

Object Oriented Analysis & Design

面向对象分析与设计

Lecture_10 GOF设计模式 (二)

1) 策略模式 2) 工厂

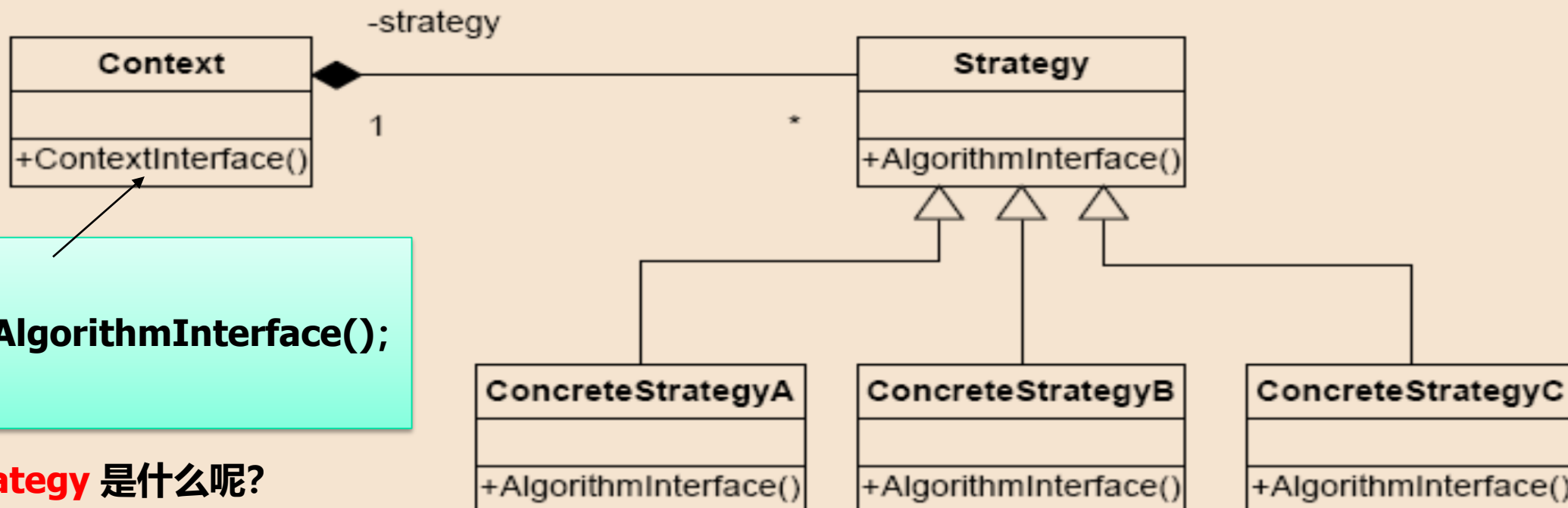
主讲: 姜宁康 博士



■ 3、简单工厂模式 **Factory Design Pattern...**

- 这不是真正意义上的工厂模式，但工厂模式是建立在它之上的

