# 设计模式

## 一、UML图

### 1、UML图包括的内容

用例图、类图、时序图、合作图、状态图、活动图、构件图、部署图

### 2、类图

#### （1）类图中的关系

##### ①泛化关系

###### 描述：类继承类

###### 箭头指向：由子类指向父类

###### 箭头样式：带空白三角形的实线

##### ②实现关系

###### 描述：类实现接口

箭头指向：由类指向接口

###### 箭头样式：带空白三角形的虚线

##### ③关联关系

###### 描述：是一种拥有关系，一个类知道另一个类的属性和方法

###### 箭头指向：拥有类的类指向被拥有的类

###### 箭头样式：带箭头的实线或者不带箭头的实线/带双箭头的实线（代表互相关联）

##### ④聚合关系

###### 描述：是关联关系的加强版，聚合是整体和个体之间的关系。如：汽车类与引擎类、轮胎类等，无法判断是否是聚合关系时，可以视为关联关系

###### 箭头指向：拥有类的类指向被拥有的类

###### 箭头样式：以空心棱形为起点，带有箭头的实线

##### ⑤组合关系（合成关系）

###### 描述：是聚合关系的强化版。代表部分的对象不能被其他的代表整体的对象共享。由代表整体的对象全权负责其生命周期。当不能判断是组合关系时，可以视为聚合关系或关联关系

###### 箭头指向：拥有类的类指向被拥有的类

###### 箭头样式：以实心棱形为起点，带有箭头的实线

###### *例子：美猴王类，四肢类，金箍棒类。美猴王对象作为整体，与负责四肢对象的生命周期（不能与他人共享），因此美猴王与四肢是组成关系。金箍棒可以被别人盗了，因此美猴王与金箍棒是聚合关系*

##### ⑥依赖关系

###### 描述：代表部分的对象以参数的形式传递到代表整体的对象的方法中

箭头指向：整体指向部分

###### 箭头样式：带箭头的虚线

## 二、面向对象的设计原则

### 1、开-闭原则

#### （1）内容：在设计一个模块的时候，应使这个模块不被修改的前提下被扩展（对接口或抽象类封闭，对实现或继承开放）

#### （2）与其他设计原则的关系：是开-闭原则的手段和工具。

### 2、里氏替换原则：任何基类可以出现的地方，子类一定可以出现，违反里氏替换原则就一定违反开-闭原则，反之不一定成立。

### 3、依赖倒转原则：依赖于抽象，不依赖于实现。是开-闭原则的手段。

### 4、合成/聚合复用原则：尽量使用合成/聚合，而不是使用继承关系达到复用的目的。在使用继承关系之前，这个条件必须满足。

### 5、迪米特原则。一个软件实体应该尽量少的与其他实体发生关系。

### 6、接口隔离原则。为客户端尽可能提供小的接口，而不要提供大的总接口。减少与其他实体发生关系（迪米特原则的方法）

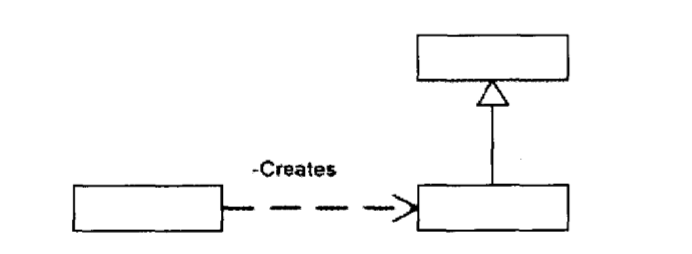
## 三、设计模式

在开始设计模式的学习时，要清楚明白的了解到各个设计模式是可以互相融合。有可能这个设计模式就使用到了另外一个设计模式。比如享元模式可以使用单例模式来实现。

### 1、工厂模式

#### 简单工厂模式（静态工厂方法模式）

##### ①简略类图



##### ②常规实例

public interface Fruit {  
 String name();  
}

public class Apple implements Fruit{  
 @Override  
 public String name() {  
 return "apple";  
 }  
}

public class Grape implements Fruit {  
 @Override  
 public String name() {  
 return "grape";  
 }  
}

public class Strawberry implements Fruit {  
 @Override  
 public String name() {  
 return "strawberry";  
 }  
}

public class FruitGardener {  
 public static Fruit factory(FruitEnum fruit) {  
 switch (fruit) {  
 case *APPLE*:  
 return new Apple();  
 case *GRAPE*:  
 return new Grape();  
 case *STRAWBERRY*:  
 return new Strawberry();  
 default:  
 throw new RuntimeException("没有这种水果");  
 }  
 }  
}

public class Demo1 {  
 public static void main(String[] args) {  
 Fruit apple = FruitGardener.*factory*(FruitEnum.*APPLE*);  
 Fruit grape = FruitGardener.*factory*(FruitEnum.*GRAPE*);  
 Fruit strawberry = FruitGardener.*factory*(FruitEnum.*STRAWBERRY*);  
 System.*out*.println(apple.name());  
 System.*out*.println(grape.name());  
 System.*out*.println(strawberry.name());  
 }  
}

##### ③退化实例

###### A.退化抽象类/接口

public class Apple {  
}

public class FruitGardener {  
 public static Apple appleFactory() {  
 return new Apple();  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Apple apple = FruitGardener.*appleFactory*();  
 System.*out*.println(apple);  
 }  
}

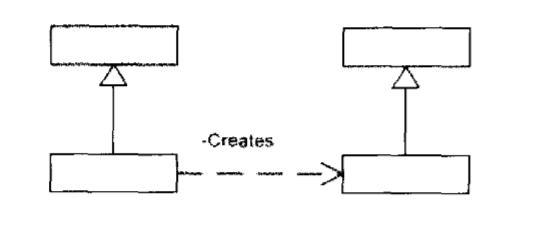
###### 退化具体产品

public class Apple {  
 public static Apple appleFactory() {  
 return new Apple();  
 }  
}

注意：与单例模式和多利模式的区别在于：单例模式和多利模式不允许外界直接将其进行实例化，而简单工厂方法允许。

#### 工厂方法模式（多态性工厂模式）

##### ①简单类图



##### ②常规实例

水果实例与简单工厂的一致

public interface FruitGardener {  
 Fruit factory();  
}

public class AppleGardener implements FruitGardener {  
 @Override  
 public Fruit factory() {  
 return new Apple();  
 }  
}

public class GrapeGardener implements FruitGardener {  
 @Override  
 public Fruit factory() {  
 return new Grape();  
 }  
}

public class StrawberryGardener implements FruitGardener {  
 @Override  
 public Fruit factory() {  
 return new Strawberry();  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Fruit apple = new AppleGardener().factory();  
 Fruit grape = new GrapeGardener().factory();  
 Fruit strawberry = new StrawberryGardener().factory();  
 System.*out*.println(apple.name());  
 System.*out*.println(grape.name());  
 System.*out*.println(strawberry.name());  
 }  
**}**

##### ③多态的丧失和模式的退化

###### 产品抽象类/接口取消

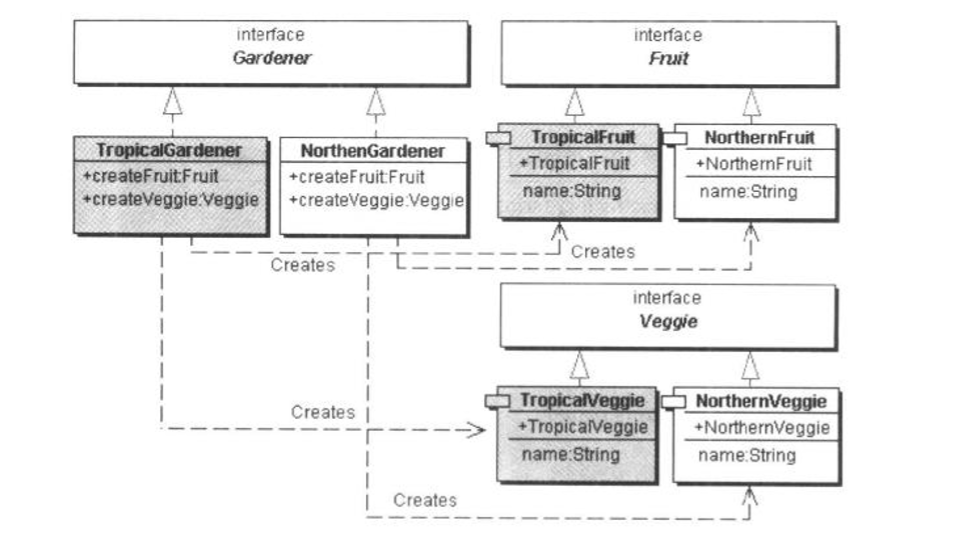
产品多态的丧失，返回值已经不是多种情况了，失去了工厂方法的最重要的设计意图，此时已经不能叫工厂方法模式了。

