

# 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](mailto:guxiwu@mail.hust.edu.cn)

# 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。Decorator提供了一种给类增加职责的方法，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  
};
```

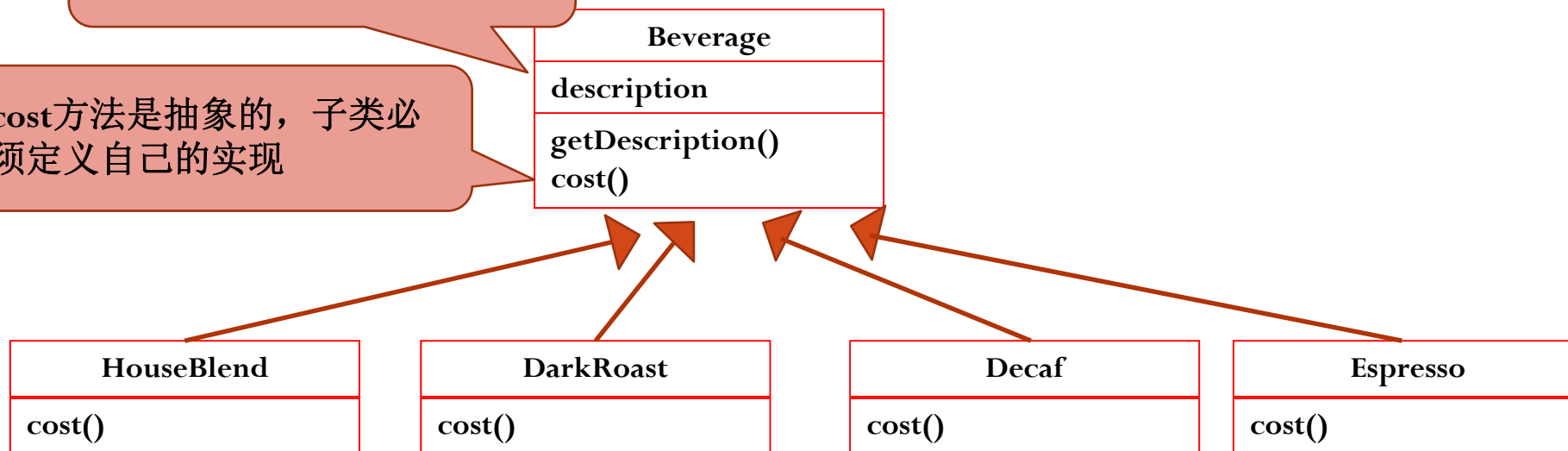
//这样C就具有A的行为f和B的行为g,达到了多重继承的效果

# 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

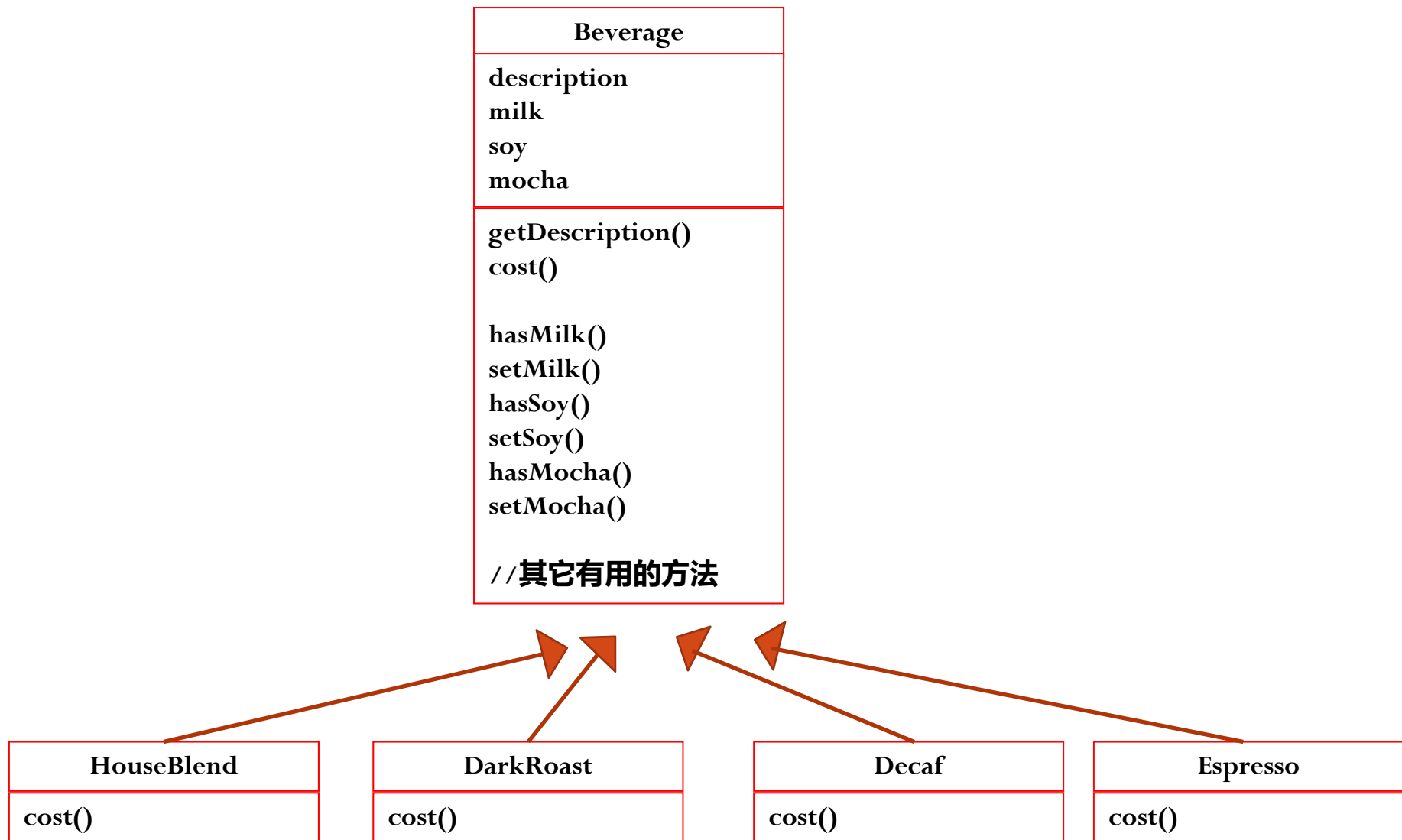
□ 能不能从基类下手，利用实例变量+继承，来追踪这些调料？

