# 装饰模式

装饰（Decorator）模式：动态给一个对象添加一些额外的职责，就好像给一个物体加上装饰物，完善其功能。装饰器模式提供了比继承更有弹性的替代方案来扩展功能。

## 一、示例

用c++实现用户界面装饰器，假设有一个名为VisualComponent的组件类。

class VisualComponent {

public:

VisualComponent();

virtual void Draw();

virtual void Resize();

// ...

};

定义了一个VisualComponent的子类Decorator，将继承它来获得不同的装饰。

class Decorator : public VisualComponent {

public:

Decorator(VisualComponent\*);

virtual void Draw();

virtual void Resize();

// ...

private:

VisualComponent\* \_component;

};

装饰器装饰由\_component实例变量引用的VisualComponent，该变量在构造函数中初始化。对于VisualComponent接口中的每个操作，装饰器都定义了一个默认的实现，它将请求传递给\_component:

void Decorator::Draw () {

\_component->Draw();

}

void Decorator::Resize () {

\_component->Resize();

}

Decorator的子类定义了特定的装饰。例如，类BorderDecorator为其外层组件添加边框。BorderDecorator是Decorator的子类，覆盖了绘制边框的操作。BorderDecorator还定义了一个私有的DrawBorder辅助操作来绘图。子类继承了Decorator的所有其他操作实现。

class BorderDecorator : public Decorator {

public:

BorderDecorator(VisualComponent\*, int borderWidth);

virtual void Draw();

private:

void DrawBorder(int);

private:

int \_width;

};

void BorderDecorator::Draw () {

Decorator::Draw();

DrawBorder(\_width);

}

类似的实现还会出现在ScrollDecorator和DropShadowDecorator中，它们会为可视组件添加滚动和投影功能。现在可以组合这些类的实例来提供不同的修饰。下面的代码演示了如何使用装饰器创建一个带边框的可滚动TextView。

首先，需要一种将可视化组件放入window对象的方法。假设Window类为此提供了SetContents操作:

void Window::SetContents (VisualComponent\* contents) {

// ...

}

现在可以创建文本视图和一个放文本的窗口:

Window\* window = new Window;

TextView\* textView = new TextView;

TextView是一个VisualComponent，可以把它放到窗口中:

window->SetContents(textView);

在把它放在窗户上之前相应地装饰它。

window->SetContents(

new BorderDecorator(new ScrollDecorator(textView), 1

)

);

因为Window通过VisualComponent接口访问它的内容，所以它不知道装饰器的存在。作为客户端仍然可以跟踪文本视图，如果必须直接与它交互，例如，当需要调用不属于VisualComponent接口的操作时，依赖于组件标识的客户端也应该直接引用它。

## 现实例子

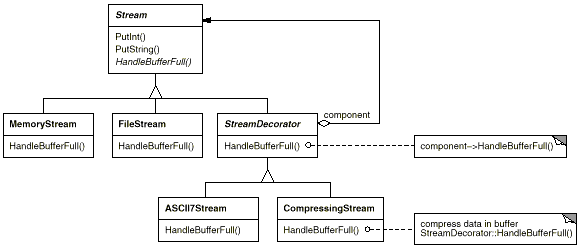
许多面向对象的用户界面工具包使用装饰器为部件添加图形装饰。示例包括interview [LVC98, LCI+92]， et++ [WGM88]和ObjectWorks\Smalltalk类库[Par90]。更多的Decorator的应用包括来自InterViews的DebuggingGlyph和来自ParcPlace Smalltalk的PassivityWrapper。DebuggingGlyph在向其组件转发布局请求之前和之后打印出调试信息。此跟踪信息可用于分析和调试复杂组合中对象的布局行为。PassivityWrapper可以启用或禁用与组件的用户交互。但是装饰器模式并不局限于图形用户界面，如下面的例子(基于et++流类[WGM88])所示。

流是大多数I/O工具的基本抽象。流可以提供一个接口，将对象转换为字节或字符序列。这可以将对象转录为文件或内存中的字符串，以便稍后检索。一种简单的方法是定义一个抽象的Stream类，包含MemoryStream和FileStream的子类。但假设还希望能够执行以下操作:

1.使用不同的压缩算法(游程编码、Lempel-Ziv等)压缩流数据。

2.将流数据压缩为7位ASCII字符，以便它可以在ASCII通信信道上传输。

装饰器模式提供了一种方式将这些职责添加到流中。下图展示了该问题的一种解决方案:



抽象类Stream维护了一个内部缓冲区，并提供了将数据存储到流中的操作(PutInt, PutString)。只要缓冲区满了，Stream就会调用抽象操作HandleBufferFull，由它来进行实际的数据传输。此操作的FileStream版本覆盖了将缓冲区传输到文件的操作。

这里的关键类是StreamDecorator，它维护一个对组件流的引用，并将请求转发给它。StreamDecorator的子类覆盖HandleBufferFull，并在调用StreamDecorator的HandleBufferFull操作之前执行其他操作。

例如，CompressingStream子类压缩数据，ASCII7Stream将数据转换为7位ASCII。现在，为了创建一个压缩数据并将压缩后的二进制数据转换为7位ASCII的FileStream，用CompressingStream和ASCII7Stream装饰一个FileStream:

Stream\* aStream = new CompressingStream(

new ASCII7Stream(

new FileStream("aFileName")

)

);

aStream->PutInt(12);

aStream->PutString("aString");

## 特点

1. 比静态继承更灵活。与静态(多)继承相比，装饰器模式提供了一种更灵活的方式来为对象添加职责。使用装饰器，可以在运行时简单地通过附加和分离职责来添加和删除职责。相比之下，继承需要为每个额外的功能创建一个新类。这产生了许多类，并增加了系统的复杂性。此外，为特定的组件类提供不同的装饰器类可以混合和匹配职责。装饰器还可以轻松地将一个属性添加两次。
2. 避免在层次结构的高层使用功能丰富的类。装饰器提供了一种随用随付的方法来添加职责。可以定义一个简单的类，并使用装饰器对象增量地添加功能，而不是试图在一个复杂的、可自定义的类中支持所有可预见的特性。功能可以由简单的部分组成。因此，应用程序不需要为它不使用的功能负责。也很容易独立于它们扩展的对象类来定义新类型的装饰器，即使是无法预料的扩展。
3. 装饰器和它的组件是不同的。装饰器充当透明的外壳。但是从对象标识的角度来看，被装饰的组件与组件本身并不相同。因此使用装饰器时，不应该依赖对象标识。
4. 使用装饰器的设计通常会导致系统由许多看起来相似的小对象组成。对象之间的区别仅在于它们之间的联系，而不是它们所属的类或变量的值。虽然这些系统很容易被理解它们的人定制，但它们可能很难学习和调试。

## 与相似模式的区别

1.适配器:装饰器与适配器的不同之处在于，装饰器只改变对象的职责，而不是它的接口;适配器将为对象提供一个全新的接口。

2.组合:装饰器可以被视为只有一个组件的退化组合。然而，装饰器有额外的职责——它不是用来聚合对象的。

3.策略:装饰器允许更改对象的外观;策略让你改变本质。