转 JAVA设计模式（11）：结构型-装饰模式（Decorator）

尽管目前房价依旧很高，但还是阻止不了大家对新房的渴望和买房的热情。如果大家买的是毛坯房，无疑还有一项艰巨的任务要面对，那就是装修。对新房进行装修并没有改变房屋用于居住的本质，但它可以让房子变得更漂亮、更温馨、更实用、更能满足居家的需求。在软件设计中，我们也有一种类似新房装修的技术可以对已有对象（新房）的功能进行扩展（装修），以获得更加符合用户需求的对象，使得对象具有更加强大的功能。这种技术对应于一种被称之为装饰模式的设计模式，本章将介绍用于扩展系统功能的装饰模式。

1 图形界面构件库的设计

Sunny软件公司基于面向对象技术开发了一套图形界面构件库VisualComponent，该构件库提供了大量基本构件，如窗体、文本框、列表框等，由于在使用该构件库时，用户经常要求定制一些特效显示效果，如带滚动条的窗体、带黑色边框的文本框、既带滚动条又带黑色边框的列表框等等，因此经常需要对该构件库进行扩展以增强其功能，如图1所示：

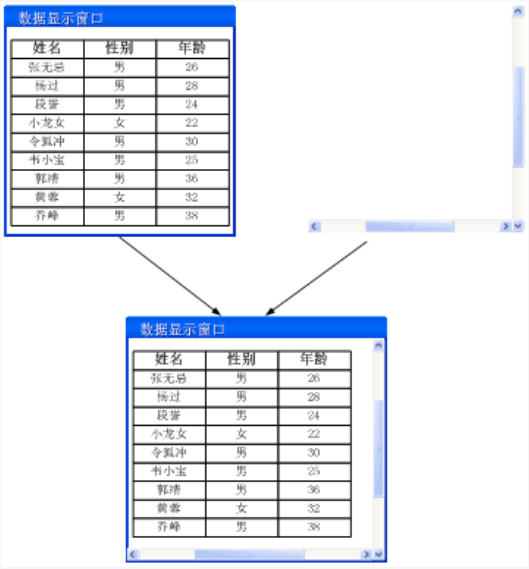

图1 带滚动条的窗体示意图

如何提高图形界面构件库性的可扩展性并降低其维护成本是Sunny公司开发人员必须面对的一个问题。

Sunny软件公司的开发人员针对上述要求，提出了一个基于继承复用的初始设计方案，其基本结构如图2所示：

最新评论

推荐文章

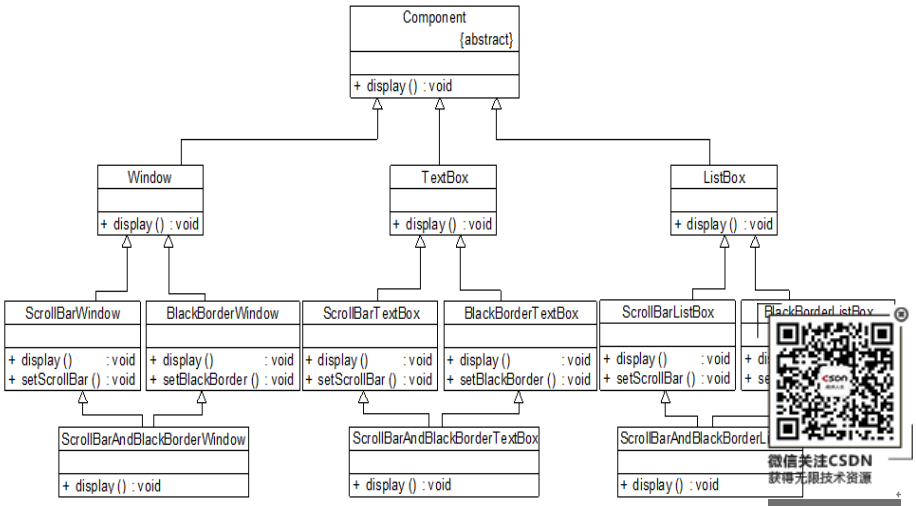

图2 图形界面构件库初始设计方案

图2中，在抽象类Component中声明了抽象方法display()，其子类Window、TextBox、ListBox等实现了display()方法，可以显示最简单的控件，再通过它们的子类来对功能进行扩展，例如，在Window的子类ScrollBarWindow、BlackBorderWindow中对Window中的display()方法进行了扩展，分别实现带滚动条和带黑色边框的窗体。仔细分析该设计方案，我们不难发现存在如下几个问题：

