|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | interface Car {// 车的父类      public void driver();// 开车  }  class Benz implements Car {// 奔驰车      public void driver() {          System.out.println("今天咱开奔驰！");      }  }  class Bike implements Car {// 自行车      public void driver() {          System.out.println("唉，现在经济危机，只能骑自行车了呀！");      }  }  class Bmw implements Car {// 宝马      public void driver() {          System.out.println("今天开宝马吧！");      }  }  class Driver {// 车的工厂      public static Car driverCar(String s) throws Exception {          if (s.equalsIgnoreCase("Benz")) {// 判断传入参数返回不同的实现类              return new Benz();          } else if (s.equalsIgnoreCase("Bmw")) {              return new Bmw();          } else if (s.equalsIgnoreCase("Bike")) {              return new Bike();          } else {              throw new Exception();// 抛出异常          }      }  }  public class TextSimpleFactory {// 操作工厂模式的类      public static void main(String[] args) {// java程序主入口处          try {              Car car = Driver.driverCar("Bike");// 调用方法返回车的实例              System.out.println("经理，今天开什么车呀？");              car.driver();// 调用方法开车          } catch (Exception e) {// 捕获异常              System.out.println("开车出现问题......");          } finally {// 代码总被执行              System.out.println("......");          }      }  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | package net.xsoftlab.baike;  interface fruit {      public abstract void eat();  }  class Apple implements fruit {      public void eat() {          System.out.println("Apple");      }  }  class Orange implements fruit {      public void eat() {          System.out.println("Orange");      }  }  class Factory {      public static fruit getInstance(String ClassName) {          fruit f = null;          try {              f = (fruit) Class.forName(ClassName).newInstance();          } catch (Exception e) {              e.printStackTrace();          }          return f;      }  } |

 \* 对于普通的工厂模式当我们在**添加一个子类**的时候，就需要对应的**修改工厂类**。 当我们添加很多的子类的时候，会很麻烦。

 \*

 \* 现在我们利用**反射机制实现工厂模式**，可以在不修改工厂类的情况下添加任意多个子类（只是有些情况可以这么做）。

 \*

 \* 但是有一点仍然很麻烦，就是需要知道**完整的包名和类名**，这里可以使用properties配置文件来完成。

但和IOC不同，仍在类和类之间有依赖，不存在第三方容器管理它们的关系。

简单工厂模式

如最上的一个例子

**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，这显然是违背开闭原则的。**

工厂方法模式（满足开闭，不需要改变工厂类源代码）

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。

**工厂方法模式使得客户程序中仅仅处理抽象产品角色提供的接口。那我们是否一定要在代码中遍布工厂呢？大可不必。也许在下面情况下你可以考虑使用工厂方法模式：   
     1)当客户程序不需要知道要使用对象的创建过程。 （屏蔽创建过程）  
     2)客户程序使用的对象存在变动的可能，或者根本就不知道使用哪一个具体的对象**

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");

}

}

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();

}

}

interface FactoryBMW {

BMW createBMW();

}

————————————————————

/\*\*

 \*

 \* 客户得到车

 \*/

class Customer {

   private $BMW;

   function  getBMW($type){

      switch ($type) {

        case 320:

           $BWM320 = new FactoryBWM320();

           return $BWM320->createBMW();

        case 523:

           $BWM523 = new FactoryBWM523();

           return $BWM320->createBMW();

            //....

      }

  }

}

抽象工厂模式

（抽象工厂模式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，多了空调）

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。 抽象工厂模式的各个角色（和工厂方法一样）：

3)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。

4)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。

5)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。

6)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。

/\*\*

 \* 创建工厂的接口

 \*

 \*/

interface FactoryBMW {

       function createBMW();

}

  /\*\*

 \*

 \* 创建BWM320车

 \*/

class FactoryBWM320 implements FactoryBMW {

   function  createBMW($type){

      return new BWM320();

   }

  }

    /\*\*

 \*

 \* 创建BWM523车

 \*/

class FactoryBWM523 implements FactoryBMW {

   function  createBMW($type){

      return new BMW523();

   }

}

/\*\*

 \* 创建工厂的接口

 \*

 \*/

interface FactoryBMW {

     function createBMW();

     function createAirC();

}

/\*\*

 \*

 \* 创建BWM320车

 \*/

class FactoryBWM320 implements FactoryBMW {

    function  createBMW(){

    return new BWM320();

}

function  createAirC(){ //空调

    return new airconditionBWM320();

}

}

/\*\*

 \*

 \* 创建BWM523车

 \*/

class FactoryBWM523 implements FactoryBMW {

    function  createBMW(){

    return new BWM523();

}

function  createAirC(){

    return new airconditionBWM523();

}

}

/\*\*

 \*

 \* 客户得到车

 \*/

class Customer {

   private $BMW;

   private $airC;

   function  getBMW($type){

       $class = new ReflectionClass('FactoryBWM' .$type );//建立 Person这个类的反射类

        $instance  = $class->newInstanceArgs();//相当于实例化Person 类

        $this->BMW =  $instance->createBMW();

       $this->airC =  $instance->createAirC();

   }

}