###### 工厂抽象类/接口的取消

工厂多态的丧失，仍可发挥工厂方法的部分意图，此时可用简单工厂方法代替。（此种退化的工厂方法模式与简单工厂模式的区别在于，工厂是否是静态的）

#### 抽象工厂模式（工具箱模式）

##### ①类图



##### ②实例

public interface Fruit {  
 String position();  
}

public class TropicalFruit implements Fruit {  
 @Override  
 public String position() {  
 return "热带水果";  
 }  
}

public class NorthernFruit implements Fruit {  
 @Override  
 public String position() {  
 return "北方水果";  
 }  
}

public interface Veggie {  
 String position();  
}

public class TropicalVeggie implements Veggie {  
 @Override  
 public String position() {  
 return "热带蔬菜";  
 }  
}

public class NorthernVeggie implements Veggie{  
 @Override  
 public String position() {  
 return "北方蔬菜";  
 }  
}

public interface Gardener {  
 Fruit fruitFactory();  
  
 Veggie veggieFactory();  
}

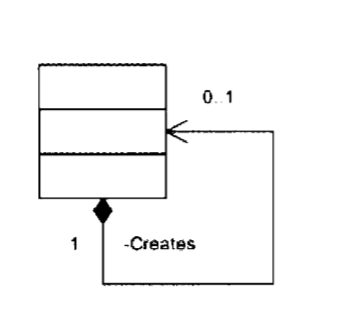
public class TropicalGardener implements Gardener{  
 @Override  
 public Fruit fruitFactory() {  
 return new TropicalFruit();  
 }  
  
 @Override  
 public Veggie veggieFactory() {  
 return new TropicalVeggie();  
 }  
}

public class NorthernGardener implements Gardener {  
 @Override  
 public Fruit fruitFactory() {  
 return new NorthernFruit();  
 }  
  
 @Override  
 public Veggie veggieFactory() {  
 return new NorthernVeggie();  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Fruit northernFruit = new NorthernGardener().fruitFactory();  
 Veggie northernVeggie = new NorthernGardener().veggieFactory();  
 Fruit tropicalFruit = new TropicalGardener().fruitFactory();  
 Veggie tropicalVeggie = new TropicalGardener().veggieFactory();  
 System.*out*.println(northernFruit.position());  
 System.*out*.println(northernVeggie.position());  
 System.*out*.println(tropicalFruit.position());  
 System.*out*.println(tropicalVeggie.position());  
 }  
}

### 单例模式

#### 单例模式简单类图



#### 饿汉式

public class Hungry {  
 private static Hungry *hungry* = new Hungry();  
 private Hungry() { }  
 public static Hungry newInstance() {  
 return *hungry*;  
 }  
}

#### 懒汉式

##### ①直接加锁

public class Ordinary {  
 private Ordinary(){};  
 synchronized public static Ordinary newInstance(){  
 return new Ordinary();  
 }  
}

##### ②静态内部类

public class StaticInnerClass {  
 private StaticInnerClass(){}  
 private static class Instacne{  
 private static StaticInnerClass *staticInnerClass*=new StaticInnerClass();  
 }  
 public static StaticInnerClass newInstance(){  
 return Instacne.*staticInnerClass*;  
 }  
}

##### ③加volatile的双重校验锁（java才需要加volatile，其他语言可以不加就直接运行）

public class DoubleCheckLock {  
 private static volatile DoubleCheckLock *doubleCheckLock*;  
 private DoubleCheckLock() {}  
 public static DoubleCheckLock newInstance() {  
 if (*doubleCheckLock* == null) {  
 synchronized (DoubleCheckLock.class) {  
 if (*doubleCheckLock* == null) {  
 *doubleCheckLock* = new DoubleCheckLock();  
 }  
 }  
 }  
 return *doubleCheckLock*;  
 }  
}

解释：

Q：为什么要加synchronized？

A：在多线程的环境下，懒汉式不能确保创建出来的对象就是唯一的，可能两个线程同时在创建新的对象

Q：为什么要有两个if语句判空

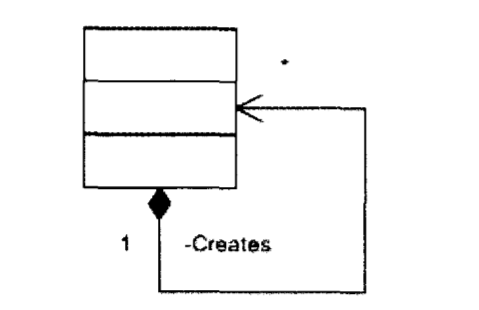
A：因为synchronized非常占用系统资源，粒度越小越好，且访问的次数越少越好，在synchronized前加一个if判断是否这个单例已经被创建了，如果被创建了就直接返回，不用再进入synchronized中。大大减少被访问的次数。

Q：java中为什么要加volatile？

A：jvm运行的原则是：在单线程中，在不影响指令结果的情况下，会对指令进行重排序，以此来达到优化程序执行的效果。 doubleCheckLock = new DoubleCheckLock()本质上不是一个原子操作，它一共分为三步：一、在内存中开辟一个新的空间；二、初始化对象；三、doubleCheckLock 指向刚分配的空间。由于在单线程中一二三和一三二执行的结果是没有变化的。因此jvm中会对其指令进行重排序。那么就可能会造成A线程按一三二的顺序执行到三，此时B线程到了 if (doubleCheckLock == null) 阶段，发现它不为空，于是直接return一个脏的值出来。（此时对象还没初始化，没成功创建）

### 多例模式（单例模式的扩展）

#### 简单类图

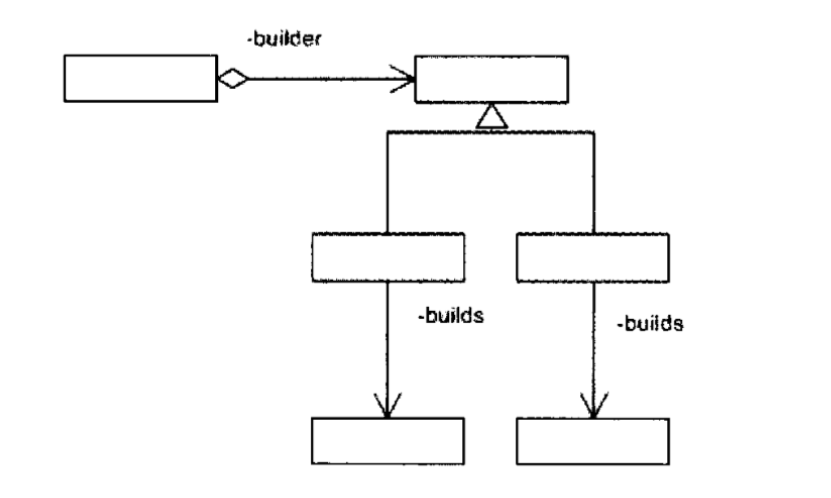


#### 实例

public class Multiton {  
 private Multiton() { }  
 private static Map<LanguageEnum, Multiton> *map*;  
 synchronized public static Multiton newIntance(LanguageEnum language) {  
 if (*map* == null) {  
 *map* = new HashMap<>();  
 }  
 if (*map*.containsKey(language)) {  
 return *map*.get(language);  
 }  
 *map*.put(language, new Multiton());  
 return *map*.get(language);  
 }  
}  
enum LanguageEnum {  
 *CHINESE*,  
 *ENGLISH*,  
 *JAPANESE*;  
}

### 建造模式

#### 简单类图



#### 实例

public interface Computer {  
 void add(String event);  
 void print();  
}

public class DesktopPC implements Computer{  
 List<String> list=new ArrayList<>();  
  
 @Override  
 public void add(String event) {  
 list.add("台式电脑："+event);  
 }  
 @Override  
 public void print() {  
 System.*out*.println(list.toString());  
 }  
}

public class Laptop implements Computer {  
 List<String> list = new ArrayList<>();  
  