(1) **系统扩展麻烦，在某些编程语言中无法实现。**如果用户需要一个既带滚动条又带黑色边框的窗体，在图12-2中通过增加了一个新的类ScrollBarAndBlackBorderWindow来实现，该类既作为ScrollBarWindow的子类，又作为BlackBorderWindow的子类；但现在很多面向对象编程语言，如Java、C#等都不支持多重类继承，因此在这些语言中无法通过继承来实现对来自多个父类的方法的重用。此外，如果还需要扩展一项功能，例如增加一个透明窗体类TransparentWindow，它是Window类的子类，可以将一个窗体设置为透明窗体，现在需要一个同时拥有三项功能（带滚动条、带黑色边框、透明）的窗体，必须再增加一个类作为三个窗体类的子类，这同样在Java等语言中无法实现。系统在扩展时非常麻烦，有时候甚至无法实现。

(2)**代码重复。**从图2中我们可以看出，不只是窗体需要设置滚动条，文本框、列表框等都需要设置滚动条，因此在ScrollBarWindow、ScrollBarTextBox和ScrollBarListBox等类中都包含用于增加滚动条的方法setScrollBar()，该方法的具体实现过程基本相同，代码重复，不利于对系统进行修改和维护。

(3) **系统庞大，类的数目非常多。**如果增加新的控件或者新的扩展功能系统都需要增加大量的具体类，这将导致系统变得非常庞大。在图12-2中，3种基本控件和2种扩展方式需要定义9个具体类；如果再增加一个基本控件还需要增加3个具体类；增加一种扩展方式则需要增加更多的类，如果存在3种扩展方式，对于每一个控件而言，需要增加7个具体类，因为这3种扩展方式存在7种组合关系（大家自己分析为什么需要7个类？）。

总之，图12-2不是一个好的设计方案，怎么办？如何让系统中的类可以进行扩展但是又不会导致类数目的急剧增加？不用着急，让我们先来分析为什么这个设计方案会存在如此多的问题。**根本原因在于复用机制的不合理**，图2采用了继承复用，例如在ScrollBarWindow中需要复用Window类中定义的display()方法，同时又增加新的方法setScrollBar()，ScrollBarTextBox和ScrollBarListBox都必须做类似的处理，在复用父类的方法后再增加新的方法来扩展功能。根据“合成复用原则”，**在实现功能复用时，我们要多用关联，少用继承**，因此我们可以换个角度来考虑，将setScrollBar()方法抽取出来，封装在一个独立的类中，在这个类中定义一个Component类型的对象，通过调用Component的display()方法来显示最基本的构件，同时再通过setScrollBar()方法对基本构件的功能进行增强。由于Window、ListBox和TextBox都是Component的子类，根据“里氏代换原则”，程序在运行时，我们只要向这个独立的类中注入具体的Component子类的对象即可实现功能的扩展。这个独立的类一般称为装饰器(Decorator)或装饰类，顾名思义，它的作用就是对原有对象进行装饰，通过装饰来扩展原有对象的功能。

装饰类的引入将大大简化本系统的设计，它也是装饰模式的核心，下面让我们正式进入装饰模式的学习。

2 装饰模式概述

装饰模式可以在不改变一个对象本身功能的基础上给对象增加额外的新行为，在现实生活中，这种情况也到处存在，例如一张照片，我们可以不改变照片本身，给它增加一个相框，使

得它具有防潮的功能，而且用户可以根据需要给它增加不同类型的相框，甚至可以在一个小相框的外面再套一个大相框。

装饰模式是一种用于替代继承的技术，它通过一种无须定义子类的方式来给对象动态增加职责，使用对象之间的关联关系取代类之间的继承关系。在装饰模式中引入了装饰类，在装饰类中既可以调用待装饰的原有类的方法，还可以增加新的方法，以扩充原有类的功能。

