装饰者模式 工厂模式

赖辉源

2017. 5. 18

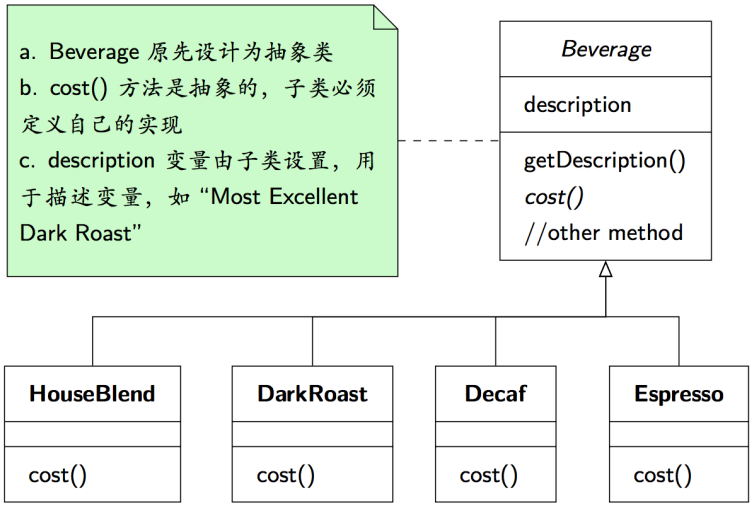
● 装饰者模式的威力

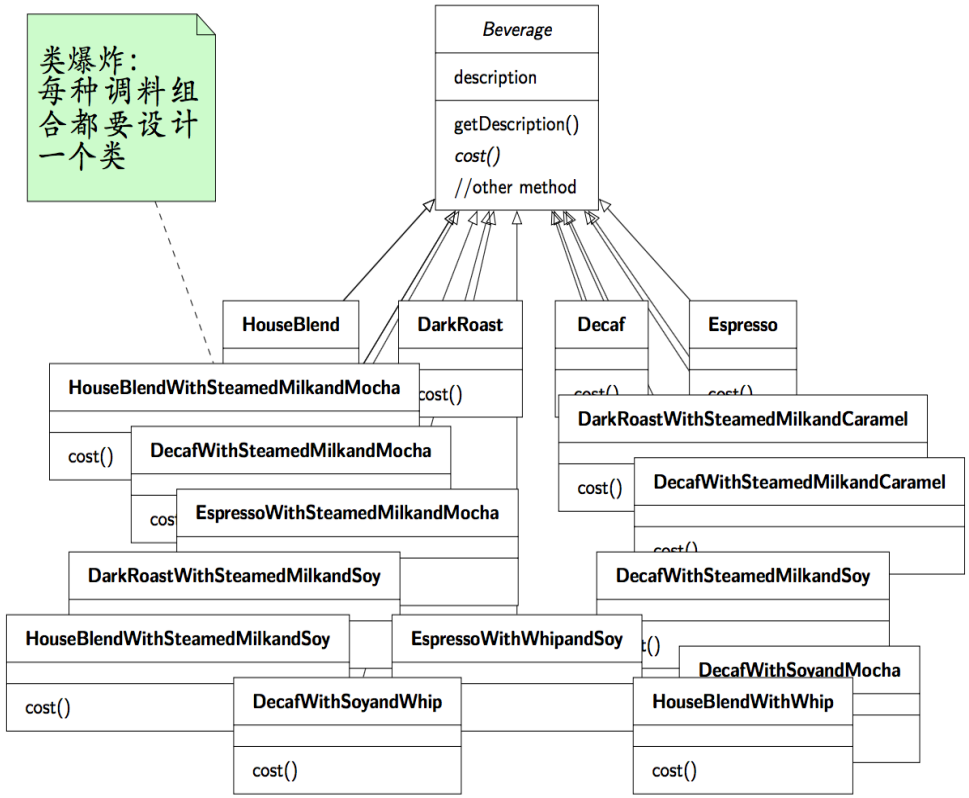
① 装饰者模式可以给爱用继承的人一个全新的设计眼界。

② 我们可以在不修改任何底层代码（被装饰者）的前提下，扩展被装饰者的功能，给对象赋予新的职责。

③ 运行时扩展远比编译时继承威力大！

● 咖啡店准备更新订单系统





- 需求：购买咖啡时，可以要求在其中加入各种调料，如Steamed Milk等。店家会根据加入的调料收取不同的费用，那么订单必须考虑到这些调料部分。

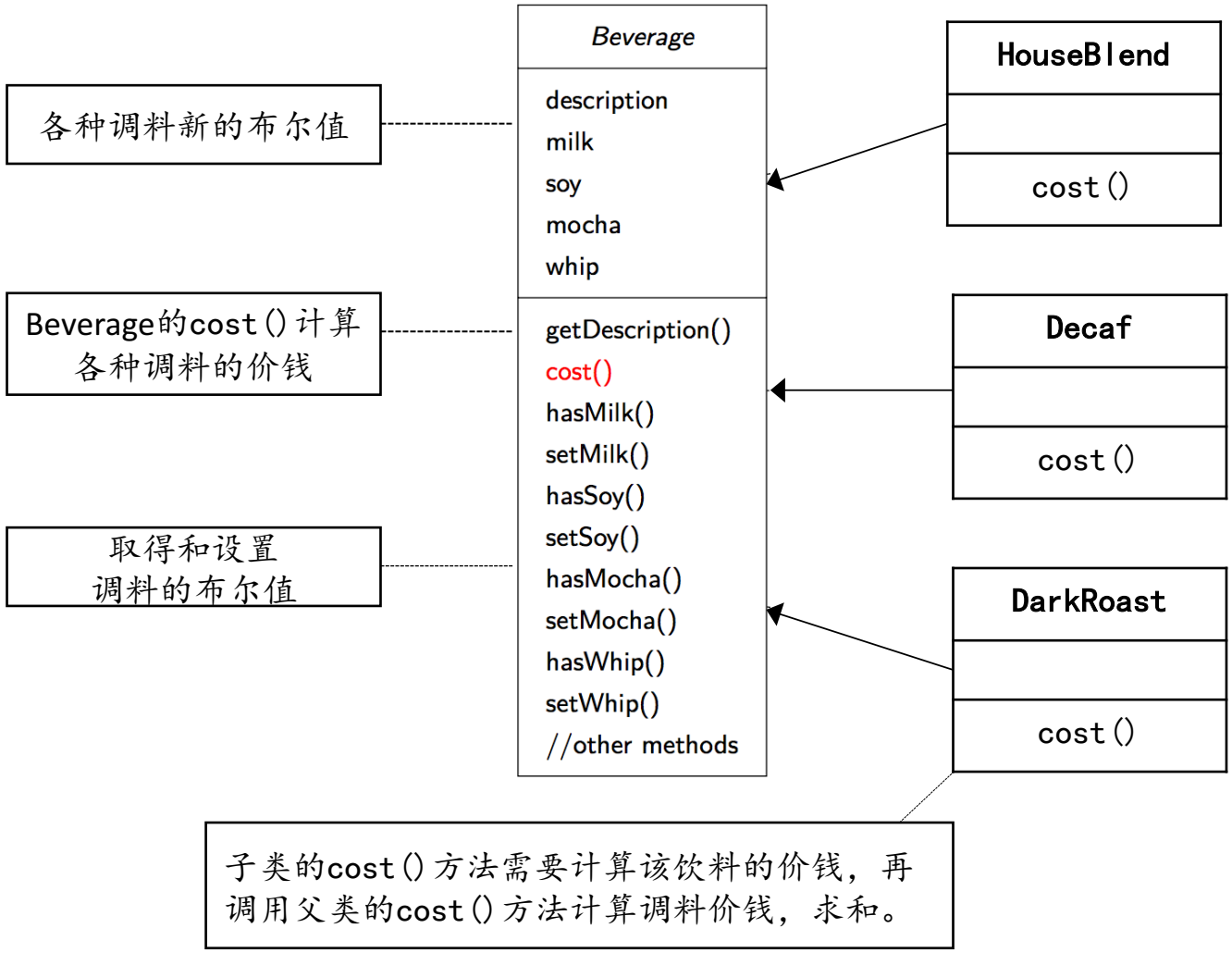
● 咖啡店准备更新订单系统

- 很明显，这是一个维护的噩梦！如果牛奶的价钱上涨、或者新增一种调料都会造成维护上的困难！
- 这是因为违反以下设计原则：

①封装变化(将系统中经常变化的部分和稳定的部分隔离，增加复用性，并降低耦合度)。

②多用聚合、少用继承。

● 新的设计：放弃调料类


Beverage
description
milk
soy
mocha
whip
getDescription()
cost()
hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()
//other methods
各种调料新的布尔值
Beverage的cost()计算
各种调料的价钱
取得和设置
调料的布尔值
HouseBlend
cost()
Decaf
cost()
DarkRoast
cost()
子类的cost()方法需要计算该饮料的价钱，再
调用父类的cost()方法计算调料价钱，求和。

