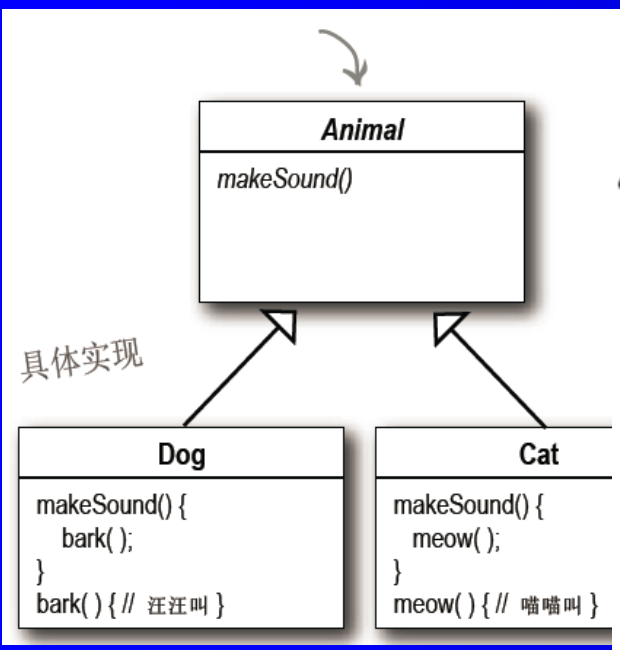
三原则（面向对象设计）

- 针对接口编程
- 封装变化
- 尽量使用对象组合而不是继承

继承的优缺点

- 1，可以实现类级别代码的重用。
- 2，不断地重写继承的方法，control-c，control-v
- 3，很难得到类的所有行为。
- 4，对类的修改会影响其他子类，同时破坏了类的封装性。

编程的三个境界



“针对实现编程”

```
Dog d = new Dog( );
d.bark( );
```

声明变量“d”为Dog类型（是Animal的具体实现），会造成我们必须针对具体实现编码。

但是，“针对接口/超类型编程”做法会如下：

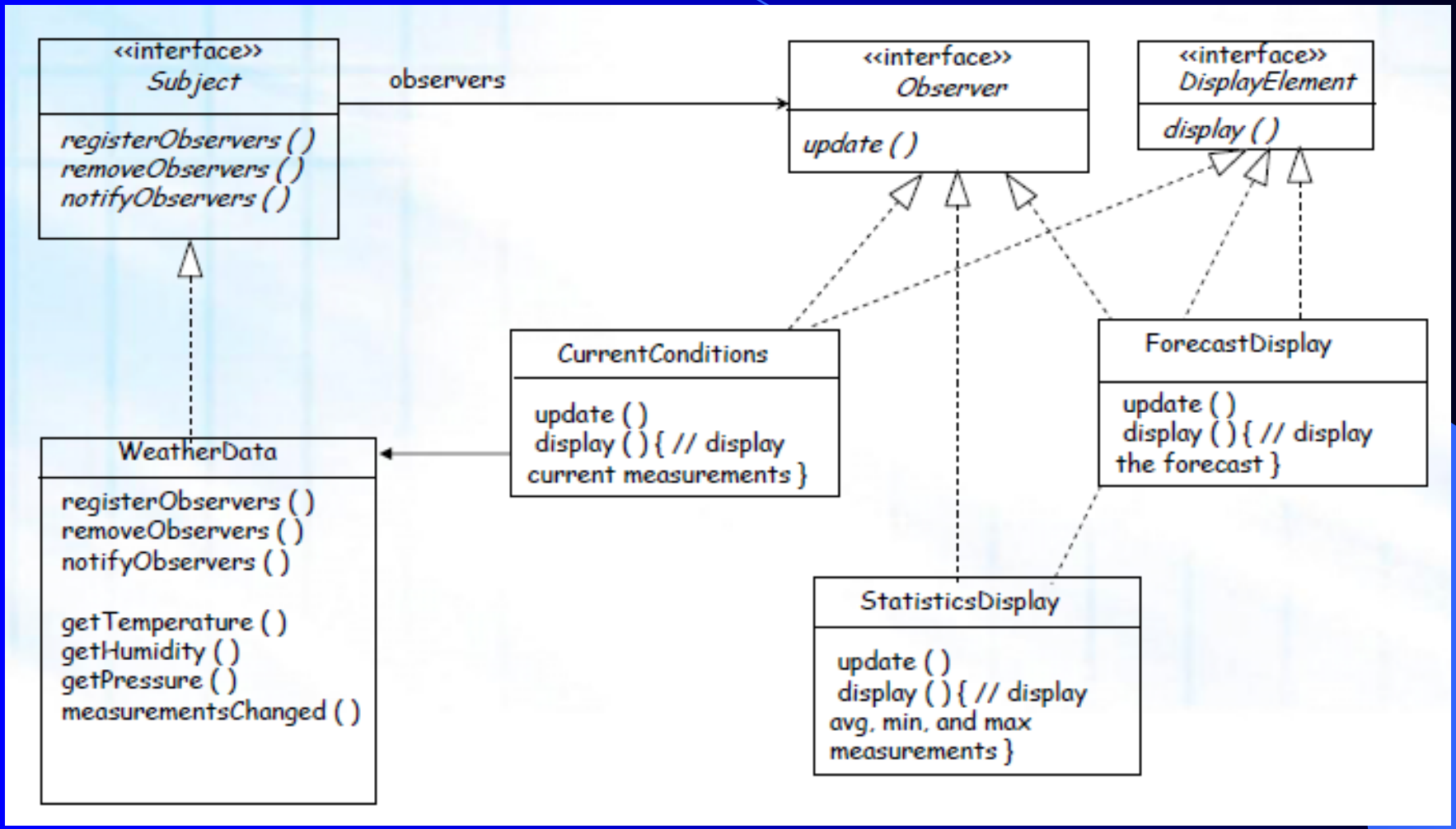
```
Animal animal = new Dog( );
animal.makeSound( );
```

