# 面向对象设计模式　工厂

曹东刚
caodg@pku.edu.cn

北京大学信息学院研究生课程 - 面向对象的分析与设计
http://sei.pku.edu.cn/~caodg/course/oo

# 内容提要

# new 操作符的问题

```
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

使用大量具体类而难以更新和维护的代码

```
1  Duck duck;
2  if (picnic) {
3    duck = new MallardDuck();
4  } else if (hunting) {
5    duck = new DecoyDuck();
6  } else if (inBathTub) {
7    duck = new RubberDuck();
8  }
```

# 怎么办?

到 OO 设计原则中寻找答案:

## 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。

如何将实例化具体类的代码抽离或封装起来?

# 识别变化的方面：从一个 Pizza 店应用开始

订购 Pizza 的代码

```
1  Pizza orderPizza() {
2    Pizza pizza = new Pizza();
3
4    pizza.prepare();
5    pizza.bake();
6    pizza.cut();
7    pizza.box();
8    return pizza;
9  }
```

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

# 识别变化的方面：从一个 Pizza 店应用开始

根据 Pizza 的类型实例化具体类

```
1  Pizza orderPizza(String type) {
2    Pizza pizza;
3    if (type.equals("cheese")) {
4      pizza = new CheesePizza();
5    } else if (type.equals("greek") {
6      pizza = new GreekPizza();
7    } else if (type.equals("pepperoni") {
8      pizza = new PepperoniPizza();
9    }
10
11   pizza.prepare();
12   // the following procedure is the same as the previous
```

# 识别变化的方面：从一个 Pizza 店应用开始

但随后需要增加和删除某些 Pizza

```
1  Pizza orderPizza(String type) {
2    Pizza pizza;
3    if (type.equals("cheese")) {
4      pizza = new CheesePizza();
5    } else if (type.equals("greek") {
6      pizza = new GreekPizza();
7    } else if (type.equals("pepperoni") {
8      pizza = new PepperoniPizza();
9    } else if (type.equals("clam") {
10     pizza = new ClamPizza();
11   } else if (type.equals("veggie") {
12     pizza = new VeggiePizza();
13   }
```

# 识别变化的方面：从一个 Pizza 店应用开始

将创建对象代码移出 orderPizza

```
1  Pizza orderPizza(String type) {
2    Pizza pizza;
3
4    将创建对象代码从此处移走
5
6    pizza.prepare();
7    pizza.bake();
8    pizza.cut();
9    pizza.box();
10   return pizza;
11 }
```

# 识别变化的方面：从一个 Pizza 店应用开始

将创建具体对象的代码放到一个工厂类里

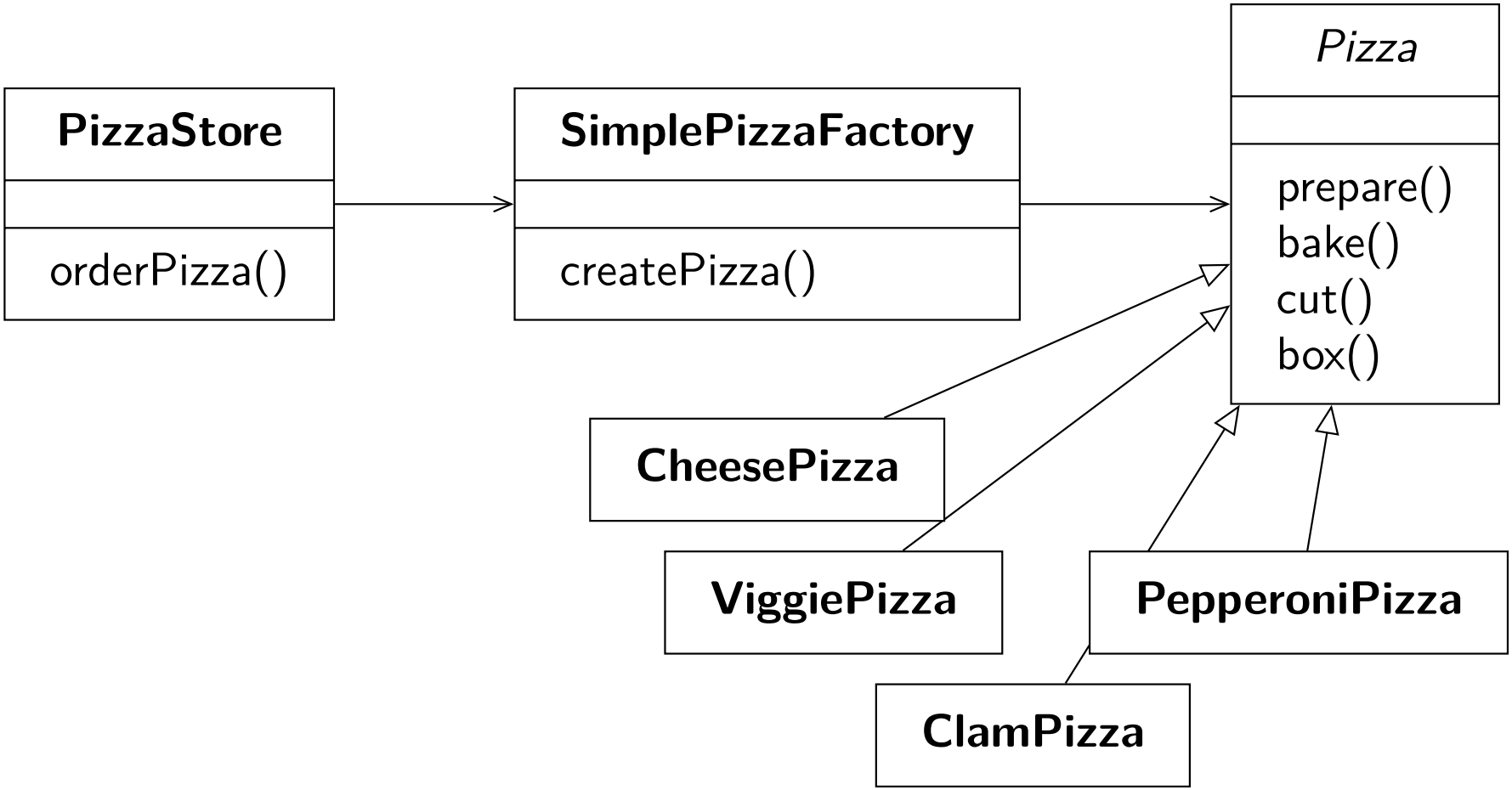
```
1  public class SimplePizzaFactory {
2    public Pizza createPizza(String type) {
3      Pizza pizza = null;
4      if (type.equals("cheese")) {
5        pizza = new CheesePizza();
6      } else if (type.equals("pepperoni")) {
7        pizza = new PepperoniPizza();
8      } else if (type.equals("clam")) {
9        pizza = new ClamPizza();
10     } else if (type.equals("veggie")) {
11       pizza = new VeggiePizza();
12     }
13     return pizza;
14   }
```

# 识别变化的方面：从一个 Pizza 店应用开始

重新实现 PizzaStore

