# 背景

## 工厂模式分类

工厂模式主要是为创建对象提供了接口。工厂模式按照《Java与模式》中的提法分为三类：  
1. 简单工厂模式(Simple Factory)  
2. 工厂方法模式(Factory Method)  
3. 抽象工厂模式(Abstract Factory)  
这三种模式从上到下逐步抽象，并且更具一般性。还有一种分类法，就是将简单工厂模式看为工厂方法模式的一种特例，两个归为一类。

# ****简单工厂模式****

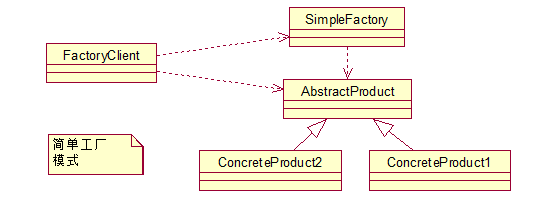
## 应用背景

这个模式本身很简单而且使用在业务较简单的情况下。一般用于小项目或者具体产品很少扩展的情况（这样工厂类才不用经常更改）。

比如：

话说十年前，有一个爆发户，他家有三辆汽车（Benz（奔驰）、Bmw（宝马）、Audi（奥迪）），还雇了司机为他开车。不过，爆发户坐车时总是这样：上Benz车后跟司机说“开奔驰车！”，坐上Bmw后他说“开宝马车！”，坐上 Audi后他说“开奥迪车！”。

它由三种角色组成：  
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，根据逻辑不同，产生具体的工厂产品。如例子中的Driver类。  
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。由接口或者抽象类来实现。如例中的Car接口。  
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，如例子中的Benz、Bmw类。  
来用类图来清晰的表示下的它们之间的关系：



下面就来给那个暴发户治病：在使用了简单工厂模式后，现在暴发户只需要坐在车里对司机说句：“开车”就可以了。

代码实现：（为方便起见，所有的类放在一个文件中，故有一个类被声明为public）

**//抽象产品**

**abstract** **class** Car{

**private** String name;

**public** **abstract** **void** drive();

**public** String getName() {

**return** name;

    }

**public** **void** setName(String name) {

**this**.name = name;

    }

}

**//具体产品**

**class** Benz **extends** Car{

**public** **void** drive(){

        System.out.println(**this**.getName()+"----go-----------------------");

    }

}

**class** Bmw **extends** Car{

**public** **void** drive(){

        System.out.println(**this**.getName()+"----go-----------------------");

    }

}

**//简单工厂**

**class** Driver{

**public** **static** Car createCar(String car){

        Car c = **null**;

**if**("Benz".equalsIgnoreCase(car))

            c = **new** Benz();

**else** **if**("Bmw".equalsIgnoreCase(car))

            c = **new** Bmw();

**return** c;

    }

}

**//老板**

**public** **class** BossSimplyFactory {

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

        //老板告诉司机我今天坐奔驰

        Car car = Driver.createCar("benz");

        car.setName("benz");

         //司机开着奔驰出发

        car.drive();

    }

## 简单工厂模式的局限性

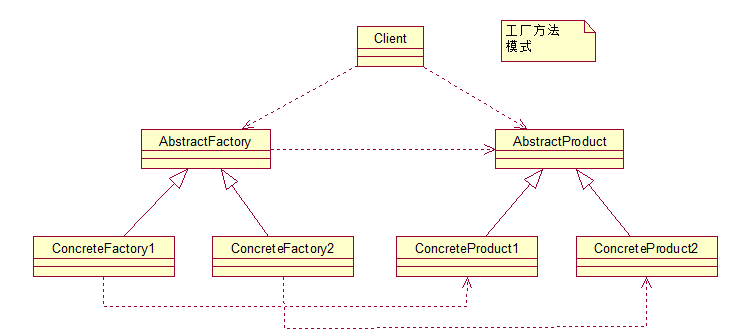
在实际应用中，很可能产品是一个多层次的树状结构。由于简单工厂模式中只有一个工厂类来对应这些产品，所以这可能会把我们的上帝类坏了。  
正如我前面提到的简单工厂模式适用于业务简单的情况下或者具体产品很少增加的情况。而对于复杂的业务环境可能不太适应了。这就应该由工厂方法模式来出场了。

