第 12 章　简单工厂（Simple Factory）模式

简单工厂模式是类的创建模式，又叫做静态工厂方法（Static Factory Method）模式。简单工厂模式是由一个工厂对象决定创建出那一种产品类的实例。

12.1　工厂模式的几种形态

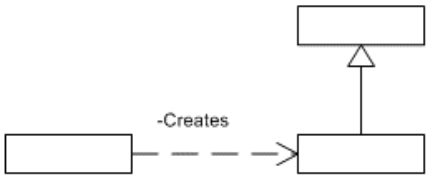
工厂模式专门负责将大量有共同接口的类实例化。工厂模式可以动态决定将哪一个类实例化，不必事先知道每次要实例化哪一个类。工厂模式有以下几种形态：

（1）简单工厂（Simple Factory）模式，又称静态工厂方法模式（Static Factory Method Pattern）。

（2）工厂方法（Factory Method）模式，又称多态性工厂（Polymorphic Factory）模式或虚拟构造子（Virtual Constructor）模式；

（3）抽象工厂（Abstract Factory）模式，又称工具箱（Kit 或 Toolkit）模式。

下面就是简单工厂模式的简略类图。



简单工厂模式，或称静态工厂方法模式，是不同的工厂方法模式的一个特殊实现。在其他文献中，简单工厂往往作为普通工厂模式的一个特例讨论。

在 Java 语言中，通常的工厂方法模式不能通过设计功能的退化给出静态工厂方法模式。因为一个方法是不是静态的，对于 Java 语言来说是一个很大的区别，必须在一开始的时候就加以考虑。这就是本书将简单工厂单独提出来讨论的一个原因。学习简单工厂模式是对学习工厂方法模式的一个很好的准备，也是对学习其他模式，特别是单例模式和多例模式的一个很好的准备，这就是本书首先讲解这一模式的另一个原因。

12.2　简单工厂模式的引进

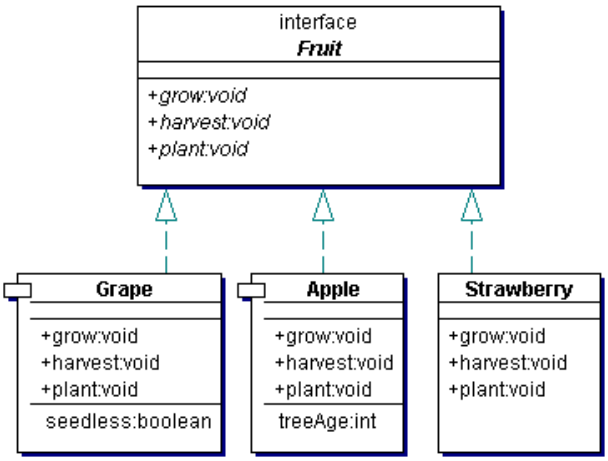
比如说有一个农场公司，专门向市场销售各类水果。在这个系统里需要描述下列的水果：

葡萄　Grape

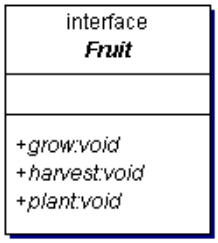
草莓　Strawberry

苹果　Apple

水果与其他的植物有很大的不同，就是水果最终是可以采摘食用的。那么一个自然的作法就是建立一个各种水果都适用的接口，以便与农场里的其他植物区分开。如下图所示。



水果接口规定出所有的水果必须实现的接口，包括任何水果类必须具备的方法：种植plant()，生长 grow()以及收获 harvest()。接口 Fruit 的类图如下所示。



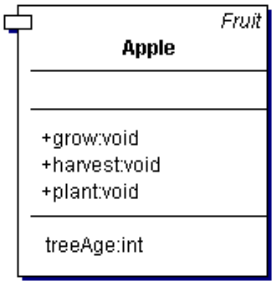
这个水果接口的源代码如下所示。

代码清单 1：接口 Fruit 的源代码

```
public interface Fruit
{
    /**
     *  生长
     */
```

```
    void grow();
    /**
     *  收获
     */
    void harvest();
    /**
     *  种植
     */
    void plant();
}
```

描述苹果的 Apple 类的源代码的类图如下所示。



Apple 类是水果类的一种，因此它实现了水果接口所声明的所有方法。另外，由于苹果是多年生植物，因此多出一个 treeAge 性质，描述苹果树的树龄。下面是这个苹果类的源代码。

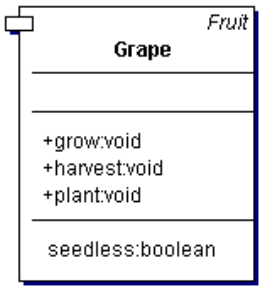
代码清单 2：类 Apple 的源代码

```
public class Apple implements Fruit
{
    private int treeAge;

    /**
     *  生长
     */
    public void grow()
    {
        log("Apple is growing...");
    }
    /**
     *  收获
     */
    public void harvest()
    {
        log("Apple has been harvested.");
    }
    /**
```

```
    *  种植
    */
    public void plant()
    {
        log("Apple has been planted.");
    }
   /**
    *  辅助方法
    */
    public static void log(String msg)
    {
        System.out.println(msg);
    }
   /**
    *  树龄的取值方法
    */
    public int getTreeAge()
    {
        return treeAge;
    }
   /**
    *  树龄的赋值方法
    */
    public void setTreeAge(int treeAge)
    {
        this.treeAge = treeAge;
    }
}
```

