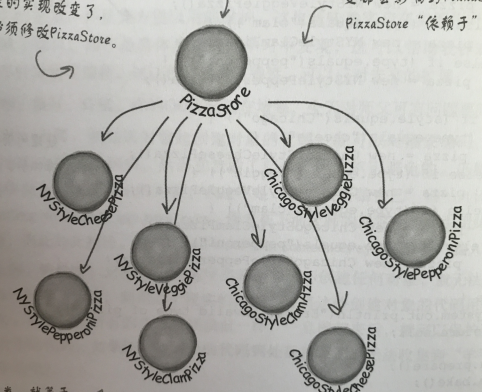
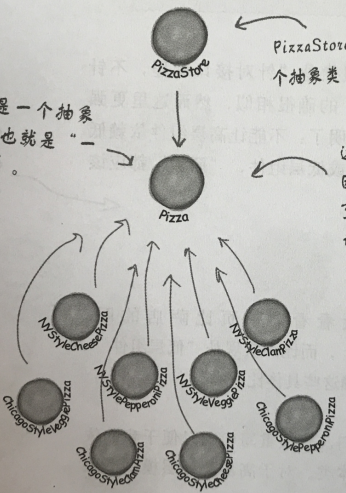
# 0、设计原则

* 封装变化，把变化封装成一个单独类里的一个方法。
* 多用组合，少用继承
* 针对接口编程
* 为交互对象之间的松耦合设计而努力
* 开闭原则：类应该对扩展开放，对修改关闭。 不可能类的所有地方都做到开闭原则，只需要把这个原则用在最重要的地方即可。
* 依赖倒置：要依赖抽象，不要依赖具体类。依赖倒置中的理解：  
     
  左图中的PizzaStore依赖于很多的具体Pizza类,而右图中的PizzaStore只依赖于Pizza类,下面的很多具体类依赖了Pizza类，看箭头就是依赖反过来了。

注意：在设计模式中，所谓的“实现一个接口”并不一定表示“写一个类，并利用implement来实现某个Java接口”，“实现一个接口”泛指“实现某个超类型（可以是类或接口）的某个方法”。

# 1、策略模式（Strategy Pattern）

简单理解就是把会变化的逻辑单独抽取出来封装成类，当然需要先定义一个接口，当又有新的变化的时候只需要创建另外的一个类实现此接口即可。就比如别人常比喻的诸葛亮给了刘备三个锦囊，刘备“怎么打仗”这个逻辑是会变化的，把“怎么打仗”逻辑封装成锦囊，给刘备传入什么锦囊，则按照什么锦囊进行打仗。

有一个游戏，在游戏中会出现各种鸭子，一边游泳一边呱呱叫。

*/\*\* 鸭子 \*/***public abstract class** Duck {  
  
 */\*\* 鸭叫声 \*/* **public void** quack() {  
 System.***out***.println(**"嘎嘎叫"**);  
 }  
  
 **public void** swim() {  
 System.***out***.println(**"游泳"**);  
 }  
  
 */\*\* 显示鸭子外观 \*/* **public abstract void** display();  
}

*/\*\* 野鸭 \*/***public class** MallardDuck **extends** Duck {  
 @Override  
 **public void** display() {  
 System.***out***.println(**"显示绿头鸭"**);  
 }  
}

*/\*\* 红头鸭 \*/***public class** RedheadDuck **extends** Duck {  
 @Override  
 **public void** display() {  
 System.***out***.println(**"显示红头鸭"**);  
 }  
}

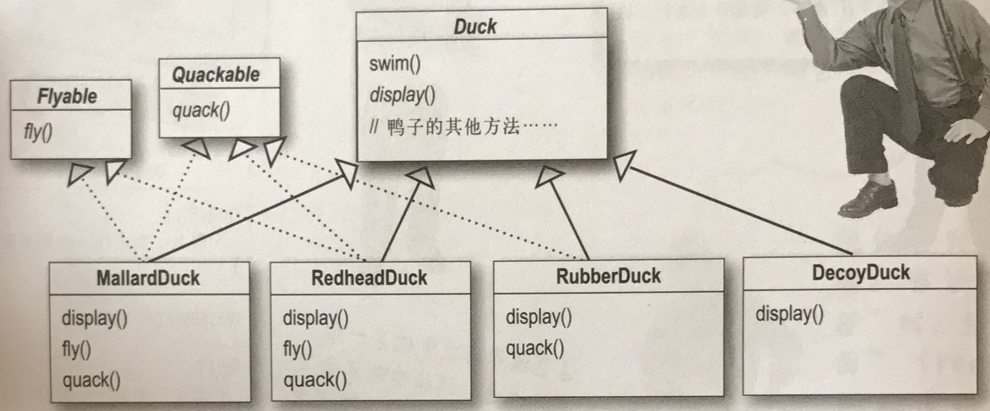
这时需要部要求鸭子会飞，开发员在Duck类上增加了fly()方法，问题来了，领导发现游戏里有很多橡皮鸭飞来飞去。修改橡皮鸭如下：