装饰模式定义如下：

装饰模式(Decorator Pattern)：动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活。装饰模式是一种对象结构型模式。

在装饰模式中，为了让系统具有更好的灵活性和可扩展性，我们通常会定义一个抽象装饰类，而将具体的装饰类作为它的子类，装饰模式结构如图3所示：

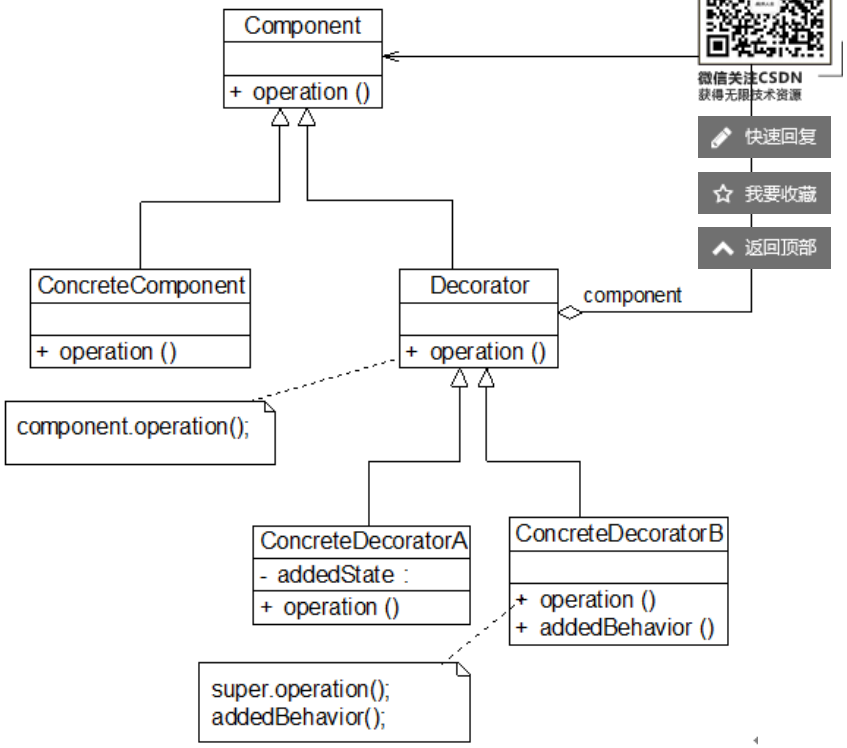

图3 装饰模式结构图

在装饰模式结构图中包含如下几个角色：

● Component（抽象构件）：它是具体构件和抽象装饰类的共同父类，声明了在具体构件中实现的业务方法，它的引入可以使客户端以一致的方式处理未被装饰的对象以及装饰之后的对象，实现客户端的透明操作。

● ConcreteComponent（具体构件）：它是抽象构件类的子类，用于定义具体的构件对象，实现了在抽象构件中声明的方法，装饰器可以给它增加额外的职责（方法）。

● Decorator（抽象装饰类）：它也是抽象构件类的子类，用于给具体构件增加职责，但是具体职责在其子类中实现。它维护一个指向抽象构件对象的引用，通过该引用可以调用装饰之前构件对象的方法，并通过其子类扩展该方法，以达到装饰的目的。

● ConcreteDecorator（具体装饰类）：它是抽象装饰类的子类，负责向构件添加新的职责。每一个具体装饰类都定义了一些新的行为，它可以调用在抽象装饰类中定义的方法，并可以增加新的方法用以扩充对象的行为。

由于具体构件类和装饰类都实现了相同的抽象构件接口，因此装饰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。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。