 @Override  
 public void add(String event) {  
 list.add("笔记本电脑：" + event);  
 }  
  
 @Override  
 public void print() {  
 System.*out*.println(list.toString());  
 }  
}

public interface ComputerBuilder {  
 void buildCPU();  
 void buildGraphicsCard();  
 Computer retrieveResult();  
}

public class DesktopPCBuilder implements ComputerBuilder {  
 private Computer desktopPC = new DesktopPC();  
 private boolean isBuildCPU = false;  
 private boolean isBuildGraphicsCard = false;  
 @Override  
 public void buildCPU() {  
 if (!isBuildCPU) {  
 desktopPC.add("组装台式型号的cpu");  
 isBuildCPU = true;  
 }  
 }  
 @Override  
 public void buildGraphicsCard() {  
 if (!isBuildGraphicsCard) {  
 desktopPC.add("组装台式型号的显卡");  
 isBuildGraphicsCard = true;  
 }  
 }  
 @Override  
 public Computer retrieveResult() {  
 return desktopPC;  
 }  
}

public class LaptopBuilder implements ComputerBuilder {  
 private Computer laptop = new Laptop();  
 private boolean isBuildCPU = false;  
 private boolean isBuildGraphicsCard = false;  
 @Override  
 public void buildCPU() {  
 if (!isBuildCPU) {  
 laptop.add("组装笔记本型号的cpu");  
 isBuildCPU = true;  
 }  
 }  
 @Override  
 public void buildGraphicsCard() {  
 if (!isBuildGraphicsCard) {  
 laptop.add("组装笔记本型号的显卡");  
 isBuildGraphicsCard = true;  
 }  
 }  
 @Override  
 public Computer retrieveResult() {  
 return laptop;  
 }  
}

#### 合并建造者和产品

public class Car {  
 private Car() {}  
 private String wheel;  
 private String engine;  
 public static class CarBuilder {  
 Car car = new Car();  
 public CarBuilder setWheel(String wheel) {  
 car.setWheel(wheel);  
 return this;  
 }  
 public CarBuilder setEngine(String engine) {  
 car.setEngine(engine);  
 return this;  
 }  
 public Car build() {  
 return car;  
 }  
 }  
 private void setWheel(String wheel) {  
 this.wheel = wheel;  
 }  
 private void setEngine(String engine) {  
 this.engine = engine;  
 }  
 public String getWheel() {  
 return wheel;  
 }  
 public String getEngine() {  
 return engine;  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 CarBuilder carBuilder = new CarBuilder();  
 carBuilder.setEngine("引擎");  
 carBuilder.setWheel("轮子");  
 Car car = carBuilder.build();  
 System.*out*.println(car.getEngine());  
 System.*out*.println(car.getWheel());  
 }  
}

Effect Java中推荐当一个类的构造函数里面需要填入的参数大于四个，且变动性比较大的时候（可以填任意0-4个参数），使用此种方法。

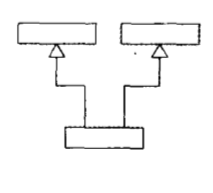
### 原型模式

略

### 适配器模式

#### 类适配器

##### ①简单类图



##### ②实例

public class TwoPinPlug {  
 private boolean fireLine = false;  
 private boolean zeroLine = false;  
 public void setFireLine(){  
 fireLine = true;  
 }  
 public void setZeroLine(){  
 zeroLine = true;  
 }  
  
 public boolean getFireLine() {  
 return fireLine;  
 }  
  
 public boolean getZeroLine() {  
 return zeroLine;  
 }  
}

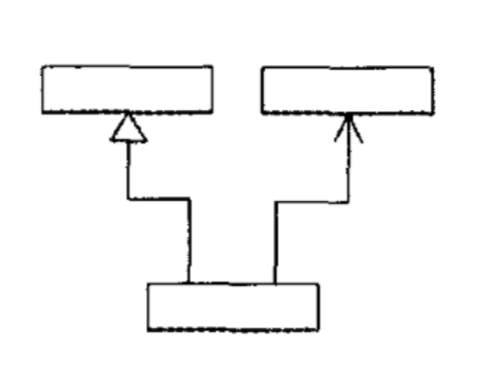
public interface ThreePinPlug {  
 void setFireLine();  
 void setZeroLine();  
 void setGroundWire();  
 boolean getGroundWire();  
}

public class PlugAdapter extends TwoPinPlug implements ThreePinPlug{  
 private boolean groundWire = false;  
 @Override  
 public void setGroundWire() {  
 groundWire = true;  
 }  
  
 public boolean getGroundWire() {  
 return groundWire;  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 ThreePinPlug plugAdapter = new PlugAdapter();  
 plugAdapter.setGroundWire();  
 boolean groundWire = plugAdapter.getGroundWire();  
 System.*out*.println(groundWire);  
 }  
}

#### 对象适配器

##### ①简单类图



##### ②实例

public class TwoPinPlug {  
 private boolean fireLine = false;  
 private boolean zeroLine = false;  
 public void setFireLine(){  
 fireLine = true;  
 }  
 public void setZeroLine(){  
 zeroLine = true;  
 }  
  
 public boolean getFireLine() {  
 return fireLine;  
 }  
  
 public boolean getZeroLine() {  
 return zeroLine;  
 }  
}

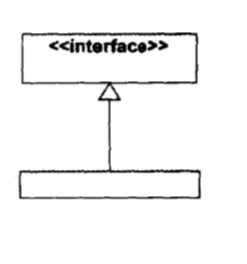
public interface ThreePinPlug {  
 void setFireLine();  
 void setZeroLine();  
 void setGroundWire();  
 boolean getFireLine();  
 boolean getZeroLine();  
 boolean getGroundWire();  
}

public class PlugAdapter implements ThreePinPlug {  
 private TwoPinPlug twoPinPlug;  
 private boolean groundWire = false;  
 @Override  
 public void setFireLine() {  
 twoPinPlug.setFireLine();  
 }  
 @Override  
 public void setZeroLine() {  
 twoPinPlug.setZeroLine();  
 }  
 @Override  
 public boolean getFireLine() {  
 return twoPinPlug.getFireLine();  
 }  
 @Override  
 public boolean getZeroLine() {  
 return twoPinPlug.getZeroLine();  
 }  
 @Override  
 public void setGroundWire() {  
 groundWire = true;  
 }  
 public boolean getGroundWire() {  
 return groundWire;  
 }  
}

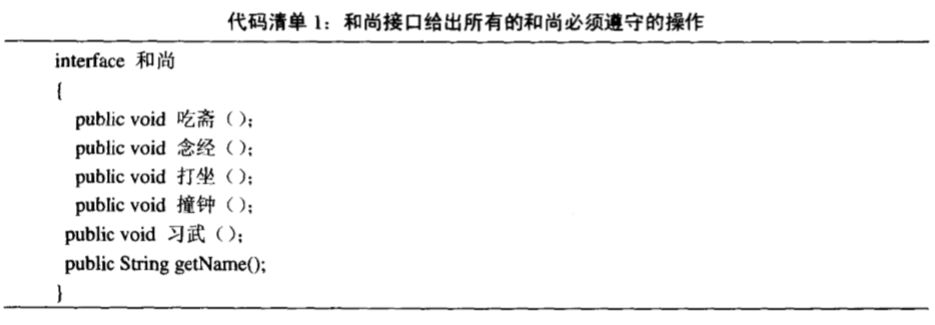
public class Main {  
 public static void main(String[] args) {  
 ThreePinPlug threePinPlug = new PlugAdapter();  
 boolean groundWire = threePinPlug.getGroundWire();  
 System.*out*.println(groundWire);  
 }  
}

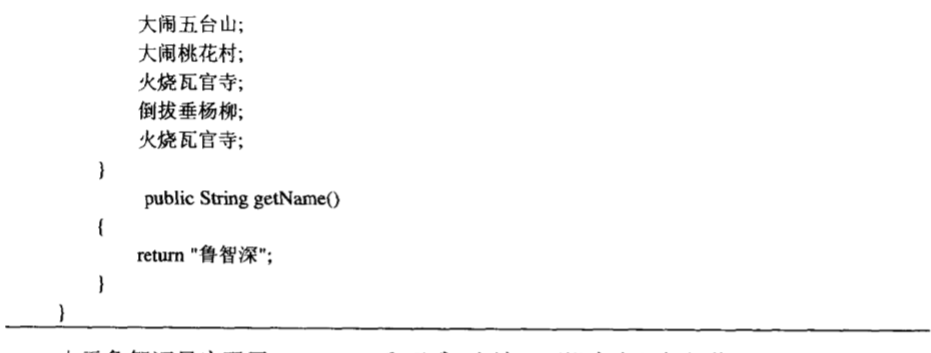
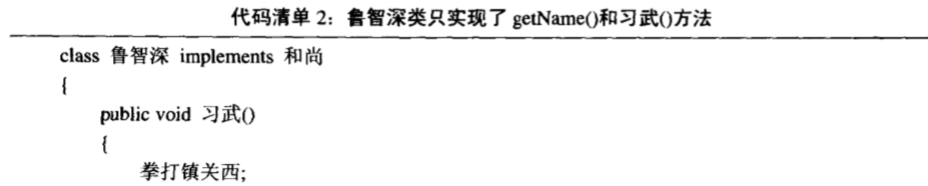
#### 缺省适配器模式

