[简单工厂、工厂方法、抽象工厂](http://www.cnblogs.com/zhangchenliang/p/3700820.html)

结合简单示例和UML图，讲解工厂模式简单原理。

**一、引子**

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！”。

你一定说：这人有病！直接说开车不就行了？！而当把这个爆发户的行为放到我们程序语言中来，我们发现C语言一直是通过这种方式来坐车的！

幸运的是这种有病的现象在OO语言中可以避免了。下面以Java语言为基础来引入我们本文的主题：工厂模式！

**二、简介**

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。两者皆可，这本为使用《Java与模式》的分类方法。  
在什么样的情况下我们应该记得使用工厂模式呢？大体有两点：  
1.在编码时不能预见需要创建哪种类的实例。  
2.系统不应依赖于产品类实例如何被创建、组合和表达的细节  
工厂模式能给我们的OOD、OOP(OOD:面向对象原则OOP:面向对象编程)带来哪些好处呢？？

**三、简单工厂模式**

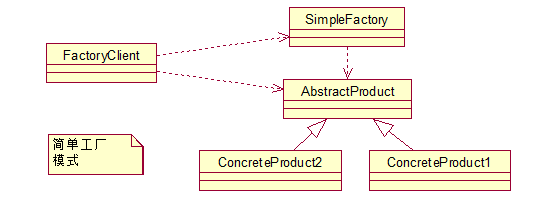
这个模式本身很简单而且使用在业务较简单的情况下。一般用于小项目或者具体产品很少扩展的情况（这样工厂类才不用经常更改）。  
它由三种角色组成：

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：

**客户端要Product，直接去抽象工厂拿就可以了。**

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。来看看怎么用代码实现的：（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）

Java代码  收藏代码

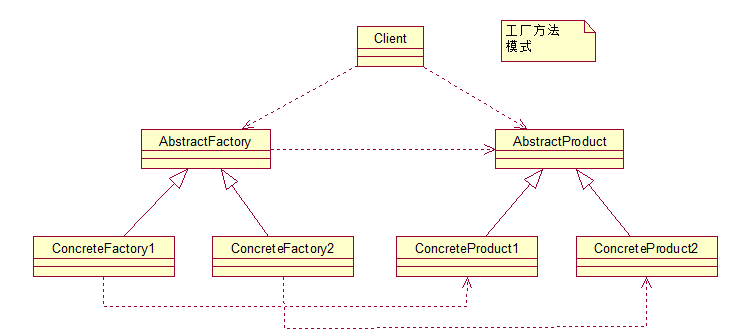
1. //抽象产品
2. **abstract** **class** Car{
3. **private** String name;
5. **public** **abstract** **void** drive();
7. **public** String getName() {
8. **return** name;
9. }
10. **public** **void** setName(String name) {
11. **this**.name = name;
12. }
13. }
14. //具体产品
15. **class** Benz **extends** Car{
16. **public** **void** drive(){
17. System.out.println(**this**.getName()+"----go-----------------------");
18. }
19. }
21. **class** Bmw **extends** Car{
22. **public** **void** drive(){
23. System.out.println(**this**.getName()+"----go-----------------------");
24. }
25. }
27. //简单工厂
28. **class** Driver{
29. **public** **static** Car createCar(String car){
30. Car c = **null**;
31. **if**("Benz".equalsIgnoreCase(car))
32. c = **new** Benz();
33. **else** **if**("Bmw".equalsIgnoreCase(car))
34. c = **new** Bmw();
35. **return** c;
36. }
37. }
39. //老板
40. **public** **class** BossSimplyFactory {
42. **public** **static** **void** main(String[] args) **throws** IOException {
43. //老板告诉司机我今天坐奔驰
44. Car car = Driver.createCar("benz");
45. car.setName("benz");
46. //司机开着奔驰出发
47. car.drive();
48. }
49. <span style="font-family: courier new,courier;">}</span>

