工厂模式

1.在介绍工厂方法模式之前，先说一下简单工厂方法。

简单工厂方法的定义：:提供创建对象的接口。通过专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

2.假设市面上有很多种类的披萨，现在我想根据需要订购披萨，于是我在某个类里面写下了如下的代码，用来根据类型而生成不同的pizza对象：

```
Pizza pizza = null;

if(type.equals("Cheese")){
pizza = new ChicagoStyleCheesePizza();
}else if(type.equals("Clam")){
pizza = new ChicagoStyleClamPizza();
}else if(type.equals("Pepperoni")){
pizza = new ChicagoStylePepperoniPizza();
}else if(type.equals("Veggie")){
pizza = new ChicagoStyleVeggiePizza();
}
pizza.prepare();
pizza.cut();
pizza.bake();
pizza.box();
```

看上去，这段代码是没有问题的，但是假如现在我还有其他的十个类，也需要根据它们的需要来进行产生Pizza对象，于是我就需要将这段代码，复制到另外的十个类里面，造成了大量的代码重复，而且，假如某天，我决定不再提供Cheese类型和Clam类型的Pizza对象，那么，我将需要删除掉那两行生产这两个类型的对象的代码，这样的删除工作，我需要在十个类里面进行，大大地增加了维护的工作量。

3.可以从上面的代码观察到，不变的部分是下面的准备，切，烘焙，打包的动作，变化的部分则是上面运行时产生披萨对象，根据编码原则：封装变化。我们可以把产生对象的部分封装起来放在一个类里面，这个类可以成为一个简单的工厂，专门用来产生对象：

```
public class SimpleFactory{

public static Pizza createPizza(String type){

if(type.equals("Cheese")){
return new NYStyleCheesePizza();
}else if(type.equals("Clam")){
return new NYStyleClamPizza();
}else if(type.equals("Pepperoni")){
return new NYStylePepperoniPizza();
}else if(type.equals("Veggie")){
```

```
 return new NYStyleVeggiePizza();
 }

 return null;

 }

}
```

使用简单工厂来产生对象，则我们只需对简单工厂类进行维护就好了，减少了工作量，也减少了代码重复，降低了出错的几率，把变化封装起来也符合了面向对象的编程原则。

4.简单工厂方法只是一种编程的好习惯，不能称之为一种模式，简单工厂方法不是俗称的工厂模式（Factory Patten），接下来就介绍真正的工厂模式。

5.工厂模式，也就是工厂方法模式，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，工厂方法让类把实例化推迟到子类。

6.刚才上面的简单工厂的例子，里面的披萨都是纽约风格的，现在假如我们要芝加哥风格的，加州风格的披萨的话，如果我们还继续使用简单工厂，则需要在工厂里写下一大堆的判断流，这样子不利于披萨的分类，也让这个简单工厂和很多的披萨类型耦合了，我们还有更好的办法来解决这个问题，就是使用工厂方法模式。首先有一个用于创建披萨对象的PizzaStore接口，但是创建的实际动作，是在它的子类ChicagoStyleStore和NYStyleStore来创建，在这里，这两个子类承担了工厂的角色。

7.代码如下：

首先是PizzaStore接口：

```
public abstract class PizzaStore{

 abstract Pizza createPizza(String type);

 public Pizza orderPizza(String type){

 Pizza pizza = createPizza(type);

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

 return pizza;

 }
```

```
}
```

然后是两个子类，用来真正地产生对象：

ChicagoPizzaStore：

```
public class ChicagoPizzaStore extends PizzaStore{

 @Override
 Pizza createPizza(String type){

 Pizza pizza = null;

 if(type.equals("Cheese")){
 pizza = new ChicagoStyleCheesePizza();
 }else if(type.equals("Clam")){
 pizza = new ChicagoStyleClamPizza();
 }else if(type.equals("Pepperoni")){
 pizza = new ChicagoStylePepperoniPizza();
 }else if(type.equals("Veggie")){
 pizza = new ChicagoStyleVeggiePizza();
 }

 return pizza;

 }

}
```

