实验七 装饰模式的应用

1 实验目的

1) 掌握装饰模式（Decorator）的特点

2) 分析具体问题，使用装饰模式进行设计。

2 实验内容和要求

“喜羊羊逃命”游戏：喜羊羊被灰太狼追，喜羊羊最多5条命，灰太狼每咬到喜羊羊一次，喜羊羊就要少一条命。在逃的过程中喜羊羊可以吃到三种苹果，吃“红苹果”可以给喜羊羊加上保护罩，吃“绿苹果”可以加快喜羊羊奔跑速度，吃“黄苹果”可以使喜羊羊趟着水跑。应用装饰模式，用C#或Java控制台应用程序实现该设计。绘制该模式的UML图。

提示：这个例子如果用类的继承来实现的话那可就麻烦了，你需要为喜羊羊派生3\*2\*1=6个子类（有保护罩的喜羊羊，奔跑速度加快的喜羊羊，会趟水的喜羊羊，既有保护罩又会趟水的喜羊羊，奔跑速度快且会趟水的喜羊羊，有保护罩且奔跑速度快的喜羊羊，有保护罩、奔跑速度快且会趟水的喜羊羊），如果使用装饰模式的那就不用派生诸多子类了，当喜羊羊每吃到一个苹果，我们就用装饰模式给喜羊羊加一个动态增加一个新功能即可。

3 UML图

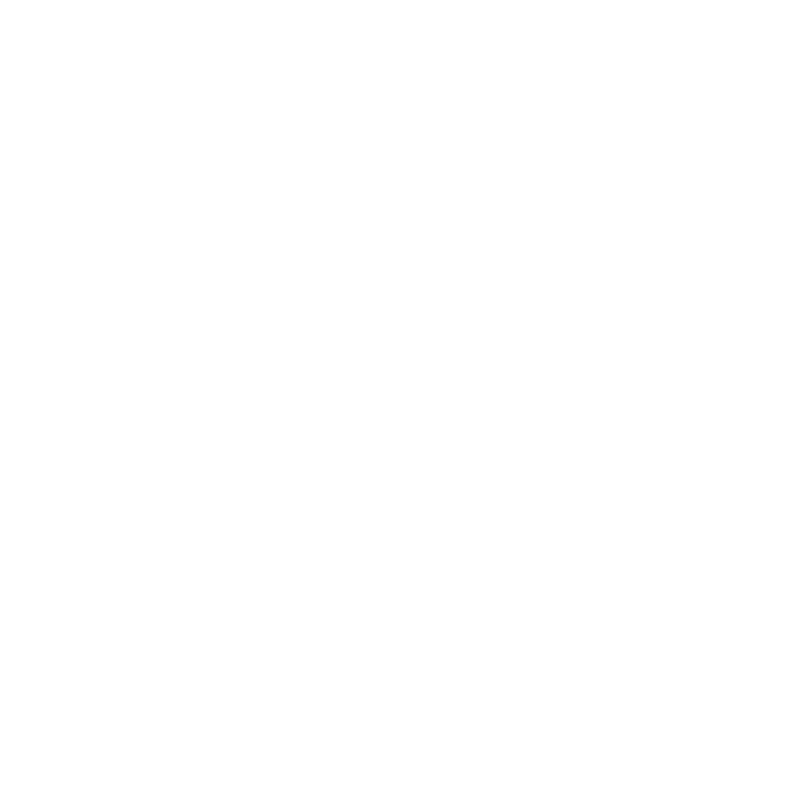


图7.3.1 类图

4 代码实现

interface Component {  
 void operation();  
}

public class Sheep implements Component {  
  
 @Override  
 public void operation() {  
 System.*out*.println("喜羊羊");  
 }  
}

abstract class Buff implements Component {  
 Component component;  
  
 public void setComponent(Component component) {  
 this.component = component;  
 }  
  
 @Override  
 public void operation() {  
 if (component!=null)  
 component.operation();  
 }  
}

public class GreenBuff extends Buff{  
 @Override  
 public void operation() {  
 System.*out*.println("奔跑速度的");  
 component.operation();  
 }  
}

public class RedBuff extends Buff{  
  
 @Override  
 public void operation() {  
 System.*out*.println("加上保护罩的");  
 component.operation();  
 }  
}

public class YellowBuff extends Buff{  
 @Override  
 public void operation() {  
 System.*out*.println("趟着水跑的");  
 component.operation();  
 }  
}

public class Main {  
 public static void main(String[] args) {  
  
 Sheep xi=new Sheep();  
  
  
 GreenBuff greenBuff = new GreenBuff();  
 greenBuff.setComponent(xi);  
  
