

Java语言程序设计 之设计模式 Design Pattern

主讲：辜希武

华中科技大学计算机学院IDC实验室

<http://idc.hust.edu.cn/>

Email: guxiwu@mail.hust.edu.cn

设计模式 (Design Pattern)

什么是设计模式

在面向对象程序设计 (OOP) 过程中，我们经常会遇到很多重复出现的问题，总结解决这些问题的成功经验和最佳实践便形成了设计模式 (Design Pattern)。

其核心思想是将可重用的解决方案总结出来，并分门别类。从而指导设计，减少代码重复和优化体系结构。

设计模式 (Design Pattern)

采用设计模式的好处

- 重用，避免代码重复冗余
- 优化体系结构
- 提升系统的可维护性和弹性
- 为性能优化提供便利
- 使软件质量更加有保证
- 增强代码可读性，便于团队交流
- 有助于整体提升团队水平

设计模式的原则

- ◆ 依赖倒置原则
 - ◆ 接口隔离原则
 - ◆ 开闭原则
 - ◆ 多用组合，少用继承
 - ◆ 职责分离原则
-
- ◆ 设计模式就是实现了上述原则，从而达到代码复用、增加可维护性的目的

依赖倒置原则 (DIP)

◆ 定义： **高层模块不应依赖低层模块，二者都应该依赖于抽象**

◆ 高层模块只应该包含重要的业务模型和策略选择，低层模块则是不同业务和策略的实现

◆ 高层模块不依赖高层和低层模块的具体实现，最多只依赖于低层的抽象

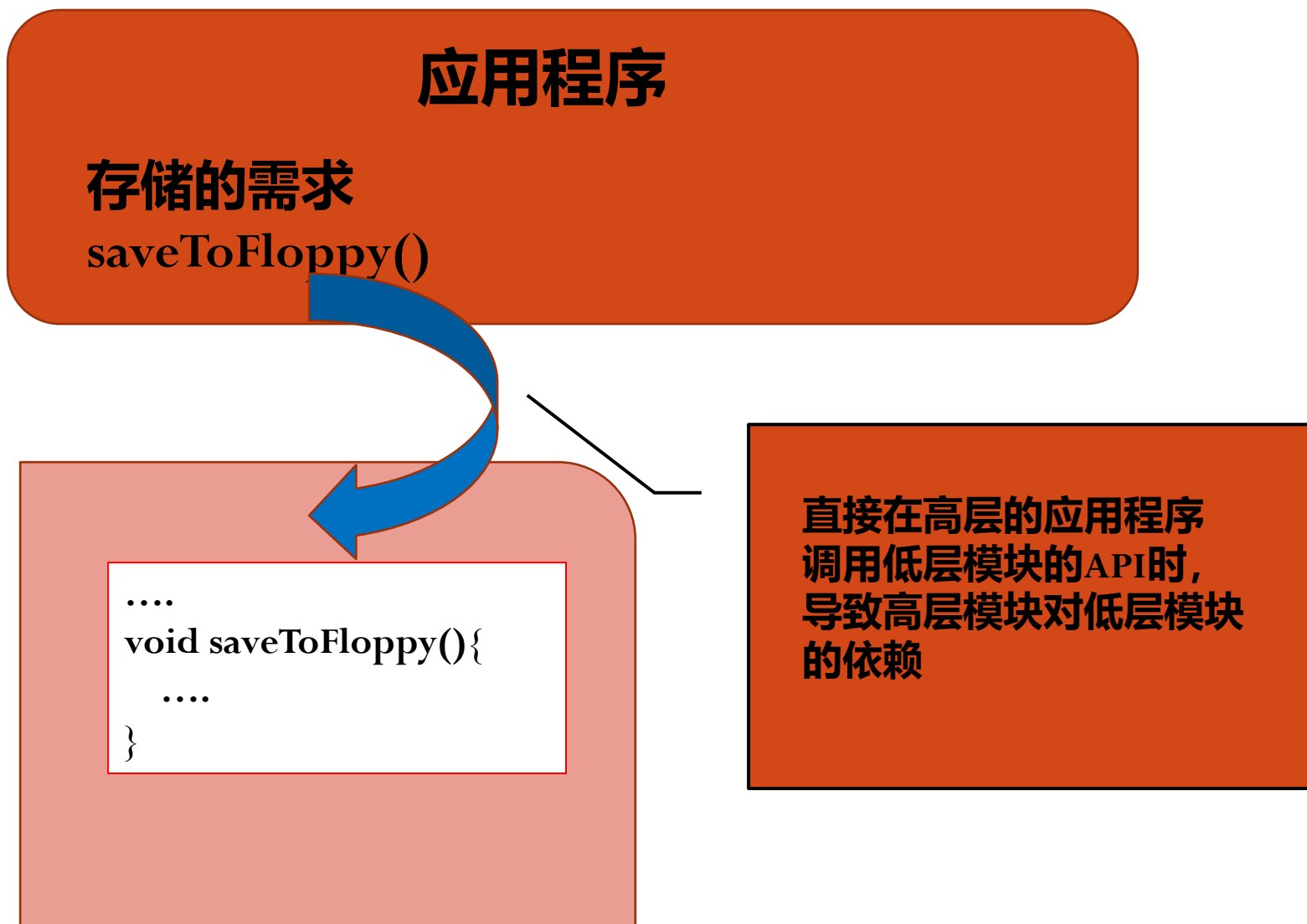
◆ 低层模块和实现也只依赖于抽象

◆ 辅助原则

◆ 任何变量的类型都不应该是具体类

◆ 任何类都不应该从具体类派生

依赖倒置原则 (DIP) 举例



依赖倒置原则 (DIP) 举例

```
public class Business{  
    private FloppyWriter writer = new FloppyWriter ();  
    ...  
    public void save(){  
        writer.saveToFloppy();  
    }  
}
```

高层业务类 Business的实现依赖于低层FloppyWriter 类，如果想把存储介质改为USB盘，则必须修改Business类，使得无法重用Business类

违反了依赖倒置原则！

依赖倒置原则 (DIP) 举例

利用接口抽象，可以改进

```
public interface IDeviceWriter{  
    public void saveToDevice();  
}
```

```
public class Business{  
    private IDeviceWriter writer;  
    ...  
    public void setDeviceWriter(IDeviceWriter writer){  
        this.writer = writer;  
    }  
    public void save(){  
        writer.saveToDevice();  
    }  
}
```

高层 (Business类) 的
实现依赖于抽象 (接口)

依赖倒置原则 (DIP) 举例

在这样的设计下，Business就是可重用的。如果今天有存储至Floppy或USB的需求，只要针对这两种存储需求分别实现IDeviceWriter接口即可。

```
public class FloppyWriter implements IDeviceWriter{  
    public void saveToDevice{  
        ... //存储至软盘的代码  
    }  
}  
  
public class USBWriter implements IDeviceWriter{  
    public void saveToDevice{  
        ...//存储至USB的代码  
    }  
}
```

低层类的实现也
依赖于抽象（接口）

依赖倒置原则（DIP）举例

如果应用程序需要USB存储，则编写如下配置程序：

```
Business business= new Business();  
business.setDeviceWriter(new USBWriter()); //对象注入  
business.save();
```

可以看到，无论低层的存储实现如何变动，对于Business类来说都无需修改。

如果采用对象工厂模式来动态创建实现存储功能的低层对象，连上面三行代码都不用写，只需要编写一个XML的配置文件，在配置文件里指定要注入到Business里面的低层存储对象即可。

这就是著名的Spring框架的一个重要功能：**对象注入，或者叫依赖注入（Dependency Injection）**

接口隔离原则 (ISP)

- ◆ 多个和客户相关的接口要好于一个通用接口
- ◆ 如果一个类有多个使用者，与其让这个类载入所有使用者需要使用的所有方法，还不如为每个使用者创建一个特定接口，并让该类分别实现这些接口

```
public class SeaPlane{  
    //与飞行有关的方法  
    void takeOff() { ... }  
    void fly() { ... }  
    void land() {...}  
  
    //与海上航行有关的方法  
    void dock() { ... }  
    void cruise() {...}  
}
```

//客户代码1
//只使用飞行功能

```
void f1(SeaPlane o){  
    o.takeOff();  
    o.fly();  
    o.land();  
}  
f1(new SeaPlane());
```

//客户代码2
//只使用海上航行功能

```
void f2(SeaPlane o){  
    o.dock();  
    o.cruise();  
}  
f2(new SeaPlane());
```

类SeaPlane封装了二类不同的功能。有二段客户代码分别只需要使用其中一类功能。但不得不声明SeaPlane类型的对象引用作为方法参数，导致客户代码被绑定到SeaPlane类型。
违反了一个原则：**基于接口编程，不要基于类编程**

```
public interface Flyer
//与飞行有关的方法
void takeOff();
void fly() ;
void land();
} //声明Flyer接口
```

```
public interface Sailer
//与海上航行有关的方法
void dock();
void cruise();
} //声明Sailer接口
```

```
public class SeaPlane implements Flyer,
Sailer{
//实现与飞行有关的方法
void takeOff() { ... }
void fly() { ... }
void land() {...}

//实现与海上航行有关的方法
void dock() { ... }
void cruise() {...}
}
```

现在首先声明二个分离的功能接口Flyer和Sailer

再用SeaPlane实现这二个接口，这样SeaPlane同时具有了
二类不同的行为特征

//客户代码1
//只使用飞行功能

```
void f1(Flyer o){  
    o.takeOff();  
    o.fly();  
    o.land();  
}
```

f1(new SeaPlane());

//客户代码2
//只使用海上航行功能

```
void f2(Sailer o){  
    o.dock();  
    o.cruise();  
}
```

f2(new SeaPlane());

接口类型的引用变量可以直接指向实现该接口的对象而不用强制类型转换！！实参传递给形参时，相当于

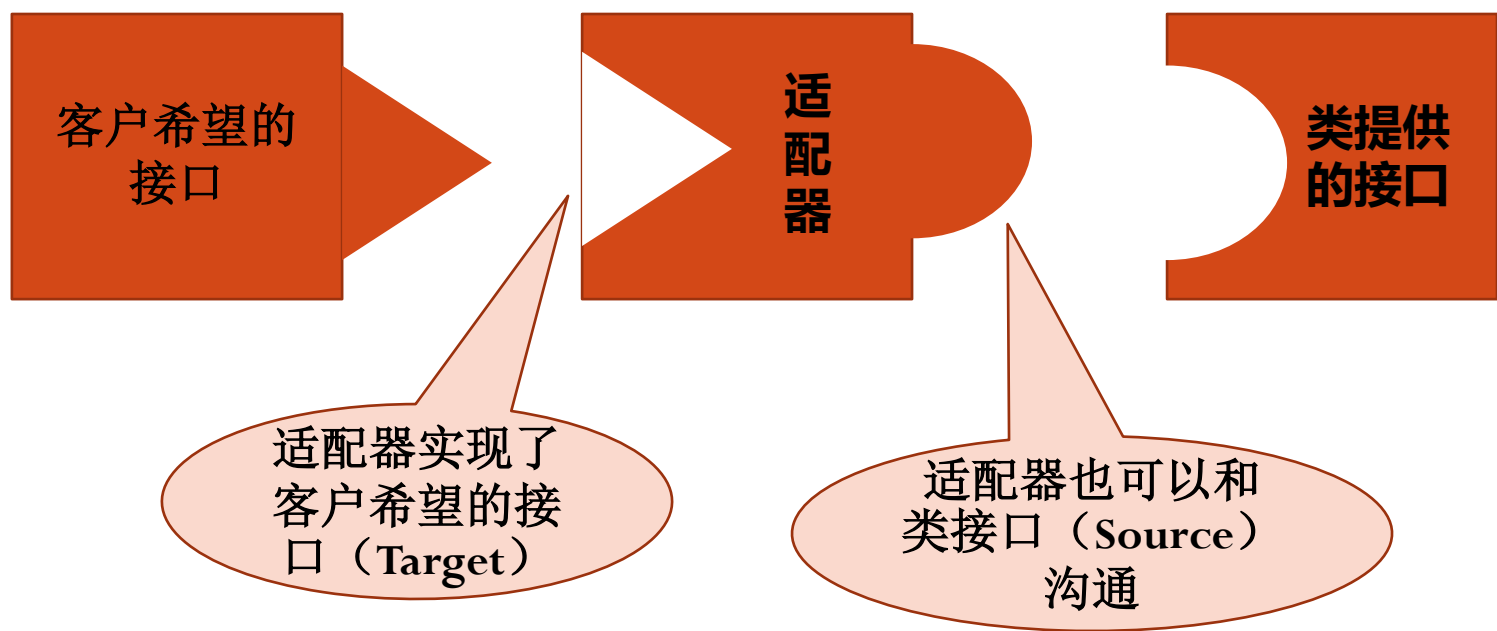
Flyer o = new SeaPlane();

当接口类型的引用变量指向不同的实现了接口的对象时，会表现出多态性

现在通过接口，将二类功能分离。同时方法的参数类型是接口，只要接口定义不变，接口的实现类再怎么修改，方法f1和f2都不需要修改。更重要的是，当传递新的接口实现类对象给方法时，方法会自动具有新的行为。这就是接口分离和基于接口编程的好处！

适配器模式的由来

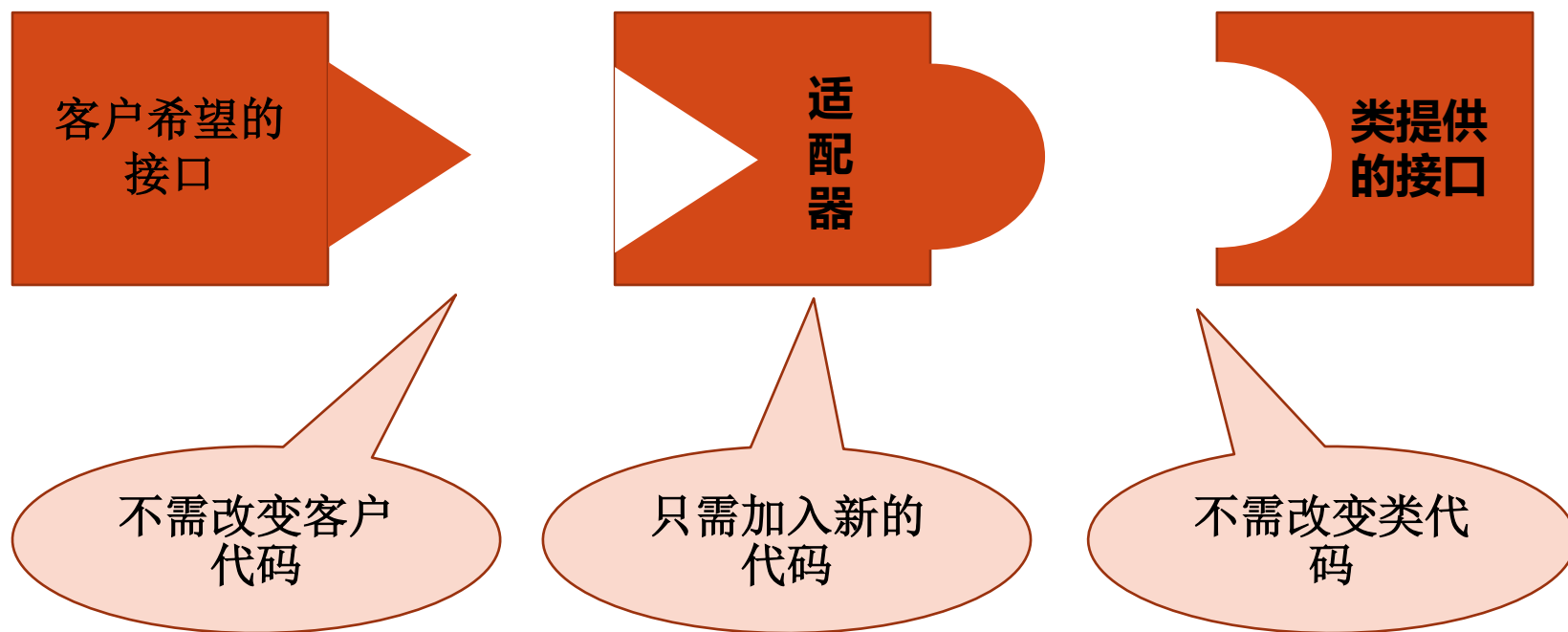
- 写一个类，将类提供的接口变成客户希望的接口
- 类提供的接口叫源接口（Source Interface），客户希望的接口叫目标接口（Target Interface）



适配器模式的由来

❑ 适配器完成了一个看似不可能的任务：把一个方块放进圆洞里

✓开闭原则：对扩展是开放的，对修改是关闭的



适配器模式的例子

- ❑ 如果它走路像只鸭子，叫起来像只鸭子，那么它必定可能是一只鸭子包装了鸭子适配器的火鸡

```
public interface Duck {  
    public void quack(); //呱呱叫  
    public void fly();   //飞行  
}
```

```
public interface Turkey{  
    public void gobble(); //咯咯叫  
    public void fly();   //飞  
}
```

适配器模式的例子

```
public class GreenDuck implements Duck{
    public void quack(){ System.out.println("Quack");}
    public void fly(){    System.out.println("I am flying");}
}

public class WildTurkey implements Turkey{
    public void gobble(){ System.out.println("Gobble gobble");}
    public void fly(){    System.out.println("I am flying");}
}
```

现在想用火鸡对象来冒充鸭子对象，由于接口不匹配，需要写个适配器。
注意：目标接口是Duck，源接口是Turkey

适配器模式的例子

适配器实现目标接口

```
public class TurkeyToDuckAdapter implements Duck{
```

```
    Turkey turkey;
```

包含一个被适配接口的引用,利用对象组合。

```
    public TurkeyToDuckAdapter (Turkey turkey){
```

```
        this.turkey = turkey;
```

在构造函数中传入被适配的接口引用

```
    }
```

```
    public void quack() {
```

```
        turkey.gobble();
```

实现Duck接口的quack方法,调用被适配接口对象的gobble方法

```
    }
```

```
    public void fly() {
```

```
        for(int i = 0; i < 5; i++)
```

```
            turkey.fly();
```

实现Duck接口的fly方法,火鸡飞行距离比鸭子短,因此连续调用5次火鸡的飞行

```
    }
```

适配器模式的例子

```
public class Client {  
    static void testDuck(Duck duck){  
        duck.quack();  
        duck.fly();  
    }  
    public static void main(String[] args){  
        GreenDuck duck = new GreenDuck();  
        testDuck(duck);  
        WildTuykey turkey = new WildTurkey();  
        Duck turkeyToDuckAdapter = new TurkeyToDuckAdapter (turkey);  
        testDuck(tukkeyDuckAdapter);  
    }  
}
```

客户程序需要Duck接口

传入一个真正的实现Duck接口的鸭子对象

把火鸡包装进适配器，使它看起来像鸭子

客户程序根本不知道，这其实是假装成鸭子的火鸡

适配器模式的意图和适用性

- **意图：** 将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的类可以一起工作
- **适用场合：**
 - 使用一个已经存在的类，而它的接口不符合要求
 - 创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作

装饰（Decorator）模式的由来

- ❑ 动态给对象添加额外职责。比如：一幅画有没有画框都可以挂在墙上，画是被装饰者。在挂在墙上之前，画可以被蒙上玻璃，装到框子里，玻璃画框就是装饰
- ❑ 不改变接口，但加入责任（功能）。Decorator提供了一种给类增加职责（功能）的方法，不是通过继承，而是通过对象组合实现的
- ❑ 就增加功能来说，Decorator模式相比生成子类（继承方式）更为灵活。
- ❑ 一个新的原则：多用组合，少用继承。利用组合、委托同样可以达到继承的目的

- Java、C#、SmallTalk等单继承语言在描述多继承的对象时，常常通过对象成员委托（代理）实现多继承。

```
class A {  
    public void f() {}  
}  
class B {  
    public void g() {}  
}
```

如果需要实现一个类，同时具有类A和类B的行为，但又不能多继承怎么办（如JAVA）？

采用对象组合方式，继承一个类，将另外一个类的对象作为数据成员

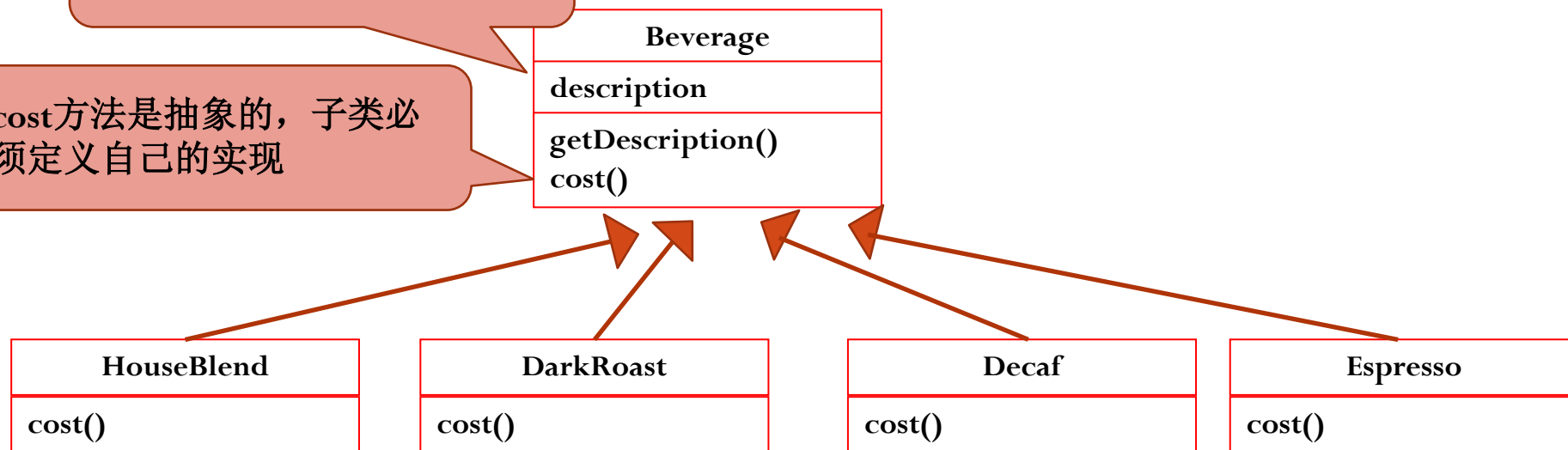
```
class C extends A {  
    B b = new B();           //B类行为的代理  
    public void g() {        //定义一个同名的g函数，但其功能  
        b.g();               //委托对象b完成(通过调用b.g()), 因此  
    }                        //C的g的行为与B的g完全一致  
};  
//这样C就具有A的行为f和B的行为g,达到了多重继承的效果
```

装饰 (Decorator) 模式的由来

□ 假设要实现一个星巴兹(Starbuzz)咖啡店的饮料类Beverage

Beverage是抽象类，店里所有饮料必须继承它

cost方法是抽象的，子类必须定义自己的实现



每个子类实现cost返回饮料的价钱

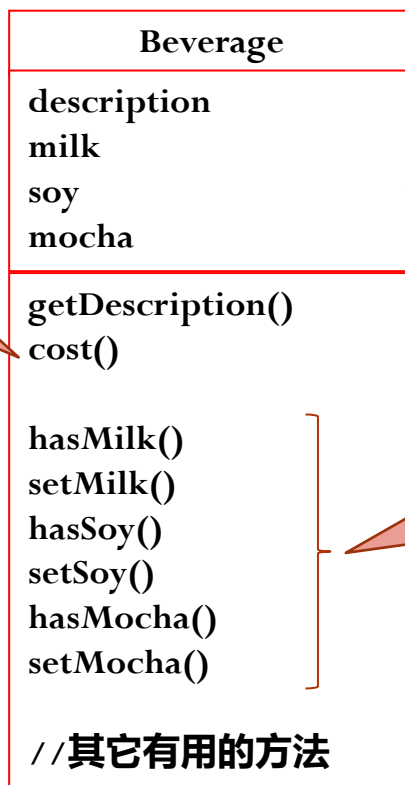
装饰 (Decorator) 模式的由来

- ❑ 购买咖啡时，可以在其中加入调料，例如牛奶(Milk)，豆浆(Soy)，摩卡(Mocha)。星巴兹会根据加入的调料收取不同的费用
- ❑ 第一个尝试：每加入一种新的调料，就派生一个新的子类，每个子类根据加入的调料实现cost方法
- ❑ 很明显，星巴兹为自己制造了一个维护的噩梦
 - ❑ 根据加入调料的不同组合，派生子类的数目呈组合爆炸
 - ❑ 如果调料价格变化，必须修改每个派生类的cost方法

装饰 (Decorator) 模式的由来

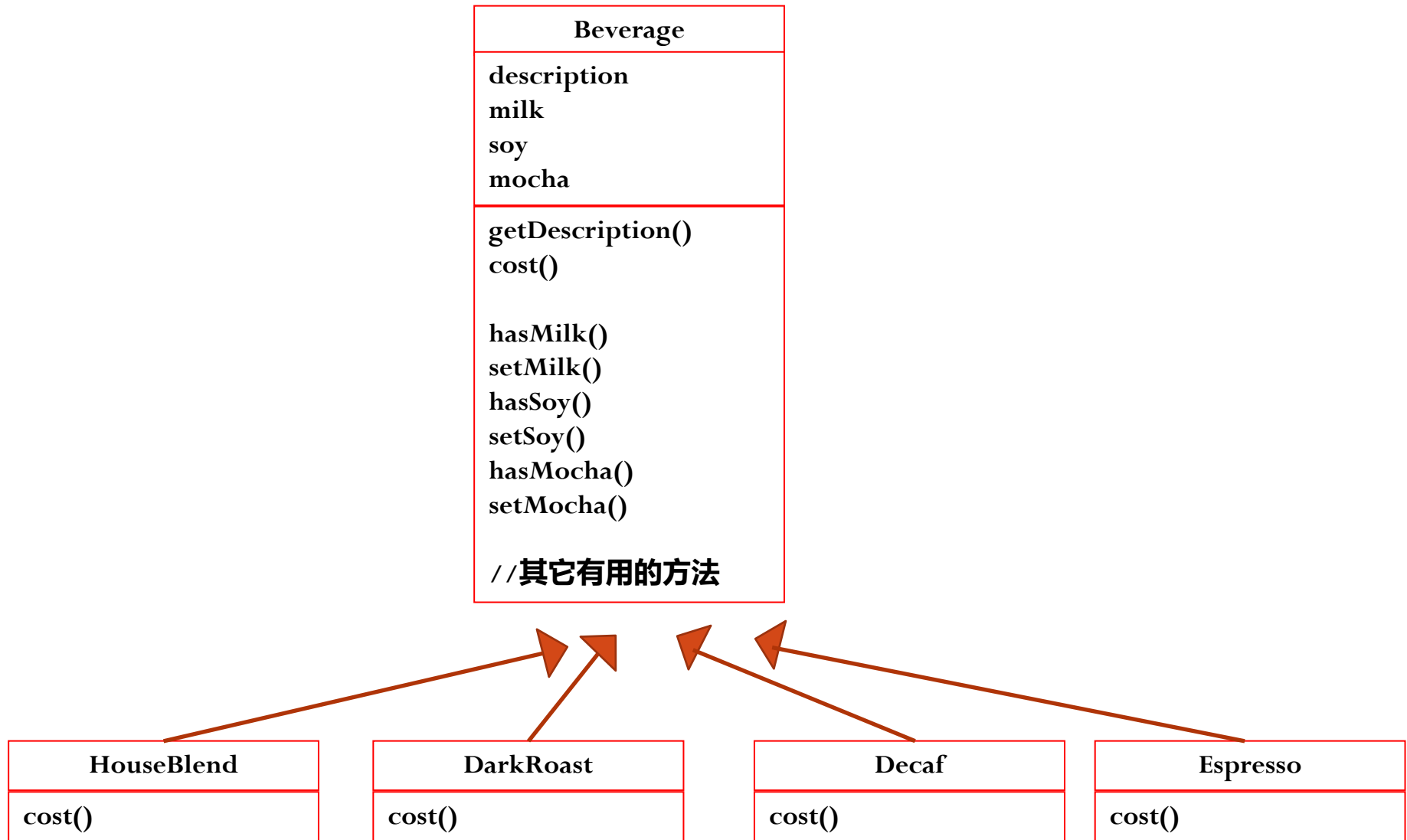
□ 能不能从基类下手，利用实例变量+继承，来追踪这些调料？

cost不再是抽象方法，而是
计算要加入的各种调料的价
钱



各种调料的布尔值

取得和设置调料的布尔值



子类的cost方法需要计算该饮料的价钱，然后通过调用超类的cost实现，加入调料的价钱

改进的Beverage实现

```
public class Beverage{
    //为milkCost、soyCost、mochaCost声明实例变量
    //为milk、soy、mocha声明getter、setter方法
    public double cost(){
        float condimentCost = 0.0;
        if(hasMilk()) condimentCost += milkCost;
        if(hasSoy()) condimentCost += soyCost;
        if(hasMocha()) condimentCost += mochaCost;
        return condimentCost;
    }
}

public class DarkRoast extends Beverage{
    public DarkRoast(){ description = "Most excellent Dark Roast";}
    public double cost(){
        return 1.99 + super.cost();
    }
}
```

改进的Beverage实现存在的问题

- ❑ 调料价钱改变会使我们更改现有Beverage代码
- ❑ 一旦出现新的调料，我们就需要在超类里加上新的实例变量来记录该调料的布尔值及价格，同时要修改cost实现
- ❑ 万一客户想要双倍mocha怎么办？
- ❑ 以后会出现新的饮料（如茶Tea）。作为Beverage的子类，某些调料可能并不适合（比如milk）。但是现在的设计方式中，Tea类仍将继承那些不适合的方法如hasMilk
 - ❑ 这就是继承带来的问题：利用继承来设计子类的行为，是在编译时静态决定的，而且所有的子类都会继承到相同的行为
- ❑ 如果利用对象组合的做法扩展对象的行为，就可以在运行时动态扩展。
- ❑ 装饰者模式就是利用这个技巧把多个新的职责（功能）、甚至是设计超类时还没有想到的职责（功能）加到对象上

