第 13 章　工厂方法（Factory Method）模式

工厂方法模式是类的创建模式，又叫做虚拟构造子（Virtual Constructor）模式或者多态性工厂（Polymorphic Factory）模式。

工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。

13.1　引　　言

在阅读本章之前，请首先阅读本书的“简单工厂（Simple Factory）模式”一章。

简单工厂模式的优缺点

正如本书在“简单工厂（Simple Factory）模式”一章里介绍过的，工厂模式有简单工厂模式、工厂方法模式和抽象工厂模式几种。

在简单工厂模式中，一个工厂类处于对产品类实例化的中心位置上，它知道每一个产品，它决定哪一个产品类应当被实例化。这个模式的优点是允许客户端相对独立于产品创建的过程，并且在系统引入新产品的时候无需修改客户端，也就是说，它在某种程度上支持“开-闭”原则。

这个模式的缺点是对“开-闭”原则的支持不够，因为如果有新的产品加入到系统中去，就需要修改工厂类，将必要的逻辑加入到工厂类中。

工厂方法模式的引进

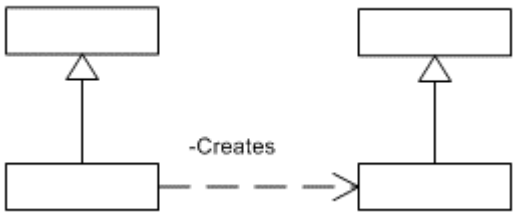
而本章要讨论的工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。

首先，在工厂方法模式中，核心的工厂类不再负责所有的产品的创建，而是将具体创建的工作交给子类去做。这个核心类则摇身一变，成为了一个抽象工厂角色，仅负责给出具体工厂子类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

这种进一步抽象化的结果，使这种工厂方法模式可以用来允许系统在不修改具体工厂

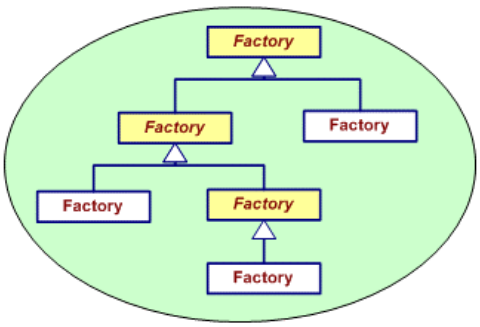
角色的情况下引进新的产品，这一特点无疑使得工厂模式具有超过简单工厂模式的优越性。

下图所示是工厂方法模式的简略类图，这个类图中仅显示了一个工厂类和一个产品类，而在实际系统里面，会遇到多个产品类以及相应的工厂类。

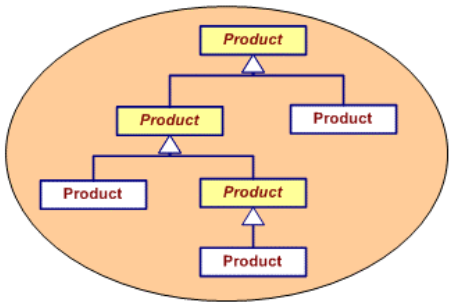


平行的等级结构

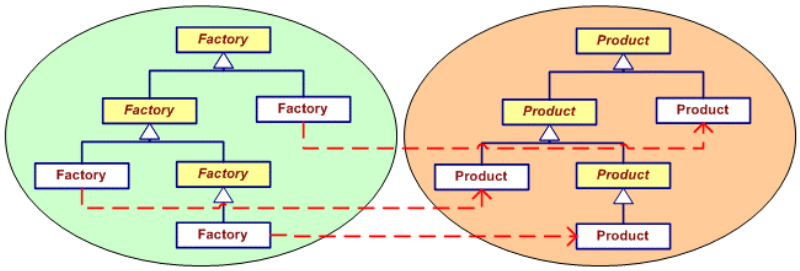
在一个系统设计中，常常是首先有产品角色，然后有工厂角色。在可以应用工厂方法模式的情形下，一般都会有一个产品的等级结构，由一个（甚至多个）抽象产品和多个具体产品组成。产品的等级结构如下图所示，树图中有阴影的是树枝型节点。



在上面的产品等级结构中，出现了多于一个的抽象产品类，以及多于两个的层次。这其实是真实的系统中常常出现的情况。当将工厂方法模式应用到这个系统中去的时候，常常采用的一个做法是按照产品的等级结构设计一个同结构的工厂等级结构。工厂的等级结构如下图所示，树图中有阴影的是树枝型节点。



然后由相应的工厂角色创建相应的产品角色，工厂方法模式的应用如下图所示，图中的虚线代表创建（依赖）关系。

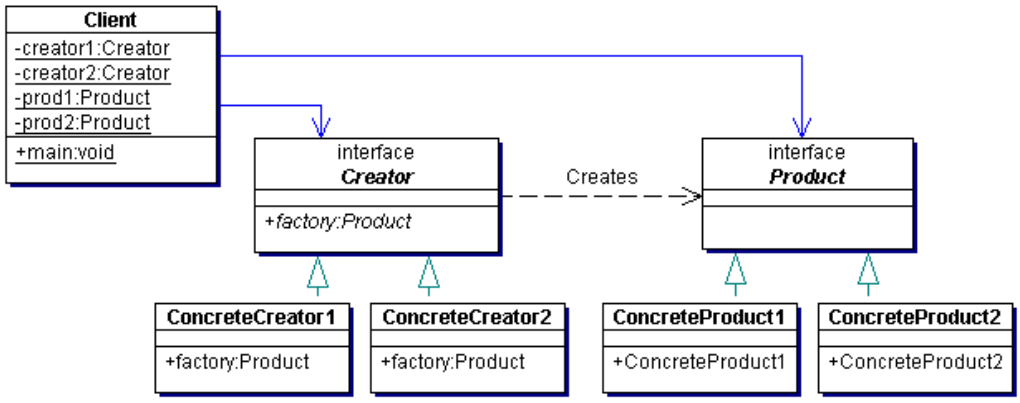


工厂方法模式并没有限制产品等级结构的层数。一般的书籍中都以两个层次为例，第一层是抽象产品层，第二层是具体产品层。但是在实际的系统中，产品常常有更为复杂的层次。

13.2　工厂方法模式的结构

结构与角色

为了说明工厂方法模式的结构，下面以一个最为简单的情形为例。这个示意性系统的类图如下图所示。



从上图可以看出，这个使用了工厂方法模式的系统涉及到以下的角色：

- 抽象工厂（Creator）角色：担任这个角色的是工厂方法模式的核心，它是与应用程序无关的。任何在模式中创建对象的工厂类必须实现这个接口。在上面的系统中这个角色由 Java 接口 Creator 扮演；在实际的系统中，这个角色也常常使用抽

象 Java 类实现。

- 具体工厂（Concrete Creator）角色：担任这个角色的是实现了抽象工厂接口的具体 Java 类。具体工厂角色含有与应用密切相关的逻辑，并且受到应用程序的调用以创建产品对象。在本系统中给出了两个这样的角色，也就是具体 Java 类 ConcreteCreator1 和 ConcreteCreator2。
- 抽象产品（Product）角色：工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口。在本系统中，这个角色由 Java 接口 Product 扮演；在实际的系统中，这个角色也常常使用抽象 Java 类实现。
- 具体产品（Concrete Product）角色：这个角色实现了抽象产品角色所声明的接口。工厂方法模式所创建的每一个对象都是某个具体产品角色的实例。

在本系统中，这个角色由具体 Java 类 CocnreteProduct1 和 ConcreteProduct2 扮演，它们都实现了 Java 接口 Product。

最后，为了说明这个系统的使用办法，特地引进了一个客户端角色 Client。这个角色创建工厂对象，然后调用工厂对象的工厂方法创建相应的产品对象。

源代码

下面就是这个示意性系统的源代码。

首先是抽象工厂角色的源代码，这个角色是用一个 Java 接口实现的，它声明了一个工厂方法，要求所有的具体工厂角色都实现这个工厂方法。

代码清单 1：抽象工厂角色 Creator 类的源代码

```
package com.javapatterns.factorymethod;

public interface Creator
{
    /**
     *  工厂方法
     */
     public Product factory();
}
```