同样，Grape 类是水果类的一种，也实现了 Fruit 接口所声明的所有的方法。但由于葡萄分有籽和无籽两种，因此，比通常的水果多出一个 seedless 性质，如下图所示。



葡萄类的源代码如下所示。可以看出，Grape 类同样实现了水果接口，从而是水果类型的一种子类型。

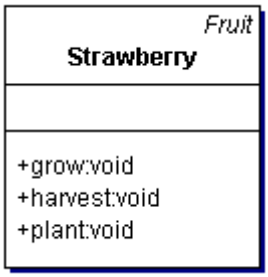
代码清单 3：类 Grape 的源代码

```
public class Grape implements Fruit
```

```
{
    private boolean seedless;

    /**
     *  生长
     */
    public void grow()
    {
        log("Grape is growing...");
    }
    /**
     *  收获
     */
    public void harvest()
    {
        log("Grape has been harvested.");
    }
    /**
     *  种植
     */
    public void plant()
    {
        log("Grape has been planted.");
    }
    /**
     *  辅助方法
     */
    public static void log(String msg)
    {
        System.out.println(msg);
    }
    /**
     *  有无籽的取值方法
     */
    public boolean getSeedless()
    {
        return seedless;
    }
    /**
     *  有无籽的赋值方法
     */
    public void setSeedless(boolean seedless)
    {
        this.seedless = seedless;
    }
}
```

下图所示是 Strawberry 类的类图。

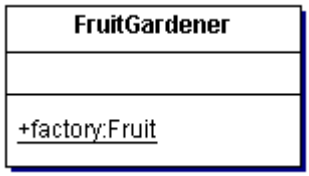


Strawberry 类实现了 Fruit 接口，因此，也是水果类型的子类型，其源代码如下所示。

代码清单 4：类 Strawberry 的源代码

```
public class Strawberry implements Fruit
{
    /**
     *  生长
     */
     public void grow()
     {
          log("Strawberry is growing...");
     }
    /**
     *  收获
     */
     public void harvest()
     {
          log("Strawberry has been harvested.");
     }
    /**
     *  种植
     */
     public void plant()
     {
          log("Strawberry has been planted.");
     }
    /**
     *  辅助方法
     */
     public static void log(String msg)
     {
          System.out.println(msg);
     }
}
```

农场的园丁也是系统的一部分，自然要由一个合适的类来代表。这个类就是 FruitGardener 类，其结构由下面的类图描述。



FruitGardener 类会根据客户端的要求，创建出不同的水果对象，比如苹果（Apple），葡萄（Grape）或草莓（Strawberry）的实例。而如果接到不合法的要求，FruitGardener 类会抛出 BadFruitException 异常，如下图所示。

园丁类的源代码如下所示。

代码清单 5：FruitGardener 类的源代码

```
public class FruitGardener
{
    /**
     *  静态工厂方法
     */
     public static Fruit factory(String which)
            throws BadFruitException
     {
          if (which.equalsIgnoreCase("apple"))
          {
               return new Apple();
          }
          else if (which.equalsIgnoreCase("strawberry"))
          {
               return new Strawberry();
          }
          else if (which.equalsIgnoreCase("grape"))
          {
               return new Grape();
          }
          else
          {
```

```
                throw new BadFruitException("Bad fruit request");
            }
        }
    }
```

可以看出，园丁类提供了一个静态工厂方法。在客户端的调用下，这个方法创建客户端所需要的水果对象。如果客户端的请求是系统所不支持的，工厂方法就会抛出一个 BadFruitException 异常。这个异常类的源代码如下所示。

代码清单 6：BadFruitException 类的源代码

```
public class BadFruitException extends Exception
{
    public BadFruitException(String msg)
    {
        super(msg);
    }
}
```

在使用时，客户端只需调用 FruitGardener 的静态方法 factory()即可。请见下面的示意性客户端源代码。

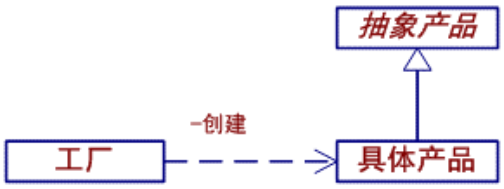
代码清单 7：怎样使用异常类 BadFruitException

```
try
{
    FruitGardener.factory("grape");
    FruitGardener.factory("apple");
    FruitGardener.factory("strawberry");
        ...
}
catch(BadFruitException e)
{
    ...
}
```

这样，农场一定会百果丰收啦!

