软件构造实验报告四

实验名称：装饰模式编程实现

实验时间：2019.4.14

学号：E21614061

姓名：徐奕

所在院系：计算机科学与技术学院

所在专业：软件工程

【实验目的和要求】

a）熟悉并理解装饰模式的原理与方法

b）熟练掌握装饰模式的代码与方法

【实验原理】

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为灵活。

动机

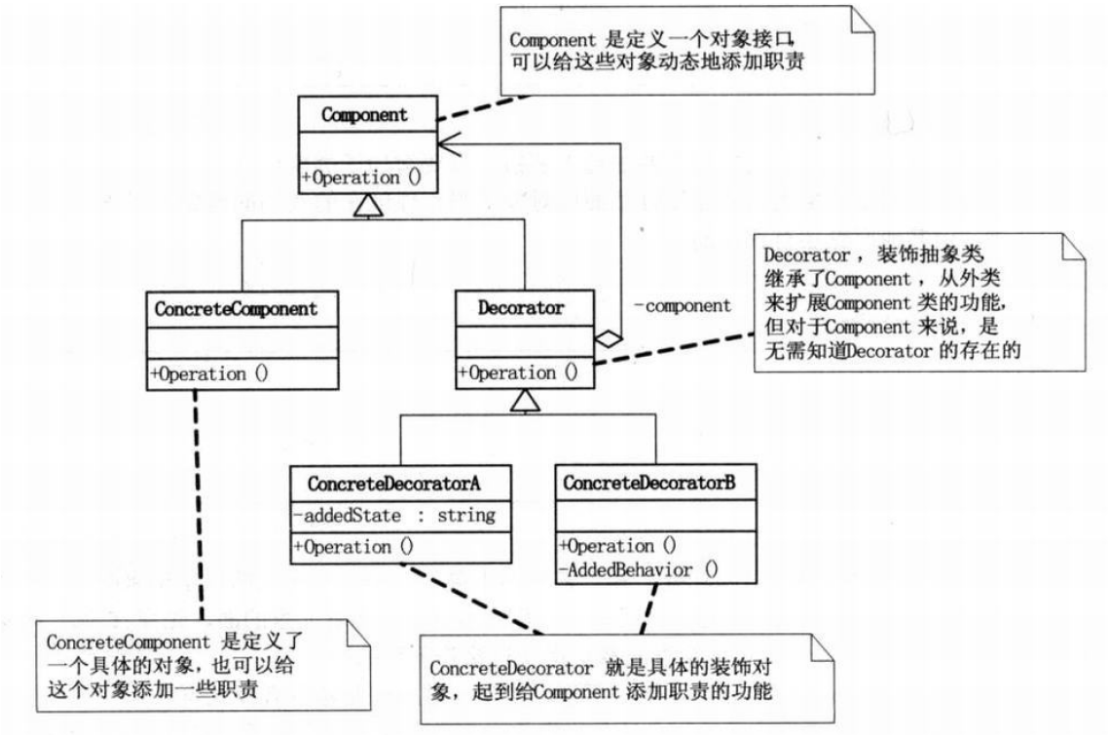
- 有时我们希望给某个对象而不是整个类添加一些功能。
- 使用继承机制是添加功能的一种有效途径，但不够灵活，用户不能控制对组件添加功能的方式和时机。
- 一种较为灵活的方式是将组件嵌入另一个对象中，由这个对象添加功能，我们称这个嵌入的对象为装饰。
- 这个装饰与它所装饰的组件接口一致，因此它对使用该组件的客户透明。

适用性

- 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。
- 处理那些可以撤销的职责。
- 当不能采用生成子类的方法进行扩充时。
    - 可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。

- 类定义被隐藏，或类定义不能用于生成子类。

结构



装饰模式的参与者

- Component
  - 定义一个对象接口，可以给这些对象动态地添加职责。
- ConcreteComponent
  - 定义一个对象，可以给这个对象添加一些职责。
- Decorator
  - 维持一个指向 Component 对象的指针，并定义一个与 Component 接口一致的接口。
- ConcreteDecorator
  - 向组件添加职责。