*/\*\* 橡皮鸭 \*/***public class** RubberDuck **extends** Duck {  
  
 @Override  
 **public void** quack() {  
 System.***out***.println(**"吱吱叫"**);  
 }  
  
 @Override  
 **public void** fly() {  
 *// 什么也不写，橡皮鸭不飞* }  
  
 @Override  
 **public void** display() {  
 System.***out***.println(**"显示橡皮鸭"**);  
 }  
}

这样做好像可以，但是以后如果我们加入一个木头鸭（DecoyDuck）呢？木头鸭不会叫也不会飞。又比如有1000个各类的鸭子呢？都要这样覆盖方法吗？来看看利用继承Duck提供行为，这会导致下列的缺点：

* 代码在多个子类中重复
* 很难知道所有鸭子的全部行为
* 改变会牵一发动全身，造成其他鸭子不想要的改变

了解到这些问题之后，利用接口如何，让需要飞的鸭子实现Flyable，不需要的就不实现，如下：



但是这样又导致了另外一个问题，就是叫的无法复用fly和quack行为。

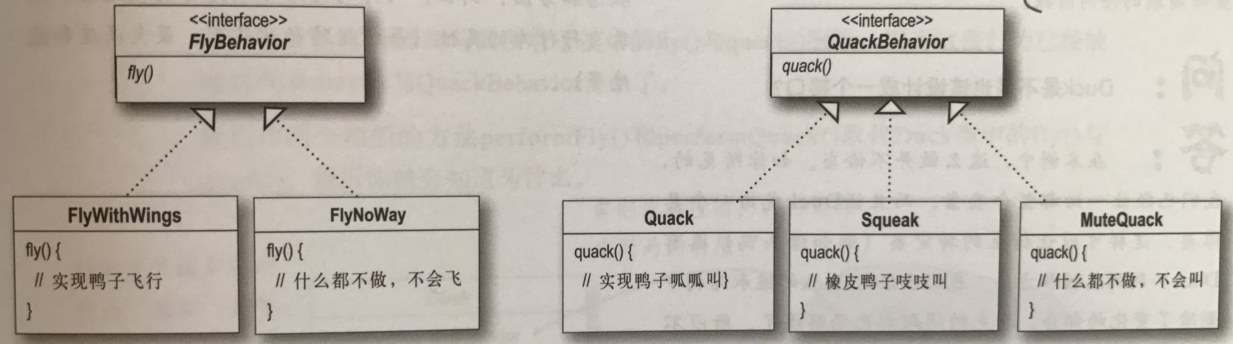
这时通过两个设计原则来解决这个问题：

* + 找出应用中可能需要变化的地方，把它们独立出来
  + 面向接口编程

鸭子的叫声和飞的行为在一些鸭子中相同需要复用，但是在另一些鸭子中又不需要这些行为，所以这些行为要独立出来，以便可以复用，如下：

*/\*\* 鸭叫的行为 \*/***public interface** QuackBehavior {  
 **void** quack();  
}

*/\*\* 飞的行为 \*/***public interface** FlyBehavior {  
 **void** fly();  
}



我们把鸭子飞行和叫的行为抽取了出来，这样，Duck鸭子类就不再需要知道这些行为的实现细节了，而且这些行为可以被其它的对象复用，而且我们新增加一些行为也不会影响到即有的行为类，也不会影响到使用飞行行为的鸭子类。

现在修改Duck类，如下：

*/\*\* 鸭子 \*/***public abstract class** Duck {  
 **protected** FlyBehavior **flyBehavior**;  
 **protected** QuackBehavior **quackBehavior**;

**public void** setFlyBehavior(FlyBehavior flyBehavior) { **this**.**flyBehavior** = flyBehavior; }  
**public void** setQuackBehavior(QuackBehavior quackBehavior) { **this**.**quackBehavior** = quackBehavior; }

