作者： AI92 yuanyk@gmail . com

工厂模式 (静态工厂模式、 工厂方法模式、 抽象工厂模式)

一、 引子

话说十年前， 有一个暴发户， 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， 暴发户坐车时总是怪怪的： 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 说“开奥迪车！ ”。 你一定说： 这人有病！ 直接说
开车不就行了？ ！

而当把这个暴发户的行为放到我们程序设计中来时， 会发现这是一个普遍存在的现象。

幸运的是， 这种有病的现象在 OO (面向对象) 语言中可以避免了。 下面就以 Java 语言为
基础来引入我们本文的主题： 工厂模式。

二、 分类

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。

工厂模式在《Java 与模式》 中分为三类：

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。

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。

两者皆可， 在本文使用《Java 与模式》 的分类方法。 下面来看看这些工厂模式是怎么
来“治病”的。

三、 简单工厂模式

简单工厂模式又称静态工厂方法模式。 重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。 在j ava中它往往由
一个具体类实现。

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。 在j ava中由接口或者抽象类来实现。

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j ava中由一个具体类实现。

来用类图来清晰的表示下的它们之间的关系(如果对类图不太了解, 请参考我关于类图的文章):

那么简单工厂模式怎么来使用呢? 我们就以简单工厂模式来改造暴发户坐车的方式——现在暴发户只需要坐在车里对司机说句: “开车”就可以了。

```
/ / 抽象产品角色
public interface Car{
public void drive() ;
}
/ / 具体产品角色
public class Benz implements Car{
public void drive() {
System. out. println("Driving Benz ") ;
}
}
public class Bmw implements Car{
public void drive() {
System. out. println("Driving Bmw ") ;
}
}
```

。 。 。 (奥迪我就不写了: P)

```
/ / 工厂类角色
public class Driver{
/ / 工厂方法. 注意 返回类型为抽象产品角色
public static Car driverCar(String s) throws Exception {
/ / 判断逻辑, 返回具体的产品角色给 Client
if(s. equalsIgnoreCase("Benz") )
```

```
return  new  Benz() ;
else  if(s. equalsIgnoreCase("Bmw") )
return  new  Bmw() ;
. . . . . .
else  throw  new  Exception() ;
。 。 。
/ / 欢迎暴发户出场. . . . . .
public  class  Magnate{
public  static  void  main(String[]  args) {
try{
/ / 告诉司机我今天坐奔驰
Car  car  =  Driver. driverCar("benz") ;
/ / 下命令: 开车
car. drive() ;
。 。 。
```

将本程序空缺的其他信息填充完整后即可运行。 如果你将所有的类放在一个文件中， 请不要忘记只能有一个类被声明为 public。 本程序在 j dk1 . 4 下运行通过。

程序中各个类的关系表达如下:

这便是简单工厂模式了。 怎么样， 使用起来很简单吧? 那么它带来了什么好处呢?

首先， 使用了简单工厂模式后， 我们的程序不在“有病”， 更加符合现实中的情况; 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（对扩展开放; 对修改封闭） 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。 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， 因为每增加一辆车， 都要在工厂类中增加相应的业务逻辑或者判断逻辑， 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（在我们的例子中是为司机师傅）， 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。

四、 工厂方法模式

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。

你应该大致猜出了工厂方法模式的结构， 来看下它的组成：

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 j ava 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 j ava 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 j ava 中由具体的类来实现。

用类图来清晰的表示下的它们之间的关系：

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（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。 可以看出工厂角色的结构也是符合开闭原则的！

我们还是老规矩， 使用一个完整的例子来看看工厂模式各个角色之间是如何来协调的。

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： 看你跟我这么多年的份上， 以后你不用这么辛苦了， 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：

```
/ / 抽象产品角色， 具体产品角色与简单工厂模式类似， 只是变得复杂了些， 这里略。
/ / 抽象工厂角色
public  interface  Driver{
```

```
public  Car  driverCar() ;
}
public  class  BenzDriver  implements  Driver{
public  Car  driverCar() {
return  new  Benz() ;
}
}
public  class  BmwDriver  implements  Driver{
public  Car  driverCar()   {
return  new  Bmw() ;
}
}
/ / 应该和具体产品形成对应关系. . .
/ / 有请暴发户先生
public  class  Magnate
{
public  static  void  main(String[]   args)
{
try{
Driver  driver  =   new  BenzDriver() ;
Car  car  =   driver. driverCar() ;
car. drive() ;
}
……
}
```

