



深圳大学  
Shenzhen University

# 第12-1讲

## 装饰模式&外观模式

软件体系结构与设计模式

Software Architecture & Design Pattern

深圳大学计算机与软件学院



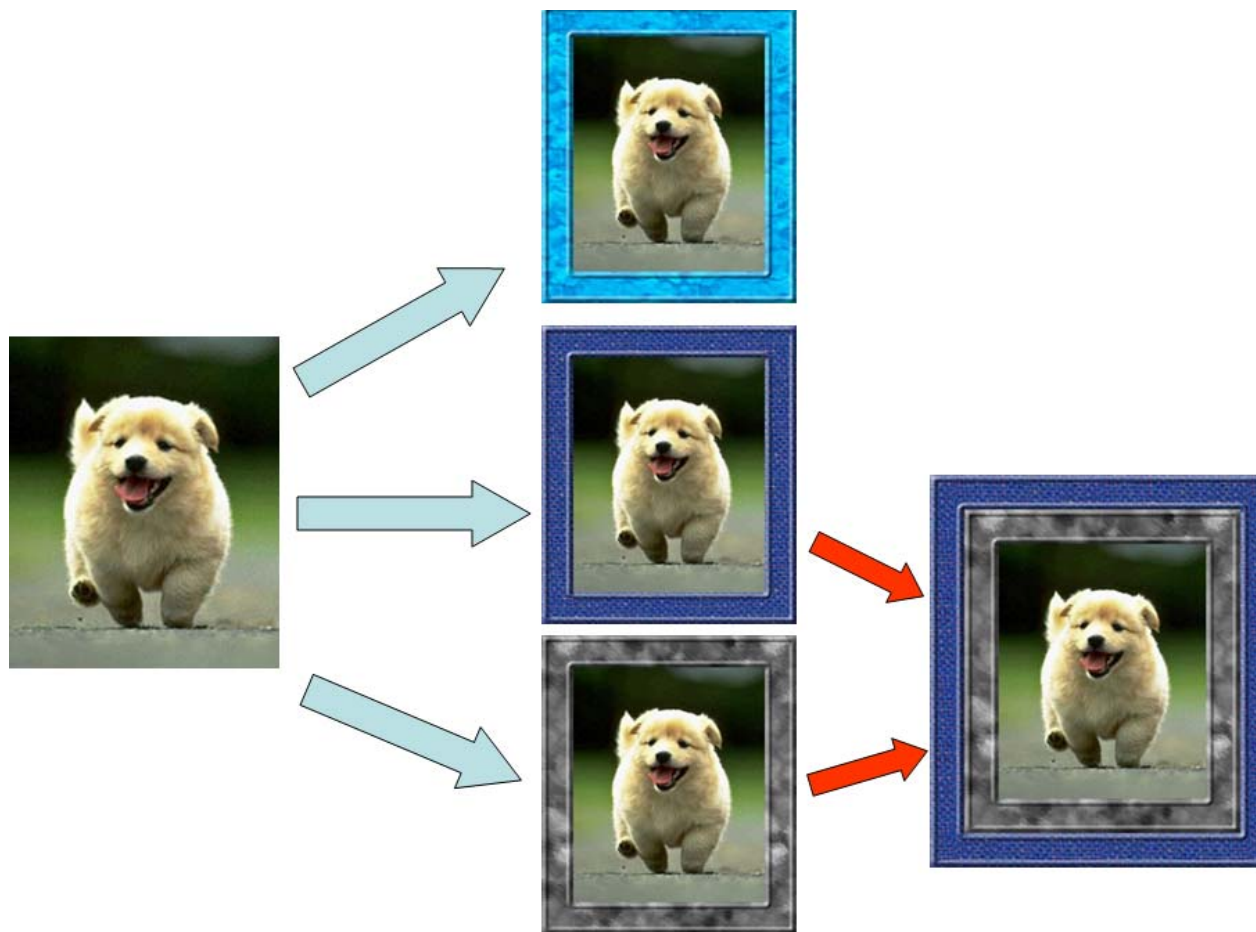
# 主要内容

- ◆ 1 装饰模式
- ◆ 2 外观模式

# 1 装饰模式

## ■ 装饰模式动机

□ 相片框





# 1 装饰模式

## ■ 装饰模式动机

- 可以在不改变一个对象本身功能的基础上给对象增加额外的新行为
- 是一种用于替代继承的技术，它通过一种无须定义子类的方式给对象动态增加职责，使用对象之间的关联关系取代类之间的继承关系
- 引入了装饰类，在装饰类中既可以调用待装饰的原有类的方法，还可以增加新的方法，以扩展原有类的功能