下面是抽象产品角色的源代码。由于这里考虑的是最为简单的情形，所以抽象产品角色仅仅为具体产品角色提供一个共同的类型而已，所以是用一个 Java 标识接口实现的。一个没有声明任何方法的接口叫做标识接口，关于标识接口的讨论请见本书的“专题：Java 接口”一章。

代码清单 2：抽象产品角色 Product 类的源代码

```
package com.javapatterns.factorymethod;
public interface Product
{
}
```

下面是具体工厂角色 CocnreteCreator1 的源代码。这个角色实现了抽象工厂角色 Creator 所声明的工厂方法。

代码清单 3：具体工厂角色 ConcreteCreator1 类的源代码

```
package com.javapatterns.factorymethod;
public class ConcreteCreator1 implements Creator
{
    /**
     *  工厂方法
     */
     public Product factory()
     {
          return new ConcreteProduct1();
     }
}
```

下面是具体工厂角色 CocnreteCreator2 的源代码。与 CocnreteCreator1 一样，这个角色实现了抽象工厂角色 Creator 所声明的工厂方法。

代码清单 4：具体工厂角色 ConcreteCreator2 类的源代码

```
package com.javapatterns.factorymethod;
public class ConcreteCreator2 implements Creator
{
    /**
     *  工厂方法
     */
     public Product factory()
     {
          return new ConcreteProduct2();
     }
}
```

下面是具体产品角色 CocnreteProduct1 的源代码。它是此系统向客户端提供的产品，在通常情况下，这个类会有复杂的商业逻辑。

代码清单 5：具体产品角色 ConcreteProduct1 类的源代码

```
package com.javapatterns.factorymethod;
public class ConcreteProduct1 implements Product
{
     public ConcreteProduct1()
     {
          //do something
     }
}
```

类似地，下面是具体产品角色 CocnreteProduct2 的源代码。

代码清单 6：具体产品角色 ConcreteProduct2 类的源代码

```
package com.javapatterns.factorymethod;
public class ConcreteProduct2 implements Product
{
    public ConcreteProduct2()
    {
        //do something
    }
}
```

这个示意性的系统只给出了两个具体工厂类和两个具体产品类。在真实的系统中，工厂方法模式一般都会涉及到更多的具体工厂类和更多的具体产品类。

下面就是客户端角色的源代码。

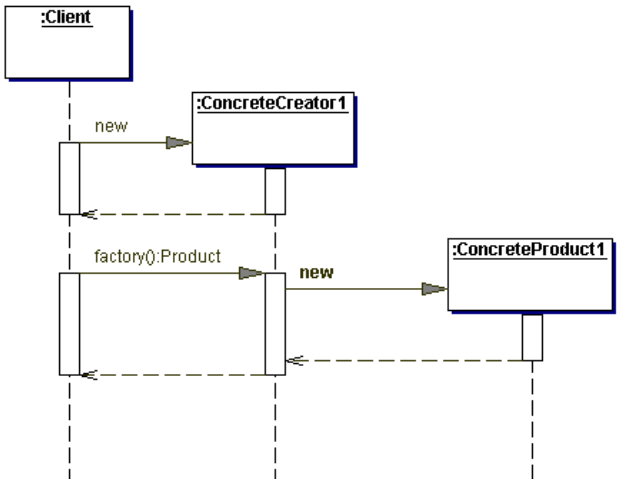
代码清单 7：客户端角色 Client 类的源代码

```
package com.javapatterns.factorymethod;
public class Client
{
    private static Creator creator1, creator2;
    private static Product prod1, prod2;
    public static void main(String[] args)
    {
        creator1 = new ConcreteCreator1();
        prod1 = creator1.factory();
        creator2 = new ConcreteCreator2();
        prod2 = creator2.factory();
    }
}
```

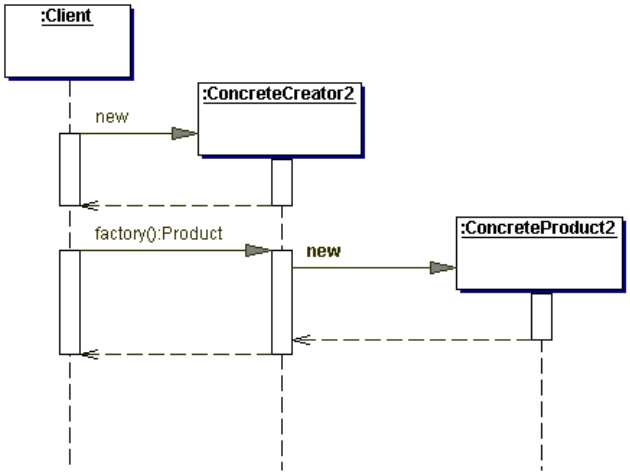
工厂方法模式的活动序列图

Client 对象的活动可以分成两部分。

（1）客户端创建 ConcreteCreator1 对象。这时客户端所持有变量的静态类型是 Creator，而实际类型是 ConcreteCreator1。然后，客户端调用 ConcreteCreator1 对象的工厂方法 factory()，接着后者调用 ConcreteProduct1 的构造子创建出产品对象。如下面的时序图所示。



（2）客户端创建一个 ConcreteCreator1 对象，然后调用 ConcreteCreator2 对象的工厂方法 factory()，而后者调用 ConcreteProduct2 的构造子创建出产品对象，如下面的时序图所示。



工厂方法模式和简单工厂模式

工厂方法模式和简单工厂模式在结构上的不同是很明显的。工厂方法模式的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体类上。工厂方法模式可以允许很多具体工厂类从抽象工厂类中将创建行为继承下来，从而可以成为多个简单工厂模式的综合，进而推广了简单工厂模式。

工厂方法模式退化后可以变得很像简单工厂模式。设想如果非常确定一个系统只需要一个具体工厂类，那么就不妨把抽象工厂类合并到具体的工厂类中去。由于反正只有一个具体工厂类，所以不妨将工厂方法改成为静态方法，这时候就得到了简单工厂模式。

与简单工厂模式中的情形一样的是，ConcreteCreator 的 factory() 方法返还的数据类型是一个抽象类型 Product，而不是哪一个具体产品类型，而客户端也不必知道所得到的产品的真实类型。这种多态性设计将工厂类选择创建哪一个产品对象、如何创建这个对象的细节完全封装在具体工厂类内部。

工厂方法模式之所以有一个别名叫多态性工厂模式，显然是因为具体工厂类都有共同的接口，或者都有共同的抽象父类。

如果系统需要加入一个新的产品，那么所需要的就是向系统中加入一个这个产品类以及它所对应的工厂类。没有必要修改客户端，也没有必要修改抽象工厂角色或者其他已有的具体工厂角色。对于增加新的产品类而言，这个系统完全支持“开-闭”原则。