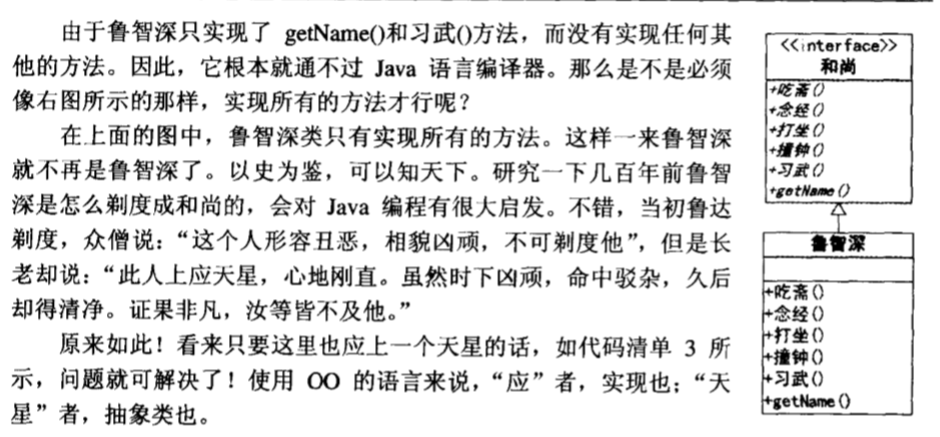
##### ①简单类图

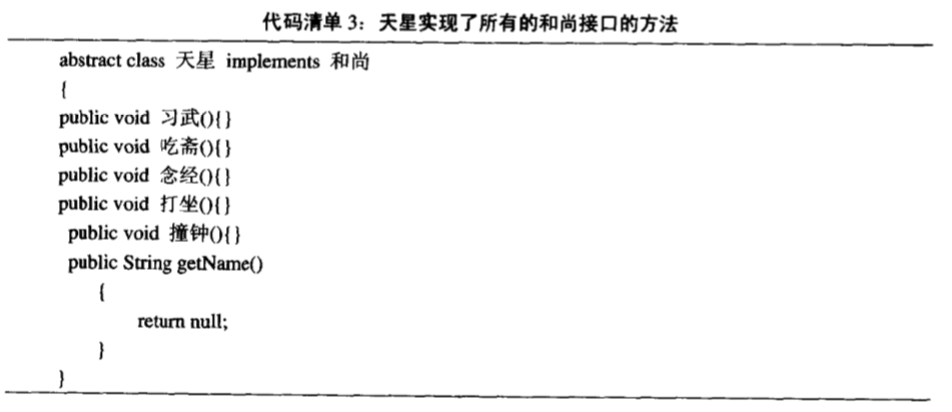


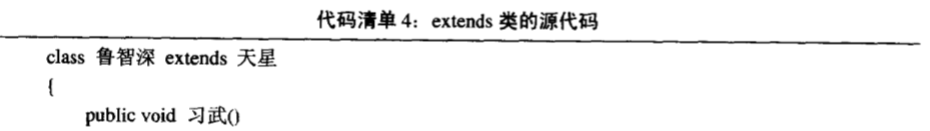
##### ②实例

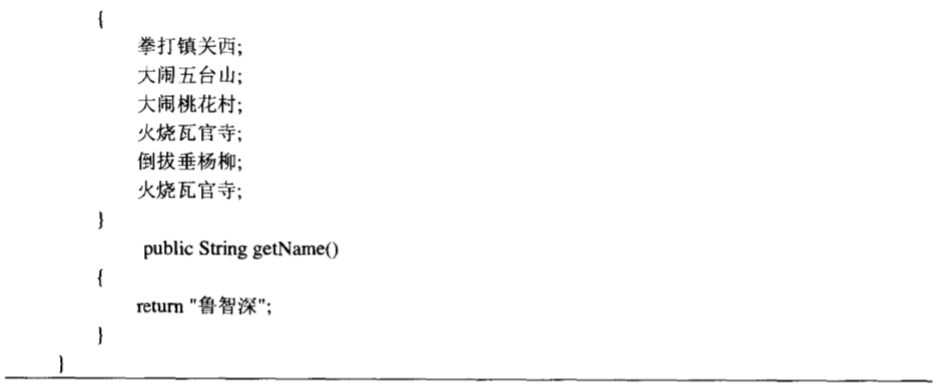


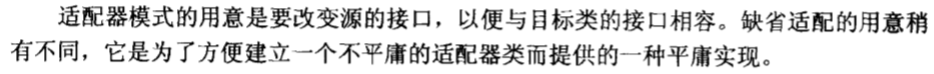
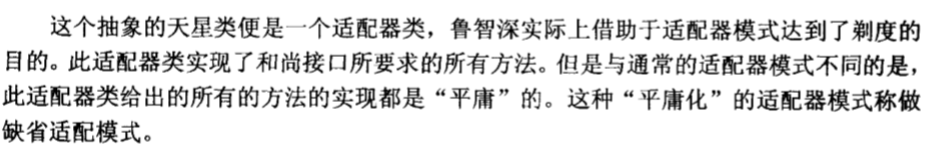






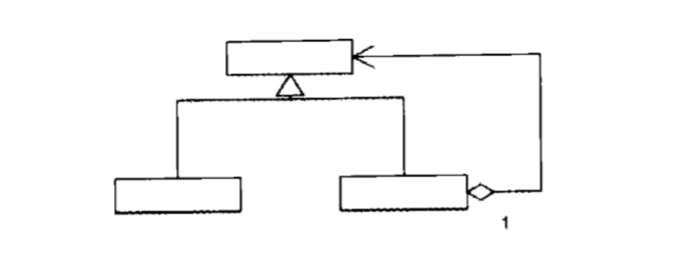






### 合成模式

#### 简单类图



#### 实例（最典型的文件系统）

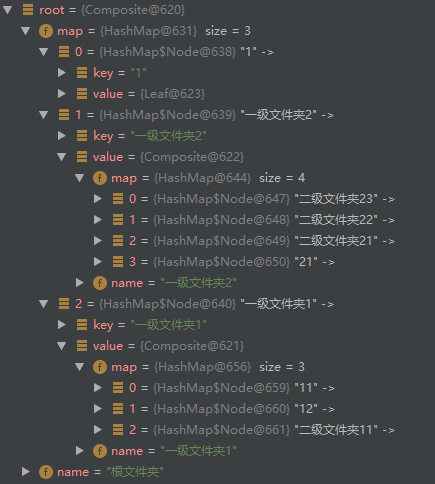
public interface Component {  
 String getName();  
 void setName(String name);  
}

public class Composite implements Component {  
  
 private Map<String, Component> map = new HashMap<>();  
 private String name;  
 public Composite(String name) {  
 this.name = name;  
 }  
 */\*\*  
 \* 添加子文件/文件夹  
 \*/* public void addChildComponent(Component component) {  
 if (!map.containsKey(component.getName())) {  
 map.put(component.getName(), component);  
 return;  
 }  
 throw new RuntimeException("存在相同的文件名，添加子文件失败");  
 }  
 */\*\*  
 \* 删除子文件/文件夹  
 \*/* public void removeChildComponent(Component component) {  
 if (map.containsValue(component)) {  
 map.remove(component.getName());  
 return;  
 }  
 throw new RuntimeException("不是当前文件夹的子文件，无法删除");  
 }  
 */\*\*  
 \* 获取名字  
 \*/* public String getName() {  
 return name;  
 }  
 */\*\*  
 \* 改名  
 \*/* public void setName(String name) {  
 Objects.*requireNonNull*(name, "文件名不能为空");  
 this.name = name;  
 }  
}

public class Leaf implements Component {  
 private String name;  
 public Leaf(String name) {  
 this.name = name;  
 }  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 Objects.*requireNonNull*(name, "文件名不能为空");  
 this.name = name;  
 }  
}