## 装饰模式定义

### □ 对象结构型模式

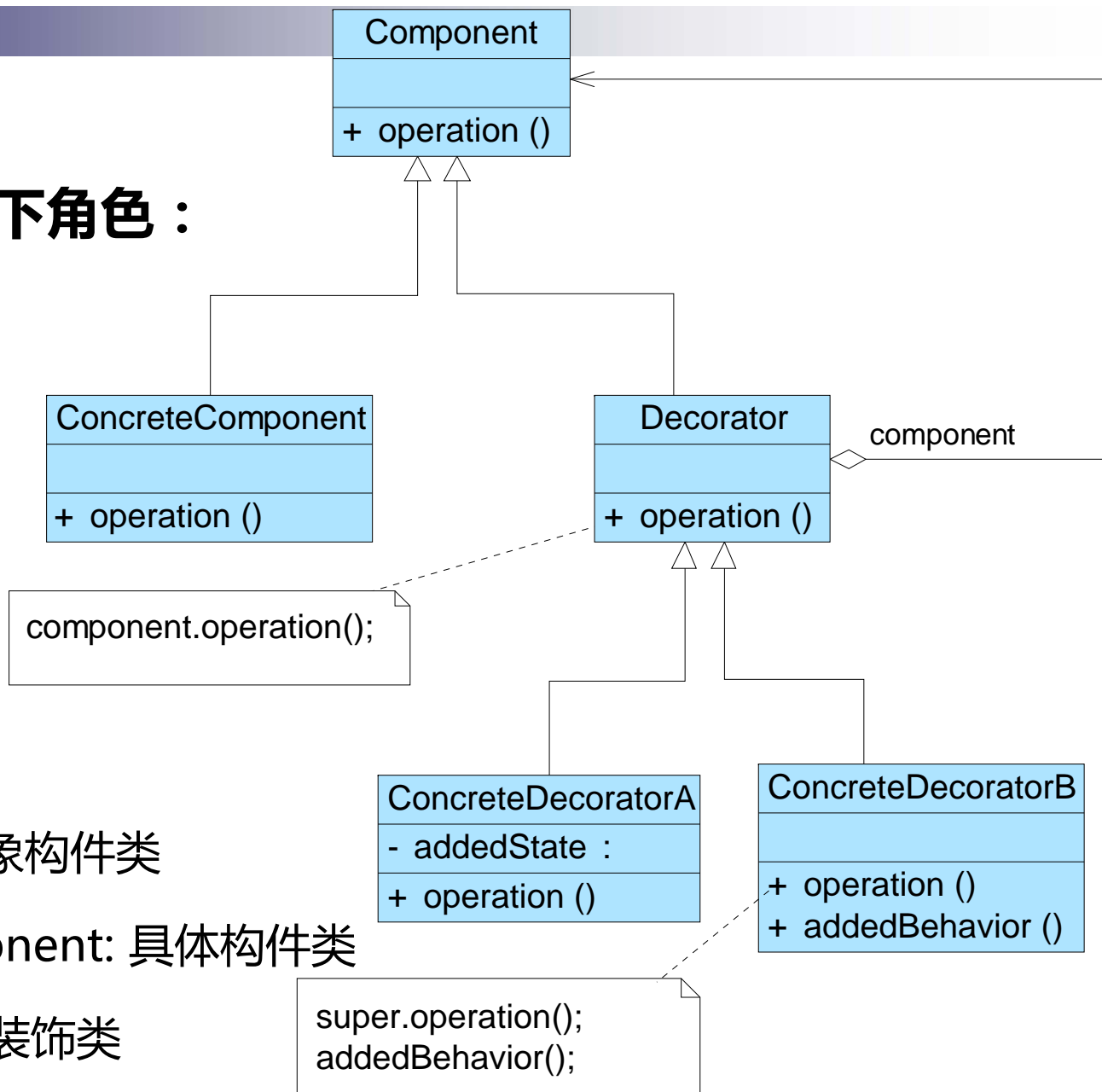
装饰模式：**动态地**给一个对象增加一些额外的职责。就扩展功能而言，装饰模式提供了一种**比使用子类更加灵活的替代方案**。

**Decorator Pattern:** Attach additional responsibilities to an object **dynamically**. Decorators provide **a flexible alternative to subclassing** for extending functionality.

- 以对客户透明的方式动态地给一个对象附加上更多的责任
- 可以在不需要创建更多子类的情况下，让对象的功能得以扩展

## 装饰模式结构

### ■ 装饰模式包含如下角色：



- Component: 抽象构件类
- ConcreteComponent: 具体构件类
- Decorator: 抽象装饰类
- ConcreteDecorator: 具体装饰类



## 装饰模式分析

### ■ 抽象装饰类示例代码：

```
public class Decorator extends Component {  
    private Component component; //维持一个对抽象构件对象的引用  
  
    //注入一个抽象构件类型的对象  
    public Decorator(Component component) {  
        this.component=component;  
    }  
  
    public void operation() {  
        component.operation(); //调用原有业务方法  
    }  
}
```



## 装饰模式分析

### ■ 具体装饰类示例代码：

```
public class ConcreteDecorator extends Decorator {  
    public ConcreteDecorator(Component component) {  
        super(component);  
    }  
  
    public void operation() {  
        super.operation(); //调用原有业务方法  
        addedBehavior(); //调用新增业务方法  
    }  
  
    //新增业务方法  
    public void addedBehavior() {  
        .....  
    }  
}
```





## 装饰模式分析

### ■ 透明装饰模式

- 透明(Transparent)装饰模式：要求客户端完全针对抽象编程，装饰模式的透明性要求客户端程序不应该将对象声明为具体构件类型或具体装饰类型，而应该全部声明为抽象构件类型
- 对于客户端而言，具体构件对象和具体装饰对象没有任何区别
- 可以让客户端透明地使用装饰之前的对象和装饰之后的对象，无须关心它们的区别
- 可以对一个已装饰过的对象进行多次装饰，得到更为复杂、功能更为强大的对象
- 无法在客户端单独调用新增方法addedBehavior()