● 新的设计：放弃调料类

```
Public double cost(){
    float  condimentCost = 0.0;
    if  (hasMilk())
        condimentCost += milkCost;
    if  (hasSoy())
        condimentCost += soyCost;
    if  (hasMocha())
        condimentCost += mochaCost;
    if  (hasWhip())
        condimentCost += whipcost;
    return condimentCost;
}
```

Beverage的cost()方法的伪代码

```
Public class DarkRoast extends Beverage{
    public DarkRoast(){
        description = “Most Excellent Dark Roast”;
    }
    public double cost(){
        return 1.99 + super.cost();
    }
}
```

DarkRoast的伪代码

新的设计：放弃调料类

- 改变的影响：

①类的数量大大的减少，有了很大的改善；

②调料价钱会引发对代码的改动，若出现新调料需加上新的方法，并改变父类的cost()方法。

③将来可能会有不适合某些调料的新饮料出现，但仍然继承了一些不适合的方法。

④顾客有可能想要双倍的摩卡咖啡。

开放-关闭设计原则

- 设计原则：开放-关闭，即类应该对扩展开放，对修改关闭
- 我们的目标是：允许类容易扩展，在不修改现有类代码的情况下，就可以让类具有新的行为。

  这种设计的好处是具有弹性应对改变，能够灵活应对改变的需求。
- 需要注意的是：我们没必要把系统的每部分都设计为可扩展的（时间、效率、代码复杂…）。

● 认识装饰者模式(Decorator)

- 我们已经了解到采用“继承”的设计可能会导致类爆炸、设计死板、父类新功能无法适应所有子类等问题。
- 在这里，采用一种不一样的做法，我们要以饮料为主体，然后在运行时以调料来“装饰”饮料，比如顾客想要摩卡好奶泡深焙咖啡，那么：