**public void** swim() {  
 System.***out***.println(**"游泳"**);  
 }**public void** performQuack() { **quackBehavior**.quack(); }  
 **public void** performFly() { **flyBehavior**.fly(); }  
**public abstract void** display();  
}

在这里，飞和叫的行为，在所有的鸭子各类中，有些各类它们的行为相同，有的不相同，所以不能简单的把把这两种行为在Duck中声明抽象方法，这样没法复用，所以把这些行为抽取成独立的类，这样可以复用，而且可以扩展而不影响Duck类。

具体鸭子的实现如下：

*/\*\* 野鸭 \*/***public class** MallardDuck **extends** Duck {  
 **public** MallardDuck() {  
 **quackBehavior** = **new** Quack();  
 **flyBehavior** = **new** FlyWithWings();  
 }  
  
 @Override  
 **public void** display() {  
 System.***out***.println(**"显示绿头鸭"**);  
 }

}

运行示例：

**public static void** main(String[] args) {  
 Duck duck = **new** MallardDuck();  
 duck.performQuack();  
 duck.performFly();  
  
 duck.setFlyBehavior(**new** FlyNoWay());  
 duck.performFly();  
}

如上面代码，我们在运行时动态的改变了鸭子的行为，就比如玩飞机大战，吃到特效后可以动态地改变发射的子弹行为（发散型、激光型、跟踪型等）。

“有一个”可能比“是一个”更好，每个鸭子都有一个FlyBehavior和一个QuackBehavior，将飞和叫的行为委托给它们代为处理。当将两个类结合起来使用就是组合，这种做法和“继承”不同的地方在于，鸭子的行为不是继承的，而是和适当的行为对象“组合”来的。这是我们的第三个设计原则：多用组合，少用继承。使用组合建立系统有很大的弹性。

# 2、观察者模式（Observer）

观察者模式是JDK中使用最多的模式之一。

模式定义：定义了对象之间一对多依赖（一个主题，多个观察者），这样一来，当一个对象改变状态时，它的所有依赖都会收到通知并自动更新。

观察者模式是松耦合的，因为主题只知道观察者实现了某个接口（也就是Observer接口），主题不需要知道观察者的具体类是谁、做了些什么或其他任何细节。改变主题或观察者其中一方，并不会影响另一方。

**public** **interface** Observer {

**void** update(**int** x, **int** y);

}

这样设计观察者不太好，因为不得于扩展，如某一天，主题要给观察者传更多的数据，则需要修改Observer的update方法增加更多的参数。更好的做法是update方法不接收具体参数，而只接收主题对象，这样它可以拿着主题引用获取任意想要的数据，主题发生改变时，Observer类也不需要做修改，而且观察者能知道是哪个主题通知的它，观察者可以观察多个主题的。所以可以这样声明：

**public** **class** MyObserver **implements** Observer {

@Override

**public** **void** update(Observable observable) {

**if** (observable **instanceof** MySubject) {

MySubject subject = (MySubject) observable;

// 调用subject的get方法获取数据

}

}

}

JDK本身有内置的主题类和观察者接口类，分别为：Observable、Observer，Observable是一个类，里面已经提供好了主题所需要的方法，如果我们的一个对象可以被观察，则需要继承Observable类，在通知观察者时需要先调用setChanged()方法，再调用notifyObserver()或notifyObserver(Object arg)，这两个方法在通知了观察者之后会把chagned变量设置为false。

但是Observable是一个类会导致一些问题：它不是面向接口编程，当你的类要继承Observable时就不能继承其它的类了，而且这个类也没有对应的接口类，意味着你不能自己实现。

# 3、装饰者模式（Decorator）

定义：动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。装饰者模式很好的运用了开闭原则，即扩展了原有类的功能，又不需要修改原有类的代码。

IO流中的包装类就是使用了装饰者模式。

运用：比如星巴克中售卖系统，对于不同的咖啡有不同的类，咖啡类有计算自己价格的方法，但是咖啡可以加配料，配料不同价格就会有变化，如果在咖啡类中增加配料变量，这样做不利于扩展，哪天有一种新配料，则又需要增加新的变量。这时可以使用装饰者模式。每个咖啡类肯定要实现一个接口。我们把配料也变成类并实现同样的接口，并且创建实现的时候需要包装一个该接口的类，这样我们的扩展就非常容易，而且扩展时不需要修改原来已有的代码。