## 装饰模式分析

### ■ 透明装饰模式

```
.....  
//全部使用抽象构件定义  
Component component_o,component_d1,component_d2;  
component_o = new ConcreteComponent();  
component_d1 = new ConcreteDecorator1(component_o);  
component_d2 = new ConcreteDecorator2(component_d1);  
component_d2.operation();  
//无法单独调用component_d2的addedBehavior()方法  
.....
```



## 装饰模式分析

### ■ 不透明装饰模式

- 半透明(Semi-transparent)装饰模式：用具体装饰类型来定义装饰之后的对象，而具体构件使用抽象构件类型来定义
- 对于客户端而言，具体构件类型无须关心，是透明的；但是具体装饰类型必须指定，这是不透明的
- 可以给系统带来更多的灵活性，设计相对简单，使用起来也非常方便
- 客户端使用具体装饰类型来定义装饰后的对象，因此可以单独调用 `addedBehavior()` 方法
- 最大的缺点在于不能实现对同一个对象的多次装饰，而且客户端需要有区别地对待装饰之前的对象和装饰之后的对象



## 装饰模式分析

### ■ 不透明装饰模式

.....

**Component component\_o;** //使用抽象构件类型定义

**component\_o = new ConcreteComponent();**

**component\_o.operation();**

**ConcreteDecorator component\_d;** //使用具体装饰类型定义

**component\_d = new ConcreteDecorator(component\_o);**

**component\_d.operation();**

**component\_d.addedBehavior();** //单独调用新增业务方法

.....



## 装饰模式实例与解析

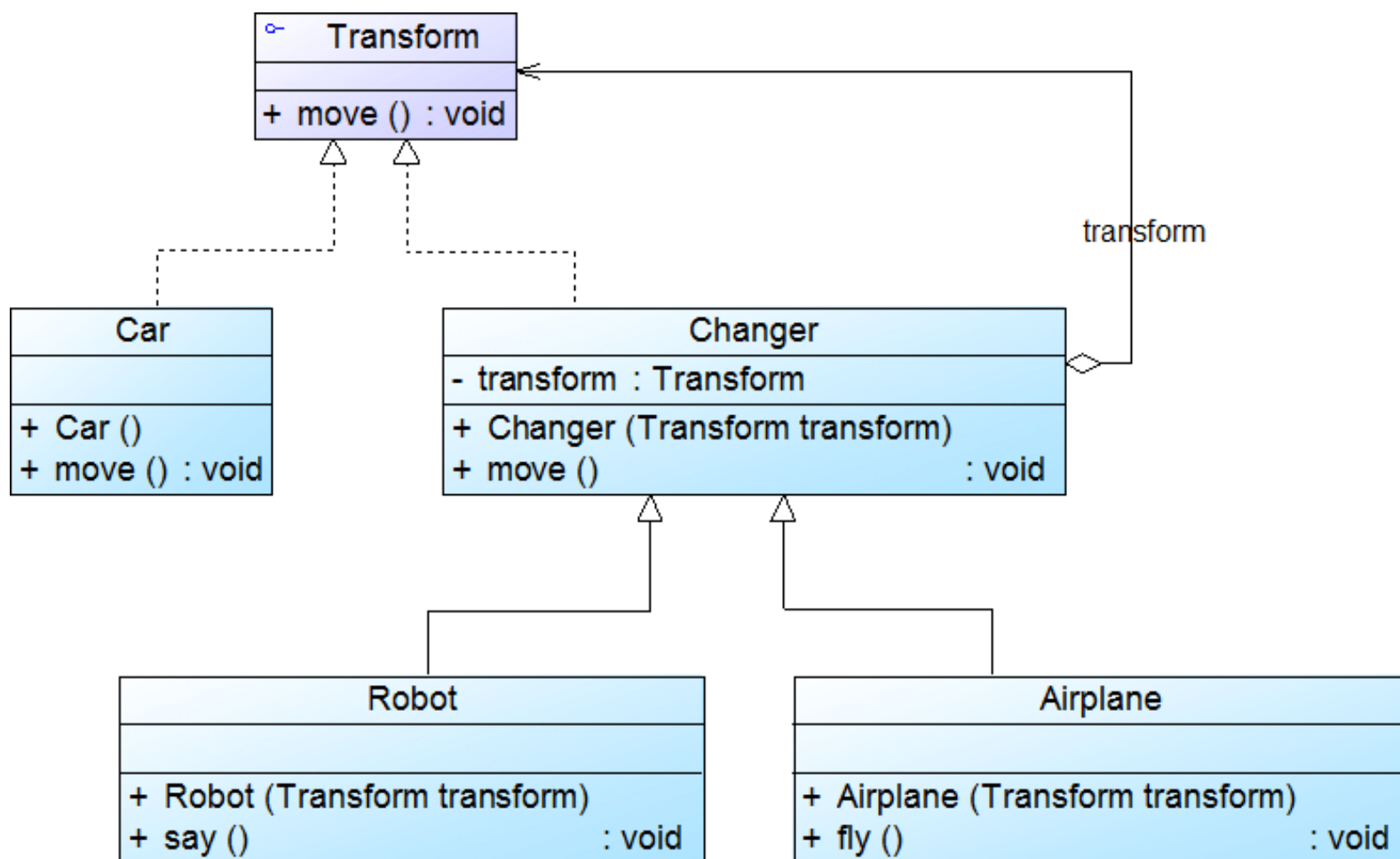
### ■ 装饰模式实例

#### □ 变形金刚：实例说明

- 变形金刚在变形之前是一辆汽车，它可以在陆地上移动。当它变成机器人之后除了能够在陆地上移动之外，还可以说话；如果需要，它还可以变成飞机，除了在陆地上移动还可以在天空中飞翔。