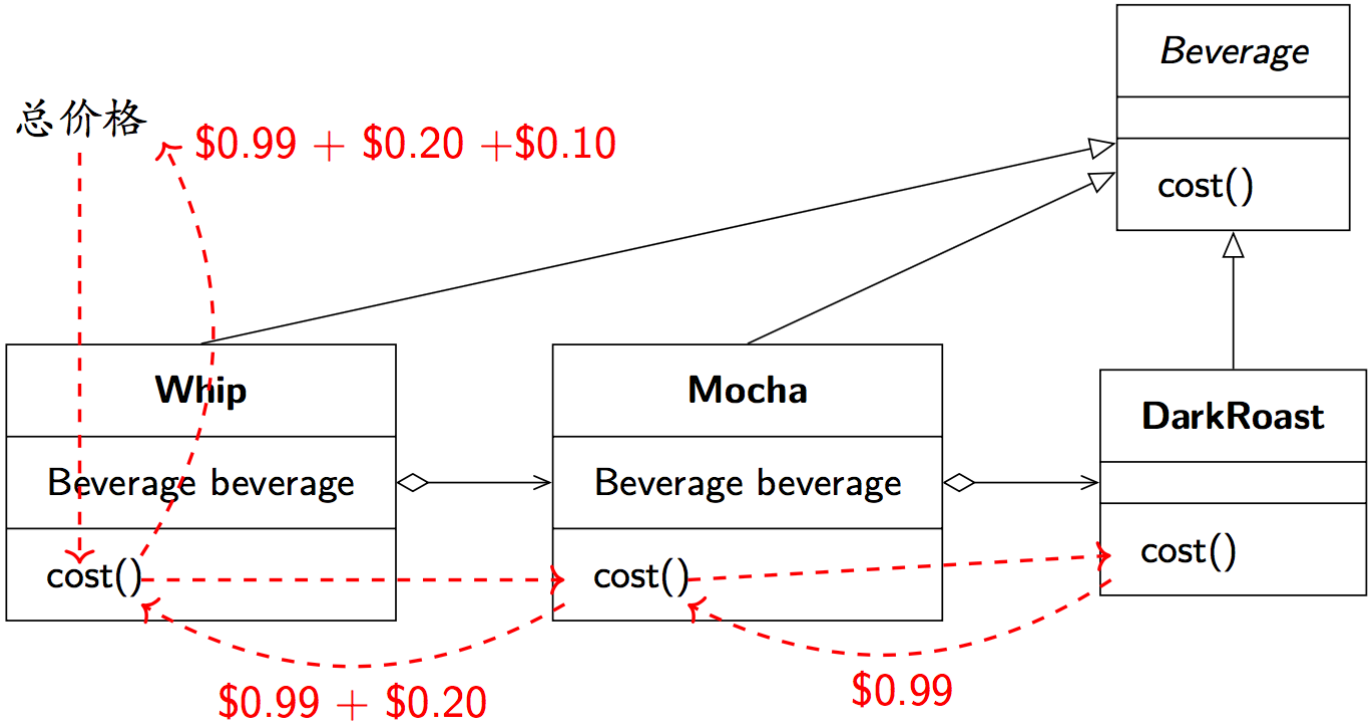
①拿一个DarkRost对象；

②用Mocha对象装饰它；

③用Whip对象装饰它；

④调用cost()方法，并且依赖委托将调料的价钱加上去。

- 以装饰者模式构造饮料订单

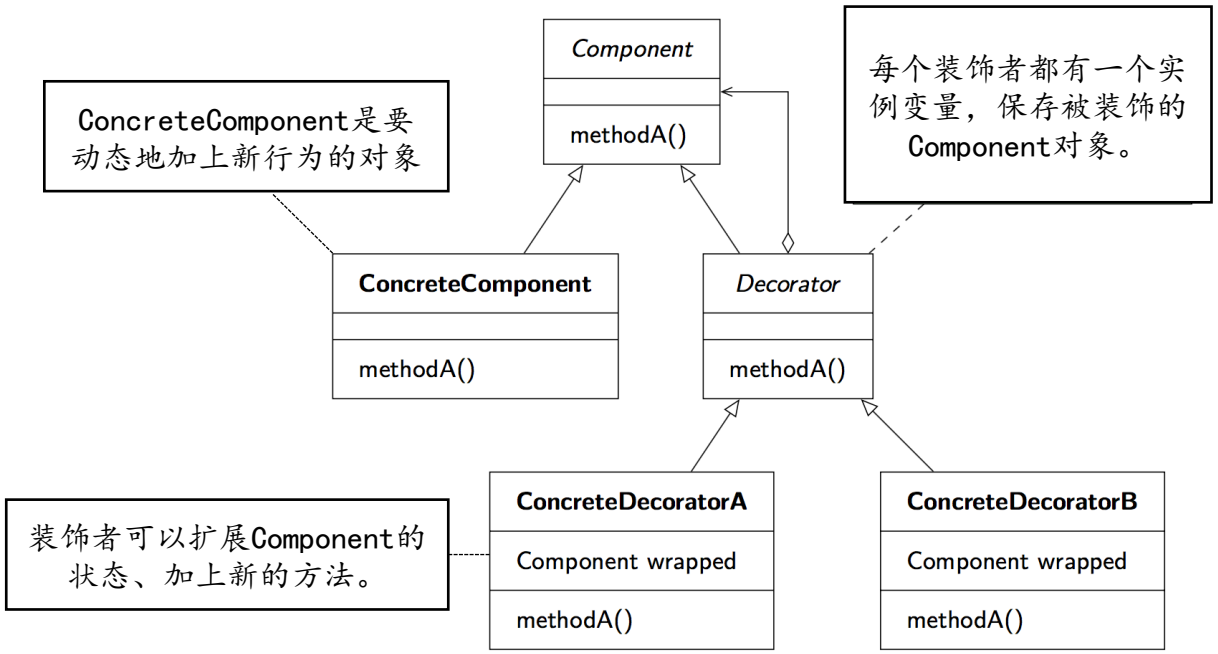


以装饰者模式构造饮料订单

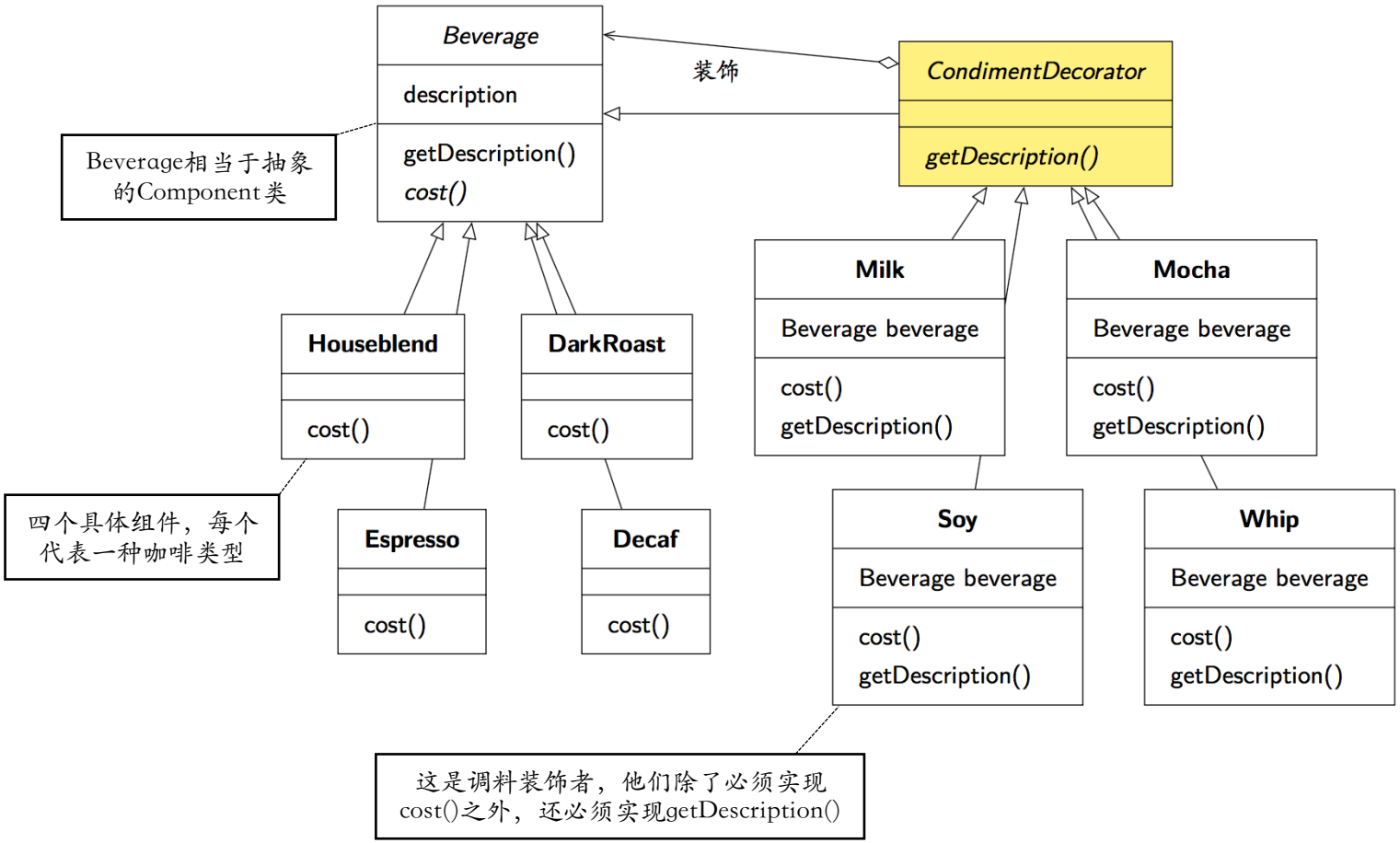
- 装饰者和被装饰者对象拥有一个相同的父类。
- 我们可以用多个装饰者对象装饰一个被装饰者对象。
- 由于装饰者和被装饰者的父类型相同，因此可以在任何需要原始对象的场合，用装饰过的对象取代它。
- 装饰者可以在被装饰者的行为之前/之后，加上自己的特定行为。
- 对象可以在任何时候被装饰，因此可以在运行时动态地、不受限制地进行装饰。

● 定义装饰者模式

- 装饰者模式：装饰者模式动态地将责任附加到对象上。在扩展功能方面，装饰者提供了比继承更有弹性的替代方案。



● 装饰者模式设计饮料系统



● 装饰者模式中的继承和聚合

- Q:装饰者模式为什么仍然用到了继承?
- A:这里的继承是使装饰者和被装饰者具有一样的类型，而不是利用继承获得父类的行为。
- Q:那行为从何而来?
- A:当将装饰者和被装饰者聚合在一起时，就是在增加新的行为。新的行为非来自继承，而是来自聚合。
- Q:这里的聚合相比继承的好处是什么?

  A:若依赖继承，那么类的行为就在编译时刻决定了。若使用聚合，即可在运行时动态增加新行为，而不必修改已有代码。

对扩展开放，对修改封闭

● 使用装饰者模式的饮料新系统

```
Public abstract class Beverage{
    String description = “Unkown Beverage”;
    public String getDescription(){
        return description;
    }
    public abstract double cost();
}
```

被装饰者抽象类

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

装饰者抽象类

```
Public class Espresso extends Beverage{
    public Espresso(){
        description = “Espresso;
    }
    public double cost(){
        return  1.99;
    }
}
```

具体的被装饰者-饮料子类

● 使用装饰者模式的饮料新系统

```
Public class Mocha extends
    CondimentDecorator{
    Beverage Beverage ;
    Beverage  Mocha(Beverage beverage){
        this.beverage = beverage;
    }
    public String getDescription(){
        return beverage.getDecription()+“,Mocha”;
    }
    public  double cost(){
        return .20+beverage.cost();
    }
}
```

用一个实体变量记录饮料，即被装饰者。

把饮料当成构造器的参数，再由构造器将此饮料记录在实例变量中。

利用委托的做法得到一个完整的叙述。

利用委托给被装饰对象以计算价钱，再加上Mocha价钱。

具体装饰者-调料子类

● 使用装饰者模式的饮料新系统

```
Public class StabuzzCoffee{
    public static void main(String arg[]){
        Beverage beverage=new Espresso();
        System.out.println(beverage.
        getDescription()+"$" +beverage.cost());
        Beverage beverage2=new DarkRoast();
        beverage2=new Mocha(beverage2);
        beverage2=new Whip(beverage2);
        System.out.println(beverage2.getDecription()
        +"$"+beverage.cost());
    }
}
```

测试类

```
% Java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee,Mocha,Whip $1.49
%
```

输出结果

装饰者模式小结

- 继承可以扩展对象行为，但不是达到弹性设计的最佳方式。
- 设计应允许扩展现有行为，而无须修改现有代码。
- 聚合和委托可用于在运行时动态增加新行为。
- 装饰者模式允许弹性地扩展现有行为。
- 装饰者模式意味着很多装饰者类。
- 可以用任意多个装饰者包装一个构件。
- 过度使用装饰者模式会使程序变得复杂。

设计原则小结

- 封装变化
- 多用聚合、少用继承
- 针对接口编程，不针对实现编程
- 尽最大可能将要交互的对象设计为松耦合的
- 对扩展开放，对修改封闭

工厂模式

- 简单工厂
- 工厂方法
- 抽象工厂

你是如何实制造对象？

- Duck duck =new MallardDuck();
- 当使用new操作符实例化对象时，是在实例化具体类，所以是针对实现编程，而不是针对接口编程，会使代码缺乏弹性。
- 当有一群相关的具体类，比如各种不同鸭子，但必须等到运行时才知道该实例化哪一个。

```
Duck duck;
If (picnic){
      duck = new MallradDuck();
} else if (hunting){
      duck = new Decoyduck();
} else if (inBathTub){
      duck = new Ruberduck();
}
```

这里有多种鸭子类，但必须等到运行时才知道实例化哪一个。一旦代码有变化或者扩展，那么就必须打开这段代码进行检查和修改！

设计原则：封装变化性

- 找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起。

  即：系统中的某部分改变不要影响其他部分。

识别变化：Pizza店应用

- 假设有一个pizza店，那么生产pizza的代码可以这么写：

```
Pizza orderPizza() {
        Pizza pizza = new pizza();
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        retunr pizaa;
}
```

- 为了让系统有弹性，将Pizza设计为抽象类，但这样就无法实例化各种具体的Pizza类。因此需要增加代码来决定创建何种Pizza。

```
Pizza orderPizza(String type) {
        Pizza pizza;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek") {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni") {
            pizza = new PepperoniPizza();
         }
        pizza.prepare();
        //类似左图
}
```

识别变化：Pizza店应用

- 有一天决定删除一些旧的口味，增加别的新口味了怎么办？
- 将有可能需要更改的，且创建具体对象的代码放到另一个对象中，由这个对象专职创建pizza，我们将这个对象称之为工厂。

```
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza ;
    }
```