装饰模式的核心在于抽象装饰类的设计，其典型代码如下所示

```
[java] view plain copy print ?
01. public class Decorator implements Component{
02.         private Component component;  //维持一个对抽象构件对象的引用
03.         public Decorator(Component component) {
04.                 this.component=component;<span style="font-family: Arial;">//注入一个抽象构件类型的对象</span>
05.         }
06.
07.         public void operation(){
08.                 component.operation();  //调用原有业务方法
```

```
09.         }
10.     }
```

在抽象装饰类Decorator中定义了一个Component类型的对象component，维持一个对抽象构件对象的引用，并可以通过构造方法或Setter方法将一个Component类型的对象注入进来，同时由于Decorator类实现了抽象构件Component接口，因此需要实现在其中声明的业务方法operation()，需要注意的是在Decorator中并未真正实现operation()方法，而只是调用原有component对象的operation()方法，它没有真正实施装饰，而是提供一个统一的接口，将具体装饰过程交给子类完成。

在Decorator的子类即具体装饰类中将继承operation()方法并根据需要进行扩展，具体装饰类代码如下：

```
[java] view plain copy print ?
01. public class ConcreteDecorator extends Decorator{
02.         public ConcreteDecorator(Component component){
03.                 super(component);
04.         }
05.
06.         public void operation(){
07.                 super.operation();  //调用原有业务方法
08.                 addedBehavior();  //调用新增业务方法
09.         }
10.
11.        //新增业务方法
12.         public  void addedBehavior(){
13.            ……
14.         }
15. }
```


在具体装饰类中可以调用到抽象装饰类的operation()方法，同时可以定义新的业务方法，如addedBehavior()。

由于在抽象装饰类Decorator中注入的是Component类型的对象，因此我们可以将一个具体构件对象注入其中，再通过具体装饰类来进行装饰；此外，我们还可以将一个已经装饰过的Decorator子类的对象再注入其中进行多次装饰，从而对原有功能的多次扩展。

思考

能否在装饰模式中找出两个独立变化的维度？试比较装饰模式和桥接模式的相同之处和不同之处？

【作者：刘伟 http://blog.csdn.net/lovelion】

3 完整解决方案

为了让系统具有更好的灵活性和可扩展性，克服继承复用所带来的问题，Sunny公司开发人员使用装饰模式来重构图形界面构件库的设计，其中部分类的基本结构如图4所示：

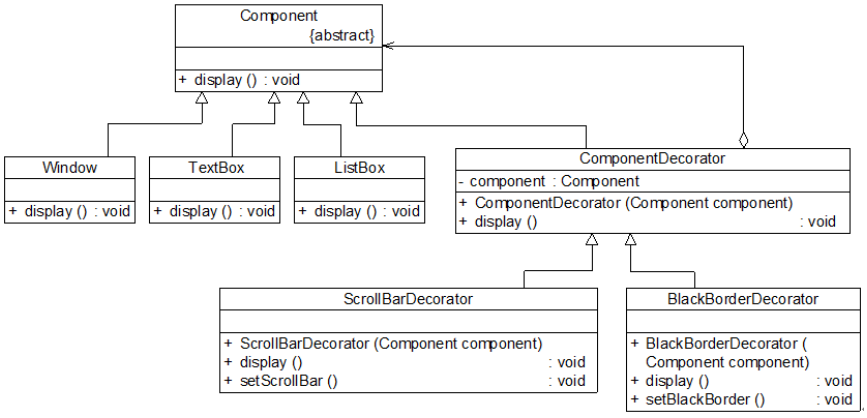

图4 图形界面构件库结构图

在图4中，Component充当抽象构件类，其子类Window、TextBox、ListBox充当具体构件类，Component类的另一个子类ComponentDecorator充当抽象装饰类，ComponentDecorator的子

