装饰者模式

2020年2月4日 12:01

1. 概念

装饰者模式又名包装(Wrapper)模式。装饰者模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案。装饰者模式动态地将责任附加到对象身上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

2. UML结构图

装饰者模式以对客户透明的方式动态地给一个对象附加上更多的责任。换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰者模式可以在不使用创造更多子类的情况下，将对象的功能加以扩展。

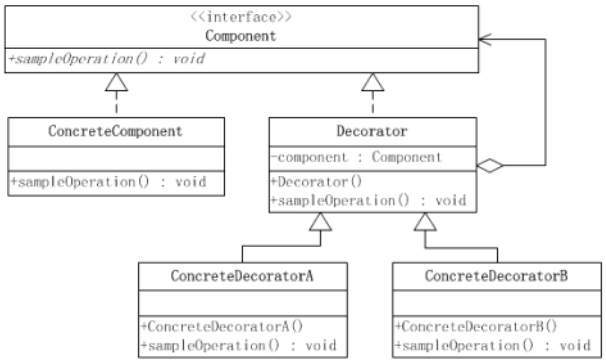

图2.1 装饰者模式结构图

在装饰模式中的角色有：

- 抽象构件(Component)角色：给出一个抽象接口，以规范准备接收附加责任的对象。
- 具体构件(ConcreteComponent)角色：定义一个将要接收附加责任的类。
- 装饰(Decorator)角色：持有一个构件(Component)对象的实例，并定义一个与抽象构件接口一致的接口。装饰角色可以为抽象接口。
- 具体装饰(ConcreteDecorator)角色：负责给构件对象“贴上”附加的责任。

代码说明如下：

1. 抽象构件角色

```
public interface Component {  // 可为接口/抽象类
   public void sampleOperation();
}
```

2. 具体构件角色

```
public class ConcreteComponent implements Component {
    @Override
    public void sampleOperation() {
        // 写相关的业务代码
    }
}
```

3. 装饰角色

```
public class Decorator implements Component{
    private Component component;

    public Decorator(Component component){
        this.component = component;
    }
    @Override
    public void sampleOperation() {
        // 委派给构件
        component.sampleOperation();
    }
}
```

4. 具体装饰角色

```
public class ConcreteDecoratorA extends Decorator {
    public ConcreteDecoratorA(Component component) {
        super(component);
    }

    @Override
    public void sampleOperation() {
      // 写相关的业务代码
     super.sampleOperation();
     // 写相关的业务代码
    }
}
```

3. 装饰者模式实例

老王有个煎饼摊，不同的用户有着不同的需求，有个要加蛋，有的要加火腿肠，使用装饰者模式设计该实例。

在本例中，煎饼是被装饰类，具体装饰类分别有鸡蛋和香肠

1. 煎饼抽象类 - 抽象构件角色

```
public abstract class ABattercake {
    public abstract String getDesc();
    public abstract int cost();
}
```

设置为抽象类便于扩展，可能有山东煎饼和河南煎饼，可以设置不同的具体构建角色。

2. 具体煎饼类 - 具体构件角色

```
public class Battercake extends ABattercake {
    @Override
    public String getDesc() {
        return "煎饼";
    }

    @Override
    public int cost() {
        return 8;
    }
}
```

3. 抽象装饰类 - 装饰角色

```
public class AbstractDecorator extends ABattercake  {
    // 与Battercake 需要被装饰的实体类建立联系
    ABattercake aBattercake;

    public AbstractDecorator(ABattercake aBattercake) {
        this.aBattercake = aBattercake;
    }

    @Override
    public String getDesc() {
        return aBattercake.getDesc();
    }

    @Override
    public int cost() {
        return aBattercake.cost();
    }
}
```

