial"**,**this**);  
 }  
 @Override  
 **public void** selfAction() {  
 System.***out***.println(**"财务部：汇报工作！给钱！"**);  
 }  
 @Override  
 **public void** outAction() {  
 System.***out***.println(**"财务部：数钱！"**);  
 }  
}

**public class** Market **implements** Department {  
 **private** Mediator **m**;*//持有中介者(总经理)的引用* **public** Market(Mediator m) {  
 **this**.**m** = m;  
 m.register(**"market"**,**this**);  
 }  
 @Override  
 **public void** selfAction() {  
 System.***out***.println(**"市场部：汇报工作！项目 承接的进度！申请资金！"**);  
 **m**.command(**"finacial"**);  
 }  
 @Override  
 **public void** outAction() {  
 System.***out***.println(**"市场部：跑去接项目！"**);  
 }  
}

**调用：**

**public class** Client {  
 **public static void** main(String[] args) {  
 Mediator m = **new** President();  
 Market market = **new** Market(m);  
 Development development = **new** Development(m);  
 Finacial finacial = **new** Finacial(m);  
 market.selfAction();  
 market.outAction();  
 }  
}

**结果:**

市场部：汇报工作！项目 承接的进度！申请资金！

财务部：汇报工作！给钱！

市场部：跑去接项目！

## 命令模式

将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。也称之为：动作Action模式、事务transaction模式。

**命令类的接口**

**public interface** Command {  
 */\*\*  
 \* 这个方法是一个返回结果为空的方法。  
 \* 实际项目中，科研根据需求设计多个不同的方法  
 \*/* **void** execute();  
}

**命令的具体类**  
**class** ConcreteCommand **implements** Command{  
 **private** Receiver **receiver**;*//命令的真正执行者* **public** ConcreteCommand(Receiver receiver) {  
 **this**.**receiver** = receiver;  
 }  
 @Override  
 **public void** execute() {  
 *//命令真正执行前或后，执行相关的命令* **receiver**.action();  
 }  
}

**调用者/发起者**

**public class** Invoke {*//调用者/发起者* **private** Command **command**;*//也可以通过容器List<Command>容纳很多命令对象，进行批处理。数据库底层的事务管理就是类似的结构！* **public** Invoke(Command command) {  
 **this**.**command** = command;  
 }  
 **public void** call(){*//业务方法，用于调用命令类的方法* **command**.execute();  
 }  
}

**真正命令的执行者**

**public class** Receiver {*//真正命令的执行者* **public void** action(){  
 System.***out***.println(**"Receiver.action()"**);  
 }  
}

**调用**

**public class** Client {  
 **public static void** main(String[] args) {  
 Command command = **new** ConcreteCommand(**new** Receiver());  
 Invoke i = **new** Invoke(command);  
 i.call();  
 *//其实就是new Receiver().action(); 但这样在执行前后执行相关的命令，方便扩展！！！* }  
}

## 策略模式

策略模式对应于解决某一个问题的一个算法族，允许用户从该算法族中任选一个算法解决某一问题，同时可以方便的更换算法或者增加新的算法。并且由客户端决定调用哪个算法。

**本质：**分离算法，选择实现

**场景：**

* + 某个市场人员接到单后的报价策略

普通客户小批量报价---原价

普通客户大批量报价---九折

老客户小批量报价-----八五折

老客户大批量报价-----八折

* + 具体选用哪个报价策略，需要根据实际情况来确定。采用策略模式

**策略接口**

**public interface** Strategy {*//所有实现该接口的类构成算法族* **public double** getPrice(**double** standardPrice);  
}

**普通客户小批量报价策略—实现策略接口**

**public class** NewCustomerFewStrategy **implements** Strategy {  
 @Override  
 **public double** getPrice(**double** standardPrice) {  
 System.***out***.println(**"不打折，原价！"**);  
 **return** standardPrice;  
 }  
}

**普通客户大批量报价策略—实现策略接口**

**public class** NewCustomerManyStrategy **implements** Strategy {  
 @Override  
 **public double** getPrice(**double** standardPrice) {  
 System.***out***.println(**"打九折"**);  
 **return** standardPrice\*0.9;  
 }  
}