装饰者和被装饰者都实现了相同的接口，这样的好处是装饰者可以取代被装饰者。

在设计装饰者类之前，也可以先设计一个装饰者的基类，这个基类用于包装最基本的接口类，如IO流中的FilterInputStream，它包装了InputStream，代码如下：

**public** **class** FilterInputStream **extends** InputStream {

**protected** **volatile** InputStream in;

**protected** FilterInputStream(InputStream in) {

**this**.in = in;

}

**public** **int** read() **throws** IOException {

**return** in.read();

}

**public** **int** read(**byte** b[]) **throws** IOException {

**return** read(b, 0, b.length);

}

// 。。。其它更多的包装方法

}

这个类的子类有：BufferedInputStream、LineNumberInputStream、DataInputStream等。

设计一个装饰者基类的好处是，基类提供了接收被装饰者的构造方法，而且把所有的方法都进行了装饰，再写具体的装饰者类时只需要覆盖想要改变功能的方法即可，如下面例子：

写一个装饰者类，装饰一个输入流，把读取的字符全部变成小写：

**public** **class** LowerCaseInputStream **extends** FilterInputStream {

**protected** LowerCaseInputStream(InputStream in) {

**super**(in);

}

@Override

**public** **int** read() **throws** IOException {

**int** aByte = **super**.read();

**return** aByte == -1 ? aByte : Character.*toLowerCase*((**char**) aByte);// -1代表读到流末尾了

}

@Override

**public** **int** read(**byte**[] b, **int** off, **int** len) **throws** IOException {

**int** length = **super**.read(b, off, len);

**for** (**int** i = off; i < off + length; i++) {

b[i] = (**byte**) Character.*toLowerCase*((**char**) b[i]);

}

**return** length;

}

}

我们知道一个char是两个字节，而读进来一个字节就进行转换，这好像不行吧，其实是没有问题的，因为Unicode中的两个字节对应一个字符，ASCII吗中1个字节对应1个字符，对于字母，虽然在Unicode中占了两个字节，但是只用了1字节存储字母的编码，因为字母用一个字节足已装下。

检验结果：

**public** **static** **void** main(String[] args) **throws** Exception {

LowerCaseInputStream lcis = **new** LowerCaseInputStream(**new** FileInputStream("haha.txt"));

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(lcis, "GBK"));

System.***out***.println(reader.readLine());