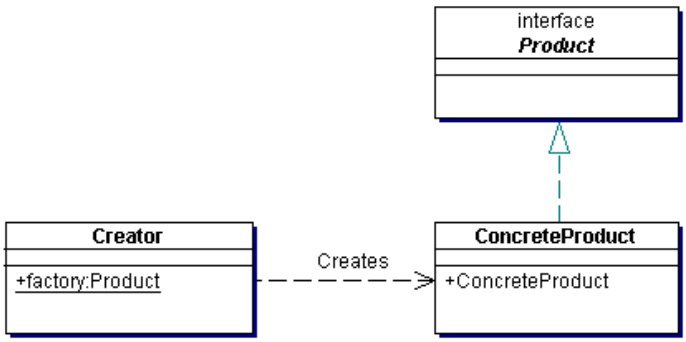
12.3　简单工厂模式的结构

简单工厂模式是类的创建模式，这个模式的一般性结构如下图所示。


抽象产品
-创建
工厂
具体产品

角色与结构

简单工厂模式就是由一个工厂类可以根据传入的参量决定创建出哪一种产品类的实例。下图所示为以一个示意性的实现为例说明简单工厂模式的结构。



从上图可以看出，简单工厂模式涉及到工厂角色、抽象产品角色以及具体产品角色等三个角色：

- 工厂类（Creator）角色：担任这个角色的是工厂方法模式的核心，含有与应用紧密相关的商业逻辑。工厂类在客户端的直接调用下创建产品对象，它往往由一个具体 Java 类实现。
- 抽象产品（Product）角色：担任这个角色的类是工厂方法模式所创建的对象的父类，或它们共同拥有的接口。抽象产品角色可以用一个 Java 接口或者 Java 抽象类实现。
- 具体产品（Concrete Product）角色：工厂方法模式所创建的任何对象都是这个角色的实例，具体产品角色由一个具体 Java 类实现。

源代码

工厂类的示意性源代码如下所示。可以看出，这个工厂方法创建了一个新的具体产品的实例并返还给调用者。

代码清单 8：Creator 类的源代码

```
public class Creator
{
    /**
     *  静态工厂方法
     */
     public static Product factory()
     {
          return new ConcreteProduct();
     }
```

```
}
```

抽象产品角色的主要目的是给所有的具体产品类提供一个共同的类型，在最简单的情况下，可以简化为一个标识接口。所谓标识接口，就是没有声明任何方法的空接口。关于标识接口的讨论，请参见本书的“专题：Java 接口”一章。

代码清单 9：抽象角色 Product 接口的源代码

```
public interface Product
{
}
```

具体产品类的示意性源代码如下。

代码清单 10：具体产品角色 ConcreteProduct 类的源代码

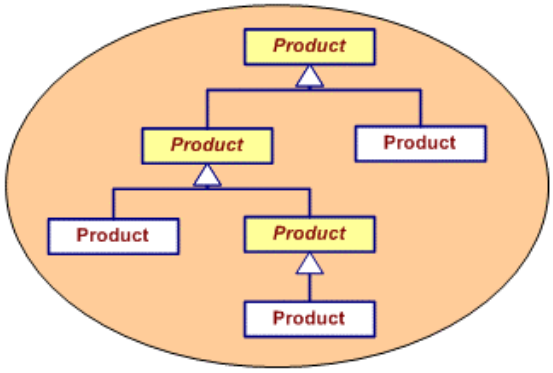
```
public class ConcreteProduct implements Product
{
    public ConcreteProduct(){}
}
```