## 装饰模式实例

### ■ 变形金刚：参考类图



## 装饰模式实例

- 简单工厂模式变形金刚：参考代码
  - DesignPatterns之decorator包

Transform.java

```
1 package decorator;
2
3 public interface Transform
4 {
5     public void move();
6 }
```

Changer.java

```
1 package decorator;
2
3 public class Changer implements Transform
4 {
5     private Transform transform;
6
7     public Changer(Transform transform)
8     {
9         this.transform=transform;
10    }
11
12    public void move()
13    {
14        transform.move();
15    }
16 }
```



Robot.java

```
1 package decorator;
2
3 public class Robot extends Changer
4 {
5     public Robot(Transform transform)
6     {
7         super(transform);
8         System.out.println("变成机器人! ");
9     }
10
11     public void say()
12     {
13         System.out.println("说话! ");
14     }
15 }
```

Airplane.java

```
1 package decorator;
2
3 public class Airplane extends Changer
4 {
5     public Airplane(Transform transform)
6     {
7         super(transform);
8         System.out.println("变成飞机! ");
9     }
10
11     public void fly()
12     {
13         System.out.println("在天空飞翔! ");
14     }
15 }
```

Car.java

```
1 package decorator;
2
3 public final class Car implements Transform
4 {
5     public Car()
6     {
7         System.out.println("变形金刚是一辆车！");
8     }
9
10    public void move()
11    {
12        System.out.println("在陆地上移动！");
13    }
14 }
```

Client.java

```
1 package decorator;
2
3 public class Client
4 {
5     public static void main(String args[])
6     {
7         Transform camaro;
8         camaro=new Car();
9         camaro.move();
10        System.out.println("-----");
11
12        Airplane bumblebee=new Airplane(camaro);
13        bumblebee.move();
14        bumblebee.fly();
15    }
16 }
```



## 装饰模式效果与应用

### ■ 装饰模式优点：

- 对于扩展一个对象的功能，装饰模式比继承更加灵活，不会导致类的个数急剧增加
- 可以通过一种动态的方式来扩展一个对象的功能，通过配置文件可以在运行时选择不同的具体装饰类，从而实现不同的行为
- 可以对一个对象进行多次装饰
- 具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，且原有类库代码无须改变，符合开闭原则



## 装饰模式效果与应用

### ■ 装饰模式缺点：

- 使用装饰模式进行系统设计时将产生很多小对象，大量小对象的产生势必会占用更多的系统资源，在一定程度上影响程序的性能
- 比继承更加易于出错，排错也更困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐



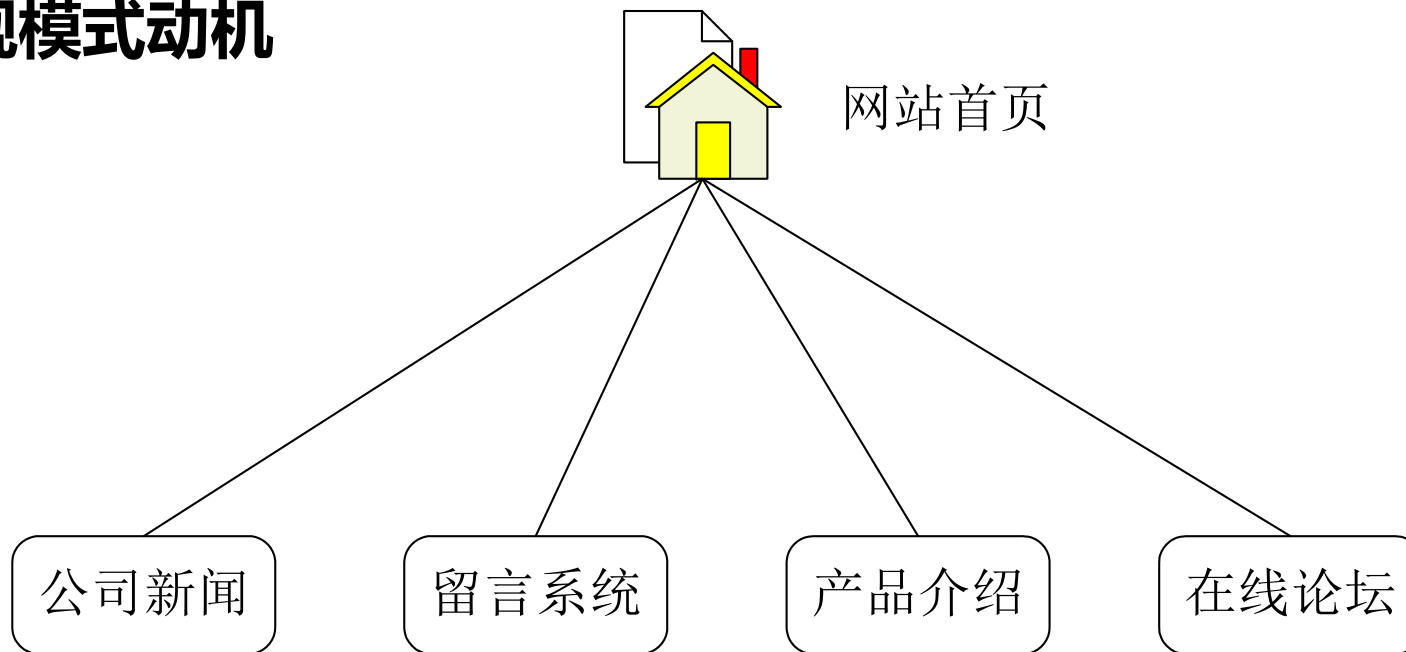
## 装饰模式效果与应用

### ■ 在以下情况下可以使用装饰模式：

- 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
- 当不能采用继承的方式对系统进行扩展或者采用继承不利于系统扩展和维护时可以使用装饰模式

