简单工厂、工厂方法、抽象工厂、策略模式、策略与工厂的区别

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 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：

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）

```
1.  //抽象产品
2.  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();
    }
<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
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：

```
1.  //抽象产品
2.  abstract class Car{
3.      private String name;
4.
5.      public abstract void drive();
6.
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. }
25.
26.
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. }
42.
43. //老板
44. public class Boss{
45.
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。

```
1.  //抽象产品（Bmw 和 Audi 同理）
2.  abstract class BenzCar{
3.      private String name;
4.
5.      public abstract void drive();
6.
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. }
25.
26. abstract class BmwCar{
27.     private String name;
28.
29.     public abstract void drive();
30.
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. }
48.
49. abstract class AudiCar{
50.     private String name;
51.
52.     public abstract void drive();
53.
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. }
71.
72.
73. //抽象工厂
74. abstract class Driver3{
75.     public abstract BenzCar createBenzCar(String car) throws Exception;
76.
77.     public abstract BmwCar createBmwCar(String car) throws Exception;
78.
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. }
104.
105. //老板
106. public class BossAbstractFactory {
107.
108.     public static void main(String[] args) throws Exception {
109.
```

```
        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。
所以可以用简单工厂配合反射来改进抽象工厂：
UML 图略。

```
abstract class BenzCar{
    private String name;

    public abstract void drive();

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
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) {
```

```
33.         this.name = name;
34.     }
35. }
36. class BmwSportCar extends BmwCar{
37.     public void drive(){
38.         System.out.println(this.getName()+"----BmwSportCar----------------------");
39.     }
40. }
41. class BmwBusinessCar extends BmwCar{
42.     public void drive(){
43.         System.out.println(this.getName()+"----BmwBusinessCar----------------------");
44.     }
45. }
46. 
47. abstract class AudiCar{
48.     private String name;
49. 
50.     public abstract void drive();
51. 
52.     public String getName() {
53.         return name;
54.     }
55.     public void setName(String name) {
56.         this.name = name;
57.     }
58. }
59. class AudiSportCar extends AudiCar{
60.     public void drive(){
61.         System.out.println(this.getName()+"----AudiSportCar----------------------");
62.     }
63. }
64. class AudiBusinessCar extends AudiCar{
65.     public void drive(){
66.         System.out.println(this.getName()+"----AudiBusinessCar----------------------");
67.     }
68. }
69. 
70. 
71. /**
72.  * 简单工厂通过反射改进抽象工厂及其子工厂
73.  * @author Administrator
74.  *
75.  */
76. class Driver3{
77.     public static BenzCar createBenzCar(String car) throws Exception {
78.         return (BenzCar) Class.forName(car).newInstance();
79.     }
```

```
80.
81.     public static BmwCar createBmwCar(String car) throws Exception {
82.         return (BmwCar) Class.forName(car).newInstance();
83.     }
84.
85.     public static AudiCar createAudiCar(String car) throws Exception {
86.         return (AudiCar) Class.forName(car).newInstance();
87.     }
88. }
89. //客户端
90. public class SimpleAndAbstractFactory {
91.
92.     public static void main(String[] args) throws Exception {
93.
94.         AudiCar car = Driver3.createAudiCar("com.java.pattendesign.factory.AudiSportCar");
95.         car.drive();
96.     }
97. }
```

策略模式

从策略一词来看,策略模式是种倾向于**行为**的模式.有点类似找仗时的做战方案，一般司令员在做战前都会根据实际情况做出几套不同的方案，如果当时情况有变，就会根据相应的条件来判定用哪一套方案来替换原定方案。但无论如何替换，替换多少次，仗还是要打的。

举例：导出成 EXCEL，WORD，PDF 文件的功能，这三类导出虽然具体操作略有不同，但是大部分都相同。

策略模式与工厂模式从 uml 图上来说，基本一致。只是强调的封装不同。我们以工厂模式和策略模式的比较来讲解策略模式。

工厂模式我们可以做如下理解：假设有 Audi 的公司生产汽车，它掌握一项核心的技术就是生产汽车，另一方面，它生产的汽车是有不同型号的，并且在不同的生产线上进行组装。当客户通过销售部门进行预定后，Audi 公司将在指定的生产线上为客户生产出它所需要的汽车。

策略(Strategy)模式在结构上与工厂模式类似，唯一的区别是工厂模式实例化一个产品的操作是在服务端来做的，换句话说客户端传达给服务端的只是某种标识，服务端根据该标识实例化一个对象。而策略模式的客户端传达给服务端的是一个实例，服务端只是将该实例拿过去在服务端的环境里执行该实例的方法。这就好比一个对汽车不甚了解的人去买车，他在那一比划，说要什么什么样的，销售部门根据他的这个"比划"来形成一份订单，这就是工厂模式下的工作方式。而策略模式下那个顾客就是个行家，他自己给出了订单的详细信息，销售部门只是转了一下手就交给生产部门去做了。通过两相对比，我们不难发现，采用工厂模式必须提供足够灵活的销售部门，如果用户有了新的需求，销售部门必须马上意识到这样才可以做出合适的订单。所以倘一款新车出来了，生产部门和销售部门都需要更新，对顾客来说也需要更新对新车的描述所以需要改动的地方有三处。而策略模式中的销售部门工作比较固定，它只负责接受订单并执行特定的几个操作。当一款新车出来时，只需要对服务端的生产部门和客户端的代码进行更新，而不需要更新销售部门的代码。

技术支持： 简单工厂和策略的基础都是因为面向对象的封装与多态。他们实现的思想都是先设定一个抽象的模型并从该模型派生出符合不同客户需求的各种方法，并加以封装。

工厂模式和策略模式的区别在于实例化一个对象的位置不同，对工厂模式而言，实例化对象是放在服务端的，即放在了工厂类里面；

而策略模式实例化对象的操作在客户端，服务端的"销售部门"只负责传递该对象，并在服务端的环境里执行特定的操作。。。

工厂模式要求服务端的销售部门足够灵敏，而策略模式由于对策略进行了封装，所以他的销售部门比较傻，需要客户提供足够能区分使用哪种策略的参数，而这最好的就是该策略的实例了。