虽然在这个简单的示意性实现里面只给出了一个具体产品类，但是在实际应用中一般都会遇到多个具体产品类的情况。

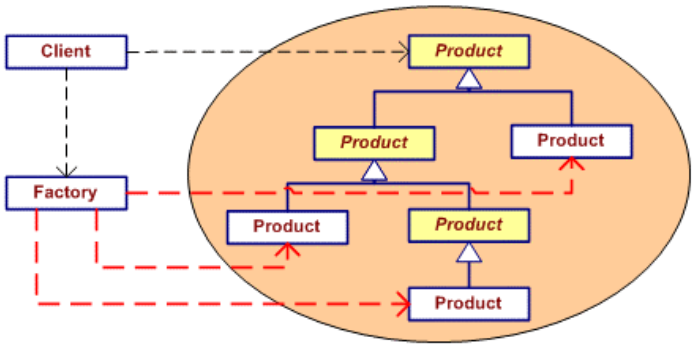
12.4　简单工厂模式的实现

多层次的产品结构

在真实的系统中，产品可以形成复杂的等级结构，比如下图所示的树状结构上就有多个抽象产品类和具体产品类。



这个时候，简单工厂模式采取的是以不变应万变的策略，一律使用同一个工厂类。如下图所示。

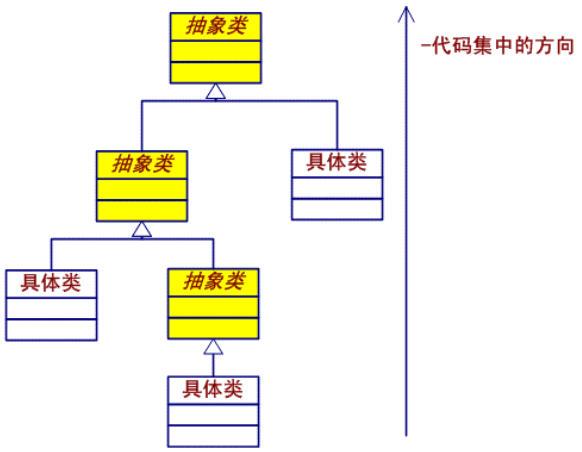


图中从 Factory 类到各个 Product 类的虚线代表创建（依赖）关系；从 Client 到其他类的联线是一般依赖关系。

这样做的好处是设计简单，产品类的等级结构不会反映到工厂类中来，从而产品类的等级结构的变化也就不会影响到工厂类。但是这样做的缺点是，增加新的产品必将导致工厂类的修改。请参见后面“模式的优点和缺点”一节，以及本书的“工厂方法（Factory Method）模式”一章。

使用 Java 接口或者 Java 抽象类

如果模式所产生的具体产品类彼此之间没有共同的商业逻辑，那么抽象产品角色可以由一个 Java 接口扮演；相反，如果这些具体产品类彼此之间确有共同的商业逻辑，那么这些公有的逻辑就应当移到抽象角色里面，这就意味着抽象角色应当由一个抽象类扮演。在一个类型的等级结构里面，共同的代码应当尽量向上移动，以达到共享的目的，如下图所示。



关于抽象类与 Java 接口的关系与区别，请参见本书的“专题：Java 接口”与“专题：抽象类”两章。

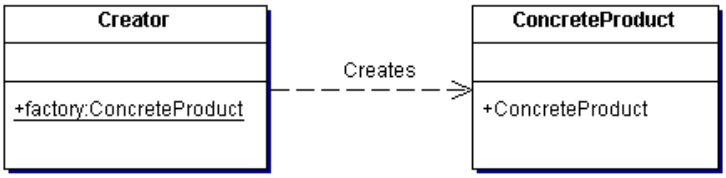
多个工厂方法

每个工厂类可以有多于一个的工厂方法，分别负责创建不同的产品对象。比如 java.text.DateFormat 类是其子类的工厂类，而 DateFormat 类就提供了多个静态工厂方法。请参见本章后面的详尽讲解。

抽象产品角色的省略

如果系统仅有一个具体产品角色的话，那么就可以省略掉抽象产品角色。省略掉抽象产品类后的简略类图如下图所示。

仍然以前面给出的示意性系统为例，这时候系统的类图就变成如下所示。



下面是工厂类的源代码。显然，这个类提供一个工厂方法，返还一个具体产品类的实例。

代码清单 11：工厂角色的源代码

```
package com.javapatterns.simplefactory.simplified;
public class Creator
{
    /**
     *  静态工厂方法
     */
     public static ConcreteProduct factory()
     {
          return new ConcreteProduct();
     }
}
```

具体产品类的源代码如下所示。

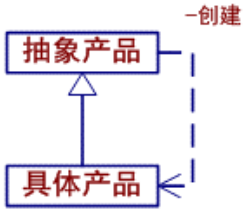
代码清单 12：具体产品角色 ConcreteProduct 类的源代码

```
package com.javapatterns.simplefactory.simplified;
public class ConcreteProduct
```

```
{
    public ConcreteProduct(){}
}
```

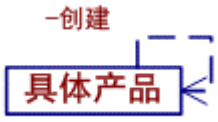
工厂角色与抽象产品角色合并

在有些情况下，工厂角色可以由抽象产品角色扮演。典型的应用就是 java.text.DateFormat 类，一个抽象产品类同时是子类的工厂，如下图所示。本章在下面给出了一个更为详尽的描述。

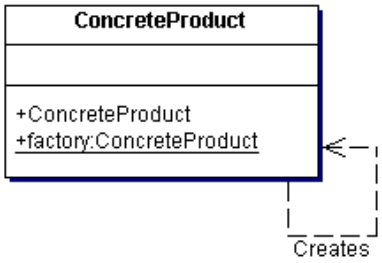


三个角色全部合并

如果抽象产品角色已经被省略，而工厂角色就可以与具体产品角色合并。换言之，一个产品类为自身的工厂，如下图所示。



如果仍然以前面讨论过的示意性系统为例，这个系统的结构图如下所示。



显然，三个原本独立的角色：工厂角色、抽象产品以及具体产品角色都已经合并成为一个类，这个类自行创建自己的实例。请见下面的源代码。

代码清单 13：具体产品角色与工厂角色合并后的代码

```
package com.javapatterns.simplefactory.simplified1;
```

```
public class ConcreteProduct
{
    public ConcreteProduct(){}
   /**
    *   静态工厂方法
    */
    public static ConcreteProduct factory()
    {
        return new ConcreteProduct();
    }
}
```