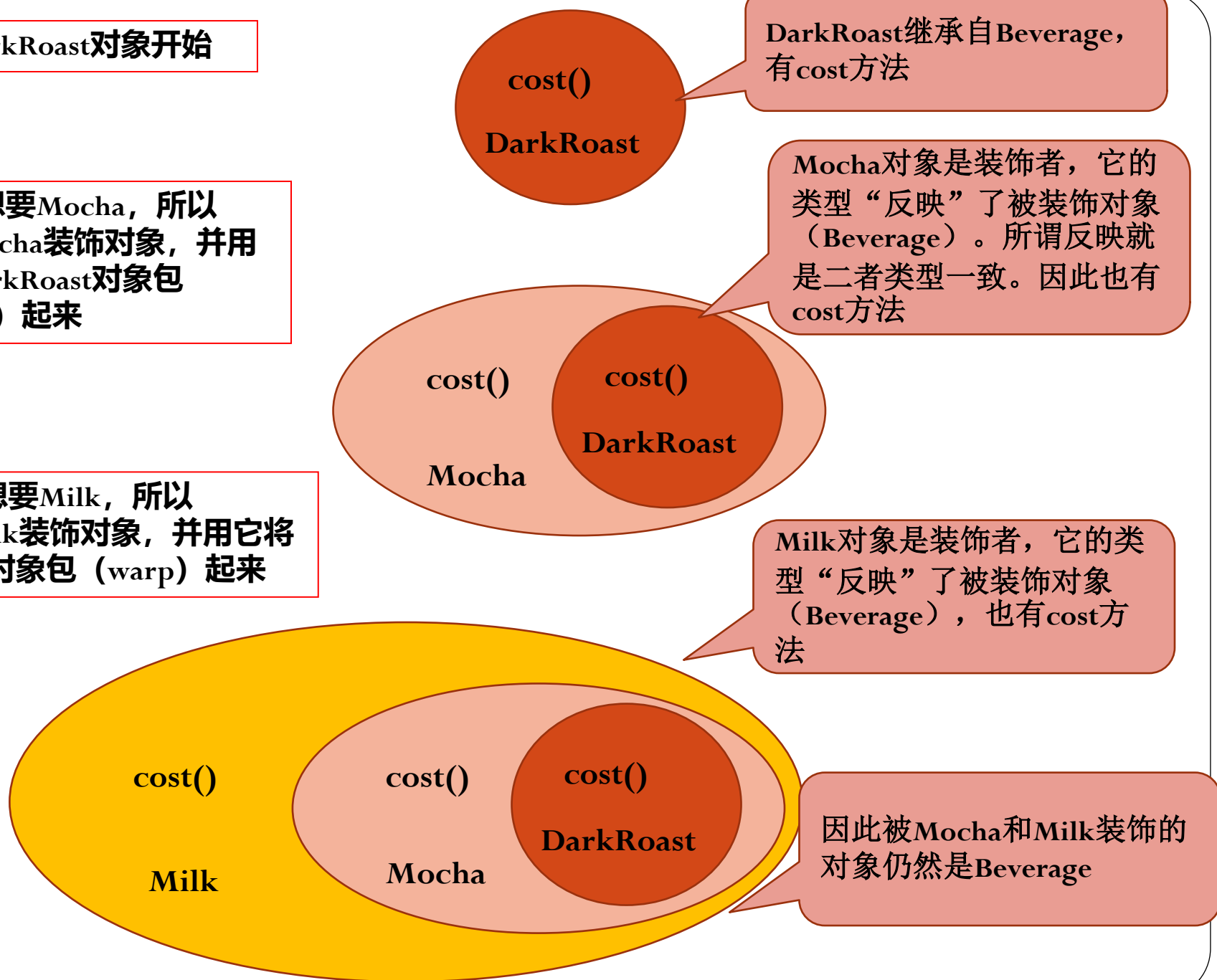
认识装饰者模式

- ❑ 现在知道了类继承带来的问题：类数量爆炸、设计死板、基类加入的新功能不适合所有子类
- ❑ 现在采用不一样的做法：以饮料为主体，然后在运行时以调料来“装饰”饮料，具体步骤为：
 - ❑ 拿一个饮料对象,如深焙咖啡(DarkRoast)
 - ❑ 以摩卡(Mocha)对象装饰它
 - ❑ 以牛奶(Milk)对象装饰它
 - ❑ 调用cost方法，并依赖委托（delegate）将调料的价钱加上去

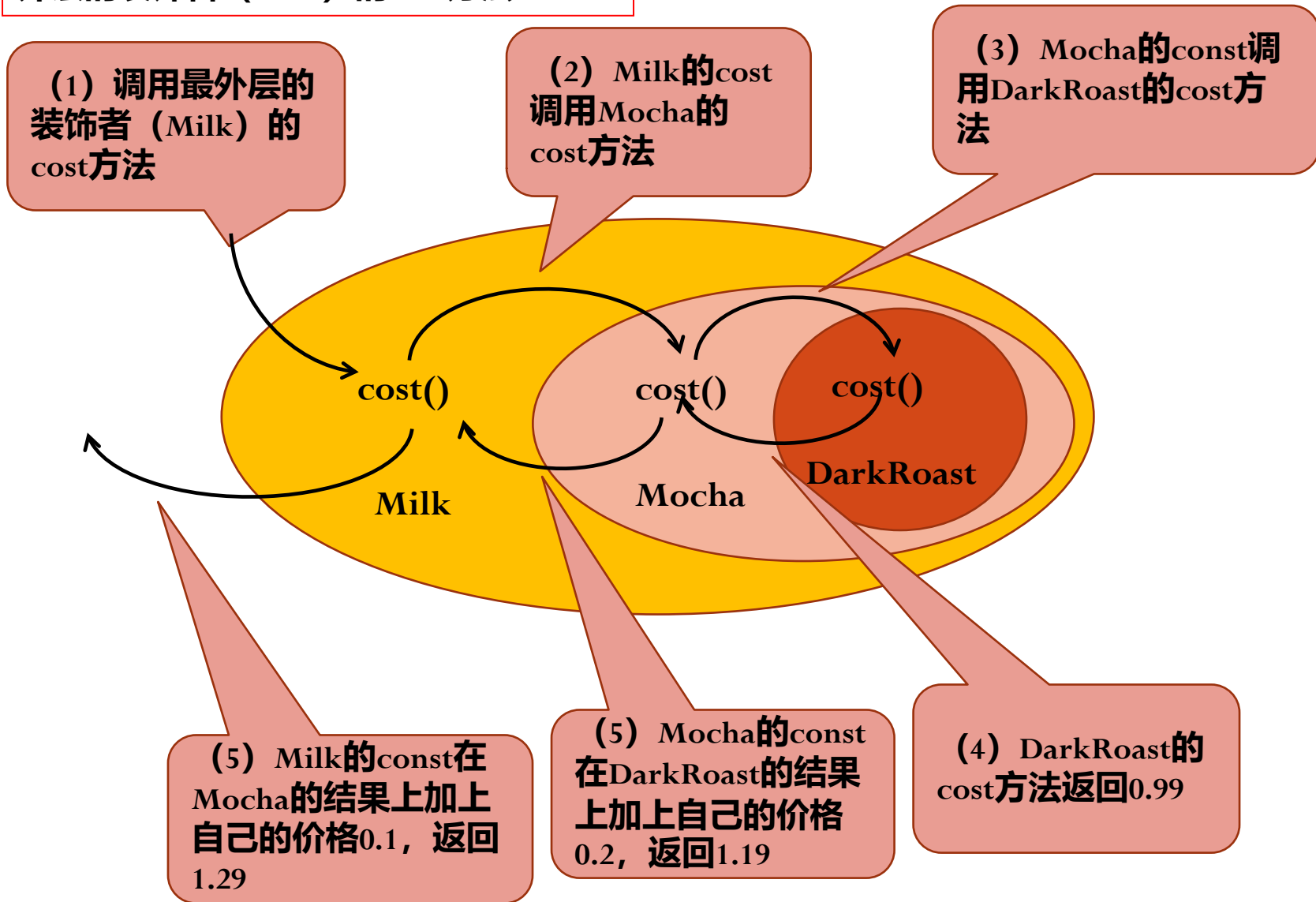
1 以DarkRoast对象开始

2 顾客想要Mocha, 所以
建立Mocha装饰对象, 并用
它将DarkRoast对象包
(wrap) 起来

3 顾客想要Milk, 所以
建立Milk装饰对象, 并用它将
Mocha对象包 (wrap) 起来



4 该是为顾客算钱的时候了。通过调用最外层的装饰者 (Milk) 的cost方法



认识装饰者模式

- 好了，这是目前所知道的一切

- 装饰者和被装饰者有相同的超类型

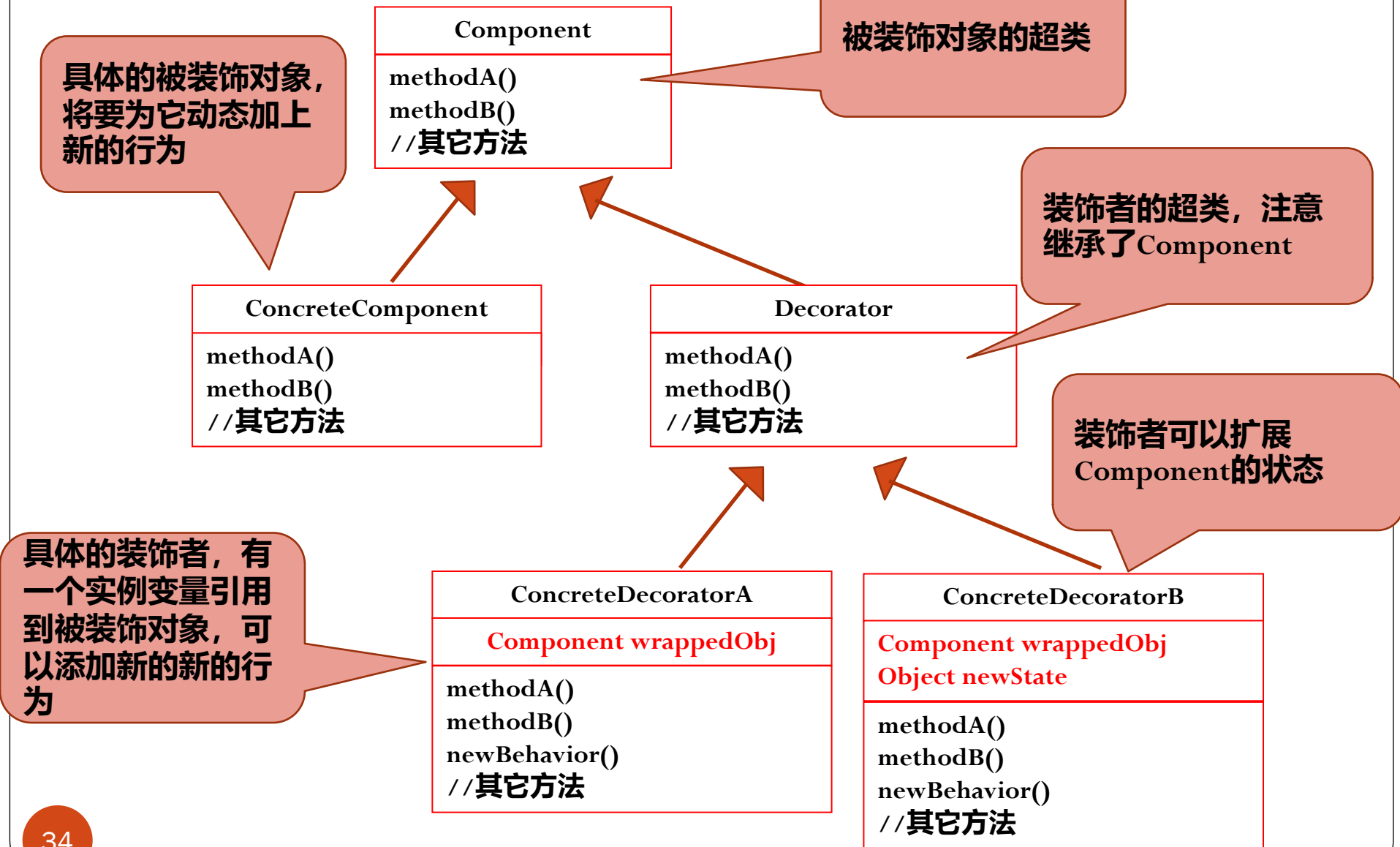
- 可以用一个或多个装饰者装饰一个对象

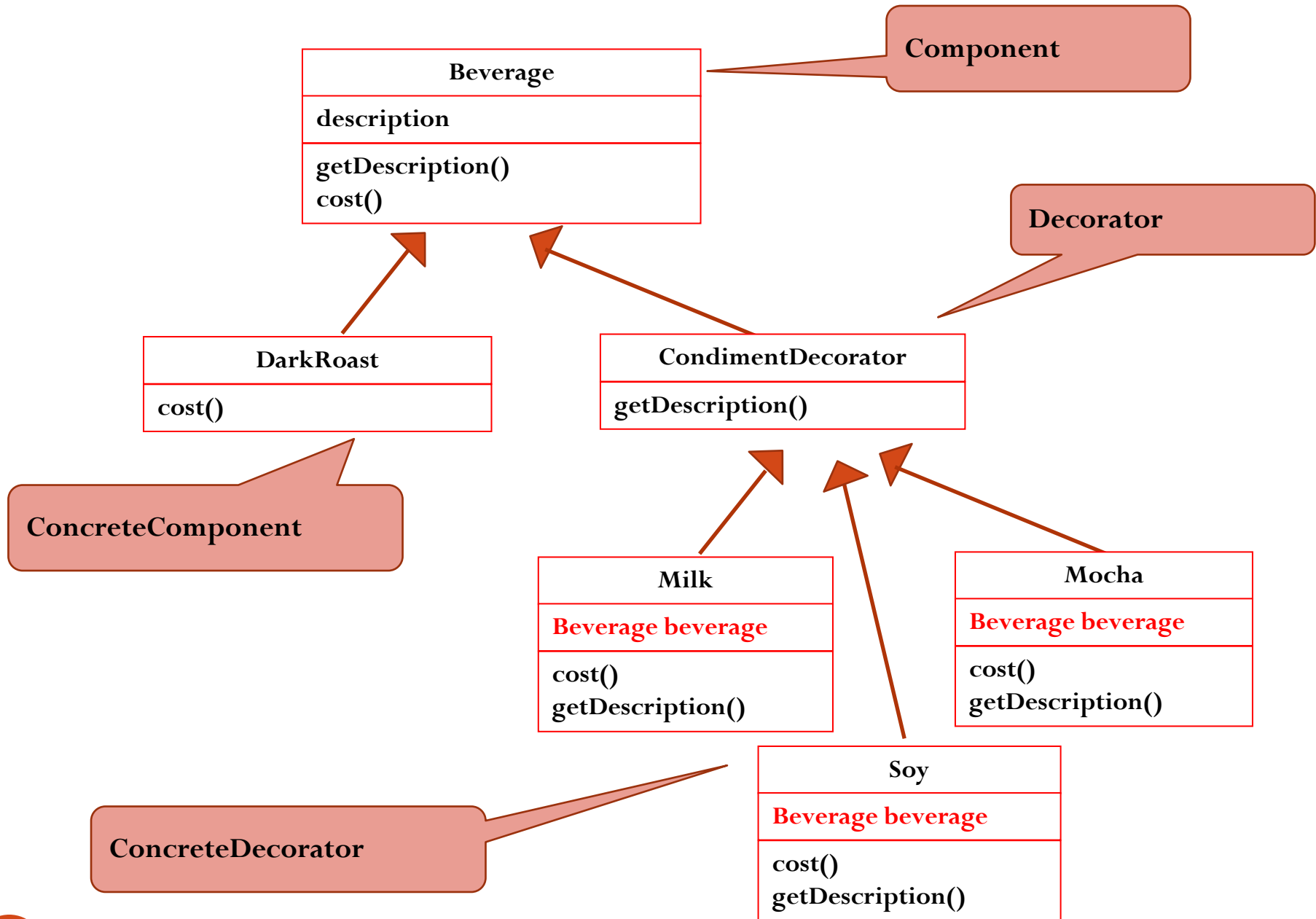
- 既然装饰者和被装饰者有相同的超类型，所以在任何需要原始对象（被装饰的）场合，可以用装饰过的对象代替它

- 装饰者可以在调用被装饰者的行为之前/之后，加上自己的行为

- 对象可以在任何时候被装饰，所以可以在运行时动态地、不限量地用你喜欢的装饰者来装饰

装饰 (Decorator) 模式的结构





把设计变成真正的代码的时候到了

```
public abstract class Beverage{  
    String description = "Unknown Beverage" ;
```

```
    public String getDescription(){  
        return description;  
    }
```

给出了getDescription的实现

```
    public abstract double cost();  
}
```

必须在子类实现

```
public abstract class CondimentDecorator extends Beverage{  
    public abstract String getDescription();  
}
```

必须在具体装饰者对象实现子类实现

把设计变成真正的代码的时候到了

```
public class DarkRoast extends Beverage{  
    public DarkRoast (){  
        description = "Dark Roast Coffee";  
    }  
}
```

具体的饮料类（被装饰对象）

```
    public double cost(){  
        return 0.99;    //返回饮料的价格  
    }  
}
```

```
public class HouseBlend extends Beverage{  
    public HouseBlend (){  
        description = "House Blend Coffee";  
    }  
}
```

具体的饮料类（被装饰对象）

```
    public double cost(){  
        return 0.89;    //返回饮料的价格  
    }  
}
```

把设计变成真正的代码的时候到了

具体的调料Mocha
(装饰者)

```
public class Mocha extends CondimentDecorator {
```

```
    Beverage beverage;
```

beverage实例变量记录被装饰对象

```
    public Mocha(Beverage beverage){  
        this.beverage = beverage;  
    }
```

利用构造函数参数，
传入被装饰对象

```
    public String getDescription(){  
        return beverage.getDescription() + ",Mocha"  
    }
```

希望描述信息不仅描述
饮料(如Dark Roast)，而
且还能包括调料。所以
利用委托，先得到被包
装对象的描述，再加上
装饰者自己的描述

```
    public double cost(){  
        return 0.20 + beverage.cost();  
    }
```

装饰者自己的价格+
被装饰对象的价格

把设计变成真正的代码的时候到了

具体的调料Milk (装饰者)

```
public class Milk extends CondimentDecorator {  
  
    Beverage beverage;  
  
    public Milk(Beverage beverage){  
        this.beverage = beverage;  
    }  
  
    public String getDescription(){  
        return beverage.getDescription() + ",Milk"  
    }  
  
    public double cost(){  
        return 0.1 + beverage.cost(); //返回饮料的价格  
    }  
}
```

把设计变成真正的代码的时候到了

具体的调料Soy (装饰者)

```
public class Soy extends CondimentDecorator {  
  
    Beverage beverage;  
  
    public Soy(Beverage beverage){  
        this.beverage = beverage;  
    }  
  
    public String getDescription(){  
        return beverage.getDescription() + ",Soy"  
    }  
  
    public double cost(){  
        return 0.3 + beverage.cost(); //返回饮料的价格  
    }  
}
```


把设计变成真正的代码的时候到了

```
public class StartBuzzCoffee {  
  
    public static void main(String args[]){  
        Bevegare beverage = new DarkRoast(); //订一杯咖啡, 不叫调料  
  
        System.out.println(beverage.getDescription() + " $" + beverage.cost());  
  
        beverage = new Mocha(beverage);    //加调料Mocha  
  
        beverage = new Milk(beverage);      //加调料Milk  
  
        beverage = new Soy(beverage);       //加调料Soy  
  
        System.out.println(beverage.getDescription() + " $" + beverage.cost());  
    }  
}  
  
Dark Roast Coffee $0.99  
Dark Roast Coffee, Mocha, Milk, Soy $1.49
```

装饰 (Decorator) 模式的评价

- 使用Decorator模式可以很容易地向对象添加职责。
- 使用Decorator模式可以很容易地重复添加一个特性，而两次继承则极易出错
- 避免在层次结构高层的类有太多的特征：可以从简单的部件组合出复杂的功能。具有低依赖性和低复杂性
- 缺点：Decorator与Component不一样；有许多小对象

JAVA API中的装饰者模式

I/O流的分层

- 流能以套接管线的方式互相连接起来，一个用于输出的数据流同时可以是另一个用于输入的流，形成所谓的过滤流。方法是将一个的流对象传递给另一个流的构造函数。

JAVA API中的装饰者模式

I/O流的分层

- 例如，为了从二进制一个文件里读取数字（假设该二进制文件内容全是浮点数），首先要创建FileInputStream，然后将其传递给DataInputStream的构造方法。

```
InputStream fin =  
    new FileInputStream("data.dat");  
  
InputStream din =  
    new DataInputStream(fin);  
  
double s = din.readDouble();
```

JAVA API中的装饰者模式

I/O流的分层

- Reader inFromUser =

```
new BufferedReader(new InputStreamReader(System.in));
```

等价于：

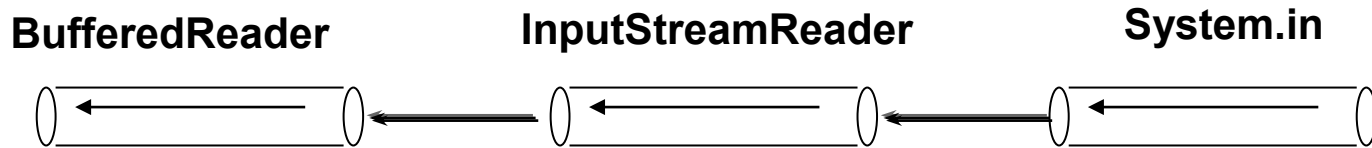
```
Reader ins = System.in;
```

```
Reader insReader =
```

```
new InputStreamReader(ins);
```

```
Reader inFromUser =
```

```
new BufferedReader (insReader )
```



JAVA IO 流的这种机制如何实现：装饰者设计模式（decorator pattern）

JAVA API实现大量用到了设计模式：比如swing的GUI事件处理采用了观察者模式（Observer Pattern）

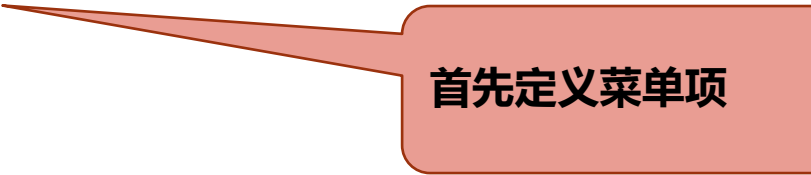
迭代模式的由来

- ❑ 将对象职责分离，最大限度减少彼此之间的耦合程度，从而建立一个松散耦合的对象网络
- ❑ 集合对象拥有两个职责：一是存储内部数据；二是遍历内部数据。从依赖性看，前者为对象的根本属性，而后者既是可变化的，又是可分离的。可将遍历行为分离出来，抽象为一个迭代器，专门提供遍历集合内部数据对象行为。这是迭代子模式的本质

迭代模式的由来

□ 考虑煎饼屋（PancakeHouse）和餐厅（Diner）的菜单

```
public class MenuItem {  
    String name;  
    String description;  
    boolean vegetarian;  
    double price;  
    public MenuItem(String name, String description,  
                     boolean vegetarian, double price) {  
        this.name = name;  
        this.description = description;  
        this.vegetarian = vegetarian;  
        this.price = price;  
    }  
    public String getName()    { return name;    }  
    public String getDescription()    { return description; }  
    public double getPrice()    { return price;  }  
    public boolean isVegetarian()    { return vegetarian; }
```



首先定义菜单项

迭代器模式的由来

煎饼屋菜单

```
public class PancakeHouseMenu {
```

```
    ArrayList menuItems;
```

```
    public PancakeHouseMenu(){
```

```
        menuItems = new ArrayList();
```

```
        addItem("Pancake1","Pancake1",true,2.99);
```

```
        addItem("Pancake2","Pancake2",false,2.99);
```

```
        addItem("Pancake3","Pancake3",true,3.49);
```

```
        addItem("Pancake4","Pancake4",true,3.59);
```

```
    }
```

```
    public void addItem(String name, String description,
```

```
        boolean vegetarian, double price){
```

```
        MenuItem item = new MenuItem(name,description,
```

```
            vegetarian,price);
```

```
        menuItems.add(item);
```

```
    }
```

```
    public ArrayList getMenuItems(){ return menuItems; }
```

用ArrayList存储菜单项

迭代器模式的由来

餐厅菜单

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    public DinerMenu(){
        menuItems = new MenuItem[MAX_ITEMS];
        addItem("Diner1","Diner1",true,2.99);
        addItem("Diner2","Diner2",false,2.99);
    }
    public void addItem(String name,String description,
                        boolean vegetarian,double price){
        MenuItem item = new MenuItem(name,description,vegetarian,price);
        if(numberOfItems >= MAX_ITEMS){
            System.out.println("Sorry,menu is full!");
        }
        else{
            menuItems[numberOfItems] = item;
            numberOfItems ++;
        }
    }
    public MenuItem[] getMenuItems(){ return menuItems; }
```

用数组存储菜单项

❑ 现在煎饼屋和餐厅合并了，存在二种菜单。如果女服务员想打印菜单怎么办

```
public class Waitress {  
    public void printMenu(){  
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();  
        ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();  
  
        DinerMenu dinerMenu = new DinerMenu();  
        MenuItem[] lunchItems = dinerMenu.getMenuItems();  
  
        for(int i = 0; i < breakfastItems.size(); i++){  
            MenuItem menuItem = (MenuItem)breakfastItems.get(i);  
            System.out.print(menuItem.getName() + " ");  
            System.out.println(menuItem.getPrice() + " ");  
            System.out.println(menuItem.getDescription());  
        }  
        for(int i = 0; i < lunchItems.length; i++){  
            MenuItem menuItem = lunchItems[i];  
            System.out.print(menuItem.getName() + " ");  
            System.out.println(menuItem.getPrice() + " ");  
            System.out.println(menuItem.getDescription());  
        }  
    }  
}
```

由于二种菜单存储在不同的集合里，因此必须写二个循环，分别打印。

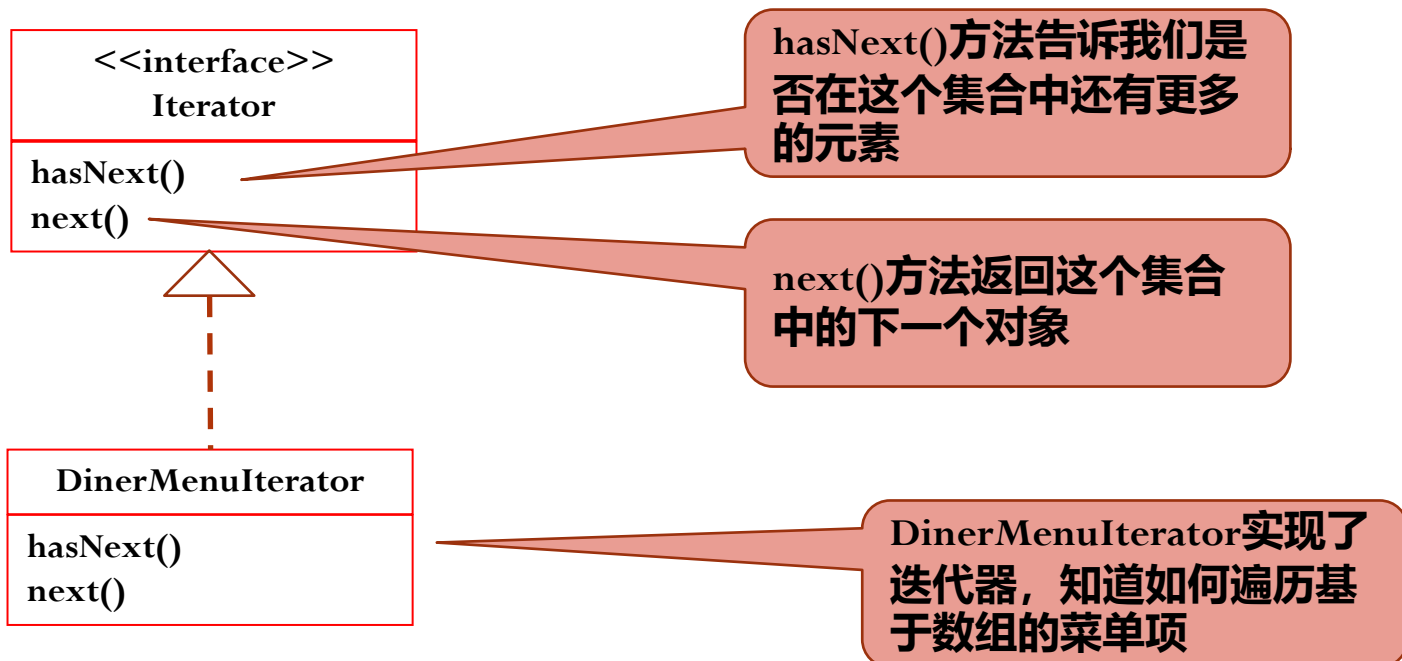
如果增加一个菜单，放在第三种集合里，还得增加一个循环

□ 这种实现的问题

- 针对具体类编程，不是针对接口
- 女招待需要知道每种菜单如何表达内部菜单项的集合，违反了封装
- 如果增加新的菜单，而且新菜单用别的集合来保存菜单项，我们又要修改女招待的代码
- 由于这二种集合遍历元素以及取元素的方法不同，导致必须写二个循环。但二个循环代码有重复
- 能不能找到一种统一的方式来遍历集合中的元素，使得只要一个循环就可以打印不同的菜单

初见迭代器模式

- 关于迭代器模式，我们需要知道的第一件事，就是它依赖于一个名为迭代器的接口



- 一旦我们有了迭代器接口，就可以为各种对象集合实现迭代器：数组，链表、Hash表，...

迭代器模式的由来