工厂定义一个createPizza()方法，所有客户使用这个方法来实例化新对象。

识别变化：Pizza店应用

- 那么新的重新实现PizzaStore为：

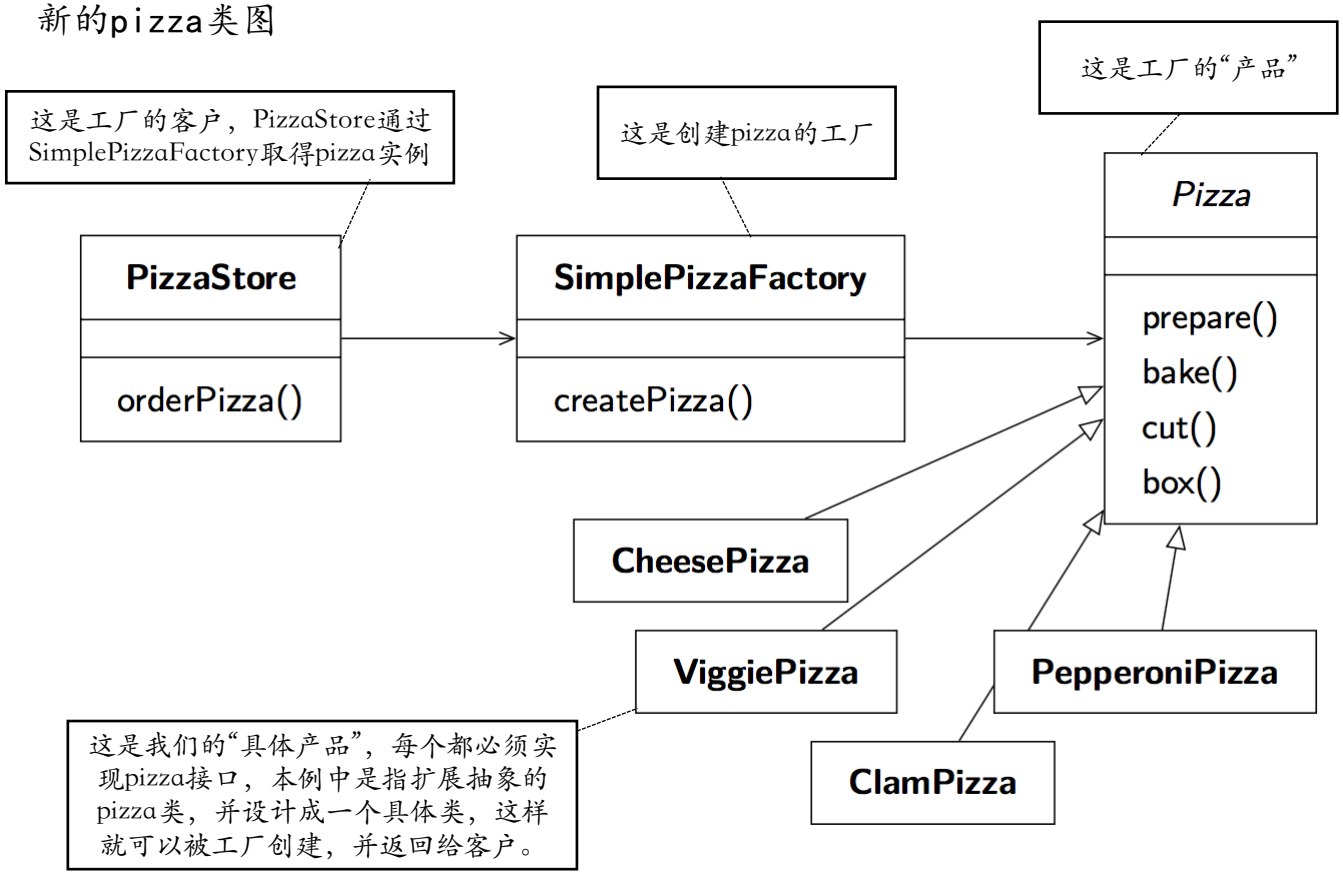
```
public class PizzaStore {
    SimplePizzaFactory factory;
    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        retunr pizaa;
    }
}
```

我们为PizzaStore 加上一个对SimplePizzaFactory的引用，PizzaStore的构造器，需要一个工厂作为参数。

orderPizza()方法通过简单传入订单类型来使用工厂创建Pizza。

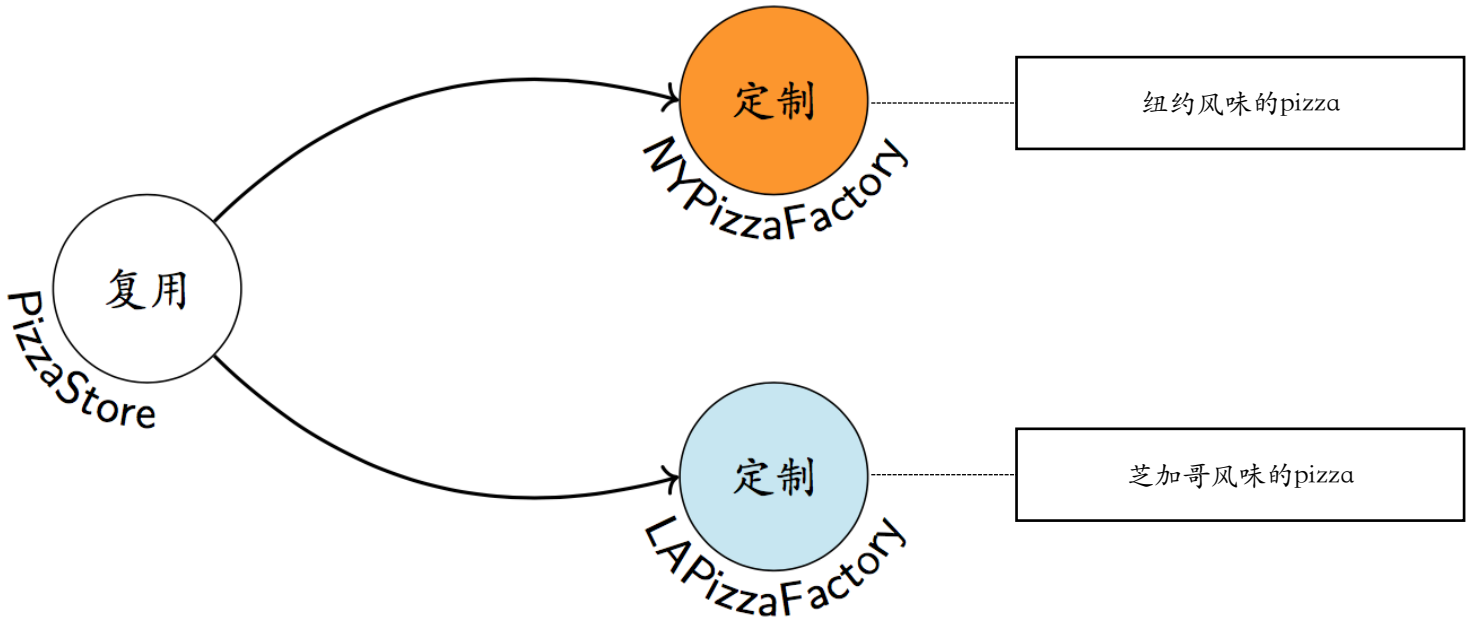
● 定义简单工厂

- 新的pizza类图



● 工厂方法(加盟Pizza店)

- 很多Pizza店希望能成为我们的加盟店，他们应使用我们现有的OO代码。但是这些加盟店希望提供不同风味的Pizza 。



● 加盟店不同的工厂类

```
//纽约风味的工厂
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new  PizzaStore(nyFactory);
nyStore.order("Veggie");

//洛杉矶风味的工厂
LAPizzaFactory laFactory = new LAPizzaFactory();
PizzaStore laStore = PizzaStore(laFactory);
laStore.order("Veggie");
```

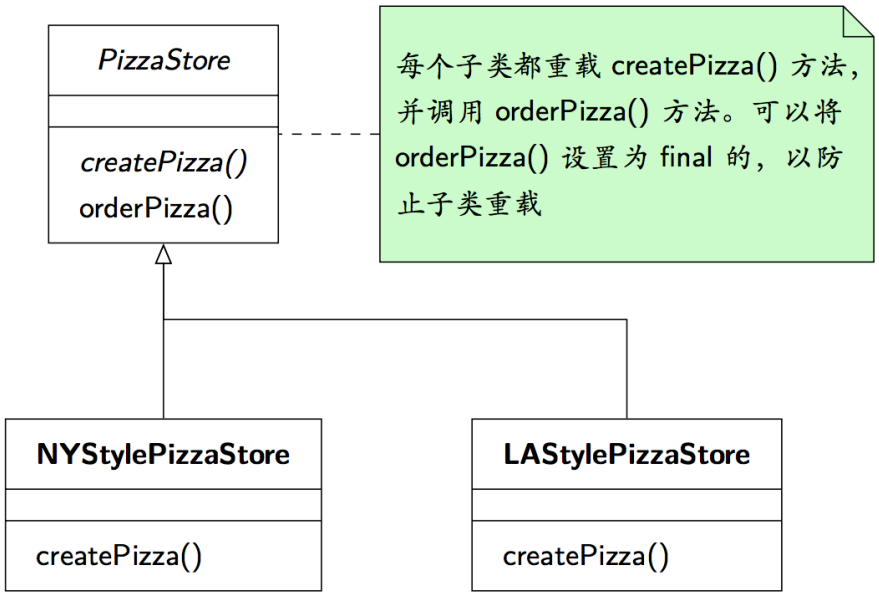
这是制造的工厂，是制造纽约风味的pizza，然后建立一个pizza店，将纽约工厂的引用作为参数，之后当我们制造pizza时得到的全是纽约风味的pizza。