如果老板要坐奥迪，同理。

这便是简单工厂模式了。那么它带了了什么好处呢？  
首先，符合现实中的情况；而且客户端免除了直接创建产品对象的责任，而仅仅负责“消费”产品（正如暴发户所为）。  
下面我们从开闭原则上来分析下简单工厂模式。当暴发户增加了一辆车的时候，只要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。（即创建一个新的车类，继承抽象产品Car）那么 对于产品部分来说，它是符合开闭原则的——对扩展开放、对修改关闭；但是工厂类不太理想，因为每增加一辆车，都要在工厂类中增加相应的商业逻辑和判 断逻辑，这显自然是违背开闭原则的。

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！！

**四、工厂方法模式（满足OCP原则）**  
**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：

Java代码  收藏代码

1. //抽象产品
2. **abstract** **class** Car{
3. **private** String name;
5. **public** **abstract** **void** drive();
7. **public** String getName() {
8. **return** name;
9. }
10. **public** **void** setName(String name) {
11. **this**.name = name;
12. }
13. }
14. //具体产品
15. **class** Benz **extends** Car{
16. **public** **void** drive(){
17. System.out.println(**this**.getName()+"----go-----------------------");
18. }
19. }
20. **class** Bmw **extends** Car{
21. **public** **void** drive(){
22. System.out.println(**this**.getName()+"----go-----------------------");
23. }
24. }

27. //抽象工厂
28. **abstract** **class** Driver{
29. **public** **abstract** Car createCar(String car) **throws** Exception;
30. }
31. **//具体工厂（每个具体工厂负责一个具体产品）**
32. **class** BenzDriver **extends** Driver{
33. **public** Car createCar(String car) **throws** Exception {
34. **return** **new** Benz();
35. }
36. }
37. **class** BmwDriver **extends** Driver{
38. **public** Car createCar(String car) **throws** Exception {
39. **return** **new** Bmw();
40. }
41. }
43. //老板
44. **public** **class** Boss{
46. **public** **static** **void** main(String[] args) **throws** Exception {
47. Driver d = **new** BenzDriver();
48. Car c = d.createCar("benz");
49. c.setName("benz");
50. c.drive();
51. }
52. }

  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，只要创建并集继承抽象产品；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。所以我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。

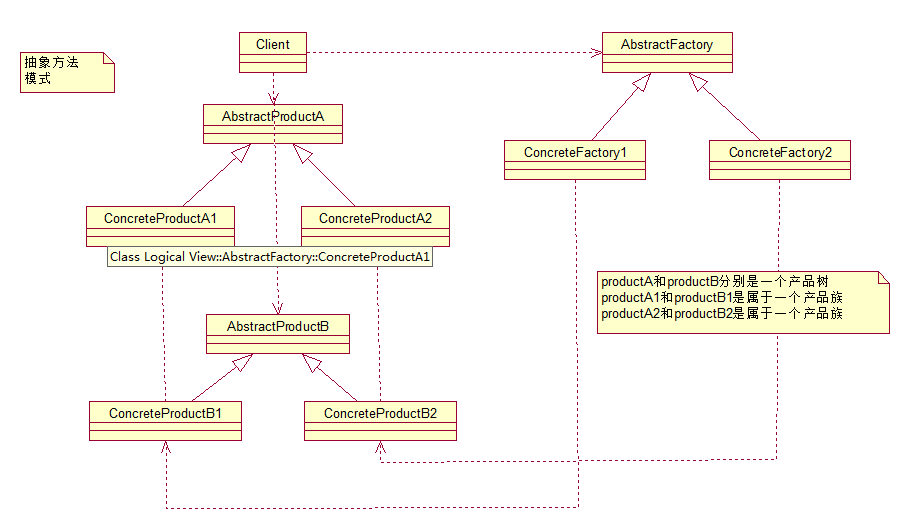
**五、小结**

让我们来看看简单工厂模式、工厂方法模式给我们的启迪：  
如果不使用工厂模式来实现我们的例子，也许代码会减少很多——**只需要实现已有的车，不使用多态。但是在可维护性上，可扩展性上是非常差的**（你可以想象一下添加一辆车后要牵动的类）。因此为了提高扩展性和维护性，多写些代码是值得的。