类ScrollBarDecorator和BlackBorderDecorator充当具体装饰类。完整代码如下所示：

```java
//抽象界面构件类：抽象构件类，为了突出与模式相关的核心代码，对原有控件代码进行了大量的简化
public abstract class Component {
    public abstract void display();
}

//窗体类：具体构件类
class Window extends Component{
    public void display(){
        System.out.println("显示窗体！");
    }
}

//文本框类：具体构件类
class TextBox extends Component{
    public void display(){
        System.out.println("显示文本框！");
    }
}

//列表框类：具体构件类
class ListBox extends Component{
    public void display(){
        System.out.println("显示列表框！");
    }
}
```



```java
// 构件装饰类：抽象装饰类
public class ComponentDecorator extends Component {
    private Component component; // 维持对抽象构件类型对象的引用
    // 注入抽象构件类型的对象
    public ComponentDecorator(Component component) {
        this.component = component;
    }

    public void display() {
        component.display();
    }
}

// 滚动条装饰类：具体装饰类
class ScrollBarDecorator extends ComponentDecorator {
    public ScrollBarDecorator(Component component) {
        super(component);
    }

    public void display() {
        this.setScrollBar();
        super.display();
    }

    public void setScrollBar() {
        System.out.println("为构件增加滚动条！");
    }
}

//黑色边框装饰类：具体装饰类
class BlackBorderDecorator extends ComponentDecorator {
    public BlackBorderDecorator(Component component) {
        super(component);
    }

    public void display() {
        this.setBlackBorder();
        super.display();
    }

    public void setBlackBorder() {
        System.out.println("为构件增加黑色边框！");
    }
}
```

编写如下客户端测试代码：

```
[java] view plain copy print ?
01. public class Client{
02.
03.     public static void main(String[] args){
04.         Component component,componentSB;  //使用抽象构件定义
05.         component = new Window(); //定义具体构件
06.         componentSB = new  ScrollBarDecorator(component); //定义装饰后的构件
07.         componentSB.display();
08.     }
09.
10. }
```


编译并运行程序，输出结果如下：

```
为构件增加滚动条！
显示窗体！
```

在客户端代码中，我们先定义了一个Window类型的具体构件对象component，然后将component作为构造函数的参数注入到具体装饰类ScrollBarDecorator中，得到一个装饰之后对象componentSB，再调用componentSB的display()方法后将得到一个有滚动条的窗体。如果我们希望得到一个既有滚动条又有黑色边框的窗体，不需要对原有类库进行任何修改，只需将客户端代码修改为如下所示：

```
[java] view plain copy print ?
01. public class Client{
02.       public  static void main(String args[]){
03.             Component  component,componentSB,componentBB; //全部使用抽象构件定义
04.             component = new Window();
05.             componentSB = new  ScrollBarDecorator(component);
06.             componentBB = new  BlackBorderDecorator(componentSB); //将装饰了一次之后的对象继续注入到另一个装饰类中，进行第二次装饰
07.             componentBB.display();
08.       }
09. }
```

编译并运行程序，输出结果如下：

```
为构件增加黑色边框！
为构件增加滚动条！
显示窗体！
```

我们可以将装饰了一次之后的componentSB对象注入另一个装饰类BlackBorderDecorator中实现第二次装饰，得到一个经过两次装饰的对象componentBB，再调用componentBB的display()方法即可得到一个既有滚动条又有黑色边框的窗体。

如果需要在原有系统中增加一个新的具体构件类或者新的具体装饰类，无须修改现有类库代码，只需将它们分别作为抽象构件类或者抽象装饰类的子类即可。与图12-2所示的继承结构相比，使用装饰模式之后将大大减少了子类的个数，让系统扩展起来更加方便，而且更容易维护，是取代继承复用的有效方式之一。

4 透明装饰模式与半透明装饰模式

装饰模式虽好，但存在一个问题。如果客户端希望单独调用具体装饰类新增的方法，而不想通过抽象构件中声明的方法来调用新增方法时将遇到一些麻烦，我们通过一个实例来对这种情况加以说明：