质量控制

- SimpleFactory工厂很容易被各加盟店采用，但他们只是利用SimpleFactory创建对象，自顾自地定义了自己的 Pizza 加工流程：例如使用第三方包装，忘记将Pizza切分等等。
- 因此，我们需要的是一个框架，将Pizza店和Pizza创建绑定到一起，并允许一定的灵活性。
- 方法:取消独立的工厂类，在PizzaStore中设立一个工厂方法 。

让子类决定如何创建对象

- 我们把createPizza()方法放回PizzaStore中，不过把它设为“抽象方法”，然后为每个区域创建一个PizzaStore的子类。



● 工厂方法的新设计

```
public abstract class PizzaStore {
    public final Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    protected abstract Pizza createPizza(String type);
    // other methods here
}
```

现在的PizzaStore是抽象的，createPizza方法从工厂对象中移回PizzaStore，此方法就相当于一个“工厂”，在这里，“工厂方法”现在是抽象的，PizzaStore的子类在creatPizza()方法中处理对象的实例化。

● 工厂方法的新设计

```
public class NYPizzaStore extends PizzaStore {
    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

所有的门店可以从PizzaStore免费取得所有的功能，然后提供createPizza()方法实现自己风味的pizza即可。

NYPizzaStore扩展PizzaStore，所以拥有了orderPizza()及其他方法，但oderPizza()并不知道将要创建哪一种pizza，它只知道被准备、被烘焙等，我们必须去实现createPizza()，因为它原来是抽象的。

纽约地区的PizzaStore

● 工厂方法的新设计

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList toppings = new ArrayList();
    void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for (int i = 0; i < toppings.size(); i++) {
            System.out.println(" " + toppings.get(i));
    }
```

```
void bake() {
    System.out.println("Bake for 25 minutes at 350");
    }
void cut() {
    System.out.println("Cutting the pizza into
    diagonal slices");
    }
void box() {
     System.out.println("Place pizza in official
      PizzaStore box");
      }
public String getName() {return name; }

}
```

Pizza

● 工厂方法的新设计

```
public class LAStyleCheesePizza extends Pizza {
    public LAStyleCheesePizza() {
        name = "LAStyle Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";
        toppings.add("Shredded Mozzarella Cheese");
    }
    void cut() {
        System.out.println("Cutting the pizza into
            square slices");
    }
}
```

定义具体的Pizza子类

```
public class PizzaTestDrive {
        public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore();
        Pizza pizza = nyStore.orderPizza("cheese");
        System.out.println("Ethan ordered a " +
            pizza.getName() + "\n");
        }
}
```

测试

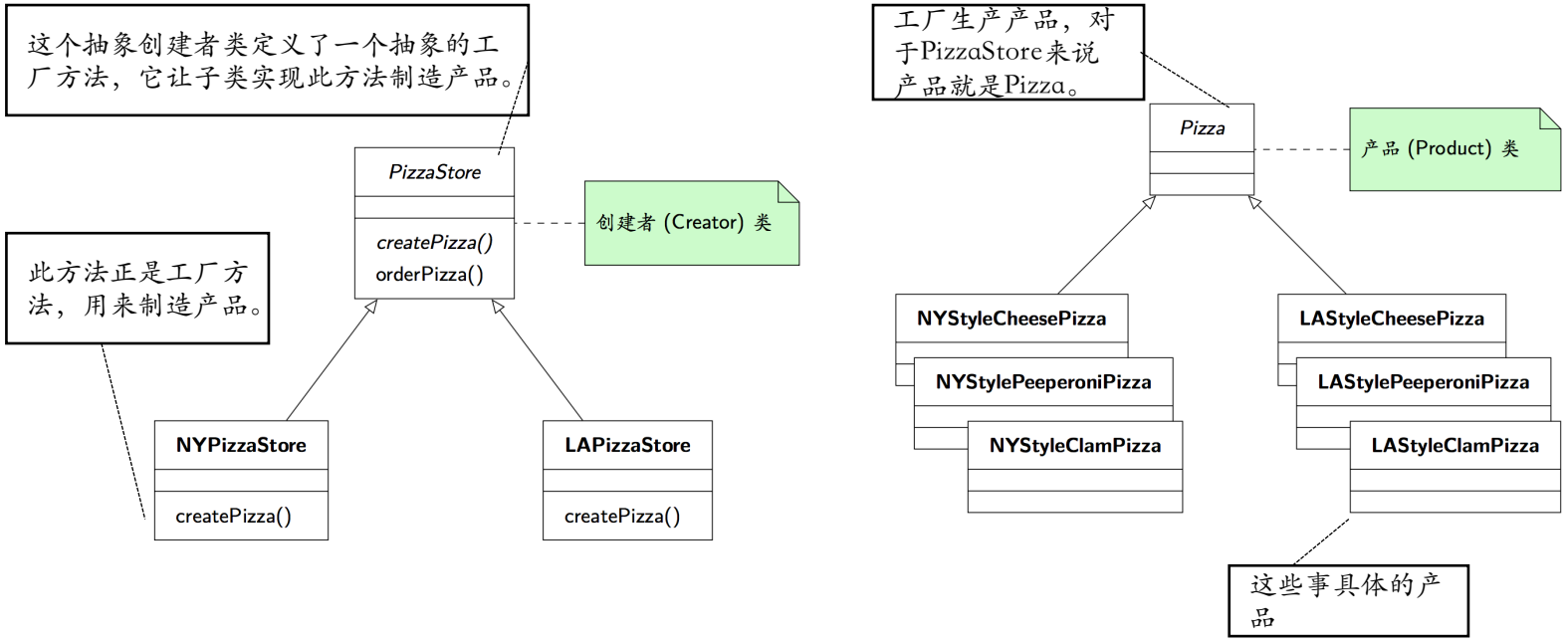
● 工厂方法的新设计

```
%java PizzaTestDrive
 Preparing NY Style Sauce and Cheese Pizza
 Tossing dough...
 Adding sauce...
 Adding toppings:
 Shredded Mozzarella Cheese
 Bake for 25 minutes at 350
 Cutting the pizza into diagonal slices
 Place pizza in official PizzaStore box
 Ethan ordered a NY Style Sauce and Cheese Pizza
 %
```

输出结果

● 认识工厂方法模式(Factory Method Pattern)

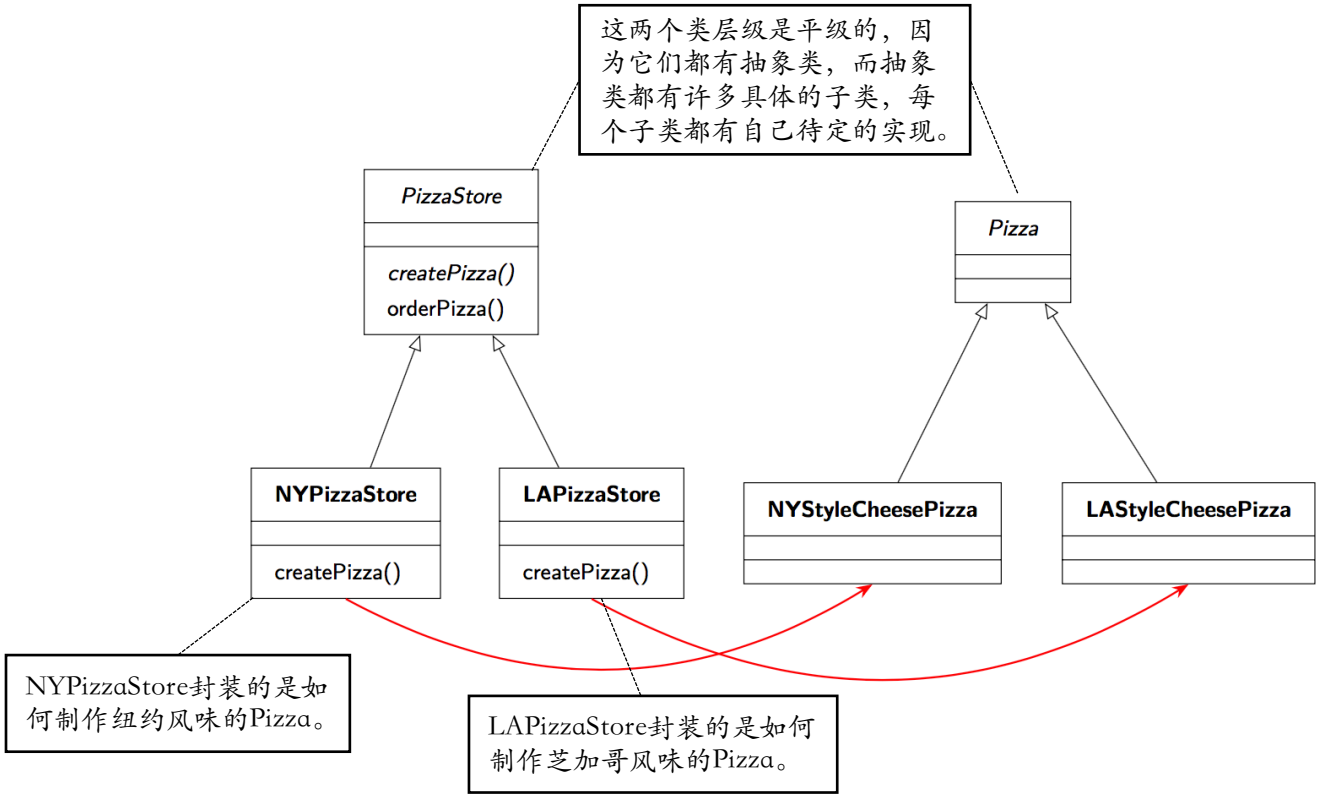
- 所有的工厂模式都用封装对象来创建，工厂方法模式通过让子类决定该创建的对象是什么，来达到将对象创建的过程封装的目的。