**违反了开发封闭原则。**

# 工厂方法模式

## 组成

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。  
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。在java中它由具体的类来实现。  
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。  
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。  
来用类图来清晰的表示下的它们之间的关系：



 话说暴发户生意越做越大，自己的爱车也越来越多。这可苦了那位司机师傅了，什么车它都要记得，维护，都要经过他来使用！于是暴发户同情他说：我给你分配几个人手，你只管管好他们就行了！于是工厂方法模式的管理出现了。代码如下：

//抽象产品

**abstract** **class** Car{

**private** String name;

**public** **abstract** **void** drive();

**public** String getName() {

**return** name;

    }

**public** **void** setName(String name) {

**this**.name = name;

    }

}

//具体产品

**class** Benz **extends** Car{

**public** **void** drive(){

        System.out.println(**this**.getName()+"----go-----------------------");

    }

}

**class** Bmw **extends** Car{

**public** **void** drive(){

        System.out.println(**this**.getName()+"----go-----------------------");

    }

}

//抽象工厂

**abstract** **class** Driver{

**public** **abstract** Car createCar(String car) **throws** Exception;

}

//具体工厂（每个具体工厂负责一个具体产品）

**class** BenzDriver **extends** Driver{

**public** Car createCar(String car) **throws** Exception {

**return** **new** Benz();

    }

}

**class** BmwDriver **extends** Driver{

**public** Car createCar(String car) **throws** Exception {

**return** **new** Bmw();

    }

}

//老板

**public** **class** Boss{

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

        Driver d = **new** BenzDriver();

        Car c = d.createCar("benz");

        c.setName("benz");

        c.drive();

    }

}

## 优缺点

 使用**开闭原则**来分析下工厂方法模式。当有新的产品（即暴发户的汽车）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。（即当有新产品时，只要创建并基础抽象产品；新建具体工厂继承抽象工厂；而不用修改任何一个类）工厂方法模式是完全符合开闭原则的！

使用工厂方法模式足以应付我们可能遇到的大部分业务需求**。但是当产品种类非常多时，就会出现大量的与之对应的工厂类，这不应该是我们所希望的。**所以建议在这种情况下使用简单工厂模式与工厂方法模式相结合的方式来减少工厂类：即对于产品树上类似的种类（一般是树的叶子中互为兄弟的）使用简单工厂模式来实现。  
当然特殊的情况，就要特殊对待了：对于系统中存在不同的产品树，而且产品树上存在产品族（下一节将解释这个名词）。那么这种情况下就可能可以使用抽象工厂模式了。