```
1. //抽象产品
2. abstract class AudiCar{
3.     private String name;
4.
5.     public abstract void makeCar();
6.
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 AudiA6 extends AudiCar{
16.    public void makeCar(){
17.        System.out.println(this.getName()+"----go----------------------");
18.    }
19. }
20. class AudiA4 extends AudiCar{
21.    public void makeCar(){
22.        System.out.println(this.getName()+"----go----------------------");
23.    }
24. }
25.
26. //销售部门----服务端
27. class CarContext {
28.    AudiCar audiCar = null;
29.
30.    public CarContext(AudiCar audiCar) {
31.        this.audiCar = audiCar;
32.    }
33.
34.    public void orderCar(){
35.        this.audiCar.makeCar();
36.    }
37. }
38.
39. //客户----客户端（这个客户是内行，什么都懂，他说我要 A6，销售部门立刻给他 a6，所以销售部门不用很懂）
40. public class SimplyFactoryAndStrategy2 {
```

```
41.
42.     public static void main(String[] args) throws IOException {
43.
44.         //客户说我要什么什么样子的车子，销售人员才知道他要什么样子的车子
45.         AudiCar car = new AudiA6();
46.         car.setName("a6");
47.
48.         CarContext context = new CarContext(car);
49.         context.orderCar();
50.     }
51. }
52.
53.
54. //工厂模式---与上面的策略模式比较
55. //抽象产品
56. abstract class AudiCar{
57.     private String name;
58.
59.     public abstract void makeCar();
60.
61.     public String getName() {
62.         return name;
63.     }
64.     public void setName(String name) {
65.         this.name = name;
66.     }
67. }
68. //具体产品
69. class AudiA6 extends AudiCar{
70.     public void makeCar(){
71.         System.out.println(this.getName()+"----go----------------------");
72.     }
73. }
74. class AudiA4 extends AudiCar{
75.     public void makeCar(){
76.         System.out.println(this.getName()+"----go----------------------");
77.     }
78. }
79.
80. //简单工厂---销售部门----服务端
81. class CarFactroy{
82.     public static AudiCar createCar(String car){
83.         AudiCar c = null;
84.         if("A6".equalsIgnoreCase(car))
85.             c = new AudiA6();
86.         else if("A4".equalsIgnoreCase(car))
87.             c = new AudiA4();
```