}

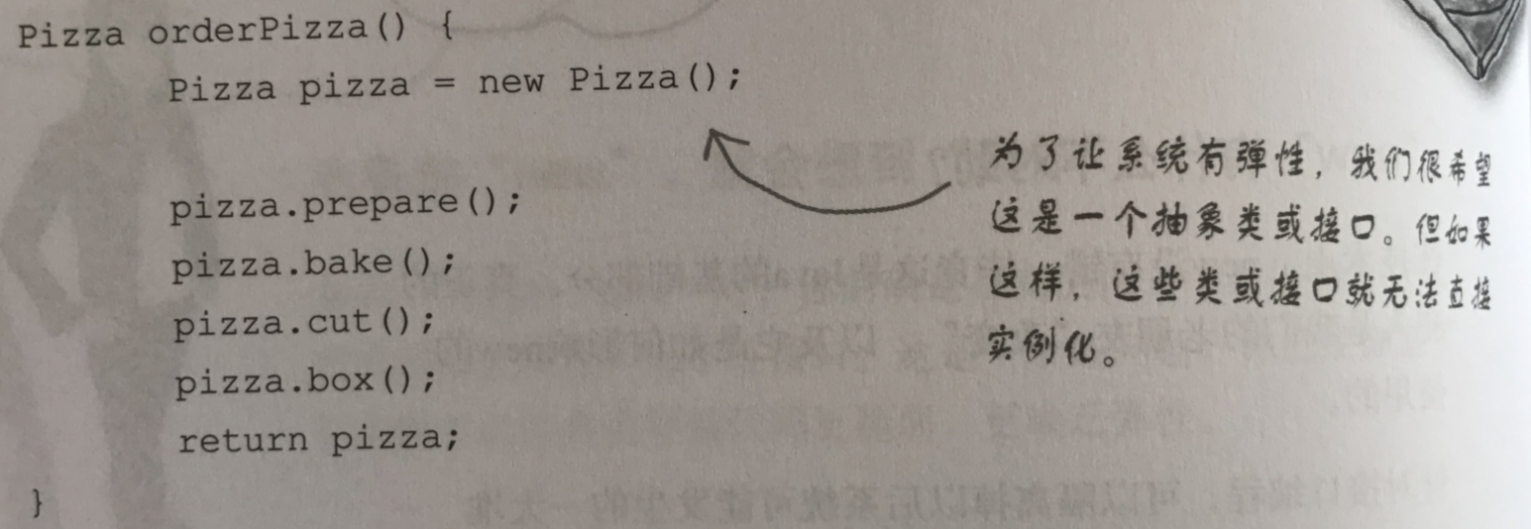
# 4、工厂模式（Factory）

定义：定义了一个创建对象的接口（这里的接口是方法的意思，称为工厂方法 ），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。

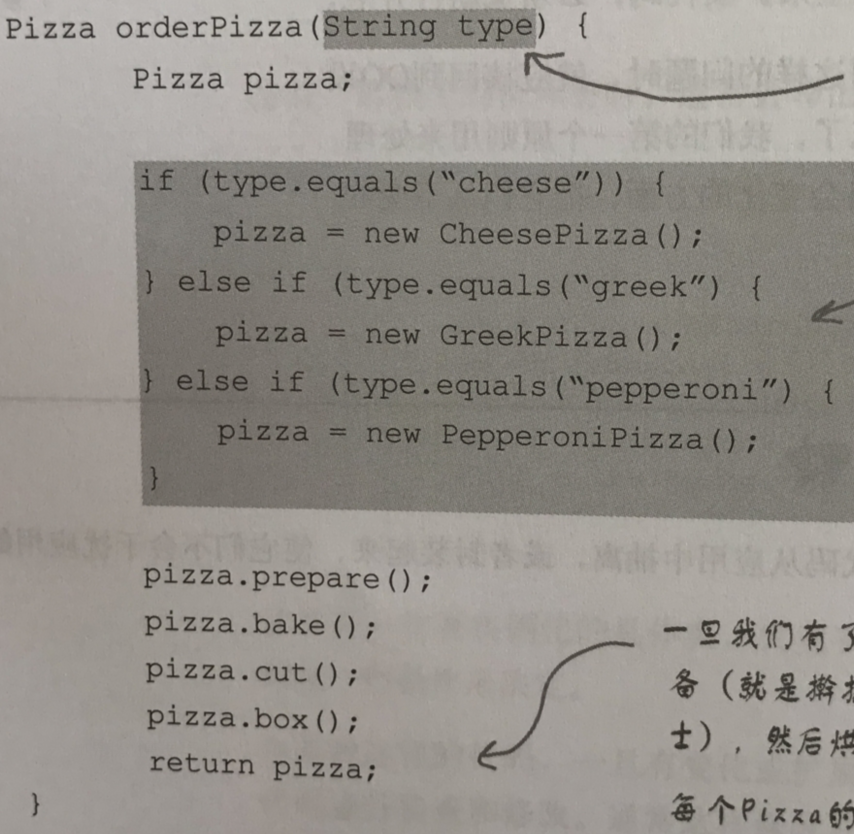
简单理解“工厂方法模式”其实就是把创建对象的代码封装成一个方法，由这个方法来产生对象。