 RedBuff redBuff = new RedBuff();  
 redBuff.setComponent(greenBuff);  
  
 YellowBuff yellowBuff=new YellowBuff();  
 yellowBuff.setComponent(redBuff);  
  
 yellowBuff.operation();  
  
 }  
}

**测试结果**

趟着水跑的

加上保护罩的

奔跑速度的

喜羊羊

5 要点总结

1. 动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。一般的，我们为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会膨胀。

实验八 外观模式的应用

1 实验目的

1) 掌握外观模式（Facade）的特点

2) 分析具体问题，使用外观模式进行设计。

2 实验内容和要求

一个保安系统的，由录像机、电灯、红外线监控和警报器组成。保安系统的操作人员需要经常将这些仪器启动和关闭。保安类需要用到所有的录像机(Camera)、电灯(Light)、感应器(Sensor)和警报器(Alarm)对象，保安觉得使用不方便。应用外观模式，用C#或Java控制台应用程序改进该设计。绘制该模式的UML图。

3 UML图

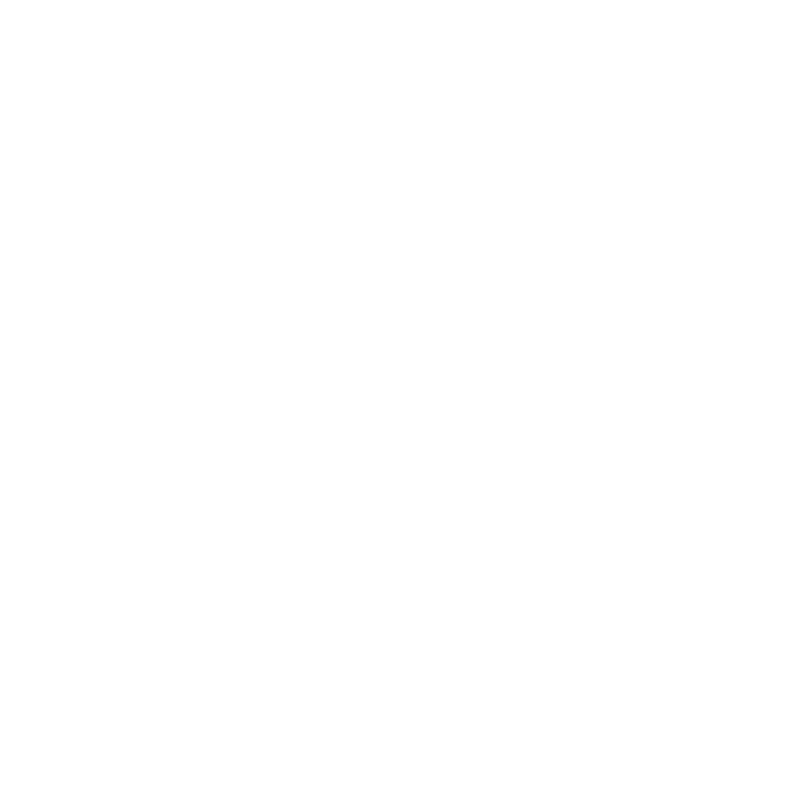


图8.3.1 类图

4 代码实现

abstract class Switch {  
 boolean status = false;  
  
 void on() {  
 status = true;  
 System.*out*.println("打开 " + getClass().getSimpleName());  
 }  
  
 void off() {  
 status = false;  
 System.*out*.println("关闭 " + getClass().getSimpleName());  
 }  
}

public class Alarm extends Switch {  
  
}

public class Camera extends Switch {  
  
}

public class Light extends Switch{  
  
}

public class Sensor extends Switch{  
  
}

public class Police {  
 private Alarm alarm;  
 private Camera camera;  
 private Light light;  
 private Sensor sensor;  
  
 public Police() {  
 alarm=new Alarm();  
 camera=new Camera();  
 light=new Light();  
 sensor=new Sensor();  
 }  
  
 public void turnOnAll(){  
 alarm.on();  
 camera.on();  
 light.on();  
 sensor.on();  
 }  
  
 public void turnOffAll(){  
 alarm.off();  
 camera.off();  
 light.off();  
 sensor.off();  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Police police=new Police();  
 police.turnOnAll();  
 }  
}

**测试结果**

打开 Alarm

打开 Camera

打开 Light

打开 Sensor

5 要点总结

1. 降低访问复杂系统的内部子系统时的复杂度，简化客户端之间的接口。
2. 为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。
3. 不涉及继承关系，是新建一个类来整合其它类之间复杂的相互依赖。