13.3　工厂方法模式在农场系统中的实现

系统的优化

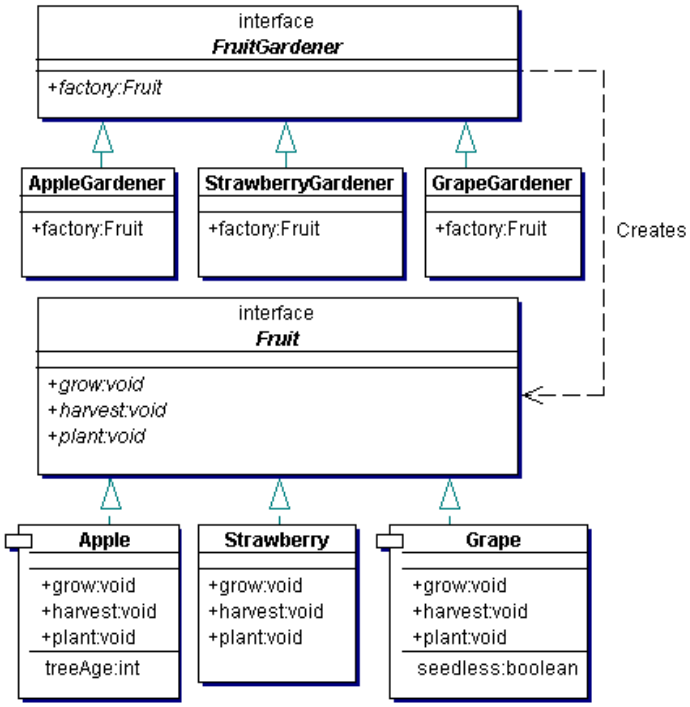
现在继续考查农场的管理系统。在本书的“简单工厂（Simple Factory）模式”一章里，讨论了支持水果类作物的系统。在那个系统中，有一个全知全能的园丁角色(FruitGardener)，控制所有作物的种植、生长和收获。现在这个农场的规模变大了，而同时发生的是管理更加专业化了。过去的全能人物没有了，每一种农作物都有专门的园丁管理，形成规模化和专业化生产。

系统的设计

取代了过去的全能角色的是一个抽象的园丁角色，这个角色规定出具体园丁角色需要实现的具体职能，而真正负责作物管理的则是负责各种作物的具体园丁角色。

这一章仍然考虑前面所讨论过的植物，包括葡萄（Grape）、草莓（Strawberry）以及萍果（Apple）等。专业化的管理要求将有专门的园丁负责专门的水果，比如苹果由“苹果园丁”负责，草莓有“草莓园丁”负责，而葡萄由“葡萄园丁”负责。这些“苹果园丁”、“草莓园丁”以及“葡萄园丁”都是实现了抽象的“水果园丁”接口的具体工厂类，而“水果园丁”则扮演抽象工厂角色。

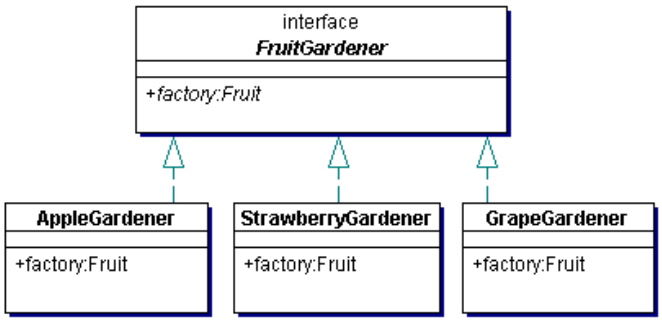
这样一来，农场系统的设计图就如下图所示。



抽象工厂类 FruitGardener 是工厂方法模式的核心，但是它并不掌握水果类或蔬菜类的生杀大权。相反地，这项权力被交给子类，即 AppleGardener StawberryGardener 以及 GrapeGardener。

工厂角色的等级结构

各个工厂角色组成一个工厂的等级结构，如下图所示。



在上图所示的等级结构中，FruitGardener 是所有具体工厂角色的超类。在本系统中，这个抽象角色是由 Java 接口 FruitGardener 接口实现的，它声明了一个工厂方法，要求所有的具体工厂角色都实现这个工厂方法。

这个角色的源代码如下所示。

代码清单 8：抽象工厂角色 FruitGardener 的源代码

```
package com.javapatterns.factorymethod.farm;
public interface FruitGardener
{
    /**
     *  工厂方法
     */
     public Fruit factory();
}
```

AppleGardener 类是具体工厂类，它实现了 FruitGardener 接口，提供了工厂方法的实现。它的源代码如下所示。

代码清单 9：具体工厂类 AppleGardener 的源代码

```
package com.javapatterns.factorymethod.farm;

public class AppleGardener
     implements FruitGardener
{
    /**
     *  工厂方法
     */
     public Fruit factory()
     {
          return new Apple();
     }
}
```

StrawberryGardener 类是一个具体工厂类，与 AppleGardener 一样，它也实现了 FruitGardener 接口。此类的源代码如下所示。

代码清单 10：具体工厂类 StrawberryGardener 的源代码

```
package com.javapatterns.factorymethod.farm;

public class StrawberryGardener
     implements FruitGardener
{
    /**
     *  工厂方法
     */
     public Fruit factory()
     {
          return new Apple();
     }
}
```

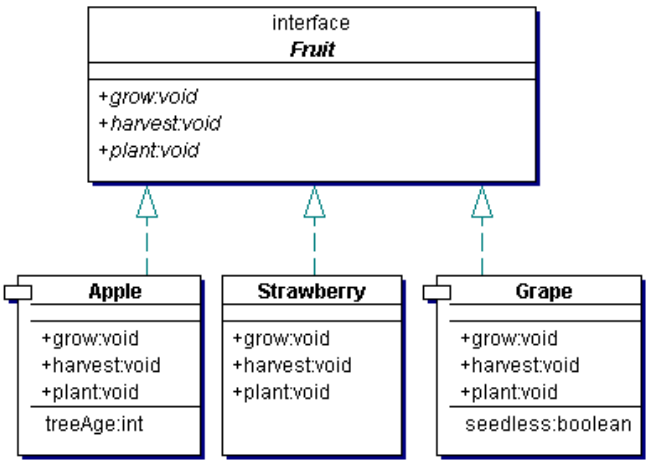
同样，具体类 GrapeGardenter 的源代码如下所示。

代码清单 11：具体工厂类 GrapeGardener 的源代码

```
package com.javapatterns.factorymethod.farm;
public class GrapeGardener
    implements FruitGardener
{
  /**
   *  工厂方法
   */
    public Fruit factory()
    {
        return new Apple();
    }
}
```

产品角色的等级结构

所有的产品角色都属于一个产品的等级结构，在这个等级结构最上面的是抽象产品角色 Product。在本系统中，这个角色是由一个 Java 接口 Fruit 实现的。由各个工厂角色组成的工厂等级结构如下图所示。



所有具体产品都必须实现抽象产品 Fruit 所声明的接口。

代码清单 12：抽象产品角色 Fruit 的源代码

```
package com.javapatterns.factorymethod.farm;
public interface Fruit
{
    void grow();
    void harvest();
    void plant();
}
```

Apple 类是一个具体产品类，实现了 Fruit 接口。

代码清单 13：水果类 Apple 的源代码

```
package com.javapatterns.factorymethod.farm;
public class Apple implements Fruit
{
    private int treeAge;
    public void grow()
    {
        System.out.println("Apple is growing...");
    }
    public void harvest()
    {
        System.out.println("Apple has been harvested.");
    }
    public void plant()
    {
        System.out.println("Apple has been planted.");
    }
    public int getTreeAge()
    {
        return treeAge;
    }
    public void setTreeAge(int treeAge)
    {
        this.treeAge = treeAge;
    }
}
```

具体产品类 Strawberry 的源代码如下所示。

代码清单 14：水果类 Strawberry 的源代码

```
package com.javapatterns.factorymethod.farm;
public class Strawberry implements Fruit
{

    public void grow()
    {
        System.out.println("Strawberry is growing...");
    }
    public void harvest()
    {
        System.out.println("Strawberry has been harvested.");
    }
    public void plant()
    {
        System.out.println("Strawberry has been planted.");
```

```
    }
}
```

Grape 类也是一个具体产品角色。

代码清单 15：水果类 Grape 的源代码

```
package com.javapatterns.factorymethod.farm;
public class Grape implements Fruit
{
    private boolean seedless;
    public void grow()
    {
        System.out.println("Grape is growing...");
    }
    public void harvest()
    {
        System.out.println("Grape has been harvested.");
    }
    public void plant()
    {
        System.out.println("Grape has been planted.");
    }
    public boolean getSeedless()
    {
        return seedless;
    }
    public void setSeedless(boolean seedless)
    {
        this.seedless = seedless;
    }
}
```