NYPizzaStore：

```
public class NYPizzaStore extends PizzaStore{

 @Override
 Pizza createPizza(String type){

 Pizza pizza = null;

 if(type.equals("Cheese")){
 pizza = new NYStyleCheesePizza();
 }else if(type.equals("Clam")){
 pizza = new NYStyleClamPizza();
 }else if(type.equals("Pepperoni")){
 pizza = new NYStylePepperoniPizza();
 }else if(type.equals("Veggie")){
 pizza = new NYStyleVeggiePizza();
 }

 return pizza;
```

```
    }

}
```

PizzaStore不需要知道创建了什么风格什么类型的pizza，它只知道有一个pizza，对他们做准备，切，烘焙，打包的动作就可以了，这个PizzaStore对象不直接依赖于底层的更种类型的具体pizza对象，而是直接和Pizza接口打交道就行了。

在这里，我们高度抽象了两个类，一个是PizzaStore，一个是Pizza，我们只针对这两个接口编程，而不是针对具体的实现编程。

假如有某些创建Pizza对象的行为需要修改，则只在子类Store里面修改就可以了，不用修改PizzaStore的代码，如果还有其他类型的子类Store，则通过继承PizzaStore扩展就行，这就体现了对修改关闭，对扩展开放的面向对象编程原则。

让PizzaStore和具体的Pizza对象通过只通过Pizza接口来交互，体现了依赖倒置原则。PizzaStore向下依赖Pizza接口，具体的Pizza对象向上依赖Pizza接口。

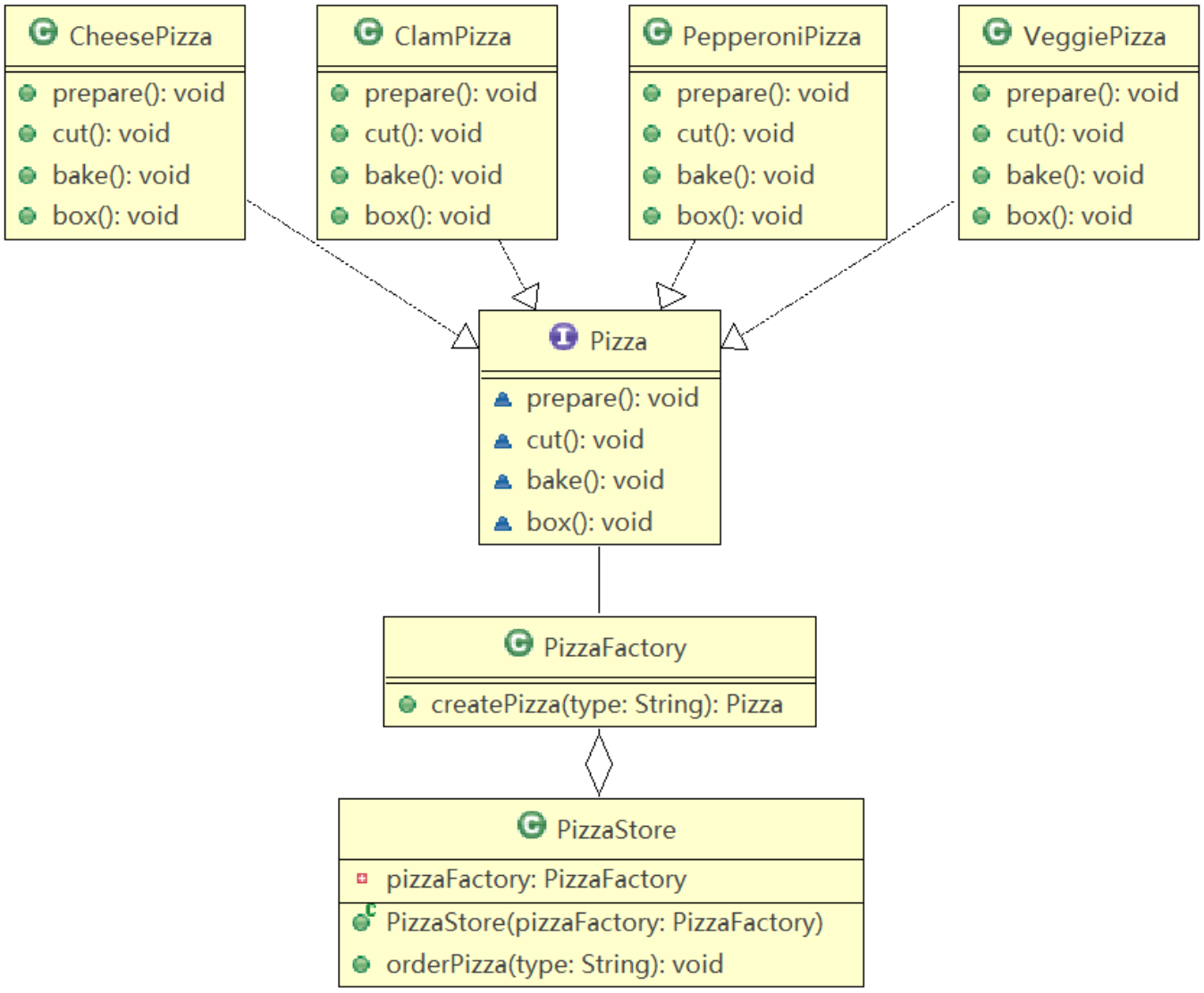
8.到目前为止，我们看到了简单工厂方法和工厂方法模式，这两个的好处都是比较相似的，将变化的部分封装起来，减少代码重复，提高可维护性，针对接口编程，而不是具体类。使程序更具弹性。