下面讲解“简单工厂”和“工厂方法”，“简单工厂”不是设计模式，而“工厂方法”是一个设计模式。

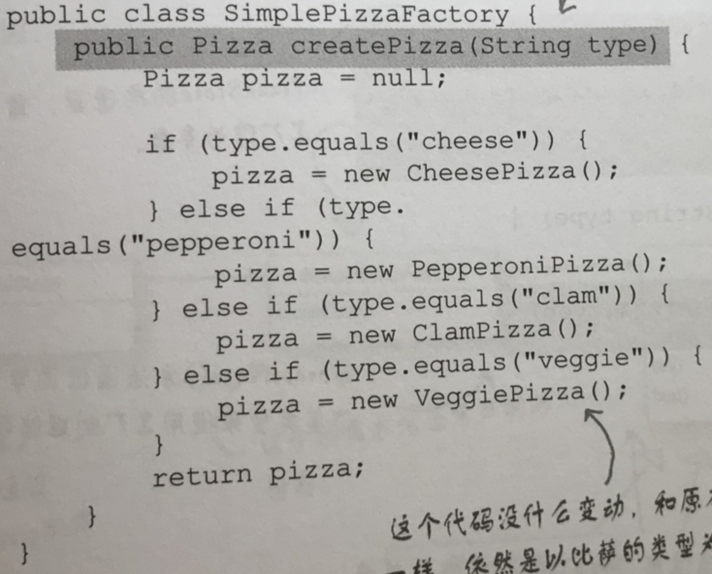
有个原则是针对接口编程，但是当我们使用new的时候，不正是针对实现编程吗？



如果我们需要更多类型的Pizza，所以必须增加一些代码来判断要创建什么类型的Pizza，如下：



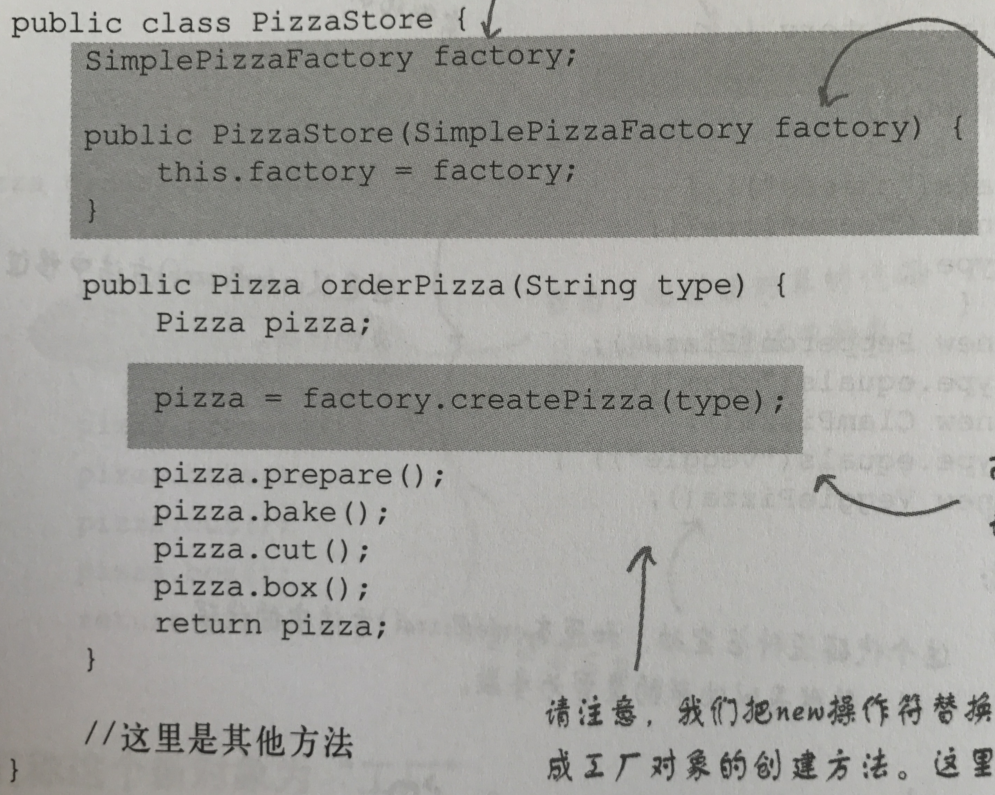
如果我们新增加了一个Pizza的类型，则上面的代码又需要修改，这样就导致无法使用orderPizza方法对修改关闭，但是现在已经知道哪些会改变，哪些不会改变，所以可以对改变进行封装了如下：