## 2 外观模式

### ■ 外观模式动机



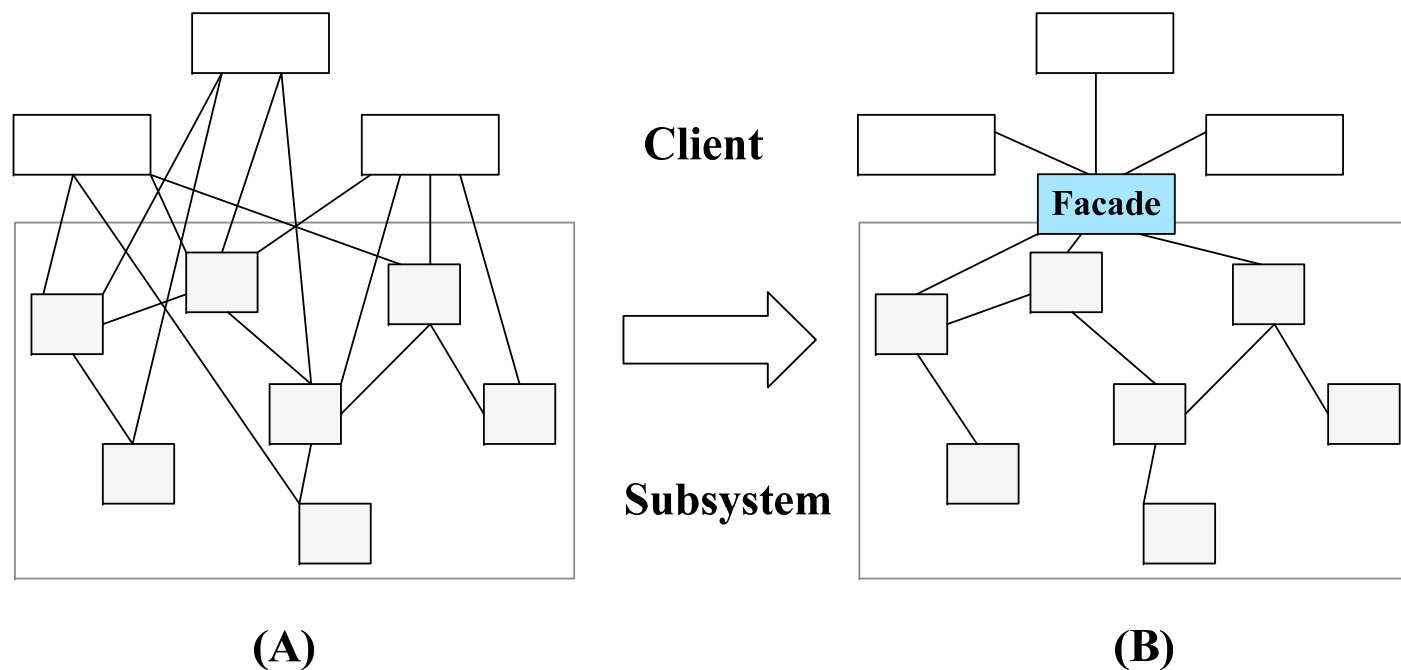




## 外观模式动机

- 一个客户类需要和多个业务类交互，有时候这些需要交互的业务类会作为一个整体出现
- 引入一个新的外观类(Facade)来负责和多个业务类【子系统(Subsystem)】进行交互，而客户类只需与外观类交互
- 为多个业务类的调用提供了一个统一的入口，简化了类与类之间的交互
- 没有外观类：每个客户类需要和多个子系统之间进行复杂的交互，系统的耦合度将很大
- 引入外观类：客户类只需要直接与外观类交互，客户类与子系统之间原有的复杂引用关系由外观类来实现，从而降低了系统的耦合度

## 外观模式动机



- 一个子系统的外部与其内部的通信通过一个统一的外观类进行，外观类将客户类与子系统的内部复杂性分隔开，使得客户类只需要与外观角色打交道，而不需要与子系统内部的很多对象打交道