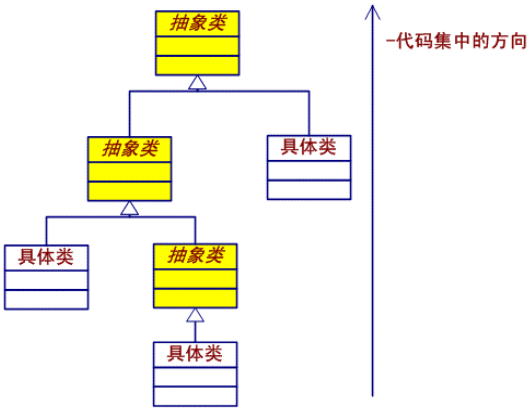
13.4　关于工厂方法模式的实现

在实现工厂方法模式时，会有一些与简单工厂模式相同或者相似的地方值得讨论。

使用 Java 接口或者 Java 抽象类

抽象工厂角色和抽象产品角色都可以选择由 Java 接口或者 Java 抽象类来实现。

如果具体工厂角色具有共同的逻辑，那么这些共同的逻辑就可以向上移动到抽象工厂角色中，这也就意味着抽象工厂角色应当用一个 Java 抽象类实现，并由抽象工厂角色提供默认的工厂方法。相反的话就应当用一个 Java 接口实现，`对抽象产品角色也是一样。共同的逻辑应当移动到超类中去，如下图所示。



关于 Java 抽象类与 Java 接口的区别，读者可以参阅本书的“专题：抽象类”和“专题：Java 接口”两章。

使用多个工厂方法

抽象工厂角色可以规定出多于一个的工厂方法，从而使具体工厂角色实现这些不同的工厂方法。这些方法可以提供不同的商业逻辑，以满足提供不同的产品对象的任务。

最后，在给相关的类和方法取名字时，应当注意让别人一看便知道在系统的设计中使用了工厂模式。

产品的循环使用

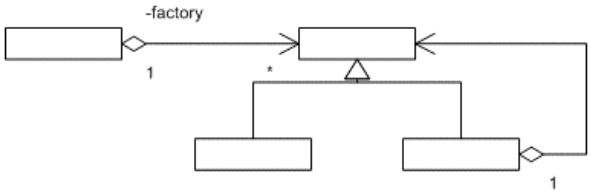
读者可以参阅本书在“工厂方法（Factory Method）模式”一章中做过的类似的分析。

在前面给出的示意性系统中，工厂方法总是调用产品类的构造子以创建一个新的产品实例，然后将这个实例提供给客户端；而在实际情形中，工厂方法所做的事情可以相当复杂。

一个常见的复杂工厂逻辑就是循环使用产品对象。如果产品对象可以由内部状态表征的话，那么对于每一个可能的内部状态，往往仅需要一个产品实例。

这时候，工厂对象就需要将已经创建过的产品对象登记到一个聚集里面，然后根据客户端所请求的产品状态，向聚集进行查询。如果聚集中有这样的产品对象，那么就直接将这个产品对象返还给客户端；如果聚集中没有这样的产品对象，那么就创建一个新的满足要求的产品对象，然后将这个对象登记到聚集中，再返还给客户端。

享元模式就使用了这样的循环工厂模式，如下图所示。



关于享元模式，请读者参阅本书的“享元模式（Flyweight Pattern）”一章。

多态性的丧失和模式的退化

一个工厂方法模式的实现依赖于工厂角色和产品角色的多态性。在有些情况下，这个模式可以出现退化，其特征就是多态性的丧失。

工厂方法创建对象

正如前面所讨论的，工厂方法不一定每一次都返还一个新的对象。但是它所返还的对象一定是他自己创建的，而不是在一个外部对象里面创建，然后传入工厂对象中的。

但是，这是否意味着凡是返还一个新的对象的方法都是工厂方法呢？不一定。

工厂方法所返还的类型

工厂方法所返还的应当是抽象类型，而不是具体类型，只有这样才能保证针对产品的多态性。换言之，调用工厂方法的客户端可以针对抽象编程，依赖于一个抽象产品类型，而不是具体产品类型。

在特殊情况下，工厂方法仅仅返还一个具体产品类型。这个时候工厂方法模式的功能就退化了，表现为针对产品角色的多态性的丧失。换言之，客户端从工厂方法的静态类型可以得知将要得到的是什么类型的对象，而这违背了工厂方法模式的用意。

当工厂方法模式发生上面的退化时，就不再是工厂方法模式了[METSKER02]。在任何一个面向对象的语言里面，都会有大量的方法返还一个新的对象，很多设计师将这种方法都叫做“工厂方法”，但是并不是所有的这种工厂方法都是工厂方法模式。本章后面的问答题给出了两个例子，读者可以参考。

工厂等级结构

工厂对象应当有一个抽象的超类型。换言之，应当有数个具体工厂类作为一个抽象超类型的具体子类存在于工厂等级结构中。如果等级结构中只有一个具体工厂类的话，那么抽象工厂角色也可以省略。

当抽象工厂角色被省略时，工厂方法模式就发生了退化，这一退化表现为针对工厂角色的多态性的丧失。这种工厂方法模式仍然可以发挥一部分工厂方法模式的用意，因此叫做退化的工厂方法模式。

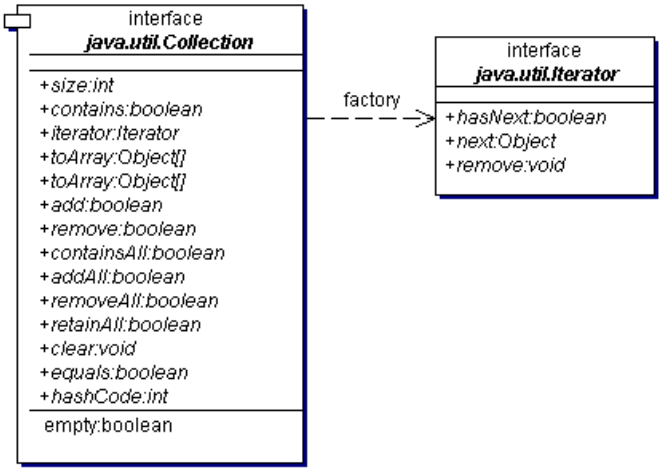
此时经常可以由简单工厂模式代替。

13.5　Java 语言中工厂方法模式的例子

工厂方法模式是一个很常见的设计模式，可以在 Java 语言 API 的各个角落里面找到。下面就给出一些读者可能已经熟悉的例子，并从模式的角度上加以分析。

在 Java 聚集中的应用（之一）

Java 聚集是 Java 1.2 版提出来的。多个对象聚在一起形成的总体称之为聚集（Aggregate），聚集对象是能够包容一组对象的容器对象。所有的 Java 聚集都实现 java.util.Collection 接口，这个接口规定所有的 Java 聚集必须提供一个 iterator()方法，返还一个 Iterator 类型的对象，如下图所示。



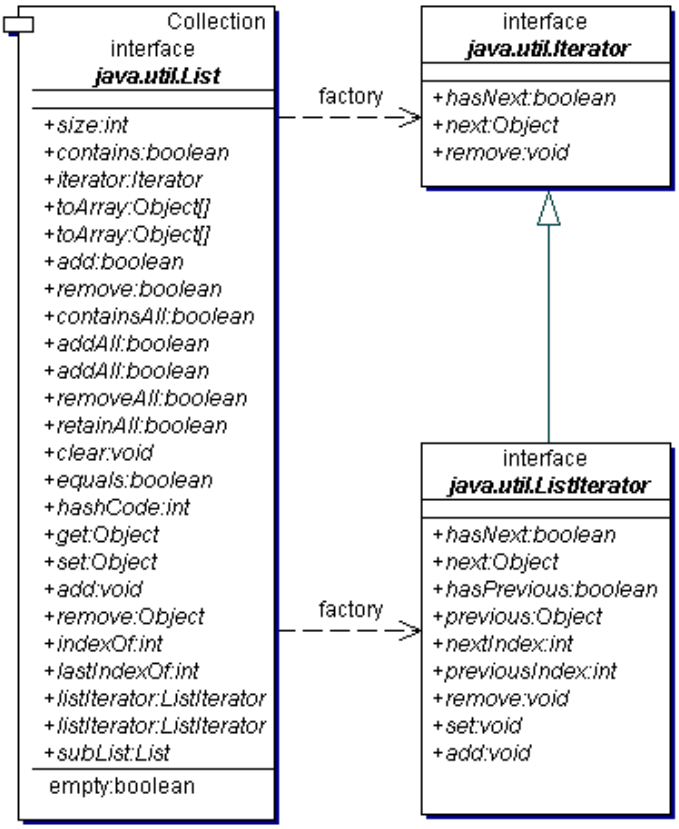
一个具体的 Java 聚集对象会通过这个 iterator()方法接口返还一个具体的 Iterator 类。可以看出，这个 iterator()方法就是一个工厂方法。