这种退化的简单工厂模式与单例模式以及多例模式有相似之处，但是并不等于单例或者多例模式。关于这一点的讨论请参见本章后面的叙述。

产品对象的循环使用和登记式的工厂方法

由于简单工厂模式是一个非常基本的设计模式，因此它会在较为复杂的设计模式中出现。在本章前面所给出的示意性例子中，工厂方法总是简单地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。

在本书所讨论的所有设计模式中，单例模式与多例模式是建立在简单工厂模式的基础之上的，而且它们都要求工厂方法具有特殊的逻辑，以便能循环使用产品的实例。

在很多情况下，产品对象可以循环使用。换言之，工厂方法可以循环使用已经创建出来的对象，而不是每一次都创建新的产品对象。工厂方法可以通过登记它所创建的产品对象来达到循环使用产品对象的目的。

如果工厂方法总是循环使用同一个产品对象，那么这个工厂对象可以使用一个属性来存储这个产品对象。每一次客户端调用工厂方法时，工厂方法总是提供这同一个对象。在单例模式中就是这样，单例类提供一个静态工厂方法，向外界提供一个惟一的单例类实例。

如果工厂方法永远循环使用固定数目的一些产品对象，而且这些产品对象的数目并不大的话，可以使用一些私有属性存储对这些产品对象的引用。比如，一个永远只提供一个产品对象的工厂对象可以使用一个静态变量存储对这个产品对象的引用。

相反，如果工厂方法使用数目不确定、或者数目较大的一些产品对象的话，使用属性变量存储对这些产品对象的引用就不方便。这时候，就应当使用聚集对象存储对产品对象的引用。

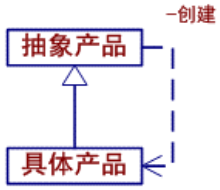
不管使用哪一种方法，工厂方法都可以做到循环使用它所创建的产品对象。循环的逻辑可能是基于这些产品类的内部状态，比如某一种状态的产品对象只创建一个，让所有需要处于这一状态上的产品对象的客户端共享这一个实例。

请参见下面对单例模式和多例模式的讨论。

12.5　简单工厂模式与其他模式的关系

单例模式

单例模式使用了简单工厂模式。换言之，单例类具有一个静态工厂方法提供自身的实例。一个抽象产品类同时是子类的工厂，如下图所示。



但是单例模式并不是简单工厂模式的退化情形，单例模式要求单例类的构造子是私有的，从而客户端不能直接将之实例化，而必须通过这个静态工厂方法将之实例化，而且单例类自身是自己的工厂角色。换言之，单例类自己负责创建自身的实例。

单例类使用一个静态的属性存储自己的惟一实例，工厂方法永远仅提供这一个实例。

多例模式

多例模式是对单例模式的推广。多例模式与单例模式的共同之处在于它们都禁止外界直接将之实例化，同时通过静态工厂方法向外界提供循环使用的自身的实例。它们的不同在于单例模式仅有一个实例，而多例模式则可以有多个实例。

多例模式往往具有一个聚集属性，通过向这个聚集属性登记已经创建过的实例达到循环使用实例的目的。一般而言，一个典型的多例类具有某种内部状态，这个内部状态可以用来区分各个实例；而对应于每一个内部状态，都只有一个实例存在。

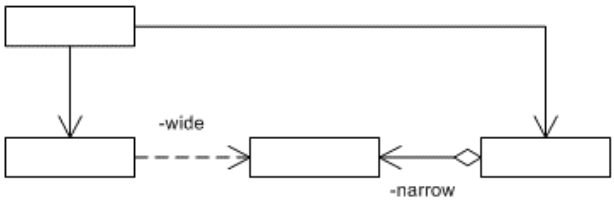
根据外界传入的参量，工厂方法可以查询自己的登记聚集，如果具有这个状态的实例已经存在，就直接将这个实例提供给外界；反之，就首先创建一个新的满足要求的实例，将之登记到聚集中，然后再提供给客户端。

关于单例模式和多例模式请读者参阅本书的“单例（Singleton）模式”一章和“多例（Multiton）模式”一章。

备忘录模式

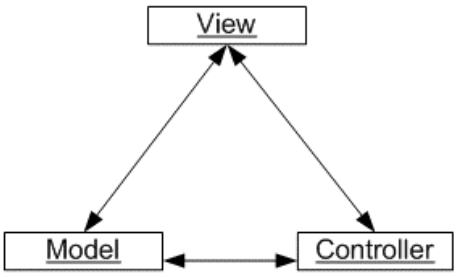
单例和多例模式使用了一个属性或者聚集属性来登记所创建的产品对象，以便可以通

过查询这个属性或者聚集属性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图所示。

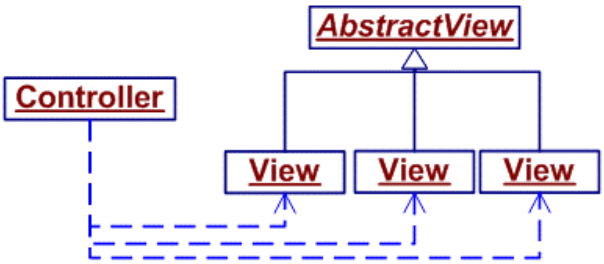


MVC 模式

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。MVC 模式的结构图如下所是。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。