装饰模式的协作

- Decorator 将请求转发给它的 Component 对象，并有可能在转发请求前后执行一些附加动作。

效果

- 比静态继承更灵活
- 避免在层次结构高层的类有太多的特征
- Decorator 与它的 Component 不一样
- 有许多小对象

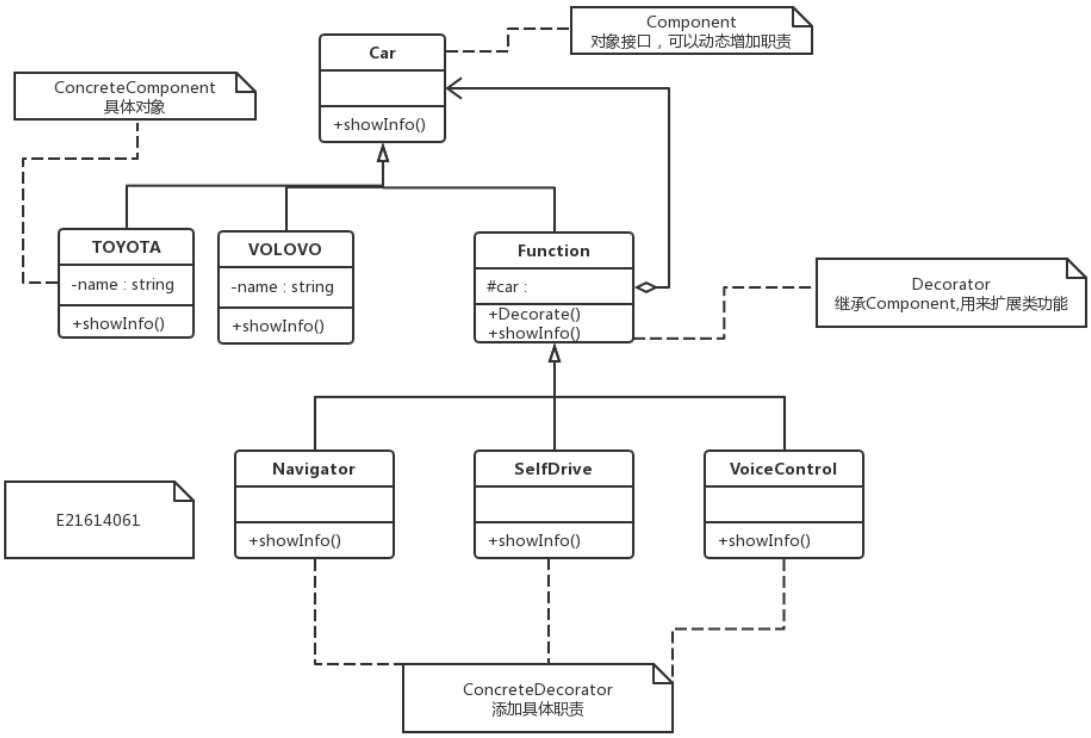
注意点

- 如果只有一个 ConcreteComponent 类而没有 Component 类，那么 Decorator 类可以是 ConcreteComponent 的一个子类。
- 同理，如果只有一个 ConcreteDecorator 类，那么就没有必要建立一个单独的 Decorator 类，而可以把 Decorator 和 ConcreteDecorator 合并成一个类。

【实验内容】

给定两种初始的汽车类，例如丰田和沃尔沃，利用装饰模式分别给它们添加新的功能，其中丰田可以导航和自动驾驶，沃尔沃可以导航和语音控制。

【实验 UML 图】



【实验代码与函数】

在本次实验中：

Component：**Car** 类

ConcreteComponent：**TOYOTA**、**VOLOVO** 类

Decorator：**Function** 类

ConcreteDecorator：**Navigator**、**SelfDrive**、**VoiceControl** 类

代码：

```
#include<iostream>
using namespace std;

/*
给定两种初始的汽车类，例如丰田和沃尔沃，
利用装饰模式分别给它们添加新的功能，
其中丰田可以导航和自动驾驶，沃尔沃可以导航和语音控制。
*/

class Car{//Component
public:
    virtual void showInfo() = 0;
};

class TOYOTA: public Car{
public:
    TOYOTA(){}
    TOYOTA(string name){
        this->name = name;
    }
    void showInfo(){
        cout<<name<<endl;
    }
private:
    string name;
};

class  VOLOVO: public Car{
public:
    VOLOVO(){}
    VOLOVO(string name){
        this->name = name;
    }
    void showInfo(){
        cout<<name<<endl;
    }
private:
    string name;
};

class Function: public Car{//Decorator
```

```
protected:
    Car* car;
public:
    void Decorate(Car* car){
        this->car = car;
    }
    void showInfo(){
        if(car != NULL)car->showInfo();
    }
};

//其中丰田可以导航和自动驾驶，沃尔沃可以导航和语音控制。

class Navigator: public Function{
public:
    void showInfo(){
        cout<<"导航 ";
        Function::showInfo();
    }
};

class SelfDrive: public Function{
public:
    void showInfo(){
        cout<<"自动驾驶 ";
        Function::showInfo();
    }
};

class VoiceControl: public Function{
public:
    void showInfo(){
        cout<<"语音控制 ";
        Function::showInfo();
    }
};


//其中丰田可以导航和自动驾驶，沃尔沃可以导航和语音控制。
int main(){
    Car* toyota = new TOYOTA("丰田卡罗拉");
    Navigator* na = new Navigator();
    SelfDrive* sd = new SelfDrive();
    na->Decorate(toyota);
    sd->Decorate(na);
```

```
    sd->showInfo();
    cout<<endl;
    Car* volovo = new VOLOVO("沃尔沃S90");
    Navigator* na2 = new Navigator();
    VoiceControl* vc = new VoiceControl();
    na2->Decorate(volovo);
    vc->Decorate(na2);
    vc->showInfo();

    return 0;
}
```

【实验结果】

分两个部分，上面是丰田汽车，可以导航和自动驾驶

下面是沃尔沃，可以导航和语言控制

```
XYs-MacBook-Pro:软件构造exp4 reacubeth$ cd "/Users/reacubeth/D
 && "/Users/reacubeth/Desktop/软件构造exp4/"decorator
自动驾驶 导航 丰田卡罗拉

语音控制 导航 沃尔沃S90
```

【实验总结】

①本次实验通过 Car 的实例掌握并编码了装饰模式。

②注意如果只有一个 ConcreteComponent 类而没有 Component 类，那么 Decorator 类可以是 ConcreteComponent 的一个子类。在本次实验中，**有 TOYOTA 和 VOLOVO 两个具体的类**，因此需要一个 **Component 的 Car 类来定义接口**。

③由②可知，如果只有一个 ConcreteDecorator 类，那么就没有必要建立一个单独的 Decorator 类，而可以把 Decorator 和 ConcreteDecorator 合并成一个类。在本次实验中有 Navigator、SelfDrive、VoiceControl 三个具体的装饰类。

④在客户端中，要最后一个添加的职责，用来是包装好了的车对象，因此需要由最后一个装饰的实例显示结果。