```
public class DinerMenuIterator implements Iterator {
```

```
    MenuItem[] items;
```

```
    int position = 0;
```

position记录当前遍历的位置，
items是集合对象

```
    public DinerMenuIterator(MenuItem[] items){
```

```
        this.items = items;
```

```
    }
```

通过构造函数，传递
集合对象给迭代器

```
    public boolean hasNext() {
```

```
        if(position >= items.length || items[position] == null)
```

```
            return false;
```

```
        else
```

```
            return true;
```

```
    }
```

因为使用数组（长度固定），所以不仅要检查position是否超出数组长度，还必须检查下一项是否为null，如果为null，表示没有下一项了

```
    public Object next() {
```

```
        MenuItem item = items[position];
```

```
        position ++;
```

```
        return item;
```

返回数组内的下一项，并递增当前遍历的位置

迭代器模式的由来

```
public class PancakeHouseMenuIterator implements Iterator {  
    ArrayList items;  
    int position = 0;
```

position记录当前遍历的位置,
items是集合对象

```
    public PancakeHouseMenuIterator(ArrayList items){  
        this.items = items;  
    }  
}
```

通过构造函数, 传递
集合对象给迭代器

```
    public boolean hasNext() {  
        if(position >= items.size())  
            return false;  
        else  
            return true;  
    }  
}
```

```
    public Object next() {  
        MenuItem item = (MenuItem)items.get(position);  
        position ++;  
        return item;  
    }  
}
```

□ 现在改写餐厅菜单

```
public class DinerMenu {  
    static final int MAX_ITEMS = 6;  
    int numberOfItems = 0;  
    MenuItem[] menuItems;  
  
    public DinerMenu(){  
        ...  
    }  
    public void addItem(String name,String description,  
                        boolean vegetarian,double price){  
        ...  
    }  
    public MenuItem[] getMenuItems(){ return menuItems; }  
  
    public Iterator createIterator(){  
        return new DinnerMenuIterator(menuItems);  
    }  
}
```

返回迭代器接口。客户不需要知道餐厅菜单如何维护菜单项，也不需要知道迭代器如何实现，只需直接使用迭代器遍历菜单项

□ 现在改写餐厅菜单

```
public class DinerMenu {  
    static final int MAX_ITEMS = 6;  
    int numberOfItems = 0;  
    MenuItem[] menuItems;  
  
    public DinerMenu(){  
        ...  
    }  
    public void addItem(String name,String description,  
                        boolean vegetarian,double price){  
        ...  
    }  
    public MenuItem[] getMenuItems(){ return menuItems; }  
  
    public Iterator createIterator(){  
        return new DinnerMenuIterator(menuItems);  
    }  
}
```

返回迭代器接口。客户不需要知道餐厅菜单如何维护菜单项，也不需要知道迭代器如何实现，只需直接使用迭代器遍历菜单项

□ 现在改写煎饼屋菜单

```
public class PancakeHouseMenu {  
    ArrayList menuItems;  
  
    public PancakeHouseMenu(){  
        ...  
    }  
    public void addItem(String name, String description,  
        boolean vegetarian, double price){  
        ...  
    }  
    public ArrayList getMenuItems(){ return menuItems; }  
  
    public Iterator createIterator(){  
        return new PancakeHouseMenuIterator(menuItems);  
    }  
}
```

返回迭代器接口。客户不需要知道餐厅菜单如何维护菜单项，也不需要知道迭代器如何实现，只需直接使用迭代器遍历菜单项

□ 现在改写女招待代码

```
public class Waitress {  
    PancakeHouseMenu pancakeHouseMenu  
    DinerMenu dinerMenu;  
    public Waitress(PancakeHouseMenu pm, DinerMenu dm){  
        pancakeHouseMenu = pm; dinerMenu = dm;  
    }  
    public void printMenu(){  
        Iterator pancakeIt = pancakeHouseMenu.createIterator();  
        Iterator dinerIt = dinerMenu.createIterator();  
        printMenu(pancakeIt );  
        printMenu(dinerIt);  
    }  
    public void printMenu(Iterator it){  
        while(it.hasNext()){  
            MenuItem menuitem = (MenuItem)it.next();  
            System.out.print(menuitem.getName() + " ");  
            System.out.println(menuitem.getPrice() + " ");  
            System.out.println(menuitem.getDescription());  
        }  
    }  
}
```

分别得到二个菜单的迭代器

现在的printMenu函数利用迭代器来遍历集合中的元素，而不用关心集合是如何组织元素的

□ 到目前为止，我们做了什么

□ 菜单的实现已经被封装起来了。女招待不知道每种菜单如何存储内部菜单项的

□ 只要实现迭代器，我们只需要一个循环，就可以多态地处理任何项的集合

□ 女招待现在只使用一个接口(迭代器)

□ 但女招待仍然捆绑于二个具体的菜单类，需要修改下。
通过定义Menu接口

```
public interface Menu{  
    public Iterator createIterator();  
}
```

□ 定义Menu接口，对二个菜单类做简单的修改

```
public interface Menu{  
    public Iterator createIterator();  
}
```

```
public class DinerMenu implements Menu{  
    //其他不变  
    public Iterator createIterator(){  
        return new DinnerMenuIterator(menuItems);  
    }  
}
```

```
public class PancakeHouseMenu implements Menu{  
    //其他不变  
    public Iterator createIterator(){  
        return new PancakeHouseMenuIterator (menuItems);  
    }  
}
```

□ 现在女招待不再捆绑到具体类了

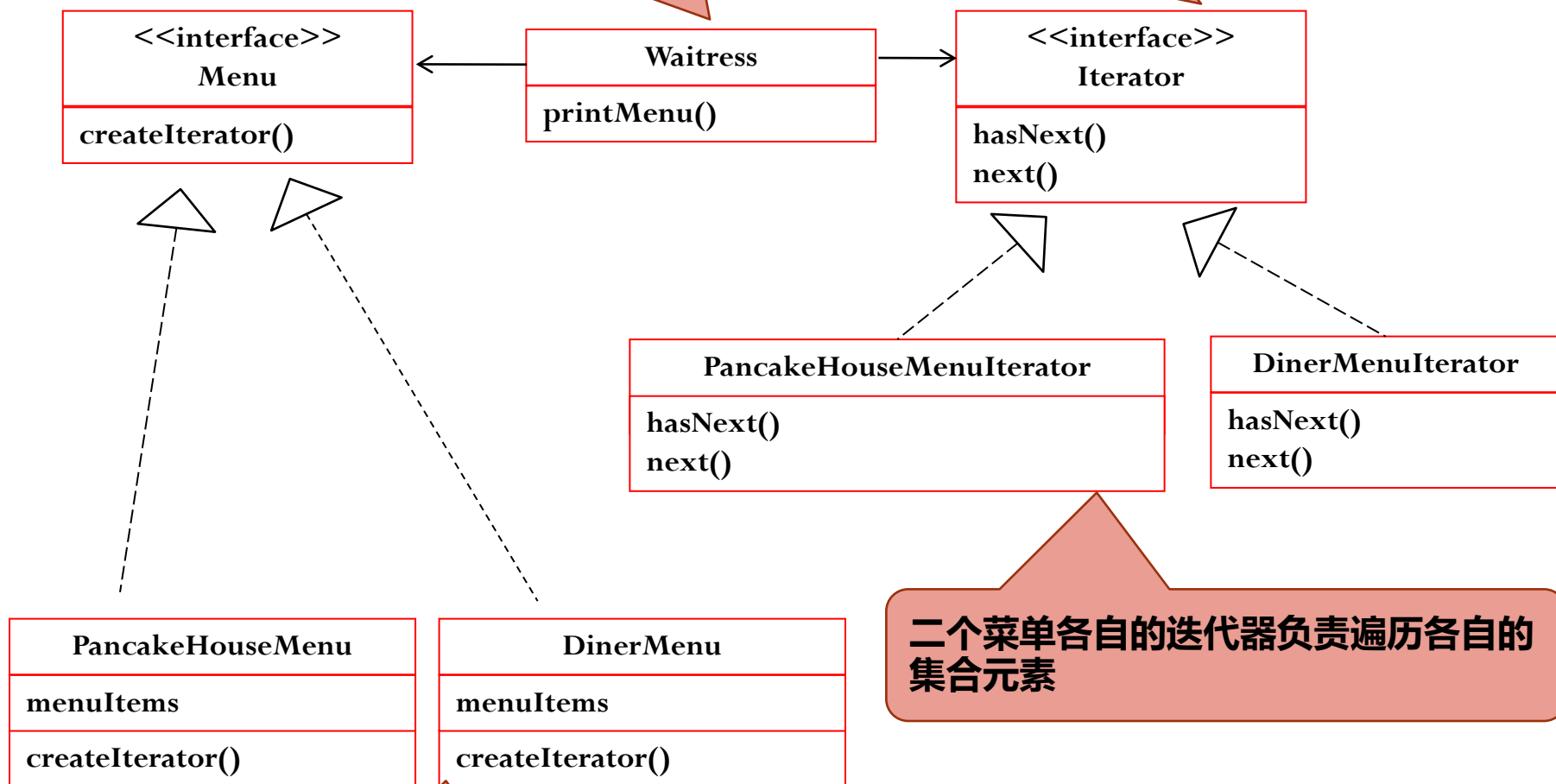
```
public class Waitress {  
    Menu pancakeHouseMenu  
    Menu dinerMenu;  
    public Waitress(Menu pm, Menu dm){  
        pancakeHouseMenu = pm; dinerMenu = dm;  
    }  
    public void printMenu(){  
        Iterator pancakeIt = pancakeHouseMenu.createIterator();  
        Iterator dinerIt = dinerMenu.createIterator();  
        printMenu(pancakeIt );  
        printMenu(dinerIt);  
    }  
    public void printMenu(Iterator it){  
        while(it.hasNext){  
            MenuItem menuItem = (MenuItem)it.next();  
            //打印MenuItem的内容  
        }  
    }  
}
```

□ 测试我们的代码

```
public class MenuDrive{  
    public void main(String args){  
        Menu pMenu = new PancakeHouseMenu();  
        Menu dMenu = new DinerMenu();  
  
        Waitress waitress = new Waitress(pMenu,dMenu);  
        waitress.printMenu();  
    }  
}
```

女招待只关心菜单和迭代器接口

女招待从菜单实现解耦，现在利用迭代器遍历菜单项，无须知道菜单具体实现



二个菜单各自的迭代器负责遍历各自的集合元素

二个菜单都实现了 `createIterator` 方法，返回各自的迭代器

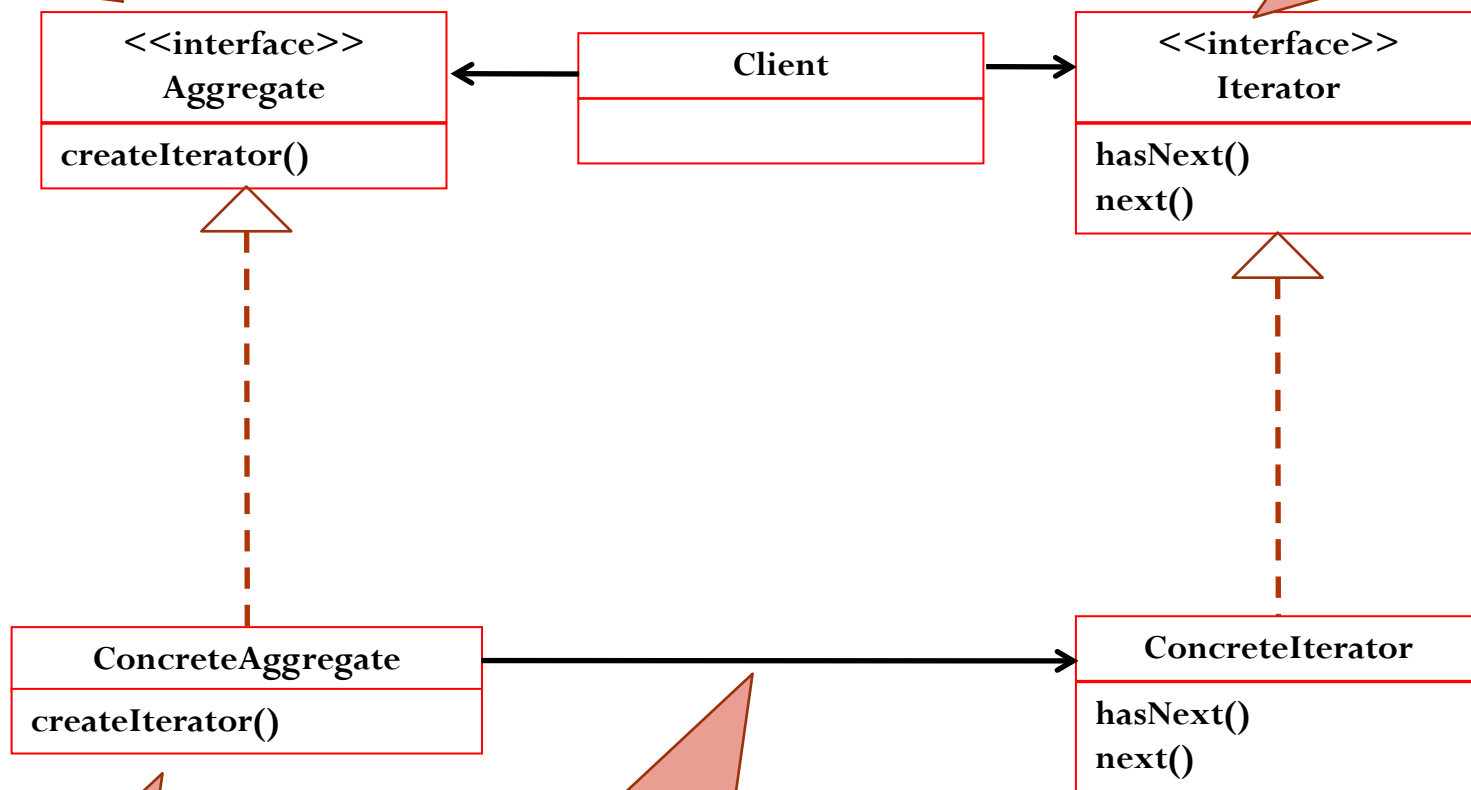
定义迭代器模式

- ❑ 迭代器模式提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示
- ❑ 把遍历元素的任务从聚合对象剥离出来，放到迭代器上。这样简化了聚合的接口和实现。也让任务各得其所
- ❑ 职责分离原则！
 - ❑ 一个类一个职责
 - ❑ 一个职责意味着一个变化的可能
 - ❑ 类的职责多了，意味着变化的可能多了，类被修改的可能性就大

迭代器模式的结构

共同的接口供所有聚合使用

所有迭代器必须实现的接口。接口中的方法可以遍历集合元素。
注意java.util.Iterator接口还定义了remove方法



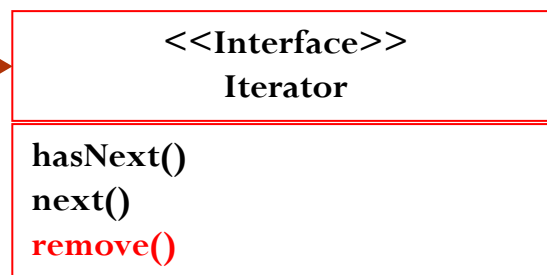
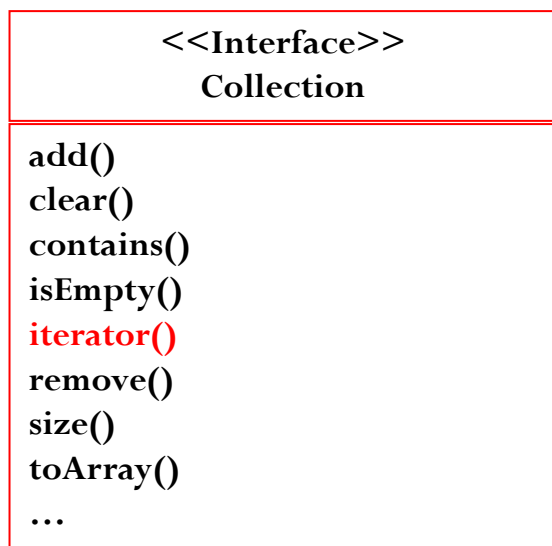
具体聚合都负责实例化一个具体迭代器

具体聚合都持有一个对象集合

具体迭代器负责遍历具体聚合对象的元素

JAVA中的迭代器模式

- JAVA中的集合类都实现了java.util.Collection接口，这个接口中定义了一个iterator()方法，返回java.util.Iterator接口



Java.util.Iterator接口还定义了remove()方法，删除由next()方法返回的最后一个元素
前面我们自己定义的Iterator接口同样可以定义该方法，并在具体的迭代器去实现该方法

因此JAVA的集合类对象直接用iterator()方法就可以返回JAVA的迭代器,而无需自己定义迭代器

单件 (Singleton) 模式的由来

- ◆ 单件模式的实例较为普遍:
 - ◆ 系统中只能有一个窗口管理器
 - ◆ 系统中只能有一个文件系统
 - ◆ 一个数字滤波器只能有一个A/D转换器
 - ◆ 一个会计系统只能专用于一个公司
 - ◆ 一个系统只有一个日志对象
 - ◆ 一个系统只有一个注册表
 - ◆ 一个系统只有一个线程池
 - ◆ . . .

单件模式的由来

- ◆ 如何才能保证一个类只有一个实例并且这个实例易于被访问呢？一个全局变量使得一个对象可以被访问，但它不能防止你实例化多个对象

单件模式的由来

- ◆ 如何做到一个类只有一个实例
- ◆ 如果一个类的构造函数是公有的，你是无法控制该类的使用者多次new出该类的多个实例
- ◆ 如果一个类的构造函数是保护的，无法控制该类的派生类多次new出该类的多个实例

单件模式的由来

- ◆ 因此该类的构造函数必须是私有的

```
public class MyClass {  
    private MyClass() { }  
}
```

- ◆ 因为只有MyClass类的实例才能调用构造函数，但是又没有其他类能够实例化MyClass，因此我们得不到这样的实例。这似乎是一个“鸡生蛋、蛋生鸡”的问题

单件模式的由来

- ◆ 一个更好的办法是，让类自身负责保存它的唯一实例。这个类可以保证没有其他实例可以被创建，并且它可以提供一个访问该实例的方法
- ◆ 如何做到？利用静态变量和静态方法

单件模式的由来

利用私有静态变量来记录唯一实例

```
public class Singleton{
```

```
    private static Singleton instance = null;
```

```
    private Singleton() {}
```

私有构造函数

```
    public static Singleton getInstance(){
```

公有静态方法实例化对象并返回实例

```
        if(instance == null){
            instance = new Singleton();
        }
        return instance;
```

如果instance为空，则实例化一个对象
并赋值给静态变量instance
否则直接返回instance
因此保证了构造函数只会被调用一次

```
    }  
    //其它的实例变量、实例方法  
    public void someMethod() {}
```


单件模式的由来

```
public class Client{  
    public static void main(String[] args){  
        Singleton instance1 = Singleton.getInstance();  
        Singleton instance2 = Singleton.getInstance();  
  
        System.out.println(instance1 == instance2); //true  
  
        instance1.someMethod();  
    }  
}
```

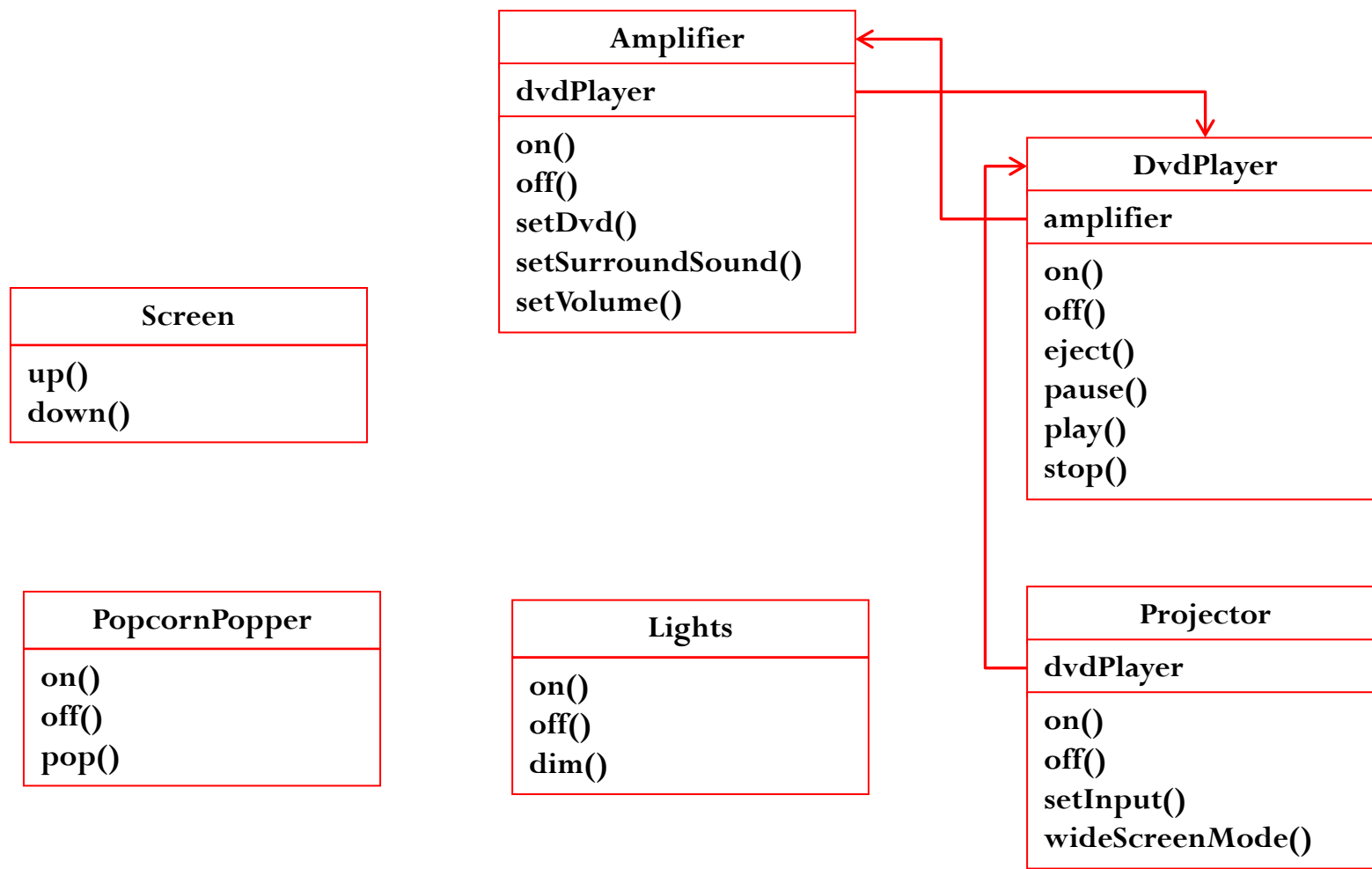
单件模式的意图和适用性

- ◆**意图：** 保证一个类仅有一个实例,并提供一个全局访问点
- ◆**适用场合：**
 - ◆当类只能有一个实例而且用户可以从一个众所周知的访问点访问它时

外观模式 (Facade pattern) 的由来

- ◆ 对于一个复杂系统提供的一群接口，外观模式提供了一个简单、高层的接口，让子系统更容易使用
- ◆ 考虑一个家庭影院系统，包括如下设备：
 - ◆ DVD播放器(DvdPlayer)
 - ◆ 功放 (Amplifier)
 - ◆ 投影仪(Projector)
 - ◆ 屏幕(Screen)
 - ◆ 灯光(Lights)
 - ◆ 爆米花机(PopCornPopper)

外观模式 (Facade pattern) 的由来



外观模式 (Facade pattern) 的由来

◆ 看一部DVD需要以下复杂操作

```
popper.on();  
popper.pop();
```

```
lights.dim(10);
```

```
screen.down();
```

```
projector.on();  
projector.setInput(dvd);  
projector.wideScreenMode();
```

```
amp.on()  
amp.setDvd(dvd)  
amp.setSurroundSound();  
amp.setVolume(5);
```

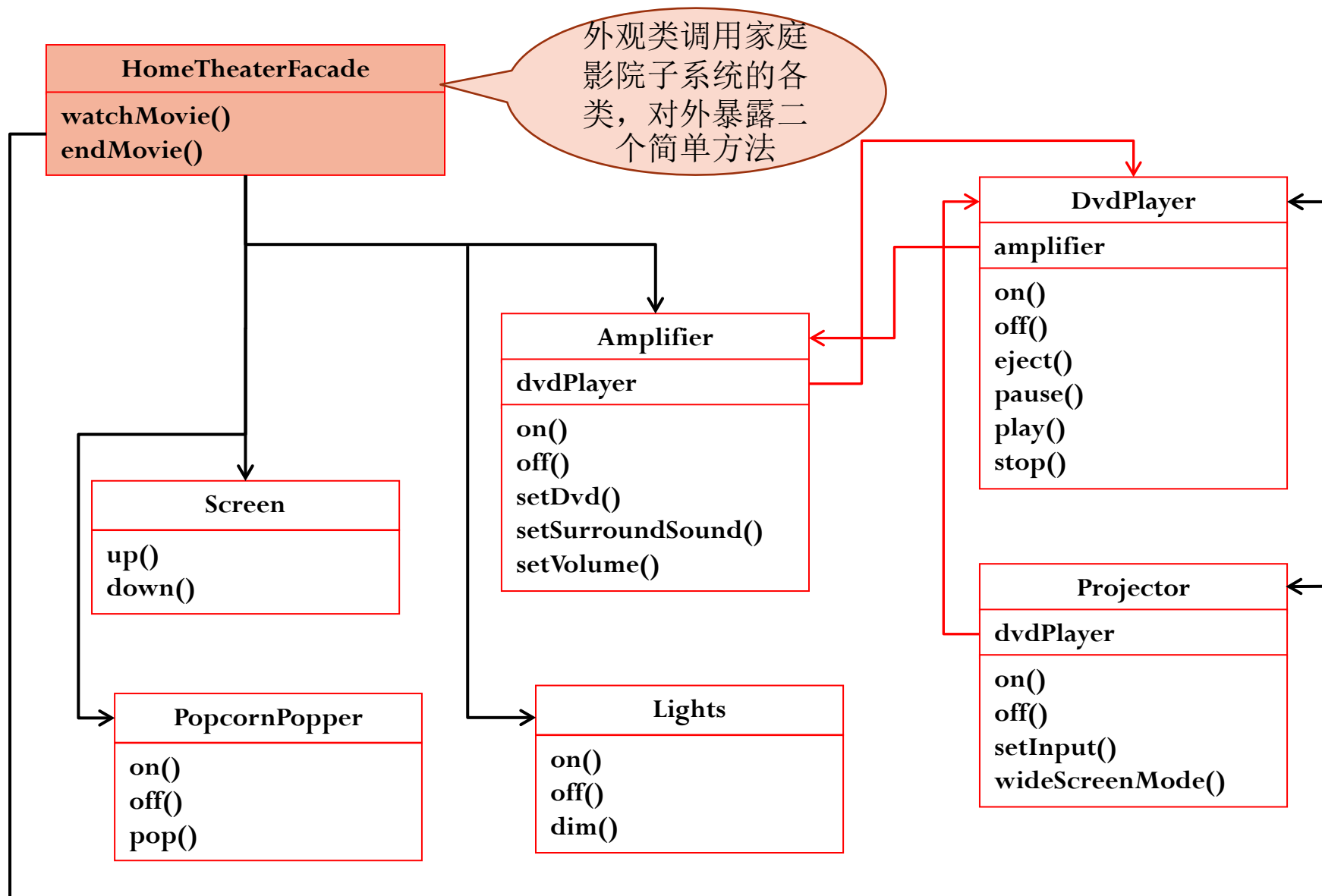
```
dvd.on();  
dvd.play(movie);
```

涉及到6个类的
不同方法

看一部电影这么复杂？
要是有个遥控器，按
一个“WatchMovie”
就帮我搞定这些事多
好

外观模式来了！，通过实现一个更合理接口的外观类，可以使得家庭影院这么一个复杂系统更容易使用

外观模式 (Facade pattern) 的由来



构造家庭影院的外观

```
public class HomeTheaterFacade{  
    Amplifier amp;  
    DvdPlayer dvd;  
    Projector projector;  
    Lights lights;  
    Screen screen;  
    PopcornPopper popper;  
    public HomeTheaterFacade(Amplifier amp, DvdPlayer dvd  
                             Projector projector, Lights lights  
                             Screen screen , PopcornPopper popper){  
  
        this.amp =amp;  
        this.dvd = dvd;  
        this.projector = projector;  
        this.lights = lights;  
        this.screen = screen;  
        this.popper = popper;  
    }  
}
```

这就是对象组合，会用到的子系统组件全都在这里

子系统中每一个组件的引用全部传入外观构造函数，然后把它们赋值给相应的引用变量

构造家庭影院的外观

```
public class HomeTheaterFacade{  
    public void watchMovie(String movie){  
        popper.on();  
        popper.pop();  
        lights.dim(10);  
        screen.down();  
        projector.on();  
        projector.setInput(dvd);  
        projector.wideScreenMode();  
        amp.on()  
        amp.setDvd(dvd)  
        amp.setSurroundSound();  
        amp.setVolume(5);  
        dvd.on();  
        dvd.play(movie);  
    }  
}
```

watchMovie将之前手动处理的任务依次处理。注意：每项任务都是委托给子系统中相应的组件处理的

构造家庭影院的外观

```
public class HomeTheaterFacade{  
    public void endMovie(){  
        popper.off();  
  
        lights.on;  
        screen.up();  
        projector.on();  
        projector.off();  
  
        amp.off()  
        dvd.stop();  
        dvd.eject();  
        dvd.off();  
    }  
}
```

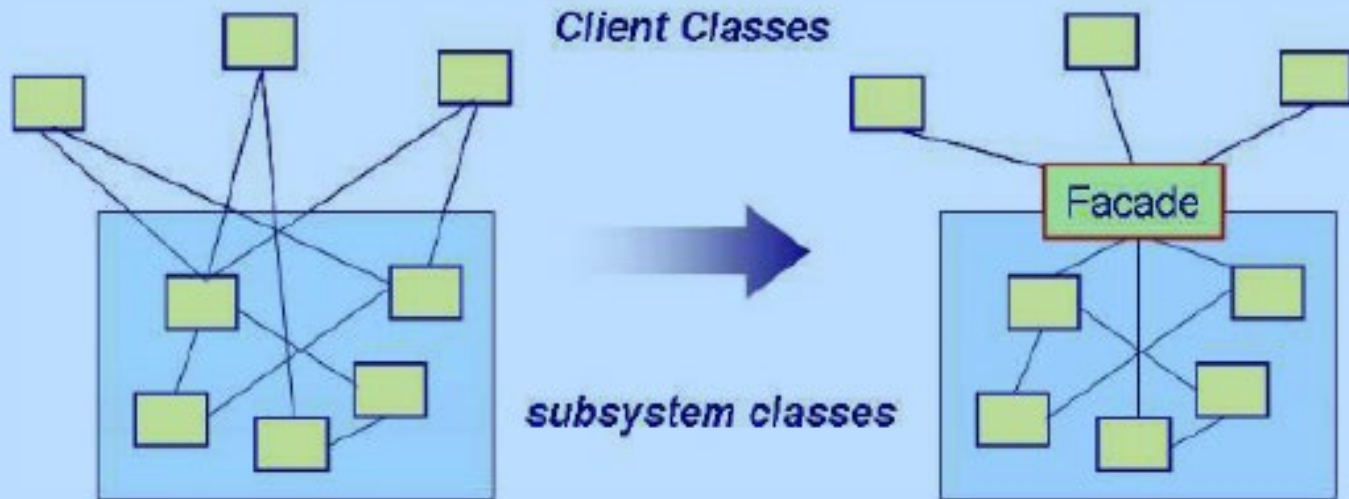
endMovie负责关闭一切。每项任务也都是委托给子系统中相应的组件处理的

现在用轻松的方式（通过外观观赏电影）

```
public class Client{  
    public static void main(String[] args){  
  
        //在这里实例化系统组件amp, dvd, ...  
  