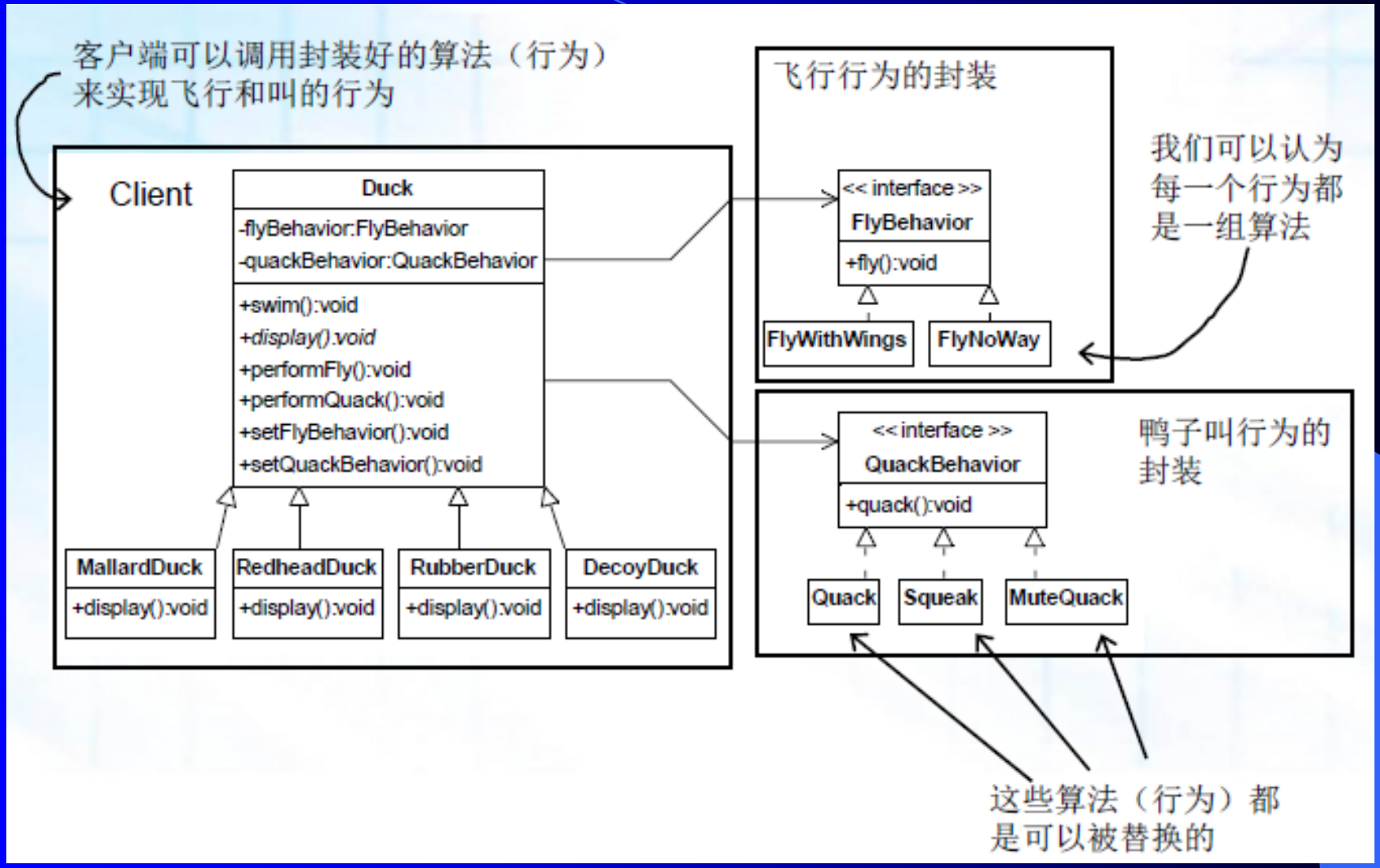
我们知道该对象是狗，但是我们现在利用animal进行多态的调用。

更棒的是，子类实例化的动作不再需要在代码中硬编码，例如new Dog()，而是“在运行时才指定具体实现的对象”。

```
a = getAnimal( );
a.makeSound( );
```

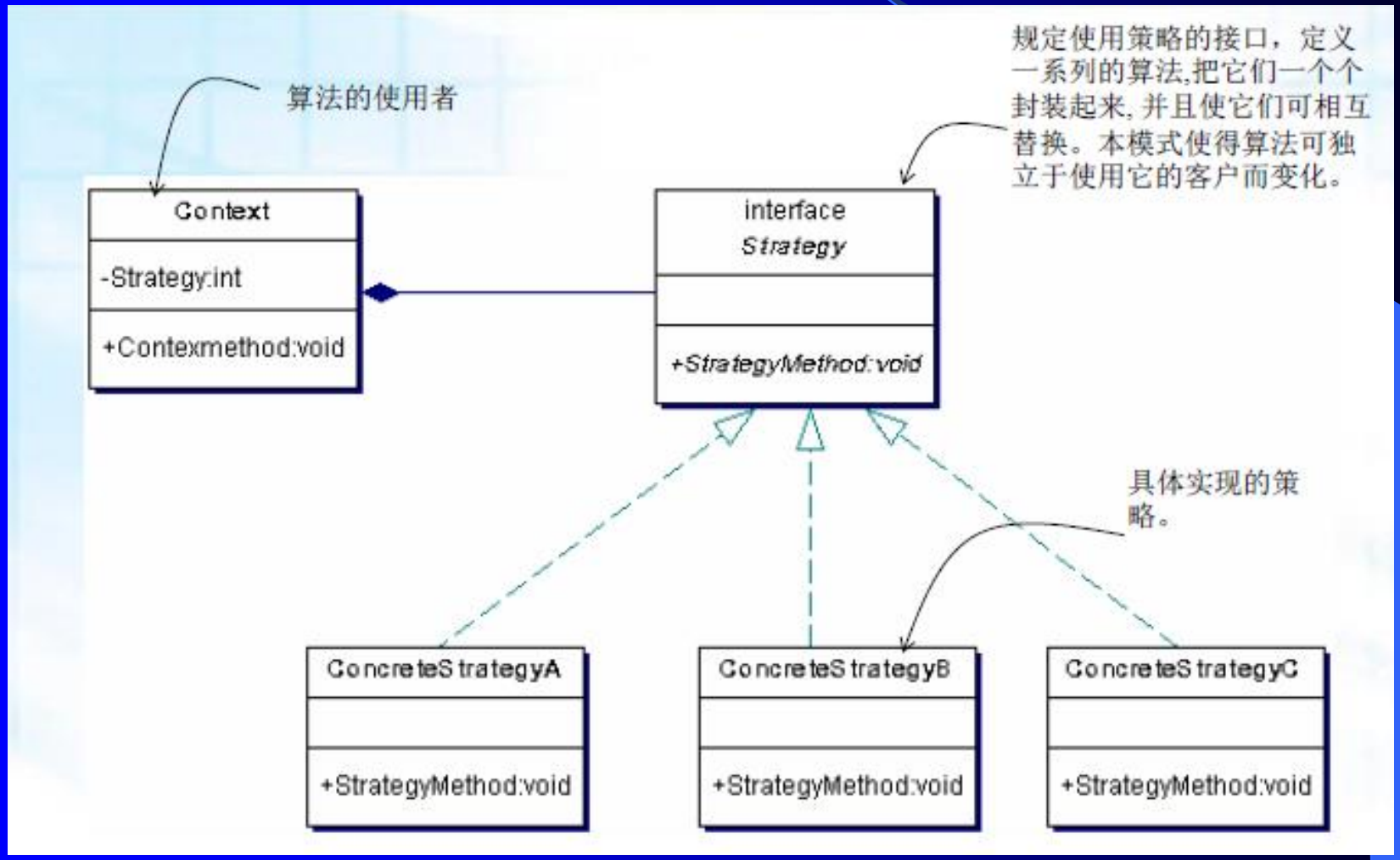
我们不知道实际的子类型是“什么”……我们只关心它知道如何正确地进行makeSound()的动作就够了。