cost不再是抽象方法，而是  
计算要加入的各种调料的价  
钱

Beverage	
description	
milk	
soy	
mocha	
getDescription()	
cost()	
hasMilk()	}
setMilk()	
hasSoy()	
setSoy()	
hasMocha()	
setMocha()	
//其它有用的方法	

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、甚至是涉及超类时还没有想到的职责加到对象上

# 认识装饰者模式

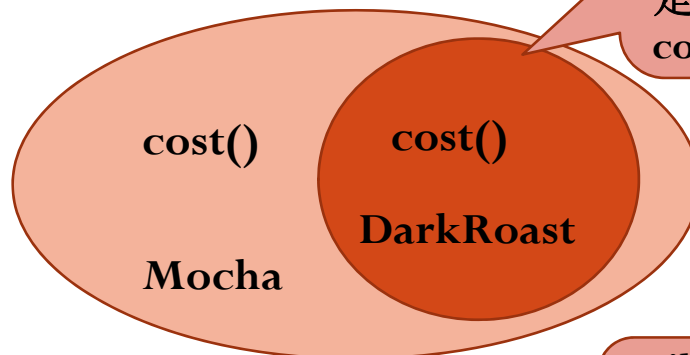
- ❑ 现在知道了类继承带来的问题：类数量爆炸、设计死板、基类加入的新功能不适合所有子类
- ❑ 现在采用不一样的做法：以饮料为主体，然后在运行时以调料来“装饰”饮料，具体步骤为：
  - ❑ 拿一个饮料对象,如深焙咖啡(DarkRoast)