        HomeTheaterFacade homeTheater =  
            new HomeTheaterFacade(amp,dvd,projector,  
                lights, screen, popper);  
  
        homeTheater.watchMovie("2012");  
        homeTheater.endMovie();  
    }  
}
```

在上面这个例子里，客户直接建立了外观对象。但实际应用中，某个外观对象会直接指派给客户使用，而不需要由用户自己创建外观对象

外观 (Facade) 模式的由来



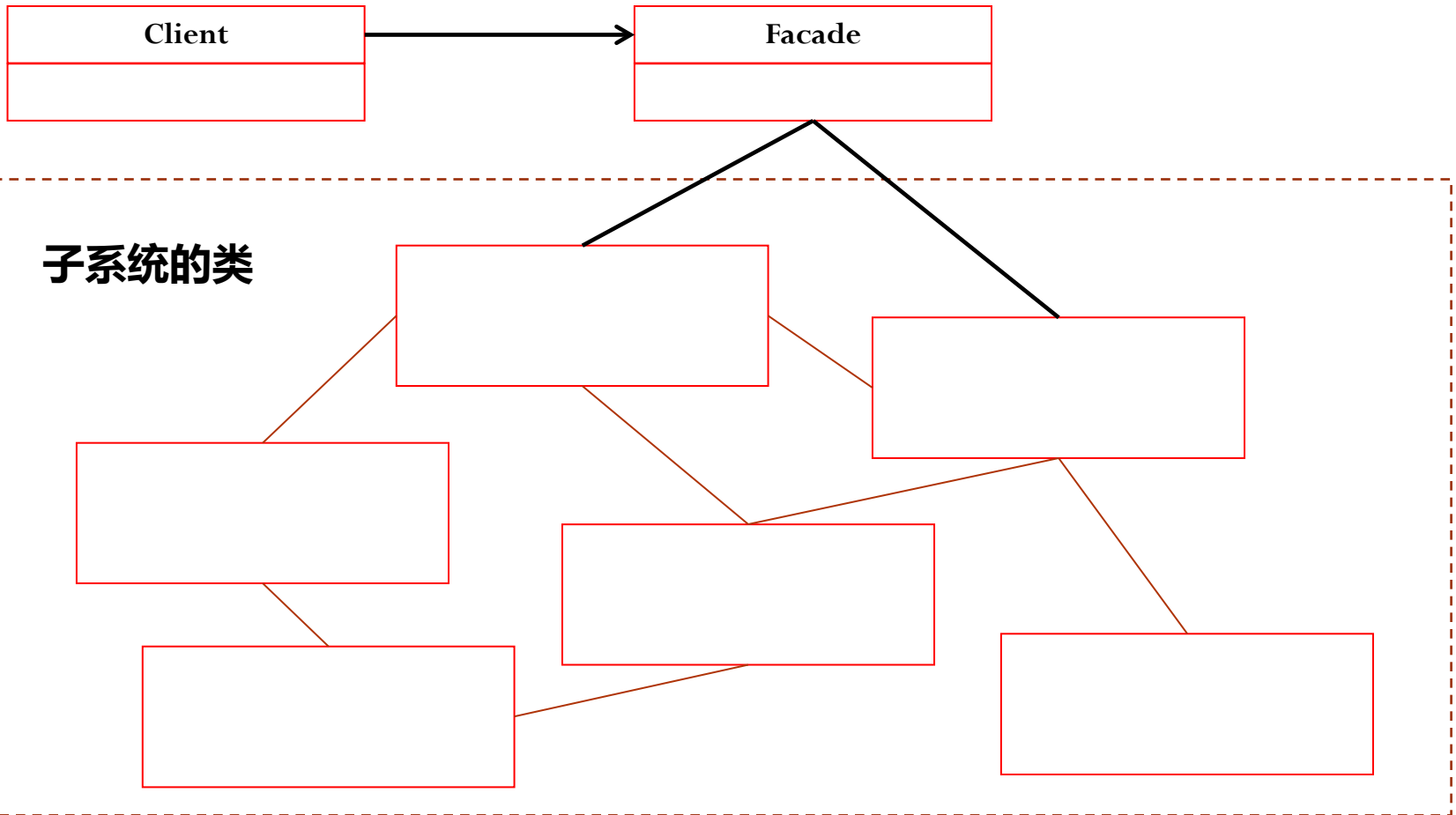
外观 (Facade) 模式的意图和适用性

- ❑ 提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用
- ❑ 解除客户程序与子系统具体实现部分的依赖性，避免了客户与子系统的紧耦合，有利于移植和更改
- ❑ 当需要构建层次结构的子系统时，使用Facade模式定义每层的入口点。如果子系统间相互依赖，他们只需通过Facade进行通讯
- ❑ 外观模式的本质是让接口变得更简单

外观 (Facade) 模式的结构

快乐的客户只依赖于简单的外观

外观实现了统一、简单的接口，易于使用

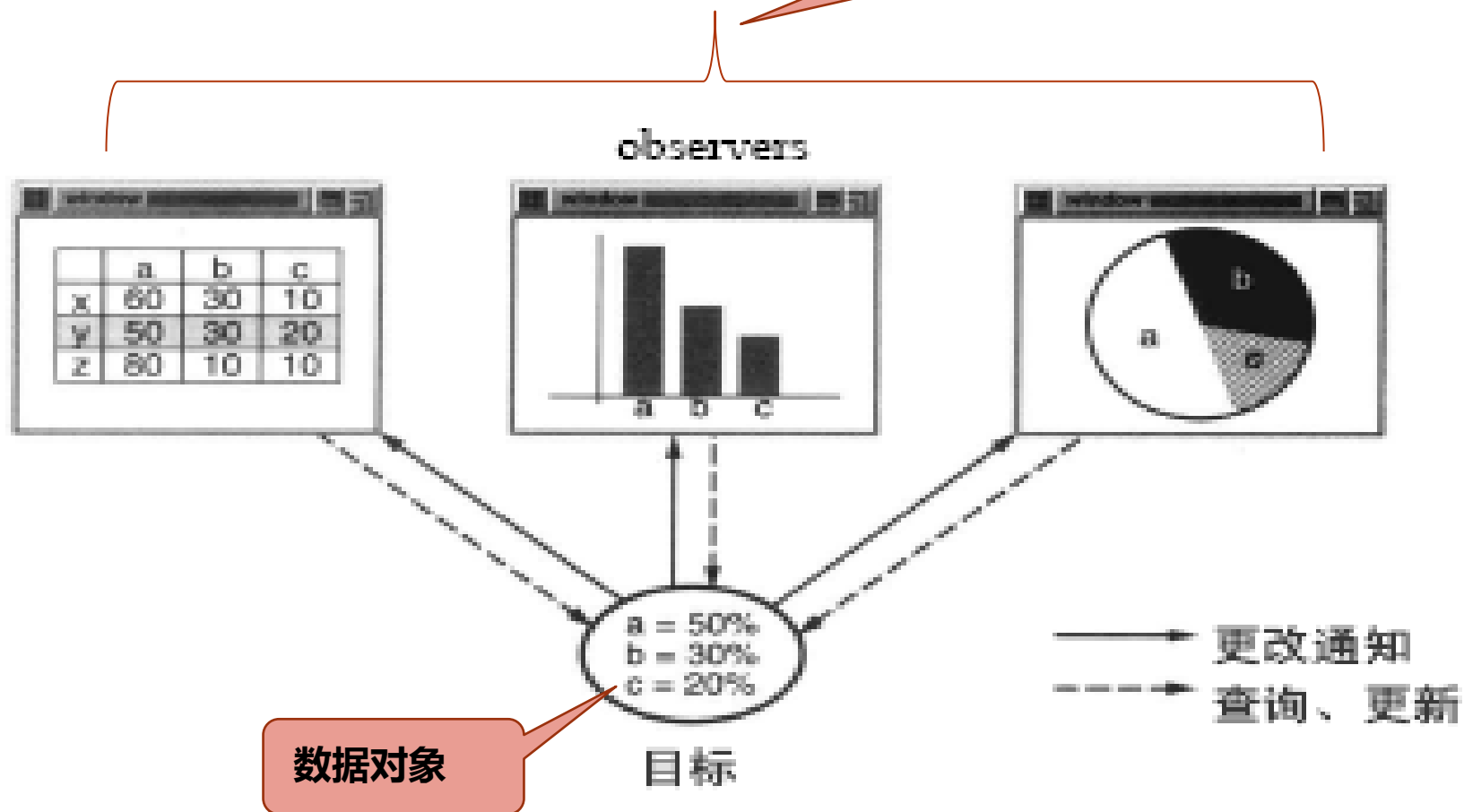


观察者模式的由来

- 将系统分割成一系列相互协作的类有也存在一些不足：
 - 需要维护相关对象间的一致性
 - 为维持一致性而使各类紧密耦合，可能降低其可重用性
- 没有理由限定依赖于某数据对象的对象数目，对相同的数据对象可以有任意数目的不同用户界面

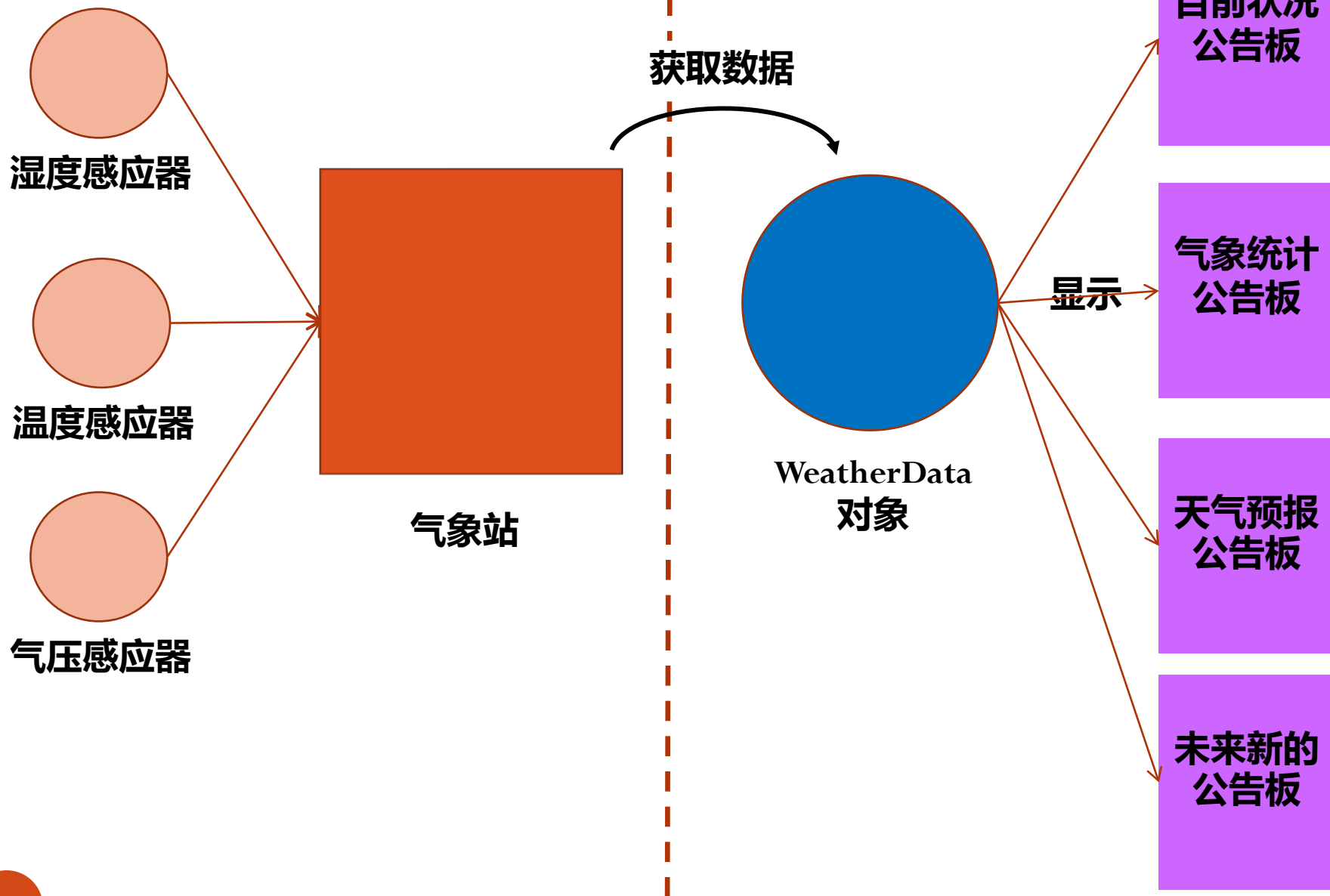
观察者模式的由来

对相同的数据对象，可以有不同
的展示方式（不同的观察者）



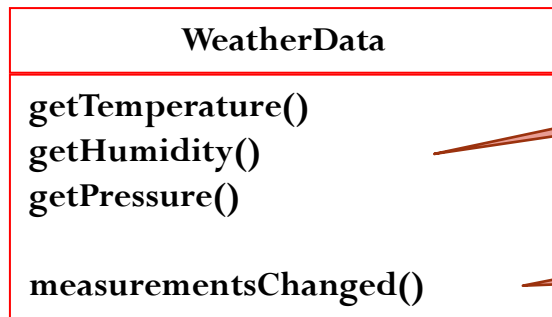
观察者模式的由来

- ❑ 某气象站通过湿度感应装置、温度感应装置、气压感应装置获取天气情况
- ❑ 实现了一WeatherData类，可以从气象站获得湿度、温度、气压数据
- ❑ 需要根据WeatherData提供的气象数据实时更新三个布告板：目前天气状况、气象统计和天气预报
- ❑ 系统必须是可扩展的，即未来可以任意加入、删除新的布告板



观察者模式的由来

□ WeatherData类应该具有的方法



三个getter方法可以取得三个测量值：温度、湿度、气压

一旦测量值被更新，此方法被调用去更新三个布告板

观察者模式的由来

□ WeatherData类的第一个实现