认识工厂方法模式(Factory Method Pattern)

- 我们已经看到将一个orderPizza()方法和一个工厂方法联合起来，就可以成为一个框架。

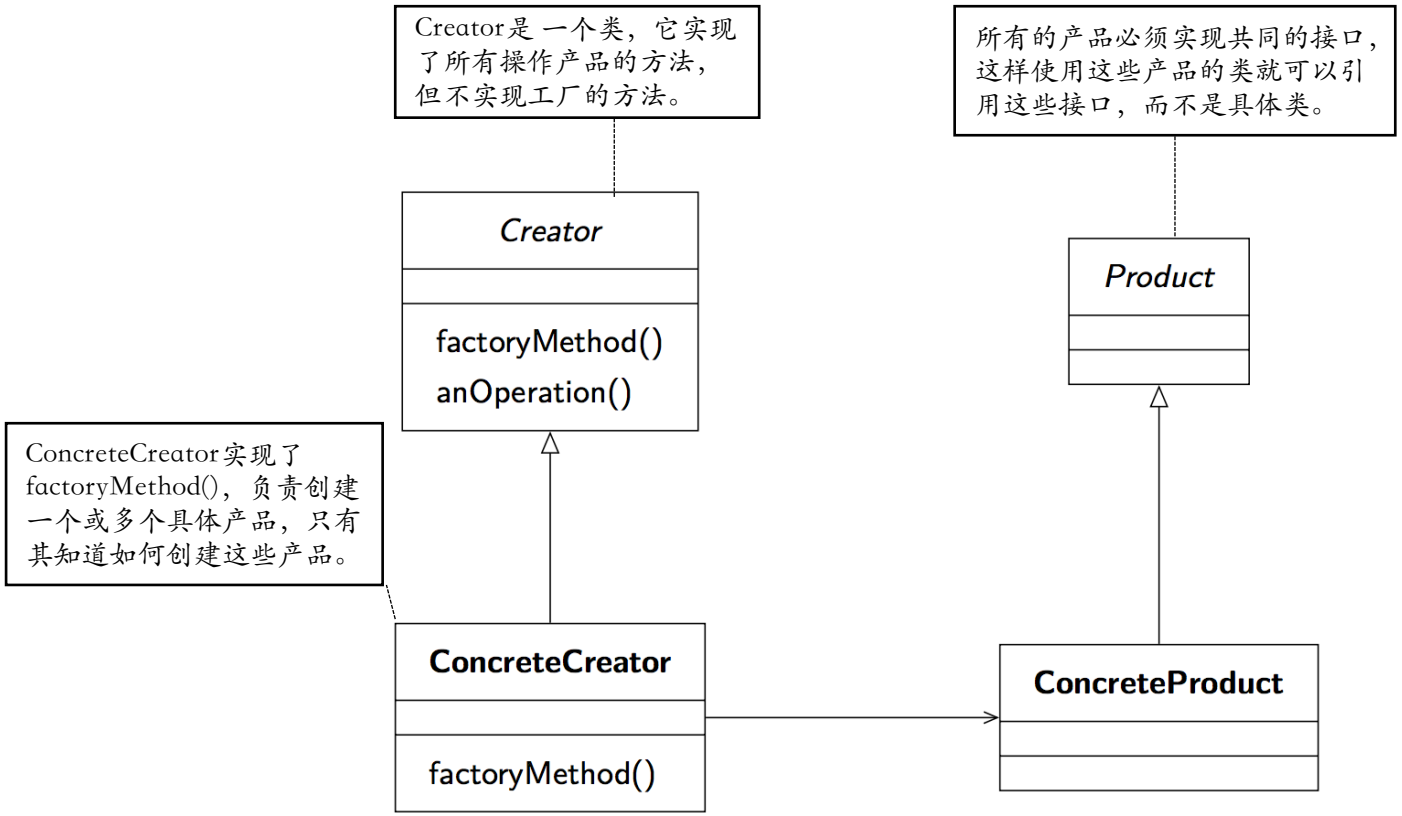
  除此之外生产方法将生产知识封装进各个创建者也可以被视为一个框架。



定义工厂方法模式(Factory Method Pattern)

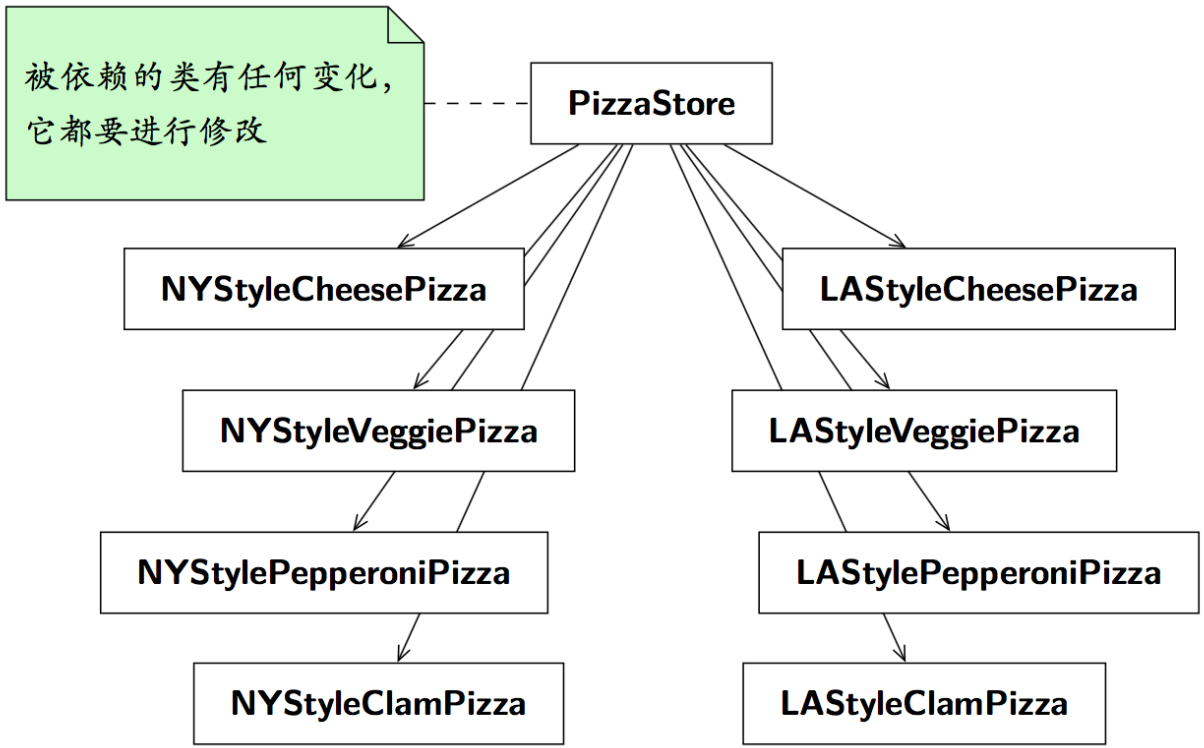
- **工厂方法模式：**定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类把如何实例化对象交由子类决定。
- 工厂方法模式能够封装具体类型的实例化。
- 所谓 “由子类决定”，并不是允许子类在运行时做决定，而是指在编写创建者类时，不需要知道实际创建的产品是什么。
- 选择使用哪一个子类，自然就决定了实际创建的产品是什么。

● 定义工厂方法模式(Factory Method Pattern)