但是这两个方法还是有区别的，体现在如下：

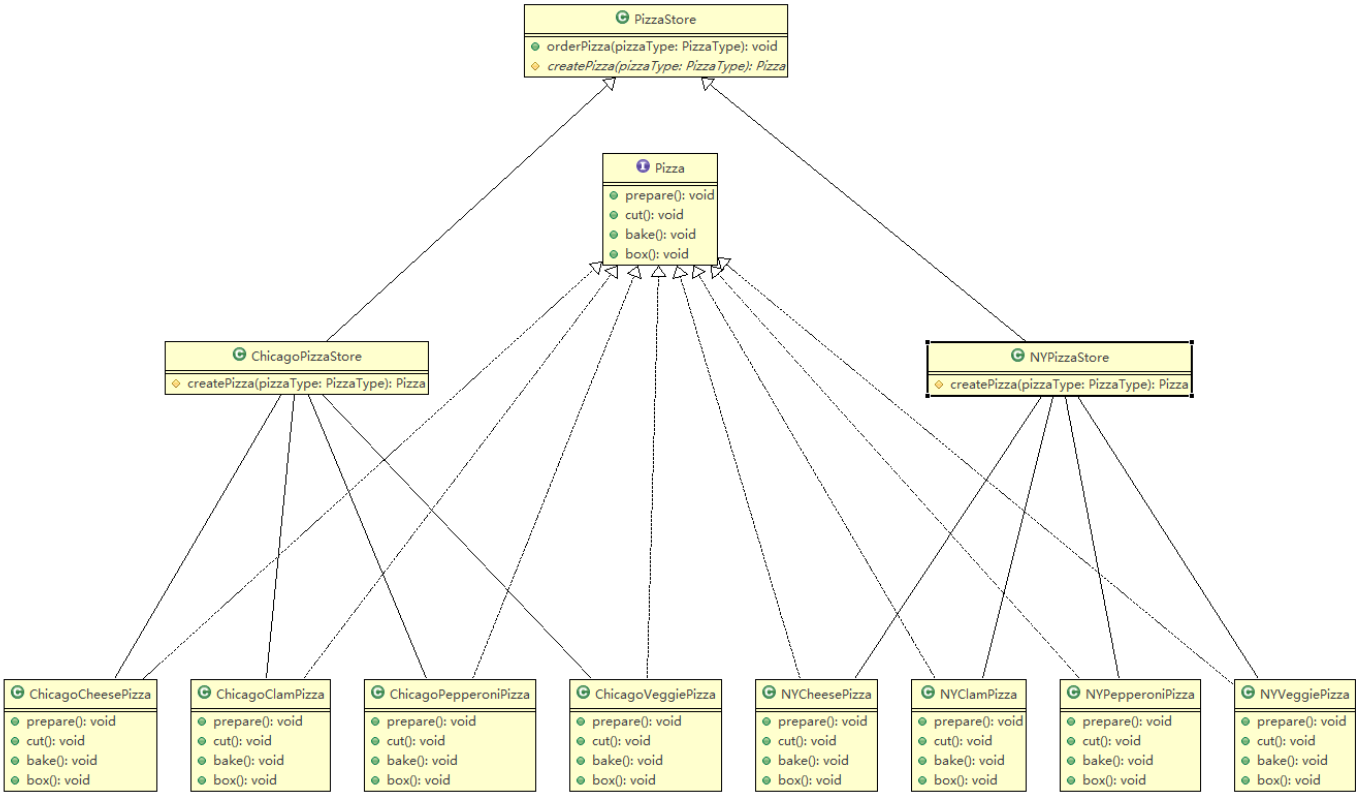
- 简单工厂方法让事情在一个地方就完成了，而工厂方法模式则创建了一个框架，让子类决定生成的具体对象。
- 简单工厂方法不具有工厂模式的弹性，不能变更正在创建的产品。