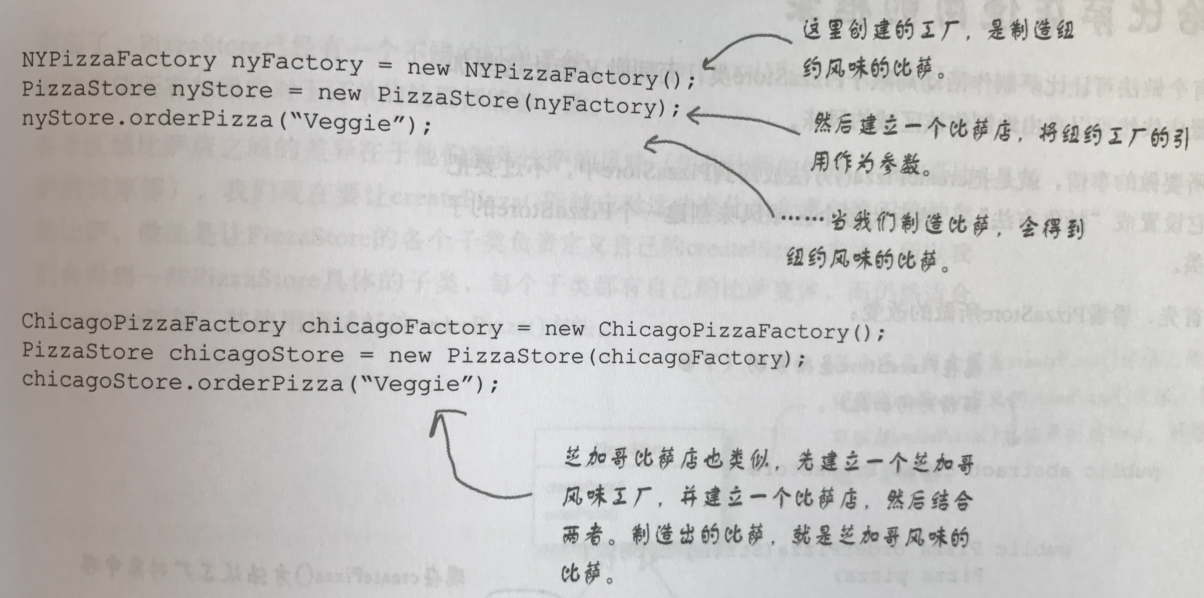
这样做有什么好处？似乎只是把问题搬到另一个对象而已，问题依然存在。 这样做的好处是：可能不止是orderPizza需要创建Pizza对象，可能其它地方也需要，这样代码就得到了复用，当有新品种时只需要改一个地方（工厂类）即可。