```
1  public class PizzaStore {
2    SimplePizzaFactory factory;
3
4    public PizzaStore(SimplePizzaFactory factory) {
5      this.factory = factory;
6    }
7
8    public Pizza orderPizza(String type) {
9      Pizza pizza;
10     pizza = factory.createPizza(type);
11     pizza.prepare();
12     //the following procedure is the same as the previous
```

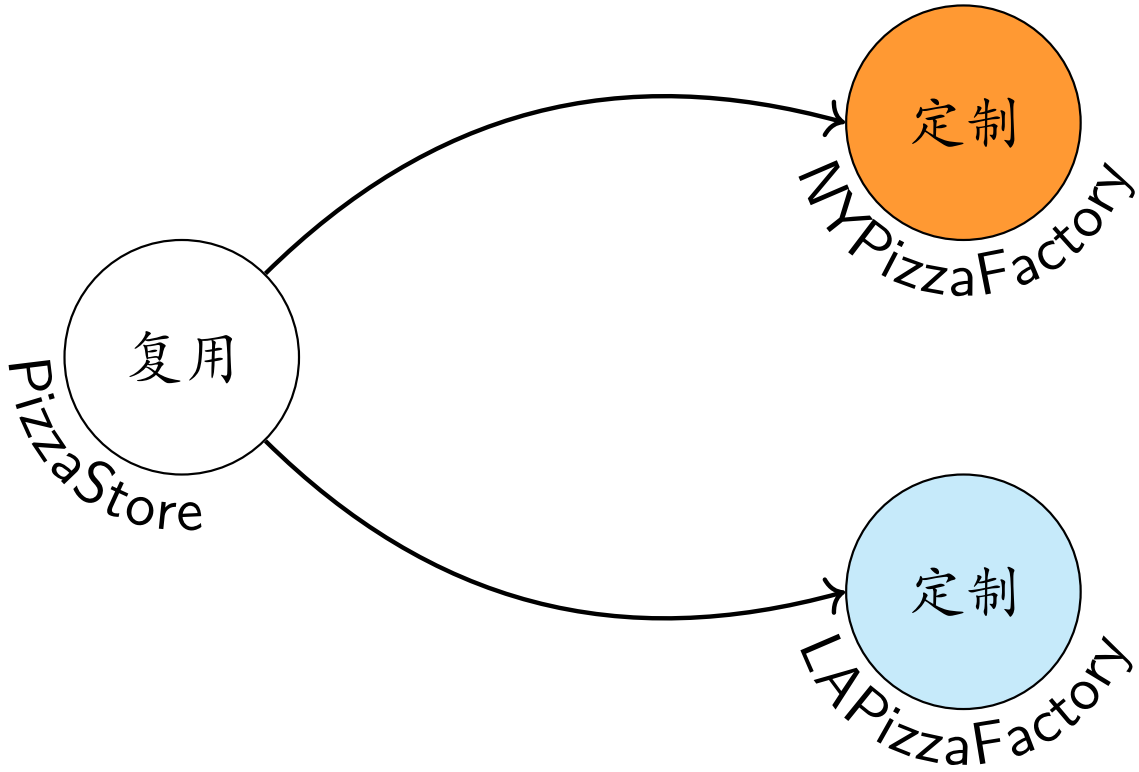
# 简单工厂类下的 Pizza 店新类图



# 内容提要

# 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



# 各加盟店不同的工厂类

纽约风味的工厂

```
1 NYPizzaFactory nyFactory = new NYPizzaFactory();
2 PizzaStore nyStore = new PizzaStore(nyFactory);
3 nyStore.order("Veggie");
```

洛杉矶风味的工厂

```
1 LAPizzaFactory laFactory = new LAPizzaFactory();
2 PizzaStore laStore = new PizzaStore(laFactory);
3 laStore.order("Veggie");
```

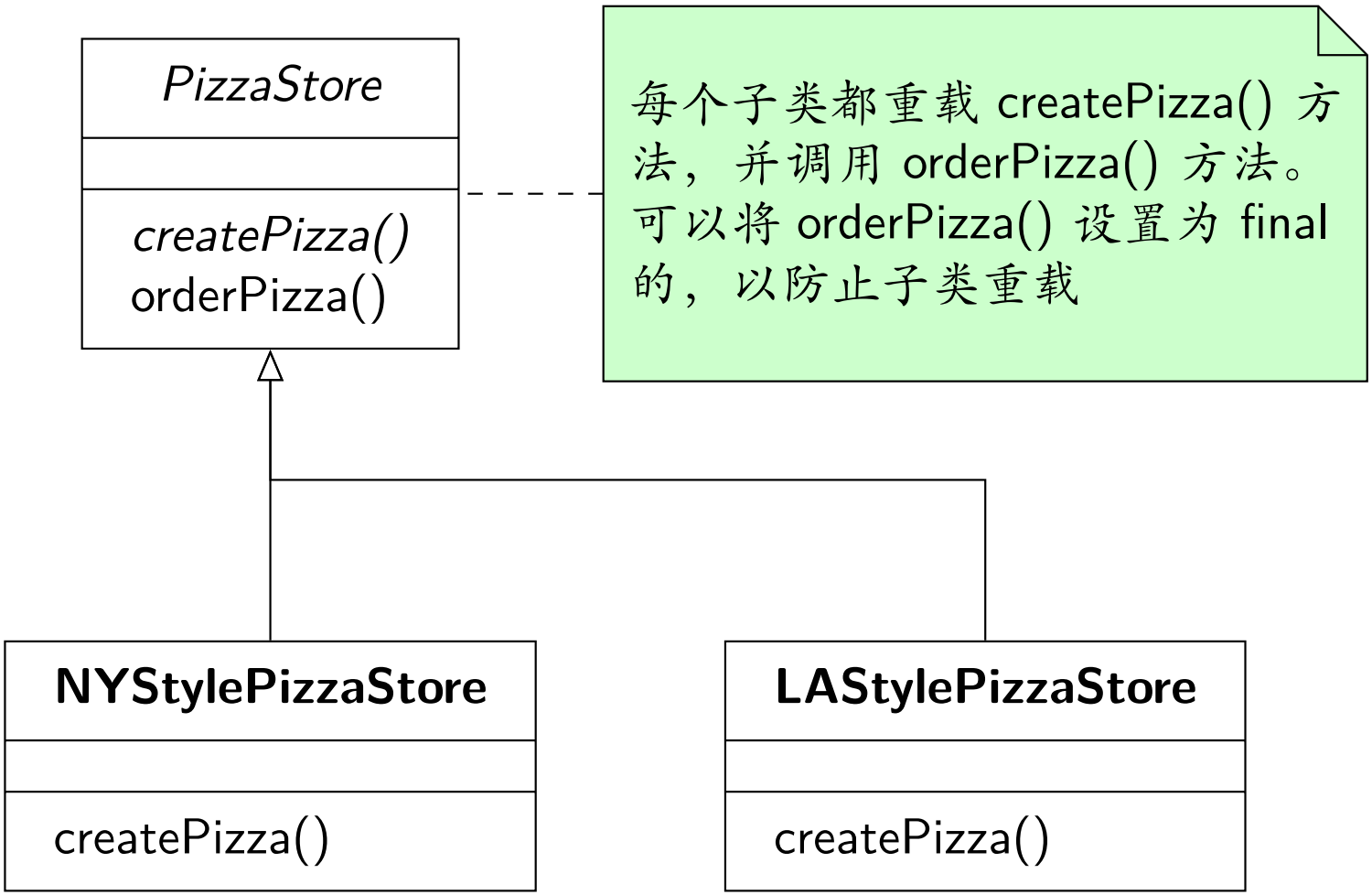
# 既有严格控制，又允许灵活性

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



# 采用工厂方法的新设计

PizzaStore

```
1  public abstract class PizzaStore {
2    public final Pizza orderPizza(String type) {
3      Pizza pizza;
4      pizza = createPizza(type);
5      pizza.prepare();
6      pizza.bake();
7      pizza.cut();
8      pizza.box();
9      return pizza;
10   }
11   protected abstract Pizza createPizza(String type);
12   // other methods here
13 }
```

# 采用工厂方法的新设计

纽约地区的 PizzaStore

```
1  public class NYPizzaStore extends PizzaStore {
2    Pizza createPizza(String item) {
3      if (item.equals("cheese")) {
4        return new NYStyleCheesePizza();
5      } else if (item.equals("veggie")) {
6        return new NYStyleVeggiePizza();
7      } else if (item.equals("clam")) {
8        return new NYStyleClamPizza();
9      } else if (item.equals("pepperoni")) {
10       return new NYStylePepperoniPizza();
11     } else return null;
12   }
13 }
```

# 采用工厂方法的新设计

Pizza