4. 具体装饰类 - 具体装饰角色

```
public class EggDecorator  extends AbstractDecorator{
    public EggDecorator(ABattercake aBattercake) {
        super(aBattercake);
    }

    @Override
    public String getDesc() {
        return super.getDesc() + "加一个鸡蛋，";
    }

    @Override
    public int cost() {
        return super.cost() + 2;
    }
}
```

```
public class SausageDecorator extends AbstractDecorator {
    public SausageDecorator(ABattercake aBattercake) {
        super(aBattercake);
    }

    @Override
    public String getDesc() {
        return super.getDesc() + "加一个香肠，";
    }

    @Override
    public int cost() {
        return super.cost() + 3;
    }
}
```

5. 测试类

```
@Test
    public void decoratorTestV2(){
        ABattercake aBattercake;
        aBattercake = new Battercake();  // 创建需要被装饰的实体类
        aBattercake = new EggDecorator(aBattercake);
        aBattercake = new EggDecorator(aBattercake);
        aBattercake = new SausageDecorator(aBattercake);

        System.out.println(aBattercake.getDesc() + "价格:" + aBattercake.cost());
    }
```

6. 结果

```
煎饼加一个鸡蛋，加一个鸡蛋，加一个香肠，价格:15
```

4. 一些变化

1. 装饰者模式简化

大多数情况下，装饰者模式的实现都要比上面给出的示意性例子要简单。如果只有一个ConcreteComponent类，那么可以考虑去掉抽象的Component类（接口），把Decorator作为一个ConcreteComponent子类。

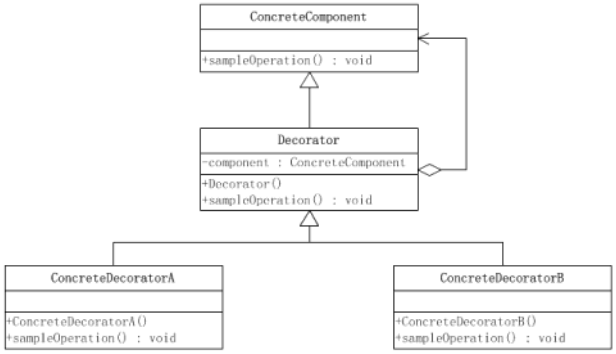

图4.1 简化装饰者模式

如果只有一个ConcreteDecorator类，那么就没有必要建立一个单独的Decorator类，而可以把Decorator和ConcreteDecorator的责任合并成一个类。甚至在只有两个ConcreteDecorator类的情况下，都可以这样做。如下图所示

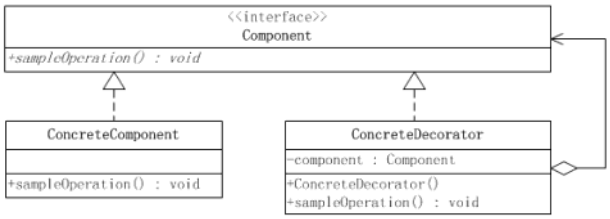

图4.2 简化装饰者模式

2. 透明性要求

装饰者模式对客户端的透明性要求程序不要声明一个ConcreteComponent类型的变量，而应当声明一个Component类型的变量。

下面的做法是对的：

```
ABattercake aBattercake  = new Battercake();
aBattercake  = new BattercakeWithEgg(aBattercake);
```

而下面的做法是不对的：

```
ABattercake aBattercake = new Battercake();
BattercakeWithEgg battercakeWithEgg = new BattercakeWithEgg(aBattercake);
```

3. 半透明性的装饰者模式

纯粹的装饰者模式很难找到。装饰者模式的用意是**在不改变接口的前提下，增强所考虑的类的性能**。在增强性能的时候，往往需要建立新的公开的方法。即便是在孙大圣的系统里，也需要新的方法。比如齐天大圣类并没有飞行的能力，而鸟儿有。这就意味着鸟儿应当有一个新的fly()方法。再比如，齐天大圣类并没有游泳的能力，而鱼儿有，这就意味着在鱼儿类里应当有一个新的swim()方法。