**老客户小批量报价策略—实现策略接口**

**public class** OldCustomerFewStrategy **implements** Strategy {  
 @Override  
 **public double** getPrice(**double** standardPrice) {  
 System.***out***.println(**"打八五折"**);  
 **return** standardPrice\*0.85;  
 }  
}

**老客户大批量报价策略—实现策略接口**

**public class** OldCustomerManyStrategy **implements** Strategy {  
 @Override  
 **public double** getPrice(**double** standardPrice) {  
 System.***out***.println(**"打八折"**);  
 **return** standardPrice\*0.8;  
 }  
}

**上下文--**负责和具体的策略类交互

**public class** Context {*//负责和具体的策略类交互，这样的话，具体的算法和直接的客户端调用分离了，使得算法可以独立于客户端独立的变化* **private** Strategy **strategy**;*//当前采用的算法对象* **public** Context(Strategy strategy) {*//通过构造器来注入* **this**.**strategy** = strategy;  
 }  
 **public void** printPrice(**double** s){  
 System.***out***.println(**"原价："**+s+**",报价："**+**strategy**.getPrice(s));  
 }  
}

**调用**

**public class** Client {  
 **public static void** main(String[] args) {  
 Strategy s1 = **new** OldCustomerFewStrategy();  
 Context ctx = **new** Context(s1);  
 ctx.printPrice(998);  
 }  
}

**结果：**

打八五折

原价：998.0,报价：848.3

## 模板方法模式

* **介绍：**

模板方法模式是编程中经常用得到的模式。它定义了一个操作中的算法骨架，将某些步骤延迟到子类中实现。这样，新的子类可以在不改变一个算法结构的前提下重新定义该算法的某些特定步骤。

* **核心：**

处理步骤父类中定义好，具体实现延迟到子类中定义

**抽象父类**—银行业务

**public abstract class** BankTemplateMethod {  
 **public void** takeNumber(){System.***out***.println(**"取号排队"**);}*//具体方法* **public abstract void** transact();*//办理具体的业务 钩子方法* **public void** evaluate(){System.***out***.println(**"反馈评分"**);}  
 **public final void** process(){*//模板方法* **this**.takeNumber();  
 **this**.transact();  
 **this**.evaluate();  
 }  
}

**调用**—子类实现

**public class** Client {  
 **public static void** main(String[] args) {  
 BankTemplateMethod btm = **new** DrawMoney();  
 btm.process();  
 *//采用匿名内部类* **new** BankTemplateMethod(){@Override  
 **public void** transact() {  
 System.***out***.println(**"我要存款"**);  
 }}.process();  
 **new** BankTemplateMethod(){@Override  
 **public void** transact() {  
 System.***out***.println(**"我要理财！！！！"**);  
 }}.process();  
 }  
}  
**class** DrawMoney **extends** BankTemplateMethod{  
 @Override  
 **public void** transact() {  
 System.***out***.println(**"我要取款!!!"**);  
 }  
}

**结果：**

取号排队

我要取款!!!

反馈评分

取号排队

我要存款

反馈评分

取号排队

我要理财！！！！

反馈评分

## 状态模式

* **核心：**

用来解决系统中复杂对象的状态转换以及不同状态下行为的封装问题

* **结构：**
  + - Context环境类
* 环境类中维护一个State对象，他是定义了当前的状态
  + - State抽象状态类
    - ConcreteState具体状态类
* 每一个类封装了一个状态对应的行为

**State抽象状态类**

**public interface** State {  
 **void** handle();  
}

**ConcreteState具体状态类**

* 空闲状态

**public class** FreeState **implements** State{*//空闲状态* @Override  
 **public void** handle() {  
 System.***out***.println(**"房间空闲，没人住！"**);  
 }  
}

* 预定状态

**public class** BookedState **implements** State{*//已预定状态* @Override  
 **public void** handle() {  
 System.***out***.println(**"房间已预定！别人不能订"**);  
 }  
}