可以看出工厂方法的加入， 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。 因为如果不能避免这种情况， 可以考虑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。

五、 小结

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。 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。

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。

简单工厂模式与工厂方法模式真正的避免了代码的改动了？ 没有。 在简单工厂模式中，新产品的加入要修改工厂角色中的判断语句； 而在工厂方法模式中， 要么将判断逻辑留在抽象工厂角色中， 要么在客户程序中将具体工厂角色写死（就象上面的例子一样）。 而且产品对象创建条件的改变必然会引起工厂角色的修改。

面对这种情况， Java 的反射机制与配置文件的巧妙结合突破了限制——这在 Spring 中完美的体现了出来。

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。

还是让我们用一个例子来形象地说明一下吧。

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 BenzSportsCar 也是一个产品族。

回到抽象工厂模式的话题上。

可以说， 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。 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）:

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 j ava 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。 在 j ava 中它由具体的类来实现。

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 j ava 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 j ava 中由具体的类来实现。

类图如下:

看过了前两个模式, 对这个模式各个角色之间的协调情况应该心里有个数了, 我就不举具体的例子了。 只是一定要注意满足使用抽象工厂模式的条件哦。

单例模式

一、 引子

单例模式是设计模式中使用很频繁的一种模式, 在各种开源框架、 应用系统中多有应用, 在我前面的几篇文章中也结合其它模式使用到了单例模式。 这里我们就单例模式进行系统的学习。 并对有人提出的“单例模式是邪恶的”这个观点进行了一定的分析。

二、 定义与结构

单例模式又叫做单态模式或者单件模式。 在 GOF 书中给出的定义为: 保证一个类仅有一个实例, 并提供一个访问它的全局访问点。 单例模式中的“单例”通常用来代表那些本质上具有唯一性的系统组件(或者叫做资源)。 比如文件系统、 资源管理器等等。

单例模式的目的就是要控制特定的类只产生一个对象, 当然也允许在一定情况下灵活的改变对象的个数。 那么怎么来实现单例模式呢? 一个类的对象的产生是由类构造函数来完成的, 如果想限制对象的产生, 一个办法就是将构造函数变为私有的(至少是受保护的), 使得外面的类不能通过引用来产生对象; 同时为了保证类的可用性, 就必须提供一个自己的对象以及访问这个对象的静态方法。

现在对单例模式有了大概的了解了吧, 其实单例模式在实现上是非常简单的——只有一个角色, 而客户则通过调用类方法来得到类的对象。

放上一个类图吧, 这样更直观一些:

S ingle ton

ins tanc e : S ingle ton

S ingle ton()

getIns tanc e ()

create

单例模式可分为有状态的和无状态的。 有状态的单例对象一般也是可变的单例对象， 多个单态对象在一起就可以作为一个状态仓库一样向外提供服务。 没有状态的单例对象也就是不变单例对象， 仅用做提供工具函数。

三、 实现

在单例模式的实现上有几种不同的方式， 我在这里将一一讲解。 先来看一种方式， 它在《j ava 与模式》 中被称为饿汉式。

```
public class Singleton {
/ / 在自己内部定义自己一个实例
/ / 注意这是 private 只供内部调用
private static Singleton instance = new Singleton() ;
/ / 如上面所述， 将构造函数设置为私有
private Singleton() {
}
/ / 静态工厂方法， 提供了一个供外部访问得到对象的静态方法
public static Singleton getInstance() {
return instance;
}
}
```

下面这种方式被称为懒汉式: P

```
public class Singleton {
/ / 和上面有什么不同?
private static Singleton instance = null;
/ / 设置为私有的构造函数
private Singleton() {
}
/ / 静态工厂方法
public static synchronized Singleton getInstance() {
/ / 这个方法比上面有所改进
```

```
if (instance= = null)
instance＝new Singleton() ;
return instance;
}
}
```

