# 装饰模式简介

## 概述

1. 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，对对象的功能进行扩展，使得对象具有更加强大的功能。
2. 在软件设计中，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。
3. 在装饰模式中引入了装饰类，在装饰类中皆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。
4. 装饰模式（Decorator Pattern）的定义如下：动态地给一个对象增加一些额外的职责。就扩展功能而言，装饰模式提供了一种比使用子类更加灵活的替代方案。
5. 装饰模式是一种对象结构型模式，它以对客户端透明的方式动态地给一个对象附加上更多的责任，可以在不需要创建更多子类的情况下让对象的功能得以扩展。

## 结构

装饰模式结构图如下：



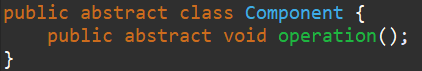
装饰模式包含以下4个角色：

1. Component（抽象构件类）：它是具体构件类和抽象装饰类的共同父类，一般定义为抽象类或接口，声明了在具体构件中要实现的业务方法，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。
2. ConcreteComponent（具体构件类）：它是抽象构件类的子类，用于定义具体的构件对象，实现了在抽象构件类中声明的方法，装饰类可以给它增加额外的职责（方法）。
3. Decorator（抽象装饰类）：它也是抽象构件类的子类，一般定义为抽象类或具体类，用于给具体构件增加职责，但是具体的职责要在其子类（具体装饰类）中实现；它维护一个抽象构件类类型的对象，通过该对象可以调用装饰之前构件对象的方法，通过其子类（具体装饰类）扩展该方法，已达到装饰的目的。
4. 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于扩充。

# 实现

## 原理

1. 在装饰模式中，抽象构件类一般设计为抽象类或者接口，在其中声明抽象业务方法，当然也可以在抽象构件类中定义一些所有具有构件都共有的业务方法。抽象构件类的典型代码如下：



1. 具体构件类作为抽象构件的子类实现了在抽象构件中声明的业务方法，通常在具体构件类中只提供基本功能的实现，一些复杂的功能需要通过装饰类进行扩展。其典型代码如下：



1. 装饰模式的核心在于抽象装饰类的设计，其典型代码如下：



在抽象装饰类Decorator中定义了一个抽象构件类类型的对象，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方法将一个抽象构件类类型的对象注入进来。

同时由于Decorator类继承了抽象构件类Component类，因此需要实现在其中声明的业务方法operation ( )，值得注意的是，在Decorator中并未真正实现operation ( )方法，而是通过抽象构件类对象component调用原有的operation ( )方法，它并没有真正实施装饰，而是提供一个统一的接口，将具体装饰过程交给子类（具体装饰类）完成。

1. 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，典型的具体装饰类代码如下：



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，例如addedBehavior ( )，如果该方法不希望客户端调用或者只在该具体装饰类内部调用，可以将其访问权限设为私有。

由于在抽象装饰类Decorator中注入的是Component类型的对象，因此可以将一个具体构件对象注入其中，再通过具体装饰类进行装饰；此外，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，从而实现对原有功能的多次扩展。

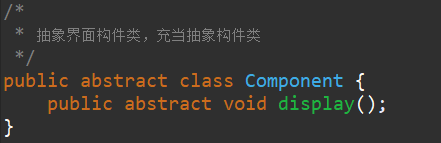
## 实例

实例说明：某软件公司基于面向对象技术开发了一套图像界面构件库，该构件库提供了大量的基本构件，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殊的显示效果，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，因此需要对该构件库进行扩展以增强其功能。

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。为了突出装饰模式的核心代码，本实例中对控件代码进行了大量的简化。其结构图如下：



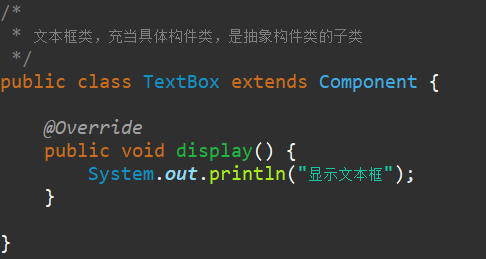
1. Component：抽象界面构件类，充当抽象构件类。



1. Window：窗体类，充当具体构件类。



1. TextBox：文本框类，充当具体构件类。



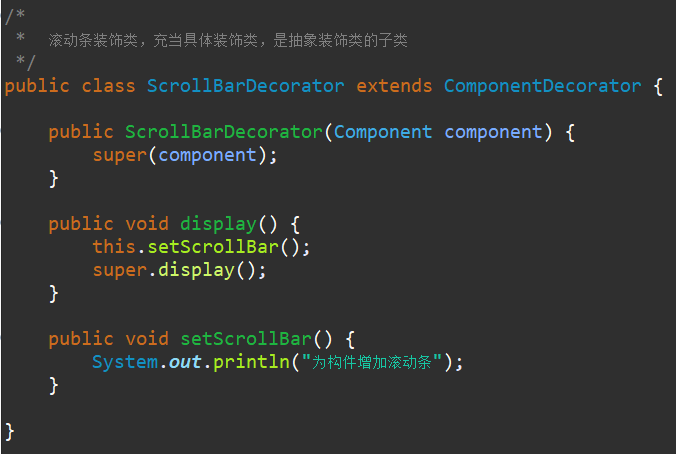
1. ListBox：列表框类，充当具体构件类。



1. ComponentDecorator：构件装饰类，充当抽象装饰类。



1. ScrollBarDecorator：滚动条装饰类，充当具体装饰类。



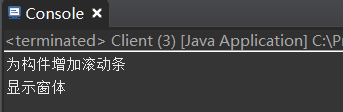
1. BlackBorderDecorator：黑色边框装饰类，充当具体装饰类。