«interface»
Subject
registerObservers ( )
removeObservers ( )
notifyObservers ( )
observers
«interface»
Observer
update ( )
«interface»
DisplayElement
display ( )
CurrentConditions
update ( )
display ( ) { // display
current measurements }
ForecastDisplay
update ( )
display ( ) { // display
the forecast }
WeatherData
registerObservers ( )
removeObservers ( )
notifyObservers ( )
getTemperature ( )
getHumidity ( )
getPressure ( )
measurementsChanged ( )
StatisticsDisplay
update ( )
display ( ) { // display
avg, min, and max
measurements }


客户端可以调用封装好的算法（行为）来实现飞行和叫的行为
Client
Duck
-flyBehavior:FlyBehavior
-quackBehavior:QuackBehavior
+swim():void
+display():void
+performFly():void
+performQuack():void
+setFlyBehavior():void
+setQuackBehavior():void
MallardDuck
+display():void
RedheadDuck
+display():void
RubberDuck
+display():void
DecoyDuck
+display():void
飞行行为的封装
<< interface >>
FlyBehavior
+fly():void
FlyWithWings
FlyNoWay
我们可以认为每一个行为都是一组算法
<< interface >>
QuackBehavior
+quack():void
鸭子叫行为的封装
Quack
Squeak
MuteQuack
这些算法（行为）都是可以被替换的

- 最佳团队：TREE团队（李海峰，亚航，刘恩泽），李孟珂团队
- 最努力团队：刘腾，王喆，张晶，朱天宝团队
- 最优爱心团队：小腿组
- 最佳合作团队：FG组

策略模式相关的实体


规定使用策略的接口，定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。
算法的使用者
Context
-Strategy:int
+Contexmethod:void
interface
Strategy
+StrategyMethod:void
具体实现的策略。
ConcreteStrategyA
+StrategyMethod:void
ConcreteStrategyB
+StrategyMethod:void
ConcreteStrategyC
+StrategyMethod:void