```
public class WeatherData {
```

```
//实例变量声明
```

```
public void measurementsChanged(){
```

```
    float temp = getTemperature();
```

```
    float humidity = getHumidity();
```

```
    float pressure = getPressure();
```

```
    currentConditionDisplay.update(temp,humidity,pressure);
```

```
    statisticsDisplay.update(temp,humidity,pressure);
```

```
    forecastDisplay.update(temp,humidity,pressure);
```

```
}
```

```
//其他方法
```

```
}
```

获得最新的测量值

更新三个公告板

针对公告板的具体实现编程
以后增加、删除公告板要修改代码，同时无法在运行时增加、删除公告板

update方法看起来像统一接口

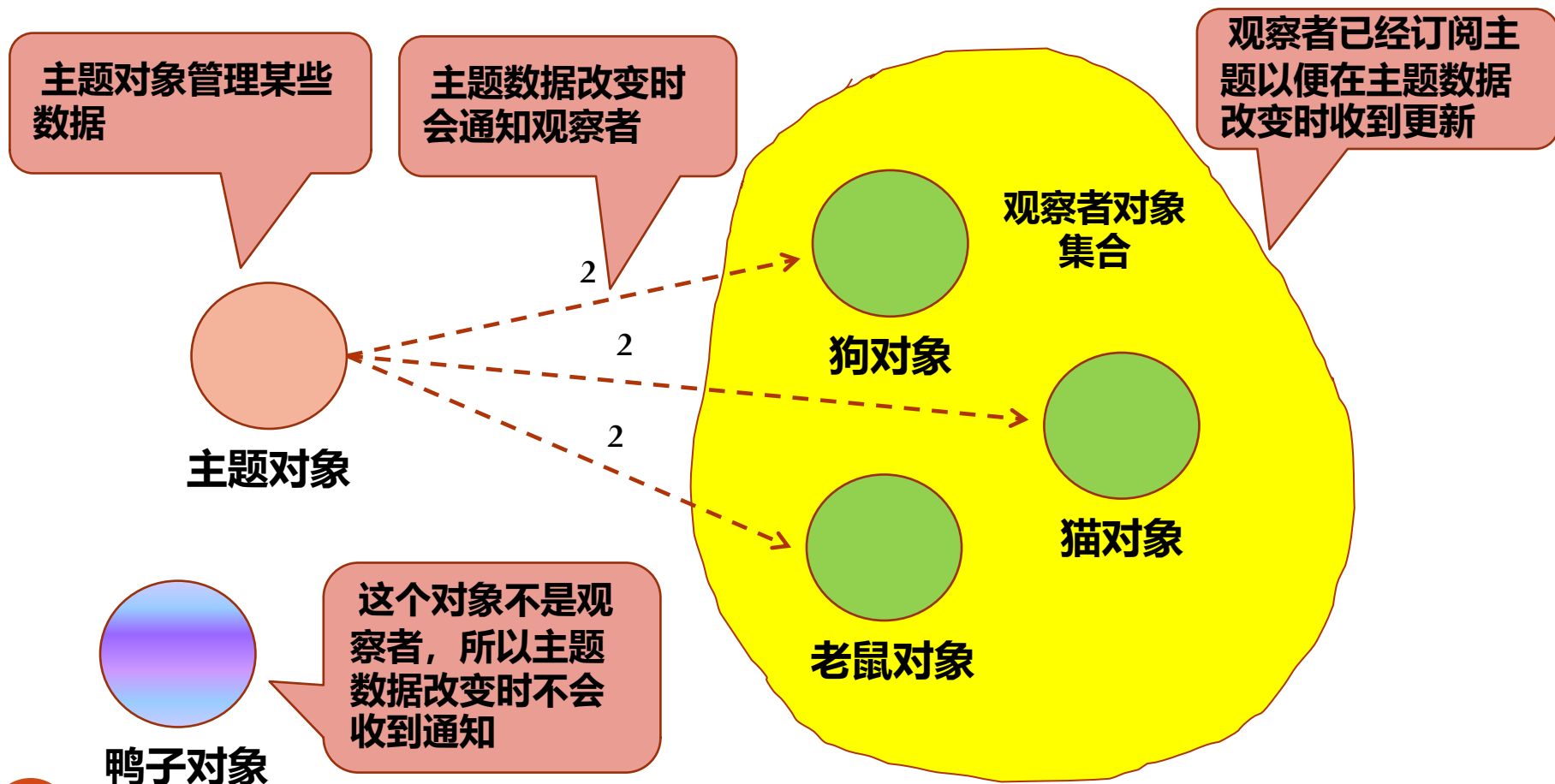
认识观察者模式

□ 看看报纸和杂志的订阅

- 用户向报社订阅报纸
- 报社只要有新报纸出版，就会给你送来。只要你一直是订户，你就会一直收到报纸
- 你不想看报纸时，可以取消订阅。报社不会再给你送来新报纸
- 只要报社还在运营，就会一直有人向报社订阅或取消订阅

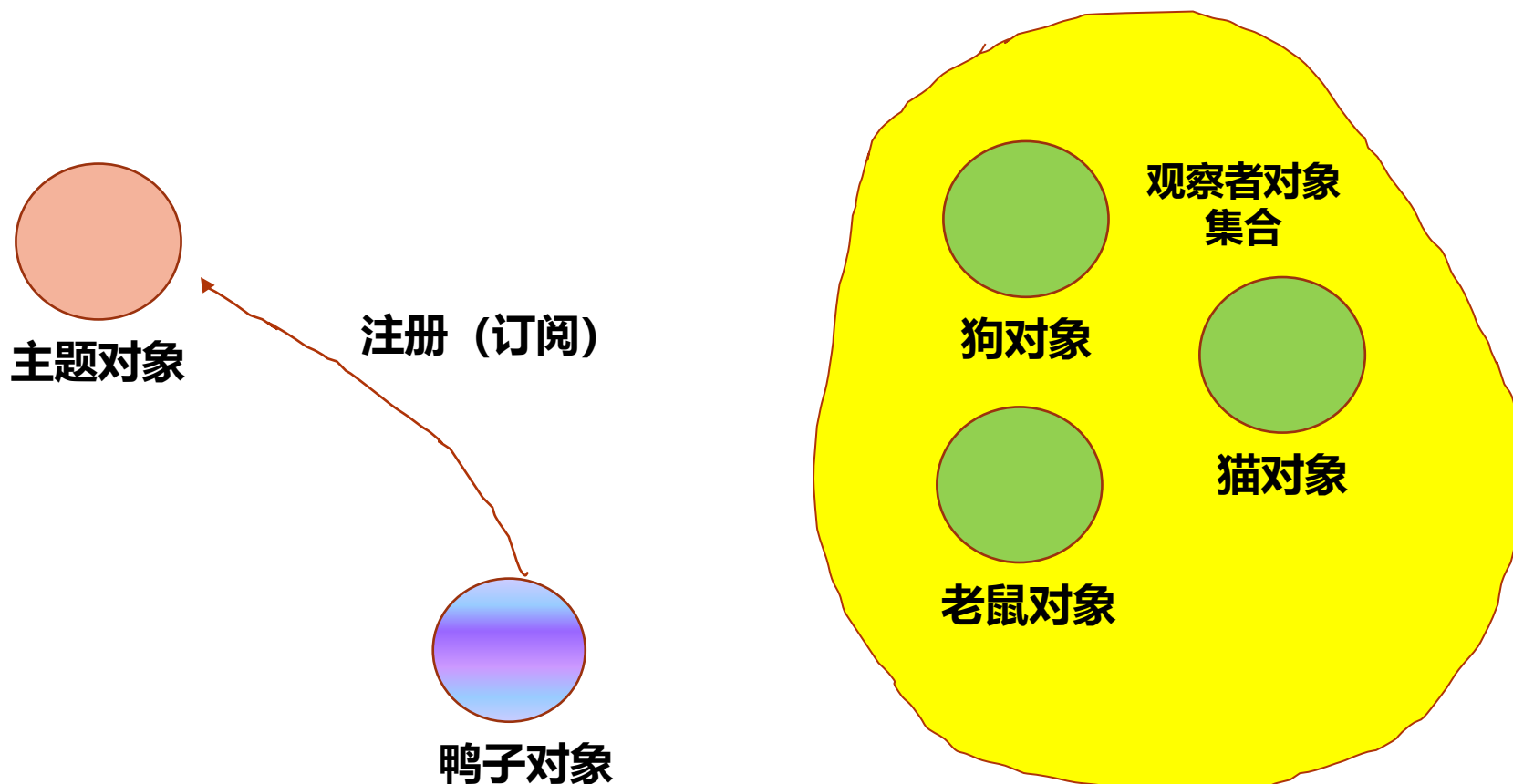
出版者+订阅者=观察者模式

□ 在观察者模式里，出版者叫“主题”（Subject），订阅者叫“观察者”（Observer）



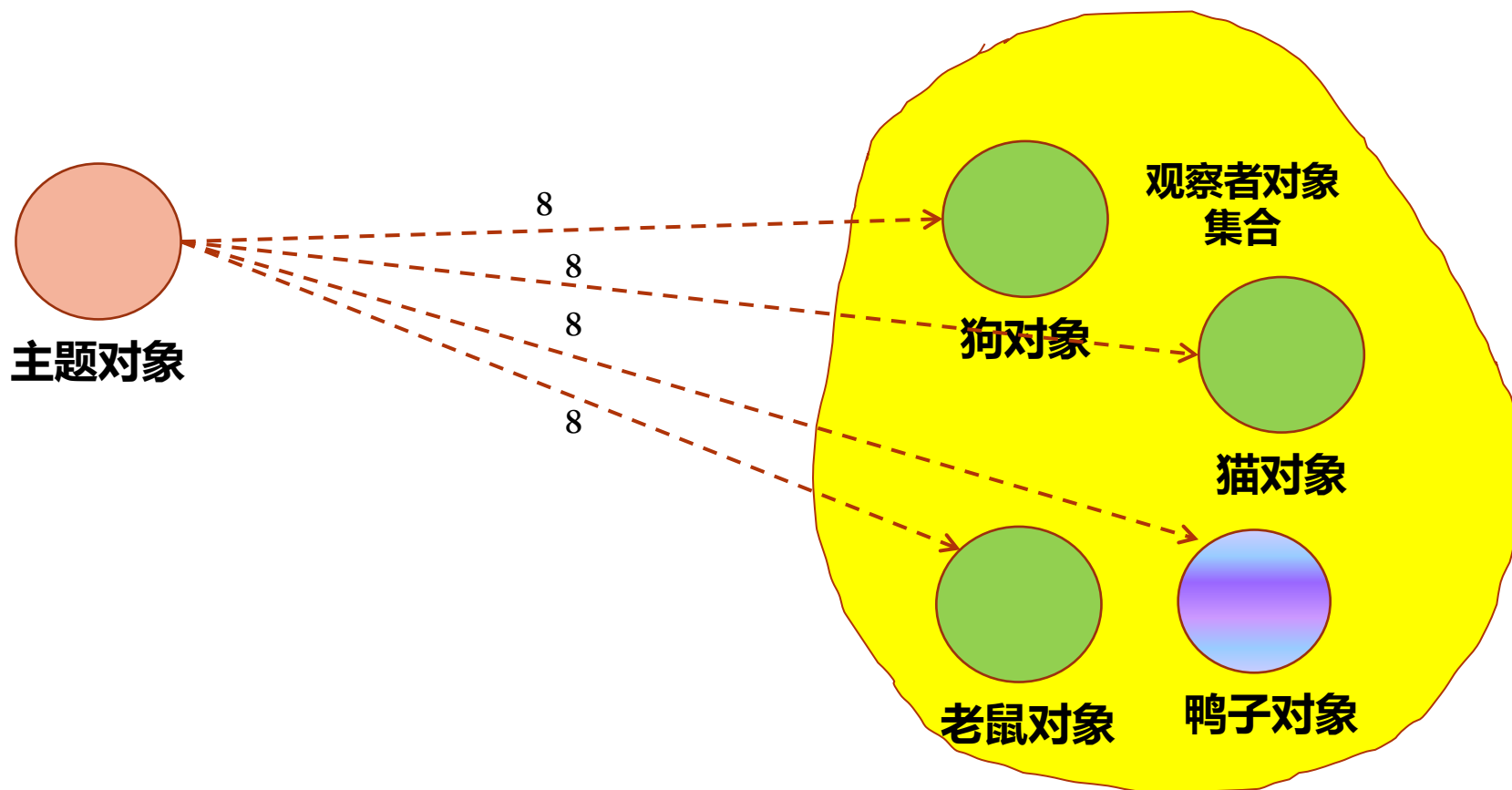
出版者+订阅者=观察者模式

□ 鸭子对象想成为观察者，首先要向主题对象注册，注册后成为观察者



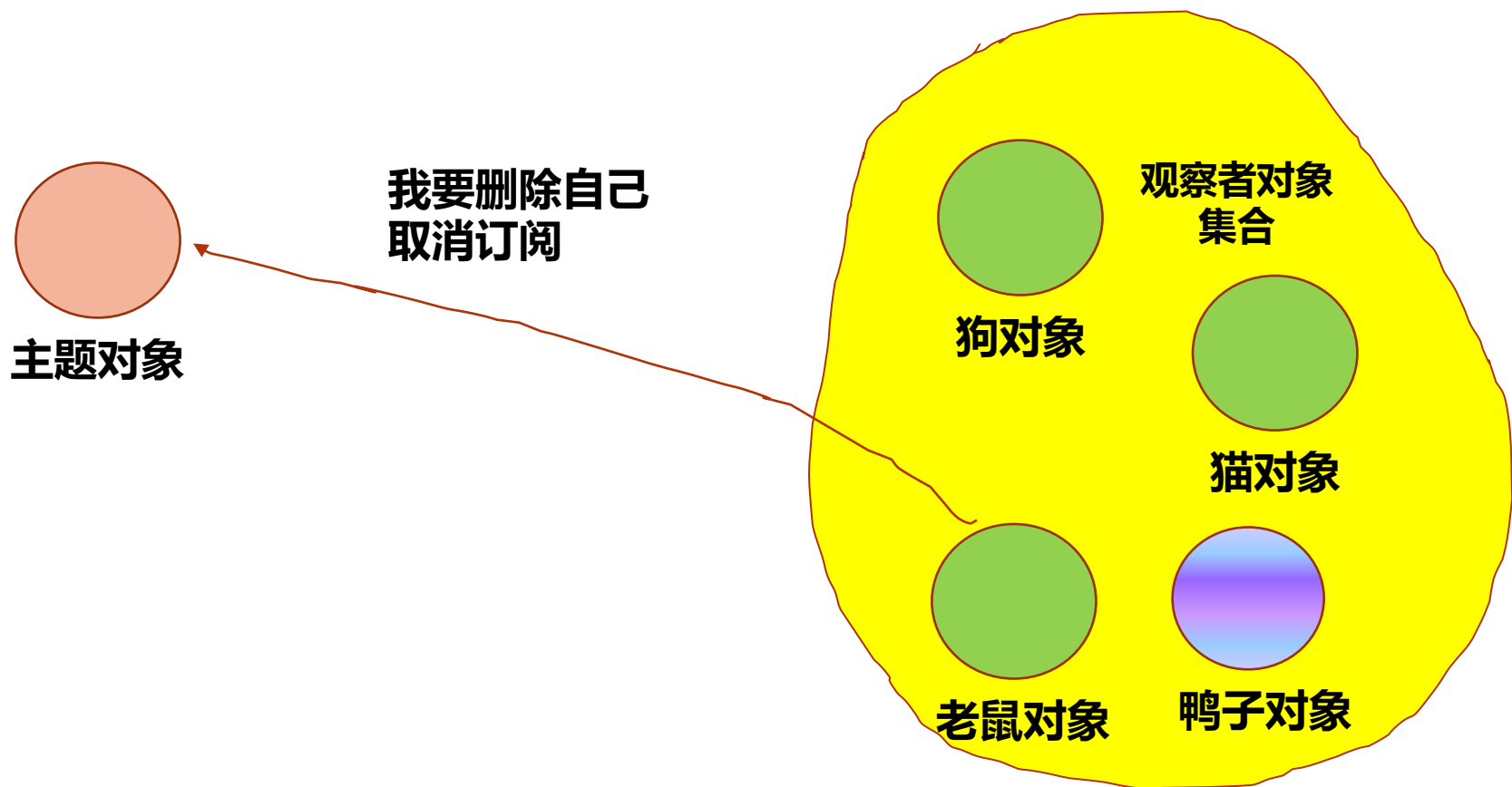
出版者+订阅者=观察者模式

□ 主题有了新数据，会通知所有观察者



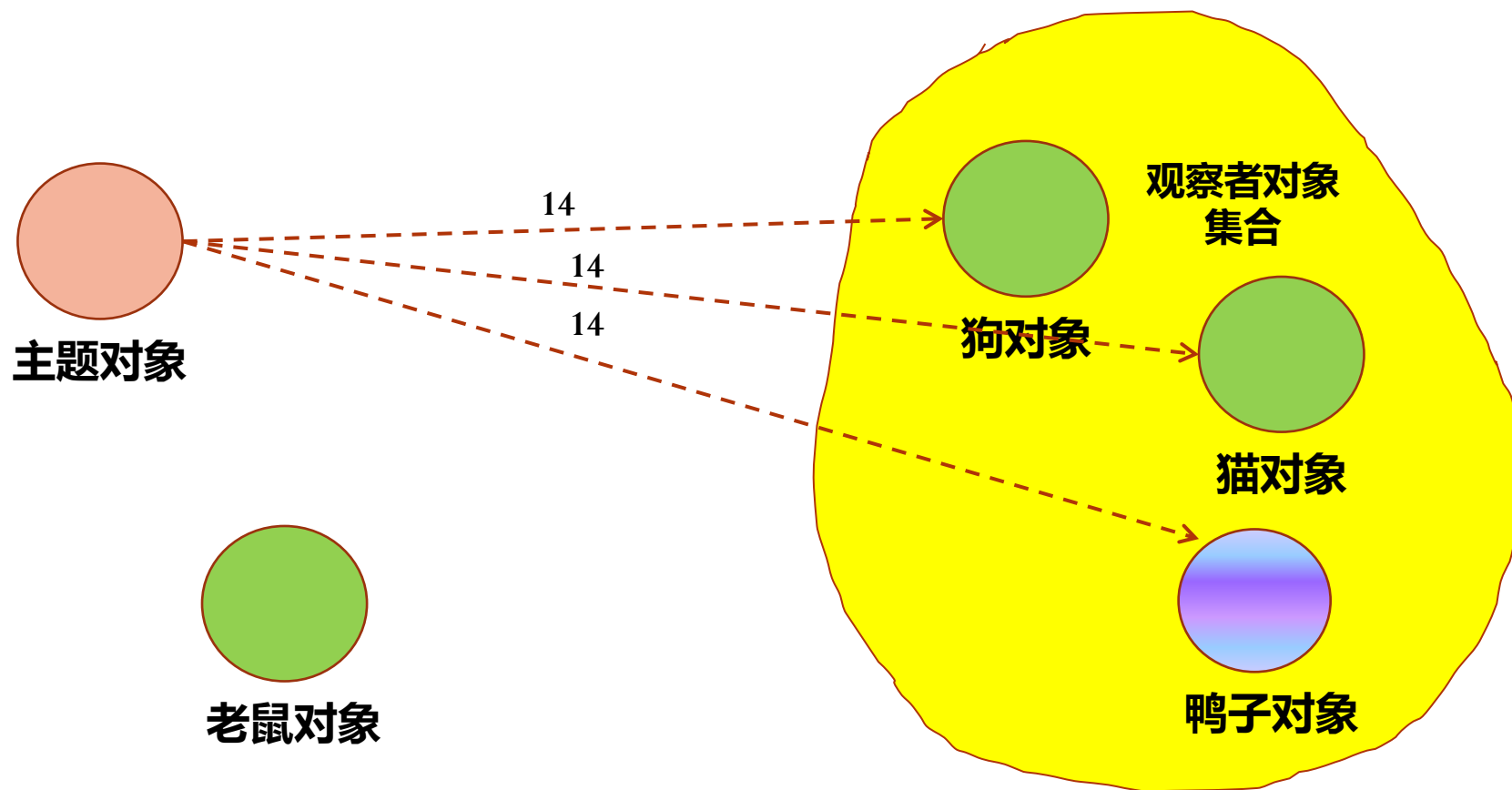
出版者+订阅者=观察者模式

□ 老鼠对象厌倦了，要求从观察者中把自己除名



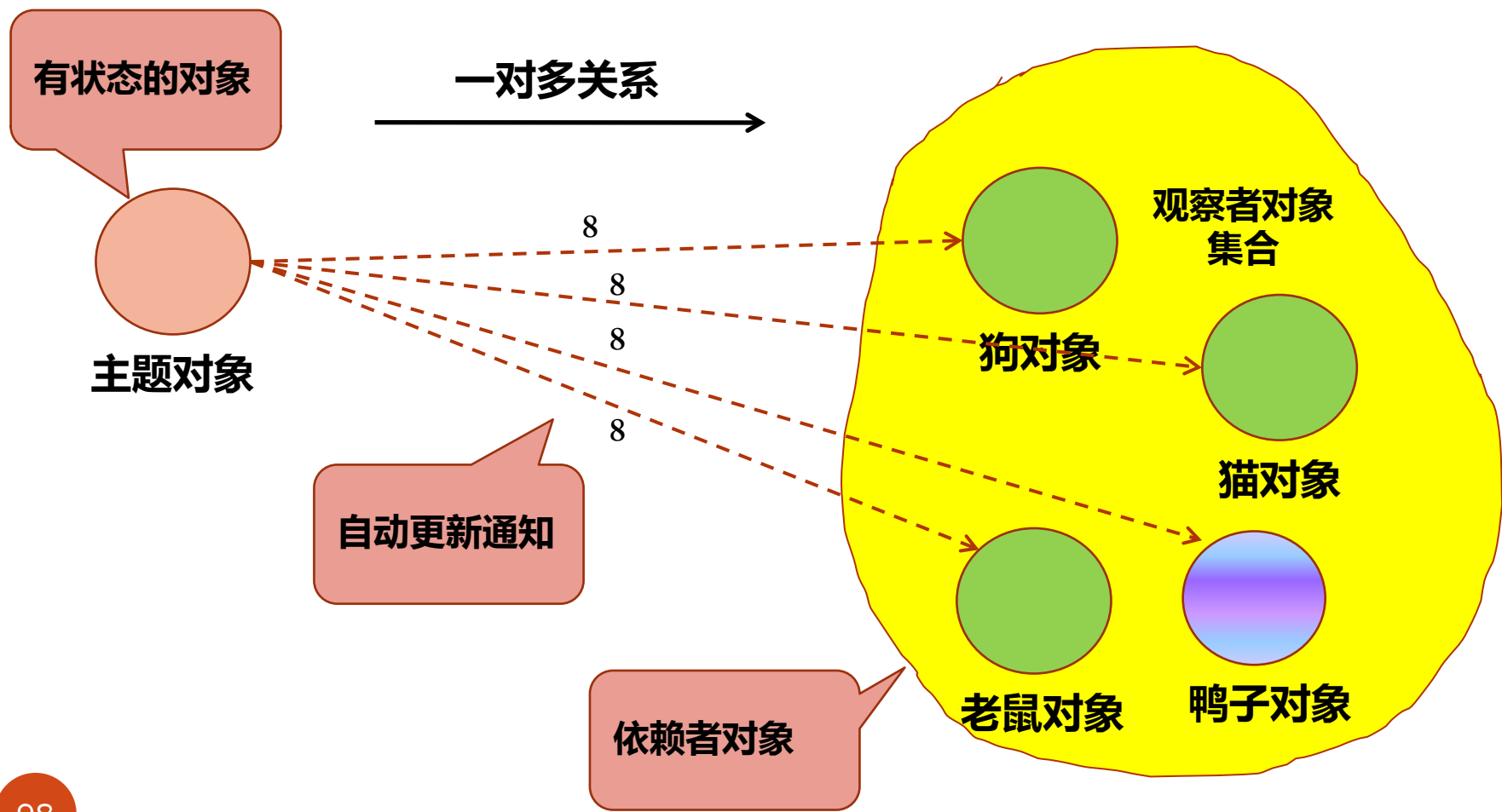
出版者+订阅者=观察者模式

□ 主题又有了新数据，会通知当前剩下的所有观察者



定义观察者模式 (Observer Pattern)

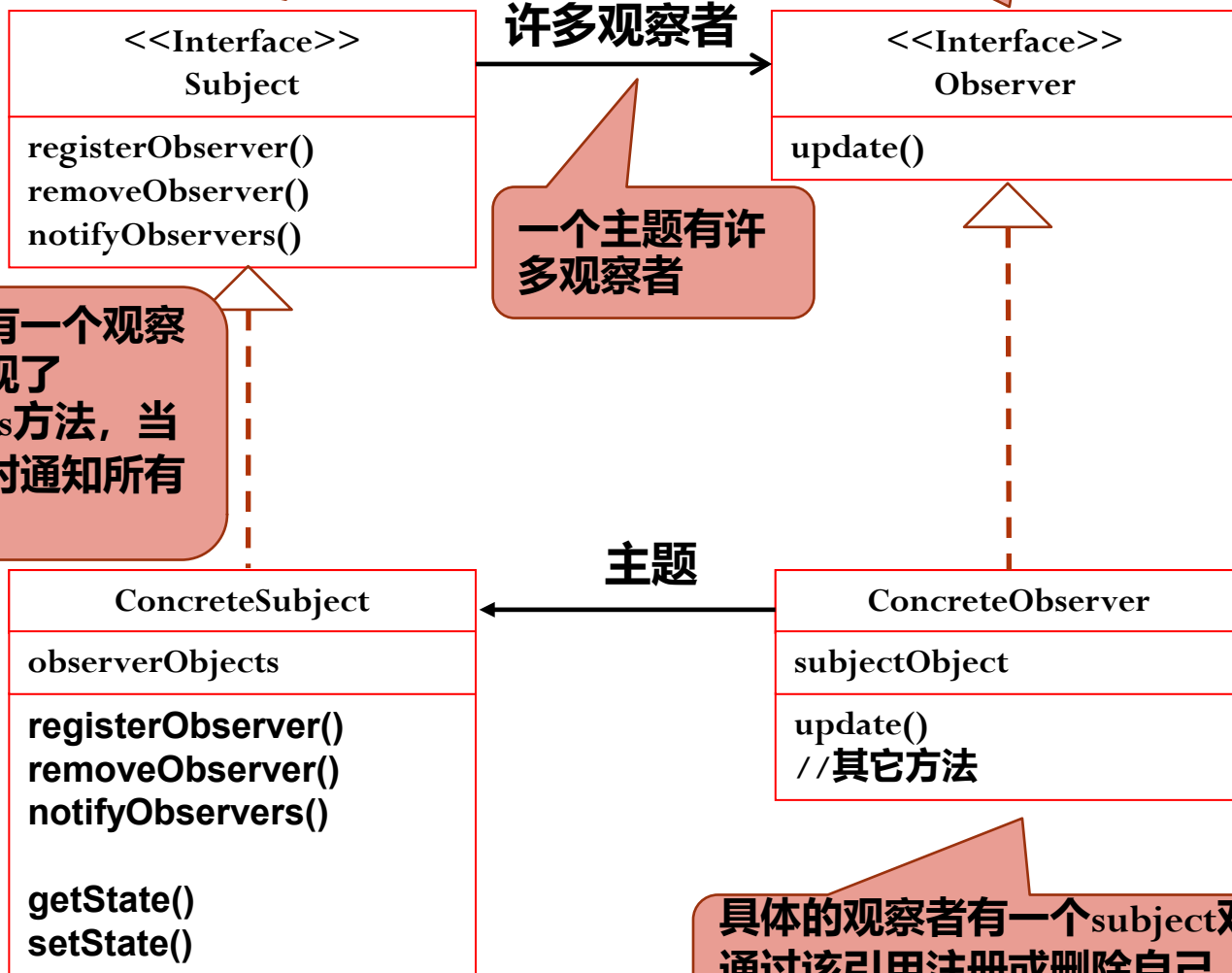
- 观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新



观察者模式的结构

主题接口，对象利用此接口注册为观察者，或者把自己从观察者中删除

观察者接口，只有update方法，当主题状态改变时被调用



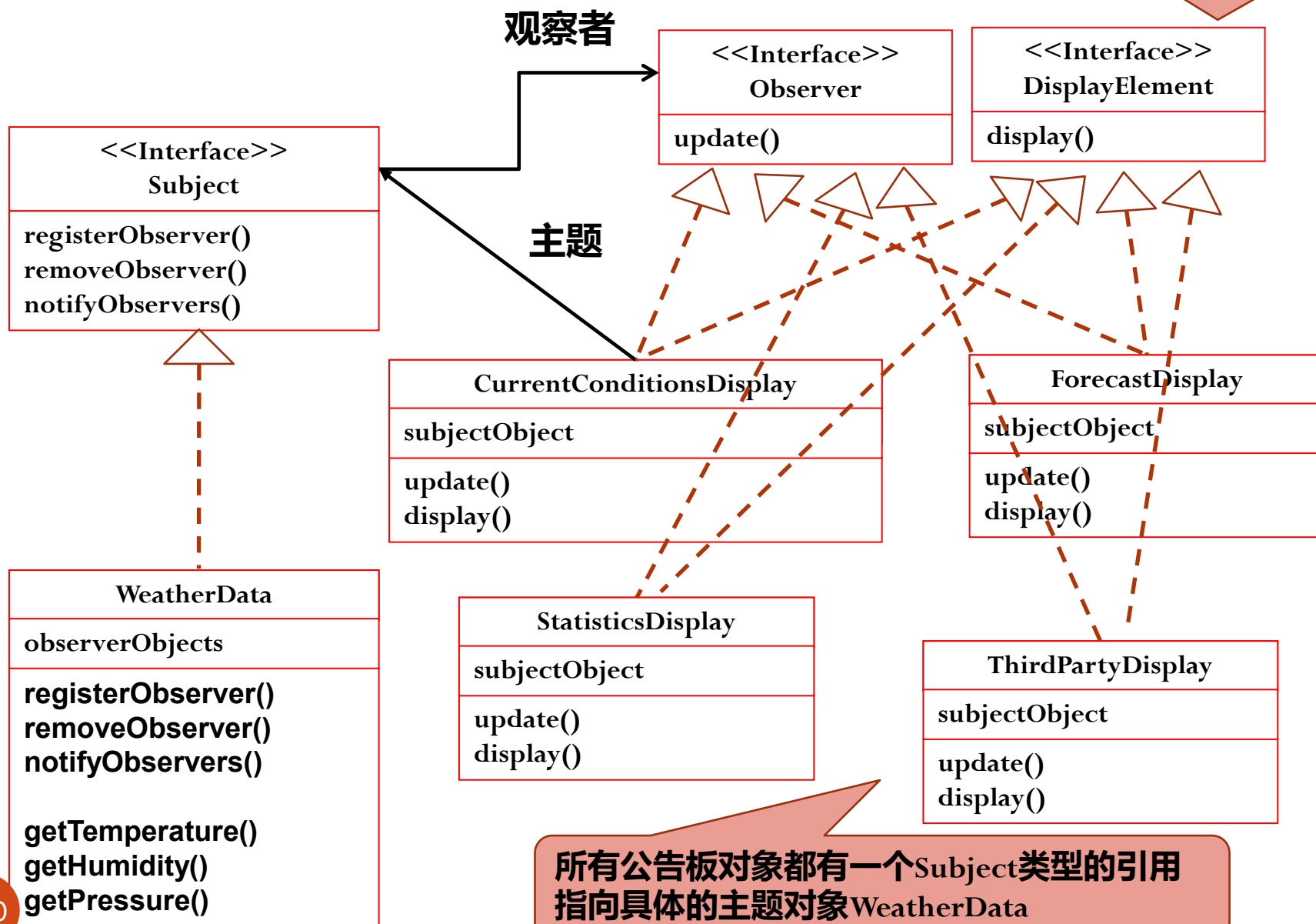
一个主题有许多观察者

具体的主题。有一个观察者列表。还实现了notifyObservers方法，当自己状态改变时通知所有观察者

具体的观察者有一个subject对象的引用，通过该引用注册或删除自己，并给出update的具体实现

设计气象站

为公告板设计接口，定义display()方法



实现气象站

```
public interface Subject {  
    public void registerObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
}
```

这二个方法需要Observer接口类型的变量，传入需要注册或删除的具体观察者对象

当主题状态改变时，这个方法被调用以通知所有的观察者

```
public interface Observer {  
    public void update(float temp, float humidity, float pressure);  
}
```

当主题状态改变时，该方法会被调用以把温度、湿度、气压传递给观察者

```
public interface DisplayElement {  
    public void display();  
}
```

当公告板需要显示时，调用该方法

实现气象站

```
public class WeatherData implements Subject {  
    private ArrayList observers;
```

主题对象将所有注册的观察者对象保持在ArrayList中

```
    private float temperature, humidity, pressure;
```

```
    public float getTemperature() { return temperature; }  
    public float getHumidity() { return humidity; }  
    public float getPressure() { return pressure; }
```

```
    public WeatherData() {  
        observers = new ArrayList();  
    }
```

```
    public void registerObserver(Observer o) {  
        observers.add(o);  
    }
```

当注册观察者对象时，将其加入到ArrayList中

```
    public void removeObserver(Observer o) {  
        int i = observers.indexOf(o);  
        if (i >= 0) {  
            observers.remove(i);  
        }  
    }  
}
```

当删除观察者对象时，将其从ArrayList中删除

实现气象站（续）

```
public class WeatherData implements Subject {
    //...
    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }
    public void measurementsChanged() {
        notifyObservers();
    }
    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
}
```

遍历ArrayList中的所有观察者对象，调用他们的update方法，将新的状态通知他们。
注意每个观察者对象都继承了Observer接口，因此都实现了update方法

当测量值变化时，调用notifyObservers通知所有的观察者

模拟读取传感器数据的动作，用来测试公告板。

实现公告板

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {  
    private float temperature;  
    private float humidity;  
    private Subject weatherData;  
  
    public CurrentConditionsDisplay(Subject weatherData) {  
        this.weatherData = weatherData;  
        weatherData.registerObserver(this);  
    }  
    public void update(float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        display();  
    }  
    public void display() {  
        System.out.println("Current conditions: " + temperature  
            + "F degrees and " + humidity + "% humidity");  
    }  
}
```

观察者对象保存一个Subject类型的引用，指向主题对象

构造函数传入主题对象，并将自己向主题对象注册

当update方法被调用时，保存温度和湿度，并调用display()以更新公告板的显示

为什么要保存Subject引用？构造函数里注册了似乎用不着了。
可能以后需要取消注册，保留该引用比较方便

启动气象站

```
public class WeatherStation {  
    public static void main(String[] args) {  
        Subject weatherData = new WeatherData();  
  
        Observer currentDisplay =  
            new CurrentConditionsDisplay(weatherData);  
  
        Observer statisticsDisplay = new StatisticsDisplay(weatherData);  
  
        Observer forecastDisplay = new ForecastDisplay(weatherData);  
  
        weatherData.setMeasurements(80, 65, 30.4f);  
        weatherData.setMeasurements(82, 70, 29.2f);  
        weatherData.setMeasurements(78, 90, 29.2f);  
    }  
}
```

观察者获取主题对象数据的二种方式

- ❑ 由主题对象在状态发生改变时，将新的数据 “推送” (Push) 到观察者（目前气象站就采用这种方式，通过调用观察者的 update 方法）。这种做法的缺点：
 - ❑ 有时观察者并不需要所有的数据，而只是需要其中一部分（如 **CurrentConditionsDisplay**）。但 “推送” 则将所有数据全部推送到观察者。
 - ❑ 如果主题对象状态变化十分频繁（比如气象站温度每变化0.01度就会更新主题对象的状态），导致频繁地将数据发送到观察者。
- ❑ 主题对象提供 getter 方法让观察者读取状态，同时主题对象在状态改变时只 “通知” 观察者数据改变了，但不将数据推送到观察者，而是由观察者调用主题对象提供的 getter 方法将数据从主题对象 “拉” (Pull) 到观察者这边。

在JDK中，还有哪些地方可以找到观察者模式

□ Java Swing是图形界面编程API，其事件处理就用到了观察者模式

```
public class SwingObserverExample {
```

```
    public static void main(String[] args) {
```

```
        JFrame frame = new JFrame();
```

```
        JButton button = new JButton("Should I do it?");
```

```
        button.addActionListener(new AngelListener());
```

```
        button.addActionListener(new DevilListener());
```

```
        frame.getContentPane().add(BorderLayout.CENTER, button);
```

```
        //设置frame其他属性
```

```
    }
```

```
    class AngelListener implements ActionListener {
```

```
        public void actionPerformed(ActionEvent event) {
```

```
            System.out.println("Don't do it, you might regret it!");
```

```
        }
```

```
    }
```

```
    class DevilListener implements ActionListener {
```

```
        public void actionPerformed(ActionEvent event) {
```

```
            System.out.println("Come on, do it!");
```

```
        }
```

```
    }
```

JButton对象就是可观察者对象

产生二个观察者对象，并注册到button上

二个观察者类定义，实现ActionListener接口

观察者模式的意图和适用性

□ **意图**：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新

□ 适用场合

- 当一个抽象模型有两个方面,一方面依赖于另一方面。将二者封装在独立对象中以使它们可以独立被改变和复用
- 一个对象的改变需要同时改变其它对象,但不知道具体有多少对象有待改变
- 当一个对象必须通知其它对象, 而它又不能假定其它对象是谁。换言之,不希望这些对象是紧密耦合的

观察者模式的参与者

- Subject: 抽象的主题, 即被观察的对象
- Observer: 抽象的观察者
- Concrete Subject: 具体被观察对象
- Concrete Observer: 具体的观察者
- 注意: 在观察者模式中, Subject通过addObserver和removeObserver方法添加或删除所关联的观察者, 并通过notifyObservers进行更新, 让每个观察者观察到最新的状态

观察者模式分析

□ 应用场景

- 对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变
- 对象仅需要将自己的更新通知给其他对象而不需要知道其他对象细节

□ 优点

- Subject和Observer之间是松耦合的，可以各自独立改变
- Subject在发送广播通知时，无须指定具体的Observer，Observer可以自己决定是否要订阅Subject的通知
- 高内聚、低耦合

□ 缺陷

- 松耦合导致代码关系不明显，有时可能难以理解
- 如果一个Subject被大量Observer订阅的话，在广播通知的时候可能会有效率问题

命令模式的由来

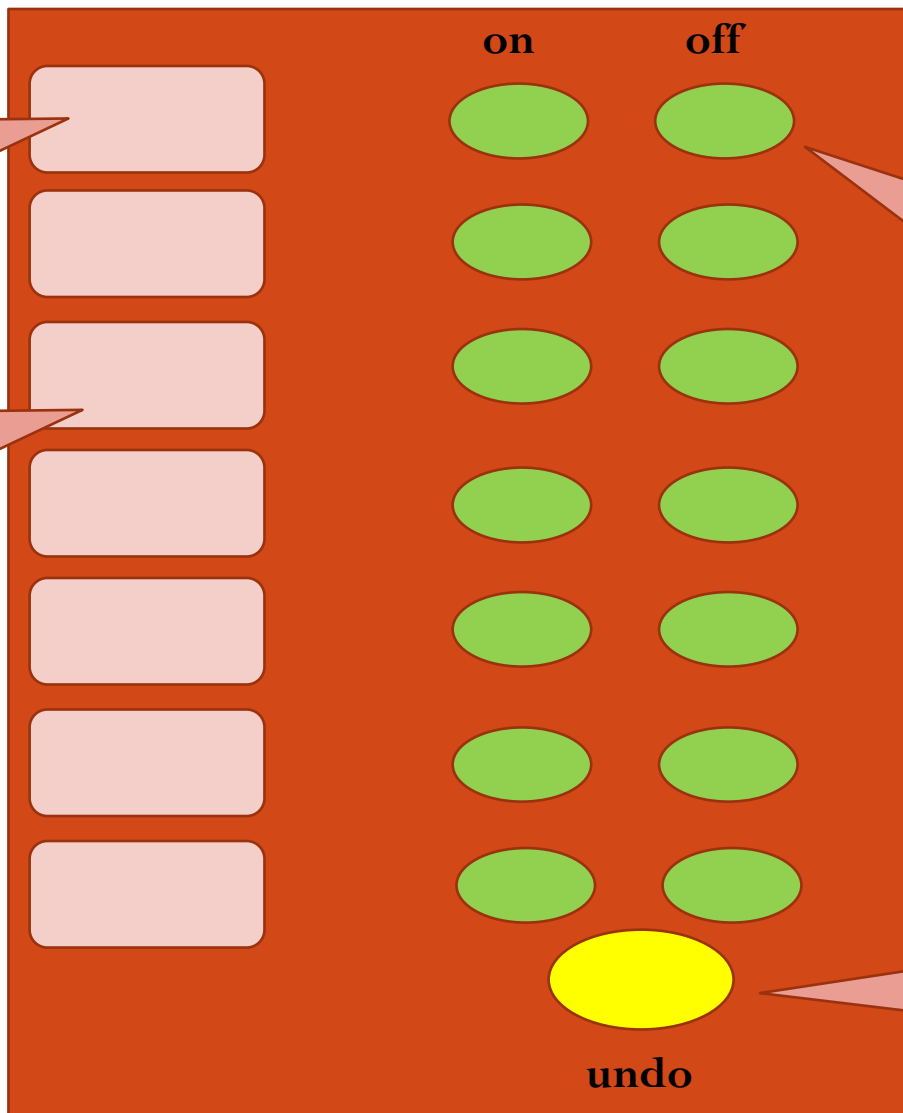
□ 实现一个家电装置的遥控器

遥控器有7个可编程的插槽，每个插槽对应不同的家电装置

可编程的插槽可以随时替换指定插槽对应的家电

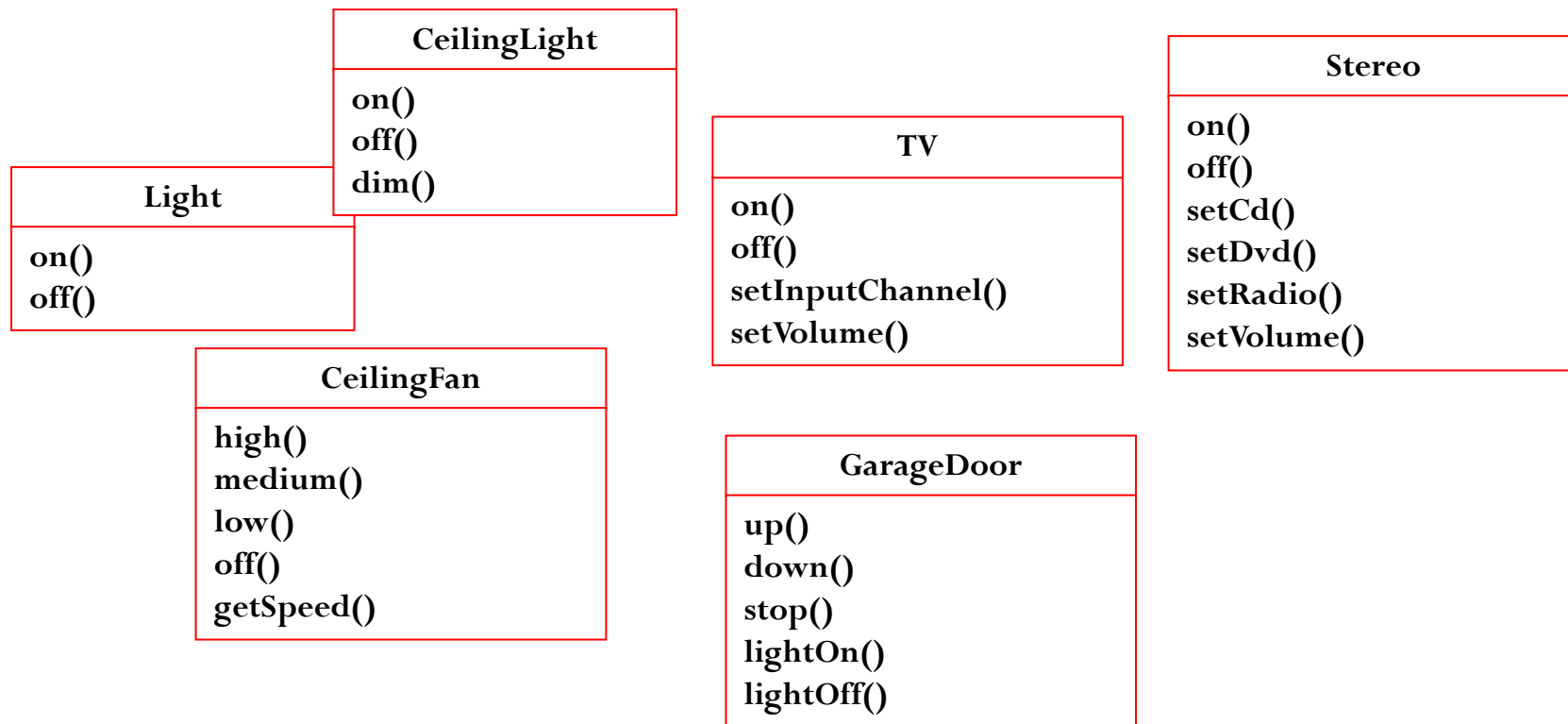
7个插槽具备各自的“开”、“关”按钮

共用的“撤销”按钮，会撤销最后一个按钮的动作



命令模式的由来

□ 有如下一些已经实现的家电类



不同的家电类接口各有差异，以后可能会有新的家电类。因此设计遥控器具有挑战性：遥控器插槽插入新的家电也要能工作

命令模式的由来

□ 遥控器设计要点

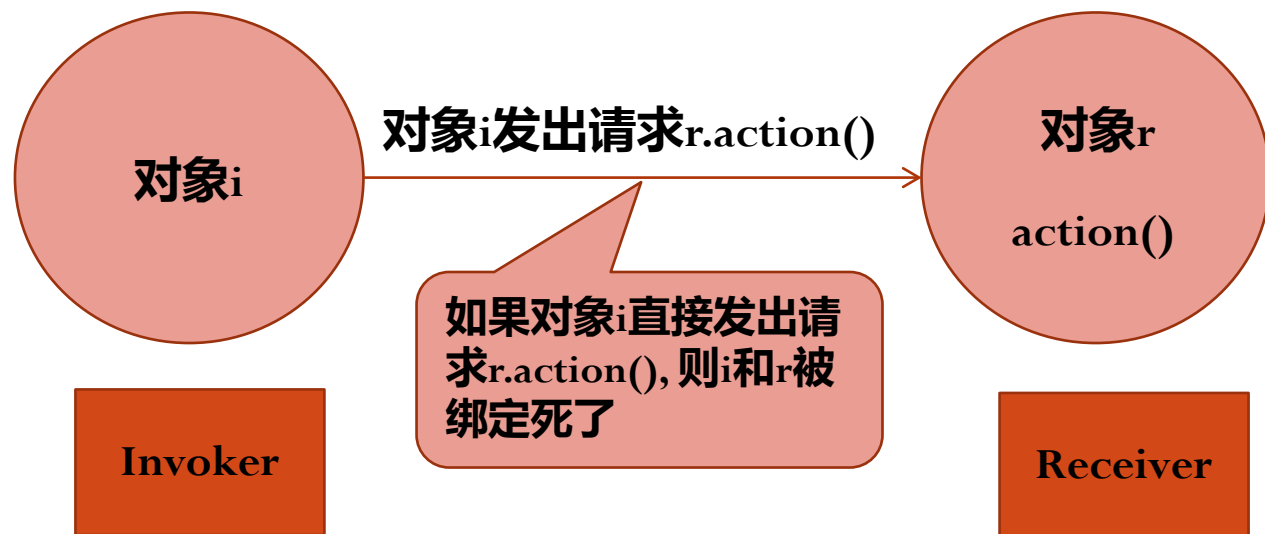
- 遥控器要知道如何解读按钮被按下的动作，然后发出正确的请求。但遥控器不能知道这些家电自动化的细节。否则遥控器就被绑定的具体的家电对象上。
- 因此不能让遥控器包含一大堆if语句，如 “if slot1==Light, then light.on(), else if slot1==TV, then tv.on()”。这样的设计很糟糕。这种设计导致新的家电插进插槽，就得修改遥控器的代码，而且工作没完没了。