这个 Iterator 接口是迭代子模式的应用。关于迭代子模式，请参阅本书的“迭代子（Iterator）模式”一章。

在 Java 聚集中的应用（之二）

类似地，Java 的列是一种特殊的 Java 聚集。所有的 Java 除了实现了 Collection 接口之外，还实现了一个 List 接口。这个 List 接口给出了两个工厂方法，一个是上面已经讨论过的 iterator()方法，返还一个 Iterator 类型的对象；另一个是 listIterator()方法，返还一个 ListIterator 类型的对象。

有趣的是，Iterator 和 ListIterator 组成一个继承的等级结构，后者是前者的子接口。如下图所示。



这个 Iterator 接口和 ListIterator 均是迭代子模式的应用。关于迭代子模式以及 Java 语言 API 对迭代子模式的支持，请参见本书的“迭代子（Iterator）模式”以及“专题：Java 对迭代子模式的支持”等章。

URL 与 URLConnection 中的应用

URL 类代表一个 Uniform Resource Locator，也就是互联网上资源的一个指针。而一个资源可以是一个简单的文件或者目录，也可以是一个更加复杂的对象，比如向数据库或者搜索引擎的查询对象，http://www.yahoo.com 就是一个合法的 URL。

创建一个 URL 对象也很简单，只要将一个合法的 URL 传入到 URL 的构造子中即可：

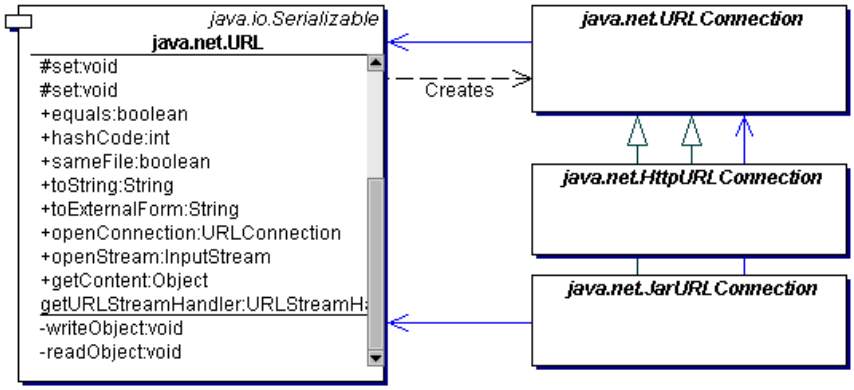
代码清单 16：怎样将 URL 类实例化

```
URL url = new URL("http://www.yahoo.com");
```

URL 对象提供一个叫做 openConnection()的工厂方法，这个方法返还一个 URL Connection 对象。而 URLConnection 对象则代表一个与远程对象的连接。URLConnection 是所有的代表应用系统与一个 URL 的连接对象的超类，使用 URLConnection 对象可以针对一个 URL 进行读写操作。

显然，URL 对象使用了工厂方法模式，以一个工厂方法 openConnection()返还一个 URLConnection 类型的对象。由于 URLConnection 是一个抽象类，因此所返还的不可能是