设计模式是什么？

- 设计模式三要素：
- 特定领域，问题，解决方案

三原则

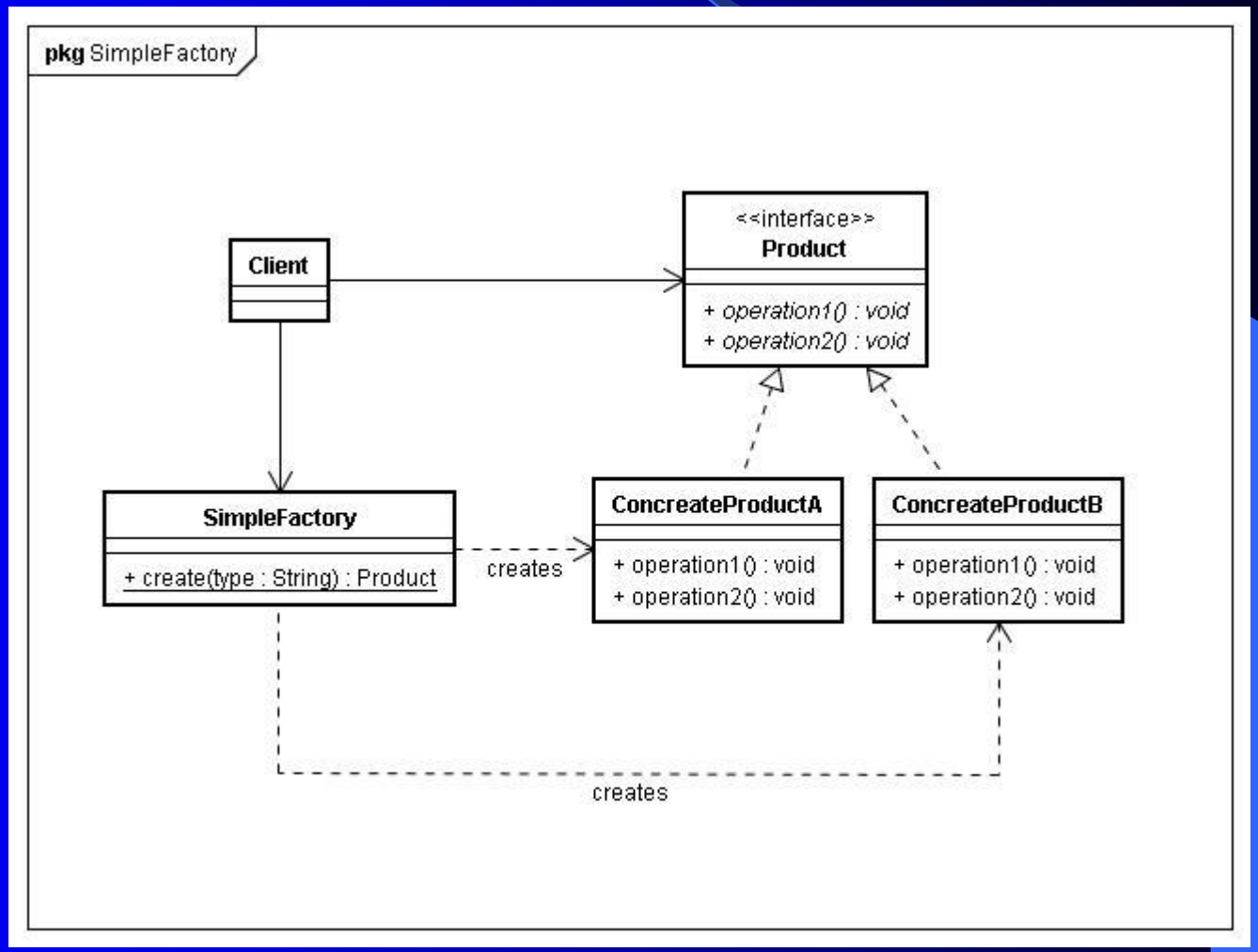
- 针对接口编程
- 封装变化
- 尽量使用对象组合而不是继承

创建型模式

- 简单工厂
- 工厂方法
- 抽象工厂

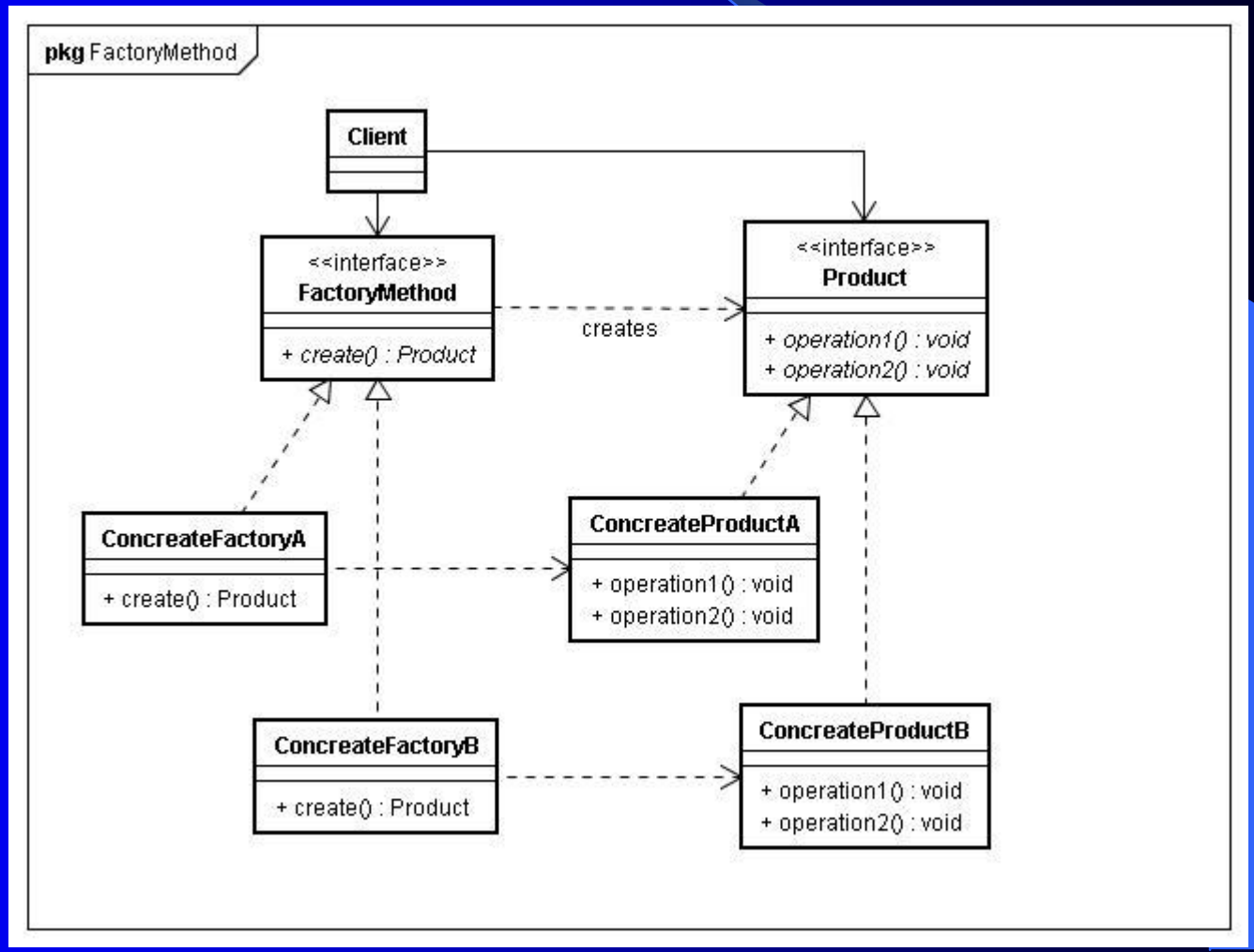
简单工厂

- 通过接收的参数的不同来返回不同的对象实例。



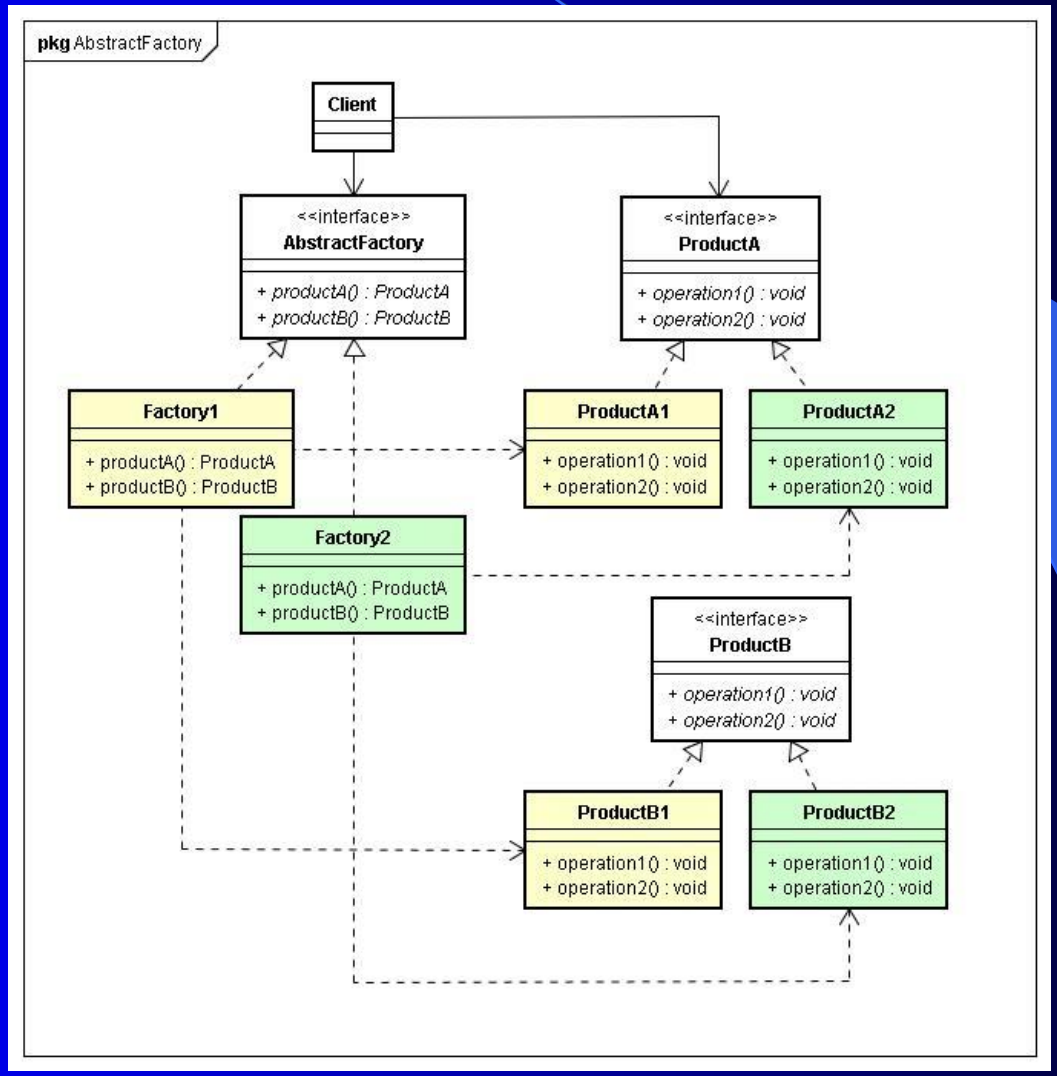
工厂方法

- 工厂方法是针对每一种产品提供一个工厂类。通过不同的工厂实例来创建不同的产品实例。



抽象工厂

- 抽象工厂是应对产品族概念的。



Pizza连锁店

- 1，顾客可以根据菜单点Pizza
- 2，Pizza的种类不同
- 3，每个店的Pizza根据地区不同风味不同
- 4，每个店可以自己加入一些秘方
- 5，比萨制作过程：准备，烘烤，切边，装盒
- 6，请给出关键方法和客户代码的实现

识别可变的特征……

假定你拥有一家比萨店，作为一家切边比萨店的拥有者，你可能写这样的代码：

```
Pizza orderPizza(){
        Pizza pizza = new Pizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
```

比萨的制作过程：比萨准备 → 烘烤 → 切边 → 装盒

为了使得代码具备柔性和扩展能力，我们非常希望这是一个抽象类或者是个接口，但如果作为抽象类或者接口，我们将不能直接实例化它。

但比萨有很多种类.....

由于比萨的种类很多，所以你需要增加一些代码来定义你需要的比萨类型，并创建对应的实例：

```
Pizza orderPizza (String type){
        Pizza pizza;

        if(type.equals("chesse")){
                pizza = new CheesePizza();
        } else if(type.equals("greek")){
                pizza = new GreekPizza();
        } else if (type.equals("pepperoni")){
                pizza = new PepperoniPizza();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

通过参数传递比萨的类型

基于各种比萨的类型，我们实例化相应的类并且指派比萨实例变量。注意，在这里每一种比萨都已经不得不实现了Pizza这个接口类，即具体实现了prepare()，bake()，cut()，box()等方法。

一旦我们获取了一种Pizza类（具体类），我们就调用该具体类的已经实现prepare()，bake()，cut()，box()的方法来制作，烘烤，切边以及装盒。

每一个Pizza的子类型都均已实现了prepare()，bake()，cut()，box()的方法

pepperoni
意大利辣香肠

真正有压力的是要增加新的pizza种类

你意识到你所有的竞争者在其菜单中已经增加了一组时尚的比萨种类： Clam Pizza（蛤蛤比萨） 和Veggie Pizza（素食比萨）。很显然，你需要保持竞争力，因此你应当在你的菜单中也加入这两种比萨。并且你已经很久没有卖Greek Pizza（希腊比萨） 了，所以你决定从菜单中删除掉它。

```
Pizza orderPizza(String type){
        Pizza pizza;

        if(type.equals("chesse")){
                pizza = new CheesePizza();
        } else if(type.equals("greek")){
                pizza = new GreekPizza();
        } else if (type.equals("pepperoni")){
                pizza = new PepperoniPizza();