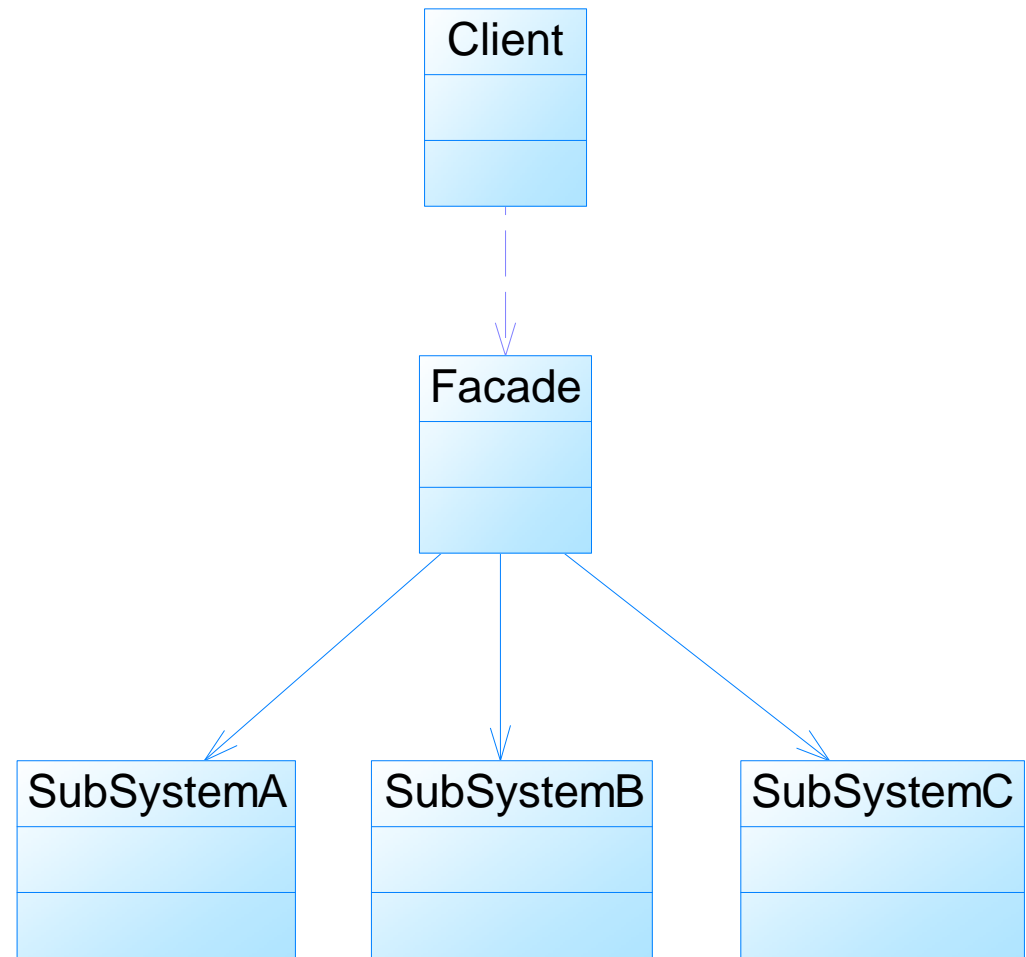
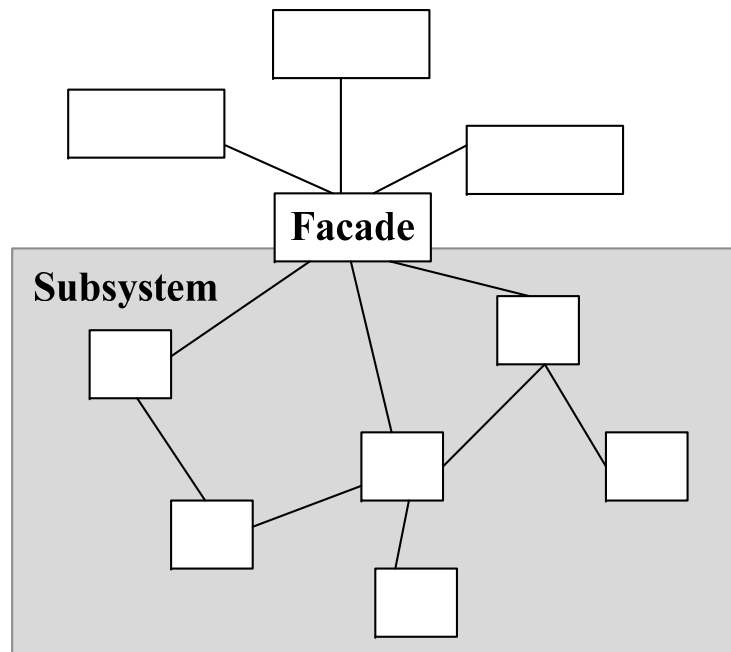


## 外观模式定义

- 外观模式(Facade Pattern)：外部与子系统的通信通过一个统一的外观对象进行，为子系统的一组接口提供一个统一的入口
- 外观模式定义了一个高层接口，这个接口使得子系统更加容易使用
- 外观模式又称为门面模式，它是一种对象结构型模式

## 外观模式结构

- 外观模式包含如下角色：
  - Facade: 外观角色
  - SubSystem: 子系统角色





## 外观分析

- 是迪米特法则的一种具体实现
- 通过引入一个新的外观角色来降低原有系统的复杂度，同时降低客户类与子系统的耦合度
- 所指的子系统是一个广义的概念，它可以是一个类、一个功能模块、系统的一个组成部分或者一个完整的系统



## 外观模式分析

### ■ 外观类示例代码：

```
public class Facade {  
    private SubSystemA obj1 = new SubSystemA();  
    private SubSystemB obj2 = new SubSystemB();  
    private SubSystemC obj3 = new SubSystemC();  
  
    public void method() {  
        obj1.method();  
        obj2.method();  
        obj3.method();  
    }  
}
```