这个抽象类的实例，而必然是其具体子类的实例。URL 类和 URLConnection 类及其子类的结构图如下所示。



为了说明 URL 作为一个工厂类，而 URLConnection 作为一个抽象产品类的使用，下面提供一个简单的例子。

代码清单 17：一个例子的源代码

```
package com.javapatterns.factorymethod.url;
import java.net.*;
import java.io.*;
public class URLConnectionReader
{
    public static void main(String[] args) throws Exception
    {
        URL yahoo = new URL("http://www.yahoo.com/");
        URLConnection yc = yahoo.openConnection();
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
            yc.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

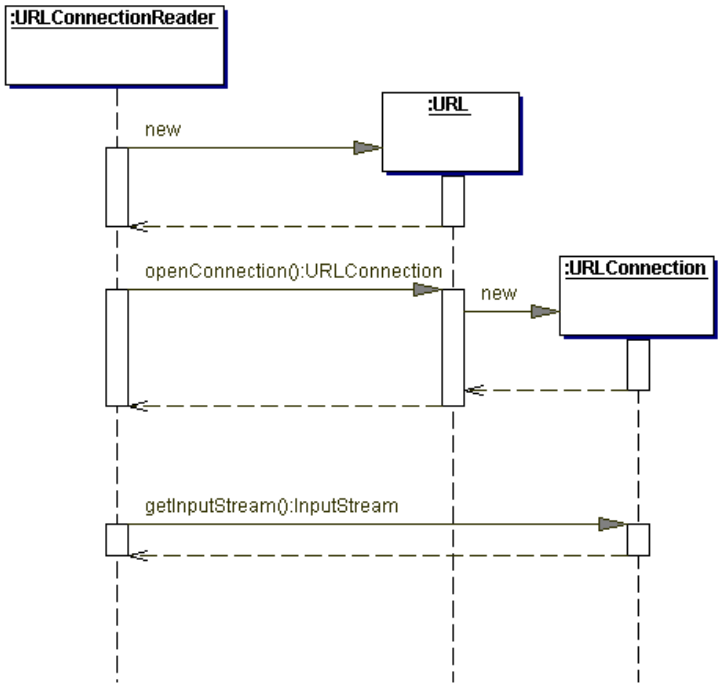
可以看出，这个例子在运行时的活动顺序如下：

（1）创建一个以 http://www.yahoo.com 为目标的 URL 对象。

（2）调用 URL 对象的 openConnection()方法，得到一个与 http://www.yahoo.com 的远程连接对象，这个对象的类型是 URLConnection。

（3）客户端调用 URLConnection 对象的 getInputStream()方法读入远程 URL 的数据。

系统的时序图如下所示。



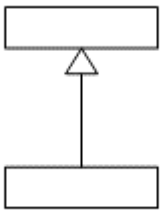
这个系统在运行时会打印出 http://www.yahoo.com 主页的全部 HTML 源代码。

13.6　工厂方法模式与其他模式的关系

模板方法模式

工厂方法模式常常与模板方法模式一起联合使用。读者可能会问，这两种模式一个是关于对象的创建，另一个是关于对象的行为，为什么会常常一起使用呢？

原因其实不难理解：第一，两个模式都是基于方法的，工厂方法模式是基于多态性的工厂方法的，而模板方法模式是基于模板方法和基本方法的；第二，两个模式都是将具体工作交给子类的。工厂方法模式将创建工作推延给子类，模板方法模式将剩余逻辑交给子类。模板方法模式的简略类图如下所示。



从各个工厂角色组成的工厂等级结构上看，抽象工厂角色中可以加入一个模板方法，代表某个顶级逻辑。而这个模板方法调用几个基本方法，这些基本方法中就可以有一个或者多个是工厂方法。这些工厂方法代表剩余逻辑，交给具体子类实现。

模版方法本身也可能是工厂方法，它的对象创建过程就是所谓的顶级结构；而这个过程可以分为数个具体步骤，每一个步骤都是顶级逻辑的组成部分。

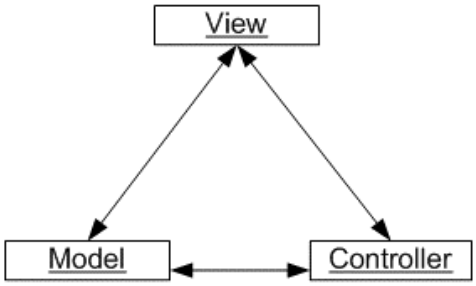
换言之，使用模板方法模式可以将某一个顶级行为分解成为数个创建行为，由子类中的工厂方法体现出来。不同的具体工厂类则提供了顶级逻辑中的剩余逻辑的不同实现。通过使用不同的具体工厂对象，客户端可以达到使用不同版本的顶级逻辑的目的。

本章在问答题中给出一个例子，详细地给出了有关模版方法模式和工厂方法模式的讨论。关于模板方法模式的介绍，请参见本书的“模板方法（Template Method）模式”一章。

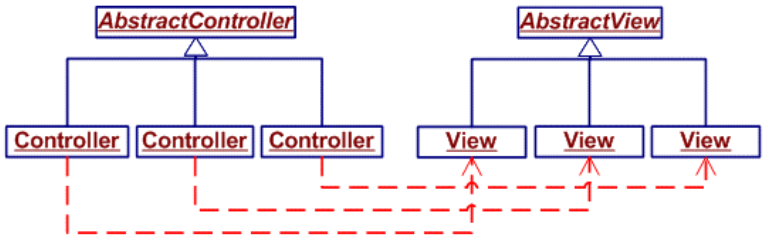
MVC 模式

MVC 模式并不是严格意义上的设计模式，而是在更高层次上的架构模式。MVC 模式可以分解成为几个设计模式的组合，包括合成模式、策略模式、观察者模式，也有可能会包括装饰模式、调停者模式、迭代子模式以及工厂方法模式等。

关于 MVC 模式的讨论可以参考本书的“专题：MVC 模式与用户输入数据检查”一章。MVC 模式的结构图如下所示。



工厂方法模式总是涉及到两个等级结构中的对象，而这两个等级结构可以分别是 MVC 模式中的控制器（Controller）和视图（View）。一个 MVC 模式可以有多个控制器和多个视图，如下图所示。



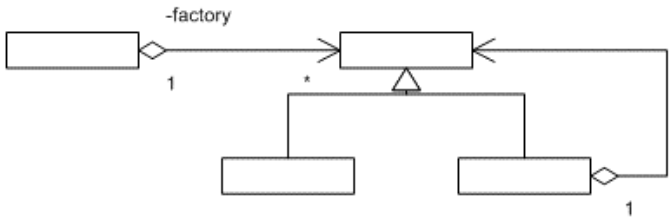
换言之，控制器端可以创建合适的视图端，就如同工厂角色创建合适的对象角色一样，

模型端则可以充当这个创建过程的客户端。

如果系统内只需要一个控制器，那么设计可能简化为简单工厂模式。请参见本书的“简单工厂（Simple Factory）模式”一章。

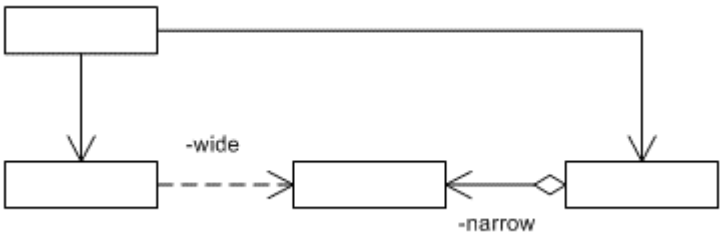