命令模式的由来

- ❑ 需要将“动作的请求者”（遥控器）和“动作的执行者”（家电对象）解耦。
- ❑ 看起来很困难，毕竟当按下遥控器按钮时，必须打开电灯，即在遥控器代码中调用`light.on()`方法。
- ❑ 采用“命令模式”可以做到：把动作请求（如打开电灯）封装成一个“命令对象”。所有如果每个插槽都存储一个命令对象，那么当对应按钮被按下时，可以请命令对象做相关的工作。遥控器并不需要动作如何执行，只要知道有个命令对象能和正确的家电对象沟通。因此，遥控器和家电对象（如电灯）解耦了。

命令模式的由来

- ❑ 二个概念：动作的请求者(Invoker)和动作的接受者(Receiver)
- ❑ 假设某个对象r具有某个方法（或叫动作）action()，如果在另外一个对象i的某个方法中调用r.action(); 则对象i称为动作的请求者，对象r称为动作的接受者。



命令模式的由来

□ 现在利用“命令对象”将Invoker和Receiver解耦

命令对象中包含指向对象r的引用，同时实现了execute方法，该方法封装了动作的调用

对象i包含了命令对象c

命令对象c
execute()

```
public void execute() {  
    r.action();  
}
```

c.execute()

Commander

r.action()

对象i

Invoker

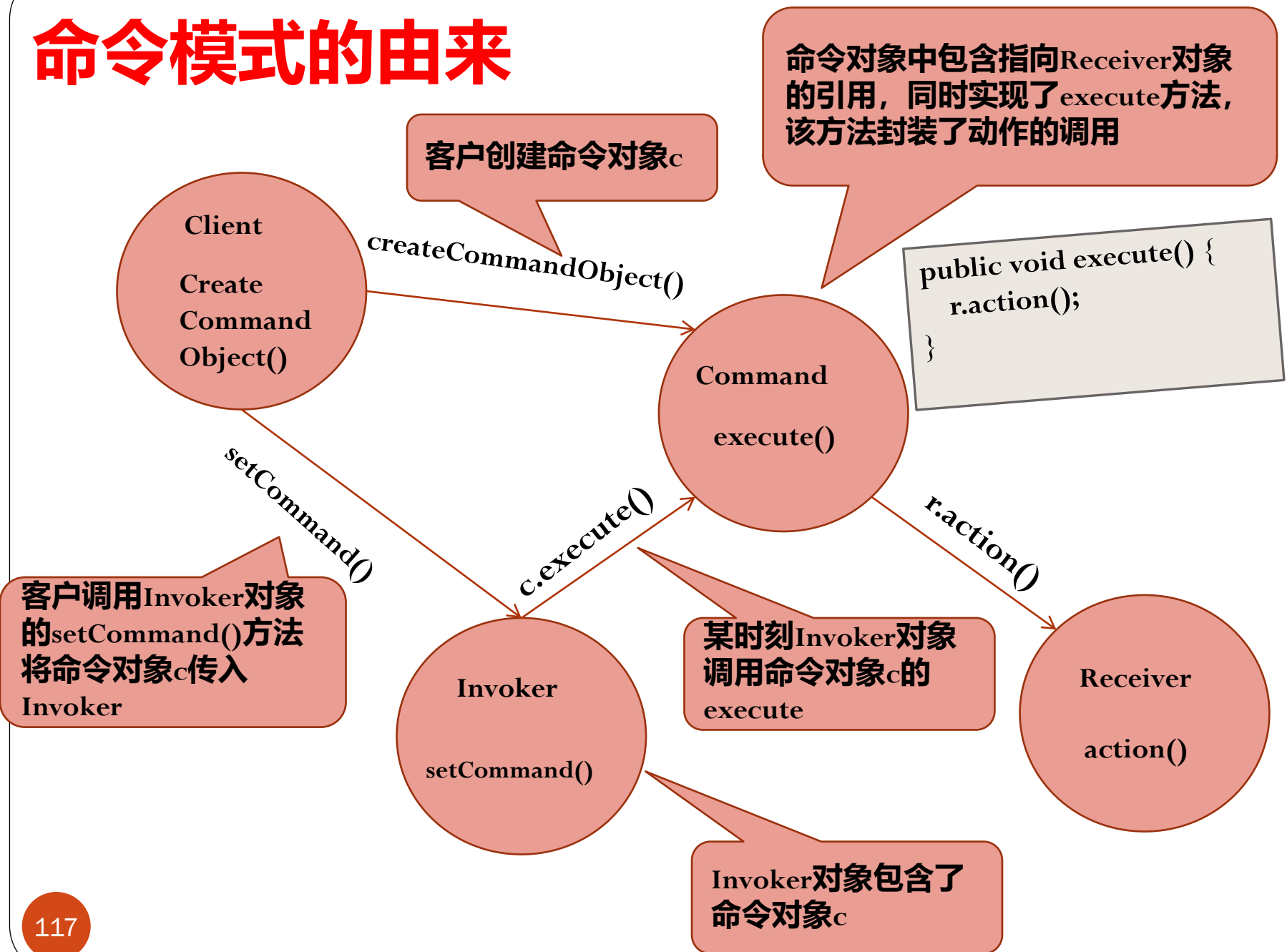
对象r

action()

把execute()定义到接口，就可以将Invoker和Receiver解耦

Receiver

命令模式的由来



实现一个命令对象

□ 首先定义命令接口

```
public interface Command{  
    public void execute();  
}
```

□ 实现打开电灯的命令

```
public class LightOnCommand implements Command{  
    Light light;  
  
    public LightOnCommand(Light light){  
        this.light = light;  
    }  
  
    public void execute(){  
        light.on();  
    }  
}
```

命令对象包含了Receiver对象（即动作的执行者）的引用

构造函数被传入Receiver对象

调用Receiver对象（light）的on()方法

使用命令对象

□ 实现Invoker (遥控器)

```
public class SimpleRemoteControl{
```

```
    Command slot;
```

一个命令插槽持有命令对象，而这个命令对象控制着一个电子设备

```
    public SimpleRemoteControl(){ }
```

```
    public void setCommand(Command command){
```

```
        this.slot = command;
```

```
    }
```

用来设置插槽持有的命令对象。如果需要改变遥控器按钮的行为，可以调用该方法设置新的命令对象

```
    public void buttonPressed(){
```

```
        slot.execute();
```

```
    }
```

```
}
```

当遥控器按钮按下时，调用命令对象的execute()方法

使用命令对象

□ 测试

```
public class SimpleRemoteControlTest{  
    public static void main(String[] args){
```

遥控器对象就是Invoker对象

```
        SimpleRemoteControl remote = new SimpleRemoteControl();
```

```
        Light light = new Light();
```

这是Receiver对象

```
        LightOnCommand lightOn = new LightOnCommand(light);
```

这是Command对象，构造命令对象时需要传入Receiver对象

```
        remote.setCommand(lightOn);  
        remote.buttonPressed();
```

调用Invoker对象的setCommand()方法把命令传给调用者

模拟按下按钮

定义命令模式

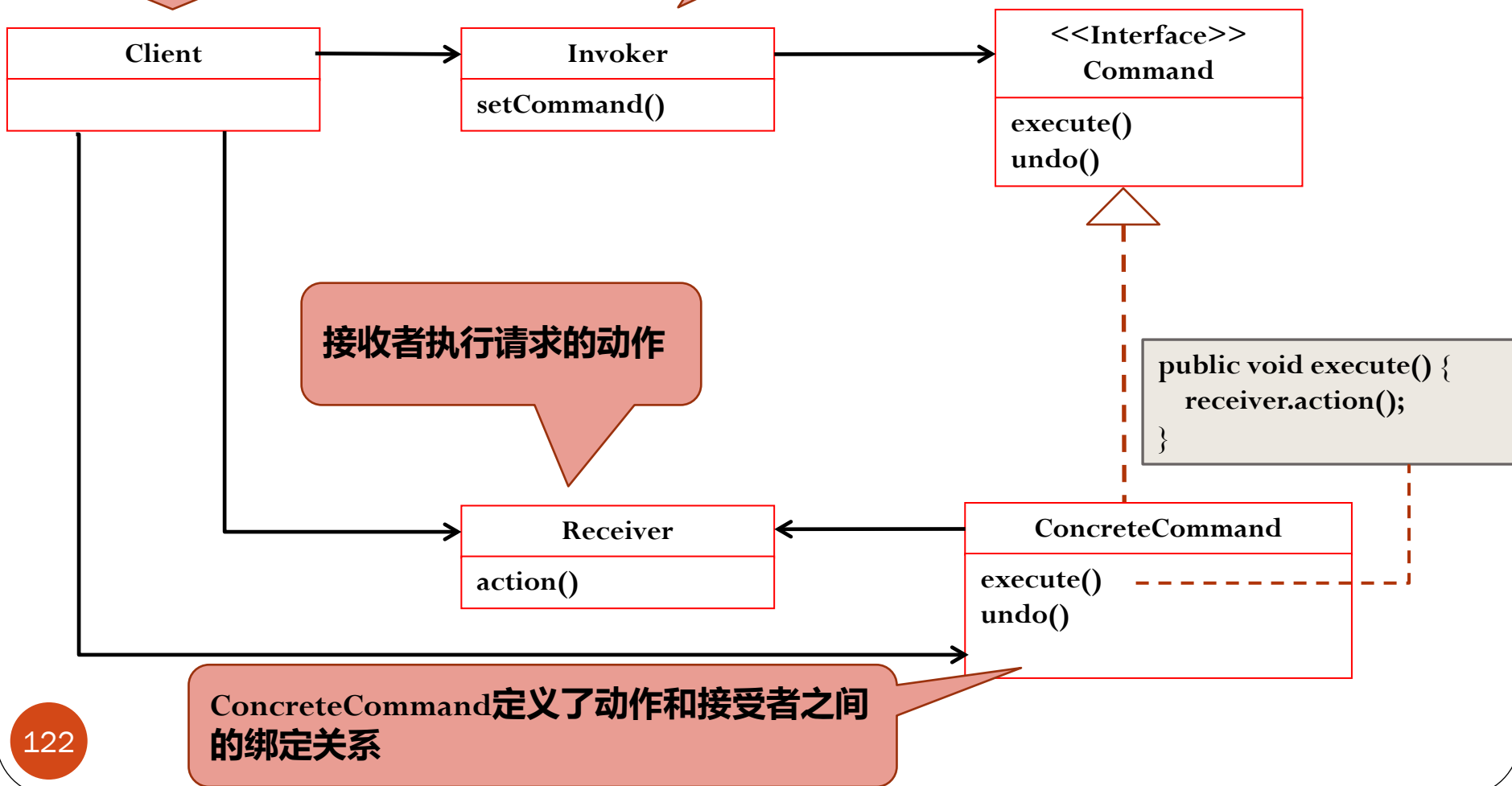
- 命令模式将“请求”封装成对象，以便参数化其它对象。命令模式也支持可撤销操作。
 - 命令对象将动作和动作的接受者包装在一起，对外只暴露出execute()方法。客户只知道如果调用execute()方法，请求的目的就可以达到。
 - 由于“请求”被封装成对象，因此可以用该对象作为参数来设置其它对象。如利用Invoker（遥控器）的setCommand()命令可以动态设置遥控器支持的命令插槽。
 - 也可以用“命令模式”来实现队列、日志和撤销。

命令模式的结构

客户负责创建ConcreteCommand对象，并对命令对象设置Receiver对象，同时对Invoker设置命令对象

Invoker持有命令对象，并在某时刻调用命令对象的execute()方法

Command为所有命令对象声明了接口，调用execute()方法可以让接受者执行动作，undo()撤销动作



将命令指定到遥控器插槽

```
public class RemoteControl {  
    Command[] onCommands;  
    Command[] offCommands;  
    public RemoteControl() {  
        onCommands = new Command[7];  
        offCommands = new Command[7];  
        Command noCommand = new NoCommand();  
        for (int i = 0; i < 7; i++) {  
            onCommands[i] = noCommand;  
            offCommands[i] = noCommand;  
        }  
    }  
    public void setCommand(int slot, Command onCommand, Command offCommand) {  
        onCommands[slot] = onCommand;  
        offCommands[slot] = offCommand;  
    }  
    public void onButtonWasPushed(int slot) {  
        onCommands[slot].execute();  
    }  
    public void offButtonWasPushed(int slot) {  
        offCommands[slot].execute();  
    }  
}
```

遥控器要处理7个“开”和7个“关”命令。用Command数组来保存

初始化命令数组，命令数组的每个元素保存的是“空命令”

设置指定插槽的On命令和Off命令

当按下开或关的按钮，就会调用相应的方法

遥控器就是Invoker

空命令

□ 空命令的作用是避免每次检查是否某个插槽加载了命令，如

```
public void onButtonWasPressed(int slot){  
    if(onCommand[slot] != null)  
        onCommand[slot].execute();  
}
```

□ 因此为了避免以上的做法，实现一个不做任何事情的命令

```
public class NoCommand implements Command{  
    public void execute() {}  
}
```

空对象在许多设计模式中被使用，
甚至空对象本身也被视为一种设计模式

实现更多的命令对象

□ 实现关闭电灯的命令

```
public class LightOffCommand implements Command{
```

```
    Light light;
```

命令对象包含了Receiver对象（即动作的执行者）的引用

```
    public LightOffCommand(Light light){
```

```
        this.light = light;
```

```
    }
```

构造函数被传入Receiver对象

```
    public void execute(){
```

```
        light.off();
```

```
    }
```

```
}
```

调用Receiver对象（light）的off()方法

实现更多的命令对象

□ 实现打开CD音响的命令

```
public class StereoOnWithCDCommand implements Command{
```

```
    Stereo stereo;
```

命令对象包含了Receiver对象（即动作的执行者）的引用

```
    public StereoOnWithCDCommand (Stereo stereo){
```

```
        this.stereo= stereo;
```

```
    }
```

构造函数被传入Receiver对象

```
    public void execute(){
```

```
        stereo.on();
```

```
        stereo.setCD();
```

```
        stereo.setVolume(11);
```

```
    }
```

```
}
```

调用Receiver对象（stereo）的三个方法

根据其他的家电类，可以实现更多的命令对象。。。

测试遥控器

```
public class RemoteLoader{  
    public static void main(String[] args){  
        RemoteControl remoteControl = new RemoteControl ();  
  
        Light light= new Light();  
        Stereo stereo = new Stereo();  
  
        LightOnCommand lightOn = new LightOnCommand(light);  
        LightOffCommand lightOff = new LightOffCommand(light);  
        StereoOnCommand stereoOnCD= new StereoOnCommand (stereo);  
        StereoOffCommand stereoOffCD= new StereoOffCommand (stereo);  
  
        remoteControl.setCommand(0, lightOn,lightOff);  
        remoteControl.setCommand(1, stereoOnCD,stereoOff);  
  
        remoteControl.onButtonWasPushed(0);  
        remoteControl.offButtonWasPushed(0);  
        remoteControl.onButtonWasPushed(1);  
        remoteControl.offButtonWasPushed(1);  
    }  
}
```

创建好家电对象 (Receiver)

创建命令对象

把命令对象加载到命令插槽命令对象

按下遥控器按钮

实现命令撤销

- ❑ 重新定义Command接口，添加undo()方法。
- ❑ undo()执行的动作和execute()相反

```
public interface Command{  
    public void execute();  
    public void undo();  
}
```


实现命令撤销

□ LightOnCommand命令的撤销

```
public class LightOnCommand implements Command{  
    Light light;  
  
    public LightOnCommand(Light light){  
        this.light = light;  
    }  
  
    public void execute(){  
        light.on();  
    }  
  
    public void undo(){  
        light.off();  
    }  
}
```

这里，undo()做的动作和execute()相反

实现命令撤销

□ LightOffCommand命令的撤销

```
public class LightOffCommand implements Command{  
    Light light;  
  
    public LightOnCommand(Light light){  
        this.light = light;  
    }  
  
    public void execute(){  
        light.off();  
    }  
  
    public void undo(){  
        light.on();  
    }  
}
```

这里，undo()做的动作和execute()相反

```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;
    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];
        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand;
    }
    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }
    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }
    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
        undoCommand = offCommands[slot];
    }
    public void undoButtonWasPushed() {
        undoCommand.undo();
    }
```

前一次被执行的命令保存在这里

刚开始，没有前一个命令，因此undoCommand被初始化为空命令

当按下按钮时，执行相应的命令。同时将这个命令保存在undoCommand中

当按下撤销按钮时，执行undoCommand.undo(),就倒转了前一个被执行的命令

思考：如何撤销多个命令？

实现基于状态的命令撤销

- ❑ 电灯开关操作的撤销太简单，因为只涉及对象的二值状态
- ❑ 有时撤销操作要基于对象更多的内部状态。例如天花板上的吊扇 (CeilingFan)，吊扇允许有多种转动速度，这时的撤销操作会更复杂

```
public class CeilingFan {  
    public static final int HIGH = 3;  
    public static final int MEDIUM = 2;  
    public static final int LOW = 1;  
    public static final int OFF = 0;  
    String location;  
    int speed;  
    public CeilingFan(String location) {  
        this.location = location;  
        speed = OFF;  
    }  
    public void high()      { speed = HIGH; }  
    public void medium()    { speed = MEDIUM; }  
    public void low()       { speed = LOW; }  
    public void off()       { speed = OFF; }  
    public int getSpeed()   { return speed; }  
}
```

吊扇的内部4种状态

设置吊扇的速度

获取吊扇的速度

支持撤销的吊扇命令对象

- 要把撤销加入到吊扇的诸多命令对象中，需要追踪吊扇的最后设置速度

```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;
    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }
    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }
    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) { ceilingFan.high(); }
        else if (prevSpeed == CeilingFan.MEDIUM) { ceilingFan.medium(); }
        else if (prevSpeed == CeilingFan.LOW) { ceilingFan.low(); }
        else if (prevSpeed == CeilingFan.OFF) { ceilingFan.off(); }
    }
}
```

记录吊扇以前的速度

在将吊扇速度设置为HIGH前，记录吊扇之前的速度

将吊扇的速度设置回之间的值，达到撤销的目的

支持宏命令

- 希望按下按钮，遥控器能执行一系列命令：灯光调暗，打开音响和电视，设置好DVD
- 制造一种新的命令对象，用来执行一堆命令

```
public class MacroCommand implements Command {
```

```
    Command[] commands;
```

```
    public MacroCommand(Command[] commands){
```

```
        this.commands = commands;
```

```
    }
```

```
    public void execute(){
```

```
        for(int i = 0; i < commands.length; i++){
```

```
            commands[i].execute();
```

```
        }
```

```
    }
```

```
    public void undo(){
```

```
        for(int i = 0; i < commands.length; i++){
```

```
            commands[i].undo();
```

```
        }
```

```
    }
```

用命令数组来存储一大堆命令

当这个宏命令被执行时，依次执行数组里的每个命令

当这个宏命令被撤销时，依次撤销数组里的每个命令

使用宏命令

□ 首先创建要进入宏的命令集合

```
Light light = new Light();
```

```
TV tv = new TV();
```

```
Stereo stereo = new Stereo();
```

```
LightOnCommand lightOn = new LightOnCommand(light);
```

```
StereoOnCommand stereoOn = new StereoOnCommand (stereo);
```

```
TVOnCommand tvOn = new TVOnCommand(tv);
```

```
LightOffCommand lightOff = new LightOffCommand(light);
```

```
StereoOffCommand stereoOff = new StereoOffCommand (stereo);
```

```
TVOffCommand tvOff = new TVOffCommand(tv);
```

使用宏命令

- 创建两个数组，一个用来记录开启命令，一个用来记录关闭命令

```
Command[] partyOn = { lightOn, stereoOn, tvOn };  
Command[] partyOff = { lightOff, stereoOff, tvOff };
```

- 创建宏命令

```
MacroCommand partyOnMacro = new MacroCommand(partyOn);  
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

- 将宏命令指定给希望的按钮

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```


命令模式的更多用途：队列请求和日志请求

- ❑ 命令可以把运算块打包（一个接受者和一组动作）。既然命令被封装成对象，因此可以像一般对象一样传来传去。
 - ❑ 命令对象被创建后，只要对象还在，运算依然可以被调用，甚至可以在不同线程中被调用。
 - ❑ 想象有一个命令队列，在一端添加命令，另一端则是工作线程。线程从队列首部取出一个命令，调用它的execute()方法；等待调用完成再取下一个命令。
-
- ❑ 在命令接口添加二个方法:store()和load(), 命令模式可以支持命令对象保存在持久化介质中，或从持久化介质中恢复出命令对象
 - ❑ 这二个方法可以用来记录日志

命令模式的参与者

- ❑ Command
 - ❑ 声明执行操作的接口
- ❑ ConcreteCommand
 - ❑ 将一个接收者对象绑定于一个动作
 - ❑ 调用接收者相应的操作
- ❑ Client
 - ❑ 创建一个具体命令对象并设定其接收者
- ❑ Invoker
 - ❑ 要求命令执行请求
- ❑ Receiver
 - ❑ 知道如何实施与执行一个请求相关的操作

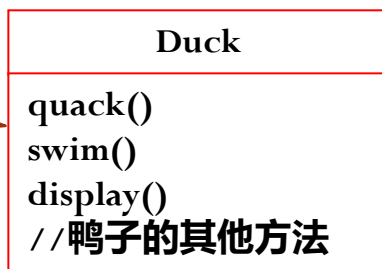
命令模式的效果分析

- ❑ 请求者不直接与接收者交互，即请求者不包接收者的引用，彻底消除了彼此之间的耦合
- ❑ 满足“开-闭原则”。如果增加新的具体命令和该命令的接受者，不必修改调用者的代码，调用者就可以使用新的命令对象；反之，如果增加新的调用者，不必修改现有的具体命令和接受者，新增加的调用者就可以使用已有的具体命令
- ❑ 由于请求者的请求被封装到了具体命令中，就可以将具体命令保存到持久化的媒介中，在需要的时候，重新执行这个具体命令。因此，使用命令模式可以记录日志
- ❑ 使用命令模式可以对“请求”进行排队。每个请求都各自对应一个具体命令，因此可以按一定顺序执行这些具体命令

策略模式的由来

- 如果需要实现一个模拟鸭子游戏：可以呱呱叫，可以游泳
- 可以首先设计一个超类Duck，并让各种鸭子继承此超类

所有的鸭子会呱呱叫，会游泳，因此超类负责这两个方法的实现



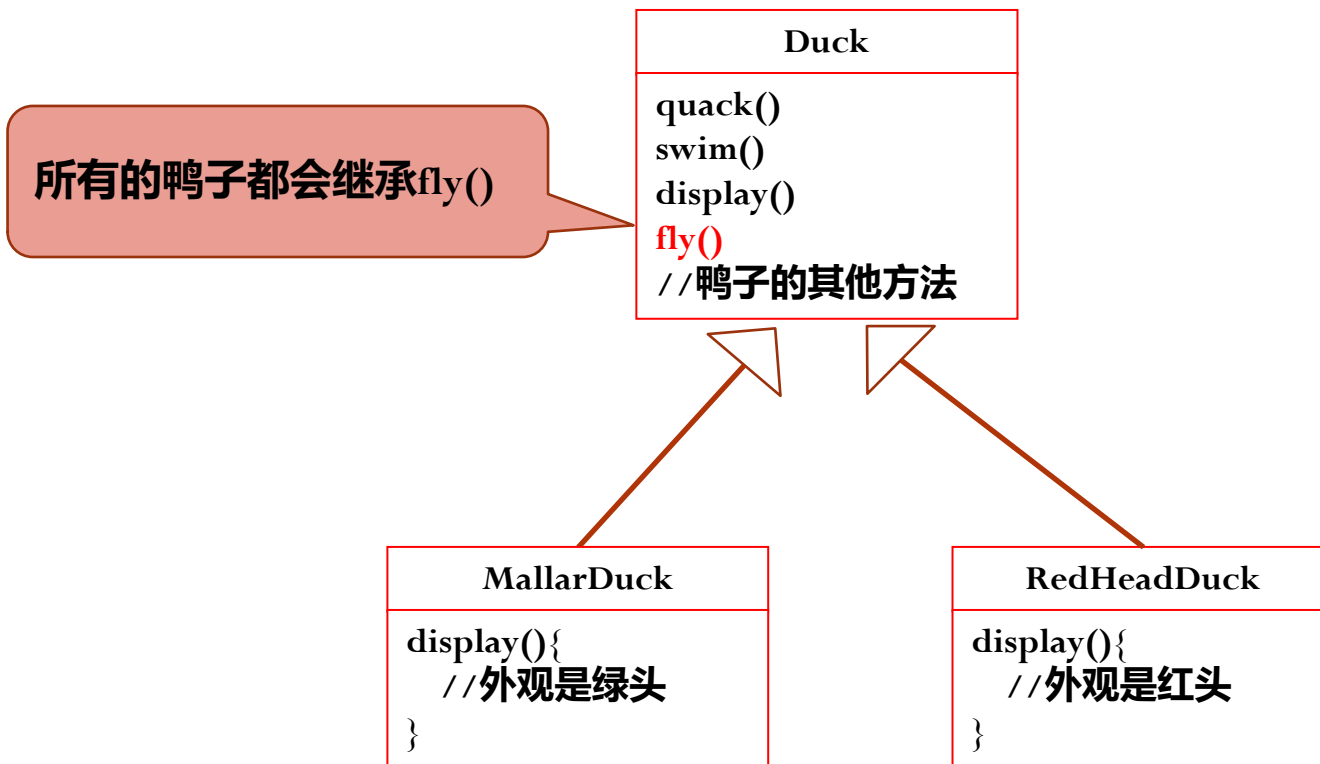
每一种鸭子的外观不同，因此超类里display()方法是抽象的

每一种具体的鸭子负责自己display()行为的实现



策略模式的由来

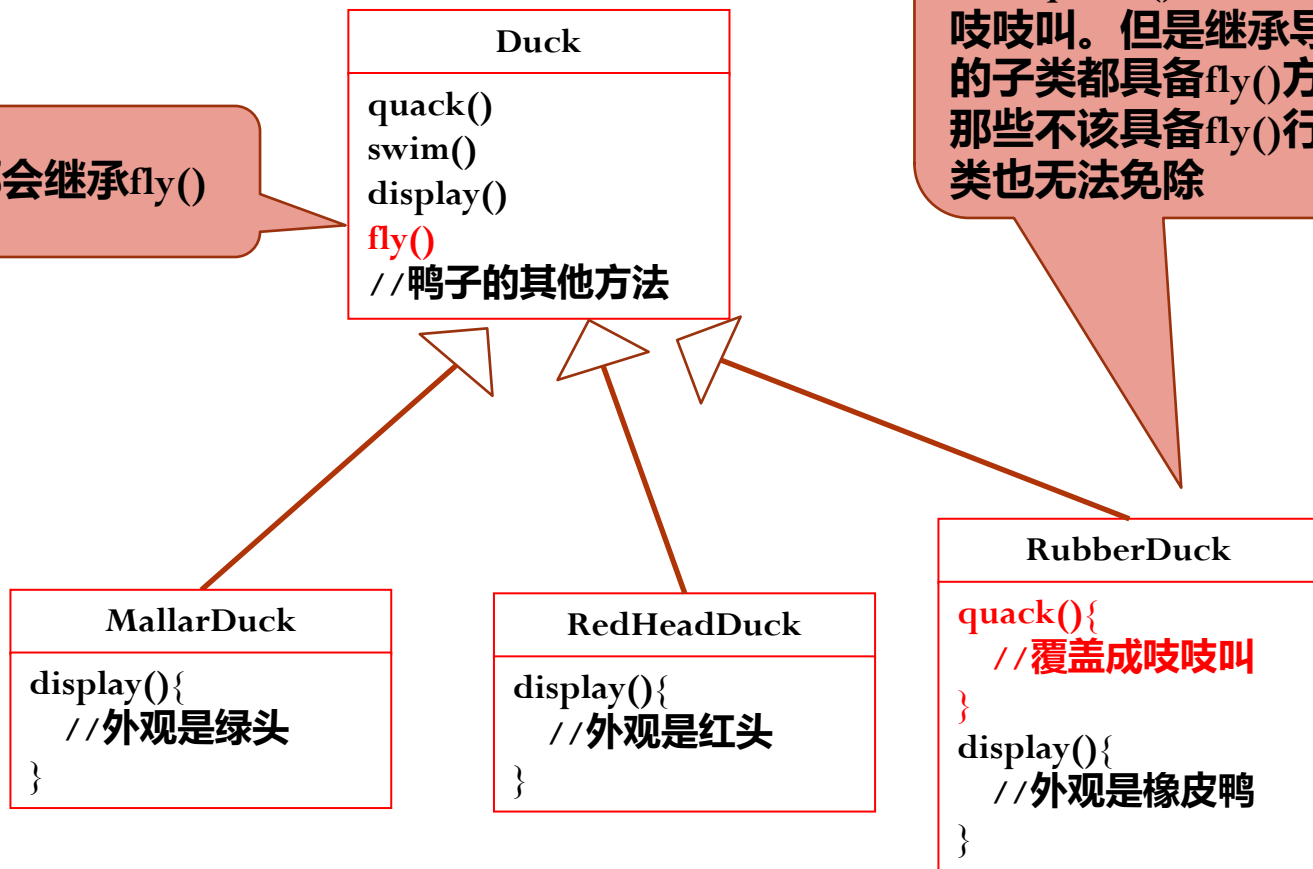
- 现在我们想让鸭子能飞
- 在超类里加上fly()方法，利用继承，所有的鸭子能飞了



策略模式的由来

□ 可怕的问题发生了：子类橡皮鸭子也能飞了

所有的鸭子都会继承fly()



橡皮鸭子不会呱呱叫，所以覆盖quack()方法，实现为吱吱叫。但是继承导致所有的子类都具备fly()方法，连那些不该具备fly()行为的子类也无法免除

策略模式的由来

□ 为了解决这个问题，可以把橡皮鸭中的fly()方法覆盖掉

RubberDuck