  - ❑ 以摩卡(Mocha)对象装饰它
  - ❑ 以牛奶(Milk)对象装饰它
  - ❑ 调用cost方法，并依赖委托（delegate）将调料的价钱加上去

### 1 以DarkRoast对象开始



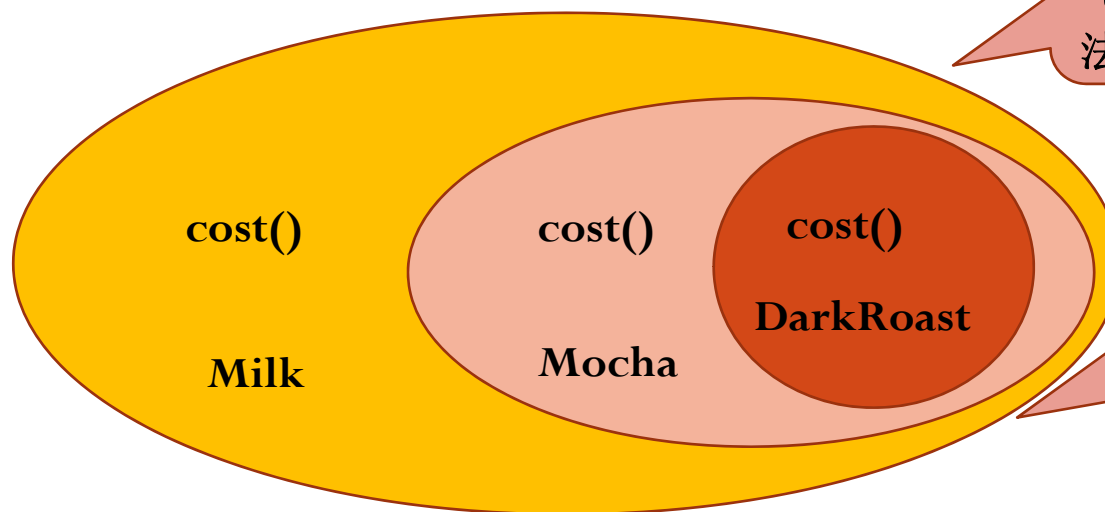
DarkRoast继承自Beverage，有cost方法

### 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



Mocha对象是装饰者，它的类型“反映”了被装饰对象（Beverage）。所谓反映就是二者类型一致。因此也有cost方法

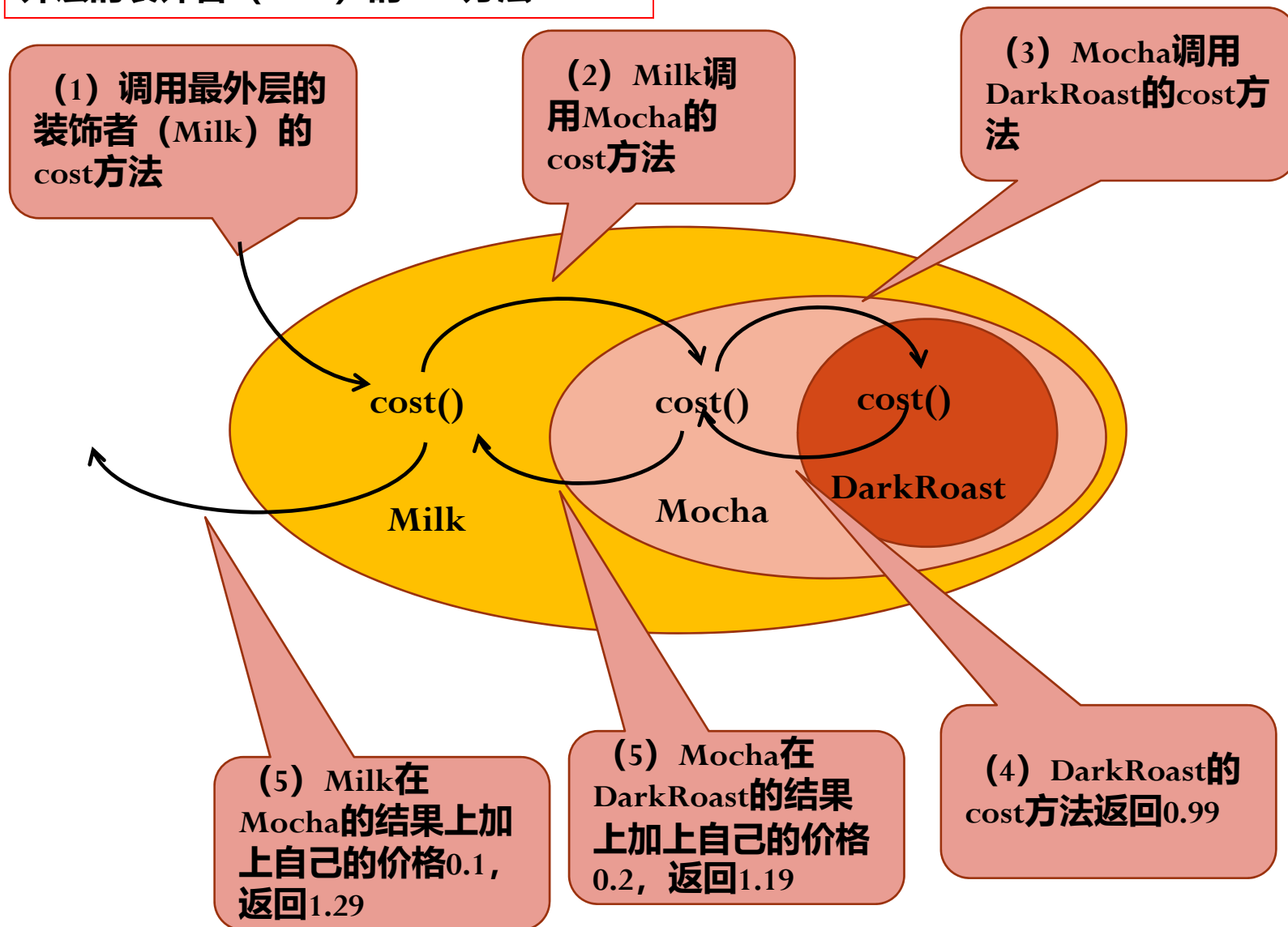
### 3 顾客想要Milk，所以建立Milk装饰对象，并用它将Mocha对象包(wrap)起来



Milk对象是装饰者，它的类型“反映”了被装饰对象（Beverage），也有cost方法