9.UML图

A.简单工厂方法的UML图：

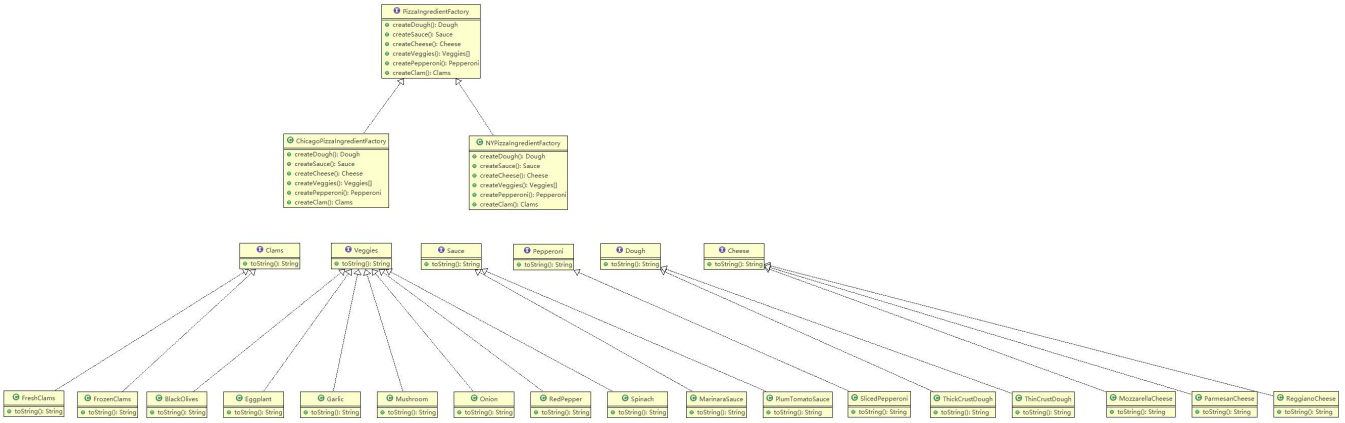


B.工厂方法模式的UML图：

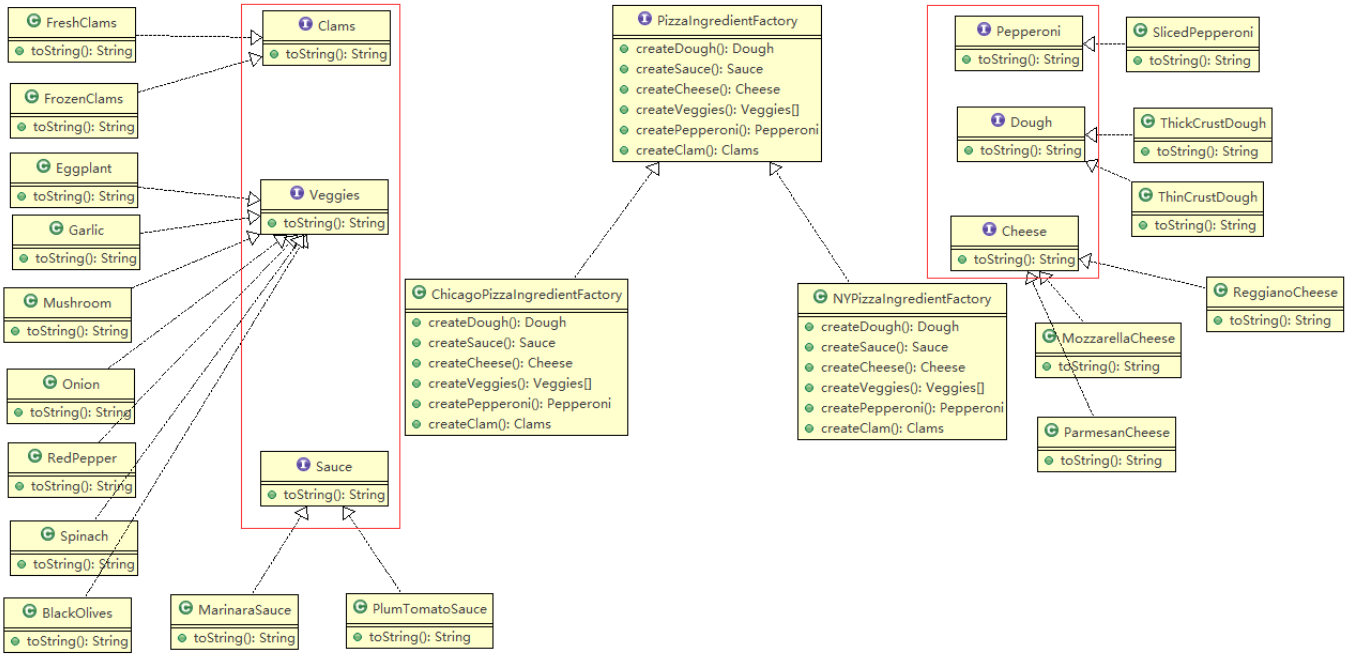


10.抽象工厂模式：提供一个接口，用于创建相关或者依赖对象的家族，而不需要明确指定具体类。

使用head first里面关于披萨原料的例子，先看一下UML图：



这是一个整体的树图，便于从整体到感性认识抽象工厂模式，但是图片太小看不清楚，看一下下面的图



假设，我们要做一个汉堡，现在汉堡的外层已经做好了，需要再加上一些原料来使最终的汉堡完成。

在这里，这些原料分别有，Clams，Veggies，Sauce，Pepperoni，Dough，Cheese六样，这六种对象属于一个家族---原料家族。

如果将这六种原料对象具体化，则有17种之多，也就是说，有十七个具体的类。

假如不使用模式来设计，而纯粹在PizzaStore中判断再加入这些具体的原料，从原料接口层面来说，至少有六个，具体到具体类来说，有十七个，也就是说，PizzaStore需要和十七个对象耦合。