```
quack(){ //覆盖成吱吱叫 }  
display(){ //外观是橡皮鸭 }  
fly(){ //什么都不做 }
```

□ 但是，如果派生出诱饵鸭DecoyDuck，又会如何？诱饵鸭既不会飞，也不会叫

DecoyDuck

```
quack(){ //覆盖成什么都不做 }  
display(){ //外观是诱饵鸭 }  
fly(){ //什么都不做 }
```

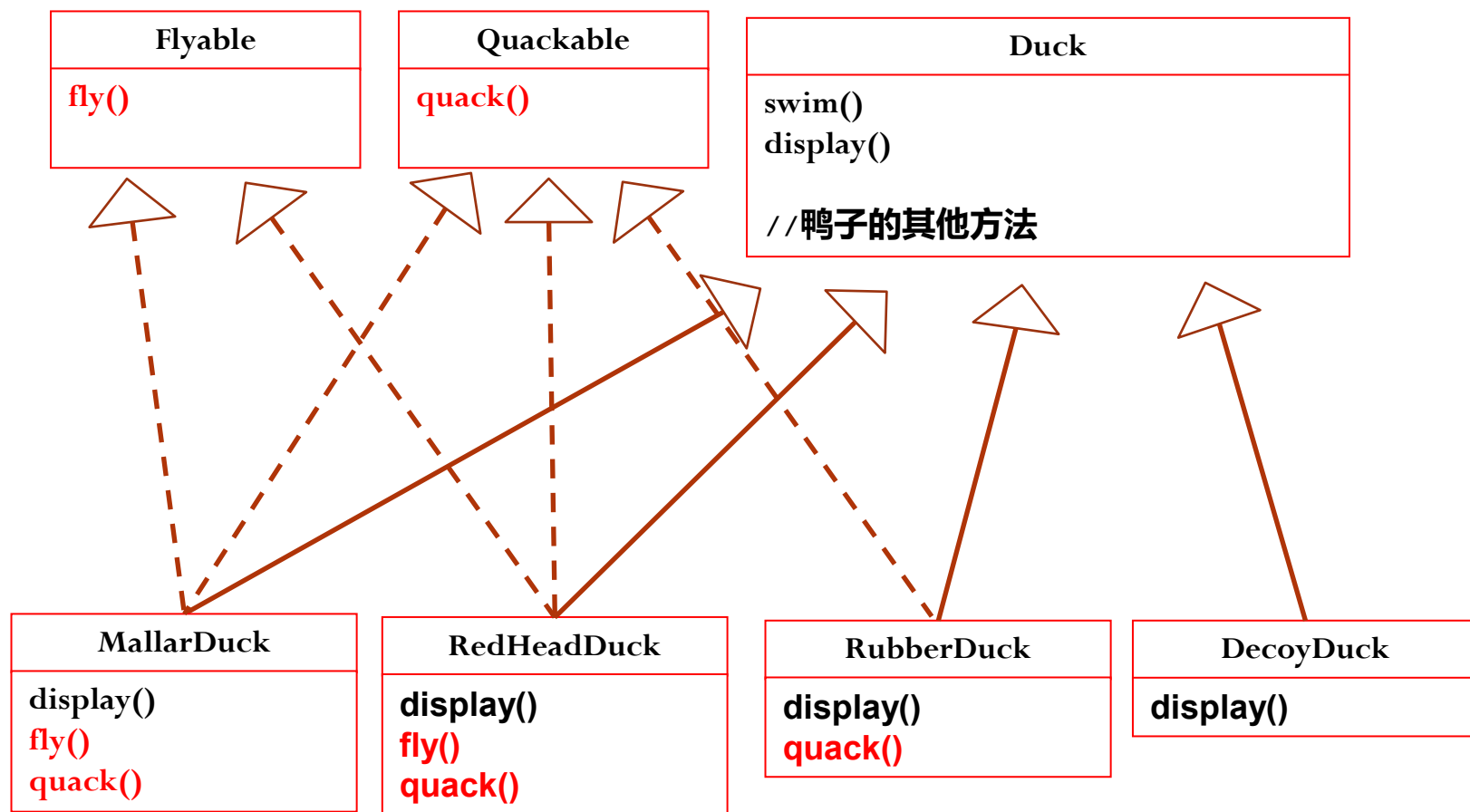
每当有新的鸭子出现，就要被迫检查并可能需要覆盖quack()和fly()，无穷无尽的噩梦

可以看到利用继承来提供鸭子的行为导致如下缺点：

- 1：代码在多个子类中重复
- 2：运行时的行为不容易改变
- 3：很难知道鸭子的全部行为
- 4：改变会牵一发而动全身，造成其它鸭子不想要的改变

策略模式的由来

□ 利用接口如何



利用接口不会再有会飞的橡皮鸭，但代码无法复用，因为每个实现接口的鸭子必须实现`quack()`和`fly()`方法

策略模式的由来

□ 利用继承和接口存在的问题

- 继承使得子类继承了不该继承的方法
- 接口不具有实现代码，因此实现接口无法达到代码的复用

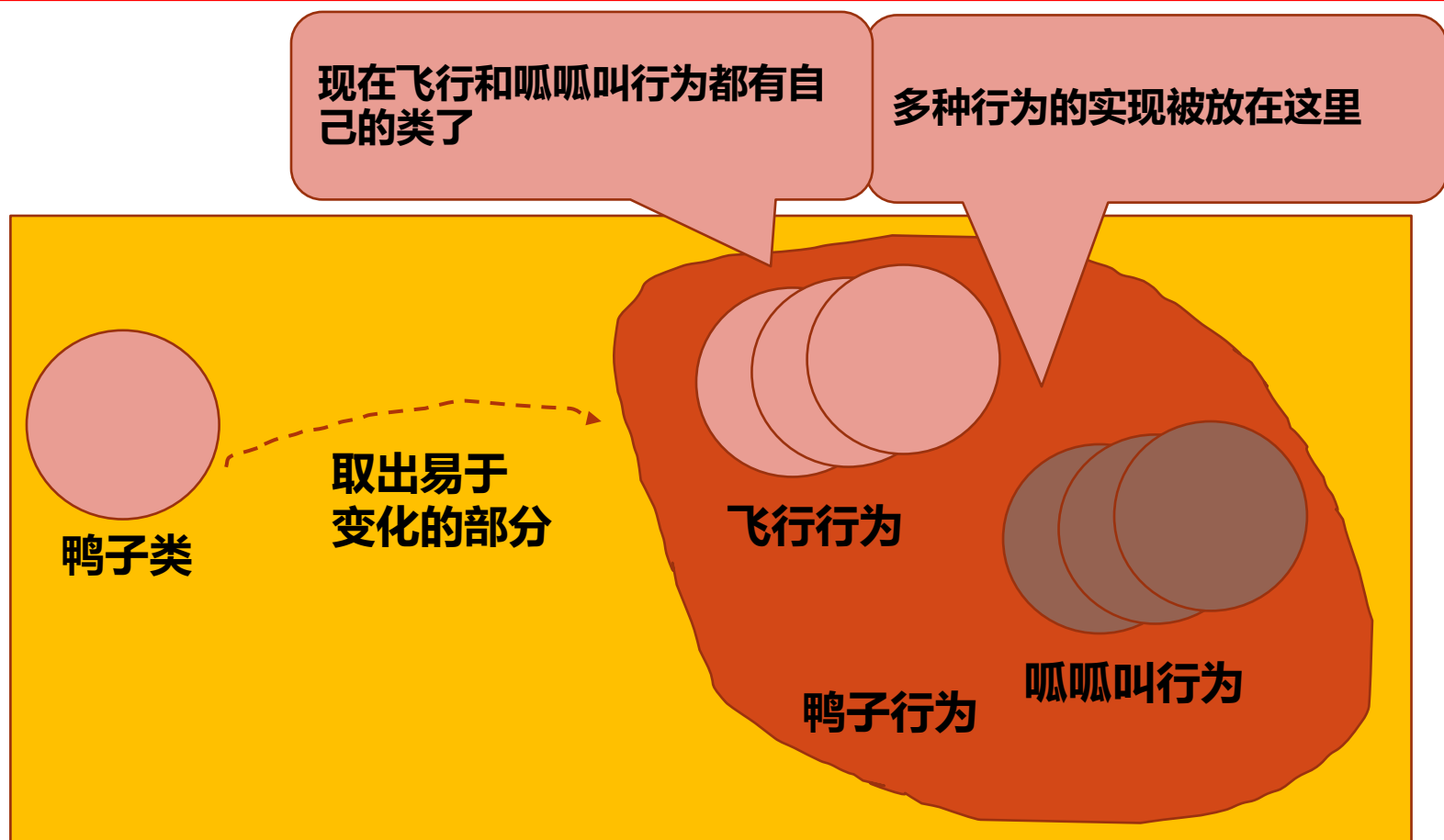
□ 幸好有个设计原则适合此状况

- 找出应用中可能需要变换之处，把它独立出来，不要和那些不需要变换的代码混在一起
- 该把鸭子的行为从Duck类中抽取出来了

策略模式的由来

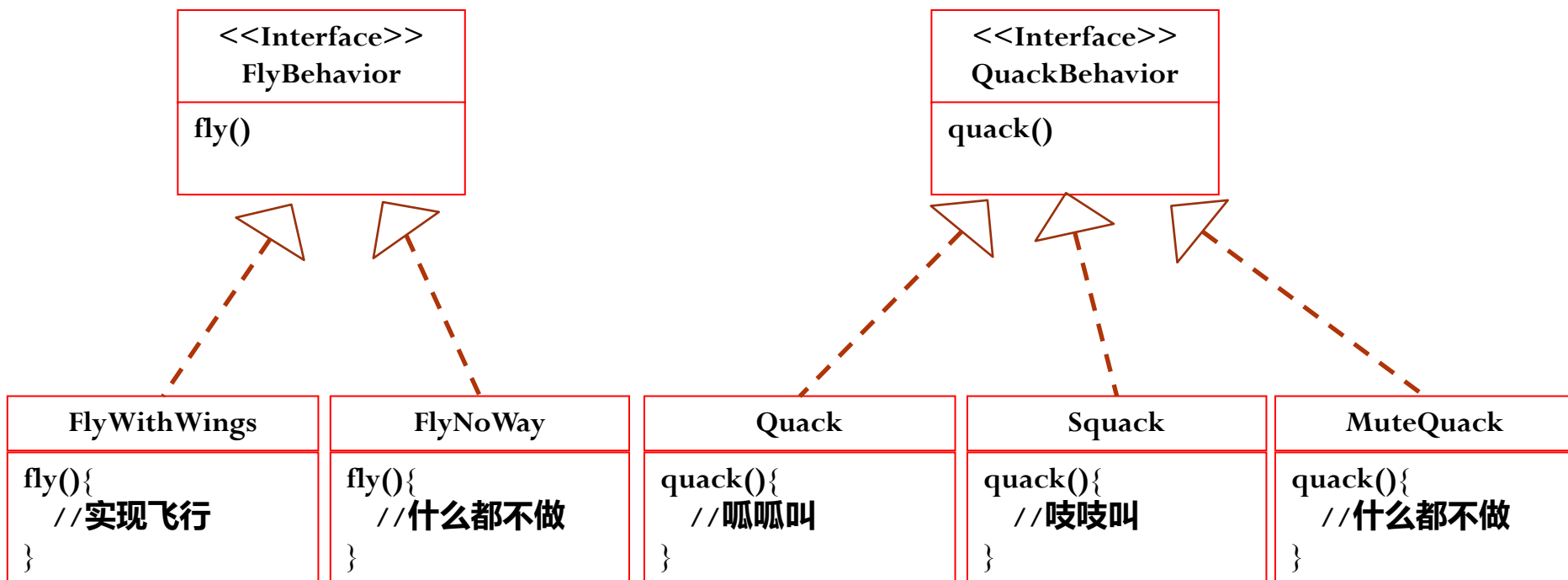
□ 分开变换和不会变化的部分

- Duck类的fly()和quack()会随着鸭子的不同而改变
- 为了把这二个行为从Duck类中分开，我们把它们从Duck类中取出来，建立一组新类来代表每个行为



策略模式的由来

□ 设计鸭子的行为：基于接口编程，不是基于实现编程



这样的设计，可以让飞行和呱呱叫的动作被其它对象复用，因为这些行为已经与鸭子无关了。而我们新增一些行为，不会影响到既有的行为类，也不会影响到“使用”到已有行为的鸭子类

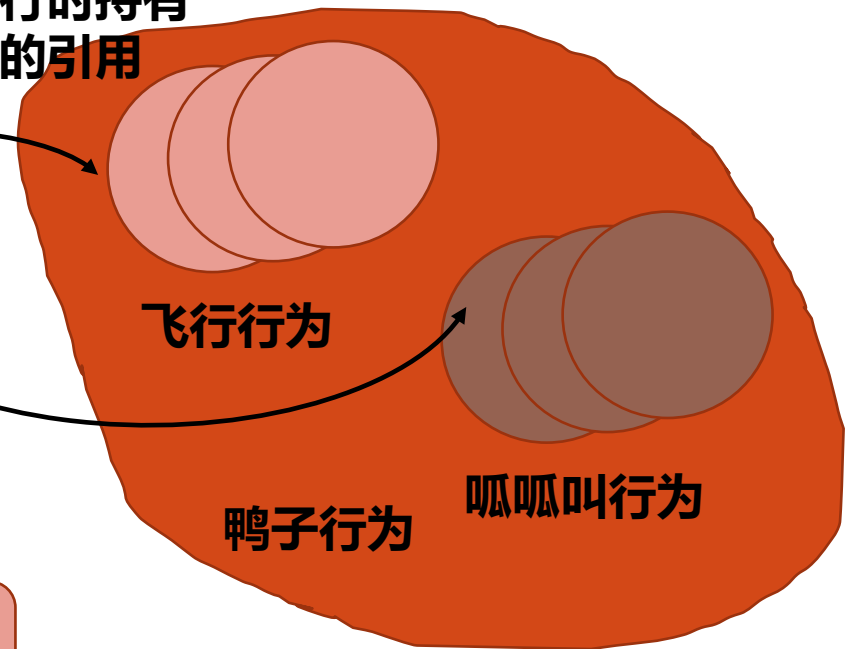
策略模式的由来

□ 整合鸭子的行为

Duck类中加入二个实例变量，
类型为接口类型

Duck
FlyBehavior flyBehavior QuackBehavior quackBehavior
swim() display() performQuack() performFly() //其它方法

实例变量在运行时持有
指定行为对象的引用



这二个方法取代fly()和quack()

策略模式的由来

□ FlyBehavior接口以及具体的飞行行为类

```
public interface FlyBehavior {  
    public void fly();  
}
```

```
public class FlyWithWings implements FlyBehavior {  
    public void fly() {  
        System.out.println("I'm flying!!");  
    }  
}
```

实现二个具体的飞行行为，你还可以实现更多的飞行行为

```
public class FlyNoWay implements FlyBehavior {  
    public void fly() {  
        System.out.println("I can't fly");  
    }  
}
```

策略模式的由来

□ QuackBehavior接口以及具体的飞行行为类

```
public interface QuackBehavior {  
    public void quack();  
}  
  
public class Quack implements QuackBehavior {  
    public void quack() {  
        System.out.println("Quack");  
    }  
}  
  
public class Squack implements QuackBehavior {  
    public void quack() {  
        System.out.println("Squeak");  
    }  
}  
  
public class MuteQuack implements QuackBehavior {  
    public void quack() {  
        System.out.println("<< Silence >>");  
    }  
}
```

实现三个具体的呱呱叫行为，你还可以实现更多的呱呱叫行为

策略模式的由来

□ Duck类

```
public abstract class Duck {  
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
    public Duck() { }  
    public void setFlyBehavior (FlyBehavior fb) {  
        flyBehavior = fb;  
    }  
    public void setQuackBehavior( QuackBehavior qb) {  
        quackBehavior = qb;  
    }  
    abstract void display();  
    public void performFly() {  
        flyBehavior.fly();  
    }  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
    public void swim() {  
        System.out.println("All ducks float, even decoys!");  
    }  
}
```

为行为接口类型声明二个引用变量，指向具体的行为实例。所有的鸭子子类都会继承它们

这二个方法可以动态地设定鸭子的飞行行为和呱呱叫行为

具体的显示行为由子类处理，因此定义为抽象方法

鸭子对象不亲自处理飞行行为，而是委托给flyBehavior引用的对象

鸭子对象不亲自处理呱呱叫行为，而是委托给quackBehavior引用的对象

策略模式的由来

□ 测试

```
public class ModelDuck extends Duck {  
    public ModelDuck() {  
        flyBehavior = new FlyNoWay();  
        quackBehavior = new Quack();  
    }  
    public void display() {  
        System.out.println("I'm a model duck");  
    }  
}
```

模型鸭一开始不会飞，但会叫

```
public class FlyRocketPowered implements FlyBehavior {  
    public void fly() {  
        System.out.println("I'm flying with a rocket");  
    }  
}
```

实现利用火箭动力的飞行行为

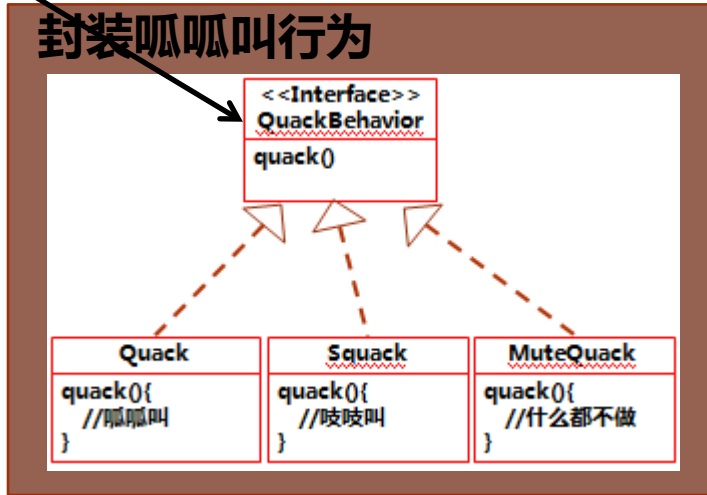
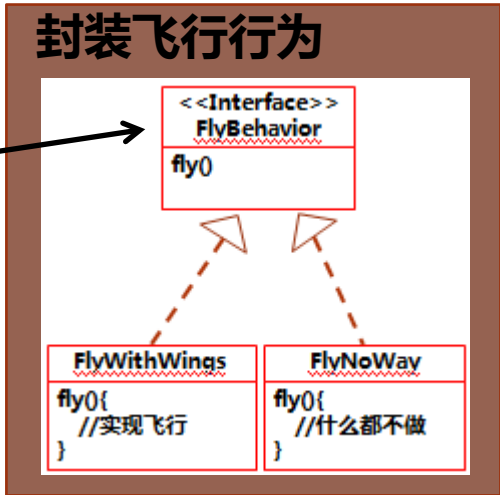
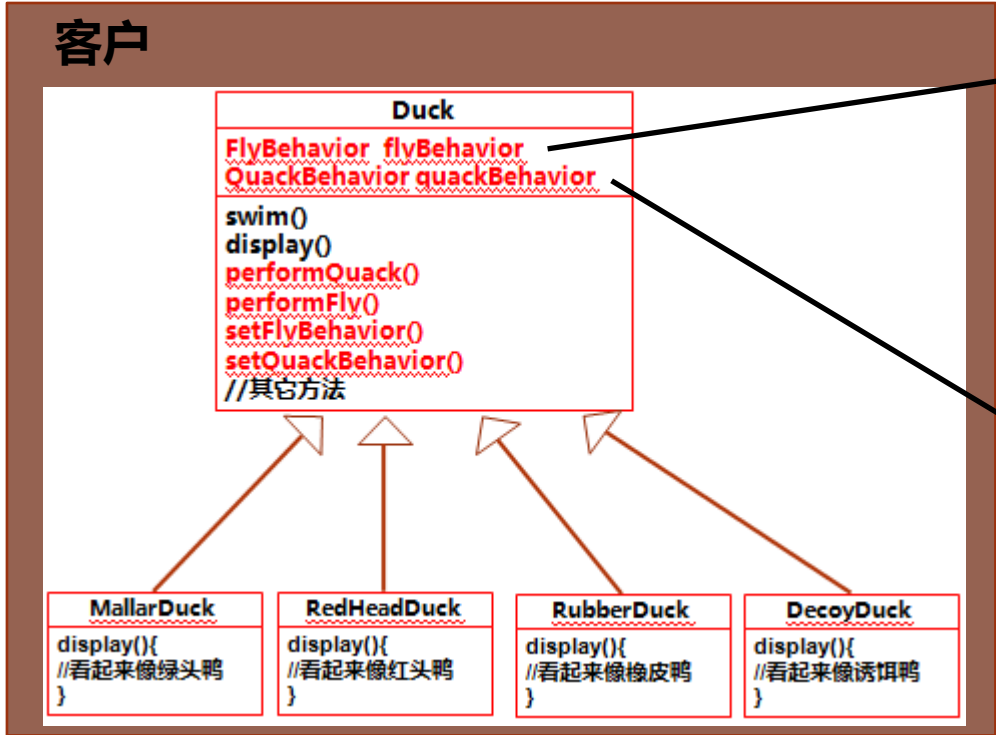
```
public class MiniDuckSimulator1 {  
    public static void main(String[] args) {  
        Duck model = new ModelDuck();  
        model.performFly();  
        model.setFlyBehavior(new FlyRocketPowered());  
        model.performFly();  
    }  
}
```

动态地改变鸭子的飞行行为，同样地，可以改变鸭子的呱呱叫行为

策略模式的整体结构

客户使用封装好的“算法”族

现在把每组行为想象成“算法”



客户和算法之间是“HAS-A”关系。”多用组合，少用继

这些“算法”是可以互换的

策略模式的定义

□ **策略模式** 定义了算法族，分别封装起来，让它们之间可以互相替换。此模式让算法的变换独立于使用算法的客户

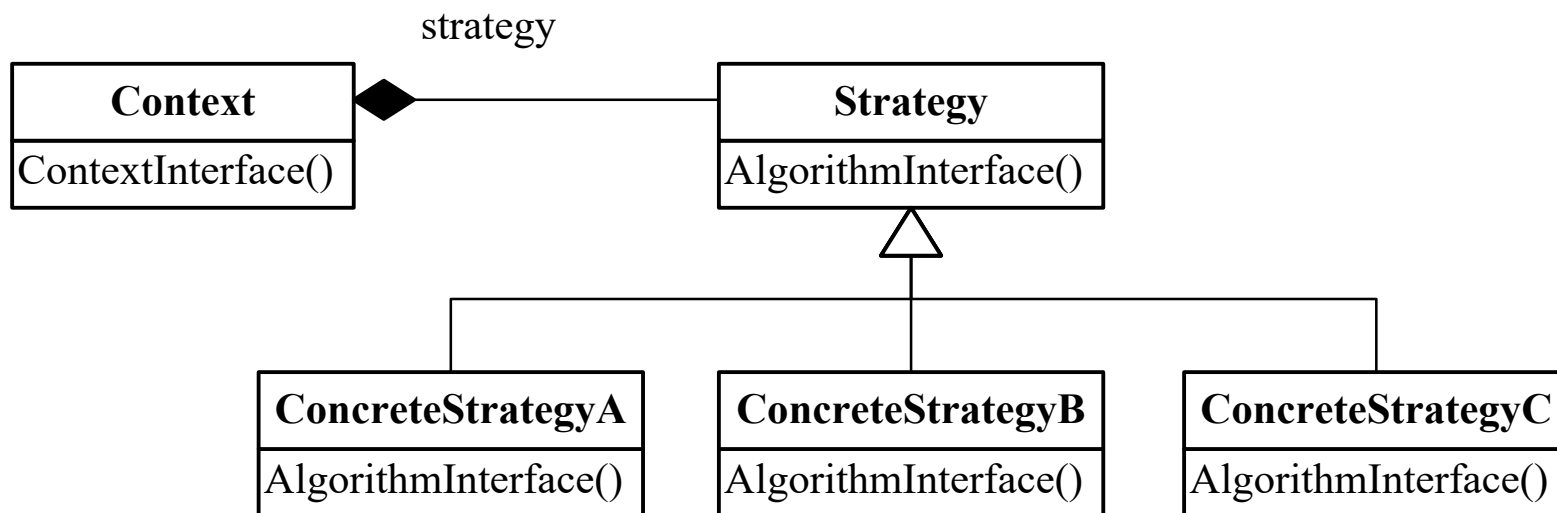
策略模式的由来

- 多种算法实现同一功能，客户希望在运行时根据上下文选择其中一个算法
- 传统策略算法的缺点
 - 使用算法的类复杂而难于维护，尤其当需要支持多种算法且每种算法都很复杂时问题会更加严重
 - 不同的时候需要不同的算法，支持并不使用的算法可能带来性能的负担
 - 算法的实现和使用算法的对象紧紧耦合在一起，使新增算法或修改算法变得十分困难，系统应对变化的能力很差
- 将每种算法的实现都剥离出来构成一个个独立算法对象，再从这些对象中抽象出公共算法接口，最后将算法接口组合到使用算法类中

策略模式的意图和适用性

- 意图：定义一系列算法，一个个进行封装，并使其可相互替换。策略模式使得算法可独立于使用它的客户而变化
- 适用场合
 - 在软件构建过程中，某些对象使用的算法可能多种多样，经常改变，如果将这些算法都编码到对象中，将会使得对象变得异常复杂；而且有时候支持不同的算法也是一个性能负担
 - 如何在运行时根据需要透明地更改对象的算法？如何将算法与对象本身解耦，从而避免上述问题？策略模式很好地解决了这两个问题

策略模式的结构



策略模式的参与者

- Strategy
 - 定义所有支持算法的公共接口
 - Context使用该接口调用某ConcreteStrategy定义的算法
- ConcreteStrategy
 - 以Strategy接口实现某具体算法
- Context
 - 用一个ConcreteStrategy对象来配置
 - 维护一个对Strategy对象的引用
 - 可定义一个接口来让Strategy访问它的数据

策略模式的效果分析

- 算法和使用算法的对象相互分离，客户程序可以在运行时动态选择算法，代码复用性好，便于修改和维护
- 用组合替代继承，效果更好。若从Context直接生成子类，也可以实现对象的多种算法，但继承使子类 and 父类紧密耦合，使Context类难以理解、难以维护和难以扩展。策略模式采用组合方式，使Context和Strategy之间的依赖很小，更利于代码复用
- 消除了冗长的条件语句序列，将不同的算法硬编码进一个类中，选择不同的算法必然要使用冗长的条件语句序列，采用策略模式将算法封装在一个个独立的Strategy类中消除了这些条件语句

模板模式的由来

- ❑ 到目前为止，所有的议题都围绕封装转
 - ❑ 封装对象的创建 – 工厂模式
 - ❑ 封装方法调用 – 命令模式
 - ❑ 封装复杂接口 – 外观模式
- ❑ 现在将要深入封装算块，好让子类可以将自己挂接进运算里
- ❑ 考虑实现冲泡咖啡和茶

模板模式的由来

□ 冲泡方法

咖啡冲泡方法

- (1) 把水煮沸
- (2) 用沸水冲泡咖啡
- (3) 把咖啡倒进杯子
- (4) 加糖和牛奶

茶冲泡方法

- (1) 把水煮沸
- (2) 用沸水浸泡茶叶
- (3) 把茶倒进杯子
- (4) 加柠檬

二者冲泡方法大致一样

初步实现

煮咖啡的算法类