在Sunny软件公司开发的Sunny OA系统中，采购单(PurchaseRequest)和请假条(LeaveRequest)等

文件(Document)对象都具有显示功能，现在要为其增加审批、删除等功能，使用装饰模式进行设计。

我们使用装饰模式可以得到如图5所示结构图：

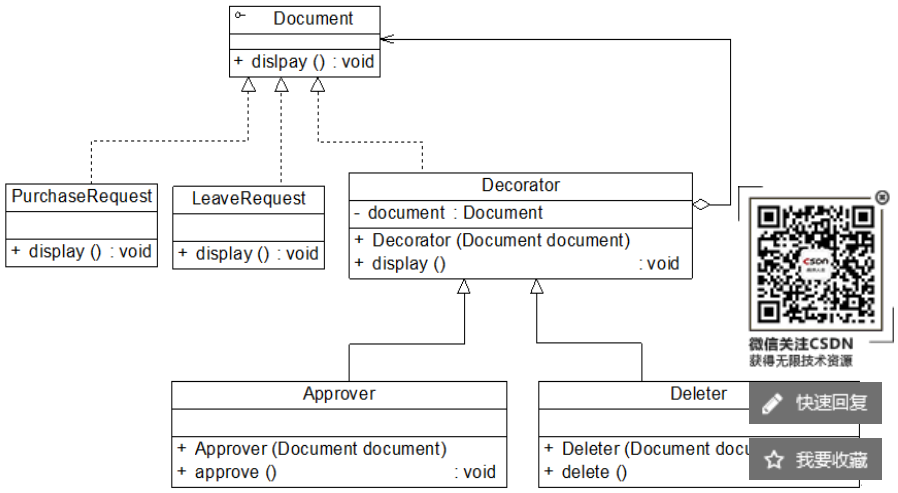

图5文件对象功能增加实例结构图

在图5中，Document充当抽象构件类，PurchaseRequest和LeaveRequest充当具体构件类，Decorator充当抽象装饰类，Approver和Deleter充当具体装饰类。其中Decorator类和Approver类的示例代码如下所示：

```java
//抽象装饰类
public class Decorator implements  Document{
	private Document  document;
	public Decorator(Document  document) {
		this.  document = document;
	}
	public void display() {
		document.display();
	}
}

//具体装饰类
class Approver extends  Decorator{
	public Approver(Document document){
		super(document);
		System.out.println("增加审批功能！");
	}
	public void approve(){
		System.out.println("审批文件！");
	}
}
```

大家注意，Approver类继承了抽象装饰类Decorator的display()方法，同时新增了业务方法approve()，但这两个方法是独立的，没有任何调用关系。如果客户端需要分别调用这两个方法，代码片段如下所示：

```java
Document  doc; //使用抽象构件类型定义
doc = new PurchaseRequest();
Approver newDoc; //使用具体装饰类型定义
newDoc = new Approver(doc);
newDoc.display();//调用原有业务方法
newDoc.approve();//调用新增业务方法
```

如果newDoc也使用Document类型来定义，将导致客户端无法调用新增业务方法approve()，因为在抽象构件类Document中没有对approve()方法的声明。也就是说，在客户端无法统一对待装饰之前的具体构件对象和装饰之后的构件对象。

在实际使用过程中，由于新增行为可能需要单独调用，因此这种形式的装饰模式也经常出现，这种装饰模式被称为**半透明(Semi-transparent)装饰模式**，而标准的装饰模式是**透明**

关闭

(Transparent)装饰模式。下面我们对这两种装饰模式进行较为详细的介绍：

(1)透明装饰模式

在透明装饰模式中，要求客户端完全针对抽象编程，装饰模式的透明性要求客户端程序不应该将对象声明为具体构件类型或具体装饰类型，而应该全部声明为抽象构件类型。对于客户端而言，具体构件对象和具体装饰对象没有任何区别。也就是应该使用如下代码：

```
[java] view plain copy print ?
01. Component c, c1; //使用抽象构件类型定义对象
02. c = new ConcreteComponent();
03. c1 = new ConcreteDecorator (c);
```

而不应该使用如下代码：

```
[java] view plain copy print ?
01. ConcreteComponent c; //使用具体构件类型定义对象
02. c = new ConcreteComponent();
```

或

```
[java] view plain copy print ?
01. ConcreteDecorator c1; //使用具体装饰类型定义对象
02. c1 = new ConcreteDecorator(c);
```