# ****抽象工厂模式****

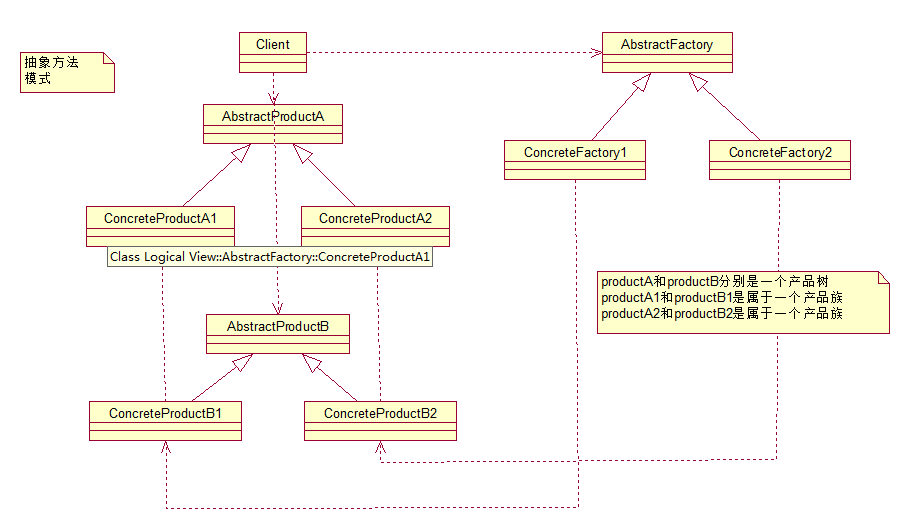
## 简介

产品族： 位于不同产品等级结构中，功能相关联的产品组成的家族。

图中的BmwCar和BenzCar就是两个产品树（产品层次结构）；而如图所示的BenzSportsCar和BmwSportsCar就是一个产品族。他们都可以放到跑车家族中，因此功能有所关联。同理BmwBussinessCar和BenzBusinessCar也是一个产品族。  
可以这么说，它和工厂方法模式的区别就在于需要创建对象的复杂程度上。而且抽象工厂模式是三个里面最为抽象、最具一般性的。抽象工厂模式的用意为：给客户端提供一个接口，可以创建多个产品族中的产品对象。  
而且使用抽象工厂模式还要满足一下条件：  
1.系统中有多个产品族，而系统一次只可能消费其中一族产品  
2.同属于同一个产品族的产品以其使用。

## 组成

来看看抽象工厂模式的各个角色（和工厂方法的如出一辙）：  
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。  
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。在java中它由具体的类来实现。  
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。  
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。



//抽象产品（Bmw和Audi同理）

**abstract** **class** BenzCar{

**private** String name;

**public** **abstract** **void** drive();

**public** String getName() {

**return** name;

    }

**public** **void** setName(String name) {

**this**.name = name;

    }

}

//具体产品（Bmw和Audi同理）

**class** BenzSportCar **extends** BenzCar{

**public** **void** drive(){

        System.out.println(**this**.getName()+"----BenzSportCar-----------------------");

    }

}

**class** BenzBusinessCar **extends** BenzCar{

**public** **void** drive(){

        System.out.println(**this**.getName()+"----BenzBusinessCar-----------------------");

    }

}

**abstract** **class** BmwCar{

**private** String name;

**public** **abstract** **void** drive();

**public** String getName() {

**return** name;

    }

**public** **void** setName(String name) {

**this**.name = name;

    }

}

**class** BmwSportCar **extends** BmwCar{

**public** **void** drive(){

        System.out.println(**this**.getName()+"----BmwSportCar-----------------------");

    }

}

**class** BmwBusinessCar **extends** BmwCar{

**public** **void** drive(){

        System.out.println(**this**.getName()+"----BmwBusinessCar-----------------------");

    }

}

**abstract** **class** AudiCar{

**private** String name;

**public** **abstract** **void** drive();

**public** String getName() {

**return** name;

    }

**public** **void** setName(String name) {

**this**.name = name;

    }

}

**class** AudiSportCar **extends** AudiCar{

**public** **void** drive(){

        System.out.println(**this**.getName()+"----AudiSportCar-----------------------");

    }

}

**class** AudiBusinessCar **extends** AudiCar{

**public** **void** drive(){

        System.out.println(**this**.getName()+"----AudiBusinessCar-----------------------");

    }

}

//抽象工厂

**abstract** **class** Driver3{

**public** **abstract** BenzCar createBenzCar(String car) **throws** Exception;

**public** **abstract** BmwCar createBmwCar(String car) **throws** Exception;

**public** **abstract** AudiCar createAudiCar(String car) **throws** Exception;

}

//具体工厂

**class** SportDriver **extends** Driver3{

**public** BenzCar createBenzCar(String car) **throws** Exception {

**return** **new** BenzSportCar();

    }

**public** BmwCar createBmwCar(String car) **throws** Exception {

**return** **new** BmwSportCar();

    }

**public** AudiCar createAudiCar(String car) **throws** Exception {

**return** **new** AudiSportCar();

    }

}

**class** BusinessDriver **extends** Driver3{

**public** BenzCar createBenzCar(String car) **throws** Exception {

**return** **new** BenzBusinessCar();

    }

**public** BmwCar createBmwCar(String car) **throws** Exception {

**return** **new** BmwBusinessCar();

    }

**public** AudiCar createAudiCar(String car) **throws** Exception {

**return** **new** AudiBusinessCar();

    }

}

//老板

**public** **class** BossAbstractFactory {

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

        Driver3 d = **new** BusinessDriver();