        } else if (type.equals("clam")){
                pizza = new ClamPizza();
        } else if (type.equals("veggie")){
                pizza = new VeggiePizza();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

这些代码并不是拒绝修改的。如果比萨店改变了比萨的供应菜单，我们将不得不进入这些代码来修改它。

这就Pizza订单中常常变化的部分，每一次Pizza种类的不变化，你将不得不一次又一次的修改这些代码

这些就是我们期望看到的不会变化的部分。对于大多数比萨种类来说，其制作，烹饪，和包装在多年都是不会产生变化的。所以我们不希望这部分代码产生变化，让具体的pizza自己操作自己的方法就可以了。

封装对象的创建

OK，我们现在意识到，我们最好是能够把对象的创建从oderPizza()中移出来。我们应该怎么做呢？我们可以把这部分代码转移到另外一个对象中，这个对象专门用来创建具体的比萨种类。

```
Pizza orderPizza(String type){
        Pizza pizza;

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
        }
```

首先我们把对象的创建部分的代码从orderPizza()方法中移出来。

```
if(type.equals("chesse")){
        pizza = new CheesePizza();
} else if(type.equals("greek")){
        pizza = new GreekPizza();
} else if (type.equals("pepperoni")){
        pizza = new PepperoniPizza();
} else if (type.equals("clam")){
        pizza = new ClamPizza();
} else if (type.equals("veggie")){
        pizza = new VeggiePizza();
        }
```

接下来我们把移出来的代码放在一个对象中，这个对象只需要关心怎么创建Pizza。如果其他对象需要某种pizza，则由这个对象来负责。

现在我们知道有这么一个对象来专门负责生产或者创建pizza，我们可以把这个对象叫做Factory（工厂）。

orderPizza()和SimplePizzaFactory的关系

在此处orderPizza()方法变成了对象SimplePizzaFactory的客户端了。当orderPizza()方法需要一份比萨时，它就要求SimplePizzaFactory生产一份给它。过去orderPizza()需要知道Greek与Clam比萨之间制作上的区别，但现在只需要关心它可以获得一个比萨对象就可以了，这个对象已经实现了Pizza接口，因此它可以调用prepare()，bake()，cut()，box()等方法。

建立一个简单比萨工厂

我们将从工厂的实现开始。定义一个类来封装所有pizza对象创建的行为……

这是我们新建的一个类，SimplePizzaFactory，它只有一个职责，就是为它的客户端生产pizza

首先我们在工厂中定义了一个createPizza()方法，其所有客户端都是通过这个方法来实例化新的对象。

```
public class SimplePizzaFactory {
        public Pizza createPizza(String type){
                Pizza pizza = null;

                if(type.equals("chesse")){
                        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;
        }
} ///:~
```