* 入住状态

**public class** CheckedInState **implements** State{*//已入住状态* @Override  
 **public void** handle() {  
 System.***out***.println(**"房间已被人入住！请勿打扰！"**);  
 }  
}

**Context环境类**

**public class** Context {*//房间状态！ 维持不同状态间的切换  
 //如果是银行系统，这个Context类就是账号。根据金额不同，切换不同的状态！* **private** State **state**;  
 **public void** setState(State s){  
 System.***out***.println(**"修改状态！"**);  
 **state** = s;  
 **state**.handle();  
 }  
}

**调用：**

**public class** Client {  
 **public static void** main(String[] args) {  
 Context context = **new** Context();  
 context.setState(**new** FreeState());  
 context.setState(**new** BookedState());  
 }  
}

**结果：**

修改状态！

房间空闲，没人住！

修改状态！

房间已预定！别人不能订

## 观察者模式

观察者模式主要用于1：N的通知。当一个对象(目标对象Subject或Observable)的状态变化时，他需要及时通知一系列对象(观察者对象，Observer),令他们做出响应

**观察者接口Obsever**

**public interface** Observer {  
 **void** update(Subject subject);  
}

**主题类**

**public class** Subject {  
 **protected** List<Observer> **list** = **new** ArrayList<>();  
  
 **public void** registerObserver(Observer observer){  
 **list**.add(observer);  
 }  
 **public void** removeObserver(Observer observer){  
 **list**.remove(observer);  
 }  
  
 **public void** notifyAllObserver(){  
 **list**.forEach(obs->obs.update(**this**));  
 }

}

**具体的主题类**

**public class** ConcreteSubject **extends** Subject{  
 **private int state**;  
  
 **public int** getState() {  
 **return state**;  
 }  
  
 **public void** setState(**int** state) {  
 **this**.**state** = state;  
 **this**.notifyAllObserver();*//目标对象值发生了变化，通知所有的观察者* }  
}

**具体的观察者**

**public class** ObserverA **implements** Observer {  
 **private int myState**;*//myState需要跟目标对象的state值保存一致* @Override  
 **public void** update(Subject subject) {  
 **myState** = ((ConcreteSubject)subject).getState();  
 }  
  
 **public int** getMyState() {  
 **return myState**;  
 }  
  
 **public void** setMyState(**int** myState) {  
 **this**.**myState** = myState;  
 }  
}

**调用**

**public class** Client {  
 **public static void** main(String[] args) {  
 *//目标对象* ConcreteSubject subject = **new** ConcreteSubject();  
  
 *//创建多个观察者* ObserverA obs1 = **new** ObserverA();  
 ObserverA obs2 = **new** ObserverA();  
 ObserverA obs3 = **new** ObserverA();  
  
 *//加入subject对象的观察者队列* subject.registerObserver(obs1);  
 subject.registerObserver(obs2);  
 subject.registerObserver(obs3);  
  
 System.***out***.println(obs1.getMyState());  
 System.***out***.println(obs2.getMyState());  
 System.***out***.println(obs3.getMyState());  
 *//改变subject的状态* subject.setState(300);  
 System.***out***.println(**"##############"**);  
 System.***out***.println(obs1.getMyState());  
 System.***out***.println(obs2.getMyState());  
 System.***out***.println(obs3.getMyState());  
 subject.setState(3240);  
 System.***out***.println(**"##############"**);  
 System.***out***.println(obs1.getMyState());  
 System.***out***.println(obs2.getMyState());  
 System.***out***.println(obs3.getMyState());  
 }  
}

结果：

0

0

0

##############

300

300

300

##############

3240

3240

3240

### Observable类和Obsever接口

*//目标对象***public class** ConcreteSubject **extends** Observable {  
  
 **private int state**;  
  
 **public void** set(**int** s){  
 **state** = s; *//目标对象的状态发生了变化* setChanged();*//表示目标对象已经做了更改* notifyObservers(**state**);*//通知所有的观察者* }  
  
 **public int** getState() {  
 **return state**;  
 }  
  
 **public void** setState(**int** state) {  
 **this**.**state** = state;  
 }  
}

**观察者**

**public class** ObserverA **implements** Observer {  
 **private int myState**;  
 @Override  
 **public void** update(Observable o, Object arg) {  
 **myState** = ((ConcreteSubject)o).getState();  
 }  
  