```
public class Coffee{  
    public void prepareRecipe(){  
        boilWater();  
        brewCoffeeGrinds();  
        pourInCup();  
        addSugarAndMilk();  
    }  
    public void boilWater (){  
        System.out.println("Boiling water");  
    }  
    public void brewCoffeeGrinds(){  
        System.out.println("Dripping coffee through filter");  
    }  
    public void pourInCup(){  
        System.out.println("Pouring into cup");  
    }  
    public void addSugarAndMilk(){  
        System.out.println("Adding sugar and milk");  
    }  
}
```

算法的每个步骤被实现在分离的方法中

每个方法实现了算法的每个步骤

初步实现

煮茶的算法类

```
public class Tea{  
    public void prepareRecipe(){  
        boilWater();  
        steepTeaBag();  
        pourInCup();  
        addLemon();  
    }  
    public void boilWater (){  
        System.out.println("Boiling water");  
    }  
    public void steepTeaBag(){  
        System.out.println("Steeping the tea");  
    }  
    public void pourInCup(){  
        System.out.println("Pouring into cup");  
    }  
    public void addLemon(){  
        System.out.println("Adding lemon");  
    }  
}
```

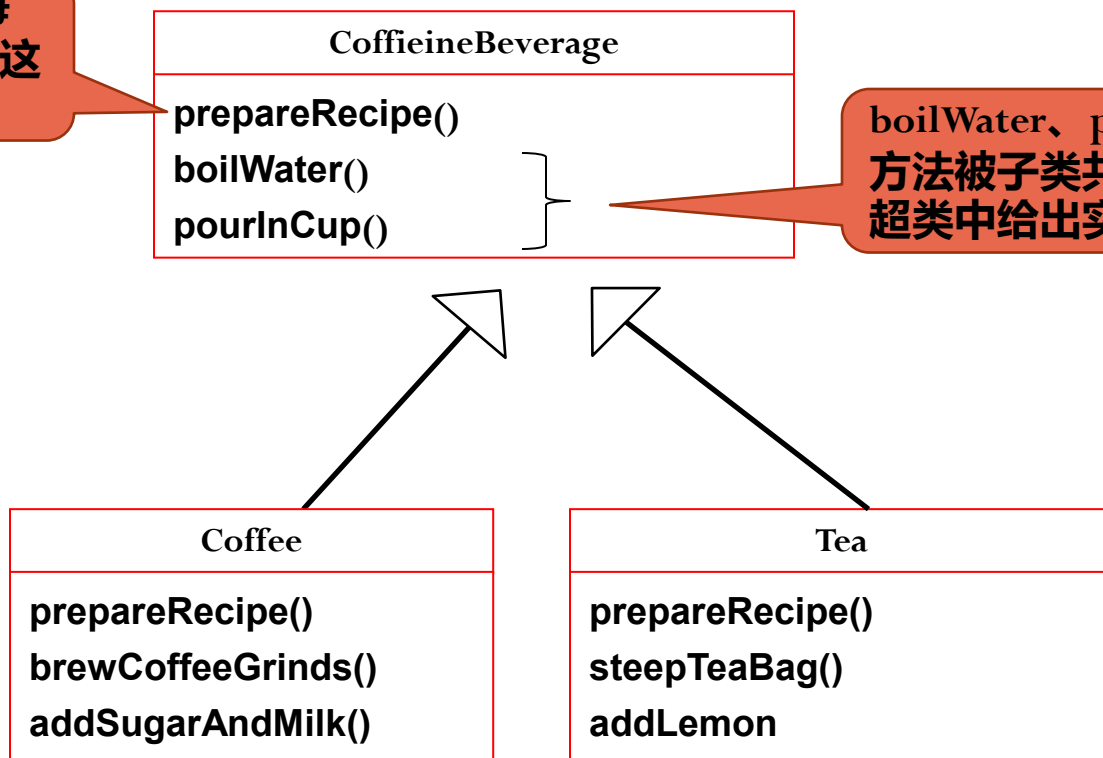
煮茶算法和煮咖啡算法很像，其中第2步和第4步不一样

boilWater和pourInCup完全一样，即这里出现了重复代码

既然二者算法如此相似，可以考虑将共同的部分抽取出来放到基类里

第一版改进

prepareRecipe方法在每个类中不一样，所以在这里定义为抽象方法



boilWater、pourInCup()方法被子类共享，所以在超类中给出实现

二个子类实现自己特有的方法，同时都实现抽象方法prepareRecipe，在prepareRecipe将特有步骤和共同步骤组合在一起

更进一步的设计

□ 冲泡咖啡和茶还有什么共同点？再研究冲泡方法

咖啡冲泡方法

- (1) 把水煮沸
- (2) 用沸水冲泡咖啡
- (3) 把咖啡倒进杯子
- (4) 加糖和牛奶

茶冲泡方法

- (1) 把水煮沸
- (2) 用沸水浸泡茶叶
- (3) 把茶倒进杯子
- (4) 加柠檬

□ 注意二种冲泡法采用了相同算法，如果采用更高级别的抽象

- (1) 把水煮沸
- (2) 用沸水泡茶或咖啡
- (3) 把饮料倒进杯子
- (4) 在饮料内加入适当的调料

(1)和(3)已经被放在基类
(2)和(4)目前没有被抽取出来放在基类，但它们本质是一样的，只是应用在不同的饮料上

抽象prepareRecipe

□ 分析子类中prepareRecipe的实现，再逐步抽象

咖啡冲泡方法

```
public void prepareRecipe(){  
    boilWater();  
    brewCoffeeGrinds();  
    pourInCup();  
    addSugarAndMilk();  
}
```

茶冲泡方法

```
public void prepareRecipe(){  
    boilWater();  
    steepTeaBag();  
    pourInCup();  
    addLemon();  
}
```

浸泡(stEEP)和冲泡(brew)其实差别不大，因此可以起一个新的方法名，如brew将二者统一起来；类似地，加糖和牛奶也和加柠檬类似，都是加调料。因此也可以用一个新方法名，如addCondiments将它们统一起来，这样一来，prepareRecipe看起来像：

```
public void prepareRecipe(){  
    boilWater();  
    brew();  
    pourInCup();  
    addCondiments();  
}
```

抽象prepareRecipe

- 现在可以将**prepareRecipe**放到超类中了，但是**brew**和**addCondiments**方法在超类中必须是抽象的

```
public abstract class CoffeeBeverage{  
    public final void prepareRecipe(){  
        boilWater();  
        brew ();  
        pourInCup();  
        addCondiments();  
    }  
    public void boilWater (){  
        System.out.println("Boiling water");  
    }  
    public void pourInCup(){  
        System.out.println("Pouring into cup");  
    }  
    public abstract void brew();  
    public abstract void addCondiments();  
}
```

现在在超类里用**prepareRecipe**来处理茶和咖啡，在方法中规定了算法的每个步骤的执行次序。我们不希望子类覆盖这个方法改变执行次序，因此声明为**final**

在超类里，**prepareRecipe**规定了算法每个步骤执行次序，因此它定义了算法的模板。

但是在超类里无法确定**brew**和**addCondiments**的实现，因此被声明为抽象方法，其具体实现留给子类实现

最后处理咖啡类和茶类

□ 现在二个子类只需要给出**brew**和**addCondiments**的具体实现即可。

```
public class Coffee extends CaffeineBeverage{
    public void brew(){
        System.out.println("Dripping coffee through filter");
    }
    public void addCondiments(){
        System.out.println("Adding sugar and milk");
    }
}

public class Tea extends CaffeineBeverage{
    public void brew(){
        System.out.println("Steeping the tea");
    }
    public void addCondiments(){
        System.out.println("Adding lemon");
    }
}
```

二个子类分别给出二个抽象方法的实现

到目前为止我们做了什么

咖啡冲泡方法

- (1) 把水煮沸
- (2) 用沸水冲泡咖啡
- (3) 把咖啡倒进杯子
- (4) 加糖和牛奶

茶冲泡方法

- (1) 把水煮沸
- (2) 用沸水浸泡茶叶
- (3) 把茶倒进杯子
- (4) 加柠檬

茶和咖啡冲泡法基本类似，
将冲茶和咖啡的步骤泛化，
并放在超类

泛化

一些步骤依赖
进行子类

咖啡因饮料

- (1) 把水煮沸
- (2) 冲泡
- (3) 把饮料倒进杯子
- (4) 加调料

泛化

一些步骤依赖
进行子类

咖啡

- (2) 用沸水冲泡咖啡
- (4) 加糖和牛奶

咖啡因饮料规定算法的步骤，
其中一些步骤亲自执行，
一些步骤交子类实现

茶

- (2) 用沸水浸泡茶叶
- (4) 加柠檬

认识模板方法

□ 看看咖啡因类的结构，它包含了“模板方法”

```
public abstract class CaffeineBeverage{
```

```
    public final void prepareRecipe(){
```

```
        boilWater();
```

```
        brew ();
```

```
        pourInCup();
```

```
        addCondiments();
```

```
    }
```

```
    public void boilWater (){ //实现 }
```

```
    public void pourInCup(){ //实现 }
```

```
    public abstract void brew();
```

```
    public abstract void addCondiments();
```

```
}
```

prepareRecipe是模板方法

1: 它用作一个算法的模板，在这个模板中，算法的每个步骤都被一个方法代表，并且规定了步骤的顺序

2: 某些方法由超类实现，被子类共享

3: 某些方法则必须由子类处理，因此在超类里被声明为抽象方法

模板方法定义了一个算法的步骤，并允许子类为每一个步骤提供实现

现在开始泡茶

□ 通过泡茶理解模板方法如何工作

1 首先创建一个茶对象

```
Tea myTea = new Tea();
```

2 然后调用模板方法

```
myTea.prepareRecipe();//它会依照算法来制作茶饮料
```

2.1 首先把水煮沸

```
boilWater(); //这件事是在超类中进行
```

2.2 接着泡茶

```
brew(); //这件事是在子类中进行，利用多态调用子类  
//brew方法
```

2.3 把茶倒进杯子

```
pourInCup(); //这件事是在超类中进行
```

2.4 最后加调料

```
addCondiments(); //这件事是在子类中进行，利用多  
//态调用子类addCondiments
```

```
boilWater();  
brew();  
pourInCup();  
addCondiments();
```

prepareRecipe方法控制了算法，没有人能改变它。这个方法也会依赖子类提供某些或所有步骤的实现

CoffeineBeverage

```
prepareRecipe()  
boilWater()  
pourInCup()
```



Tea

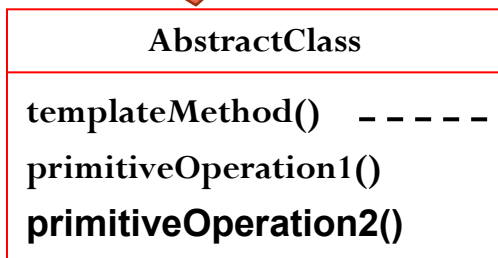
```
brew() 173  
addCondiments();
```

定义模板方法模式

抽象类包含了模板方法templateMethod

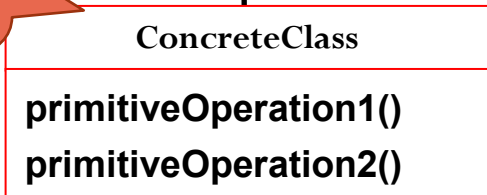
模板方法在实现算法的过程中，用到了这二个原语操作。模板方法本身和这二个操作的具体实现之间解耦了

同时将模板方法中需要子类实现的原语定义为抽象方法



primitiveOperation1()
primitiveOperation2()

可能有许多具体类，每个都实现了模板中的抽象方法



这个具体类实现抽象的操作，当模板方法调用抽象操作时，会调用到具体类的实现

模板方法在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤

看看抽象类是如何定义的

这是抽象类，被声明为抽象的，用来作为基类，其子类必须实现其抽象操作

这是模板方法，它被声明为 `final`，以避免子类改变这个算法中步骤的次序

```
public abstract class AbstractClass{  
    public final void templateMethod(){  
        primitiveOperation1();  
        primitiveOperation2();  
        concreteOperation();  
    }  
    public abstract void primitiveOperation1();  
    public abstract void primitiveOperation2();  
  
    public void concreteOperation(){  
        //具体实现  
    }  
}
```

模板方法定义了一连串的步骤，每个步骤由一个方法代表

模板方法中这二个抽象步骤必须由具体子类实现

这是模板方法中的具体步骤，可以由基类实现

给模板方法安装钩子

```
public abstract class AbstractClass{  
    public final void templateMethod(){  
        primitiveOperation1();  
        primitiveOperation2();  
        concreteOperation();  
        hook();  
    }
```

模板方法定义了一连串的步骤，每个步骤由一个方法代表。现在在模板方法里添加一个钩子方法

```
    public abstract void primitiveOperation1();  
    public abstract void primitiveOperation2();  
  
    public void concreteOperation(){  
        //具体实现  
    }
```

模板方法中这二个抽象步骤必须由具体子类实现

```
    void hook(){  
        //什么都不做  
    }
```

这是模板方法中的具体步骤，可以由基类实现

我们也可以有“默认不做事”的方法，这种方法称为“hook”。子类视情况决定是不是要覆盖它。下一页就可以知道钩子的用途

给模板方法安装钩子

- 钩子是一种声明在抽象类中的方法，但只有空的或默认的实现。钩子的存在，可以让子类有能力对算法的不同点进行挂钩。要不要挂钩，由子类自行决定

```
public abstract class CoffeeBeverageWithHook{
    public final void prepareRecipe(){
        boilWater();
        brew ();
        pourInCup();
        if( customWantCondiments() ){
            addCondiments();
        }
    }
    public void boilWater (){ System.out.println("Boiling water"); }
    public void pourInCup(){ System.out.println("Pouring into cup");}
    public abstract void brew();
    public abstract void addCondiments();
    public boolean customWantCondiments() {
        return true;
    }
}
```

这里加上了一个条件语句，该条件是否成立，是由具体方法`customWantCondiments`决定。如果客户想要调料，才调用`addCondiments`

在基类给出`customWantCondiments`方法的缺省实现。这就是一个钩子，子类可以改变这个缺省实现，但不是必须的

使用钩子

```
public class CoffeWithHook extends CoffeineBeverageWithHook{  
    public void brew(){ System.out.println("Dripping coffee through filter");}  
  
    public void addCondiments(){ System.out.println("Adding sugar and milk");}  
  
    public boolean customWantCondiments(){  
        String answer = getUserInput();  
        if(answer.toLowerCase().startsWith("y")){  
            return true;  
        } else {  
            return false;  
        }  
    }  
    public String getUserInput(){  
        //从键盘获取用户的选择  
        //如果用户选择"y",表示用户需要调料; 否则表示用户不需要调料  
    }  
}
```

为了使用钩子，必须在子类覆盖钩子方法。
让用户输入他们对是否加调料的决定，根据用户的输入，返回true或false。

在这里，钩子方法决定了是否执行算法中的某些步骤：是否加调料。

执行测试程序

```
public class BeverageTestDrive{  
    public static void main(String[] args){  
        CaffeineBeverageWithHook teaHook = new TeaWithHook ();  
        CaffeineBeverageWithHook coffeeHook = new CoffeWithHook ();  
  
        System.out.println("\nMaking tea...");  
        teaHook .prepareRecipe();  
  
        System.out.println("\nMaking coffee...");  
        coffeeHook .prepareRecipe();  
  
    }  
}
```

钩子的讨论

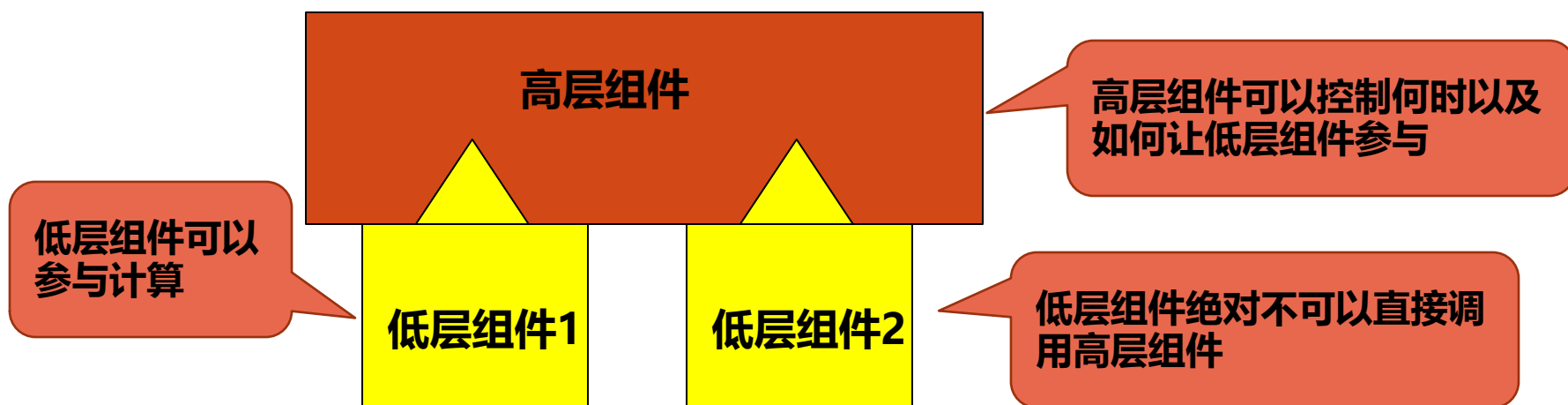
钩子能够作为条件控制，影响抽象类中的算法流程。算法流程的选择交给子类去决定。

钩子同样能够作为循环迭代的执行控制，影响抽象类中的算法迭代执行。算法迭代何时结束交给子类去决定。

钩子机制可以让子类决定算法的执行过程。

好莱坞原则

- ❑ 模板模式体现了新的设计原则，称为好莱坞原则
 - ❑ 别调用（打电话）给我们，我们会调用（打电话）给你们
- ❑ 好莱坞原则可以防止“依赖腐败”
 - ❑ 高层组件依赖低层组件，而低层组件又依赖高层组件，而高层组件又依赖边侧组件，而边侧组件又依赖底层组件。在这种情况下，没有人可以轻易搞清楚系统是如何设计的

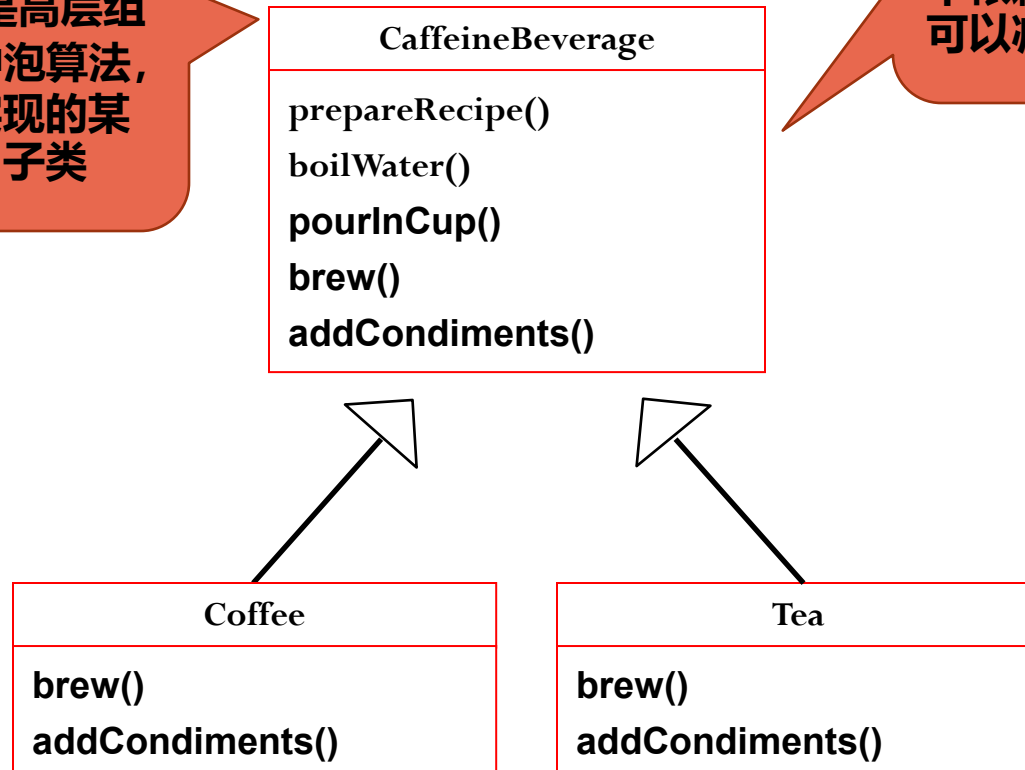


在好莱坞原则下，我们允许低层组件将挂接到系统上，但高层组件会决定什么时候和怎样使用这些低层组件。换句话说，高层组件对待低层组件的方式是：别调用我们，我们会调用你

好莱坞原则和模板方法

CaffeineBeverage是高层组件，它可以控制冲泡算法，只有在需要子类实现的某个方法时，才调用子类

饮料的客户代码只依赖于CaffeineBeverage抽象，而不依赖于具体的子类。这可以减少整个系统的依赖。



这些子类只简单用来提供一些实现细节

Tea和Coffee绝对不会直接调用抽象类

模板方法模式的由来

- 关注类对象职责分配，实现父类和子类对象之间职责的划分
- 将多个类的共有内容提取到一个模板中的想法，是模板模式的重要思想
- 使用继承机制使得父类和子类之间达到分工合作目的，父类完成流程（算法、框架），子类实现其具体工作

模板方法模式的意图和适用性

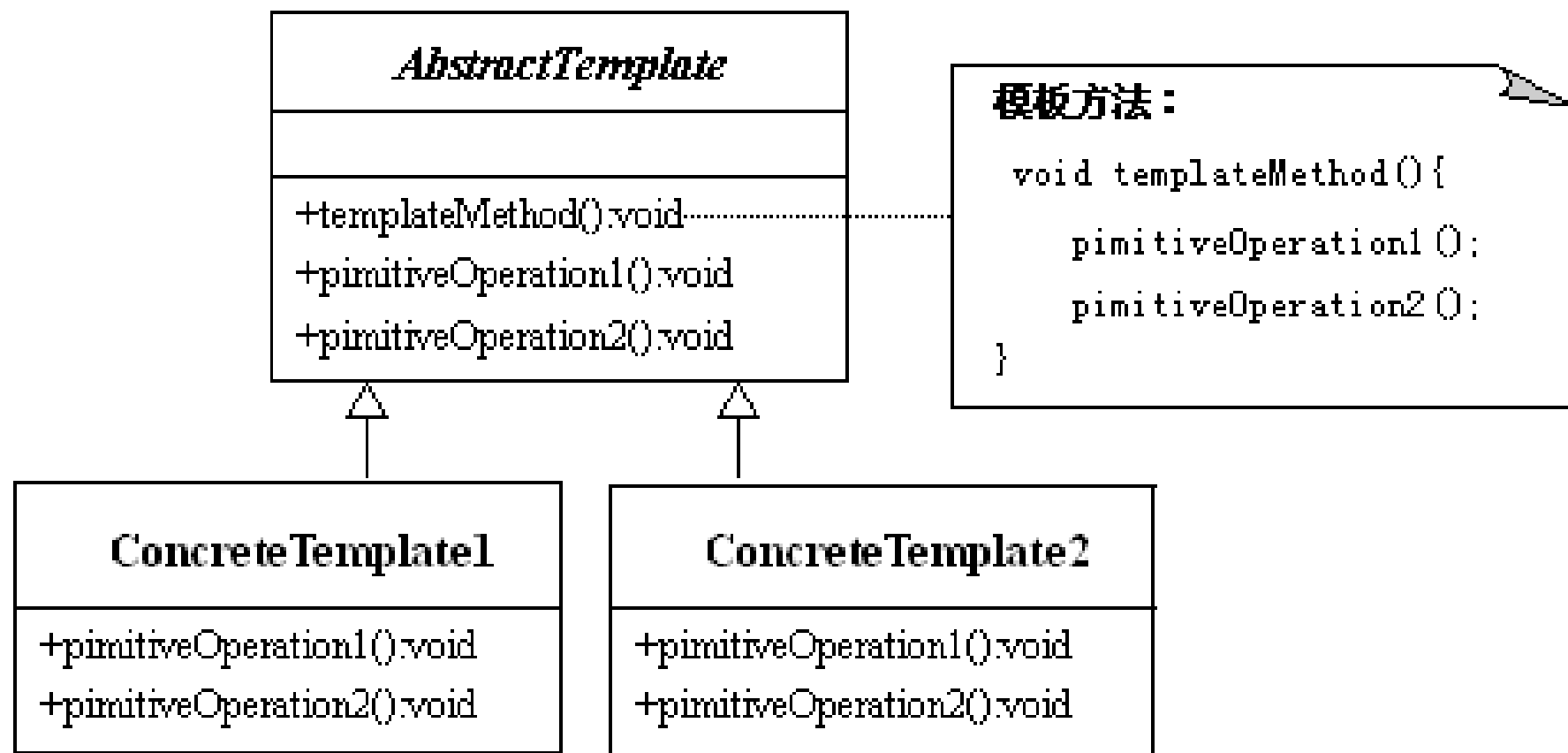
□ 意图

- 定义操作中算法的骨架，将一些步骤的执行延迟到其子类中，子类不需要改变算法结构即可重定义算法的某些步骤

□ 适用场合

- 具有统一的操作步骤或操作过程，具有不同的操作细节，即存在多个具有同样操作步骤的应用场景，但某些具体的操作细节却各不相同

模板方法模式的结构



模板方法模式的参与者

□ AbstractTemplate

- 定义一个“模板” (或算法骨架)。子类(使用该模板的客户)重定义模板中的操作(相当于根据模板填写内容)

□ ConcreteTemplate

- 实现模板定义的操作以完成算法中与特定子类相关的步骤

模板方法模式的效果分析

- ❑ 模板方法是一种代码复用技术，模板提取了“子类”的公共行为
- ❑ 模板方法导致一种反向的控制结构：“你别来找我，让我去找你”，即：一个父类调用子类的操作，而不是相反
- ❑ 可通过在抽象模板定义模板方法给出成熟算法步骤，同时又不限制步骤细节，具体模板实现算法细节不会改变整个算法骨架
- ❑ 在抽象模板模式中，可以通过钩子方法对某些步骤进行挂钩，具体模板通过钩子可以选择算法骨架中的某些步骤

JAVA API中的模板模式

□ Java API中的Arrays类的静态方法提供了数组元素排序功能

```
public static void sort(Object[] a) {  
    Object[] aux = (Object[])a.clone();  
    mergeSort(aux,a,0,a.length,0);  
}
```

为了便于解释，代码被简化。详细源码可参看Java API源码。

Merge采用冒泡排序算法。

```
private static void mergeSort(Object src[],Object[] dest, int low, int high,int off){  
    for(int i = low; i < high; i++){  
        for(int j = i; j > low &&  
            ((Comparable)dest[j-1]).compareTo((Comparable)dest[j]) > 0; j--)  
        {  
            swap(dest,j,j-1);  
        }  
    }  
    return;  
}
```

这是一个具体方法，已经在数组类里定义

mergeSort()包含排序算法，此算法依赖于compareTo()方法。因此mergeSort是模板方法，依赖于被比较对象的compareTo方法

JAVA API中的模板模式：Comparable接口

- 有时需要比较二个对象，但不同类型对象的比较具有不同的含义，因此Java定义了Comparable接口。
- 因此，任何需要比较对象的类，都要实现该接口。
- 该接口定义如下：

```
package java.lang;  
  
public interface Comparable{  
    public int compareTo (Object o);  
}
```

- compareTo判断this对象相对于给定对象o的顺序，当this对象小于、等于或大于给定对象o时，分别返回负数、0或正数

Comparable接口实现可比较的Circle类

```
public class Circle implements Comparable {  
    private double radius = 1.0;  
    Circle() {}  
    Circle(double r) { radius = r;}  
    public int compareTo(Object o) {  
        if (this.radius > ((Circle)o).radius) return 1;  
        else if (this.radius < ((Circle)o).radius) return -1;  
        else return 0;  
    }  
}
```

//测试

```
Circle[] ca = { new Circle(1.0), new Circle(2.0), new Circle(3.0)};  
Arrays.sort(ca);
```