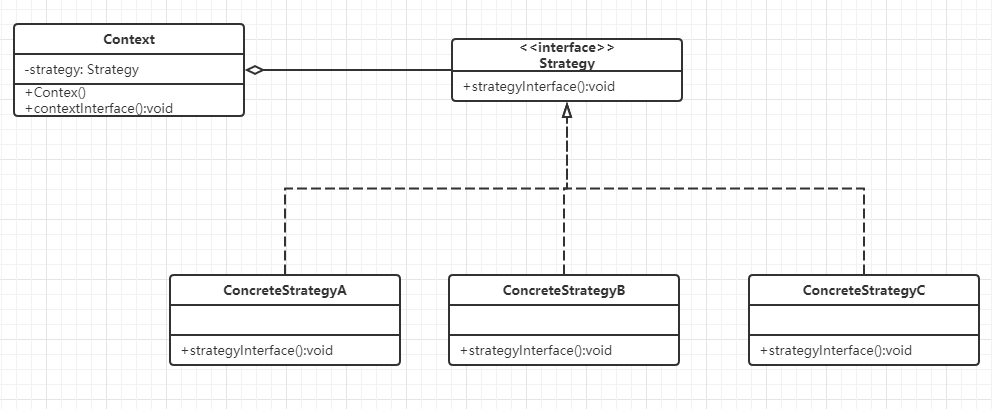
# 策略模式

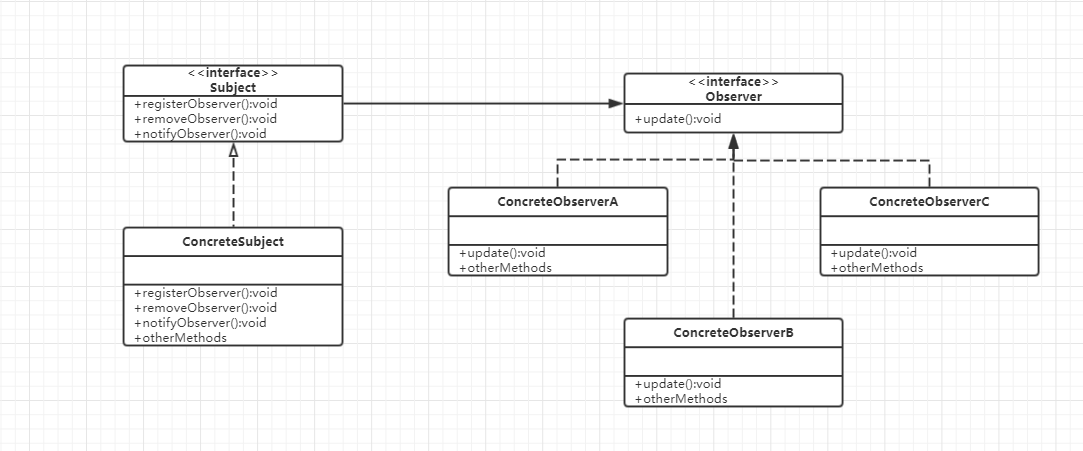
策略模式定义了算法族，分别封装起来，让它们之间可以相互替换，此模式让算法的变化独立于使用算法的客户端。



如上UML图所示，Context对象中保存着一个Strategy的引用，Strategy是一个接口，并且有很多该接口的实现类，这样Context对象里面就可以根据具体的需求来改变Strategy的引用对象，例如，现在有一款游戏，游戏人物可以手持武器，而武器是可以根据不同的场景变换的，人物升级，打怪都是可以改变当前的武器，这样武器就是一个策略接口，而具体的武器，例如手枪，长剑等都是武器接口的具体实现。人物可以根据需求来改变武器的种类，而不需要写死，依赖具体实现。