先让我们来比较一下这两种实现方式。

首先他们的构造函数都是私有的， 彻底断开了使用构造函数来得到类的实例的通道， 但是这样也使得类失去了多态性（大概这就是为什么有人将这种模式称作单态模式）。

在第二种方式中， 对静态工厂方法进行了同步处理， 原因很明显——为了防止多线程环境中产生多个实例； 而在第一种方式中则不存在这种情况。

在第二种方式中将类对自己的实例化延迟到第一次被引用的时候。 而在第一种方式中则是在类被加载的时候实例化， 这样多次加载会照成多次实例化。 但是第二种方式由于使用了同步处理， 在反应速度上要比第一种慢一些。

在 《j ava 与模式》 书中提到， 就 j ava 语言来说， 第一种方式更符合 j ava 语言本身的特点。

以上两种实现方式均失去了多态性， 不允许被继承。 还有另外一种灵活点的实现， 将构造函数设置为受保护的， 这样允许被继承产生子类。 这种方式在具体实现上又有所不同， 可以将父类中获得对象的静态方法放到子类中再实现； 也可以在父类的静态方法中进行条件判断来决定获得哪一个对象； 在 GOF 中认为最好的一种方式是维护一张存有对象和对应名称的注册表（可以使用 HashMap 来实现）。 下面的实现参考《j ava 与模式》 采用带有注册表的方式。

```
import j ava. util. HashMap;
public class Singleton
{
/ / 用来存放对应关系
private static HashMap sinRegistry = new HashMap() ;
static private Singleton s = new Singleton() ;
/ / 受保护的构造函数
protected Singleton()
```

```
{ }
public  static  Singleton  getInstance(String  name)
{
if(name  = =   null)
name  =   "Singleton";
if(sinRegistry. get(name) = = null)
{
try{
sinRegistry. put(name  ,   Class. forName(name) . newInstance() ) ;
} catch(Exception  e)
{
e. printStackTrace() ;
}
}
return  (Singleton) (sinRegistry. get(name) ) ;
}
public  void  test()
{
System. out. println("getclasssuccess! ") ;
}
}
public  class  SingletonChild1   extends  Singleton
{
public  SingletonChild1 () { }
static    public  SingletonChild1   getInstance()
{
return  (SingletonChild1 ) Singleton. getInstance("SingletonChild1 ") ;
}
public  void  test()
```

{

System. out. println("getclasssuccess1 1 1 ! ") ;

}

}

由于在 j ava 中子类的构造函数的范围不能比父类的小， 所以可能存在不守规则的客户程序使用其构造函数来产生实例， 造成单例模式失效。

四、 单例模式邪恶论

看这题目也许有点夸张， 不过这对初学者是一个很好的警告。 单例模式在 j ava 中的使用存在很多陷阱和假象， 这使得没有意识到单例模式使用局限性的你在系统中布下了隐患……

其实这个问题早在 2001 年的时候就有人在网上系统的提出来过， 我在这里只是老生常谈了。 但是对于大多的初学者来说， 可能这样的观点在还很陌生。 下面我就一一列举出单例模式在 j ava 中存在的陷阱。

多个虚拟机

当系统中的单例类被拷贝运行在多个虚拟机下的时候， 在每一个虚拟机下都可以创建一个实例对象。 在使用了 EJB、 JINI、 RMI 技术的分布式系统中， 由于中间件屏蔽掉了分布式系统在物理上的差异， 所以对你来说， 想知道具体哪个虚拟机下运行着哪个单例对象是很困难的。

因此， 在使用以上分布技术的系统中， 应该避免使用存在状态的单例模式， 因为一个有状态的单例类， 在不同虚拟机上， 各个单例对象保存的状态很可能是不一样的， 问题也就随之产生。 而且在 EJB 中不要使用单例模式来控制访问资源， 因为这是由 EJB 容器来负责的。在其它的分布式系统中， 当每一个虚拟机中的资源是不同的时候， 可以考虑使用单例模式来进行管理。

多个类加载器

当存在多个类加载器加载类的时候， 即使它们加载的是相同包名， 相同类名甚至每个字节都完全相同的类， 也会被区别对待的。 因为不同的类加载器会使用不同的命名空间（namespace） 来区分同一个类。 因此， 单例类在多加载器的环境下会产生多个单例对象。

也许你认为出现多个类加载器的情况并不是很多。 其实多个类加载器存在的情况并不少见。 在很多 J2EE 服务器上允许存在多个 servlet 引擎， 而每个引擎是采用不同的类加载器的;