复习：策略模式 strategy Pattern

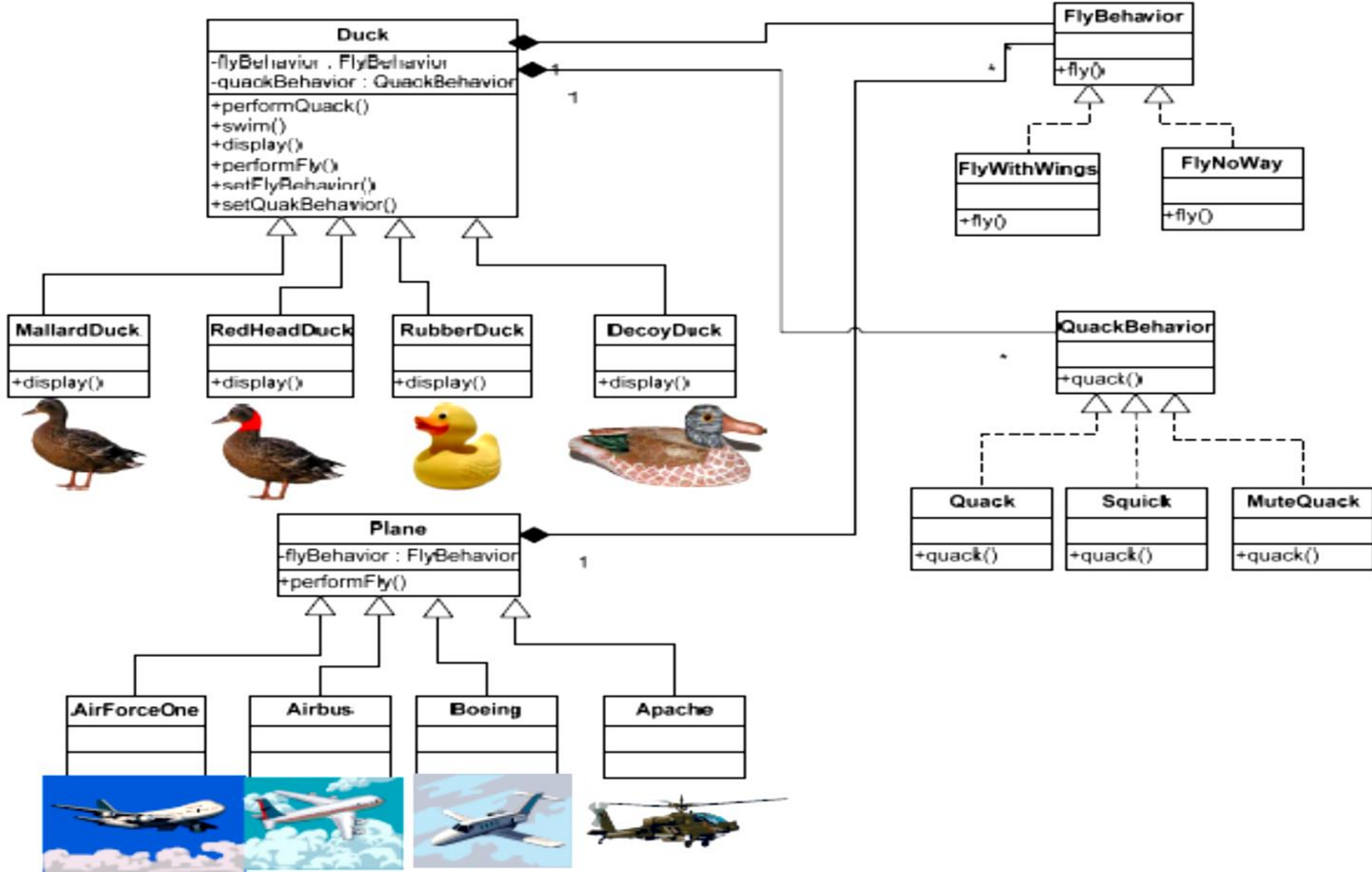


这里的 **strategy** 是什么呢?

Strategy – defines a family of algorithms, encapsulate each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