**六、抽象工厂模式**

先来认识下什么是产品族： 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。  
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。



Java代码  收藏代码

1. //抽象产品（Bmw和Audi同理）
2. **abstract** **class** BenzCar{
3. **private** String name;
5. **public** **abstract** **void** drive();
7. **public** String getName() {
8. **return** name;
9. }
10. **public** **void** setName(String name) {
11. **this**.name = name;
12. }
13. }
14. //具体产品（Bmw和Audi同理）
15. **class** BenzSportCar **extends** BenzCar{
16. **public** **void** drive(){
17. System.out.println(**this**.getName()+"----BenzSportCar-----------------------");
18. }
19. }
20. **class** BenzBusinessCar **extends** BenzCar{
21. **public** **void** drive(){
22. System.out.println(**this**.getName()+"----BenzBusinessCar-----------------------");
23. }
24. }
26. **abstract** **class** BmwCar{
27. **private** String name;
29. **public** **abstract** **void** drive();
31. **public** String getName() {
32. **return** name;
33. }
34. **public** **void** setName(String name) {
35. **this**.name = name;
36. }
37. }
38. **class** BmwSportCar **extends** BmwCar{
39. **public** **void** drive(){
40. System.out.println(**this**.getName()+"----BmwSportCar-----------------------");
41. }
42. }
43. **class** BmwBusinessCar **extends** BmwCar{
44. **public** **void** drive(){
45. System.out.println(**this**.getName()+"----BmwBusinessCar-----------------------");
46. }
47. }
49. **abstract** **class** AudiCar{
50. **private** String name;
52. **public** **abstract** **void** drive();
54. **public** String getName() {
55. **return** name;
56. }
57. **public** **void** setName(String name) {
58. **this**.name = name;
59. }
60. }
61. **class** AudiSportCar **extends** AudiCar{
62. **public** **void** drive(){
63. System.out.println(**this**.getName()+"----AudiSportCar-----------------------");
64. }
65. }
66. **class** AudiBusinessCar **extends** AudiCar{
67. **public** **void** drive(){
68. System.out.println(**this**.getName()+"----AudiBusinessCar-----------------------");
69. }
70. }

73. //抽象工厂
74. **abstract** **class** Driver3{
75. **public** **abstract** BenzCar createBenzCar(String car) **throws** Exception;
77. **public** **abstract** BmwCar createBmwCar(String car) **throws** Exception;
79. **public** **abstract** AudiCar createAudiCar(String car) **throws** Exception;
80. }
81. //具体工厂
82. **class** SportDriver **extends** Driver3{
83. **public** BenzCar createBenzCar(String car) **throws** Exception {
84. **return** **new** BenzSportCar();
85. }
86. **public** BmwCar createBmwCar(String car) **throws** Exception {
87. **return** **new** BmwSportCar();
88. }
89. **public** AudiCar createAudiCar(String car) **throws** Exception {
90. **return** **new** AudiSportCar();
91. }
92. }
93. **class** BusinessDriver **extends** Driver3{
94. **public** BenzCar createBenzCar(String car) **throws** Exception {
95. **return** **new** BenzBusinessCar();
96. }
97. **public** BmwCar createBmwCar(String car) **throws** Exception {
98. **return** **new** BmwBusinessCar();
99. }
100. **public** AudiCar createAudiCar(String car) **throws** Exception {
101. **return** **new** AudiBusinessCar();
102. }
103. }
105. //老板
106. **public** **class** BossAbstractFactory {
108. **public** **static** **void** main(String[] args) **throws** Exception {
110. Driver3 d = **new** BusinessDriver();
111. AudiCar car = d.createAudiCar("");
112. car.drive();
113. }
114. }

**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。

同理BmwBussinessCar和BenzBusinessCar也是一个产品族。

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（即不可以增加产品，不然会导致每个抽象工厂都要加入该产品。），(只可以增加产品族，直接新建一个抽象工厂即可。)这样大批量的改动是很丑陋的做法。