由于，这六种原料都属于原料家族的，那么，我们可以使用抽象工厂模式来设计，则我们只提供一个PizzaIngredientFactory接口给客户，让客户代码只和这个接口耦合就可以了，这样一来，客户就从具体的产品中被解耦了。

11.抽象工厂模式和工厂方法模式的联系和区别：

- 这些工厂模式，包括简单工厂方法，都是对对象创建的过程进行封装。能让使用工程的客户从具体的对象中解耦出来，降低对特定实现的依赖，提高代码的复用率，增加程序的可维护性和扩展性。
- 工厂方法模式适合用于产生一种对象，比如上面的例子，说到底，只是分区域地产生了Pizza对象。
- 抽象工厂模式适用于产生一组相关的对象，就是一个家族，比如上面的例子，定义了一个原料工厂，里面是一个原料家族，包括了一组对象，分别是，Clams，Veggies，Sauce，Pepperoni，Dough，Cheese六样。
- 工厂方法使用的是继承的方式，让子类实现具体的创建对象的过程。
- 抽象工厂模式更多的是使用了对象组合的方式，例如上面，一个ChicagoPizzaIngredientFactory组合了六种对象。

12.使用反射机制配合工厂方法

在以上列举的工厂方法中，使用了一些工厂类来负责类的创建，这避免了在其他类想使用某一对象的时候，重新编码对象的生成，减少了代码的重复。

但是，具体到工厂类里面，对象的生成，还是使用了new的方式来创建，这意味着，假如我有100个对象需要创建，那么我就至少需要一百个new语句。这种情况，理论上是会产生类爆炸的。

我们可以利用Java的反射机制，使同一接口的对象，只需要通过同一行代码，就可以让对象被创建。

下面举例，让Java的反射机制配合简单工厂使用

首先，我有一个通用的接口Fruit，里面有一个eat()方法

```
package com.designpattern.factory;
public interface Fruit {
    public void eat();

}
```

分别有苹果、橙子、香蕉三种水果实现了这个接口：

```
package com.designpattern.factory;
public class Apple implements Fruit {
    @Override
    public void eat() {
        // TODO Auto-generated method stub
        System.out.println("I am eating a apple!");
    }
}
```

```
package com.designpattern.factory;
public class Orange implements Fruit {
    @Override
    public void eat() {
        // TODO Auto-generated method stub
```

```
        System.out.println("I am eating a orange!");
    }
}
```

```
package com.designpattern.factory;
public class Banana implements Fruit {
    @Override
    public void eat() {
        // TODO Auto-generated method stub
        System.out.println("I am eating a banana!");
    }
}
```

接下来，创建一个水果工厂，里面实现了常见的创建方法和使用反射机制的创建方法。

```
package com.designpattern.factory;
public class SimpleFruitFactory {
//使用了反射机制
 public static Fruit getInstance(String className) {

    Fruit f = null;

     try {
         //将Fruit对象的创建集中到这一行代码
         f = (Fruit) Class.forName(className).newInstance();
     } catch (InstantiationException e) {
         // TODO Auto- generated catch block
         e.printStackTrace();
     } catch (IllegalAccessException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
     } catch (ClassNotFoundException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
     }

     return f;
 }

     //普通的对象创建方式，可能造成类爆炸
     public static Fruit getInstanceNormal(String className) {
         Fruit f = null;

         //假如Fruit接口有一百个实现类，则这里需要写一百个If语句
         if("apple".equals(className)) {
             f = new Apple();
         }else if("orange".equals(className)){
             f = new Orange();
         }else if("banana".equals(className)) {
```

```
            f = new Banana();
        }

        return f;
    }

}
```

反射机制的主要优点就是他的灵活性，那么其实反射机制也有它的缺点，主要是两方面：

其一是对性能会有影响。反射机制生成对象有一个内部的程序解释过程，比直接new一个对象要慢，这也是应用的时候需要考虑的。

其二是反射机制会模糊程序内部实际会发生的事情。