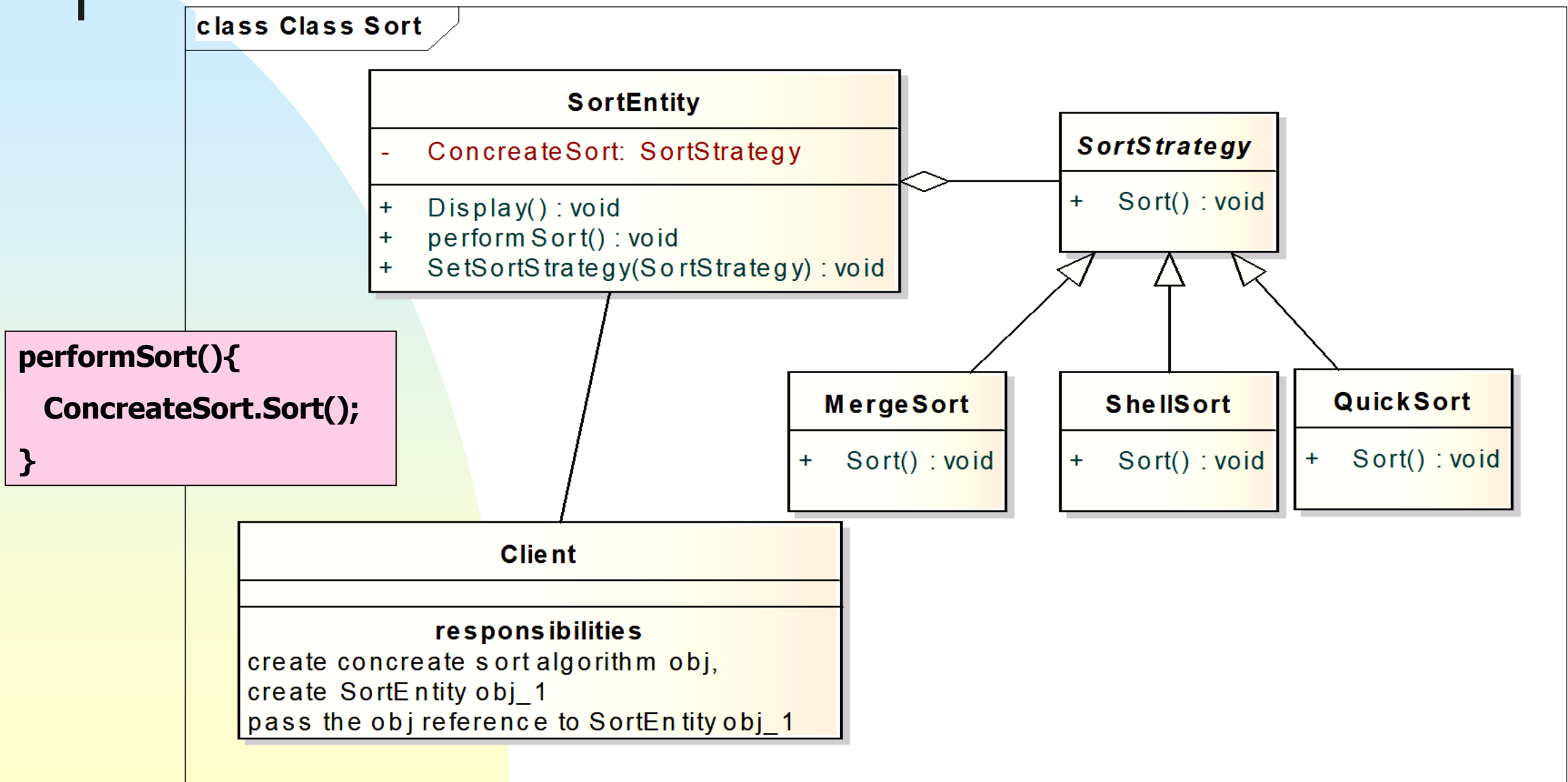
定义了算法族，分别封装起来，让他们之间可以互相替换，此模式让算法的变化独立于使用算法的客户

复习：策略模式案例



复习：策略模式案例

- 某一个应用系统需要用到排序算法，同时允许用户动态地选择不同的排序算法。请用设计模式给出设计方案



3.1 工厂模式准备 Factory Patterns

- 设计原则: “Don’t program to an implementation, program to an interface (abstraction)”
- However, every time you do a “new” you need to deal with a “concrete” class, not an abstraction

Duck duck = new MallardDuck ();

We want to use
interfaces to keep
code flexible

But we have to
create an instance
of a concrete class

Whats wrong with this?
What principle is broken here?

违反了OCP原则

With a whole set of related concrete
classes:

```
{...  
    Duck duck;  
  
    if (picnic) duck = new MallardDuck ();  
    else if (hunting)  
        duck = new DecoyDuck ( );  
    else if (inBathTub)  
        duck = new RubberDuck ( );  
}
```

3.1 工厂模式准备 new()方法有什么问题呢?

从技术的角度来说，通过new来创建对象是没错的，问题在于“变化”，以及变化对使用new方法带来的影响。

不过，当代码需要使用很多**具体的类**时，增加新的具体类的时候，我们不得不修改代码

← 这里不符合“开-闭原则了”！

但是你却不得不在某些地方创建一个对象，并且Java中只提供了一种方法（new）来创建对象，不是吗？难道还有其他途径吗？

那怎么办呢？



3.1 工厂模式准备

- **Principle:** 标识程序中那些多变的部分，并且把它们和稳定的部分隔离开来 Identify the aspects that vary and separate them from what stays the same
 - 参考GRASP principle: Protected variation 隔离变化
- 如何把软件系统中实例化具体类的所有场合、与系统的其余部分隔离或者封装起来？ How might you take all the parts of your application that instantiate concrete classes and separate or encapsulate them from the rest of your application

3.2 案例背景

■ 背景

- 在高科技的“对象村”拥有一家pizza店，负责为附近的居民提供比萨
- 设计一个仿真的pizza店软件系统

3.3 正常设计

/*Own a pizza store in cutting edge Objectville!