因此被Mocha和Milk装饰的对象仍然是Beverage

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

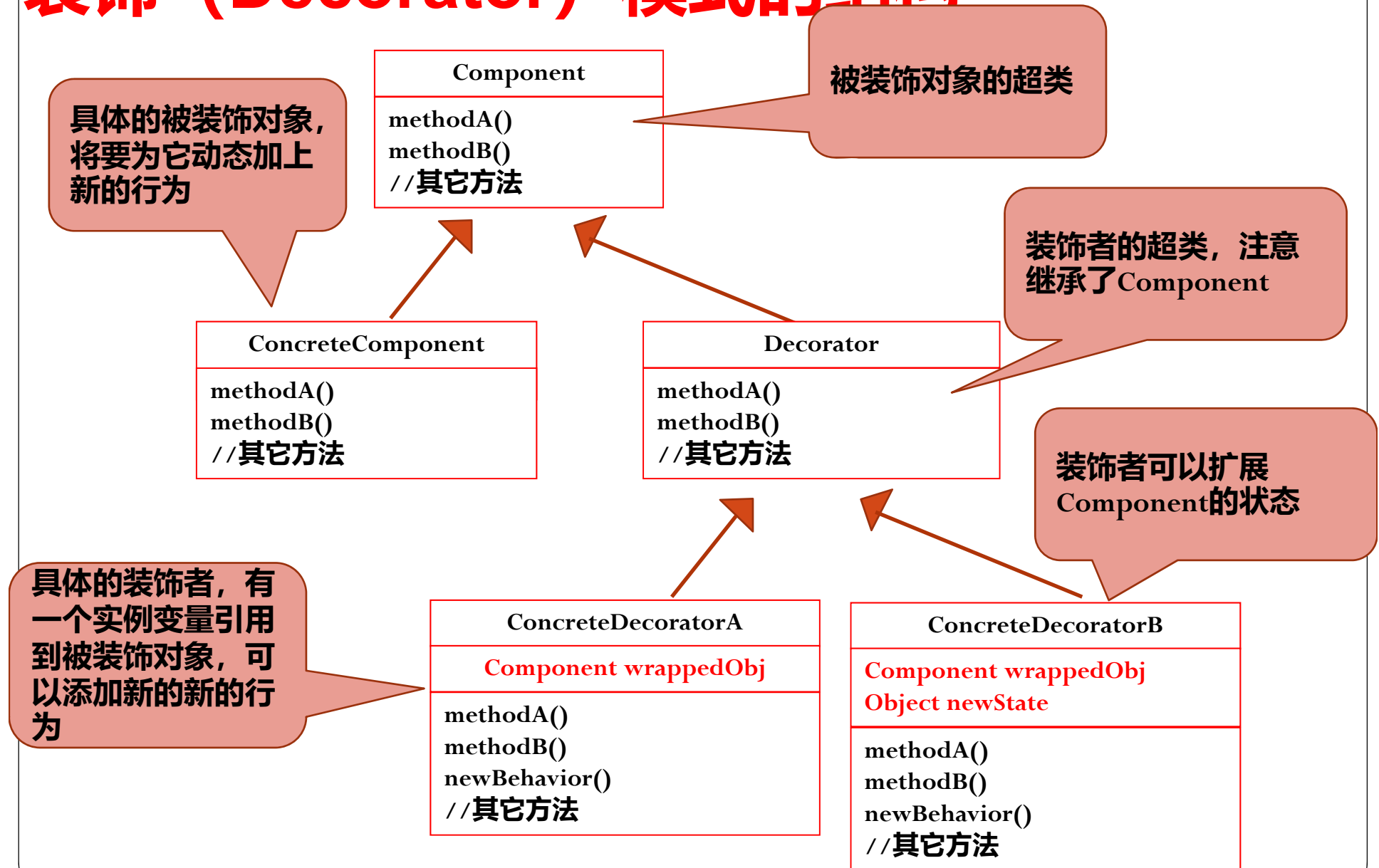
- 可以用一个或多个装饰者装饰一个对象

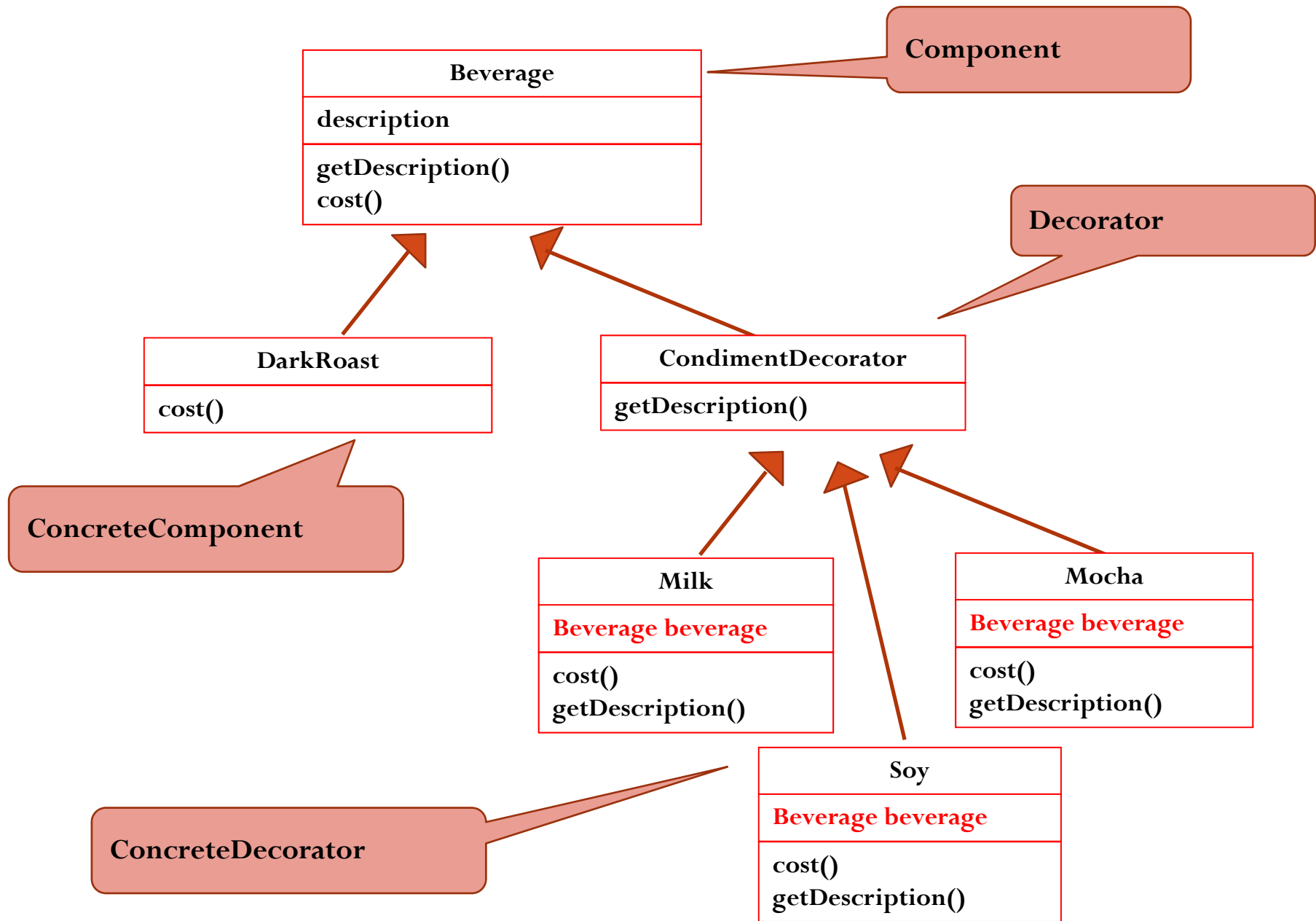
- 既然装饰者和被装饰者有相同的超类型，所以在任何需要原始对象（被装饰的）场合，可以用装饰过的对象代替它

- 装饰者可以在所委托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";  
    }  
  
    public double cost(){  
        return 0.99;    //返回饮料的价格  
    }  
}
```