        AudiCar car = d.createAudiCar("");

        car.drive();

    }

}

其中：BenzSportCar和BenzBusinessCar属于产品树；同理BmwSportCar和BmwBusinessCar。而BenzSportCar和BmwSportCar和AudiSportCar属于产品族。

所以抽象工厂模式一般用于具有产品树和产品族的场景下。

## 抽象工厂模式的缺点

如果需要增加新的产品树，那么就要新增三个产品类，比如VolvoCar，VolvoSportCar,VolvoSportCar，并且要修改三个工厂类。这样大批量的改动是很丑陋的做法。

# 建造者模式

## 应用背景

在软件系统中，有时候面临着“一个复杂对象”的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法确相对稳定。如何应对这种变化？如何提供一种“封装机制”来隔离出“复杂对象的各个部分”的变化，从而保持系统中的 “稳定构建算法”不随着需求改变而改变？这就是要说的建造者模式。

## 定义

建造者模式：是将一个复杂的对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

## 组成

建造者模式通常包括下面几个角色：

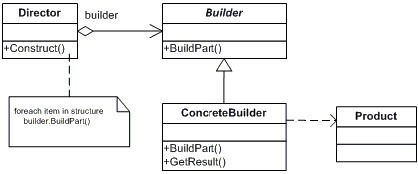
1. builder：给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。

2. ConcreteBuilder：实现Builder接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。 在建造过程完成后，提供产品的实例。

3. Director：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。

4. Product：要创建的复杂对象。

　　按照惯例，还是给出建造者模式的结构图



Product：

public class Food {

private String mFoodName;

private String mFoodTastes;

private String mFoodFeatures;

private String mChillies;

private String mSugar;

private String mSalt;

public void setFoodName(String name) {

mFoodName = name;

}

public String getFoodName() {

return mFoodName;

}

public void setFoodTastes(String tastes) {

mFoodTastes = tastes;

}

public String getFoodTastes() {

return mFoodTastes;

}

public void setFoodFeatures(String features) {

mFoodFeatures = features;

}

public String getFoodFeatures() {

return mFoodFeatures;

}

public String getChillies() {

return mChillies;

}

public void setChillies(String mChillies) {

this.mChillies = mChillies;

}

public String getSugar() {

return mSugar;

}

public void setSugar(String mSugar) {

this.mSugar = mSugar;

}

public String getSalt() {

return mSalt;

}

public void setSalt(String salt) {

mSalt = salt;

}

@Override

public String toString() {

return "Food Name: " + getFoodName() + "\r\n"

+ "Food Feature: " + getFoodFeatures() + "\r\n"

+ "Food Tastes: " + getFoodTastes() + "\r\n"

+ "Food Chillies: " + getChillies() + "\r\n"

+ "Food Salt: " + getSalt() + "\r\n"

+ "Food Sugar: " + getSugar();

}

}

上面菜肴中各种属性，比较简单，就是一个实体类。

**下面是Builder：**

public abstract class Builder {

protected Food mFood = new Food();

public abstract void setFoodName(String name);

public abstract void setFoodTastes(String tastes);

public abstract void setFoodFeature(String feature);

public abstract void addSugar(String sugar);

public abstract void addSalt(String salt);

public abstract void addChillies(String chillies);

public Food getResult() {

return mFood;

}

}

**实体Builder：**

public class ShandongCuisineChefBuilder extends Builder{

@Override

public void setFoodName(String name) {

mFood.setFoodName(name);

}

@Override

public void setFoodTastes(String tastes) {

mFood.setFoodTastes(tastes);

}

@Override

public void setFoodFeature(String feature) {

mFood.setFoodFeatures(feature);

}

@Override

public void addSugar(String sugar) {

mFood.setSugar(sugar);

}

@Override

public void addSalt(String salt) {

mFood.setSalt(salt);

}

@Override

public void addChillies(String chilies) {

mFood.setChillies(chilies);

}

}

public class SiChuanChefBuilder extends Builder {

@Override

public void setFoodName(String name) {

mFood.setFoodName(name);

}

@Override

public void setFoodTastes(String tastes) {

mFood.setFoodTastes(tastes);