享元模式

正如在本章“关于工厂方法模式的实现”一节中讨论过的，享元模式使用了带有循环逻辑的工厂方法。享元模式的简略类图如下所示。



备忘录模式

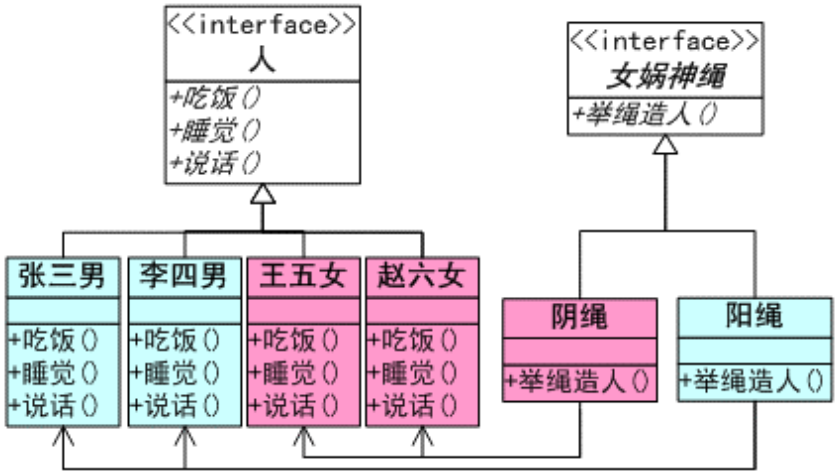
享元模式使用了一个聚集来登记所创建的产品对象，以便可以通过查询这个聚集找到和共享已经创建了的产品对象。这就是备忘录模式的应用。备忘录模式的简略类图如下所示。



13.7　女娲举绳造人

仍以女娲造人的故事为例，《风俗通》中说：“俗说天地开辟，未有人民。女娲抟黄土为人，剧务，力不暇供，乃引绳于桓泥中，举以为人。”参考“简单工厂模式”一节，女娲造人的初期，是使用简单工厂模式。然后，女娲发现她不能用这种方法造出所有的人，于是她就想出了一个聪明的办法：使用一根绳子，在泥水里搅，然后一甩，所有的泥点都变成了人。当然，女娲造出的人有男女之别，是因为女娲使用的绳子有阴阳之分。在下面，读者可以看出，女娲采用了工厂方法模式来达到造人的目的。

女娲举绳造人的故事，是应用工厂方法模式的一个实例。抽象角色“女娲神绳”是系统的中心，但是它仅仅声明出“举绳造人”方法，而没有实现它。真正做这个工作的是具体工厂类，也就是“阴绳”和“阳绳”类。工厂方法模式在女娲举绳造人系统中的实现如下图所示。



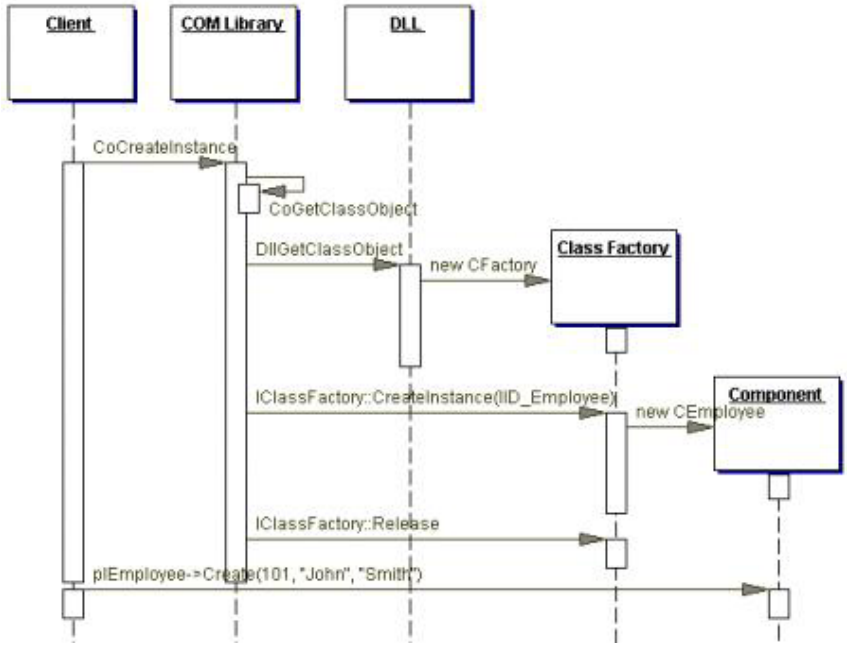
13.8　其他的例子

COM 技术架构中的工厂方法模式

在微软公司所提倡的 COM（Component Object Model）技术架构中， 工厂方法模式起着关键的作用。

在 COM 架框里，Creator 接口的角色是由一个叫做 IClassFactory 的 COM 接口来担任的。而具体类 ConcreteCreator 的角色是由实现 IClassFactory 接口的类 CFactory 来担任的。一般而言，对象的创建可能要求分配系统资源，要求在不同的对象之间进行协调等。因为

IClassFactory 的引进，所有这些在对象的创建过程中出现的细节问题， 都可以封装在一个实现 IClassFactory 接口的具体的工厂类里面。这样一来， 一个 COM 架构的支持系统只需要创建这个工厂类 CFactory 的实例就可以了。微软的 COM（Component Object Model）技术架构如下图所示。



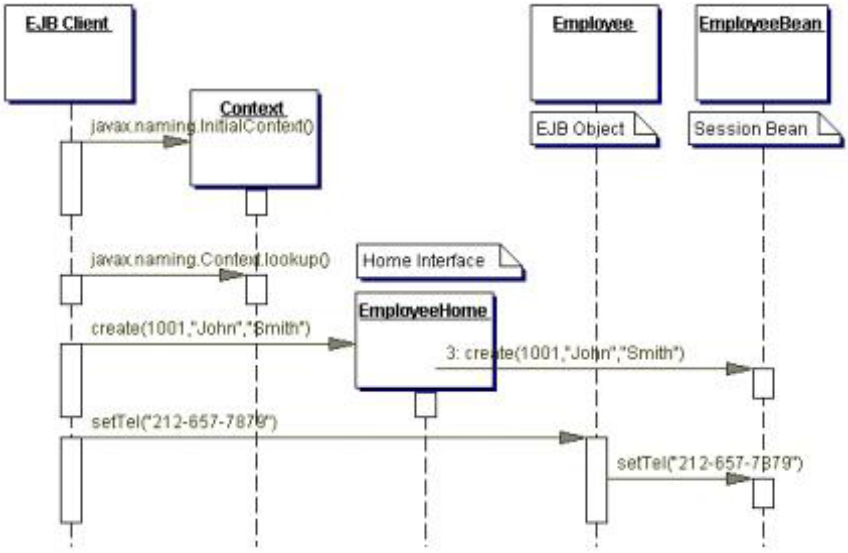
在上面的序列活动（Sequence Activity）图中，用户端调用 COM 的库函数 CoCreate-Instance。CoCreateInstance 在 COM 架构中以 CoGetClassObject 实现。CoCreateInstance 会在视窗系统的 Registry 里搜寻所要的构件（在这个例子中即 Cemployee）。如果找到了这个构件，就会加载支持此构件的 DLL。当此 DLL 加载成功后，CoGetClassObject 就会调用 DllGetClassObject。后者使用 new 操作符将工厂类 CFactory 实例化。

下面，DllGetClassObject 会向工厂类 Cfactory 查询 IClassFactory 接口，返还给 CoCreate Instance。CoCreateInstance 接下来利用 IClassFactory 接口调用 CreateInstance 函数。此时，IClassFactory::CreateInstance 调用 new 操作符来创建所要的构件（Cemployee）。此外，它搜寻 IEmployee 接口。在拿到接口的指针后， CoCreateInstance 释放掉工厂类并把接口的指针返还给客户端。

客户端现在就可以利用这个接口调用此构件中的方法了。

EJB 技术架构中的工厂方法模式

Sun Microsystem 所倡导的 EJB（Enterprise Java Beans）技术架构是一套为 Java 语言设计的，用来开发企业规模应用系统的构件模型。为了说明 EJB 架构是怎样利用工厂方法模式的，请考查下面所示的序列活动图。



在 EJB 技术架构中，工厂方法模式也起着关键的作用。

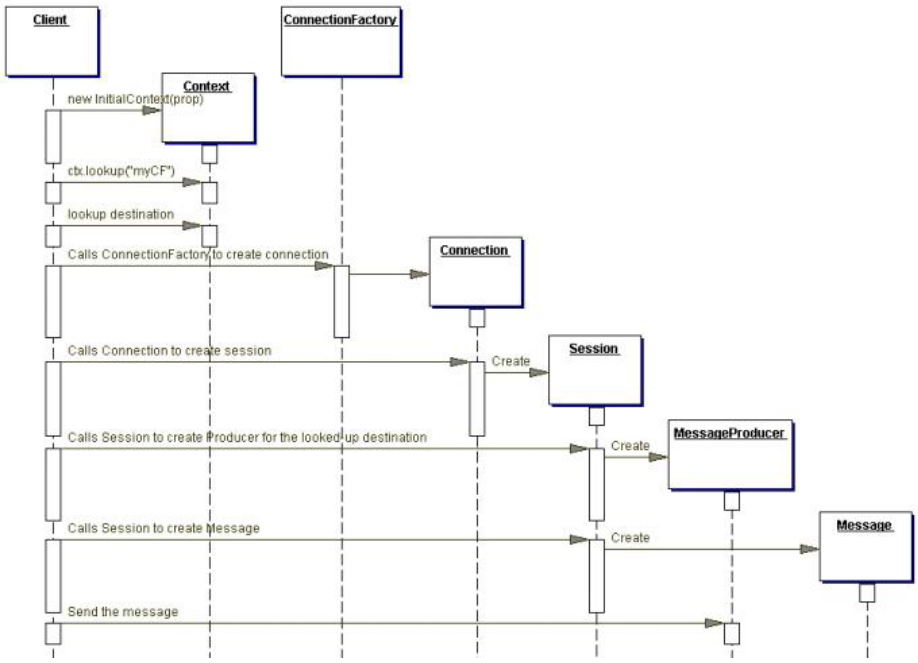
在这个时序图中，用户端首先创建一个新的 Context 对象，以便利用 JNDI 服务器寻找 EJBObject。在得到这个 Context 对象后，就可以使用 JNDI 名，比如"Employee"，来拿到 EJB 类 Employee 的 Home 接口。通过使用 Employee 的 Home 接口，客户端可以创建 EJB 对象，比如 EJB 类 Employee 的实例，然后调用 Employee 的方法。

代码清单 18： Home 接口提供工厂方法的实例