```
1  public abstract class Pizza {
2    String name;
3    String dough;
4    String sauce;
5    ArrayList toppings = new ArrayList();
6    void prepare() {
7      System.out.println("Preparing " + name);
8      System.out.println("Tossing dough...");
9      System.out.println("Adding sauce...");
10     System.out.println("Adding toppings: ");
11     for (int i = 0; i < toppings.size(); i++) {
12       System.out.println(" " + toppings.get(i));
13     }
14   }
```

# 采用工厂方法的新设计

Pizza

```
15    void bake() {
16      System.out.println("Bake for 25 minutes at 350");
17    }
18
19    void cut() {
20      System.out.println("Cutting the pizza into diagonal slices")
21    }
22
23    void box() {
24      System.out.println("Place pizza in official PizzaStore box")
25    }
26
27    public String getName() { return name; }
28  }
```

# 采用工厂方法的新设计

定义具体的 Pizza 子类

```
1  public class LAStyleCheesePizza extends Pizza {
2    public LAStyleCheesePizza() {
3      name = "LA Style Deep Dish Cheese Pizza";
4      dough = "Extra Thick Crust Dough";
5      sauce = "Plum Tomato Sauce";
6      toppings.add("Shredded Mozzarella Cheese");
7    }
8
9    void cut() {
10     System.out.println("Cutting the pizza into
11       square slices");
12   }
13 }
```

# 采用工厂方法的新设计

测试

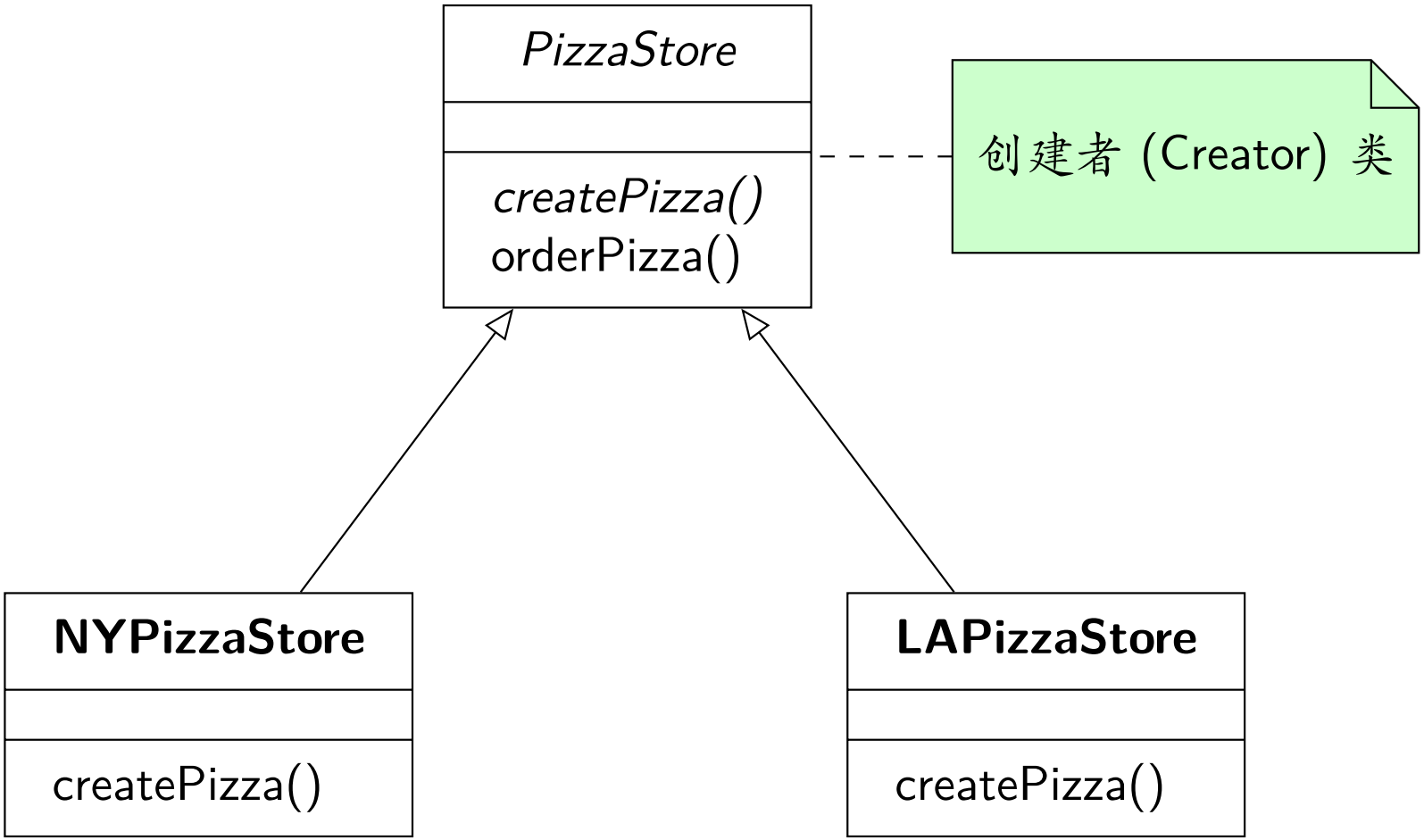
```
1  public class PizzaTestDrive {
2    public static void main(String[] args) {
3      PizzaStore nyStore = new NYPizzaStore();
4
5      Pizza pizza = nyStore.orderPizza("cheese");
6      System.out.println("Ethan ordered a " +
7        pizza.getName() + "\n");
8    }
9  }
```

# 采用工厂方法的新设计

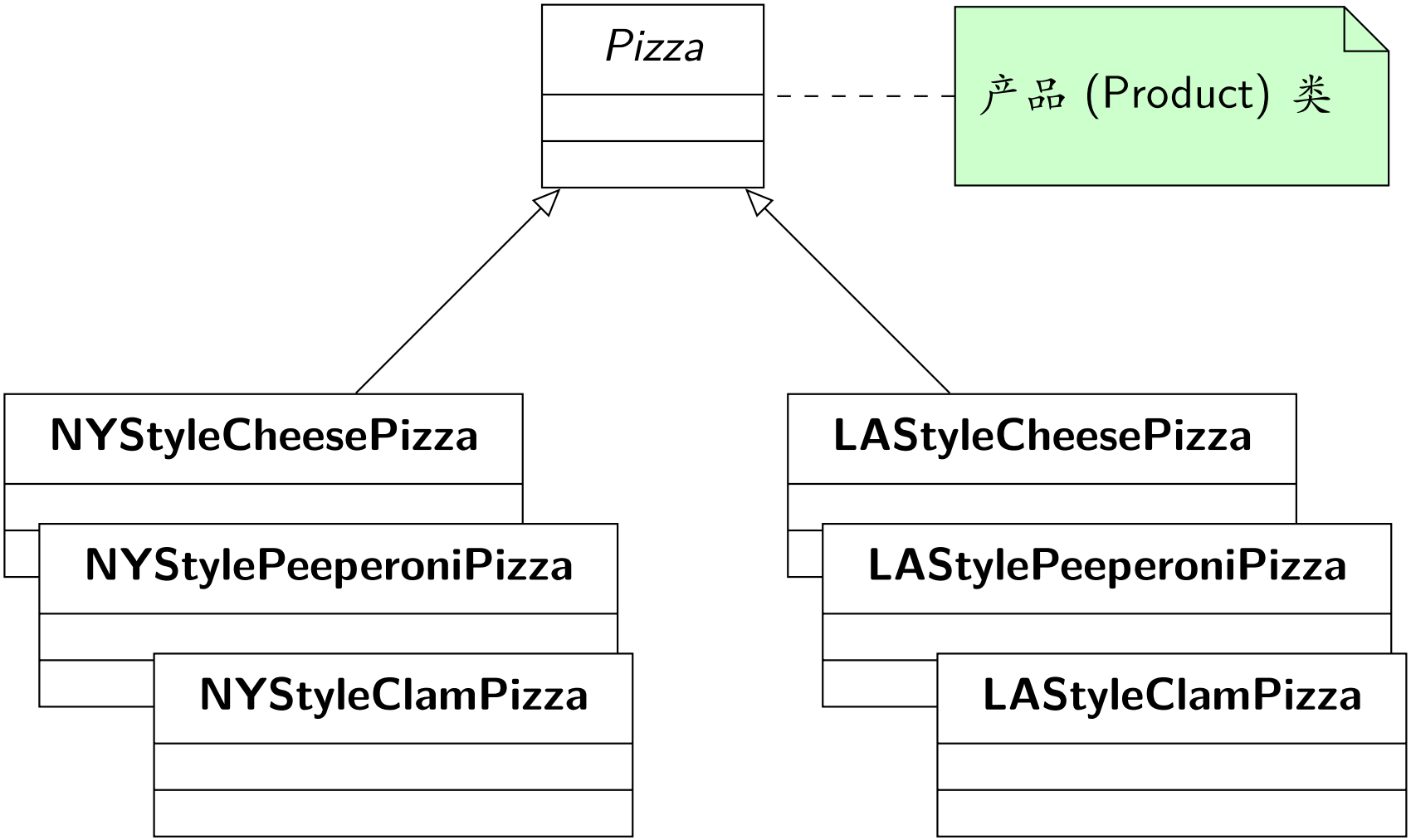
输出结果