具体的饮料类（被装饰对象）

```
public class HouseBlend extends Beverage{  
    public HouseBlend (){  
        description = "House Blend Coffee";  
    }  
  
    public double cost(){  
        return 0.89;    //返回饮料的价格  
    }  
}
```

具体的饮料类（被装饰对象）

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

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

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：

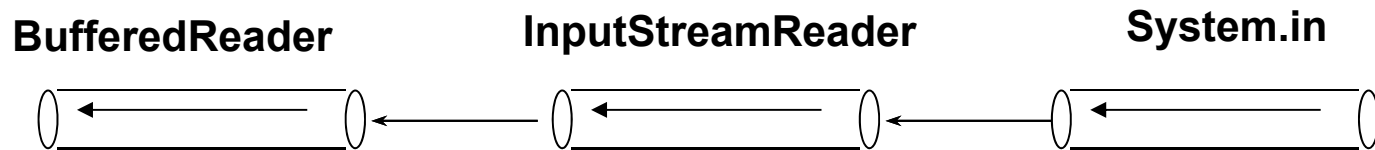
```
Reader ins = System.in;
```

Reader insReader =

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

Reader inFromUser =

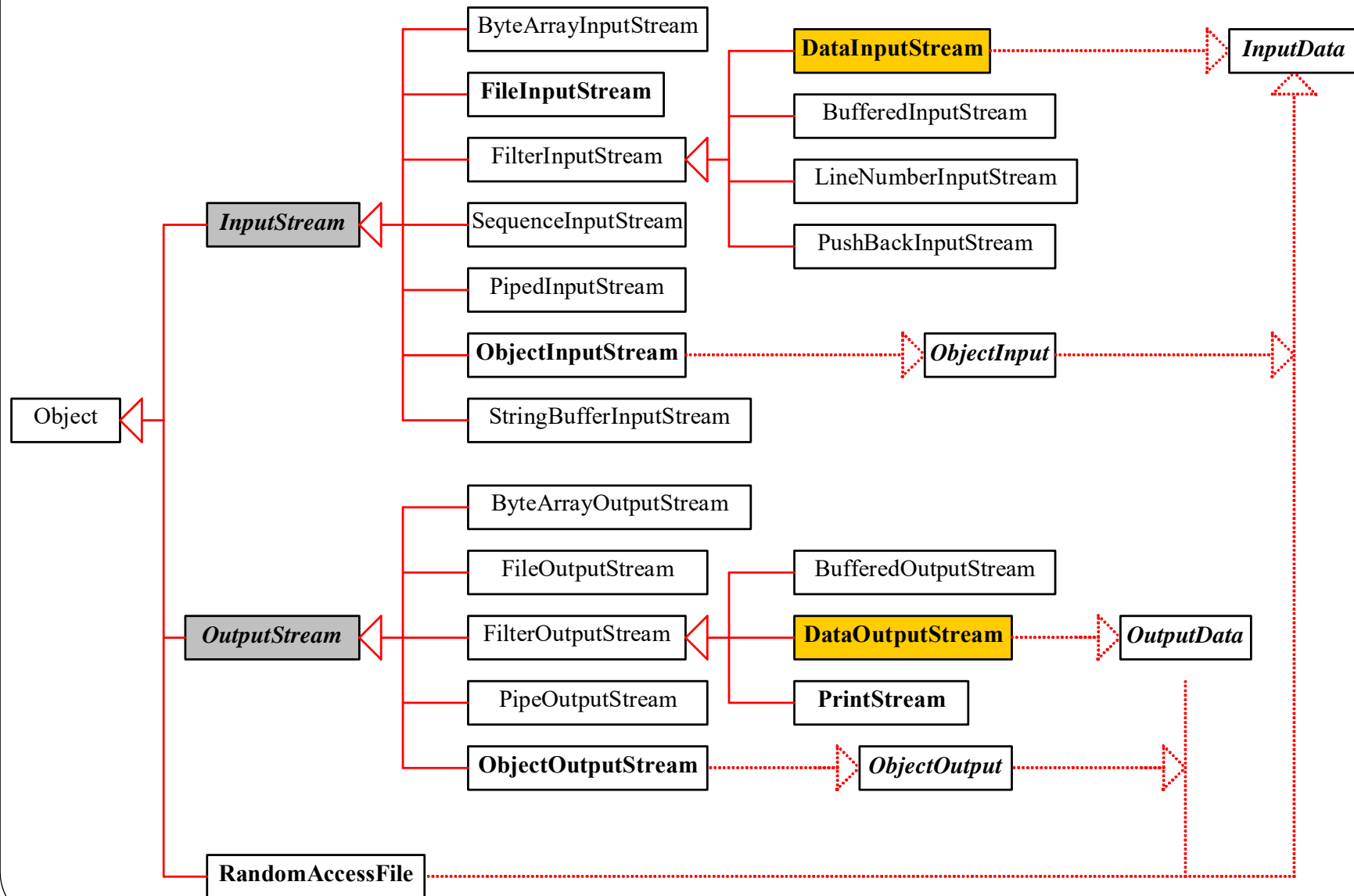
```
new BufferedReader (insReader )
```



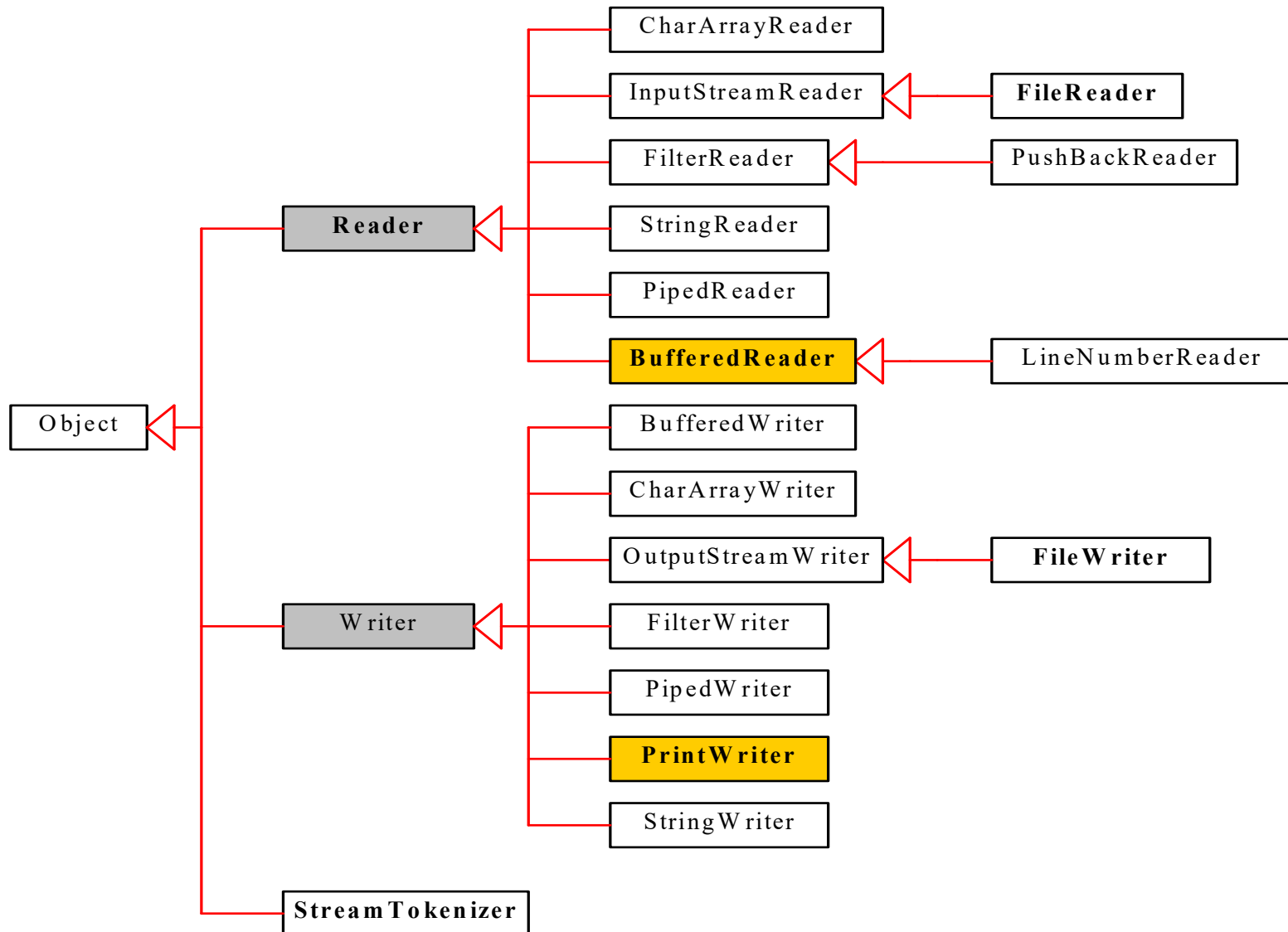
## JAVA IO 流的这种机制如何实现：装饰者设计模式（decorator pattern）