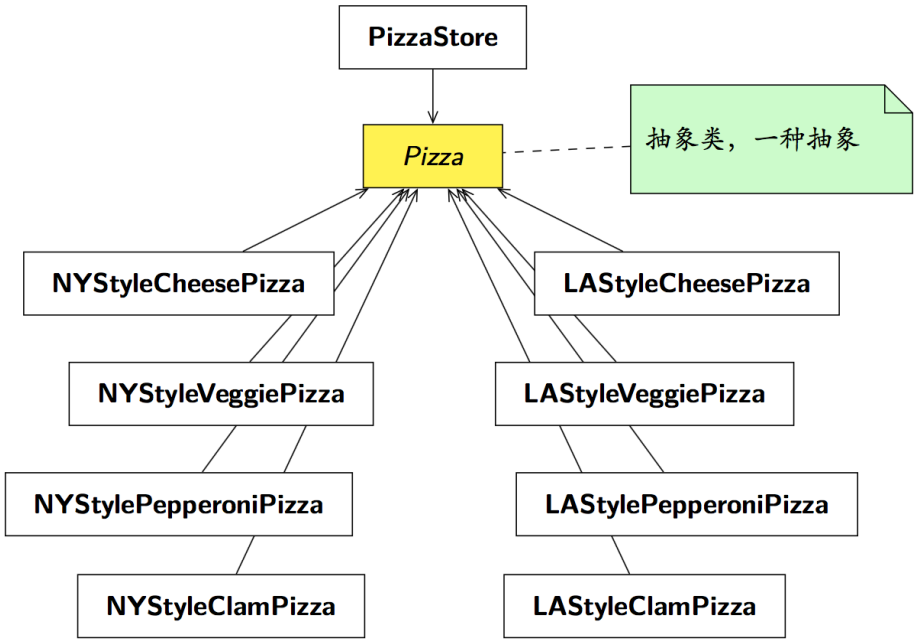
一个很依赖的Pizza店

- 当我们直接实例化一个对象时，就是在依赖它的具体类，下面是一个不使用工厂模式的Pizza版本。



依赖反转原则

- 设计原则:依赖反转(Dependency Inversion Principle)，即依赖抽象，不要依赖具体类。
- PizzaStore是“高层组件”，Pizza实现是“低层组件”，PizzaStore依赖这些具体的Pizza类，根据依赖反转原则应该重写代码便于高层和低层模块都依赖抽象类，而不是依赖具体类。



理解“反转”：以PizzaStore为例

- 架构师：如果要实现一个PizzaStore系统，首先会想到什么？
- 程序员：PizzaStore无非准备、烘烤、包装Pizza，所以要能生产各种Pizza:CheesePizza，VeggiePizza，ClamPizza，等等。
- 架构师：没错。但就要了解各种具体的Pizza，进而对他们产生依赖。所以不妨反过来想，从Pizza着手，看能不能抽象。
- 程序员：那样的话，CheesePizza，VeggiePizza，ClamPizza都是Pizza， 因此他们应该共享一个Pizza接口。
- 架构师：对。Pizza就是抽象。现在可以重新设计了。
- 程序员：由于有了Pizza抽象，所以可以据此设计PizzaStore，使之不依赖于具体的Pizza类了。
- 架构师：对。还需要一个工厂方法，来实例化各种具体的类。这样不同的具体类就只依赖一个抽象的Pizza，如此以来，该设计就把原来的依赖关系反转了。

帮你遵循依赖反转原则

- 不要让变量持有具体类的引用

  不要用new，用工厂方法

- 不要让类派生自具体类

  让类派生自抽象（接口或抽象类）

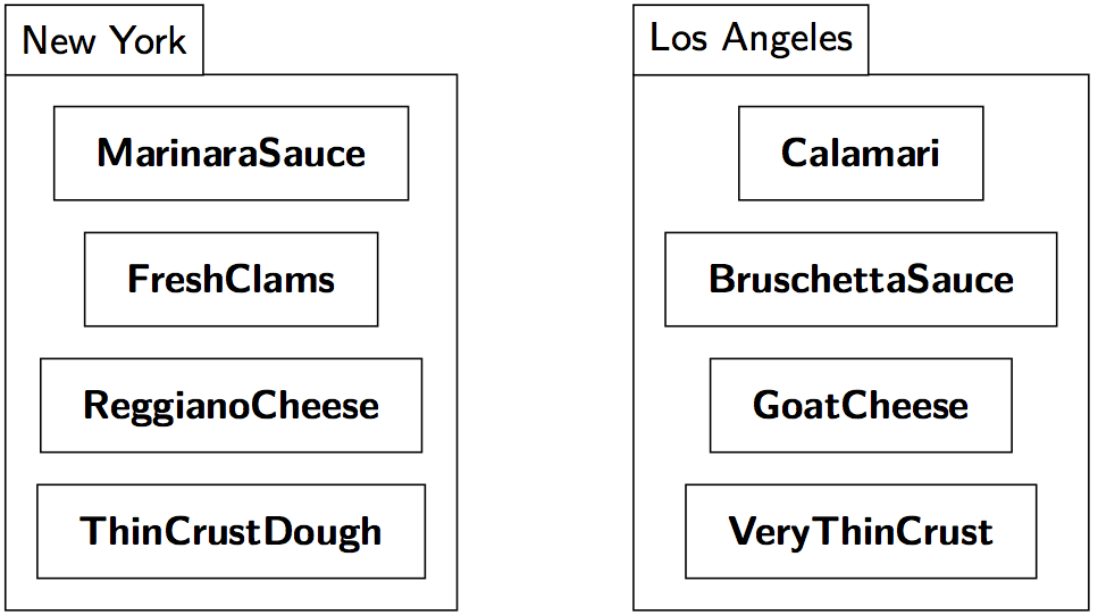
- 不要覆盖父类中已实现的方法

  如果覆盖了父类已实现的方法，那么父类就不是一个真正适合被继承的抽象。父类中已实现的方法，应该由所有的子类共享。

完全做到很困难的，尽量有意识遵循

● 抽象工厂(原料家族)

- 所有Pizza都由面团（dough)、酱料（sauce)、芝士（cheese）及若干佐料加工而成，不同店对这些原料的实现可能是不同的。



● 建造原料工厂

- 现在我们建造一个原料工厂来生产原料，这个工厂将负责创建原料家族中的每一种原料。开始为工厂定义一个接口，这个接口负责创建所有的原料。

```
public interface PizzaIngredientFactory {
    public Dough createDough();
    public Sauce  createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();
}
```

在接口中，每一个原料都有对应的方法创建该原料。

这里有许多类，每一个原料都是一个类。

定义原料工厂接口

● 建造原料工厂

接下来需要做得事情是：

- 为每个地区设计一个工厂类，实现PizzaIngredientFactory接口。
- 实现该PizzaFactory需要的各种原料类，如ReggianoCheese，RedPeppers等。
- 将新的原料工厂相关类和原有PizzaStore代码组合到一起。

● 建造原料工厂

```
public class NYPizzaIngredientFactory
    implements PizzaIngredientFactory {
    public Dough createDough() {
        return new ThinCrustDough();
    }
    public Veggies[] createVeggies() {
        Veggies veggies[] = { new Garlic(), new
        Onion(),new Mushroom(), new RedPepper() };
        return veggies;
    }
     public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
}
```

具体原料工厂必须实现这个接口，纽约原料工厂也不例外，对于原料家族内的每一种原料，我们都提供了纽约的版本。

纽约原料工厂部分代码

● 重做Pizza

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;
    //将准备原料的 prepare 方法声明为抽象的
    abstract void prepare();
```

```
void bake() {
    System.out.println("Bake for 25 minutes at 350");}
void cut() {
    System.out.println("Cutting the pizza into
    diagonal slices");}
void box() {
    System.out.println("Place pizza in official
     PizzaStore box");}
 // other methods such as setName, getName, toString
}
```

新的Pizza类

每个Pizza都持有一组准备时会用到的原料，并且把prepare()方法声明成抽象，在这个方法中，我们需要收集Pizza所需的原料，而这些原料来自原料工厂。

重做Pizza

- 现在已经有了一个抽象Pizza，可以开始创建纽约和芝加哥风味的Pizza了，并且加盟店必须直接从工厂获取原料。

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingf;
    public CheesePizza(PizzaIngredientFactory ingf) {
        this.ingf = ingf; }
    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingf.createDough();
        sauce = ingf.createSauce();
        cheese = ingf.createCheese();}
}
```

要制作Pizza，需要工厂提供原料，所以每个Pizza都需要从构造器参数中得到一个工厂，并把这个工厂存在一个实例变量中。

prepare()方法一步步地创建芝士Pizza，每当需要原料时，就跟工厂要。

不同风味的Pizza做法一样，原料不同

● 重做Pizza

```
public class NYPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingf =
          new NYPizzaIngredientFactory();
       if (item.equals("cheese")) {
          pizza = new  CheesePizza(ingredientFactory);
          pizza.setName("New York Style Cheese Pizza");
       }else if (item.equals("veggie")) {
          // 制造各种本地风味的 Pizza
    }
}
```