## 外观模式实例

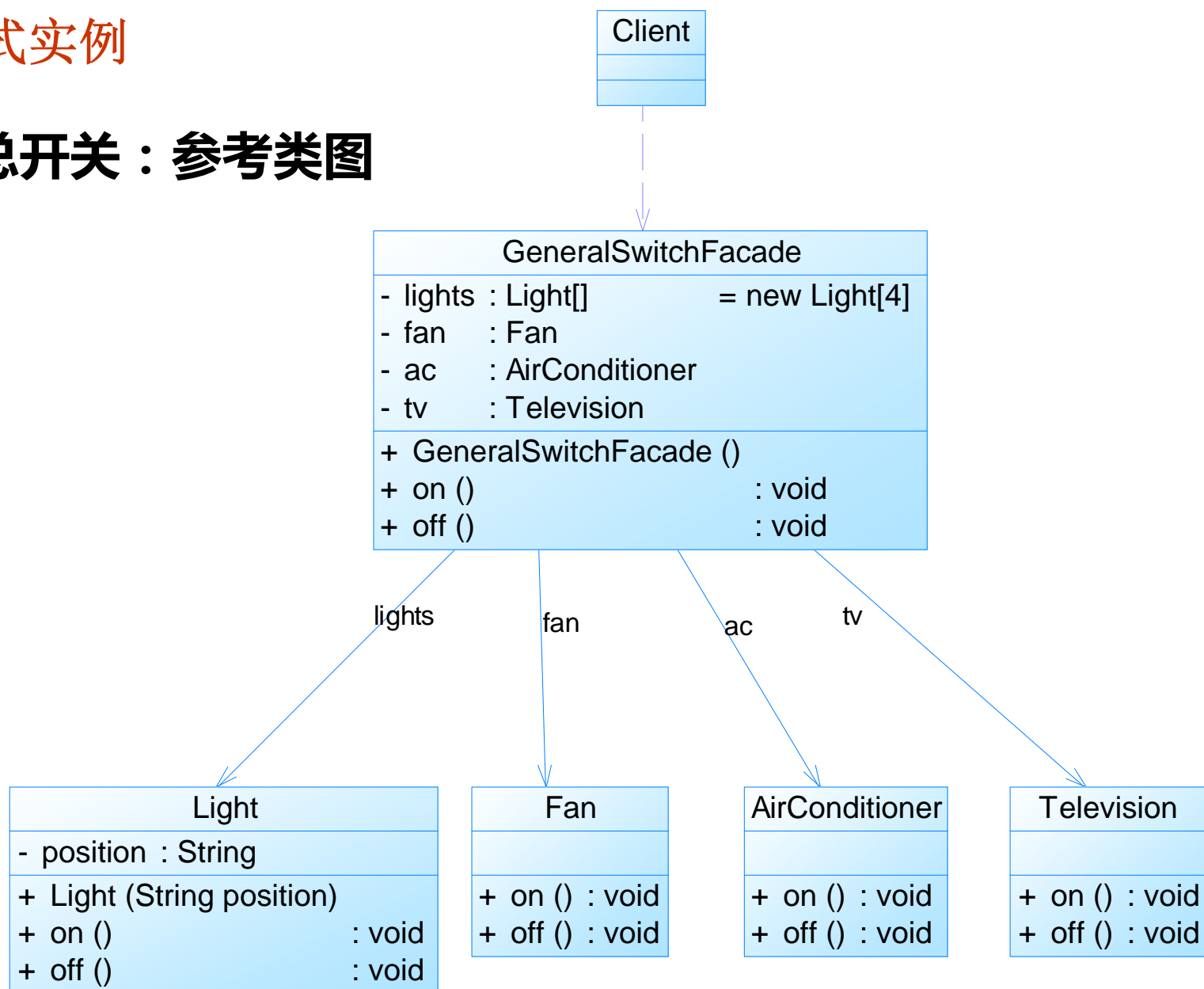
### ■ 外观模式实例

#### □ 电源总开关：实例说明

- 现在考察一个电源总开关的例子，以便进一步说明外观模式。为了方便，一个电源总开关可以控制四盏灯、一个风扇、一台空调和一台电视机的启动和关闭。通过该电源总开关可以同时控制上述所有电器设备，使用外观模式设计该系统。

## 外观模式实例

### □ 电源总开关：参考类图





## 外观模式实例

### ■ 电源总开关：参考代码

#### □ DesignPatterns之facade包

```
Client.java ✕
1 package facade;
2
3 public class Client
4 {
5     public static void main(String args[])
6     {
7         GeneralSwitchFacade gsf=new GeneralSwitchFacade();
8         gsf.on();
9         System.out.println("-----");
10        gsf.off();
11    }
12 }
```

GeneralSwitchFacade.java

```
3 public class GeneralSwitchFacade {
4     private Light lights[]=new Light[4];
5     private Fan fan;
6     private AirConditioner ac;
7     private Television tv;
8
9     public GeneralSwitchFacade()
10    {
11        lights[0]=new Light("左前");
12        lights[1]=new Light("右前");
13        lights[2]=new Light("左后");
14        lights[3]=new Light("右后");
15        fan=new Fan();
16        ac=new AirConditioner();
17        tv=new Television();
```

GeneralSwitchFacade.java

```
19
20    public void on()
21    {
22        lights[0].on();
23        lights[1].on();
24        lights[2].on();
25        lights[3].on();
26        fan.on();
27        ac.on();
28        tv.on();
29    }
```