```
%java PizzaTestDrive
Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
    Grated Regiano 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
%
```

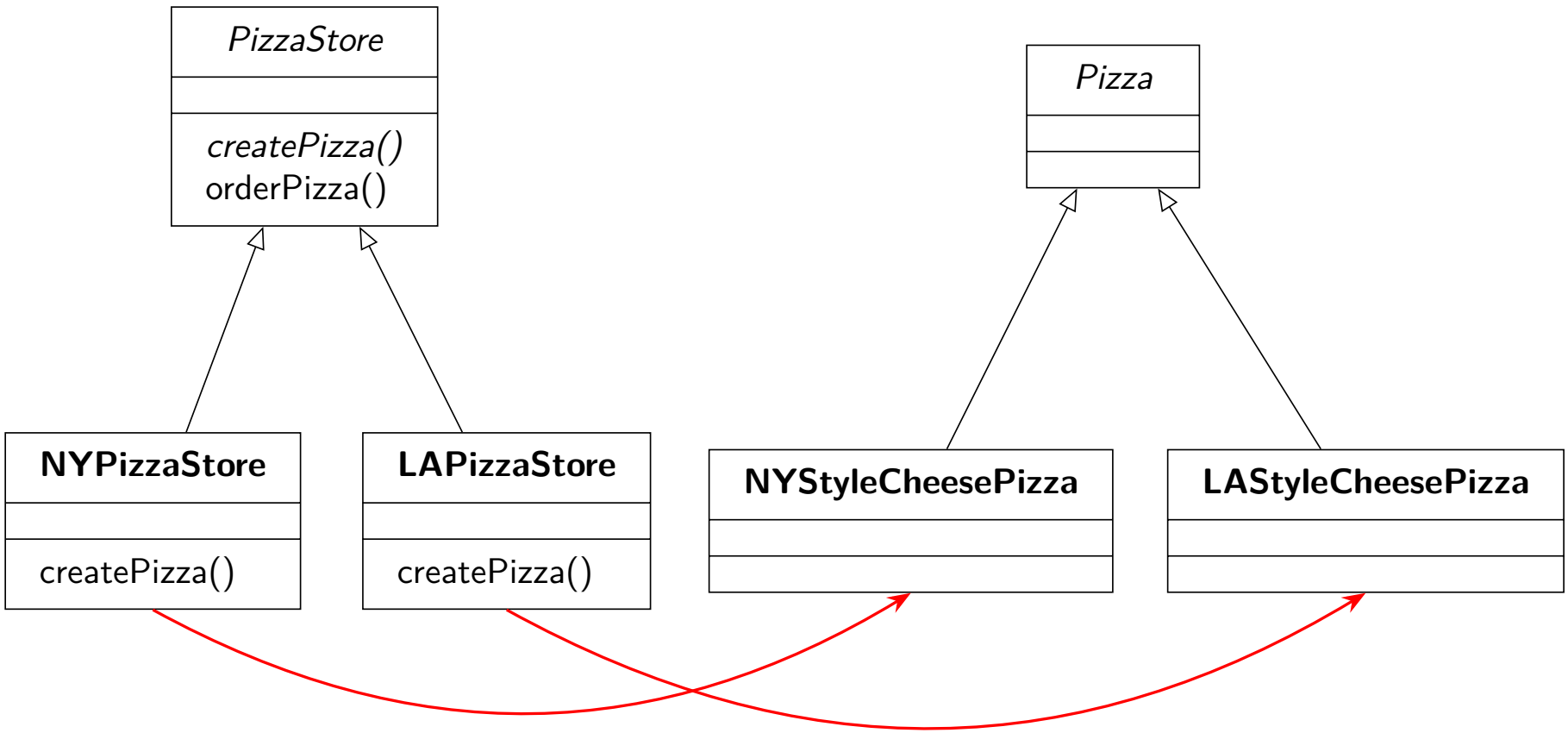
# 认识工厂方法模式 (Factory Method Pattern)



# 认识工厂方法模式 (Factory Method Pattern)



# 认识工厂方法模式 (Factory Method Pattern)



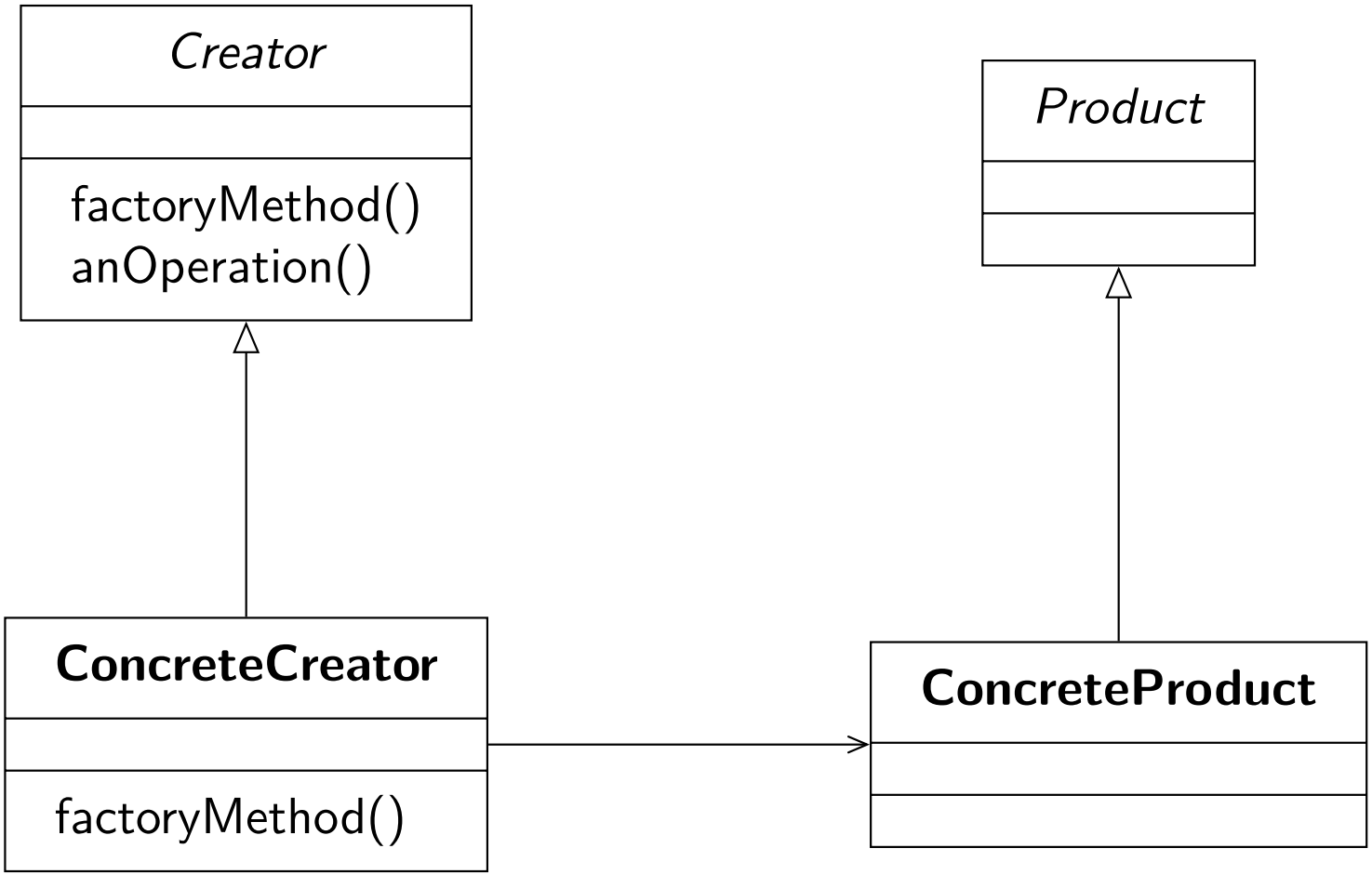
# 定义工厂方法模式 (Factory Method Pattern)

## 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)



# 关于工厂方法模式的讨论

Q: 当只有一个 ConcreteFactory 的时候，还有必要采用工厂方法模式吗？

# 关于工厂方法模式的讨论

Q: 当只有一个 ConcreteFactory 的时候，还有必要采用工厂方法模式吗？

Q: 工厂方法和创建者是否总是抽象的？

# 关于工厂方法模式的讨论

Q: 当只有一个 ConcreteFactory 的时候，还有必要采用工厂方法模式吗？

Q: 工厂方法和创建者是否总是抽象的？

Q: 工厂方法的字符串参数总是必须的吗？传错了怎么办？

# 关于工厂方法模式的讨论

Q: 当只有一个 ConcreteFactory 的时候，还有必要采用工厂方法模式吗？

Q: 工厂方法和创建者是否总是抽象的？

Q: 工厂方法的字符串参数总是必须的吗？传错了怎么办？

Q: 简单工厂和创建者这两种方法的区别是什么？