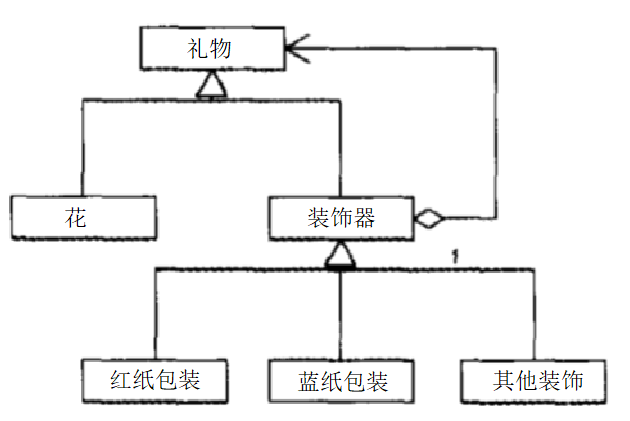
public class Main {  
 public static void main(String[] args) {  
 Composite root = new Composite("根文件夹");  
 Composite dir1 = new Composite("一级文件夹1");  
 Composite dir2 = new Composite("一级文件夹2");  
 Leaf file1 = new Leaf("1");  
 Composite dir11 = new Composite("二级文件夹11");  
 Composite dir21 = new Composite("二级文件夹21");  
 Composite dir22 = new Composite("二级文件夹22");  
 Composite dir23 = new Composite("二级文件夹23");  
 Leaf file11 = new Leaf("11");  
 Leaf file12 = new Leaf("12");  
 Leaf file21 = new Leaf("21");  
 root.addChildComponent(dir1);  
 root.addChildComponent(dir2);  
 root.addChildComponent(file1);  
 dir1.addChildComponent(dir11);  
 dir1.addChildComponent(file11);  
 dir1.addChildComponent(file12);  
 dir2.addChildComponent(dir21);  
 dir2.addChildComponent(dir22);  
 dir2.addChildComponent(dir23);  
 dir2.addChildComponent(file21);  
 System.*out*.println(root);  
 }  
}

结果：



### 装饰器模式

#### 类图



#### 实例（最典型的实例是io流）

public interface Gift {  
 void packing();  
}

public class Flower implements Gift {  
  
 @Override  
 public void packing() {  
 System.*out*.println("花");  
 }  
}

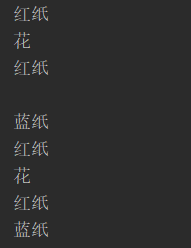
public class Decorator implements Gift {  
 private Gift gift;  
  
 public Decorator(Gift gift) {  
 this.gift = gift;  
  
 }  
  
 public void packing() {  
 //要有多层包装的效果关键在于调用聚合的对象的方法，而不是调用父类的。  
 gift.packing();  
 }  
}

public class RedWrapping extends Decorator {  
 public RedWrapping(Gift gift) {  
 super(gift);  
 }  
 public void packing() {  
 System.*out*.println("红纸");  
 super.packing();  
 System.*out*.println("红纸");  
 }  
}

public class BlueWrapping extends Decorator {  
 public BlueWrapping(Gift gift) {  
 super(gift);  
 }  
 public void packing() {  
 System.*out*.println("蓝纸");  
 super.packing();  
 System.*out*.println("蓝纸");  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 RedWrapping redWrapping = new RedWrapping(new Flower());  
 redWrapping.packing();  
 System.*out*.println();  
 BlueWrapping blueWrapping = new BlueWrapping(redWrapping);  
 blueWrapping.packing();  
 }  
}

结果：



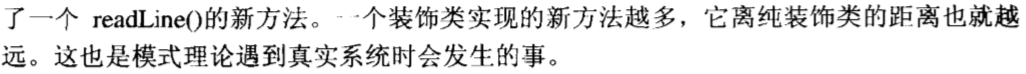
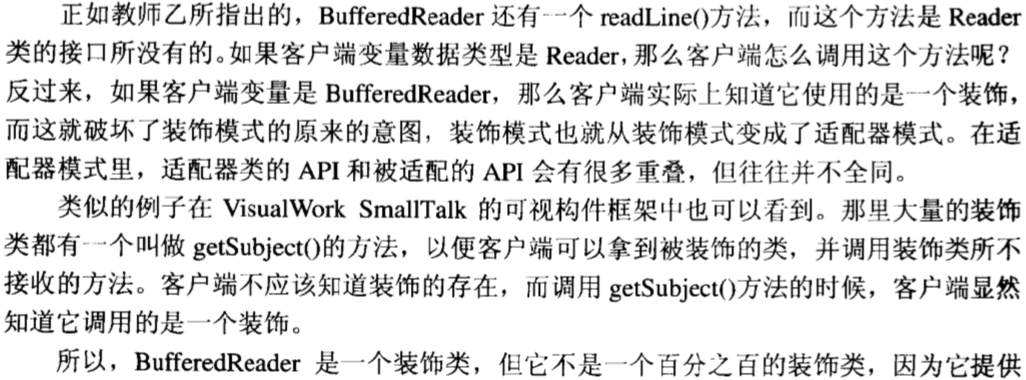
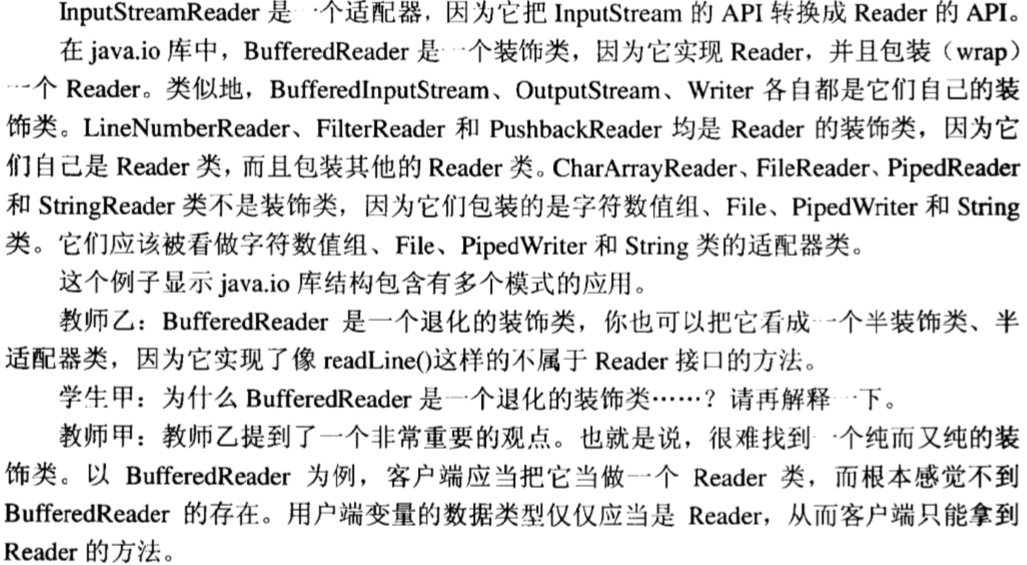
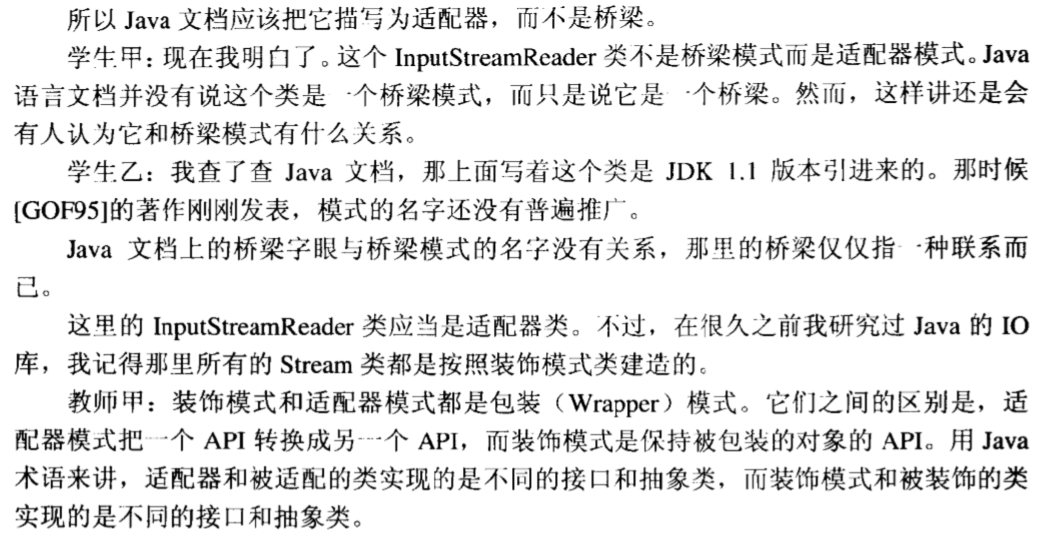
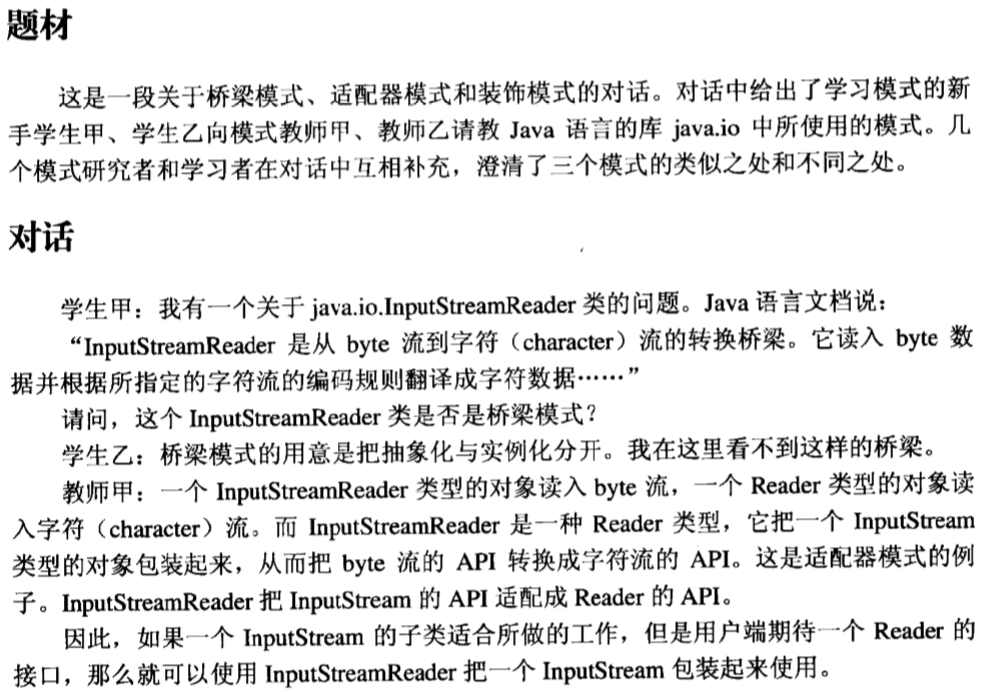
可以不断的组合，往外面多包几层

