# 1 单例模式（Singleton）

# 2 装饰模式（Decorator）

装饰模式和适配器模式都属于包装模式（Wrapper）

<https://www.cnblogs.com/java-my-life/archive/2012/04/20/2455726.html>

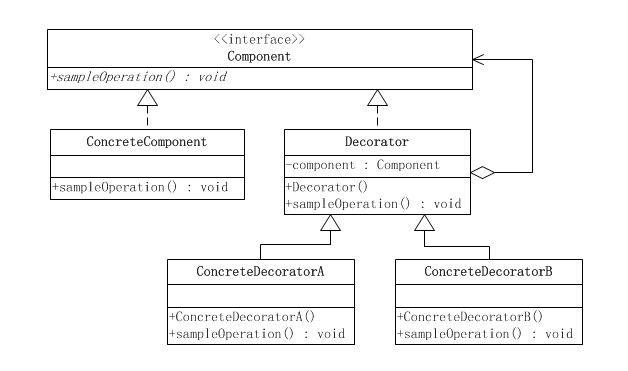
对客户透明的方式动态的**给一个对象附加上更多的责任**。装饰模式的典型应用是javaI/O流。

如：BufferReader:

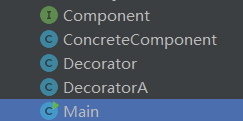
String str = "abcdefghijklmnopqrstuvwxyz";  
BufferedReader reader = new BufferedReader(new StringReader(str));  
reader.readLine();

BufferReader和StringReader都实现了Reader接口，这里BufferReader为装饰者，StringReader为被装饰者，Reader为两者都实现的接口。

BufferReader为StringReader所实现的Reader接口的各个方法进行了**包装**，对使用者而言，使用者看似调用的Reader接口是BufferReader实现的，然而实际调用的是StringReader实现的，BufferReader**间接**调用StringReader的方法，并**附加了更多的责任**（BufferReader包装了更多的功能）。



1. 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。
2. 具体构件(**ConcreteComponent**)角色：定义一个将要接收附加责任的类。
3. 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。
4. 具体装饰(**ConcreteDecorator**)角色：负责给构件对象“贴上”附加的责任。



// 使用装饰  
 Component c = new DecoratorA(new ConcreteComponent());  
 // 不使用装饰  
 Component cc = new ConcreteComponent();  
 // 违反客户端透明性要求  
// ConcreteComponent ccc = new ConcreteComponent();  
 //使用装饰及更多额外功能  
 Decorator d = new DecoratorA(new ConcreteComponent());  
 //使用装饰及更多额外功能,“半透明”的装饰模式  
 DecoratorA dd = new DecoratorA(new ConcreteComponent());  
 d.sameOperation();  
 System.*out*.println();  
 dd.sameOperation();  
 dd.a(); // 半透明  
 System.*out*.println();  
 cc.sameOperation();  
 System.*out*.println();  
 c.sameOperation();

**控制台输出：**

实际功能

抽象装饰

具体装饰

实际功能

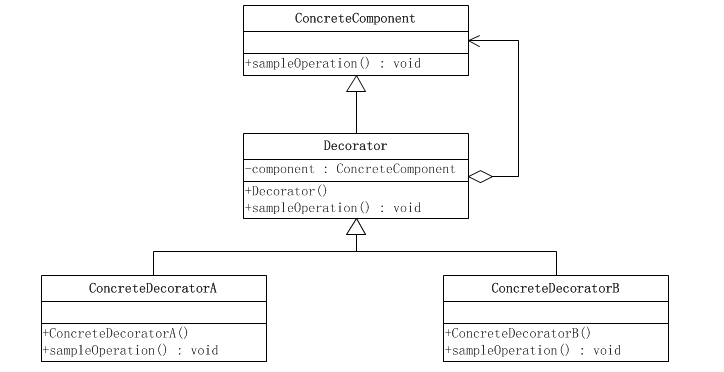
实际功能

抽象装饰

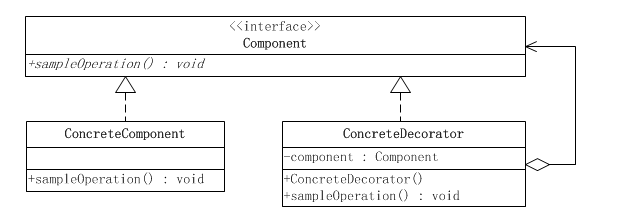
具体装饰

## 装饰模式的简化

省去抽象构件



省去抽象装饰



## 透明性的要求

装饰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。

## 半透明的装饰模式（半装饰、半适配器模式。）

**装饰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。**

大多数的装饰模式的实现都是“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。**

## 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。

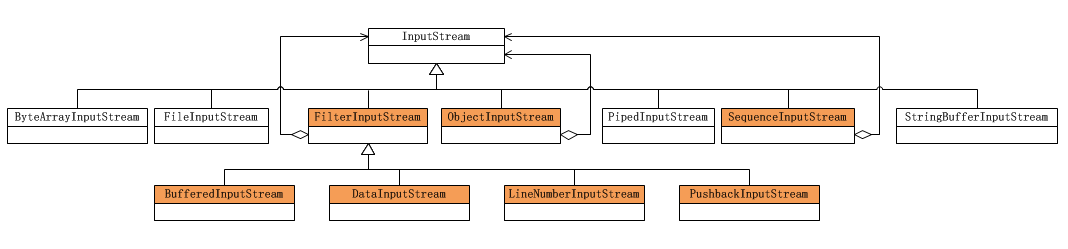
## 装饰模式的缺点

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。

## 在JAVA I/O库中的应用

1. 为什么使用装饰模式

由于Java I/O库需要**很多性能的各种组合**，如果这些性能都是用继承的方法实现的，那么每一种组合都需要一个类，这样就会造成大量性能重复的类出现。而如果采用装饰模式，那么类的数目就会大大减少，**性能的重复**也可以减至最少。因此装饰模式是Java I/O库的基本模式。



根据上图可以看出：

**抽象构件(**Component)角色：由InputStream扮演。这是一个抽象类，为各种子类型提供统一的接口。

**具体构件**(ConcreteComponent)角色：由ByteArrayInputStream、FileInputStream、PipedInputStream、StringBufferInputStream等类扮演。它们实现了抽象构件角色所规定的接口。

**抽象装饰**(Decorator)角色：由FilterInputStream扮演。它实现了InputStream所规定的接口。

**具体装饰**(ConcreteDecorator)角色：由几个类扮演，分别是BufferedInputStream、DataInputStream以及两个不常用到的类LineNumberInputStream、PushbackInputStream。

# 3 外观模式（Facade）