# 关于工厂方法模式的讨论

Q: 当只有一个 ConcreteFactory 的时候，还有必要采用工厂方法模式吗？

Q: 工厂方法和创建者是否总是抽象的？

Q: 工厂方法的字符串参数总是必须的吗？传错了怎么办？

Q: 简单工厂和创建者这两种方法的区别是什么？

Q: 所谓的“工厂”究竟能带来什么好处？

# 关于工厂方法模式的讨论

Q: 当只有一个 ConcreteFactory 的时候，还有必要采用工厂方法模式吗？

Q: 工厂方法和创建者是否总是抽象的？

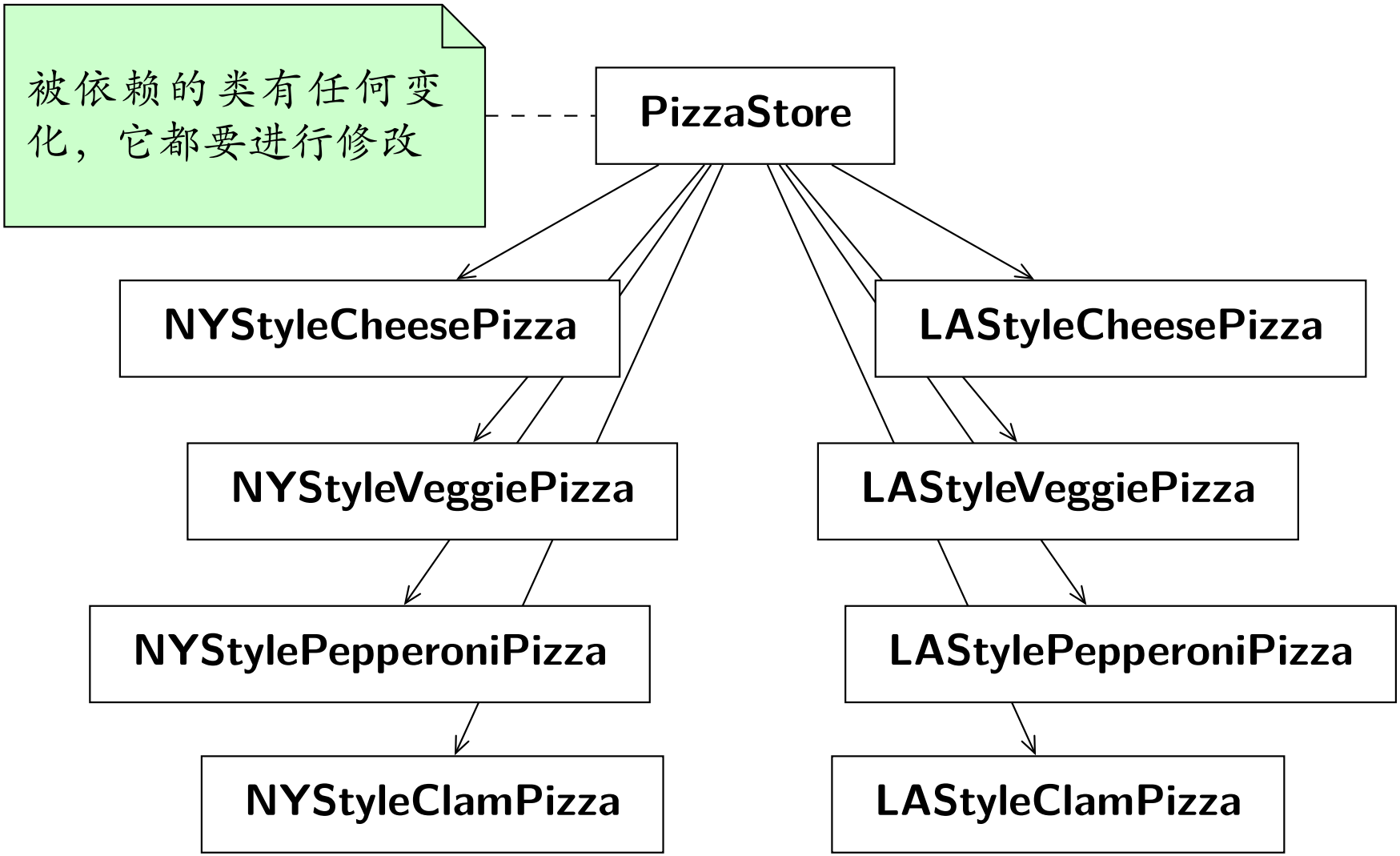
Q: 工厂方法的字符串参数总是必须的吗？传错了怎么办？

Q: 简单工厂和创建者这两种方法的区别是什么？

Q: 所谓的“工厂”究竟能带来什么好处？

Q: 为什么“工厂”方法里仍然使用了具体化的类来实例化对象？

# 最开始版本的 PizzaStore 中的对象依赖



# 依赖反转原则

应该尽量减少对具体类的依赖:

**设计原则: 依赖反转 (Dependency Inversion Principle)**

依赖抽象，不要依赖具体类

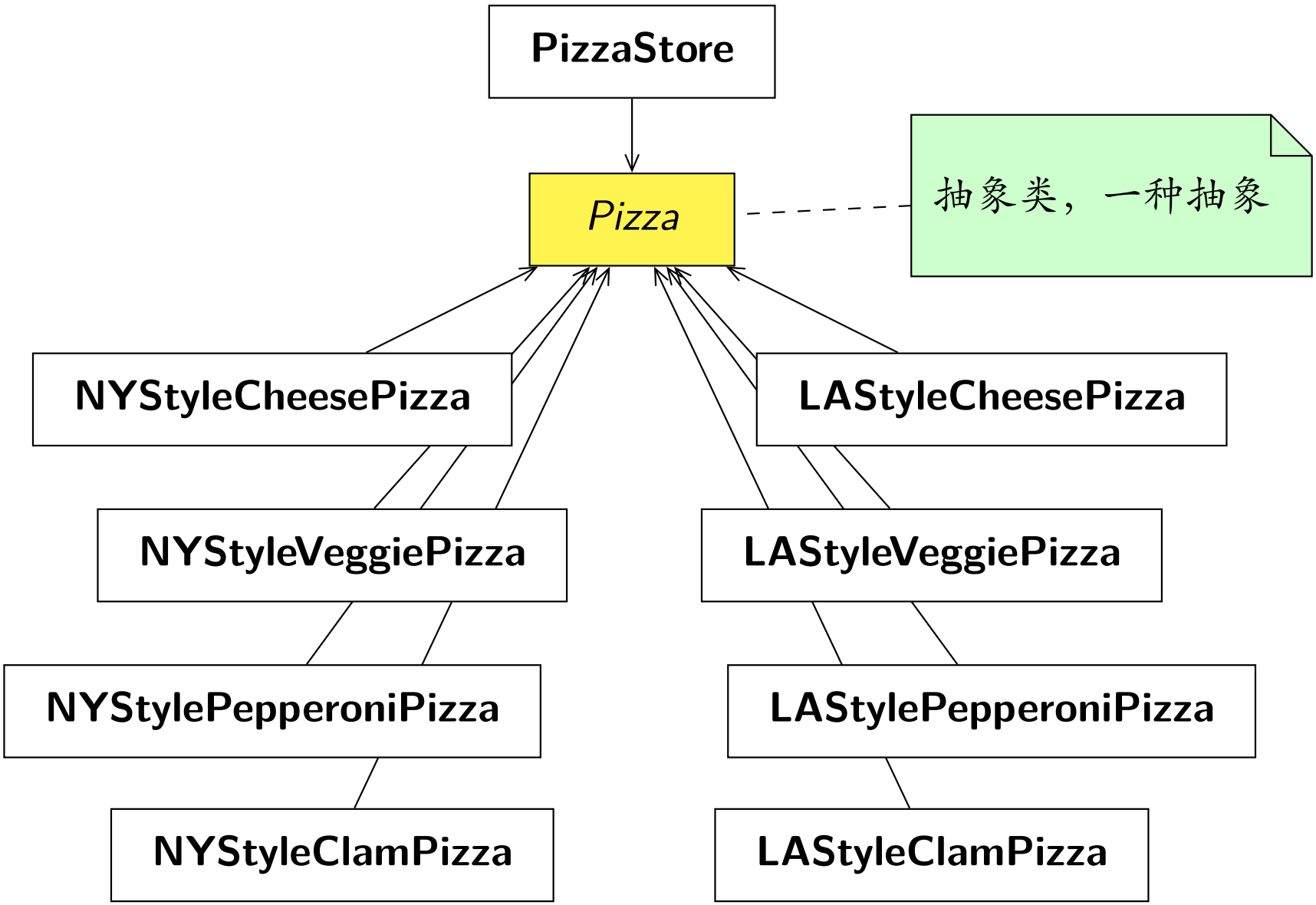
比较:

对接口编程

启发:

高层对象不应该依赖低层对象，双方都应依赖 抽象

# 应用依赖反转原则


PizzaStore
Pizza
抽象类，一种抽象
NYStyleCheesePizza
LAStyleCheesePizza
NYStyleVeggiePizza
LAStyleVeggiePizza
NYStylePepperoniPizza
LAStylePepperoniPizza
NYStyleClamPizza
LAStyleClamPizza