可以看到装饰器模式和组合模式的类图很像。

两者的代码结构也很相似，区别在于实现的目的不同，导致其代码的略微区别。

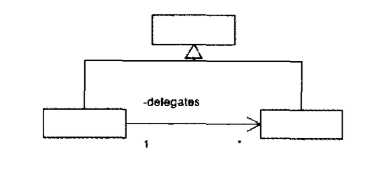
装饰器模式是为了对功能的增强，因此代码更侧重于在传入的对象原有的方法上包装一层

组合模式是为了让其各部分的结构更加清晰，是一个树形结构。将传入的对象放到了属性中保存。做为其子部分。



### 代理模式

#### 类图



#### 静态代理实例（在编译后就进行了代理）

public interface Subject {  
 void insert();  
}

public class RealSubject implements Subject{  
 @Override  
 public void insert() {  
 System.*out*.println("插入数据库");  
 }  
}

public class ProxySubject implements Subject {  
  
 private RealSubject realSubject;  
  
 public ProxySubject(RealSubject realSubject) {  
 this.realSubject = realSubject;  
 }  
  
 @Override  
 public void insert() {  
 System.*out*.println("开启事务");  
 realSubject.insert();  
 System.*out*.println("关闭事务");  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Subject subject=new ProxySubject(new RealSubject());  
 subject.insert();  
 }  
}