这就导致了大多数的装饰者模式的实现都是“半透明”的，而不是完全透明的。换言之，允许装饰者模式改变接口，增加新的方法。这意味着客户端可以声明ConcreteDecorator类型的变量，从而可以调用ConcreteDecorator类中才有的方法：

```
TheGreatestSage sage = new Monkey();
Bird bird = new Bird(sage);
bird.fly();
```

上述例子参考：https://www.jianshu.com/p/d7f20ae63186

半透明的装饰者模式是介于装饰者模式和适配器模式之间的。适配器模式的用意是改变所考虑的类的接口，也可以通过改写一个或几个方法，或增加新的方法来增强或改变所考虑的类的功能。大多数的装饰者模式实际上是半透明的装饰者模式，这样的装饰者模式也称做半装饰、半适配器模式。

5. 装饰者模式优缺点

1. 装饰模式的优点

（1）装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。装饰模式允许系统动态决定“贴上”一个需要的“装饰”，或者除掉一个不需要的“装饰”。继承关系则不同，继承关系是静态的，它在系统运行前就决定了。

（2）通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

2. 装饰模式的缺点

由于使用装饰模式，可以比使用继承关系需要较少数目的类。使用较少的类，当然使设计比较易于进行。但是，在另一方面，使用装饰模式会产生比使用继承关系更多的对象。更多的对象会使得查错变得困难，特别是这些对象看上去都很相像。

6. 装饰者模式和适配器模式

装饰者模式和适配器模式都是“包装模式(Wrapper Pattern)”，它们都是通过封装其他对象达到设计的目的的，但是它们的形态有很大区别。

理想的装饰者模式在对被装饰对象进行功能增强的同时，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。而适配器模式则不然，一般而言，适配器模式并不要求对源对象的功能进行增强，但是会改变源对象的接口，以便和目标接口相符合。

装饰者模式有透明和半透明两种，这两种的区别就在于装饰角色的接口与抽象构件角色的接口是否完全一致。透明的装饰者模式也就是理想的装饰者模式，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。相反，如果装饰角色的接口与抽象构件角色接口不一致，也就是说装饰角色的接口比抽象构件角色的接口宽的话，装饰角色实际上已经成了一个适配器角色，这种装饰者模式也是可以接受的，称为“半透明”的装饰模式，如下图所示。

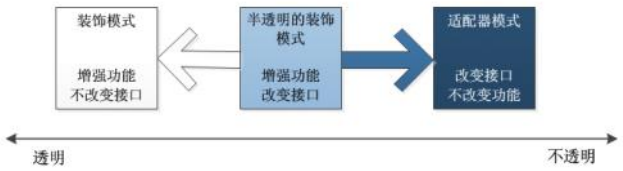

图6.1 关系图

在适配器模式里面，适配器类的接口通常会与目标类的接口重叠，但往往并不完全相同。换言之，适配器类的接口会比被装饰的目标类接口宽。

显然，半透明的装饰者模式实际上就是处于适配器模式与装饰者模式之间的灰色地带。如果将装饰者模式与适配器模式合并成为一个“包装模式”的话，那么半透明的装饰者模式倒可以成为这种合并后的“包装模式”的代表。

7. I/O流中装饰者实例

I/O流读取文件内容的简单操作示例。

```
public class IOTest {


    public static void main(String[] args) throws IOException {
        // 流式读取文件
        DataInputStream dis = null;
        try{
            dis = new DataInputStream(
                    new BufferedInputStream(
                            new FileInputStream("test.txt")
                    )
            );
            //读取文件内容
            byte[] bs = new byte[dis.available()];
            dis.read(bs);
            String content = new String(bs);
            System.out.println(content);
        }finally{
            dis.close();
        }
    }
}
```

观察上面的代码，会发现最里层是一个FileInputStream对象，然后把它传递给一个BufferedInputStream对象，经过BufferedInputStream处理，再把处理后的对象传递给了DataInputStream对象进行处理，这个过程其实就是装饰器的组装过程，FileInputStream对象相当于原始的被装饰的对象，而BufferedInputStream对象和DataInputStream对象则相当于装饰器。