简单工厂模式所创建的对象往往属于一个产品等级结构，这个等级结构可以是 MVC 模式中的视图（View）；而工厂角色本身可以是控制器（Controller）。一个 MVC 模式可以有一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；而模型端则可以充当这个创建过程的客户端。

如果系统需要有多个控制器参与这个过程的话，简单工厂模式就不适用了，应当考虑使用工厂方法模式。请参阅本书的“工厂方法（Factory Method）模式”一章。

12.6　模式的优点和缺点

模式的优点

模式的核心是工厂类。这个类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例。而客户端则可以免除直接创建产品对象的责任，而仅仅负责“消费”产品。简单工厂模式通过这种做法实现了对责任的分割。

模式的缺点

正如同在本章前面所讨论的，当产品类有复杂的多层次等级结构时，工厂类只有它自己。以不变应万变，就是模式的缺点。

这个工厂类集中了所有的产品创建逻辑，形成一个无所不知的全能类，有人把这种类叫做上帝类（God Class）。如果这个全能类代表的是农场的一个具体园丁的话，那么这个园丁就需要对所有的产品负责，成了农场的关键人物，他什么时候不能正常工作了，整个农场都要受到影响。

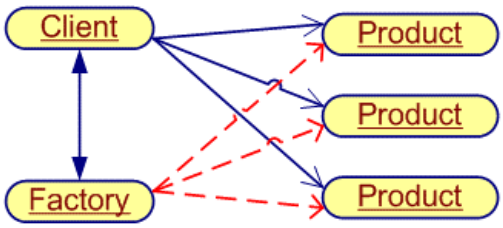
将这么多的逻辑集中放在一个类里面的另外一个缺点是，当产品类有不同的接口种类时，工厂类需要判断在什么时候创建某种产品。这种对时机的判断和对哪一种具体产品的判断逻辑混合在一起，使得系统在将来进行功能扩展时较为困难。这一缺点在工厂方法模式中得到克服。

由于简单工厂模式使用静态方法作为工厂方法，而静态方法无法由子类继承，因此，工厂角色无法形成基于继承的等级结构。这一缺点会在工厂方法模式中得到克服。

“开–闭”原则

“开–闭”原则要求一个系统的设计能够允许系统在无需修改的情况下，扩展其功能。那么简单工厂模式是否满足这个条件？

要回答这个问题，首先需要将系统划分成不同的子系统，再考虑功能扩展对于这些子系统的要求。一般而言，一个系统总可以划分成为产品的消费者角色（Client）、产品的工厂角色（Factory）以及产品角色（Product）三个子系统，如下图所示。



在这个系统中，功能的扩展体现在引进新的产品上。“开–闭”原则要求系统允许当新的产品加入系统中，而无需对现有代码进行修改。这一点对于产品的消费角色是成立的，而对于工厂角色是不成立的。

对于产品消费角色来说，任何时候需要某种产品，只需向工厂角色请求即可。而工厂角色在接到请求后，会自行判断创建和提供哪一个产品。所以，产品消费角色无需知道它得到的是哪一个产品；换言之，产品消费角色无需修改就可以接纳新的产品。

对于工厂角色来说，增加新的产品是一个痛苦的过程。工厂角色必须知道每一种产品，如何创建它们，以及何时向客户端提供它们。换言之，接纳新的产品意味着修改这个工厂角色的源代码。

综合本节的讨论，简单工厂角色只在有限的程度上支持“开–闭”原则。

12.7　简单工厂模式在 Java 中的应用

简单工厂模式是一个很基本的设计模式，读者可以在 Java 语言的 API 中看到这个模式的应用。下面就以几个 Java API 中的应用为例讲解怎样实现简单工厂模式。

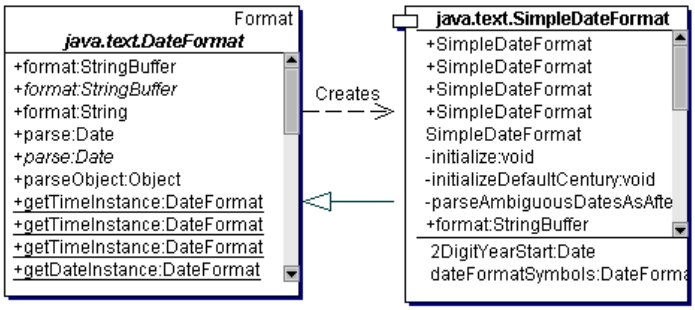
DateFormat 与简单工厂模式

想必大多数的读者都曾经使用工具类 java.text.DateFormat 或其子类来格式化一个本地日期或者时间，这个工具类在处理英语和非英语的日期及时间格式上很有用处。