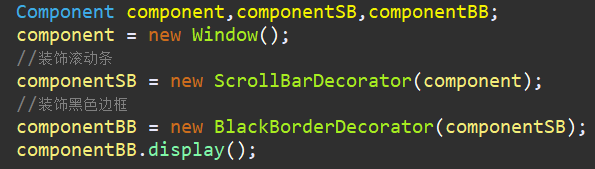
1. Client：客户端测试类，测试装饰模式。



运行结果如下：



如果既要装饰滚动条和黑色边框，则可以：



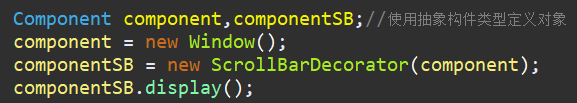
如果需要在原有系统中增加一个新的具体构件类或者新的具体装饰类，无须修改现有类库代码，只需将它们分别作为抽象构件或者抽象装饰类的子类即可。

# 透明装饰模式与半透明装饰模式

在装饰模式中，具体装饰类通过新增成员变量或成员方法来扩充具体构件类的功能。在标准的装饰模式中，新增行为需要在原有业务方法中调用，无论是具体构件对象还是装饰过的构件对象，对于客户端而言都是透明的，这种装饰模式被称为透明（Transparent）装饰模式。但在某些情况下，有些新增行为可能需要单独被调用，此时客户端不能再一致地处理装饰之前的对象和装饰之后的对象，这种装饰模式被称为半透明（Semi-transparent）装饰模式。

## 透明装饰模式

1. 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。也就是应该使用以下代码：



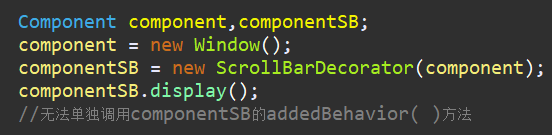
而不应该使用以下代码：



或



2）在透明装饰模式中，客户端中无法单独调用具体装饰类中扩展的业务方法。如下：



因为在抽象构件中并没有声明这些方法。

1. 透明装饰模式可以让客户端透明地使用装饰之前和装饰之后的对象，无须关系它们的区别。

## 半透明装饰模式

透明装饰模式的设计难度较大，而且有时客户端需要单独调用新增的业务方法。为了能够调用到新增的方法，不得不用具体装饰类型来定义装饰之后的对象，而具体构件类型仍然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。客户端代码片段如下：



半透明装饰模式可以给系统带来更多的灵活性，但是其最大的缺点在于不能实现对同一对象的多次修饰，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只需在具体装饰类中增加相应的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，因此可以单独调用这些方法。

# 优缺点及适用环境

1. 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。
2. 适用装饰模式将大大减少子类的个数，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。
3. 在软件开发中装饰模式的应用较为广泛，例如在Java IO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。

## 优点

装饰模式的优点主要如下：

1. 对于扩展一个对象的功能，装饰模式比继承更加灵活，不会调至类的个数急剧增加。
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，得到功能更加强大的对象。
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，原有类库代码无需改变，符合开闭原则。

## 缺点

装饰模式的缺点主要如下：

1. 在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互相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，在一定程度上影响程序的性能。
2. 装饰模式提供了一种比继承更灵活、机动的解决方案，但同时也意味着比继承更加易于出错，排错也更困难，对于多次装饰的对象，在调试时寻找错误可能需要逐级排查，较为繁琐。

## 适用环境

在以下情况下可以考虑适用装饰模式：

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时可以使用装饰模式。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展，为支持每一种扩展或者扩展之间的组合将产生大量的子类，使得子类数量呈爆炸性增长；第二类是因为类已定义为不能被继承（例如在Java语言中使用final关键字修饰的类）。

## 小结

1. 装饰模式用于动态地给一个对象增加一些额外的职责。就扩展功能而言，装饰模式提供了一种比使用子类更加灵活的替代方案，它是一种对象结构性模式。
2. 装饰模式包含抽象构件、具体构件、抽象装饰类和具体装饰类4个角色。其中，抽象构件时具体构件类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；具体构件实现了在抽象构件中声明的方法，装饰类可以给它增加额外的职责（方法）；抽象装饰类用于给具体装饰类增加职责，但是具体职责在其子类中实现；具体装饰类负责向构件添加新的职责。
3. 装饰模式的优点主要是在扩展功能时比继承更加灵活，不会导致类的个数急剧正价；它通过一种动态的方式来扩展一个对象的功能，可以对一个对象进行多次装饰，还通过使用不同的具体装饰类以及这些装饰类的排列组合创造出很多不同行为的组合，得到功能更加强大的对象；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。其缺点主要是使用装饰模式进行系统设计时将产生很多小对象；此外，装饰模式比继承更加易于错误，排错也更加困难，对于多次装饰的对象，在调试时寻找错误可能需要逐级排查，较为繁琐。
4. 装饰模式适用于以下环境：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；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时也可以使用装饰模式。
5. 装饰模式分为透明装饰模式和半透明装饰模式：在透明装饰模式中要求客户端完全针对抽象构件类进行编程，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；半透明装饰模式允许用户在客户端声明具体装饰类型的对象，从而可以单独调用在具体装饰者中新增的方法，但不能实现对同一对象的多次修饰。