浏览器中 applet 小程序通过网络加载类的时候， 由于安全因素， 采用的是特殊的类加载器，等等。

这种情况下， 由状态的单例模式也会给系统带来隐患。 因此除非系统由协调机制， 在一般情况下不要使用存在状态的单例模式。

错误的同步处理

在使用上面介绍的懒汉式单例模式时， 同步处理的恰当与否也是至关重要的。 不然可能会达不到得到单个对象的效果， 还可能引发死锁等错误。 因此在使用懒汉式单例模式时一定要对同步有所了解。 不过使用饿汉式单例模式就可以避免这个问题。

子类破坏了对象控制

在上一节介绍最后一种扩展性较好的单例模式实现方式的时候， 就提到， 由于类构造函数变得不再私有， 就有可能失去对对象的控制。 这种情况只能通过良好的文档来规范。

串行化（可序列化）

为了使一个单例类变成可串行化的， 仅仅在声明中添加“implements Serializable”是不够的。 因为一个串行化的对象在每次返串行化的时候， 都会创建一个新的对象， 而不仅仅是一个对原有对象的引用。 为了防止这种情况， 可以在单例类中加入 readResolve 方法。 关于这个方法的具体情况请参考《Effective Java》 一书第 57 条建议。

其实对象的串行化并不仅局限于上述方式， 还存在基于 XML 格式的对象串行化方式。 这种方式也存在上述的问题， 所以在使用的时候要格外小心。

上面罗列了一些使用单例模式时可能会遇到的问题。 而且这些问题都和 j ava 中的类、线程、 虚拟机等基础而又复杂的概念交织在一起， 你如果稍不留神……。 但是这并不代表着单例模式就一无是处， 更不能一棒子将其打死。 它还是不可缺少的一种基础设计模式， 它对一些问题提供了非常有效的解决方案， 在 j ava 中你完全可以把它看成编码规范来学习， 只是使用的时候要考虑周全些就可以了。

五、 题外话

抛开单例模式， 使用下面一种简单的方式也能得到单例， 而且如果你确信此类永远是单例的， 使用下面这种方式也许更好一些。

```
public static final Singleton INSTANCE = new Singleton() ;
```

而使用单例模式提供的方式， 这可以在不改变 API 的情况下， 改变我们对单例类的具体要求。

建造模式

一、 引子

前几天陪朋友去装机店攒了一台电脑， 看着装机工在那里熟练的装配着机器， 不禁想起来了培训时讲到的建造模式。 作为装机工， 他们不用管你用的 CPU 是 Intel 还是 AMD， 也不管你的显卡是 2000 千大元还是白送的， 都能三下五除二的装配在一起——一台 PC 就诞生了！ 当然对于客户来说， 你也不知道太多关于 PC 组装的细节。 这和建造模式是多么的相像啊！

今天就来探讨一下建造模式

二、 定义与结构

GOF 给建造模式的定义为： 将一个复杂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。 这句话说得很抽象， 不好理解， 其实它的意思可以理解为： 将构造复杂对象的过程和组成对象的部件解耦。 就像攒电脑一样， 不管什么品牌的配件， 只要兼容就可以装上； 同样， 一样的配件， 可以有好多组装的方式。 这是对降低耦合、 提高可复用性精神的一种贯彻。

当要生成的产品有复杂的内部结构——比如由多个对象组成； 而系统中对此产品的需求将来可能要改变产品对象的内部结构的构成， 比如说产品的一些属性现在由一个小对象组成， 而更改后的型号可能需要 N 个小对象组成； 而且不能将产品的内部构造完全暴露给客户程序， 一是为了可用性， 二是为了安全等因素。 满足上面的设计环境就可以考虑使用建造模式来搭建框架了。

来看看建造模式的组成吧。

1 ） 抽象建造者角色： 这个角色用来规范产品对象的各个组成成分的建造。 一般而言， 此角色独立于应用程序的业务逻辑。

2） 具体建造者角色： 担任这个角色的是于应用程序紧密相关的类， 它们在指导者的调用下创建产品实例。 这个角色在实现抽象建造者角色提供的方法的前提下， 达到完成产品组装， 提供成品的功能。

