# 工厂设计模式

目录

[工厂设计模式 1](#_Toc22825206)

[**一、 综述** 1](#_Toc22825207)

[**二、 简单工厂模式** 1](#_Toc22825208)

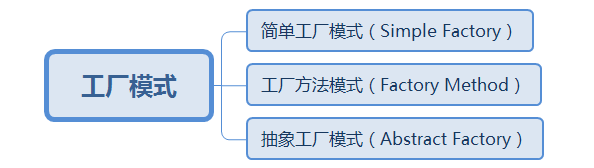
[**三、 工厂方法模式** 4](#_Toc22825209)

[**四、 抽象工厂模式（Abstract Factory）** 6](#_Toc22825210)

1. **综述**

工厂模式主要是为创建对象提供过渡接口，以便将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的。

工厂模式可以分为三类：



这三种模式从上到下逐步抽象，并且更具一般性。

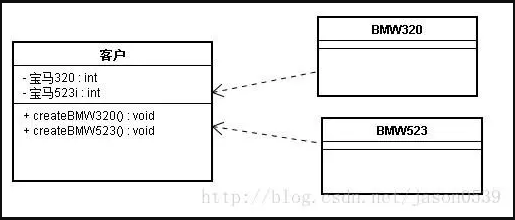
GOF在《设计模式》一书中将工厂模式分为两类：工厂方法模式（Factory Method）与抽象工厂模式（Abstract Factory）。将简单工厂模式（Simple Factory）看为工厂方法模式的一种特例，两者归为一类。

**区别**

| **工厂方法模式：** | **抽象工厂模式：** |
| --- | --- |
| 一个抽象产品类，可以派生出多个具体产品类。 | 多个抽象产品类，每个抽象产品类可以派生出多个具体产品类。 |
| 一个抽象工厂类，可以派生出多个具体工厂类。 | 一个抽象工厂类，可以派生出多个具体工厂类。 |
| 每个具体工厂类只能创建一个具体产品类的实例。 | 每个具体工厂类可以创建多个具体产品类的实例。 |

**区别总结：**  
**工厂方法模式只有一个抽象产品类，而抽象工厂模式有多个。  
工厂方法模式的具体工厂类只能创建一个具体产品类的实例，而抽象工厂模式可以创建多个。**

1. **简单工厂模式（Simple Factory）**

建立一个工厂（一个函数或一个类方法）来制造新的对象。  


public class BMW320 {

public BMW320(){

System.out.println("制造-->BMW320");

}

}

public class BMW523 {

public BMW523(){

System.out.println("制造-->BMW523");

}

}

public class Customer {

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

BMW320 bmw320 = new BMW320();

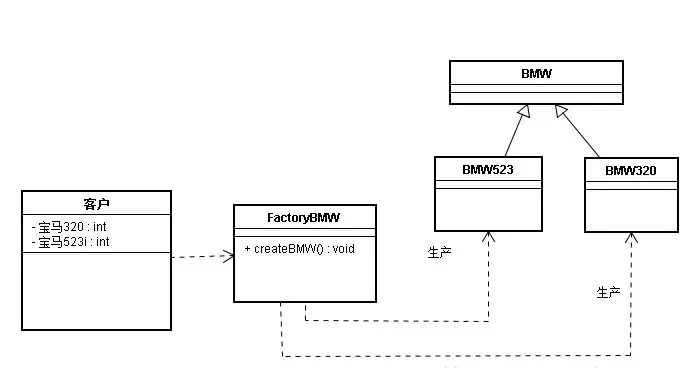
BMW523 bmw523 = new BMW523();

}

}

客户需要知道怎么去创建一款车,客户和车就紧密耦合在一起了.为了降低耦合,就出现了工厂类,把创建宝马的操作细节都放到了工厂里面去,客户直接使用工厂的创建工厂方法,传入想要的宝马车型号就行了,而不必去知道创建的细节.这就是工业革命了：**简单工厂模式**

即我们建立一个工厂类方法来制造新的对象。如图：



**产品类：**

//抽象产品

abstract class BMW {

public BMW(){

}

}

//具体产品

public class BMW320 extends BMW {

public BMW320() {

System.out.println("制造-->BMW320");

}

}

//具体产品

public class BMW523 extends BMW{

public BMW523(){

System.out.println("制造-->BMW523");

}

}

**工厂类：**

public class Factory {

public BMW createBMW(int type) {

switch (type) {

case 320:

return new BMW320();

case 523:

return new BMW523();

default:

break;

}

return null;

}

}

**客户类：**

public class Customer {

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

Factory factory = new Factory();

BMW bmw320 = factory.createBMW(320);

BMW bmw523 = factory.createBMW(523);

}

}

**简单工厂模式又称静态工厂方法模式。从命名上就可以看出这个模式一定很简单。它存在的目的很简单：定义一个用于创建对象的接口。**  
先来看看它的组成：  
1) 工厂类角色：这是本模式的核心，含有一定的商业逻辑和判断逻辑，用来创建产品  
2) 抽象产品角色：它一般是具体产品继承的父类或者实现的接口。  
3) 具体产品角色：工厂类所创建的对象就是此角色的实例。在java中由一个具体类实现。