使用的目的

在本地机器上，时间和日期的格式存在一些标准的风格，比如一个日期可以被格式化为 FULL、LONG、MEDIUM 或者 SHORT，但是它们所代表的意义随着实现的不同而不同。

以英语为例，SHORT 代表完全是数字型的短格式，例如“1/20/2002”或者“15:20”；而 MEDIUM 代表的中格式则是把缩写文字增加到短格式中，比如：“Jul 22, 2002”或者“3:20pm”；而 LONG 代表的长格式用的是完整的词，比如：“July 2, 2002”或者“3:20:10pm”；而 FULL 所代表的全格式包括了完整的日期信息，比如：“Monday, July 22, 2002, AD”或者“3:20:10pm EST”等。DateFormat 与 SimpleDateFormat 的类图如下所示。



getDateInstance()方法

如果仔细考查这个 DateFormat 类就会发现，它是一个抽象类，但是却提供了很多的静态工厂方法，比如 getDateInstance()为某种本地日期提供格式化，它由三个重载的方法组成。

代码清单 14：DateFormat 的部分源代码

```
public final static DateFormat getDateInstance();
public final static DateFormat getDateInstance(int style);
public final static DateFormat getDateInstance(int style, Locale locale);
```

第一次接触这个类的初级程序员会有一些困惑，比如有的人会问，为什么一个抽象类可以有自己的实例，并通过几个方法提供自己的实例。实际上，一个抽象类不能有自己的实例，这是绝对正确的。而应当注意的是，DateFormat 的工厂方法是静态方法，并不是普通的方法。

getDateInstance()方法并没有调用 DateFormat 的构造子来提供自己的实例，作为一个工厂方法，getDateInstance()方法做了一些有趣的事情。它所做的事情基本上可以分成两部分：一是运用多态性；二是使用静态工厂方法。

从上面给出的 DateFormat 和 SimpleDateFormat 的类图可以看出，SimpleDateFormat 是抽象类 DateFormat 的具体子类，这就意味着 SimpleDateFormat 是一个 DateFormat 类型的子类型；而 getDateInstance()方法完全可以返回 SimpleDateFormat 的实例，而仅将它声明为 DateFormat 类型。这就是最纯正的多态性原则的运用[SINTES00]。

getDateInstance()方法是一个静态方法。如果它是一个非静态的普通方法会怎样呢？要使用这个（非静态）方法，客户端必须首先取得这个类的实例或者其子类的实例。而这个类是一个抽象类，不可能有自己的实例，所以客户端就只好首先取得其具体子类的实例。如果客户端能够取得它的子类的实例，那么还需要这个工厂方法干什么呢？

显然，这里使用静态工厂方法是为了将具体子类实例化的工作隐藏起来，从而客户端不必考虑如何将具体子类实例化，因为抽象类 DateFormat 会提供它的合适的具体子类的实例。这是一个简单工厂方法模式的绝佳应用。

针对抽象编程

这样做是利用具体产品类的超类类型将它的真实类型隐藏起来，其好处是提供了系统的可扩展性。如果将来有新的具体子类被加入到系统中来，那么工厂类可以将交给客户端

的对象换成新的子类的实例，而对客户端没有任何影响。

这种将工厂方法的返还类型设置成抽象产品类型的做法，叫做针对抽象编程，这是依赖倒转原则（DIP）的应用。关于这一设计原则的详细讨论，请参见本书的“依赖倒转原则（DIP）”一章。

本地时间

与本地日期的格式化相对应的是为某种本地时间提供格式化，这一功能由三个重载的getTimeInstance()方法提供。

代码清单 15：DateFormat 的部分源代码

```
public final static DateFormat getTimeInstance();
public final static DateFormat getTimeInstance(int style);
public final static DateFormat getTimeInstance(int style, Locale locale);
```

显然它们所提供的也是其具体子类的实例，而不是自身的实例。因为，它自身是一个抽象类，不可能有自己的实例。由于其子类必然是 DateFormat 的子类型，因此返还类型可以是 DateFormat 类型，这也是多态性的体现。

一个法语日期的例子

为了进一步说明这个工具类的使用办法，在下面给出的例子中，假定本地语言是法语，并针对法语进行时间和日期的格式化。

代码清单 16：DateTester 的部分源代码

```
package com.javapatterns.simplefactory.dateformat;
import java.util.Locale;
import java.util.Date;
import java.text.DateFormat;
import java.text.ParseException;
public class DateTester
{
    public static void main(String[] args)
    {
        Locale locale = Locale.FRENCH;
        Date date = new Date();
        String now = DateFormat.getTimeInstance(DateFormat.DEFAULT, locale)
            .format(date);
        System.out.println(now);
        try
        {
            date = DateFormat.getDateInstance(DateFormat.DEFAULT, locale)
                .parse("16 nov. 01");
            System.out.println(date);
        }
        catch(ParseException e)
        {
```

```
                System.out.println("Parsing exception:"+e);
            }
        }
    }
```