这些代码是我们从orderPizza()方法中移出来的。

和原来的代码一样，这些代码通过pizza类型而参数化的。

重构PizzaStore类

现在我们来修补我们客户端的代码，我们期望通过工厂来生产客户端想要的Pizza。
来看看我们改变的代码：

给PizzaStore设置一个引用指向SimolePizzaFactory

```
public class PizzaStore{
        SimplePizzaFactory factory;

        public PizzaStore(SimplePizzaFactory factory){
                this.factory = factory;
        }

        public Pizza orderPizza(String type){

                Pizza pizza = factory.createPizza(type);

                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
                return pizza;
        }

        // other methods here
} ///:~
```

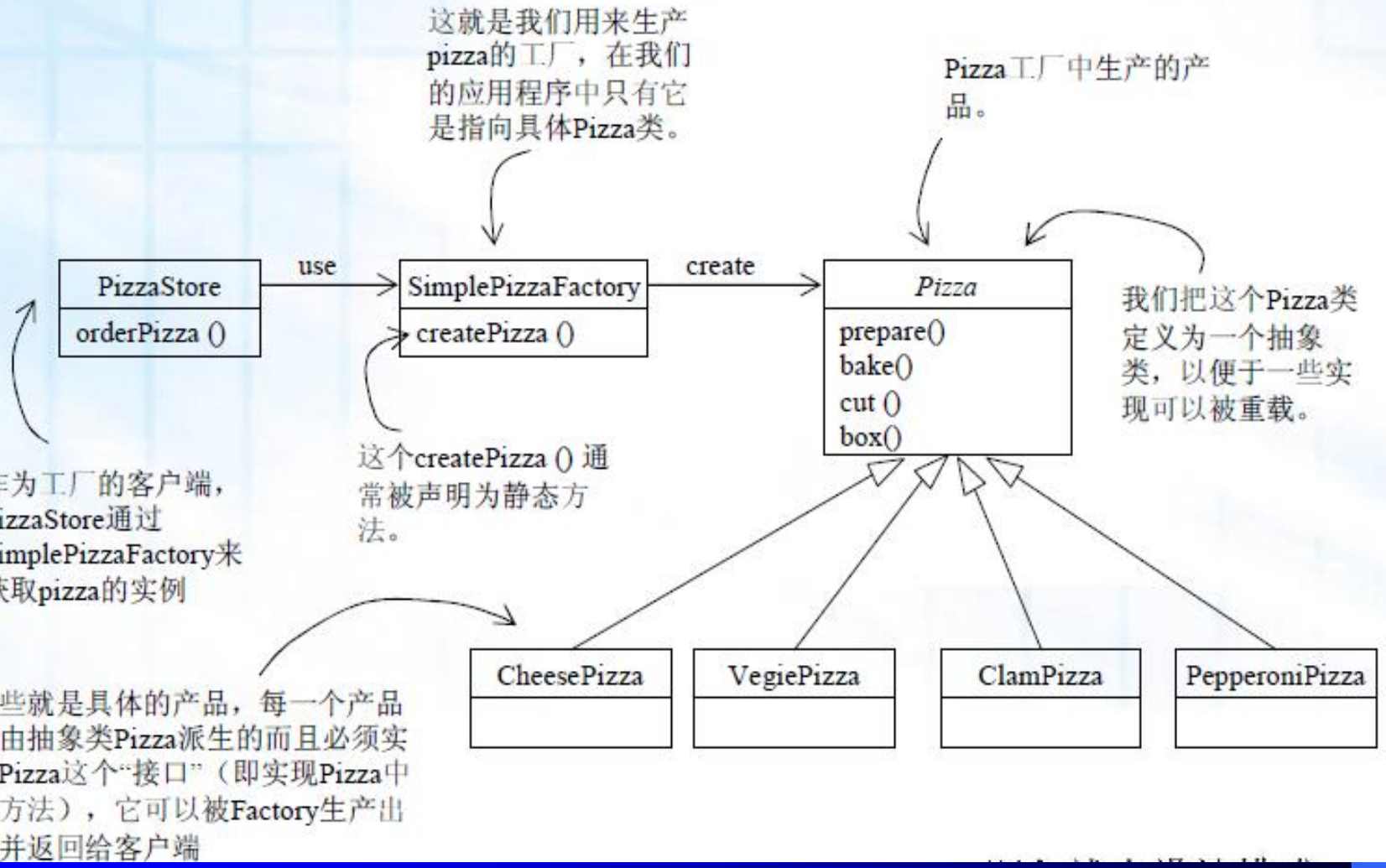
PizzaStore获取factory并把它传递给构造函数

orderPizza()方法只需要传入订单类型就可以使用factory来生产它想要的pizza

注意：我们已经替换了工厂对象中的生产方法。在这里并没有具体的实现。

简单工厂模式的定义

简单工厂模式是类的创建模式，又叫做静态工厂方法模式。就是由一个工厂类根据传入的参量决定创建出哪一种产品类的实例

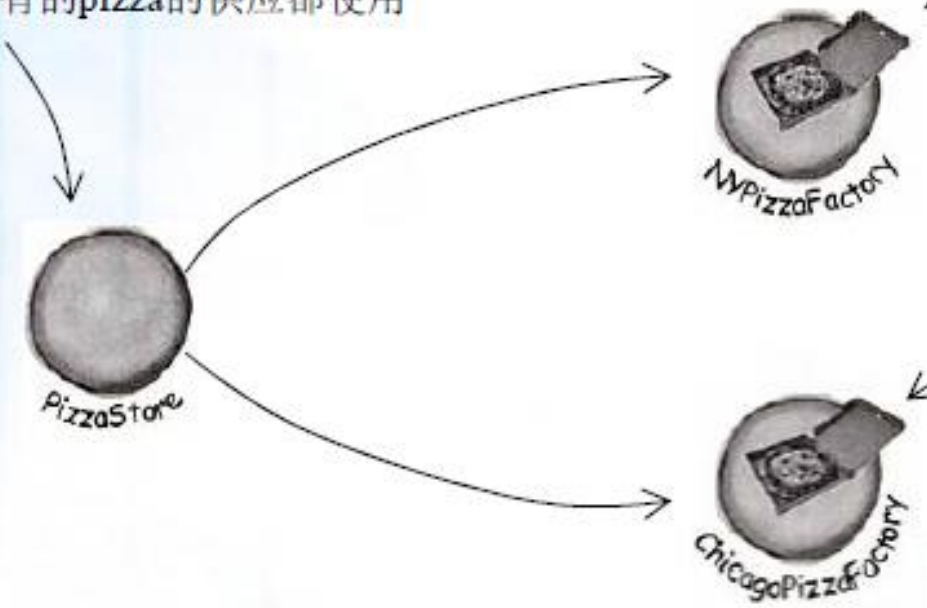


练习

现在pizza店已经做得很不错了，接下来我们需要对我们的竞争对手来个迎头痛击了，再者，我们的邻居们都希望在身边有一个Pizza店。作为经销商，你想确保所有销售都能够保证质量，所以你希望他们都能使用我们这些经受了时间考验的代码。