public class PizzaStore {

Pizza orderPizza () {

Pizza pizza = new Pizza ();

pizza.prepare () ;

pizza.bake ();

pizza.cut ();

pizza.box ();

return pizza;

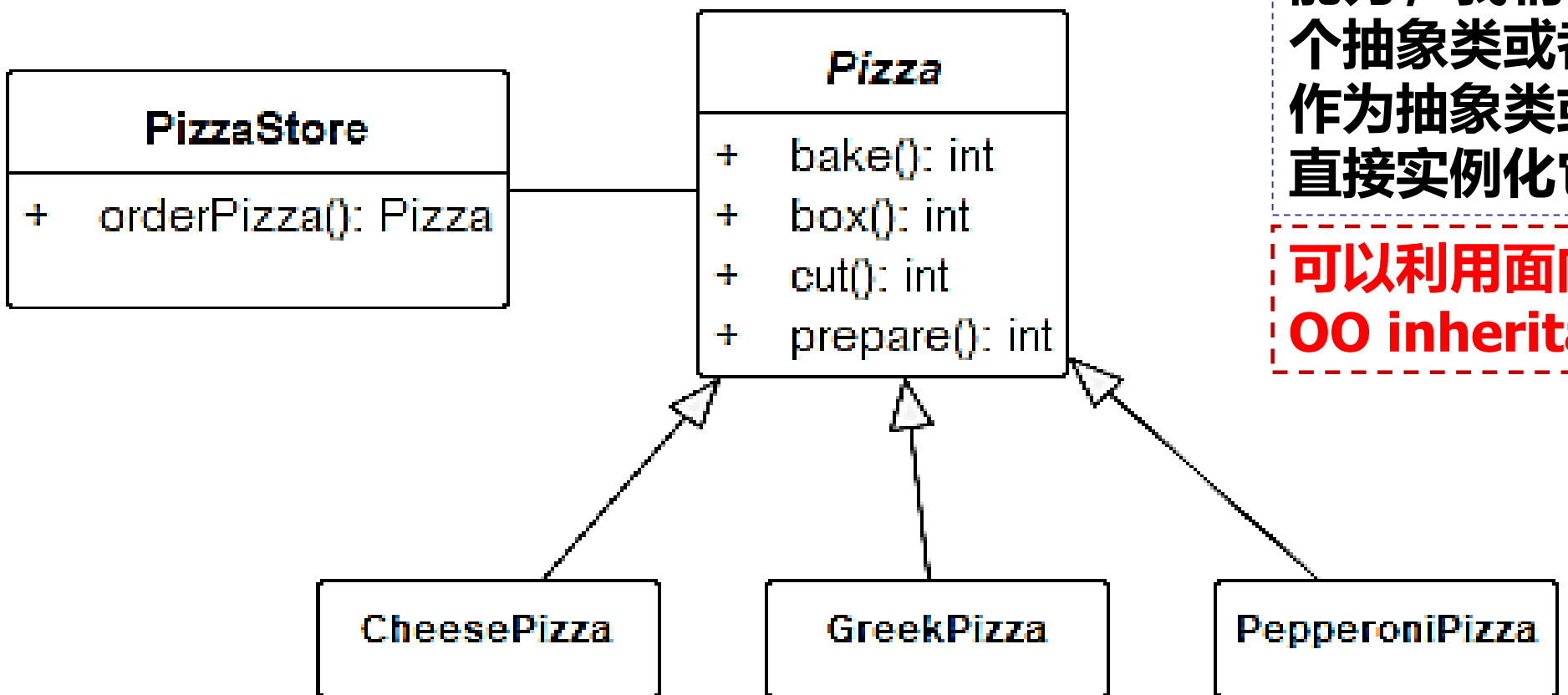
}

}



为了使得代码具备灵活性和扩展能力，我们非常希望**Pizza**是一个抽象类或者是个接口，但如果作为抽象类或者接口，我们又不能直接实例化它

3.3 正常设计



为了使得代码具备灵活性和扩展能力，我们非常希望Pizza是一个抽象类或者是个接口，但如果作为抽象类或者接口，我们又能直接实例化它

可以利用面向对象的继承机制
OO inheritance can help us

意大利辣香肠披萨

3.4 哪些是可能的变化点呢?