**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）**

# Byte Stream Classes



# Character Stream Classes



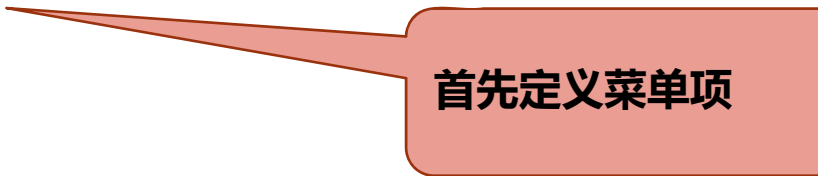
## 迭代模式的由来

- 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;
```

用数组存储菜单项

```
    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; }  
}
```

❑ 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

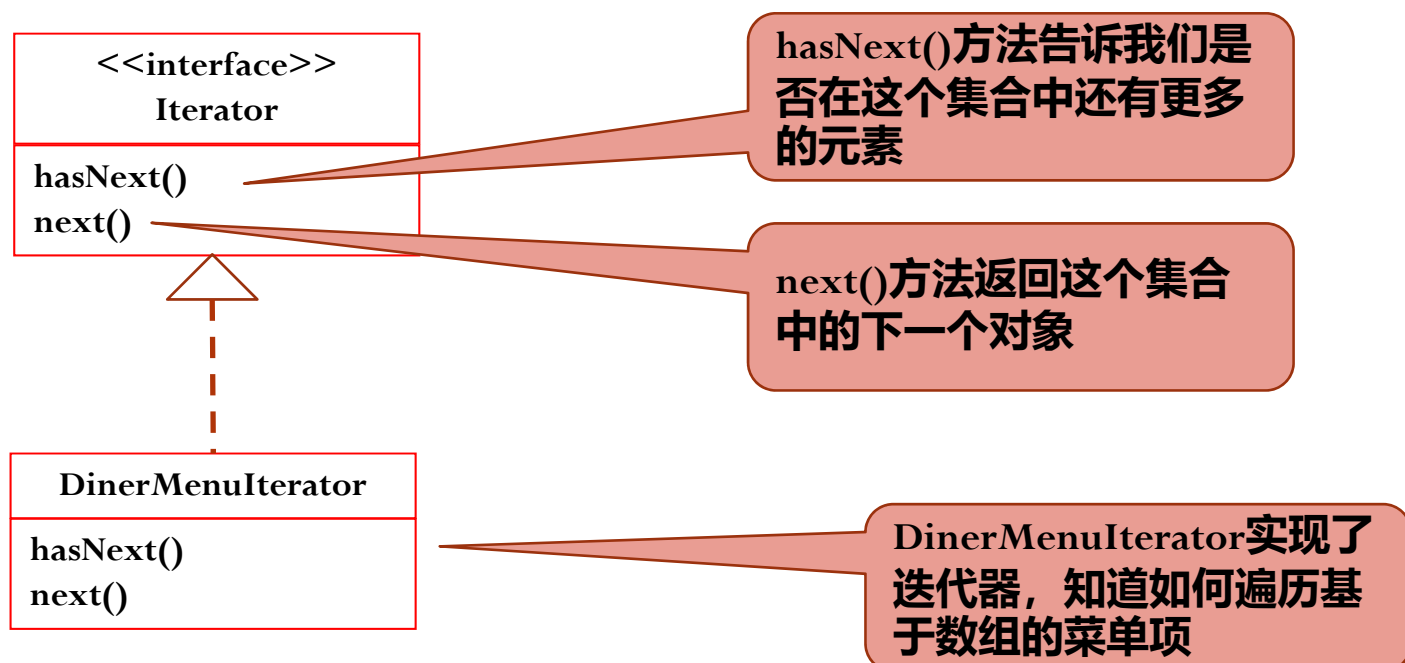


## □ 这种实现的问题

- 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

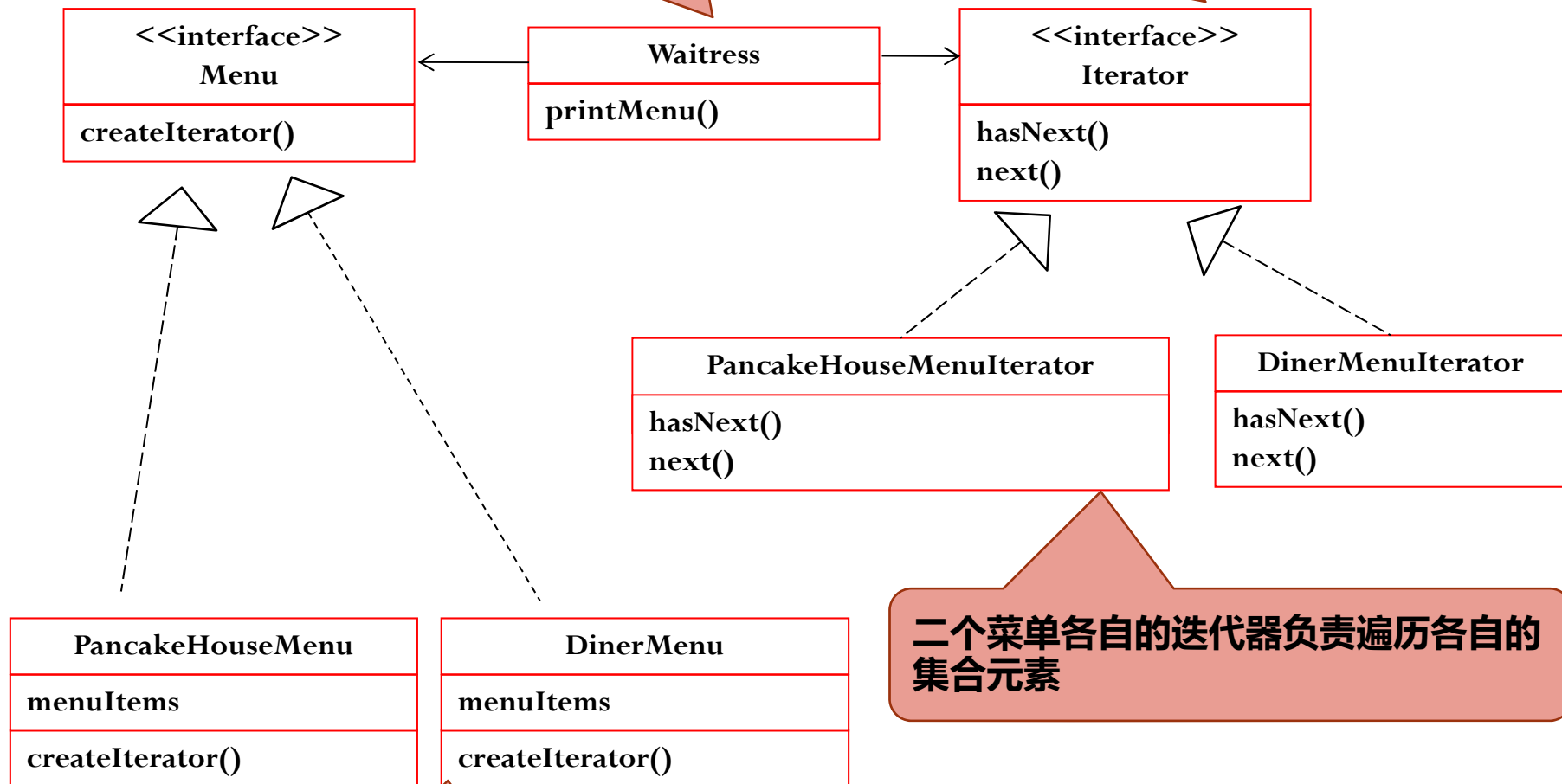
```
public class Waitress {  
    Menu pancakeHouseMenu  
    Menu dinerMenu;  
    public Waitress(Menu pm, Menu dm){  
        pancakeHouseMenu = pm; dinerMenu = dm;  
    }  
    public void printMenu(){  
        Iterator pancakeIt = pancakeHouseMenu.createIterator();  
        Iterator dinerIt = 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！
  - 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