但是我们如何保证地区之间不同风味呢？每个专卖店都希望能提供不同风味的pizza（例纽约风味，芝加哥风味，以及加州风味等）。这取决于专卖店所在的位置及其顾客群的口味。

我们希望专卖店能够重用用PizzaStore的代码，以便于所有的pizza的供应都使用的同一种方法。



一个专卖店希望有一个工厂能生产纽约风味的pizza，有薄面包皮，可口的沙司，以及少许干酪。

另外一个专卖店希望有一个工厂能生产芝加哥风味的pizza，有厚厚的面包皮，口味沙司，还有较多的干酪。

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("Veggie");
```

此处我们建立了一个工厂专门来生产纽约风味的pizza。

这里我们创建了一个PizzaStore并把它指向nyFactory。

当我们制造pizza时，得到是纽约风味的pizza。

```
ChicagoPizzaFactory chicagoFactory = new
ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.orderPizza("Veggie");
```

- 专卖店想加入自己的一些秘方

PizzaStore的框架（Framework）

有一个方法可以使得PizzaStore类中的pizza制作活动变得本地化，并且可以给经销商一些自由使之拥有自己地域的风味。

我们将要做的是把createPizza()方法再放回PizzaStore中，但是这一次是一个抽象方法（abstract method），我们可以在PizzaStore的子类实现符合本地风味。

来看看PizzaStore的变化：

PizzaStore 现在变成了一个抽象类了。

```
public abstract class PizzaStore {
    public Pizza orderPizza (String type) {
        Pizza pizza;

        pizza = createPizza (type);

        pizza.prepare ( );
        pizza.bake ( );
        pizza.cut ( );
        pizza.box ( );

        return pizza;
    }

    abstract createPizza (String type);
}
```

现在createPizza()重新放了回来，在PizzaStore中调用比在一个工厂对象中调用要好得多。

这些看起来没什么变化

现在我们把工厂对象转移到这个方法中了。

在PizzaStore中“工厂方法”是抽象方法。

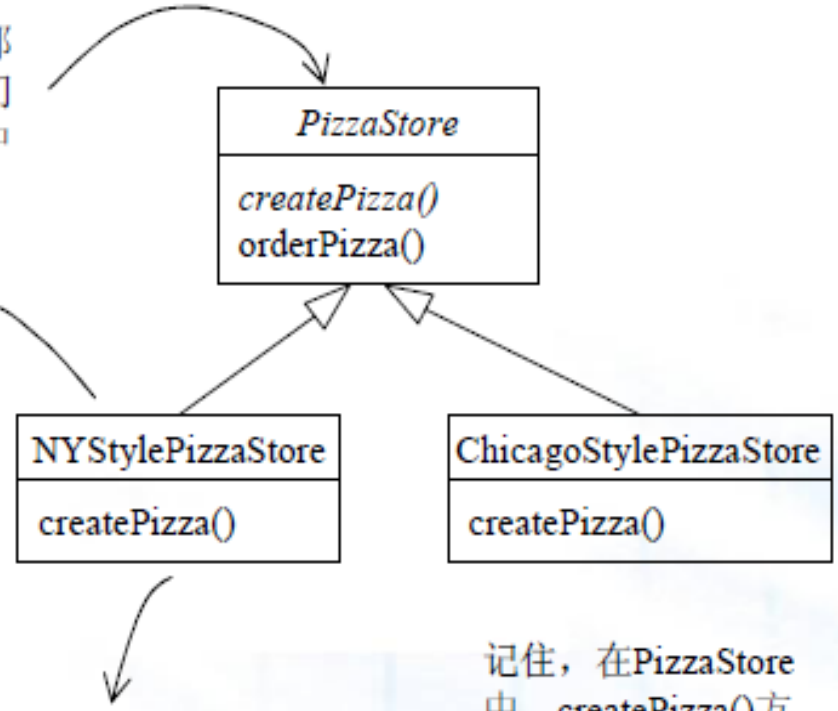
允许子类自己来决定如何做.....

记住，PizzaStore系统中已经有一个很好的订单系统，即orderPizza()方法，我们可以确认这个方法在所有的专卖店都是一致的。

变化的部分是Pizza的制造部分-----纽约风味需要薄面包皮，芝加哥需要厚面包皮等等。我们打算把这些变化的部分都封装在createPizza()方法中，并使之可以制作合适的pizza种类。我们允许每个PizzaStore的子类实现具体的createPizza(),那么我我们将拥有一些PizzaStore具体子类，每一个子类都可以制造一类不同的pizza。

每个子类都重载了createPizza()方法，并且每个子类都使用PizzaStore中定义的orderPizza()方法时，如果我们希望子类都强制执行orderPizza()方法，在PizzaStore中我们可以把这个方法定义为final。