■ 客户需要多种类型的Pizza

```
Pizza orderPizza ( String type) { //通过参数传递pizza的类型  
    Pizza pizza;
```

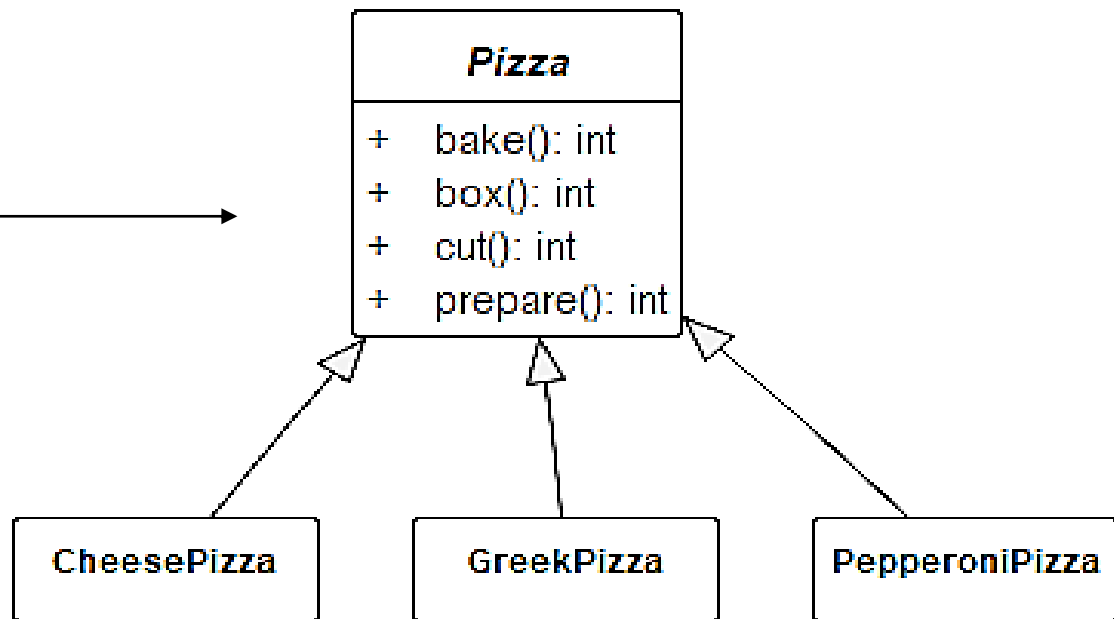
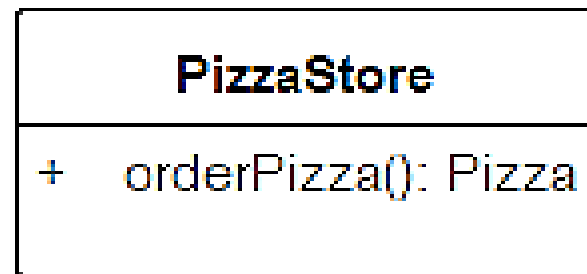
```
    if (type.equals ("cheese")) {  
        pizza = new CheesePizza ();  
    } else if (type.equals("greek")) {  
        pizza = new GreekPizza ( );  
    } else if (type.equals("pepperoni")) {  
        // Pepperoni 意大利辣香肠  
        pizza = new PepperoniPizza ( );  
    }  
}
```

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

```
}
```

一旦我们获取了一种具体的Pizza类实例，我们就调用该具体类已经实现的prepare(), bake(), cut(), box()方法来制作、烘烤、切边以及装盒

依据类型实例化不同的 Pizza 原型
Instantiate based on type of pizza



3.4 哪些是可能的变化点呢？

- 当需要增加新的种类、删除不受欢迎的种类的时候
 - 店主意识到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（蛤蜊比萨）和 Veggie Pizza（素食比萨） ...
 - 很显然，为了保持竞争力，店主应当在菜单中也加入这两种比萨
 - 并且发现Greek Pizza（希腊比萨）已经很久没人买了，所以决定从菜单中删除掉它
- 怎么做呢？
- 在前面的设计中，哪些是不变的？哪些是可能会变化的？ What do you think would need to vary and what would stay constant?

3.4 哪些是可能的变化点呢?

```
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("veggie")) {  
        pizza = new VeggiePizza ( );  
    } else if (type.equals("Clam")) {  
        pizza = new ClamPizza ( );  
    } else if (type.equals("pepperoni")) {  
        pizza = new PepperoniPizza ( );  
    }  
    pizza.prepare ();  
    pizza.bake ();  
    pizza.cut ();  
    pizza.box ();  
    return pizza;  
}
```

■ 修改orderPizza ()

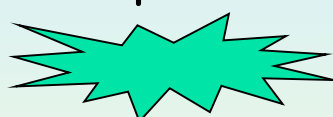
这部分会发生变化

这部分代码不符合OCP原则
增加功能时修改了原来的代码!