纽约店全部用到纽约Pizza原料工厂，由该原料工厂负责生产全部纽约风味Pizza所需的原料。

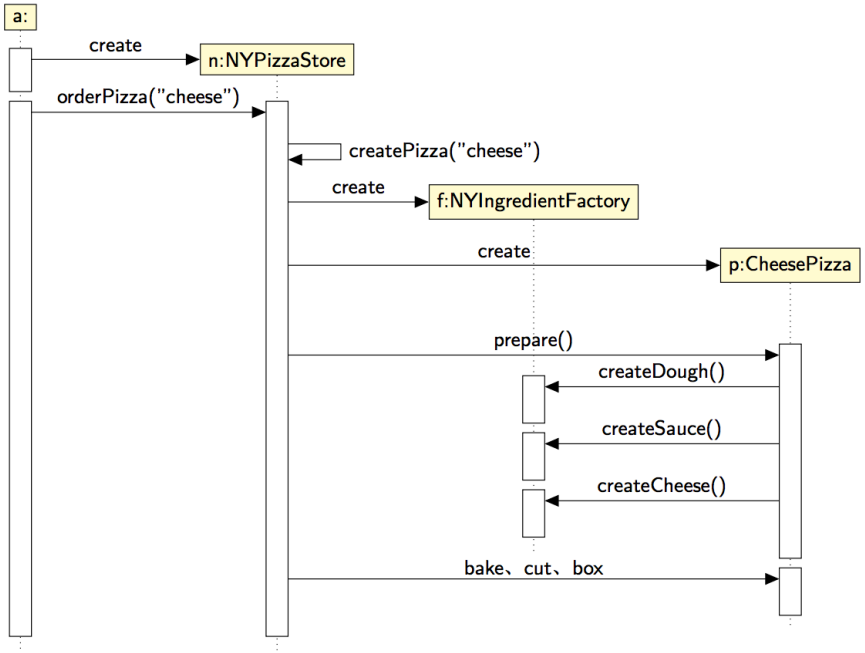
把工厂传递给每一个Pizza，以便Pizza从工厂获取所需的原料。

使用了本地原料工厂的PizzaStore

我们做了什么？

- 引入新类型的工厂，即抽象工厂，来创建原料家族。
- 通过抽象工厂提供的接口，创建产品的代码将和实际工厂解耦，从而可以在不同上下文中实现不同工厂，制造不同产品。
- 由于代码从实际的产品中解耦，所以可以替换不同的工厂来取得不同的行为。
- 客户代码始终保持不变。

● 怎么订购一个Pizza



①首先我们需要一个PizzaStore；

②之后可以接受订单；

③orderPizza()方法首先调用createPizza()方法；

④开始涉及原料工厂；

⑤准备Pizza，调用prepare()方法，准备原料；

⑥得到Pizza，orderPizza接着烘烤、切片、装盒。

定义抽象工厂模式

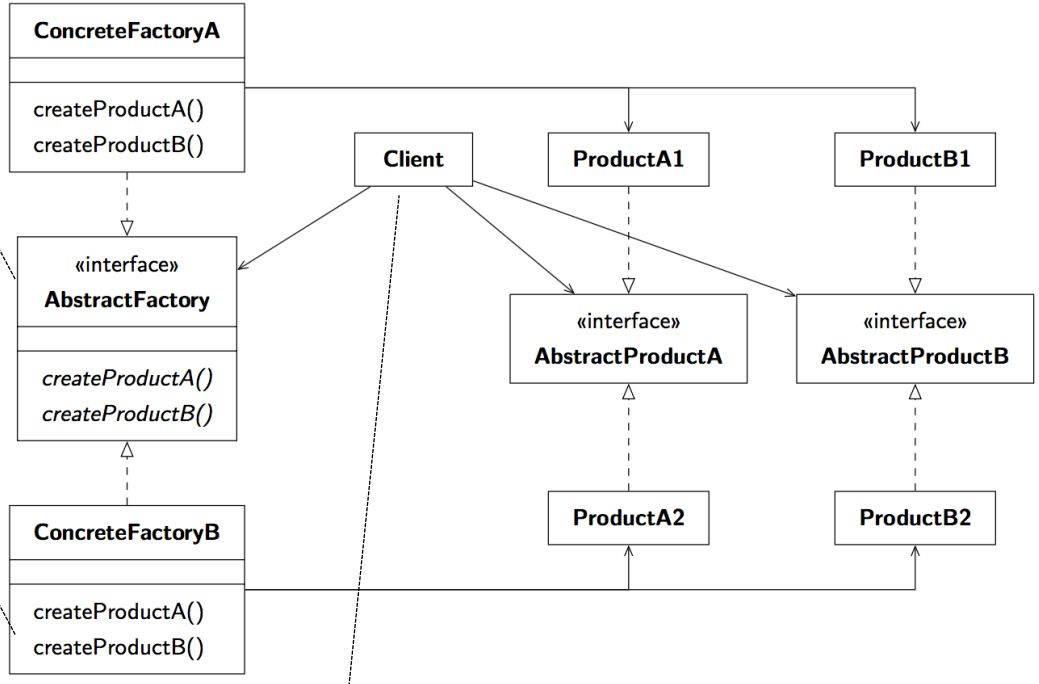
- **抽象工厂模式**：抽象工厂模式提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类。
- 抽象工厂允许客户使用抽象的接口来创建一组相关的产品，而不需要知道实际产出的具体产品是什么。如此一来，客户就从具体的产品中被解耦。

● 定义抽象工厂模式

抽象工厂定义了一个接口，所有具体工厂都必须实现该接口，这个接口包含一组方法用来生产产品。

具体工厂实现不同的产品家族，要创建一个产品，客户只要使用其中的一个工厂而完全不需要实例化任何产品对象。

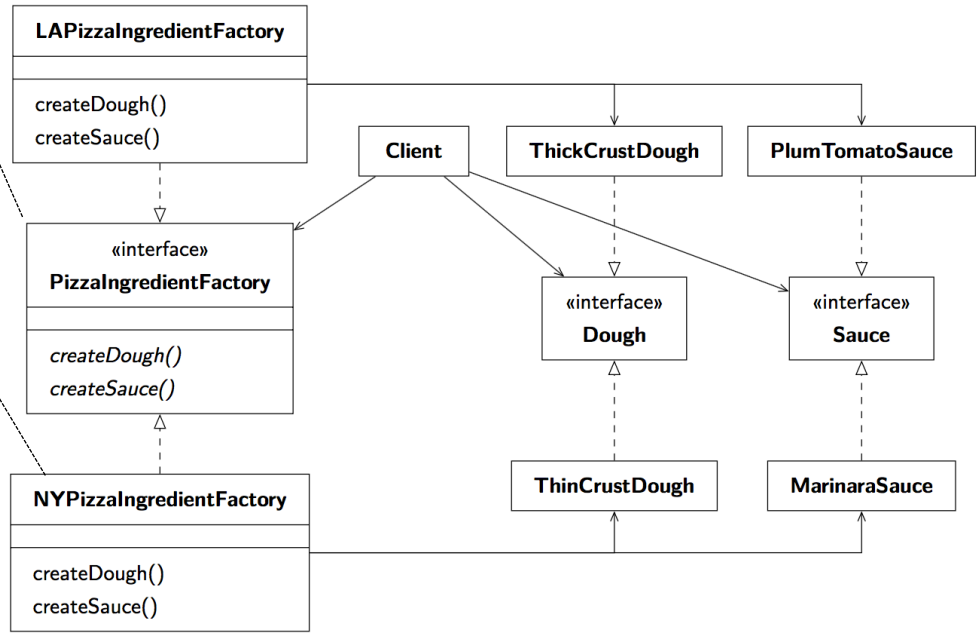
客户的代码只需涉及抽象工厂，运行时自动使用实际的工厂。



● 从PizzaStore的观点看抽象工厂

这是抽象的Pizza原料工厂借口，它定义了如何产生一个相关产品的家族，这个家族包含了所有制作Pizza的原料。

这些具体工厂负责生产Pizza原料，每个工厂都知道如何生产符合自己区域的原料。



● 工厂模式小结

- 所有的工厂都是用来封装对象的创建，所有工厂模式都通过减少应用程序和具体类之间的依赖，促进松耦合；
- 简单工厂不是真正的模式，但常常简单有效；
- 工厂方法使用继承；
- 抽象工厂使用对象聚合；
- 依赖倒置原则指导我们编程时尽量依赖抽象；

设计原则小结

- 封装变化
- 多用聚合、少用继承
- 针对接口编程，不针对实现编程
- 尽最大可能将要交互的对象设计为松耦合的
- 对扩展开放，对修改封闭
- 依赖抽象，不要依赖具体类

谢谢 ！