### 第4章：工厂模式

案例：

|  |
| --- |
| 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();  }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;  } |

有new的地方都是针对实现编程。很显然，上面的代码并没有对修改封闭，当有新的pizza加入进来时，我们就必须要修改这部分代码。

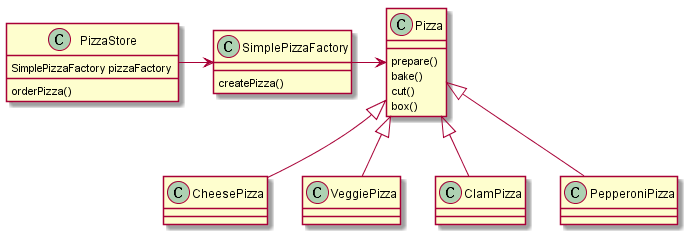
修改方法：把红色字体部分可变化的部分提取出来进行封装。

|  |
| --- |
| 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(“pepperoni”)){  pizza = new PepperoniPizza();  }else if(type.equals(“clam”)){  pizza = new ClamPizza();  }else if(type.equals(“veggie”)){  pizza = new VeggiePizza();  }  return pizza;  }  } |

上述也可以用静态工厂实现，静态工厂比较简单，缺点是不能通过继承来改变创建方法的行为，也就是只有唯一的一个创建方式。

看到这里我们可能会有疑问，修改依然存在啊，代码依然是对实现编程的，只不过是从一个类移到另一个类中罢了？

我们想象一下获取Pizza对象的可能有许多客户，按照原先的方式，我们需要在很多地方对orderPizza方法进行修改；而现在只需要在一个封装的类中修改即可，而且这个修改是对客户关闭的。（这才是封装，复用，修改关闭，扩展开放的真实意义所在）。



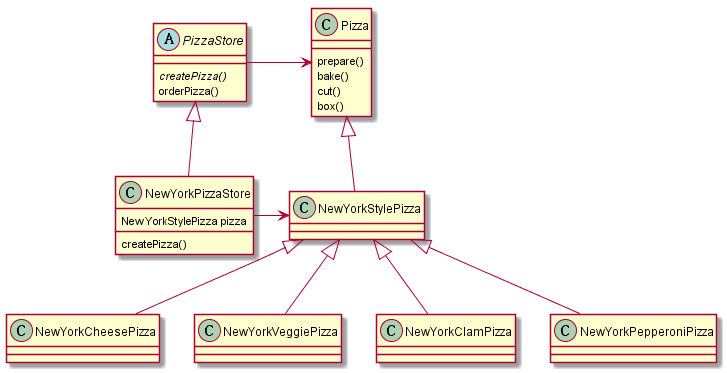
假如获取Pizza对象的客户，即PizzaStore有了有多个子类，此时我们可以制定一个PizzaStore框架：

|  |
| --- |
| 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 Pizza createPizza(String type);  } |

createPizza方法从简单工厂对象中移回到PizzaStore，现在变成了抽象的“工厂方法”。这个框架比原先的简单工厂好在，我可以在子类中有更大的延展性，创建子类需要的pizza风味，而工厂方法却只能提供适合所有的pizzaStore的pizza风味。

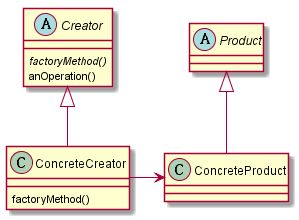
上面的框架中我们称createPizza为抽象工厂方法。工厂的责任就是创建对象。

工厂方法用例处理对象的创建，并将这样的行为封装在子类中。这样，客户程序中关于超类的代码就和子类对象创建代码解耦了。



实现见源码：src/simplefactory

**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。**



工厂方法的好处是将对象的创建factoryMethod()和对象的使用anOperation()完全分离。将实现从使用中解耦。如果增加产品或者改变产品的实现，不会影响到产品的使用。

在这里ConcreteCreator看起来的作用就像是原先的SimpleFactory(简单工厂)。简单工厂把全部的事情在一个地方都处理完了，工厂方法却是创建一个框架，让子类决定要如何实现。

|  |
| --- |
| *设计原则6——要依赖抽象，不要依赖具体类* |

这一原则有一个正式名称叫：依赖倒置原则（Dependency Inversion Principle）。

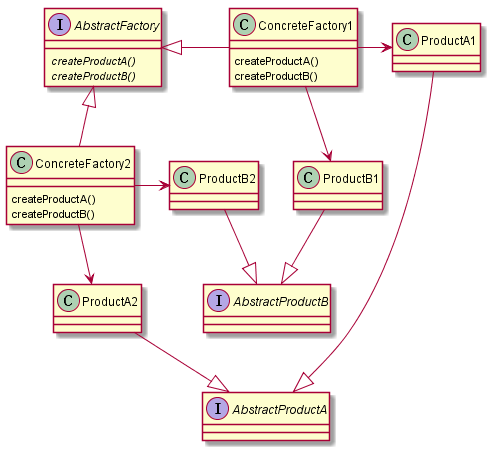
查看文章开头的那部分代码，PizzaStore明显的依赖了许多new 的具体类，此时我们应该考虑使用抽象。

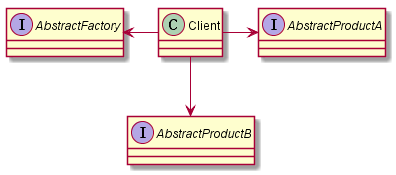
1. 我们从PizzaStore和Pizza的例子上来理解下依赖倒置。这里PizzaStore是上层组件，因为它的行为需要Pizza组件来完成，也就是PizzaStore包含Pizza，被包含的是下层组件。
2. 对于实现一个披萨店，我们一般的思维方式是先建立PizzaStore，然后再往下到具体组件，不可避免的PizzaStore会依赖具体下层组件。
3. 因此倒置我们的想法，先抽象一个Pizza类，再来设计PizzaStore。此时PizzaStore就只依赖Pizza抽象类，而不是具体类了。

避免违反依赖倒置原则的方针：

* 变量不可以持有具体类的引用。
* 不要让类派生自具体类。
* 不要覆盖基类中已实现的方法。

**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。**





从上面的图表来看，可以知道客户已经完全的从具体的产品中被解耦出来。

同时要说明一点，抽象工厂中的抽象方法往往就是以工厂方法的方式实现的。

要点：

* 所有的工厂都是用来封装对象的创建。
* 工厂方法使用继承：把对象的创建委托给子类，子类实现工厂方法来创建对象。
* 抽象工厂使用对象组合：对象的创建被实现在工厂接口所暴露出来的方法中。
* 工厂方法允许类将实例化延迟到子类进行。
* 抽象工厂创建相关的对象家族，而不需要依赖他们的具体实现。
* 依赖倒置原则知道我们尽量避免依赖具体类型，而要尽量依赖抽象。