这部分功能保持不变

3.5 改进设计方案

- 把有关创建对象 (Pizza) 的一部分功能移出 orderPizza () 方法
- 定义一个新的类 SimplePizzaFactory, 专门负责创建 pizza 对象

```
Pizza orderPizza ( String type) {  
    Pizza pizza;  
      
    pizza.prepare ();  
    pizza.bake ();  
    pizza.cut ();  
    pizza.box ();  
    return pizza;  
}
```

移出来

这一部分保持不变

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

放到新定义类 SimplePizzaFactory

3.5 改进设计方案

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

简单工厂Factories处理创建对象的细节

与原来一样，依然通过pizza类型创建不同的Pizza

这是从orderPizza() 方法抽出来的功能

3.5 改进设计方案

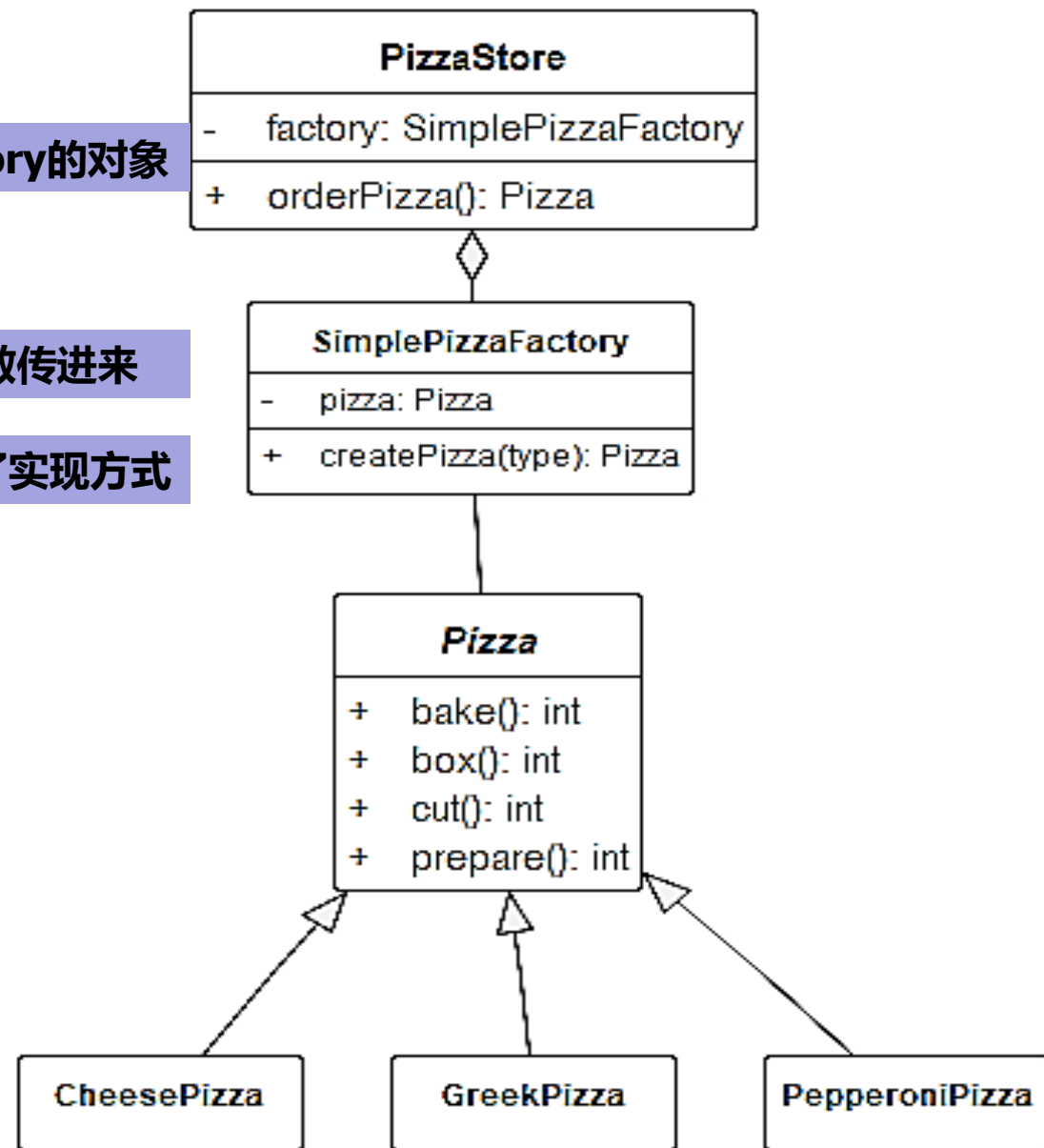
新的 PizzaStore class

```
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 ();  
        return pizza;  
    }  
}
```