# 观察者模式

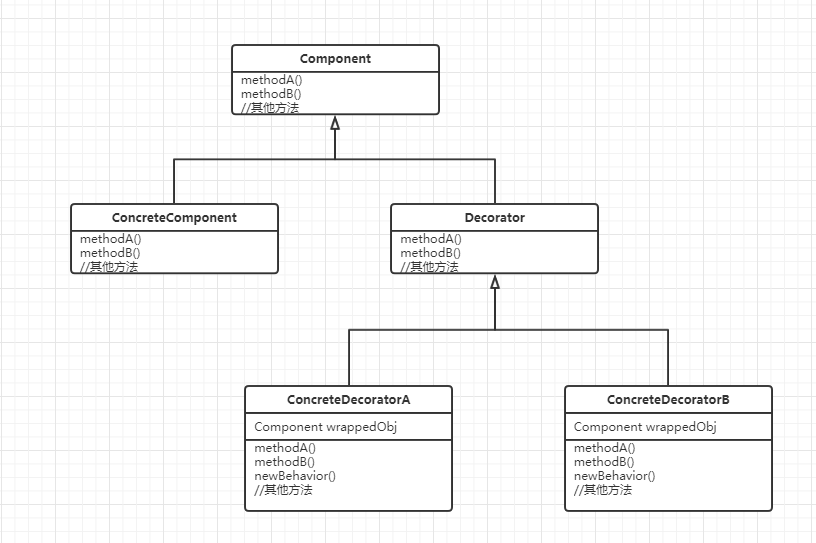
观察者模式中定义了对象之间的一对多依赖，这样一来，当一个对象状态发生变化时候，它的所有依赖者都会收到通知并自动更新



如上图的UML所示，Subject是被观察者的接口，被观察者们需要实现该接口。ConcreteSubject类里面维护了一个观察者list，有三个方法需要注意，一个是registerObserver，该方法是用来在被观察者中注册观察者的，该方法一般实在观察者中调用，当有新的观察者想要加入观察队列的时候，就需要在观察者中调用registerObserver方法，观察者也维护了一个被观察者，所以可以进行这样的操作，同理removeObserver也是这样的。而对于notifyObserver，则是在ConcreteSubject中调用的，当被观察者等待的事件触发的时候，被观察者就会调用该方法，去通知各个观察者进行更新。更新有两种方法，一种是推送，就是把所有的更新内容都推送个观察者，还有一种是拉，就是在观察者中的update方法里面，直接获取自己想要的数据，避免了不必要的数据传过来。

# 装饰者模式

装饰模式是指在不必要改变原类文件和使用继承的情况下，动态的扩展一个对象的一个功能。通过创建一个包装对象来包裹真是对象。

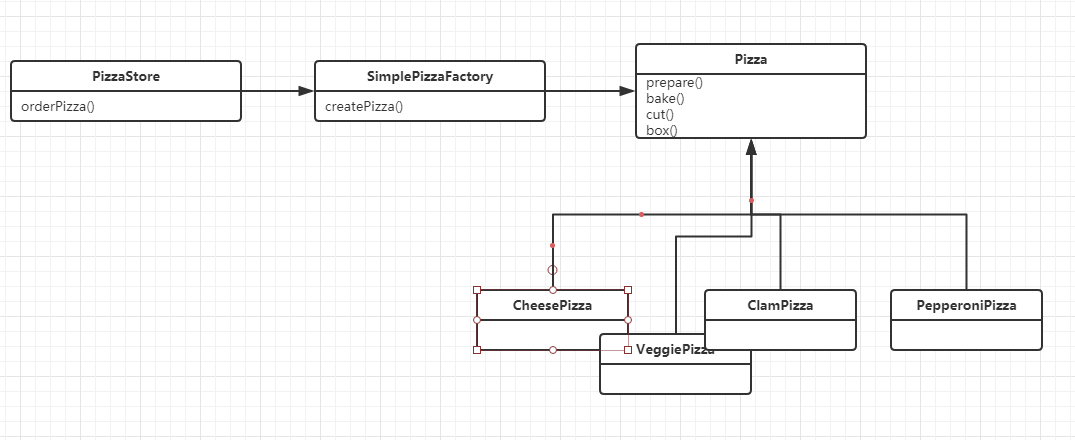


如上UML图所示，Component一般来说是一个抽象的类，装饰类以及被装饰类都需要继承自该Component。这样做的目的是为了达到类型匹配，这样装饰者就能够替换被装饰者，装饰者内部是维护了一个被装饰者对象，装饰者可以在主要方法的前后进行一些操作，从而达到新的目的。Java.io类中就使用了很多装饰模式。