```
88.         return c;
89.     }
90. }
91.
92. //客户----客户端（这个客户是外行，什么都不懂，只要随便描述下车，销售部门才能知道他要那款车，所以销售部门比较牛）
93. public class SimplyFactoryAndStrategy {
94.
95.     public static void main(String[] args) throws IOException {
96.
97.         System.out.print("请输入您要坐的车：（A6、A4）");
98.         String carName = new BufferedReader(new InputStreamReader(System.in)).readLine();
99.
100.         //客户说我要什么什么样子的车子，销售人员才知道他要什么样子的车子
101.         AudiCar car = CarFactroy.createCar(carName);
102.         car.setName(carName);
103.         car.makeCar();
104.     }
105. }
```

策略模式的优缺点

策略模式的主要优点有：

- 策略类之间可以自由切换，由于策略类实现自同一个抽象，所以他们之间可以自由切换。
- 易于扩展，增加一个新的策略对策略模式来说非常容易，基本上可以在不改变原有代码的基础上进行扩展。
- 避免使用多重条件，如果不使用策略模式，对于所有的算法，必须使用条件语句进行连接，通过条件判断来决定使用哪一种算法，在上一篇文章中我们已经提到，使用多重条件判断是非常不容易维护的。

策略模式的缺点主要有两个：

- 维护各个策略类会给开发带来额外开销，可能大家在这方面都有经验：一般来说，策略类的数量超过 5 个，就比较令人头疼了。
- 必须对 客户端（调用者）暴露所有的策略类，因为使用哪种策略是由客户端来决定的，因此，客户端应该知道有什么策略，并且了解各种策略之间的区别，否则，后果很严 重。例如，有一个排序算法的策略模式，提供了快速排序、冒泡排序、选择排序这三种算法，客户端在使用这些算法之前，是不是先要明白这三种算法的适用情况？ 再比如，客户端要使用一个容器，有链表实现的，也有数组实现的，客户端是不是也要明白链表和数组有什么区别？就这一点来说是有悖于迪米特法则的。

适用场景

做面向对象设计的，对策略模式一定很熟悉，因为它实质上就是面向对象中的继承和多态，在看完策略模式的通用代码后，我想，即使之前从来没有听说过策略模式，在开发过程中也一定使用过它吧？至少在在以下两种情况下，大家可以考虑使用策略模式，

- 几个类的主要逻辑相同，只在部分逻辑的算法和行为上稍有区别的情况。
- 有几种相似的行为，或者说算法，客户端需要动态地决定使用哪一种，那么可以使用策略模式，将这些算法封装起来供客户端调用。

策略模式是一种简单常用的模式，我们在进行开发的时候，会经常有意无意地使用它，一般来说，策略模式不会单独使用，跟模版方法模式、工厂模式等混合使用的情况比较多。

大粒度的 if --else if...可以使用 工厂+策略模式搞定。

分类: CSharp Technical

Danny Chen
关注 - 10
粉丝 - 204

« 上一篇：C#常见算法题目

» 下一篇：Javascript 闭包——懂不懂由你，反正我是懂了

posted @ 2014-04-30 10:49 Danny Chen 阅读(14501) 评论(6) 编辑 收藏

评论列表

#1 楼 2016-03-24 11:15 KevinGarnett

这个讲解真不错。

支持(0)反对(0)

#2 楼 2016-05-19 17:18 eversliver

讲的不错

支持(0)反对(0)

#3 楼 2016-05-24 16:01 tony_2016

写得非常不错，可是我有一个问题：

避免使用多重条件，如果不使用策略模式，对于所有的算法，必须使用条件语句进行连接，通过条件判断来决定使用哪一种算法，在上一篇文章中我们已经提到，使用多重条件判断是非常不容易维护的。

对于这段话你是如何理解的，我觉得在客户端写代码的时候尽管这些具体策略类的对外接口都是统一的，在具体的使用的时候还是需要使用条件判断来决定采用那种策略的啊。

谢谢

支持(0)反对(0)

#4 楼 2016-06-03 10:20 技术狩猎

@ tony_2016

作者写的不错。一个设计不是光要一种模式。是多个模式混合使用。

对于条件的判断，简单的后期不变的可以用 if 来处理，如果后期变动大。可以用一个叫命令模式来结合使用

支持(0)反对(0)

#5 楼 2016-06-08 16:03 tony_2016

哦，谢谢

支持(0)反对(0)

#6 楼 2016-07-12 14:50 jcxxxxx

抽象工厂里面的产品族概念搞错了，

同一个品牌下面的不同类别车构成一个产品族：

例如你举的例子：

BenzSportCar

BenzBusinessCar

以上两种车构成一个产品族。

增加产品族符合开闭原则，例如加一个长安汽车，很容易的；但是想要在产品族里面新增新的车，例如：加一个电动车，就需要更改抽象工厂类了，不符合开闭原则。

以上，

支持(0)反对(0)