# 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？

# 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

# 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

# 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

# 理解“反转”：以 PizzaStore 为例

架构师 对。*Pizza* 就是抽象。现在可以重新设计了

# 理解“反转”：以 PizzaStore 为例

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

# 理解“反转”：以 PizzaStore 为例

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

# 帮助遵循依赖反转原则

- 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

完全遵循这三条是很困难的，但应有意识尽量遵循

# 内容提要

# PizzaStore 新的需求

问题:

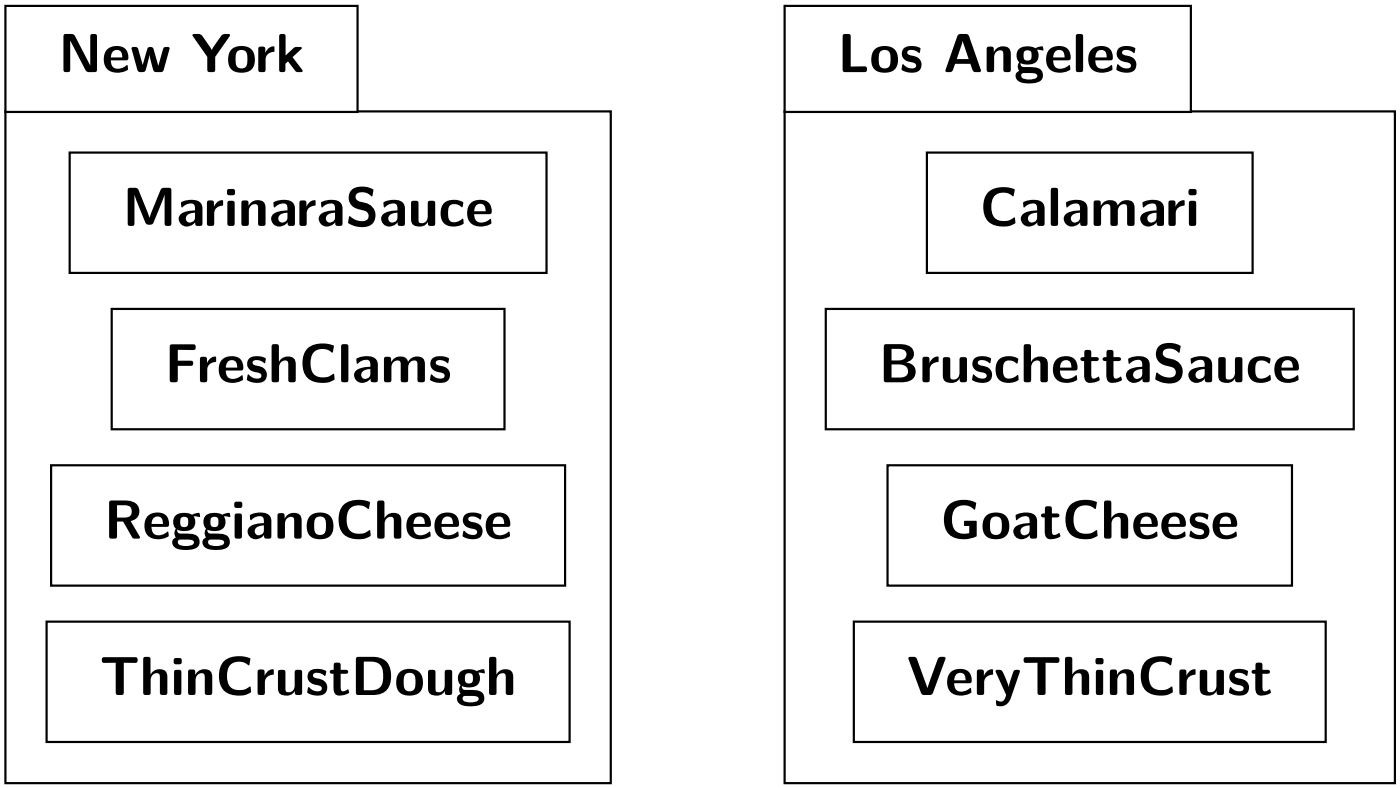
PizzaStore 获得成功的一个重要原因是坚持使用新鲜、优质的原料。但某些加盟店为了降低成本，偷偷使用低质廉价的原料，减少馅的用量。长此以往必将对品牌造成伤害。

需求:

保证每家店用同样质量的原料。可以设计一个工厂生产原料，然后发送给各店

另一个问题: 不同店的相同名字的原料可能是两种东西，例如纽约和洛杉矶的红酱料，所以需要配备几组不同的原料

# 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

# 构建原料工厂

定义原料工厂接口