# 工厂模式

## 简单工厂

简单工厂其实不是一种设计模式，反而像是一种编程习惯。以pizza店做披萨为例：

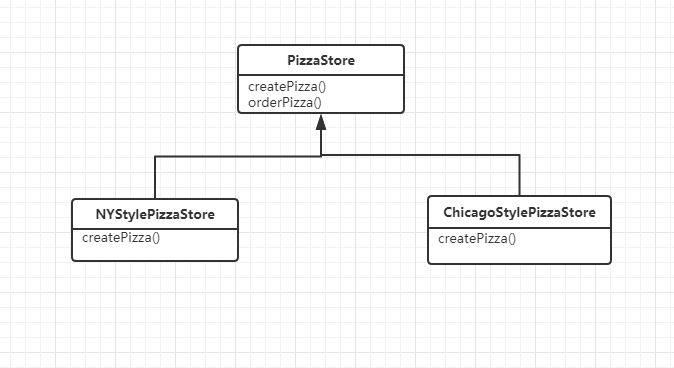


如上图的UML所示，PizzaStore中的方法orderPizza()通过simplePizzaFactory来获取具体的pizza，如果没有SimplePizzaFactory对象，那么orderPizza方法就需要自己创建pizza对象来进行返回，这样如果pizza类型发生了改变，例如需要添加一些新的类型或者需要去除一些不流行的类型，那么就需要修改orderPizza()当中的代码，该方法就依赖具体实现了。

而存在SimplePizzaFactory后，就不存在上述的问题，当pizza类型需要改动的时候，只需要更改SimplePizzaFactory对象内容，而orderPizza()方法中只需要保存一个Pizza接口，就可以应对变化，PizzaStore并不关心到底是什么类型的pizza，这个由用户传入的type来决定。

## 工厂方法

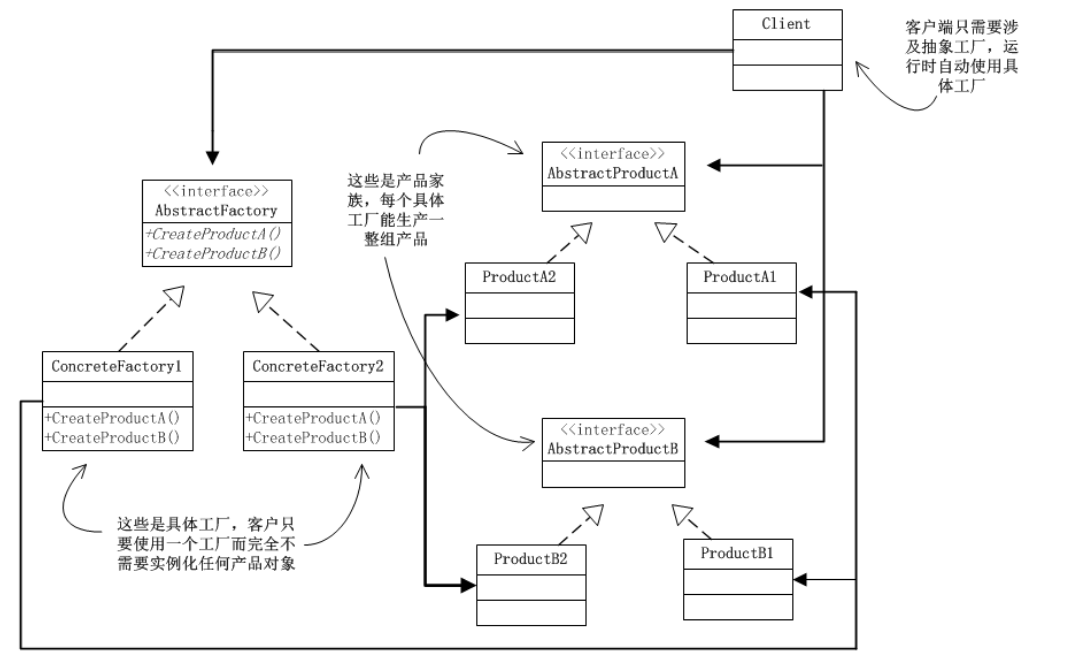
现在不仅仅局限于能够将pizza抽象出来了，现在需要让pizza具有当地的风味，例如芝加哥地区需要有芝加哥地区的风味，而纽约地区需要有纽约地区的风味。这样如果我们还是像之前那样使用简单工厂方法，分别创建出两种工厂，一种是NYPizzaFactory，另一种是ChicagoPizzaFactory，其实也未尝不可，但是这样其他的行为就不受约束，所以需要了解下工厂方法。



如上UML所示，PizzaStore是一个抽象类，其中createPizza()方法是一个抽象方法，由各个子类去实现，而orderPizza()方法是一个已经实现的方法，并且在该方法中调用了createPizza()方法，子类不需要覆盖orderPizza()方法，这样就可以保证让pizza的制作局限于Pizza Store类，而同时又能让加盟店可以自由地制作区域风味了。