在3节图形界面构件库的设计方案中使用的就是透明装饰模式，在客户端中存在如下代码片段：

```
[java] view plain copy print ?
01. ……
02. Component component,componentSB,componentBB; //全部使用抽象构件定义
03. component = new Window();
04. componentSB = new ScrollBarDecorator(component);
05. componentBB = new BlackBorderDecorator(componentSB);
06. componentBB.display();
07. ……
```

使用抽象构件类型Component定义全部具体构件对象和具体装饰对象，客户端可以一致地使用这些对象，因此符合透明装饰模式的要求。

透明装饰模式可以让客户端透明地使用装饰之前的对象和装饰之后的对象，无须关心它们的区别，此外，还可以对一个已装饰过的对象进行多次装饰，得到更为复杂、功能更为强大的对象。在实现透明装饰模式时，要求具体装饰类的operation()方法覆盖抽象装饰类的operation()方法，除了调用原有对象的operation()外还需要调用新增的addedBehavior()方法来增加新行为，

(2)半透明装饰模式

透明装饰模式的设计难度较大，而且有时我们需要单独调用新增的业务方法。为了能够调用到新增方法，我们不得不用具体装饰类型来定义装饰之后的对象，而具体构件类型还是可以使用抽象构件类型来定义，这种装饰模式即为半透明装饰模式，也就是说，对于客户端而言，具体构件类型无须关心，是透明的；但是具体装饰类型必须指定，这是不透明的。如本节前面所提到的文件对象功能增加实例，为了能够调用到在Approver中新增方法approve()，客户端代码片段如下所示：

```
[java] view plain copy print ?
01. ……
02. Document doc; //使用抽象构件类型定义
03. doc = new PurchaseRequest();
04. Approver newDoc; //使用具体装饰类型定义
05. newDoc = new Approver(doc);
06. ……
```

半透明装饰模式可以给系统带来更多的灵活性，设计相对简单，使用起来也非常方便；但是其最大的缺点在于不能实现对同一个对象的多次装饰，而且客户端需要有区别地对待装饰之前的对象和装饰之后的对象。在实现半透明的装饰模式时，我们只需在具体装饰类中增加一个独立的addedBehavior()方法来封装相应的业务处理，由于客户端使用具体装饰类型来定义装饰后的对象，因此可以单独调用addedBehavior()方法来扩展系统功能。

思考

为什么半透明装饰模式不能实现对同一个对象的多次装饰？

5 装饰模式注意事项

在使用装饰模式时，通常我们需要注意以下几个问题：

(1) 尽量保持装饰类的接口与被装饰类的接口相同，这样，对于客户端而言，无论是装饰之前的对象还是装饰之后的对象都可以一致对待。这也就是说，在可能的情况下，我们应该尽量使用透明装饰模式。