Store有一个factory的对象

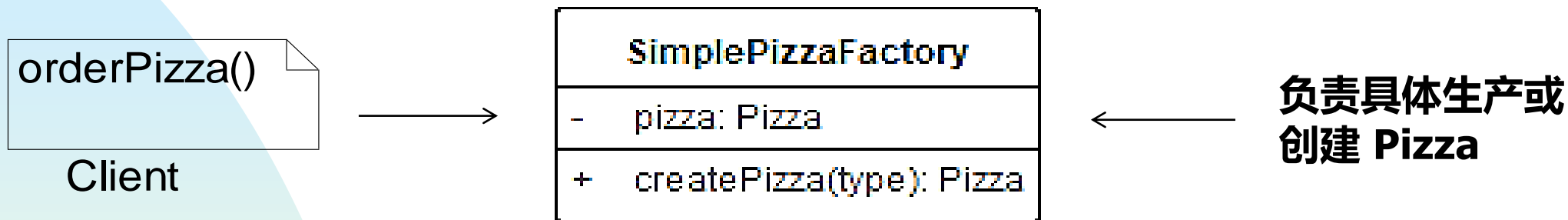
factory的对象通过构造函数传进来

orderPizza()改变了实现方式



3.5 改进设计方案的优点

orderPizza() 和 SimplePizzaFactory 的关系

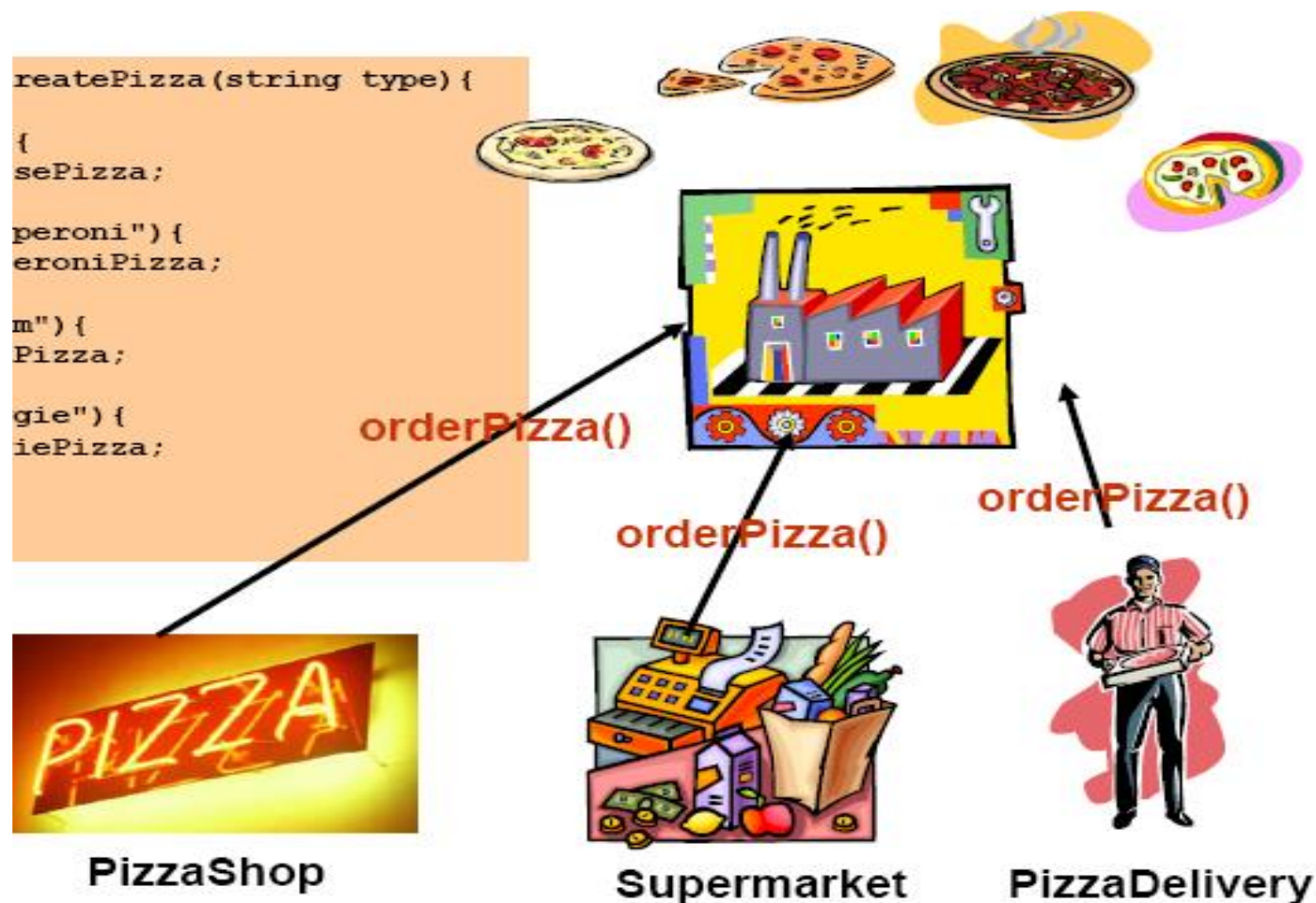


新方案，**orderPizza()**变成了对象SimplePizzaFactory的客户端，当orderPizza()需要一份比萨时，它就要求SimplePizzaFactory生产一个比萨对象，这个对象已经实现了Pizza接口，因此它可以调用prepare(), bake(), cut(), box()等方法

原来方案，**orderPizza()**需要知道Greek与Clam比萨之间制作上的区别，需要判断Pizza的类型

3.5 改进设计方案的优点

- SimplePizzaFactory 可以服务多个客户
 - PizzaShopMenu, HomeDelivery etc.
- 客户的代码再也不用关注具体的Pizzaa类型、制作细节

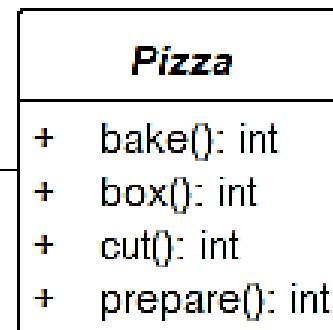