#### 动态代理（在编译前就进行了代理，不需要代理类）

使用反射

#### 代理模式和装饰器模式的区别

两者的目的是一样的，都是为了增强功能。但是又略有区别：

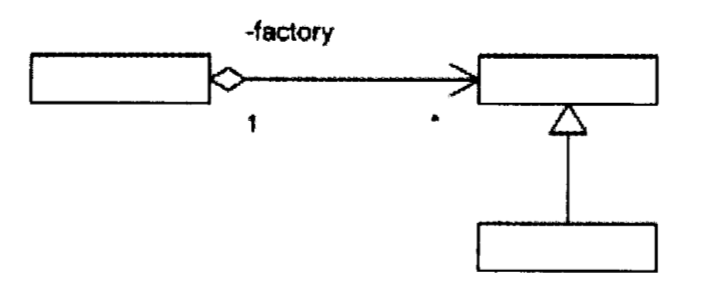
装饰器模式是可以对接口下的所有类都进行反复的增强，不断地在外面进行包装

代理模式则是代理类只对被代理的对象的一次增强。

两者并没有很明显的界限，如果非要严格区分的话，一个增强的是自身（装饰器模式），你还是你。一个则是委托给了别人了（代理模式），你不是你了。

### 享元模式

#### 类图



#### 实例（典型实例String）

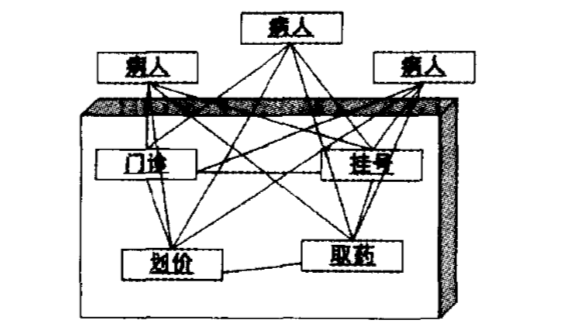
略

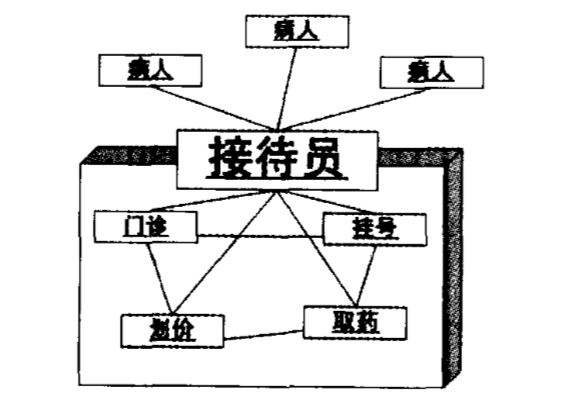
#### 享元模式与其他模式组合

享元模式组合单例模式，组合工厂模式，组合备忘录模式状态模式、策略模式等进行组合。享元模式不是单纯的设计模式。可以和多种设计模式组合以此来达到效果。

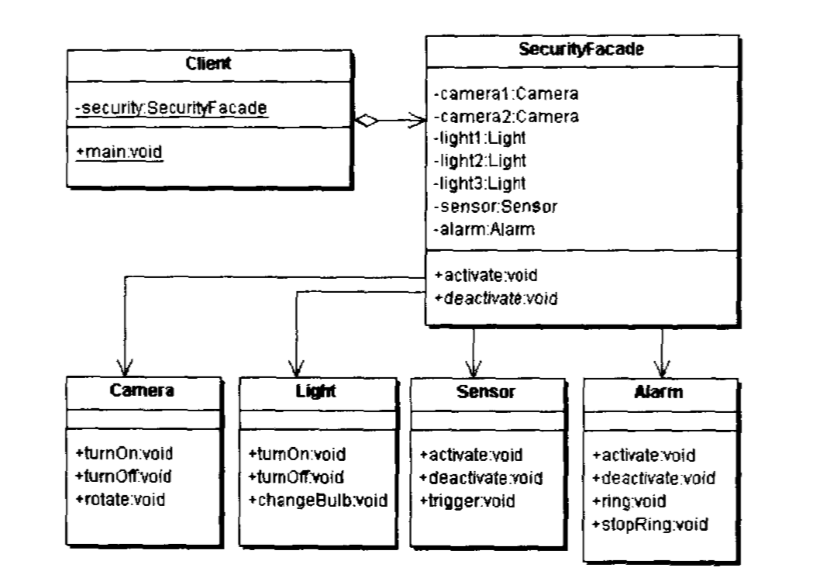
### 门面模式

#### 示意图



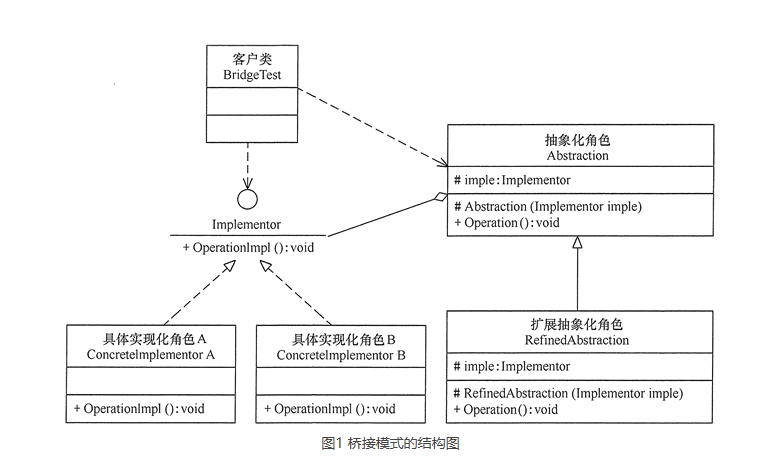


#### 实例



### 桥梁模式（变继承为聚合）

#### 类图



#### 代码实例

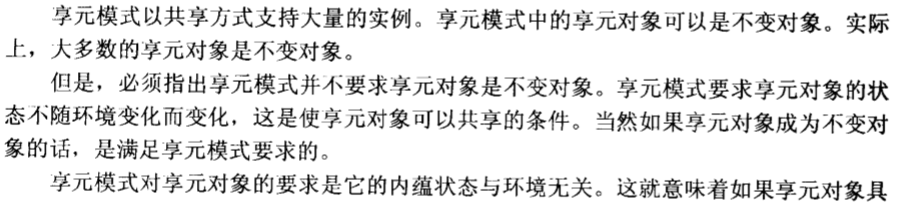
public class BridgeTest  
{  
 public static void main(String[] args)  
 {  
 Implementor imple=new ConcreteImplementorA();  
 Abstraction abs=new RefinedAbstraction(imple);  
 abs.Operation();  
 }  
}  
//实现化角色  
interface Implementor  
{  
 public void OperationImpl();  
}  
//具体实现化角色  
class ConcreteImplementorA implements Implementor  
{  
 public void OperationImpl()  
 {  
 System.*out*.println("具体实现化(Concrete Implementor)角色被访问" );  
 }  
}  
//抽象化角色  
abstract class Abstraction  
{  
 protected Implementor imple;  
 protected Abstraction(Implementor imple)  
 {  
 this.imple=imple;  
 }  
 public abstract void Operation();  
}  
//扩展抽象化角色  
class RefinedAbstraction extends Abstraction  
{  
 protected RefinedAbstraction(Implementor imple)  
 {  
 super(imple);  
 }  
 public void Operation()  
 {  
 System.*out*.println("扩展抽象化(Refined Abstraction)角色被访问" );  
 imple.OperationImpl();  
 }  
}

### 不变模式

#### 特点：不变

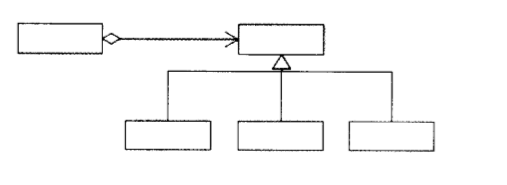
#### 实例：java中的String以及包装类（Integer、Long...）

#### 享元模式与不变模式的关系



### 策略模式

#### 简单类图



#### 实例

public interface Sort {  
 void execute();  
}

public class MergeSort implements Sort {  
 @Override  
 public void execute() {  
 System.*out*.println("归并排序");  
 }  
}

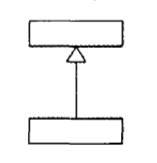
public class QuickSort implements Sort {  
 @Override  
 public void execute() {  
 System.*out*.println("快速排序");  
 }  
}

public class Sorter {  
 private Sort sort;  
 public Sorter(Sort sort) {  
 this.sort = sort;  
 }  
 public void sort() {  
 sort.execute();  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Sorter sorter = new Sorter(new MergeSort());  
 sorter.sort();  
 }  
}

### 模板方法模式——最常见的设计模式之一

#### 简单类图（实现类实现抽象类）

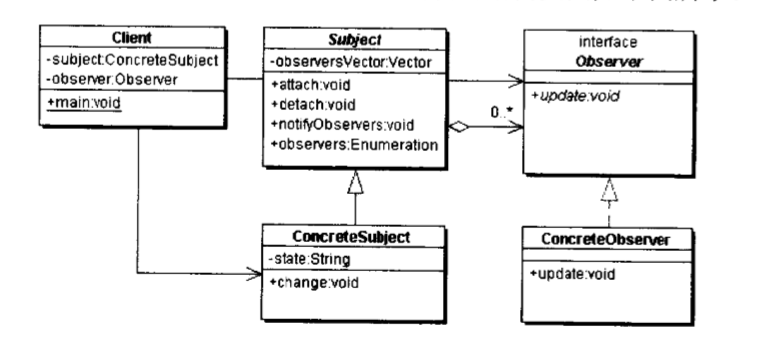


#### 实例

略

### 观察者模式

#### 类图



#### 实例

public interface Observer {  
 void update();  
}

public class ConcreteObserver implements Observer {  
 @Override  
 public void update() {  
 System.*out*.println("被观察的主题更新了");  
 }  
}

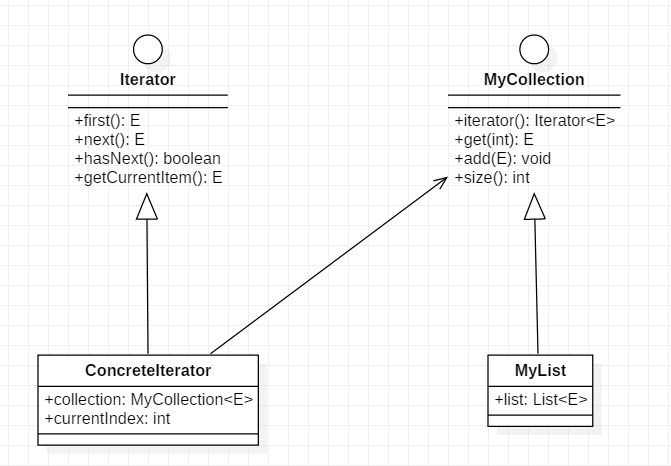
public abstract class Subject {  
 */\*\*  
 \* 多个观察者监听一个主题  
 \*/* private List<Observer> list = new ArrayList<>();  
  
 public void register(Observer observer) {  
 if (!list.contains(observer)) {  
 list.add(observer);  
 return;  
 }  
 throw new RuntimeException("该观察者已经注册");  
 }  
  
 public void remove(Observer observer) {  
 if (!list.contains(observer)) {  
 list.remove(observer);  
 return;  
 }  
 throw new RuntimeException("没有这个观察者可被移除");  
 }  
  
 public abstract void execute();  
  
 void notifyObserver() {  
 list.forEach(observer ->observer.update());  
 }  
}

public class ConcreteSubject extends Subject {  
 @Override  
 public void execute() {  
 System.*out*.println("实体的主题执行了");  
 //通知观察者  
 notifyObserver();  
 }  
}

注意：java自身就支持观察者模式，有Observer的接口

### 迭代器模式

#### 类图



#### 实例

public interface Iterator<E> {  
 E first();  
 E next();  
 boolean hasNext();  
 E getCurrentItem();  
}

public class ConcreteIterator<E> implements Iterator<E> {  
 private MyCollection<E> collection;  
 private int currentIndex = -1;  
 public ConcreteIterator(MyCollection<E> collection) {  
 this.collection = collection;  
 }  
 public E first() {  
 return collection.get(0);  
 }  
 public E next() {  
 if (hasNext()) {  
 return collection.get(++currentIndex);  
 }  
 return null;  
 }  
 public boolean hasNext() {  
 if (currentIndex + 1 < collection.size()) {  
 return true;  
 }  
 return false;  
 }  
 public E getCurrentItem() {  
 if (currentIndex < 0) {  
 return null;  
 }  
 return collection.get(currentIndex);  
 }  
}

