[23种设计模式（2）：工厂方法模式](http://blog.csdn.net/zhengzhb/article/details/7348707)

标签： [设计模式](http://www.csdn.net/tag/%e8%ae%be%e8%ae%a1%e6%a8%a1%e5%bc%8f)[产品](http://www.csdn.net/tag/%e4%ba%a7%e5%93%81)[class](http://www.csdn.net/tag/class)[interface](http://www.csdn.net/tag/interface)[扩展](http://www.csdn.net/tag/%e6%89%a9%e5%b1%95)[string](http://www.csdn.net/tag/string)

2012-03-15 09:05 63194人阅读 [评论](http://blog.csdn.net/zhengzhb/article/details/7348707#comments)(51) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhengzhb/article/details/7348707#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

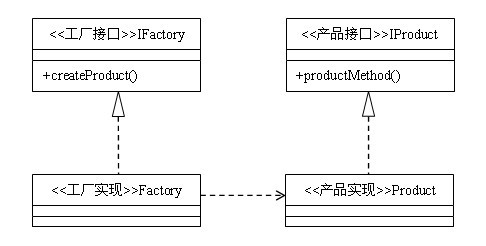
设计模式（26） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

**定义：**定义一个用于创建对象的接口，让子类决定实例化哪一个类，工厂方法使一个类的实例化延迟到其子类。

**类型：**创建类模式

**类图：**



**工厂方法模式代码**

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7348707) [copy](http://blog.csdn.net/zhengzhb/article/details/7348707)

1. **interface** IProduct {
2. **public** **void** productMethod();
3. }
5. **class** Product **implements** IProduct {
6. **public** **void** productMethod() {
7. System.out.println("产品");
8. }
9. }
11. **interface** IFactory {
12. **public** IProduct createProduct();
13. }
15. **class** Factory **implements** IFactory {
16. **public** IProduct createProduct() {
17. **return** **new** Product();
18. }
19. }
21. **public** **class** Client {
22. **public** **static** **void** main(String[] args) {
23. IFactory factory = **new** Factory();
24. IProduct prodect = factory.createProduct();
25. prodect.productMethod();
26. }
27. }

**工厂模式：**

        首先需要说一下工厂模式。工厂模式根据抽象程度的不同分为三种：简单工厂模式（也叫静态工厂模式）、本文所讲述的**工厂方法模式**、以及抽象工厂模式。工厂模式是编程中经常用到的一种模式。它的主要优点有：

* 可以使代码结构清晰，有效地封装变化。在编程中，产品类的实例化有时候是比较复杂和多变的，通过工厂模式，将产品的实例化封装起来，使得调用者根本无需关心产品的实例化过程，只需依赖工厂即可得到自己想要的产品。
* 对调用者屏蔽具体的产品类。如果使用工厂模式，调用者只关心产品的接口就可以了，至于具体的实现，调用者根本无需关心。即使变更了具体的实现，对调用者来说没有任何影响。
* 降低耦合度。产品类的实例化通常来说是很复杂的，它需要依赖很多的类，而这些类对于调用者来说根本无需知道，如果使用了工厂方法，我们需要做的仅仅是实例化好产品类，然后交给调用者使用。对调用者来说，产品所依赖的类都是透明的。

**工厂方法模式：**

通过工厂方法模式的类图可以看到，工厂方法模式有四个要素：

* 工厂接口。工厂接口是工厂方法模式的核心，与调用者直接交互用来提供产品。在实际编程中，有时候也会使用一个抽象类来作为与调用者交互的接口，其本质上是一样的。
* 工厂实现。在编程中，工厂实现决定如何实例化产品，是实现扩展的途径，需要有多少种产品，就需要有多少个具体的工厂实现。
* 产品接口。产品接口的主要目的是定义产品的规范，所有的产品实现都必须遵循产品接口定义的规范。产品接口是调用者最为关心的，产品接口定义的优劣直接决定了调用者代码的稳定性。同样，产品接口也可以用抽象类来代替，但要注意最好不要违反里氏替换原则。
* 产品实现。实现产品接口的具体类，决定了产品在客户端中的具体行为。

        前文提到的简单工厂模式跟工厂方法模式极为相似，区别是：简单工厂只有三个要素，他没有工厂接口，并且得到产品的方法一般是静态的。因为没有工厂接口，所以在工厂实现的扩展性方面稍弱，可以算所工厂方法模式的简化版，关于简单工厂模式，在此一笔带过。

**适用场景：**

        不管是简单工厂模式，工厂方法模式还是抽象工厂模式，他们具有类似的特性，所以他们的适用场景也是类似的。

        首先，作为一种创建类模式，在任何需要生成**复杂对象**的地方，都可以使用工厂方法模式。有一点需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过new就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。

       其次，工厂模式是一种典型的解耦模式，迪米特法则在工厂模式中表现的尤为明显。假如调用者自己组装产品需要增加依赖关系时，可以考虑使用工厂模式。将会大大降低对象之间的耦合度。

       再次，由于工厂模式是依靠抽象架构的，它把实例化产品的任务交由实现类完成，扩展性比较好。也就是说，当需要系统有比较好的扩展性时，可以考虑工厂模式，不同的产品用不同的实现工厂来组装。

**典型应用**

       要说明工厂模式的优点，可能没有比组装汽车更合适的例子了。场景是这样的：汽车由发动机、轮、底盘组成，现在需要组装一辆车交给调用者。假如不使用工厂模式，代码如下：

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7348707) [copy](http://blog.csdn.net/zhengzhb/article/details/7348707)

1. **class** Engine {
2. **public** **void** getStyle(){
3. System.out.println("这是汽车的发动机");
4. }
5. }
6. **class** Underpan {
7. **public** **void** getStyle(){
8. System.out.println("这是汽车的底盘");
9. }
10. }
11. **class** Wheel {
12. **public** **void** getStyle(){
13. System.out.println("这是汽车的轮胎");
14. }
15. }
16. **public** **class** Client {
17. **public** **static** **void** main(String[] args) {
18. Engine engine = **new** Engine();
19. Underpan underpan = **new** Underpan();
20. Wheel wheel = **new** Wheel();
21. ICar car = **new** Car(underpan, wheel, engine);
22. car.show();
23. }
24. }

        可以看到，调用者为了组装汽车还需要另外实例化发动机、底盘和轮胎，而这些汽车的组件是与调用者无关的，严重违反了迪米特法则，耦合度太高。并且非常不利于扩展。另外，本例中发动机、底盘和轮胎还是比较具体的，在实际应用中，可能这些产品的组件也都是抽象的，调用者根本不知道怎样组装产品。假如使用工厂方法的话，整个架构就显得清晰了许多。

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7348707) [copy](http://blog.csdn.net/zhengzhb/article/details/7348707)

1. **interface** IFactory {
2. **public** ICar createCar();
3. }
4. **class** Factory **implements** IFactory {
5. **public** ICar createCar() {
6. Engine engine = **new** Engine();
7. Underpan underpan = **new** Underpan();
8. Wheel wheel = **new** Wheel();
9. ICar car = **new** Car(underpan, wheel, engine);
10. **return** car;
11. }
12. }
13. **public** **class** Client {
14. **public** **static** **void** main(String[] args) {
15. IFactory factory = **new** Factory();
16. ICar car = factory.createCar();
17. car.show();
18. }
19. }

        使用工厂方法后，调用端的耦合度大大降低了。并且对于工厂来说，是可以扩展的，以后如果想组装其他的汽车，只需要再增加一个工厂类的实现就可以。无论是灵活性还是稳定性都得到了极大的提高。