这里的工厂方法不是静态的，如果是静态的，则称为静态工厂，上面是简单工厂。静态工厂不需要实例化就能创建对象，但是它的缺点是不能通过继承来改变创建方法的行为。

Pizza店的完整代码如下：

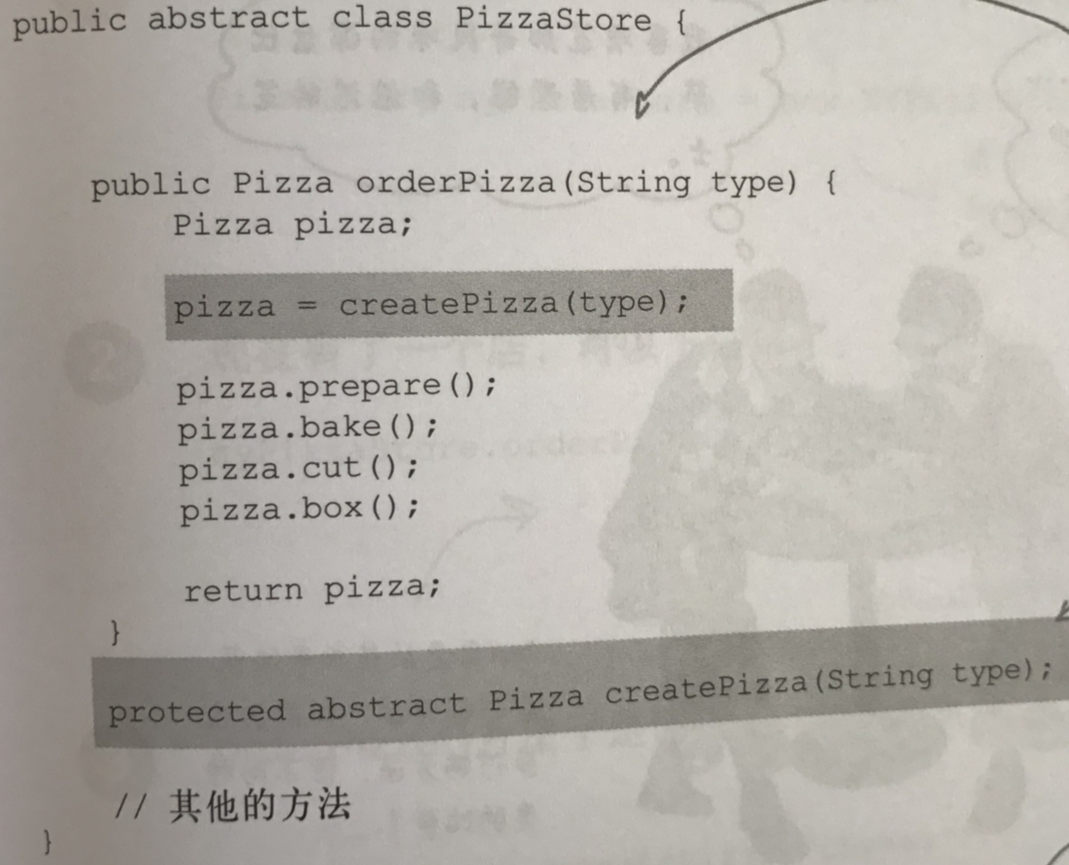


我们发现这里的工厂没有写死，因为生产Pizza的工厂可能不止一个，再比如产生手机工厂有很多。如下：



简单工厂其实不是一个设计模式，反而比较像是一种编程习惯。

这里面PizzaStore的实例只有一个，我们假设要开很多PizzaStore的分店，这些分店因为地域位置不同，可以决定生产Pizza的各类，所以也可以这样写：



这样不同的分店只需要写一个类继承这个PizzaStore，然后实现createPizza方法即可，不再需要Factory类。其实这个时候PizzaStore的子类就相当于工厂类了。而createPizza方法就是工厂方法。 这种形式就是工厂方法模式。

还有一个抽象工厂模式。其实就是简单工厂的升级，先写一个工厂接口，其它工厂都来实现这个工厂，比如生产手机需要电池配件，而电池配件的产生厂商有很多，如果这些厂商都实现了一个接口，则我们在保存电池厂商工厂变量的时候，可以声明为接口类型，哪天要换电池厂商了，则只需要把new的实例换一下，其它所有代码都不用变。这时抽象工厂中的方法就是工厂方法。

工厂方法和抽象工厂都属于工厂模式。

工厂方法用继承实现，抽象工厂用对象的组合实现

# 5、单例模式（Singleton）

有些对象只需要一个，如：线程池、缓存、对话、日志对象。

## 饿汉式单例：

**public class** Singleton {  
 **private static final** Singleton ***instance*** = **new** Singleton();  
 **private** Singleton(){}  
 **public static** Singleton getInstance() { **return *instance***; }  
}

这种方式的缺点是类一加载就创建实例。优点是线程安全，且没有同步的消耗。如果一个类一定会使用的，则可以使用这种方式。

## 懒汉式单例：

**public class** Singleton {  
 **private static** Singleton *instance*;  
 **private** Singleton(){}  
 **public static synchronized** Singleton getInstance() {  
 **if** (*instance* == **null**) {  
 *instance* = **new** Singleton();  
 }  
 **return** *instance*;  
 }  
}

优点是：全程安全，且在用到的时候才创建实例，但是同步消耗大。 如果你的多线程访问并不多，可以使用这个方式。

下面的代码验证带了同步后的效率：

、

**public** **static** **int** *count* = 0;

**public** **static** **void** main(String[] args) {

**long** start = System.*currentTimeMillis*();

**int** size = 1000;

**for** (**int** i = 0; i < size; i++) {

**new** Thread(){

**public** **void** run() {

Singleton instance = Singleton.*getInstance*();

*count* = *count* + 1;

**if** (*count* > (size - 5)) { // 因为count没有同步，最终结果不一定和size相等

**long** parseTime = System.*currentTimeMillis*() - start;

System.***out***.println("parseTime = " + parseTime);

}

};

}.start();

}

}

运行结果需要79毫秒，我们把getInstance中的synchronized删除，然后再次运行上面的代码进行测试，结果是77。所以1000个线程这种影响不大，我们把size设置成10万，输出的判断为size减1千，再次进行运行查看结果，同步与不同步结果分别为：6341和6323，相关18毫秒，好像影响也不大啊。怪了。

## 双检查单例：

**public class** Singleton {  
 **private volatile static** Singleton *instance*;  
 **private** Singleton(){}  
 **public static** Singleton getInstance() {  
 **if** (*instance* == **null**) {  
 **synchronized** (Singleton.**class**) {  
 **if** (*instance* == **null**) {  
 *instance* = **new** Singleton();  
 }  
 }  
 }  
 **return** *instance*;  
 }  
}

优点是：全程安全，且在用到的时候才创建实例，并且同步消耗小。

注：双重检查加锁适用于1.4及更高版本的Java，因为这些版本对于volatile关键字的实现会导致双重检查加锁的失效。如果你不能用Java 5以上版本就不要用此技术实现单例模式。

如果有多个类加载器去加载单例，可能会产生多个单例，解决方案：指定使用同一个类加载器。

# 命令模式