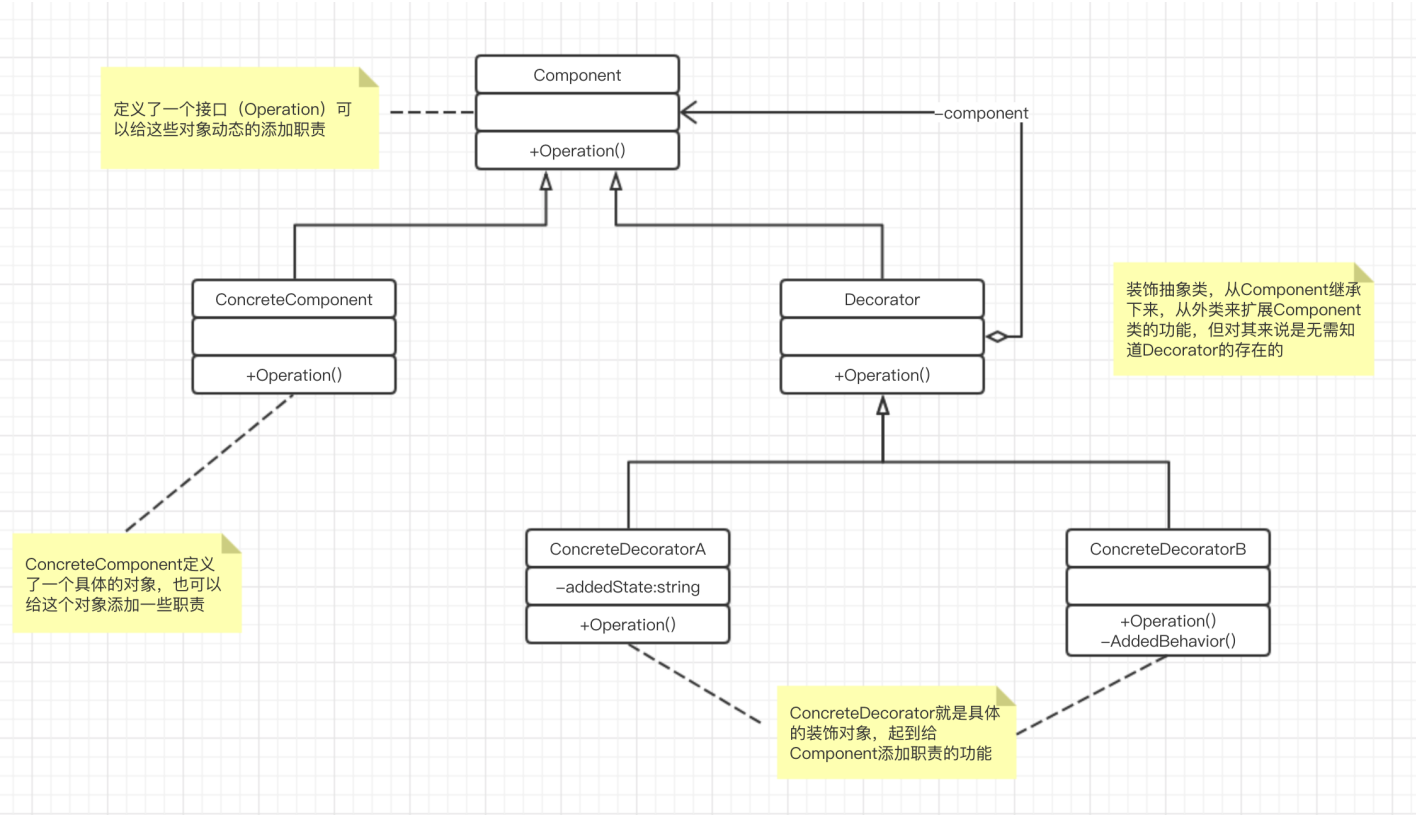
装饰模式

`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。

- `装饰模式`，动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更灵活。
- `装饰模式`是为已有功能动态地添加更多功能的一种方式。
- 当系统需要新功能的时候，向旧的类中添加新的代码，这些新加的代码通常装饰了原有类的核心职责或主要行为。而这些新加入的东西仅仅是为了满足一些只在特定情况下才会执行的特殊行为的需要，而`装饰模式`却提供了一个非常好的解决方案，它吧每个要装饰的功能放在单独的类中，并让这个类包装它索要装饰的对象，因此，当需要执行特殊行为的时候，客户代码就可以在运行时根据需要有选择地、按
  顺序地使用装饰功能包装对象了。
- `装饰模式`的优点：有效地把类的核心职责和装饰功能区分开，而且可以去除相关类中重复的装饰逻辑。




```
//
//  T3_20190114.hpp
//  DesignPattern
//
//  Created by shadot on 2019/1/14.
//  Copyright © 2019 shadot. All rights reserved.
//
```

```cpp

//装饰模式

#ifndef T3_20190114_hpp
#define T3_20190114_hpp

#include <stdio.h>
using namespace std;

class Component
{
public:
    Component(){}

    virtual void Operation() = 0;
};

class ConcreteComponent : public Component
{
public:
    ConcreteComponent():Component(){}

    virtual void Operation()
    {
        cout << "具体对象的操作！" << endl;
    }
};

class Decorator : public Component
{
public:
    Decorator():Component(){}

    void SetComponent(Component* component)
    {
        m_component = component;
    }

    void Operation() override //重写Operation(),实际执行的是Component的Operation()
    {
        if (m_component)
            m_component->Operation();
    }

private:
    Component* m_component;
```

```
};

class ConcreteDecoratorA : public Decorator
{
public:
    ConcreteDecoratorA():Decorator(){
        m_strAddState = "具体装饰对象A的操作1!";
    }

    void Operation() override
    {
        Decorator::Operation();//调用父类的同名方法
        cout << m_strAddState << endl;
        cout << "具体装饰对象A的操作2! " << endl;
    }

private:
    string m_strAddState;
};

class ConcreteDecoratorB : public Decorator
{
public:
    ConcreteDecoratorB():Decorator(){}

    void Operation() override
    {
        Decorator::Operation();
        AddedBehavior();
    }

private:
    void AddedBehavior()
    {
        cout << "具体装饰对象B的操作" << endl;
    }
};

#endif /* T3_20190114_hpp */

int main(int argc, const char * argv[]) {

    //T3_20190114
    {
        ConcreteComponent* c = new ConcreteComponent();
        ConcreteDecoratorA* a = new ConcreteDecoratorA();
```

```
        ConcreteDecoratorB* b = new ConcreteDecoratorB();

        a->SetComponent(c);
        b->SetComponent(a);
        b->Operation();

    }

    return 0;
}

//输出
具体对象的操作!
具体装饰对象A的操作1!
具体装饰对象A的操作2!
具体装饰对象B的操作


```