```
1  public interface PizzaIngredientFactory {
2    public Dough createDough();
3    public Sauce createSauce();
4    public Cheese createCheese();
5    public Veggies[] createVeggies();
6    public Pepperoni createPepperoni();
7    public Clams createClam();
8  }
```

# 构建原料工厂

接下来需要

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

# 构建原料工厂

纽约原料工厂部分代码

```
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();
  }
```

# 构建原料工厂

新的 Pizza 类部分代码

```
1  public abstract class Pizza {
2    String name;
3    Dough dough;
4    Sauce sauce;
5    Veggies veggies[];
6    Cheese cheese;
7    Pepperoni pepperoni;
8    Clams clam;
9
10   //将准备原料的 prepare 方法声明为抽象的
11   abstract void prepare();
```

# 构建原料工厂

新的 Pizza 类部分代码，其他方法不变

```
12    void bake() {
13      System.out.println("Bake for 25 minutes at 350");
14    }
15    void cut() {
16      System.out.println("Cutting the pizza into
17        diagonal slices");
18    }
19    void box() {
20      System.out.println("Place pizza in official
21        PizzaStore box");
22    }
23    // other methods such as setName, getName, toString
24  }
```

# 构建原料工厂

不同风味的 Pizza 做法一样，原料不同

```
1  public class CheesePizza extends Pizza {
2    PizzaIngredientFactory ingf;
3    public CheesePizza(PizzaIngredientFactory ingf) {
4      this.ingf = inf;
5    }
6    void prepare() {
7      System.out.println("Preparing " + name);
8      dough = ingf.createDough();
9      sauce = ingf.createSauce();
10     cheese = ingf.createCheese();
11   }
12 }
```

# 构建原料工厂

多一种原料的另一种 Pizza

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

# 构建原料工厂

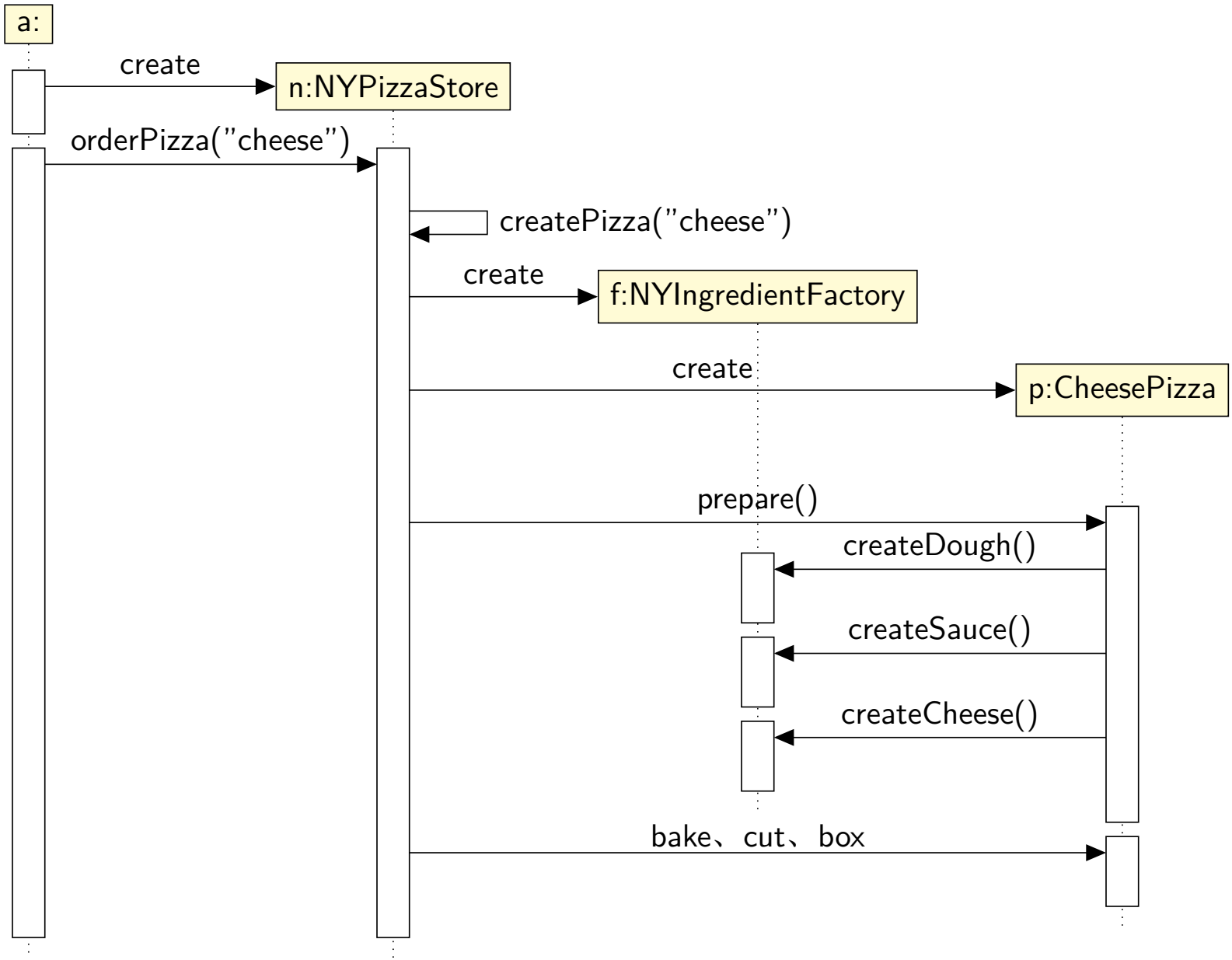
使用了本地原料工厂的 PizzaStore