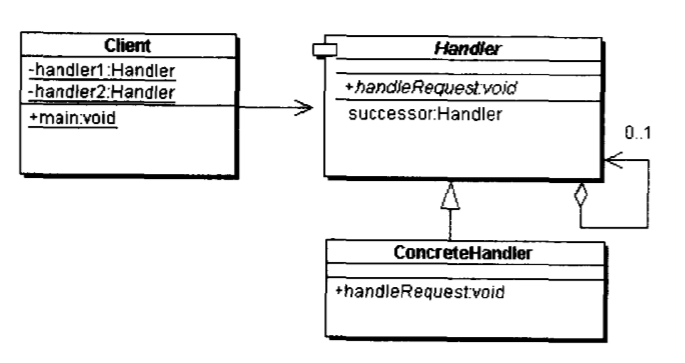
public interface MyCollection<E> {  
 Iterator<E> iterator();  
 E get(int index);  
 void add(E e);  
 int size();  
}

public class MyList<E> implements MyCollection<E> {  
 private List<E> list = new ArrayList<>();  
 public void add(E e) {  
 list.add(e);  
 }  
 public int size() {  
 return list.size();  
 }  
 public Iterator<E> iterator() {  
 return new ConcreteIterator<>(this);  
 }  
 public E get(int index) {  
 return list.get(index);  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 MyCollection<String> collection=new MyList<>();  
 collection.add("a");  
 collection.add("b");  
 collection.add("c");  
 Iterator<String> iterator = collection.iterator();  
 while (iterator.hasNext()){  
 System.*out*.println("当前："+iterator.getCurrentItem());  
 System.*out*.println("下个："+iterator.next());  
 System.*out*.println("----------------------");  
 }  
 System.*out*.println("当前："+iterator.getCurrentItem());  
 System.*out*.println(iterator.first());  
 }  
}

### 责任链模式

#### 类图



#### 实例

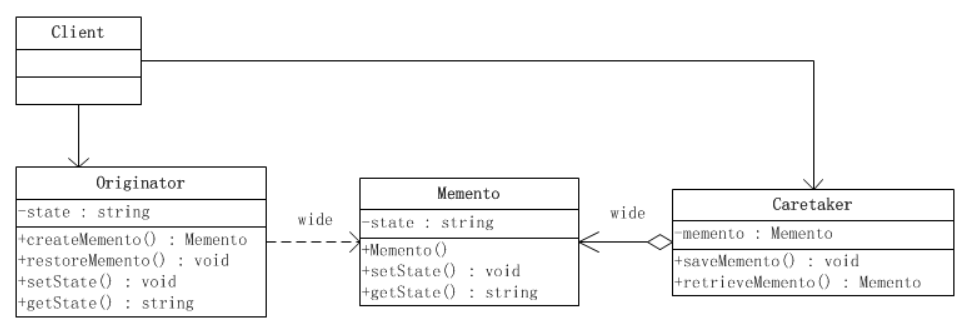
public abstract class Handler {  
 private Handler handler;  
 public abstract void handlerRequest();  
 public void setHandler(Handler handler){  
 this.handler=handler;  
 }  
 public Handler getHandler(){  
 return handler;  
 }  
}

public class ConcreteHandler extends Handler {  
 @Override  
 public void handlerRequest() {  
 if(getHandler()!=null){  
 System.*out*.println("传给下一个handler");  
 getHandler().handlerRequest();  
 }else {  
 System.*out*.println("执行了这个handler");  
 }  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Handler handler1 = new ConcreteHandler();  
 Handler handler2 = new ConcreteHandler();  
 Handler handler3 = new ConcreteHandler();  
 Handler handler4 = new ConcreteHandler();  
 handler1.setHandler(handler2);  
 handler2.setHandler(handler3);  
 handler3.setHandler(handler4);  
 handler1.handlerRequest();  
 }  
}

### 备忘录模式（将发起人的状态存储到备忘录中，只有发起人才能对备忘录进行操作）

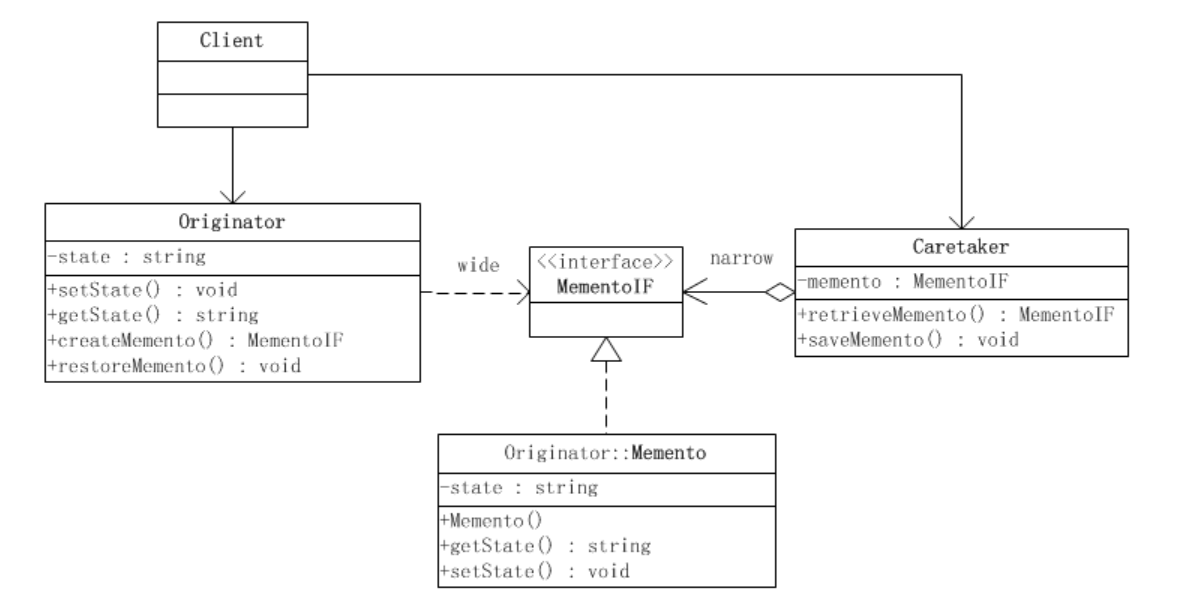
#### 白箱备忘录。（白箱备忘录将备忘录的内容暴露大家都看得到的地方，破坏了封装性，但是通过程序员的自律，仍可达到备忘录模式的部分效果）



#### 黑箱备忘录

黑箱备忘录是对白箱备忘录封装性的加强。将备忘录类放在发起人类中，并将方法设置为私有，达到其的数据只能被发起人访问的目的

##### ①类图



##### ②实例

public interface MementoIF {  
}

public class Originator {  
 private String state;  
 public String getState() {  
 return state;  
 }  
 public void setState(String state) {  
 this.state = state;  
 System.*out*.println("赋值状态：" + state);  
 }  
 public MementoIF createMemento(){  
 return new Memento(state);  
 }  
 */\*\*  
 \* 发起人恢复到备忘录对象记录的状态  
 \*/* public void restoreMemento(MementoIF memento){  
 this.setState(((Memento)memento).getState());  
 }  
 private class Memento implements MementoIF{  
 private String state;  
 */\*\*  
 \* 构造方法  
 \*/* private Memento(String state){  
 this.state = state;  
 }  
 private String getState() {  
 return state;  
 }  
 private void setState(String state) {  
 this.state = state;  
 }  
 }  
}

public class Caretaker {  
 private MementoIF memento;  
 */\*\*  
 \* 备忘录取值方法  
 \*/* public MementoIF retrieveMemento() {  
 return memento;  
 }  
 */\*\*  
 \* 备忘录赋值方法  
 \*/* public void saveMemento(MementoIF memento) {  
 this.memento = memento;  
 }  
}

代码解析：

先创建一个没有任何方法的接口（标志接口），这样子就不能对其进行任何操作。发起人（Originator ）内部创建一个内部类（备忘录），方法属性构造函数全部设置为private，且继承了标志接口，这样子就只有发起人能对备忘录进行修改操作。最后传递到外面给Caretaker 类的对象保存。但是Caretaker 无法对备忘录进行操作，只是对他起到保存的作用（保存备忘录对象的地址）