## 一句话概括：

**在不改变对象结构的情况下向一个现有对象添加新的功能**。

关键点：不改变现有，加新的功能

## 补充介绍：

**装饰器模式（Decorator Pattern）**允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

就增加功能来说，装饰器模式相比生成子类更为灵活。该模式以对客户端透明的方式扩展对象的功能。

**比喻：**

假如我有一个蛋糕，如果在上面加上奶油其他什么都不加，那么它就成了一个奶油蛋糕。如果再加上草莓，那就是草莓奶油蛋糕，如果再加上一块巧克力板，上面写上姓名，然后插上蜡烛，就变成了一块生日蛋糕。

不论是蛋糕，奶油蛋糕，草莓蛋糕还是生日蛋糕，它们的核心都是蛋糕。不过，在加上一系列装饰之后，它变得更加甜美了，目的也更加明确了。装饰器模式中的被装饰对象和蛋糕很相似。

**装饰器模式和代理模式的区别：**

**装饰器模式关注于在一个对象上动态的添加方法**，然而**代理模式关注于控制对对象的访问**。换句话 说，用代理模式，代理类（proxy class）可以对它的客户隐藏一个对象的具体信息。因此，当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。并且，当我们使用装饰器模 式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。

## 参与角色:

1）抽象基类（也可以是接口）拥有被装饰对象的方法

2）抽象基类的实现类（被装饰对象）

3）装饰类（构造方法以被装饰对象为参数，增加了新的功能，实现了被装饰对象一样的抽象基类）

## 优点：

装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

## 缺点：

多层装饰比较复杂。

## 使用案例或场景：

1) 比如Java里面的基本数据类型int, boolean, char…… 都有他们对应的装饰类Integer, Boolean, Character……

2) 在Java IO中，具体构件角色是节点流，装饰角色是过滤流。

FilterInputStream和FilterOutputStream是装饰角色，而其他派生自它们的类则是具体装饰角色。

DataoutputStream out=new DataoutputStream(new FileoutputStream());

这就是 装饰者模式，DataoutputStream是装饰者子类，FileoutputStream是实现接口的子类。

这里不会调用到装饰者类--FilteroutputStream,只是作为继承的另一种方案，对客户端来说是透明的，是为了功能的扩张。

## 示例程序

需要源码的朋友可以前往github下载：

<https://github.com/aharddreamer/chendong/tree/master/design-patterns/demo-code/design-patterns>

程序简介：

在这段示例程序中，观察者将观察一个会生成数值的对象，并将它生成的数值结果显示出来。不过，不同的观察者显示的方式不一样。DigitObserver会以数字的形式显示数值，而GraphObserver则会以简单的图示形式来呈现数值。

示例程序类/接口一览：

Display 用于显示字符串的抽象类

StringDisplay 用于显示单行字符串的类

Border 用于显示装饰边框的抽象类

SideBorder 用于显示左右边框的类

FullBorder 用于显示上下边框的类

DecoratorPatternTest 用于测试程序的类

代码：

**public abstract class** Display {  
 **public abstract int** getColumns();  
 **public abstract int** getRows();  
 **public abstract** String getRowText(**int** row);  
 **public final void** show() {  
 **for** (**int** i = 0; i < getRows(); i++) {  
 System.***out***.println(getRowText(i));  
 }  
 }  
}

**public class** StringDisplay **extends** Display {  
 **private** String **string**;  
 **public** StringDisplay(String string) {  
 **this**.**string** = string;  
 }  
 @Override  
 **public int** getColumns() {  
 **return string**.getBytes().**length**;  
 }  
 @Override  
 **public int** getRows() {  
 **return** 1;  
 }  
 @Override  
 **public** String getRowText(**int** row) {  
 **if** (row == 0) {  
 **return string**;  
 } **else** {  
 **return null**;  
 }  
 }  
}

**public abstract class** Border **extends** Display {  
 **protected** Display **display**; *//表示被装饰物* **protected** Border(Display display) { *//在生成实例时通过参数指定被装饰物* **this**.**display** = display;  
 }  
}

**public class** SideBorder **extends** Border {  
 **private char borderChar**;  
 **public** SideBorder(Display display, **char** ch) {  
 **super**(display);  
 **this**.**borderChar** = ch;  
 }  
 @Override  
 **public int** getColumns() {  
 **return** 1 + **display**.getColumns() + 1;  
 }  
 @Override  
 **public int** getRows() {  
 **return display**.getRows();  
 }  
 @Override  
 **public** String getRowText(**int** row) {  
 **return borderChar** + **display**.getRowText(row) + **borderChar**;  
 }  
}

**public class** FullBorder **extends** Border {  
 **public** FullBorder(Display display) {  
 **super**(display);  
 }  
 @Override  
 **public int** getColumns() {  
 **return** 1 + **display**.getColumns() + 1; *//加上两边字符数* }  
 @Override  
 **public int** getRows() {  
 **return** 1 + **display**.getRows() + 1; *//上下行数* }  
 @Override  
 **public** String getRowText(**int** row) {  
 **if** (row == 0) {*//打印上边框* **return "+"** + makeLine(**'-'**, **display**.getColumns()) + **"+"**;  
 }**else if** (row == **display**.getRows() + 1) { *//打印下边框* **return "+"** + makeLine(**'-'**, **display**.getColumns()) + **"+"**;  
 }**else** { *//打印中间字符串段* **return "|"** + **display**.getRowText(row - 1) + **"|"**;  
 }  
 }  
 **private** String makeLine(**char** ch, **int** count) {  
 StringBuffer stringBuffer = **new** StringBuffer();  
 **for** (**int** i = 0; i < count ; i++) {  
 stringBuffer.append(ch);  
 }  
 **return** stringBuffer.toString();  
 }  
}

**public class** DecoratorPatternTest {  
 **public static void** main(String[] args) {  
 Display coreValue = **new** StringDisplay(**"Decorator Pattern"**);  
 Display sideDecorate = **new** SideBorder(coreValue, **'#'**);  
 Display fullDecorate = **new** FullBorder(coreValue);  
 System.***out***.println(**"显示原始数据（被装饰对象）:"**);  
 coreValue.show();  
  
 System.***out***.println(**"显示两边被加工过的数据:"**);  
 sideDecorate.show();  
  
 System.***out***.println(**"显示上下左右都加上边框的数据:"**);  
 fullDecorate.show();  
  
 Display complexDecorate = **new** SideBorder(  
 **new** FullBorder(  
 **new** SideBorder(  
 **new** FullBorder(  
 **new** StringDisplay(**"Complex Decorator"**)  
 ), **'#'** )  
 ), **'\*'** );  
 System.***out***.println(**"显示多层嵌套的复杂装饰数据:"**);  
 complexDecorate.show();  
 }  
}

**运行结果：**



参考：

《图解设计模式》【日】结城浩著

《装饰器模式》菜鸟教程网站

《装饰器模式 – IO流案例》https://www.cnblogs.com/toov5/p/9874556.html