(2) 尽量保持具体构件类ConcreteComponent是一个“轻”类，也就是说不要把太多的行为放在具体构件类中，我们可以通过装饰类对其进行扩展。

(3) 如果只有一个具体构件类，那么抽象装饰类可以作为该具体构件类的直接子类。如图6所示：

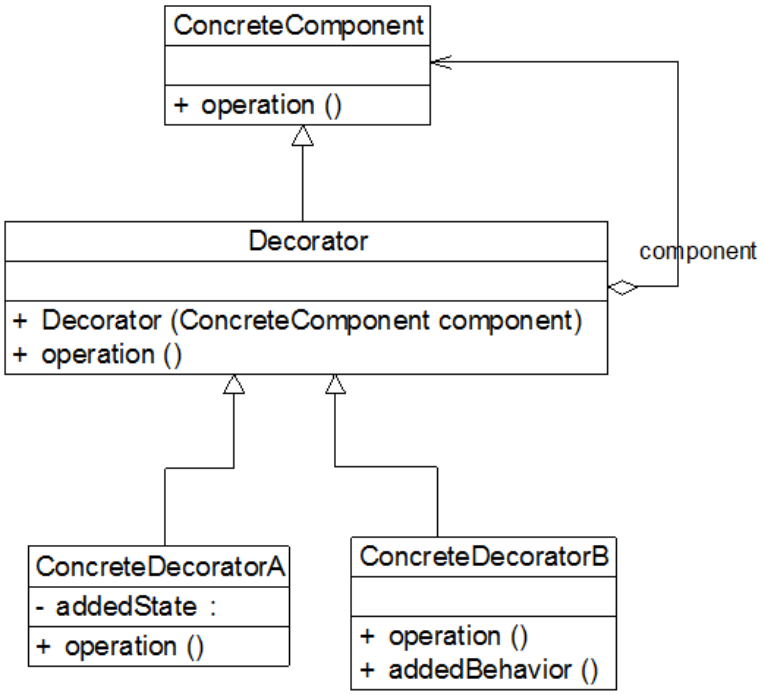

图6 没有抽象构件类的装饰模式

6 装饰模式总结

装饰模式降低了系统的耦合度，可以动态增加或删除对象的职责，并使得需要装饰的具体构件类和具体装饰类可以独立变化，以便增加新的具体构件类和具体装饰类。在软件开发中，装饰模式应用较为广泛，例如在JavaIO中的输入流和输出流的设计、javax.swing包中一些图形界面构件功能的增强等地方都运用了装饰模式。

1.主要优点

装饰模式的主要优点如下：

(1) 对于扩展一个对象的功能，装饰模式比继承更加灵活性，不会导致类的个数急剧增加。

(2) 可以通过一种动态的方式来扩展一个对象的功能，通过配置文件可以在运行时选择不同的具体装饰类，从而实现不同的行为。

(3) 可以对一个对象进行多次装饰，通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合，得到功能更为强大的对象。

(4) 具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，原有类库代码无须改变，符合“开闭原则”。

2.主要缺点

装饰模式的主要缺点如下：

(1) 使用装饰模式进行系统设计时将产生很多小对象，这些对象的区别在于它们之间相互连接的方式有所不同，而不是它们的类或者属性值有所不同，大量小对象的产生势必会占用更多的系统资源，在一定程序上影响程序的性能。

(2) 装饰模式提供了一种比继承更加灵活机动的解决方案，但同时也意味着比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为繁琐。

3.适用场景

在以下情况下可以考虑使用装饰模式：

(1) 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

(2) 当不能采用继承的方式对系统进行扩展或者采用继承不利于系统扩展和维护[illegible]装饰模式。不能采用继承的情况主要有两类：第一类是系统中存在大量独立的扩[illegible]种扩展或者扩展之间的组合将产生大量的子类，使得子类数目呈爆炸性增长；[illegible]类已定义为不能被继承（如Java语言中的final类）。

练习

Sunny软件公司欲开发了一个数据加密模块，可以对字符串进行加密。最简单的加密算法通过对字母进行移位来实现，同时还提供了稍复杂的逆向输出加密，[illegible]提供了更为高级的求模加密。用户先使用最简单的加密算法对字符串进行加密，如果觉得还不够可以对加密之后的结果使用其他加密算法进行二次加密，当然也可以进行第三次加密。试使用装饰模式设计该多重加密系统。

【作者：刘伟 http://blog.csdn.net/lovelion】

- 上一篇 jsp中EL表达式
- 下一篇 JAVA设计模式（12）：结构型-门面模式（Facade）

相关文章推荐

- 扩展系统功能——装饰模式（四）
- 携程机票大数据基础平台架构演进-- 许鹏
- 扩展系统功能——装饰模式（四）：透明与半透明...
- Python可以这样学--董付国
- JAVA设计模式（09）：结构型-代理模式（Proxy）
- 一步一步学Spring Boot
- JAVA设计模式（06）：结构型-适配器模式（Ada...
- 深入浅出C++程序设计
- 不兼容结构的协调——适配器模式（一）
- Android Material Design 新控件
- JAVA设计模式（01）：创建型-工厂模式【简单工...
- 机器学习需要用到的数学知识
- JAVA设计模式（09）：结构型-代理模式（Proxy）
- JAVA设计模式（05）：创建型-原型模式（Protot...
- JAVA设计模式（16）：行为型-策略模式（Strate...
- Java设计模式菜鸟系列(八)适配器模式建模与实现

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场