GeneralSwitchFacade.java

```
30
31    public void off()
32    {
33        lights[0].off();
34        lights[1].off();
35        lights[2].off();
36        lights[3].off();
37        fan.off();
38        ac.off();
39        tv.off();
40    }
41 }
```

AirConditioner.java

```
1 package facade;
2
3 public class AirConditioner
4 {
5     public void on()
6     {
7         System.out.println("空调打开!");
8     }
9
10    public void off()
11    {
12        System.out.println("空调关闭!");
13    }
14 }
```

Fan.java

```
1 package facade;
2
3 public class Fan
4 {
5     public void on()
6     {
7         System.out.println("风扇打开!");
8     }
9
10    public void off()
11    {
12        System.out.println("风扇关闭!");
13    }
14
15 }
```

Television.java

```
1 package facade;
2
3 public class Television
4 {
5     public void on()
6     {
7         System.out.println("电视机打开!");
8     }
9
10    public void off()
11    {
12        System.out.println("电视机关闭!");
13    }
14 }
```

Light.java

```
1 package facade;
2
3 public class Light
4 {
5     private String position;
6
7     public Light(String position)
8     {
9         this.position=position;
10    }
11
12    public void on()
13    {
14        System.out.println(this.position + "灯打开! ");
15    }
16
17    public void off()
18    {
19        System.out.println(this.position + "灯关闭! ");
20    }
21 }
```



## 外观模式效果与应用

### ■ 外观模式优点：

- 它对客户端屏蔽了子系统组件，减少了客户端所需处理的对象数目，并使得子系统使用起来更加容易
- 它实现了子系统与客户端之间的松耦合关系，这使得子系统的变化不会影响到调用它的客户端，只需要调整外观类即可
- 一个子系统的修改对其他子系统没有任何影响，而且子系统的内部变化也不会影响到外观对象



## 外观模式效果与应用

### ■ 外观模式缺点：

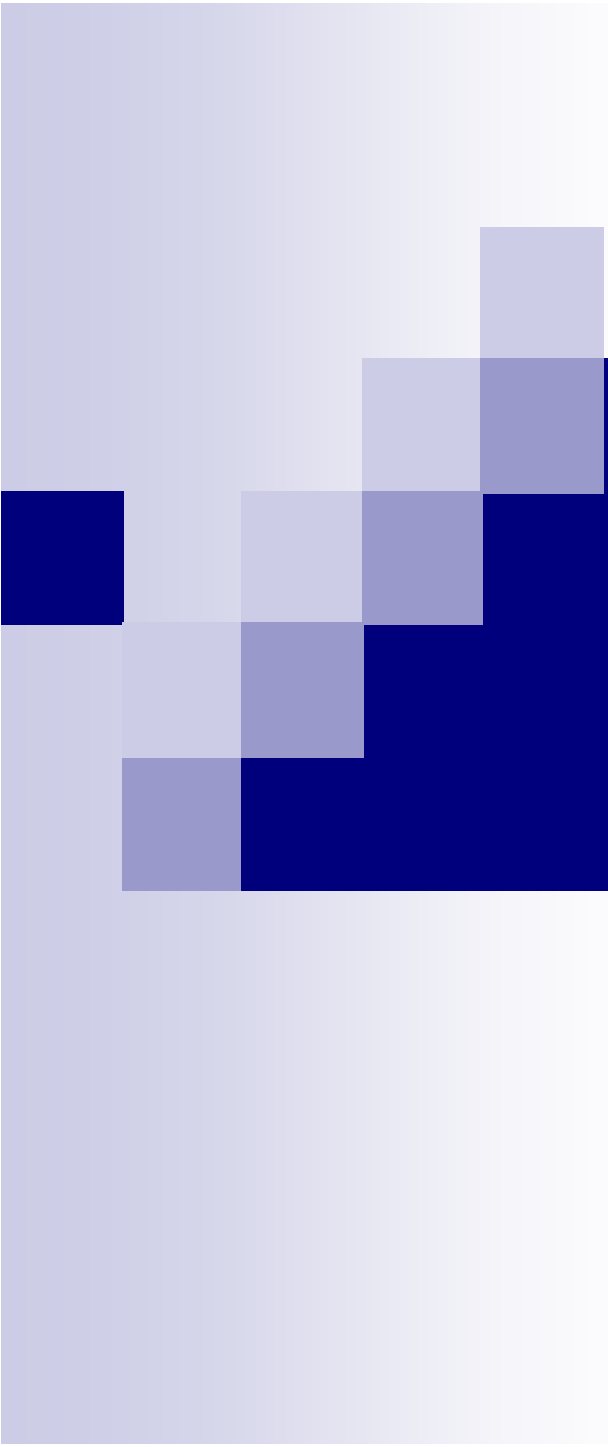
- 不能很好地限制客户端直接使用子系统类，如果对客户端访问子系统类做太多的限制则减少了可变性和灵活性
- 如果设计不当，增加新的子系统可能需要修改外观类的源代码，违背了开闭原则



## 外观模式效果与应用

### ■ 在以下情况下可以使用外观模式：

- 要为访问一系列复杂的子系统提供一个简单入口
- 客户端程序与多个子系统之间存在很大的依赖性
- 在层次化结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而是通过外观类建立联系，降低层之间的耦合度



谢谢