```
// 取到 JNDI naming context
Context ctx = new InitialContext ();
// 利用 ctx 索取 EJB Home 接口
EmployeeHome home = (EmployeeHome)ctx.lookup("Employee");
// 利用 Home 接口创建一个 Session Bean 对象
// 这里使用的是标准的工厂方法模式
Employee emp = home.create (1001， "John"， "Smith");
// 调用方法
emp.setTel ("212-657-7879");
```

JMS 技术架构中的工厂方法模式

JMS（Java Messaging Service）定义了一套标准的 API，让 Java 语言程序能通过支持 JMS 标准的 MOM（Message Oriented Middleware 或者面向消息的中间服务器）来创建和交换消息（message）。现在来举例看一看 JMS（Java Messaging Service）技术架构是怎样使用工厂方法模式的，如下图所示。



在上面的序列图中，用户端创建一个新的 Context 对象，以便利用 JNDI 伺服器寻找 Topic 和 ConnectionFactory 对象。在得到这个 ConnectionFactory 对象后，就可以利用 Connection 创建 Session 的实例。有了 Session 的实例后， 就可以利用 Session 创建 TopicPublisher 的实例，并利用 Session 创建消息实例。

代码清单 19：工厂模式被用于创建 Connection、Session、Producer 的实例

```
Properties prop = new Properties();
prop.put(Context.INITIAL_CONTEXT_FACTORY，
      "com.sun.jndi.fscontext.RefFSContextFactory");
prop.put(Context.PROVIDER_URL， "file:C:\temp");
// 取到 JNDI context
Context ctx = new InitialContext(prop);
// 利用 ctx 索取工厂类的实例
Topic topic = (Topic) ctx.lookup("myTopic");
TopicConnectionFactory tcf = (TopicConnectionFactory) ctx.lookup("myTCF");
// 利用工厂类创建 Connection，这是典型的工厂模式
TopicConnection tCon = tcf.createTopicConnectoin();
// 利用 Connection 创建 Session 的实例，又是工厂模式
TopicSession tSess = tCon.createTopicSession(false，
      Session.AUTO_ACKNOWLEDGE);
// 利用 Session 创建 Producer 的实例，又是工厂模式
TopicPublisher publisher = tSess.createPublisher(topic);
// 利用 Session 创建消息实例，又是工厂模式
TextMesage msg = tSess.createTextMessage("Hello from Jeff");
//发送消息
publisher.publish(msg);
```

问答题

1. 有很多的 Java 语言中的 API 提供一些返还新的 Java 对象的方法。能否举出两个这样的方法的例子？请问它们是工厂方法模式吗？

2. 请问下面这句话对吗？

“一个工厂方法必须返还一个新的对象。如果返还的不是一个新的对象，就不符合工厂方法模式的描述。”

3. 请问工厂方法可不可以返还在另一个对象里实例化的一个对象？

4. 某一个商业软件产品需要支持 Sybase 和 Oracle 数据库。这个系统需要这样的一个查询运行器系统，根据客户端的需要，可以随时向 Sybase 或者 Oracle 数据库引擎发出查询。请给出这样的一个系统的示意性设计，并且请考虑在设计中使用工厂方法模式是否合适。暂时可以假定所发出的查询总是同一个 SQL 语句。

（本问题和答案受到文献[SHALLOWAY02]中的一个相似的例子的启发。本书鼓励读者阅读原著）

5. 请阅读上一题的答案，并分析上题的设计与模版方法模式有无关系。

问答题答案

1. Java 对象的 toString()方法和 clone()方法会给出一个新的 Java 对象。

一个 Java 对象的 toString()方法会给出一个 String 类型的对象；而 clone()方法会给出与原对象类型相同的对象。

它们都不是工厂方法模式，因为它们都不能返还一个抽象类型，客户端在事先都知道将要得到的对象类型。

换言之，并非每一个返还一个新的对象的方法都是工厂方法模式。

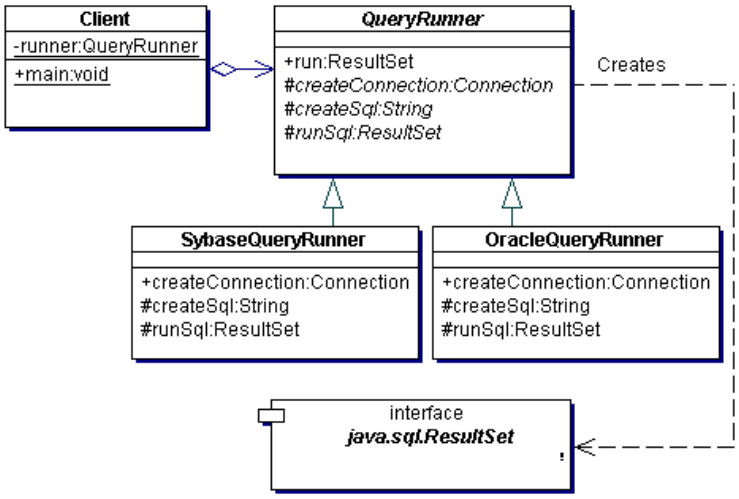
2. 这是不对的。一个工厂方法可以通过登记自己所创建的产品对象来实现循环提供相同的一些产品对象的功能。在享元模式中，就使用了这种工厂方法模式。

换言之，一个工厂方法所提供的产品对象不一定每次都是新的。

3. 不可以。工厂方法模式是创建模式，工厂方法的用意就是对对象创建过程的封装。虽然工厂方法不一定每一次都返还一个新的对象，但是工厂方法所返还的都应当是在工厂角色中被实例化的对象。

如果一个方法返还的对象是在另外一个对象中实例化的，那么这个方法不是工厂方法。

4. 下图所示就是这个查询运行器系统的设计图。



可以看出，这个系统是由一个客户端 Client（也就是产品的消费角色），一个抽象工厂角色 QueryRunner，两个具体工厂角色 SybaseQueryRunner 和 OracleQueryRunner，以及产品角色组成。

对于客户端 Client 而言，系统的抽象产品角色是 ResultSet 接口，而具体产品角色就是 java.sql.Connection 所返还的具体 ResultSet 对象。但是如果仔细查看各个工厂角色就可以发现，createSql()方法和 createConnection()方法实际上也是工厂方法，它们的产品是 SQL 语句和 Connection 对象。

下面就是抽象工厂角色 QueryRunner 的源代码。

代码清单 20：Java 抽象类 QueryRunner 的源代码

```
package com.javapatterns.factorymethod.query;
import java.sql.Connection;
import java.sql.ResultSet;
abstract public class QueryRunner
{
    public ResultSet run() throws Exception
    {
        Connection conn = createConnection();
        String sql = createSql();
        return runSql(conn, sql);
    }
    protected abstract Connection createConnection();
    protected abstract String createSql();
    protected abstract ResultSet runSql(Connection conn, String sql)
        throws Exception;
}
```

下面是具体工厂角色 SybaseQueryRunner 的源代码。这个具体类实现了 Java 抽象类 QueryRunner 所声明的抽象方法。

代码清单 21：SybaseQueryRunner 的源代码

```
package com.javapatterns.factorymethod.query;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
public class SybaseQueryRunner extends QueryRunner
{
    public Connection createConnection()
    {
        //示意性代码
        return null;
    }
    protected String createSql()
    {
        return "SELECT * FROM customers";
    }
    protected ResultSet runSql(Connection conn, String sql)
        throws Exception
    {
        Statement stmt = conn.createStatement();
        return stmt.executeQuery(sql);
    }
}
```

同样，OracleQueryRunner 也是一个具体工厂角色，它也是抽象工厂的子类，实现了 QueryRunner 所声明的抽象方法。

代码清单 22：OracleQueryRunner 的源代码

```
package com.javapatterns.factorymethod.query;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
public class OracleQueryRunner extends QueryRunner
{
    public Connection createConnection()
    {
        //这下面仅仅是一个示意性的实现
        return null;
    }
    protected String createSql()
    {
        //示意性的 SQL 语句
        return "SELECT * FROM customers";
    }
    protected ResultSet runSql(Connection conn, String sql)
        throws Exception
```

```
        {
            Statement stmt = conn.createStatement();
            return stmt.executeQuery(sql);
        }
    }
```

由于两个具体工厂子类 OracleQueryRunner 和 SybaseQueryRunner 分别处理与 Oracle 和 Sybase 数据库的连接和查询，因此，客户端就可以动态地决定何时采取哪一个具体工厂对象来创建 ResultSet 对象。

下面是一个示意性的客户端的源代码。

代码清单 23：客户端 Client 类的示意性源代码

```
package com.javapatterns.factorymethod.query;
import java.sql.ResultSet;
public class Client
{
    private static QueryRunner runner;
    public static void main(String[] args)
        throws Exception
    {
        runner = new SybaseQueryRunner();
        ResultSet rs = runner.run();
    }
}
```

5．在上题给出的答案中，确实使用了模版方法模式。如果读者仔细查看一下系统设计图就会发现，在 QueryRunner 中，run()方法就是一个模版方法，这个方法代表一个顶级逻辑：返还查询的结果 ResultSet。

此逻辑在细节上分成几个步骤，分别由下面的基本方法完成：createSql()，createConnection()以及 runSql()，而这几个基本方法被当做剩余逻辑交给具体子类实现。

具体子类为两种数据库引擎提供了不同的实现，从而为一个顶级逻辑提供了不同的具体实现。

参考文献

[METSKER02] Steven J. Metsker. Design Patterns Java Workbook. published by Addison Wesley, 2002.