}

@Override

public void setFoodFeature(String feature) {

mFood.setFoodFeatures(feature);

}

@Override

public void addSugar(String sugar) {

mFood.setSugar(sugar);

}

@Override

public void addSalt(String salt) {

mFood.setSalt(salt);

}

@Override

public void addChillies(String chilies) {

mFood.setChillies(chilies);

}

}

这两个大厨类写的有点多余，因为实现完全一样。

**director类：**

public class Director {

private Builder mBuilder;

public Director(){}

public Director(Builder builder) {

mBuilder = builder;

}

public void setDirector(Builder builder) {

mBuilder = builder;

}

public Food construct(String name, String feature, String tastes,

String chillies, String salt, String sugar) {

mBuilder.setFoodName(name);

mBuilder.setFoodFeature(feature);

mBuilder.setFoodTastes(tastes);

mBuilder.addChillies(chillies);

mBuilder.addSalt(salt);

mBuilder.addSugar(sugar);

return mBuilder.getResult();

}

}

client类：

public class Client {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

Builder shandongCuisineChef = new ShandongCuisineChefBuilder();

Builder siChuanChef = new SiChuanChefBuilder();

Director director = new Director(shandongCuisineChef);

Food food = director.construct("糖醋里脊", "色淡黄,形光润饱满.外松脆酥香,里软嫩鲜美", "又甜又香", "无", "适量", "加糖");

System.out.println("经典鲁菜： " + "\r\n" + food);

Director director2 = new Director();

director2.setDirector(siChuanChef);

Food food2 = director2.construct("水煮肉片", "肉味香辣，软嫩，易嚼", "麻、辣、鲜、烫", "多", "适量", "无");

System.out.println("经典川菜： " + "\r\n" + food2);

}

}

### 建造者模式的优点

* 封装性

      使用建造者模式可以使客户端不必知道产品内部组成的细节，如例子中我们就不需要关心每一个具体的模型内部是如何实现的，产生的对象类型就是CarModel。

* 建造者独立，容易扩展

      Builder之间是相互独立的，与其它的Builder无关，对系统的扩展非常有利。

* 便于控制细节风险

      模式所建造的最终产品更易于控制：由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他的模块产生任何影响。

### 建造者模式的使用场景

* 相同的方法，不同的执行顺序，产生不同的事件结果时，可以采用建造者模式，需要生成的产品对象的属性相互依赖，建造者模式可以强迫生成顺序。
* 多个部件或零件,都可以装配到一个对象中，但是产生的运行结果又不相同时，则可以使用该模式。
* 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效能，这个时候使用建造者模式是非常合适。
* 在对象创建过程中会使用到系统中的一些其它对象，这些对象在产品对象的创建过程中不易得到时，也可以采用建造者模式封装该对象的创建过程。该种场景，只能是一个补偿方法，因为一个对象不容易获得，而在设计阶段竟然没有发觉，而要通过创建者模式柔化创建过程，本身已经违反设计最初目标。

### 建造者模式的注意事项

      建造者模式关注的是的零件类型和装配工艺（顺序），这是它与工厂方法模式最大不同的地方，虽然同为创建类模式，但是注重点不同。

## 和其他模式的区别：

1. 与抽象工厂模式相比，建造者模式返回一个组装好的完整产品，而抽象工厂模式返回一系列相关的产品，这些产品位于不同的产品等级结构，构成了一个产品族。
2. 在抽象工厂模式中，客户端实例化工厂类，然后调用工厂方法获取所需产品对象，而在建造者模式中，客户端可以不直接调用建造者的相关方法，而是通过指挥者类来指导如何生成对象，包括对象的组装过程和建造步骤，它侧重于一步步构造一个复杂对象，返回一个完整的对象。
3. 如果将抽象工厂模式看成汽车配件生产工厂，生产一个产品族的产品，那么建造者模式就是一个汽车组装工厂，通过对部件的组装可以返回一辆完整的汽车。

## 在android中的应用

Dialog：Dialog的建造过程就是一个典型的“建造者模式”，通过Builder来设置不同的参数，创建出不同的Dialog。