```
public Pizza createPizza (type) {
    if (type.equals("cheese")){
        pizza = new NYStyleCheesePizza();
        }
    else if (type.equals("pepperoni")){
        pizza = new NYStylePepperoniPizza();
        }
    else if (type.equals("clam")){
        pizza = new NYStyleClamPizza();
        }
    else if (type.equals("vggie")) {
        pizza =new NYStyleVeggiePizza();
        }
}
```



如果某个经销商想为顾客提供纽约风味的pizza，他可以使用这个子类，它有自己的createPizza()方法，可以制造纽约风味的pizza。

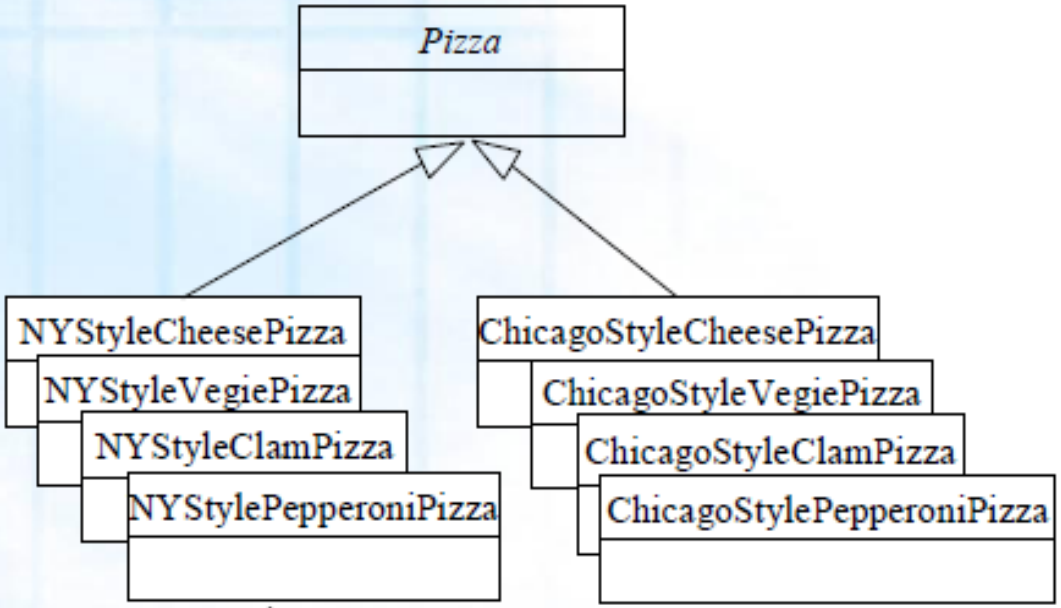
记住，在PizzaStore中，createPizza()方法是抽象的，它所有的子类必须实现它。

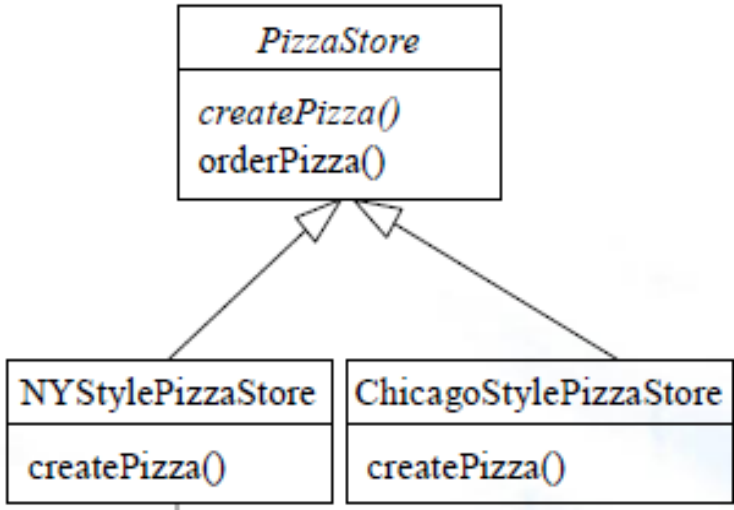
从平行的角度来看类的层次

从平行的角度来看看类的层次：两个都是抽象类，均可以派生出具体的类。

The Product Classes

The Creator Classes





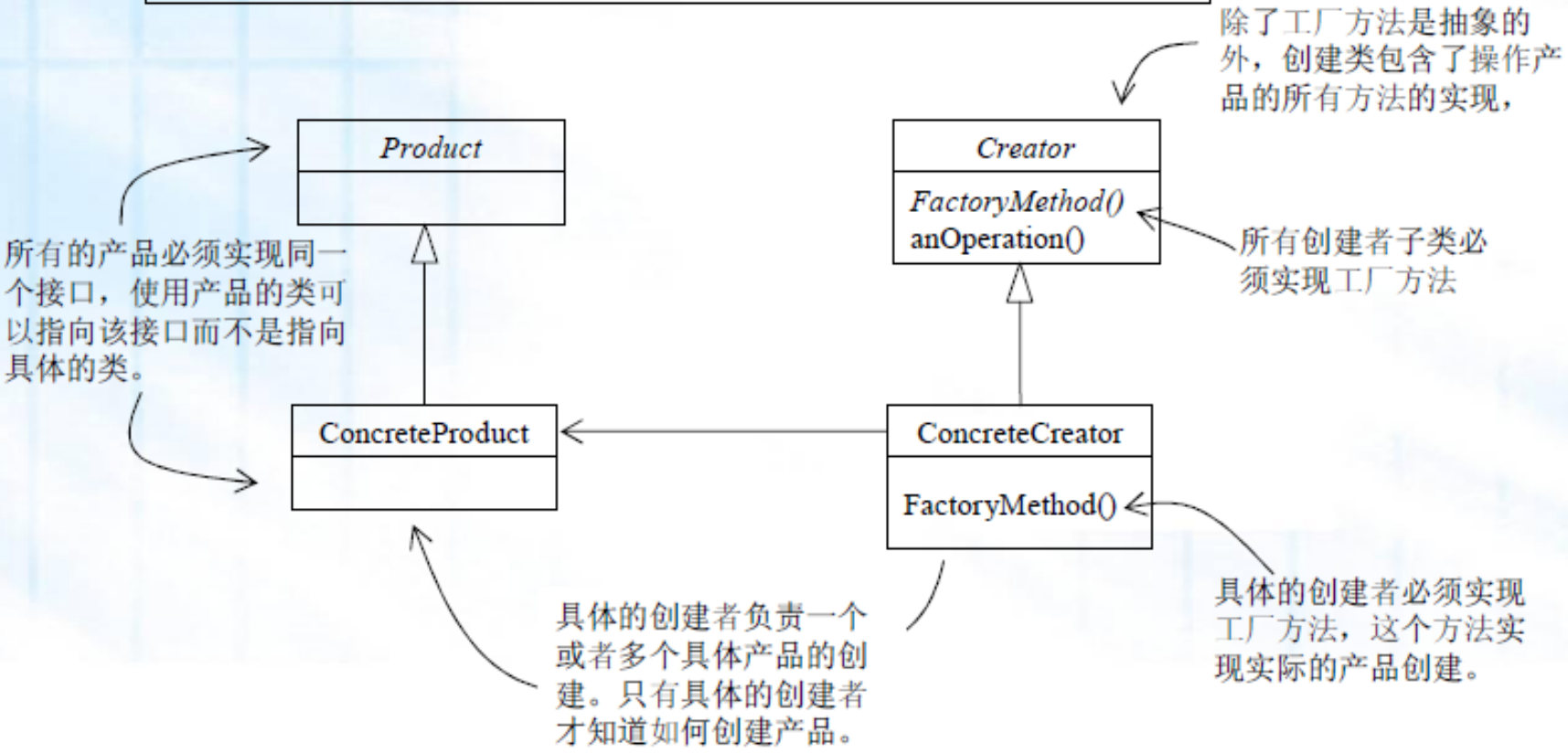
NYStylePizzaStore 把如何制作纽约风味Pizza封装了起来。

工厂方法是封装“知识”的关键

工厂方法的定义

现在我们来看工厂方法的官方定义：

工厂方法定义了定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。



练习

- 生产汽车
- 轿车、货车、客车