# 迭代器模式的意图和适用性

## □ 意图

- 迭代器模式的目的是设计一个迭代器，提供一种可顺序访问聚合对象中各个元素的方法，但不暴露该对象内部表示

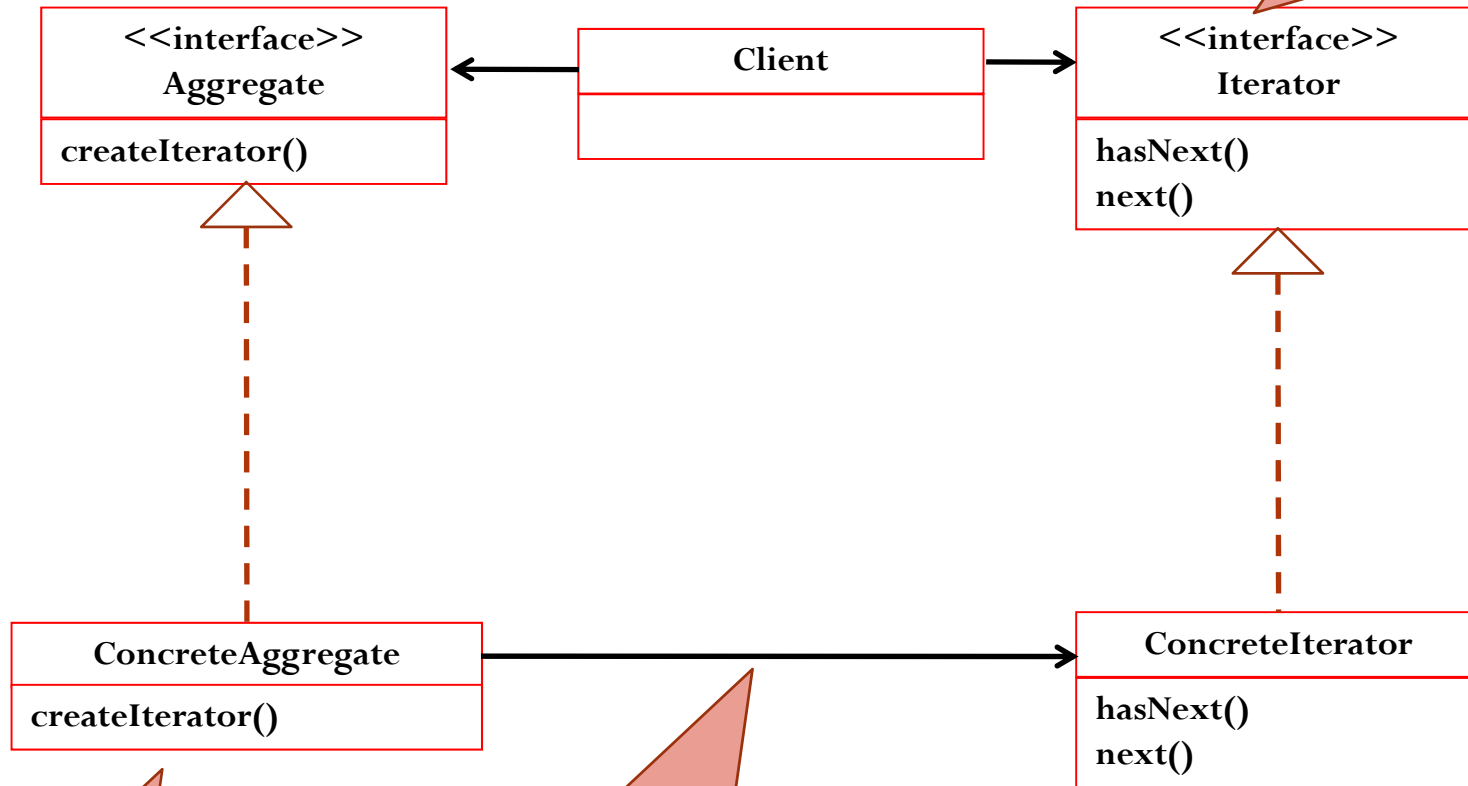
## □ 适用场合

- 访问一个聚合对象的内容而无需暴露其内部表示
- 支持对聚合对象的多种遍历
- 为遍历不同的聚合结构提供一个统一接口 (支持多态迭代)

# 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



具体聚合都持有一个对象集合

具体聚合都负责实例化一个具体迭代器

具体迭代器负责遍历具体聚合对象的元素



# 迭代器模式的参与者

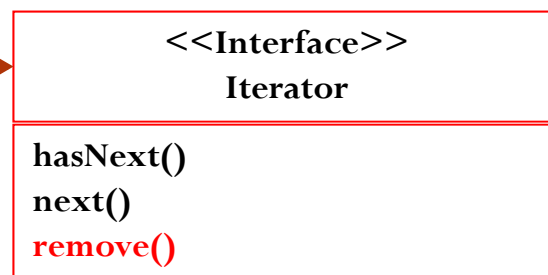
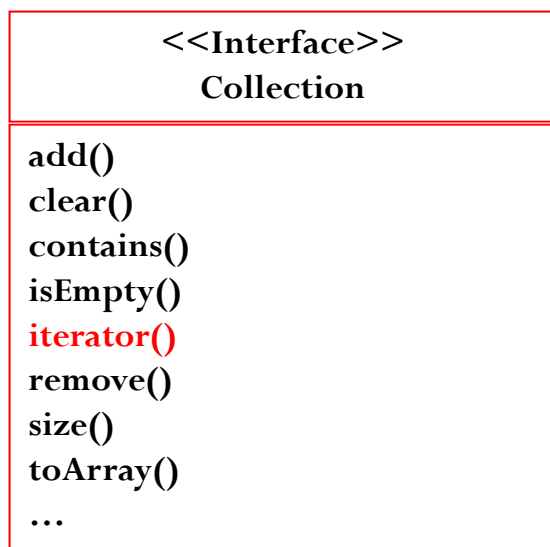
- ❑ Iterator
  - ❑ 定义访问和遍历元素的接口
- ❑ Concrete Iterator
  - ❑ 实现迭代器接口
- ❑ Aggregate
  - ❑ 定义创建迭代器对象的接口
- ❑ Concrete Aggregate
  - ❑ 实现创建迭代器的接口，返回具体迭代器的一个实例

## 迭代器模式的效果分析

- ❑ 简化了聚集的行为，迭代器具备遍历接口的功能，聚集不必具备遍历接口
- ❑ 每一个聚集对象都可以有一个或者更多的迭代器对象，每一个迭代器的迭代状态可以彼此独立
- ❑ 遍历算法被封装到迭代器对象中，迭代算法可以独立于聚集对象变化。客户端不必知道聚集对象的类型，通过迭代器就可以读取和遍历聚集对象。聚集内部数据发生变化不影响客户端程序

# 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