 **public int** getMyState() {  
 **return myState**;  
 }  
  
 **public void** setMyState(**int** myState) {  
 **this**.**myState** = myState;  
 }  
}

**调用**

**public class** Client {  
 **public static void** main(String[] args) {  
 *//创建目标对象* ConcreteSubject subject = **new** ConcreteSubject();  
  
 *//创建观察者* ObserverA obs1 = **new** ObserverA();  
 ObserverA obs2 = **new** ObserverA();  
 ObserverA obs3 = **new** ObserverA();  
  
 *//将观察者对象添加到目标对象subject的观察者容器种* subject.addObserver(obs1);  
 subject.addObserver(obs2);  
 subject.addObserver(obs3);  
  
 System.***out***.println(obs1.getMyState());  
 System.***out***.println(obs2.getMyState());  
 System.***out***.println(obs3.getMyState());  
 *//改变subject的状态* subject.set(3000);  
 System.***out***.println(**"===================="**);  
 *//观察者的状态发生了变化* System.***out***.println(obs1.getMyState());  
 System.***out***.println(obs2.getMyState());  
 System.***out***.println(obs3.getMyState());  
 subject.set(6666);  
 System.***out***.println(**"===================="**);  
 *//观察者的状态发生了变化* System.***out***.println(obs1.getMyState());  
 System.***out***.println(obs2.getMyState());  
 System.***out***.println(obs3.getMyState());  
  
 }  
}

**结果：**

0

0

0

====================

3000

3000

3000

====================

6666

6666

6666

## 备忘录模式

就是保存某个对象内部状态的拷贝，这样以后就可以将该对象恢复到原先的状态。

**结构：**

* + **源发射器类Originator**
  + **备忘录类Memento**
  + **负责人类CareTaker**

**源发射器类Originator**

**public class** Emp {  
 **private** String **ename**;  
 **private int age**;  
 **private double salary**;  
  
 *//进行备忘操作,并返回备忘录对象* **public** EmpMemento memento(){  
 **return new** EmpMemento(**this**);  
 }  
 *//进行数据恢复，恢复成指定备忘录对象的值* **public void** recovery(EmpMemento mmt){  
 **this**.**ename** = mmt.getEname();  
 **this**.**age** = mmt.getAge();  
 **this**.**salary** = mmt.getSalary();  
 }

。。。。省略get、set、构造器。。。。

**备忘录类Memento**

**public class** EmpMemento {  
 **private** String **ename**;  
 **private int age**;  
 **private double salary**;  
  
 **public** EmpMemento(Emp e) {  
 **this**.**ename** = e.getEname();  
 **this**.**age** = e.getAge();  
 **this**.**salary** = e.getSalary();  
 }

。。。。省略get、set、构造器。。。。

**负责人类CareTaker**

**public class** CareTaker {  
 **private** EmpMemento **memento**;  
 *//private Stack<EmpMemento> stack; //备忘点较多时压栈* **public** EmpMemento getMemento() {  
 **return memento**;  
 }  
  
 **public void** setMemento(EmpMemento memento) {  
 **this**.**memento** = memento;  
 }  
}

**调用：**

**public class** Client {  
 **public static void** main(String[] args) {  
 CareTaker taker = **new** CareTaker();  
 Emp emp = **new** Emp(**"diao"**,23,-770);  
 System.***out***.println(**"第一次打印对象:"**+emp);  
 taker.setMemento(emp.memento());*//备忘一次* emp.setAge(38);  
 emp.setSalary(666666);  
 System.***out***.println(**"第二次打印对象:"**+emp);  
 emp.recovery(taker.getMemento());*//恢复到备忘录对象保存的状态* System.***out***.println(**"第三次打印对象:"**+emp);  
 }  
}

**结果：**

第一次打印对象:Emp{ename='diao', age=23, salary=-770.0}

第二次打印对象:Emp{ename='diao', age=38, salary=666666.0}

第三次打印对象:Emp{ename='diao', age=23, salary=-770.0}