## 抽象工厂

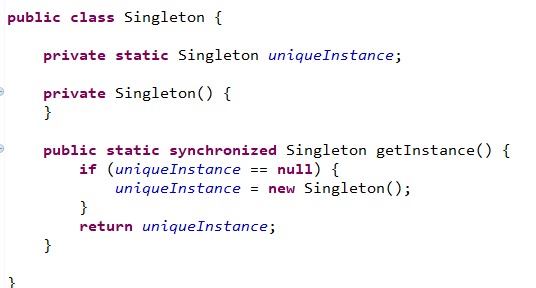
抽象工厂提供了一个接口，用于创建相关或者以来对象的家族，而不需要明确指定具体类。例如，制作pizza的原料各有不同，例如芝加哥风味的使用的是一种类型，而纽约使用的是另一种类型。但是这两者的体系相同，只不过具体的内容不同，所以这边可以使用抽象工厂。抽象工厂提供一个统一的接口，用于返回的也是接口。芝加哥风味的调料工厂就返回芝加哥风味的调料，纽约风味调料工厂就返回纽约风味的调料。由于是统一的调料接口，在使用的时候，调料也只需要声明为接口类型，而不需要具体的类型。Uml图如下：



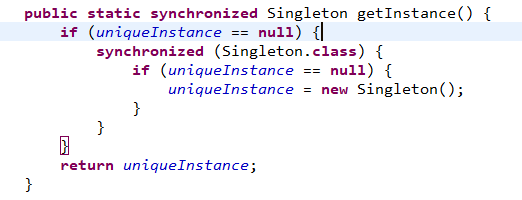
# 单例模式

单例模式确保了一个类只能有一个实例，并提供一个全局访问点。

单例模式可以有两种实现方式，一种是懒汉式：

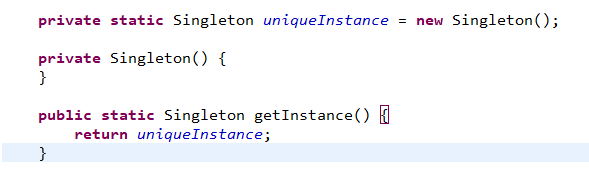


直到被需要的时候，才会去创建这个对象，但是上述的这个方法，虽然不会有并发问题，但是对性能方面有很大的影响，每次获取该对象的时候，都会要去争抢这把锁，其实这是多余的，只有第一次创建的时候才需要获取所来实现唯一的创建，所以上述饿汉式的写法有问题，应该改写成下述样子：



同时，uniqueInstance对象需要加上volatile修饰，为了能够将变化通知给其他线程。

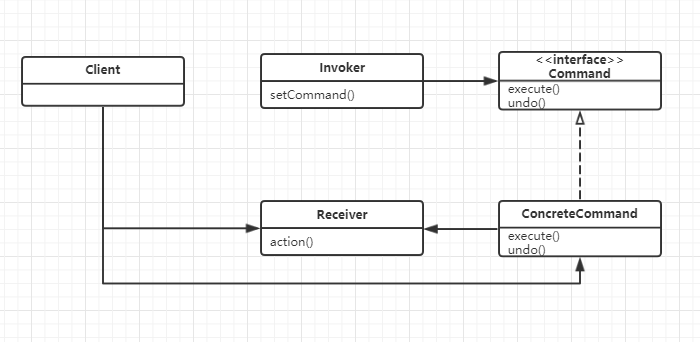
还有一种模式是饿汉式：



JVM在加载这个类的时候，就可以立马创建该唯一的单例实例，JVM保证在任何线程访问uniqueInstance静态变量之前，一定先创建此实例。

# 命令模式

将“请求”封装成对象，以便用不同的请求、队列、或者日志来参数化其他对象。具体类图如下



其中，client负责创建ConcreteCommand，并设置其接收者，Invoker对象会持有一个命令对象，并在某个时候调用命令对象的execute方法，Command为所有的对象声明了一个接口，调用命令对象的execute()方法，就可以让接收者进行相关的动作。ConcreateCommand对象实现了Command接口，并持有一个Receiver对象，在execute方法中最终会调用Receiver对象的方法。

命令模式能够让Invoker与Receiver之间解耦，Invoker并不关心具体是哪个Receiver执行了，他只需要知道调用着实现了Command接口，而具体调用哪一个，需要去创建ConcreteCommand。这样。Invoker就可以更改Command引用的对象了，而不需要更改源码。