```
1  public class NYPizzaStore extends PizzaStore {
2    protected Pizza createPizza(String item) {
3      Pizza pizza = null;
4      PizzaIngredientFactory ingf =
5        new NYPizzaIngredientFactory();
6      if (item.equals("cheese")) {
7        pizza = new CheesePizza(ingredientFactory);
8        pizza.setName("New York Style Cheese Pizza");
9      } else if (item.equals("veggie")) {
10       // 制造各种本地风味的 Pizza
11     }
12 }
```

# 这些代码有何效果?

- 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

# 订购一个 Pizza 的顺序图



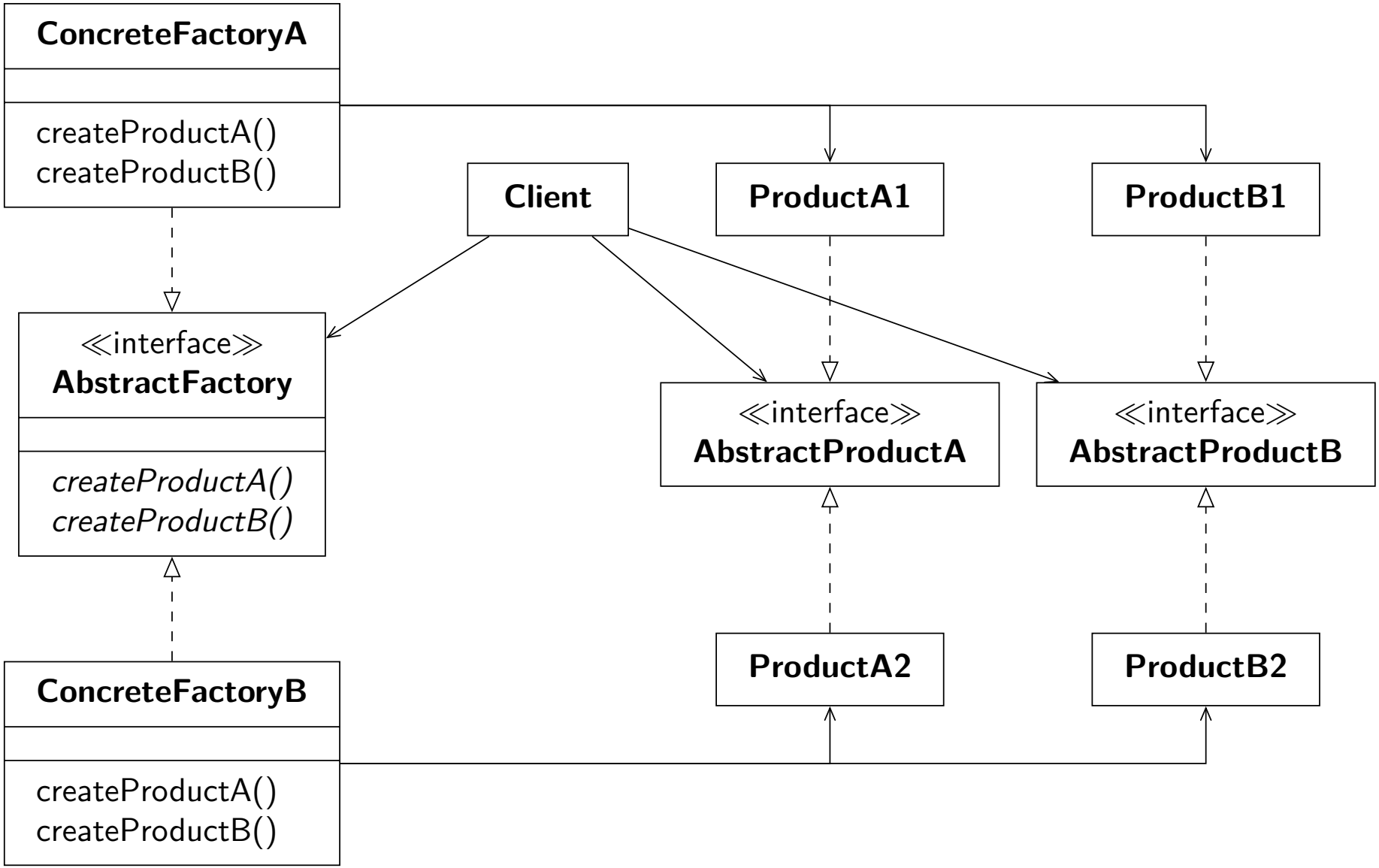
# 定义抽象工厂模式

## 抽象工厂模式

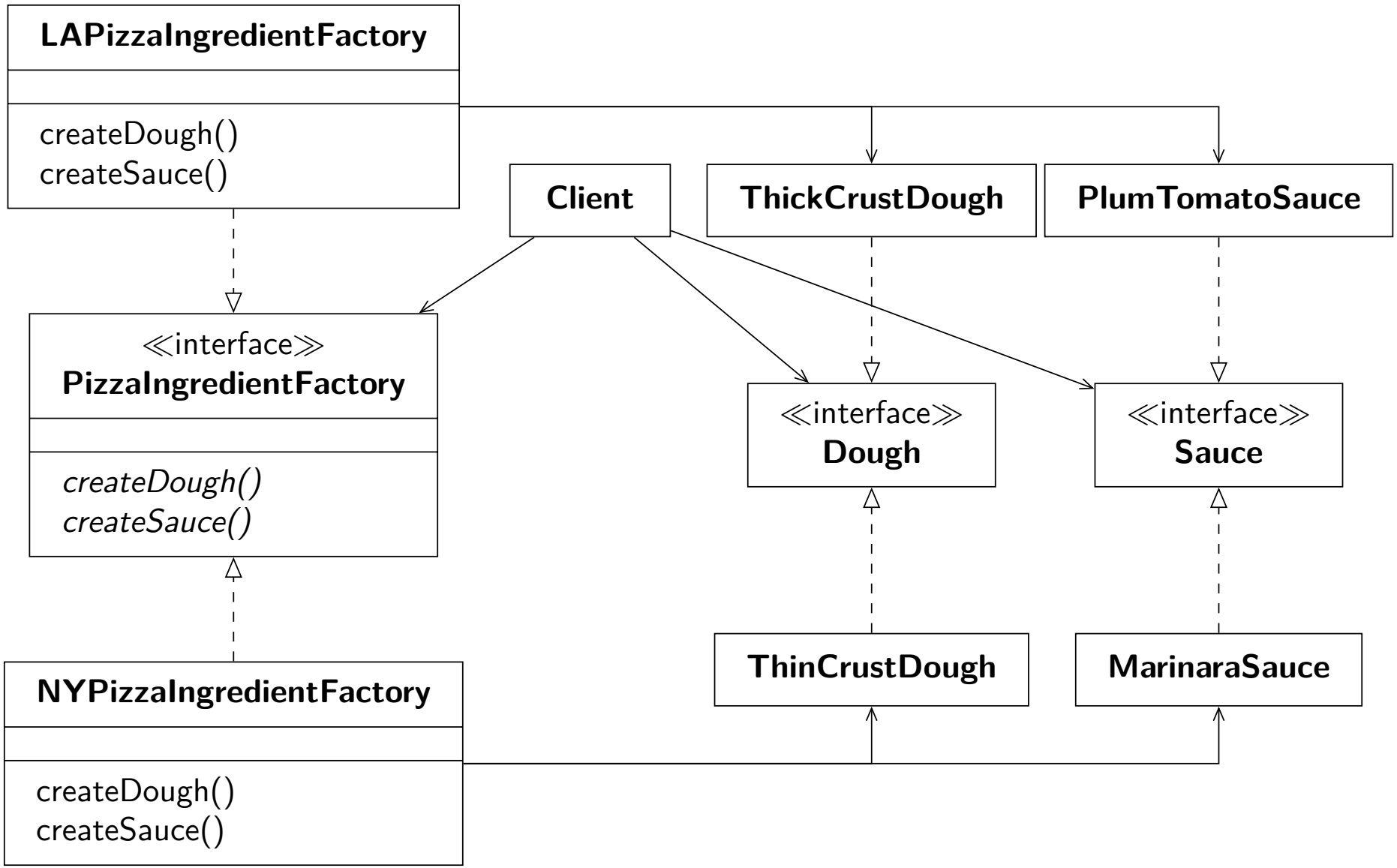
抽象工厂模式提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类

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


ConcreteFactoryA
createProductA()
createProductB()
Client
ProductA1
ProductB1
≪interface≫
AbstractFactory
createProductA()
createProductB()
≪interface≫
AbstractProductA
≪interface≫
AbstractProductB
ProductA2
ProductB2
ConcreteFactoryB
createProductA()
createProductB()

# 从 PizzaStore 的观点看抽象工厂



# 关于工厂模式的小结

- 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

# 关于设计原则的小结

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