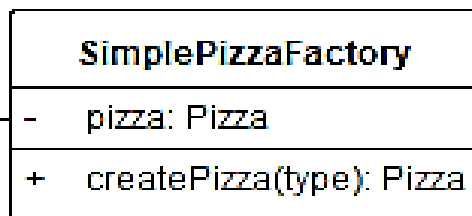
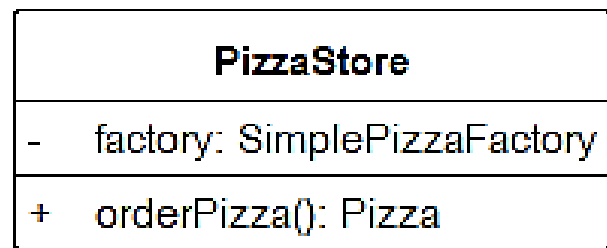


3.6 简单工厂模式 SimpleFactory Pattern

工厂的客户，从工厂得到Pizza实例

这是生产Pizza的工厂，这也应当是整个系统唯一涉及到具体Pizza类的地方

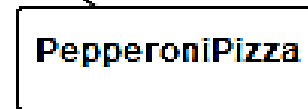
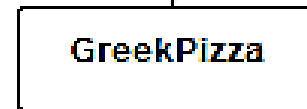
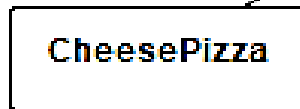
工厂的产品（抽象）
Product of the factory (abstract)



抽象类对实现有帮助，具体类需要时可以覆盖实现

create()方法有时被定义成静态的

具体产品，各自实现Pizza接口



3.6 简单工厂模式 SimpleFactory Pattern

- **SimpleFactory 不是真正意义上的工厂模式，但工厂模式却是建立在它之上的** SimpleFactory is not really a design pattern but the actual Factory patterns are based on it !





■ **本讲结束**