3） 指导者角色： 调用具体建造者角色以创建产品对象。 指导者并没有产品类的具体知识， 真正拥有产品类的具体知识的是具体建造者对象。

4） 产品角色： 建造中的复杂对象。 它要包含那些定义组件的类， 包括将这些组件装配成产品的接口。

来看下这些角色组成的类图:

Concre te Builde r

buildP art1 ()

buildP artn()

retrie ve Re s ult()

Product

Dire ctor

Dire c tor()

c ons truct()

Builder

buildPart1()

buildPartn()

retrieveResult()

builder

cre ate

首先客户程序创建一个指导者对象， 一个建造者角色， 并将建造者角色传入指导者对象进行配置。 然后， 指导者按照步骤调用建造者的方法创建产品。 最后客户程序从建造者或者指导者那里得到产品。

从建造模式的工作流程来看， 建造模式将产品的组装“外部化”到了建造者角色中来。 这是和任何正规的工厂模式不一样的——产品的创建是在产品类中完成的。

三、 实现

没有找到好的例子， 觉得《j ava 与模式》中发邮件的例子有点牵强。 于是我将 Bruce Eckel 在《Think in Patterns with Java》 中用的例子放到这里权且充个门面。 我们知道媒体可以存在不同的表达形式， 比如书籍、 杂志和网络。 这个例子表示不同形式的媒体构造的步骤是相似的， 所以可以被提取到指导者角色中去。

```
import j ava. util. *;
import j unit. framework. *;
/ / 不同的媒体形式:
class Media extends ArrayList { }
```

```
class Book extends Media {}
class Magazine extends Media {}
class WebSite extends Media {}
// 进而包含不同的媒体组成元素:
class MediaItem {
private String s;
public MediaItem(String s) { this.s = s; }
public String toString() { return s; }
}
class Chapter extends MediaItem {
public Chapter(String s) { super(s); }
}
class Article extends MediaItem {
public Article(String s) { super(s); }
}
class WebItem extends MediaItem {
public WebItem(String s) { super(s); }
}
// 抽象建造者角色, 它规范了所有媒体建造的步骤:
class MediaBuilder {
public void buildBase() {}
public void addMediaItem(MediaItem item) {}
public Media getFinishedMedia() { return null; }
}
// 具体建造者角色
class BookBuilder extends MediaBuilder {
private Book b;
public void buildBase() {
System.out.println("Building book framework");
```

```
b = new Book() ;
}
public void addMediaItem(MediaItem chapter) {
System. out. println("Adding chapter " + chapter) ;
b. add(chapter) ;
}
public Media getFinishedMedia() { return b; }
}
class MagazineBuilder extends MediaBuilder {
private Magazine m;
public void buildBase() {
System. out. println("Building magazine framework") ;
m = new Magazine() ;
}
public void addMediaItem(MediaItem article) {
System. out. println("Adding article " + article) ;
m. add(article) ;
}
public Media getFinishedMedia() { return m; }
}
class WebSiteBuilder extends MediaBuilder {
private WebSite w;
public void buildBase() {
System. out. println("Building web site framework") ;
w = new WebSite() ;
}
public void addMediaItem(MediaItem webItem) {
System. out. println("Adding web item " + webItem) ;
w. add(webItem) ;
```

```
}
public Media getFinishedMedia() { return w; }
}
// 指导者角色, 也叫上下文
class MediaDirector {
private MediaBuilder mb;
public MediaDirector(MediaBuilder mb) {
this.mb = mb; // 具有策略模式相似特征的
}
public Media produceMedia(List input) {
mb.buildBase();
for(Iterator it = input.iterator(); it.hasNext(); )
mb.addMediaItem((MediaItem) it.next() );
return mb.getFinishedMedia();
}
}
// 测试程序——客户程序角色
public class BuildMedia extends TestCase {
private List input = Arrays.asList(new MediaItem[] {
new MediaItem("item1 "), new MediaItem("item2"),
new MediaItem("item3"), new MediaItem("item4"),
} );
public void testBook() {
MediaDirector buildBook = new MediaDirector(new BookBuilder() );
Media book = buildBook.produceMedia(input);
String result = "book: " + book;
System.out.println(result);
assertEquals(result, "book: [item1 , item2, item3, item4] ");
}
```

```
public void testMagazine() {
MediaDirector buildMagazine = new MediaDirector(new MagazineBuilder() ) ;
Media magazine = buildMagazine. produceMedia(input) ;
```