下面我们从开闭原则（对扩展开放；对修改封闭）上来分析下简单工厂模式。当客户不再满足现有的车型号的时候，想要一种速度快的新型车，只要这种车符合抽象产品制定的合同，那么只要通知工厂类知道就可以被客户使用了。所以对产品部分来说，它是符合开闭原则的；但是工厂部分好像不太理想，**因为每增加一种新型车，都要在工厂类中增加相应的创建业务逻辑（createBMW(int type)方法需要新增case），这显然是违背开闭原则的**。可想而知对于新产品的加入，工厂类是很被动的。对于这样的工厂类，我们称它为全能类或者上帝类。   
        我们举的例子是最简单的情况，而在实际应用中，很可能产品是一个多层次的树状结构。由于简单工厂模式中只有一个工厂类来对应这些产品，所以这可能会把我们的上帝累坏了，也累坏了我们这些程序员。  
        于是工厂方法模式作为救世主出现了。 工厂类定义成了接口,而每新增的车种类型,就增加该车种类型对应工厂类的实现,这样工厂的设计就可以扩展了,而不必去修改原来的代码。

1. **工厂方法模式（Factory Method）**

工厂方法模式去掉了简单工厂模式中工厂方法的静态属性，使得它可以被子类继承。这样在简单工厂模式里集中在工厂方法上的压力可以由工厂方法模式里不同的工厂子类来分担。   
工厂方法模式组成：   
       1)抽象工厂角色： 这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在java中它由抽象类或者接口来实现。   
       2)具体工厂角色：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。   
       3)抽象产品角色：它是具体产品继承的父类或者是实现的接口。在java中一般有抽象类或者接口来实现。   
       4)具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在java中由具体的类来实现。   
       工厂方法模式使用继承自抽象工厂角色的多个子类来代替简单工厂模式中的“上帝类”。正如上面所说，这样便分担了对象承受的压力；而且这样使得结构变得灵活 起来——当有新的产品产生时，只要按照抽象产品角色、抽象工厂角色提供的合同来生成，那么就可以被客户使用，而不必去修改任何已有 的代码。可以看出工厂角色的结构也是符合开闭原则的！ 代码如下：

产品类：

abstract class BMW {

public BMW(){

}

}

public class BMW320 extends BMW {

    public BMW320() {

        System.out.println("制造-->BMW320");

    }

}

public class BMW523 extends BMW{

    public BMW523(){

        System.out.println("制造-->BMW523");

    }

}

创建工厂类：

interface FactoryBMW {

    BMW createBMW();

}

public class FactoryBMW320 implements FactoryBMW{

    @Override

    public BMW320 createBMW() {

        return new BMW320();

    }

}

public class FactoryBMW523 implements FactoryBMW {

    @Override

    public BMW523 createBMW() {

        return new BMW523();

    }

}

客户类：

public class Customer {

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

        FactoryBMW320 factoryBMW320 = new FactoryBMW320();

        BMW320 bmw320 = factoryBMW320.createBMW();

        FactoryBMW523 factoryBMW523 = new FactoryBMW523();

        BMW523 bmw523 = factoryBMW523.createBMW();

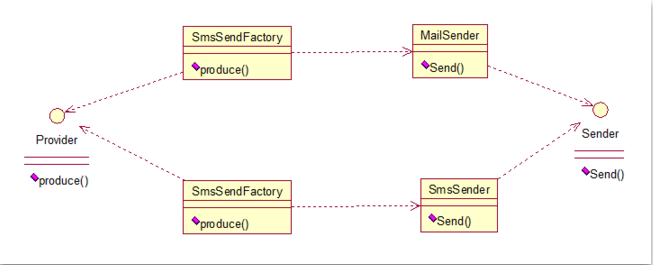
    }

}

工厂方法模式仿佛已经很完美的对对象的创建进行了包装，使得客户程序中仅仅处理抽象产品角色提供的接口，但使得对象的数量成倍增长。当产品种类非常多时，会出现大量的与之对应的工厂对象，这不是我们所希望的。

1. **抽象工厂模式（Abstract Factory）**

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。因为抽象工厂不太好理解，我们先看看图，然后就和代码，就比较容易理解。



请看例子：

public interface Sender {

    public void Send();

}

两个实现类：

public class MailSender implements Sender {

    @Override

    public void Send() {

        System.out.println("this is mailsender!");

    }

}

public class SmsSender implements Sender {

    @Override

    public void Send() {

        System.out.println("this is sms sender!");

    }

}

两个工厂类：

public class SendMailFactory implements Provider {

    @Override

    public Sender produce(){

        return new MailSender();

    }

}

public class SendSmsFactory implements Provider{

    @Override

    public Sender produce() {

        return new SmsSender();

    }

}

再提供一个接口：

public interface Provider {

    public Sender produce();

}

测试类：

public class Test {

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

        Provider provider = new SendMailFactory();

        Sender sender = provider.produce();

        sender.Send();

    }

}

其实这个模式的好处就是，如果你现在想增加一个功能：发及时信息，则只需做一个实现类，实现Sender接口，同时做一个工厂类，实现Provider接口，就OK了，无需去改动现成的代码。这样做，拓展性较好！