其中 now 包含了按照法语格式写出的当前时间，而 date 则读入了以法语方式书写的一个日期“16 nov. 01”。系统运行时会打印出下面的结果。

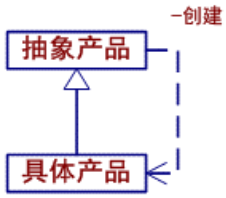
代码清单 17：DateTester 的运行结果

```
13:18:36
Fri Nov 16 00:00:00 PST 2001
```

简单工厂模式的应用

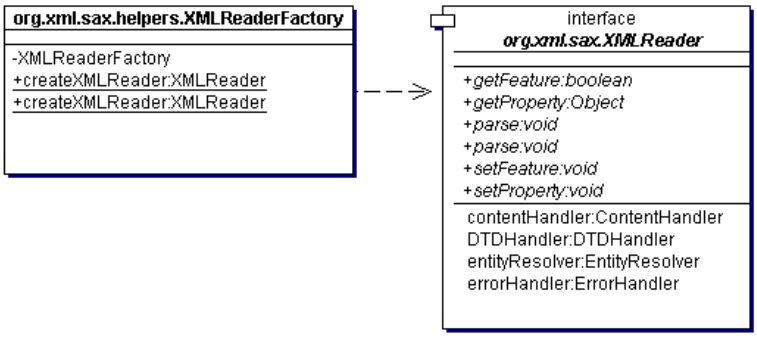
从上面的例子可以看出，由于使用了简单工厂模式，客户端完全不必操心工厂方法所返还的对象是怎样创建和构成的。工厂方法将实例化哪些对象以及如何实例化这些对象的细节隐藏起来，使得对这些对象的使用得到简化。

与一般的简单工厂模式不同的地方在于，这里的工厂角色与抽象产品角色合并成一个类。换言之，抽象产品角色负责具体产品角色的创建，这是简单工厂模式的一个特例。如下图所示。



SAX2 库中的 XMLReaderFactory 与简单工厂模式

在 SAX2 库中，XMLReaderFactory 类使用了简单工厂模式，用来创建产品类 XMLReader 的实例。下面是 XMLReaderFactory 和 XMLReader 类型的关系图。



XMLReaderFactory 提供了两种不同的静态方法，适用于不同的驱动软件参数。

关于 SAX2 库的知识以及 SAX2 库所涉及到的其他模式的讨论，请阅读本书“专题：XMLProperties 与适配器模式”和“专题：观察者模式与 SAX2 浏览器”两章。

12.8　女娲抟土造人

《风俗通》中说：“俗说天地开辟，未有人民。女娲抟黄土为人。”女娲需要用土造出一个个的人，这就是简单工厂模式的应用。

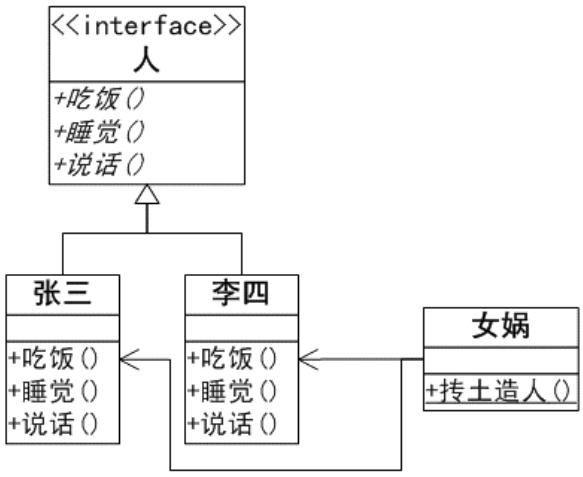
女娲抟土造人的思想便是简单工厂模式的应用。现在本章就试图使用 UML 和模式的语言来解释女娲的做法。首先，在这个造人的思想里面，有几个重要的角色：女娲本身、抽象的人的概念和女娲所造出的具体的人们。

（1）女娲是一个工厂类，也就是简单工厂模式的核心角色；

（2）具体的一个个的人，包括张三、李四等。这些人便是简单工厂模式里面的具体产品角色；

（3）抽象的人便是最早只存在于女娲的头脑里的一个想法，女娲按照这个想法造出的一个一个具体的人便都符合这个抽象的人的定义。换言之，这个抽象的想法规定了所有具体的人必须具有的接口。

女娲抟土造人的 UML 类图如下所示。



问答题

1. 在本节开始时不是说，工厂模式就是在不使用 new 操作符的情况下，将类实例化的吗，可为什么在具体实现时，仍然使用了 new 操作符呢?

2. 请使用简单工厂模式设计一个创建不同几何形状，如圆形，方形和三角形实例的描图员（Art Tracer）系统。每个几何图形都要有画出 draw()和擦去 erase()两个方法。 当描图

员接到指令，要求创建不支持的几何图形时，要提出 BadShapeException 异常。

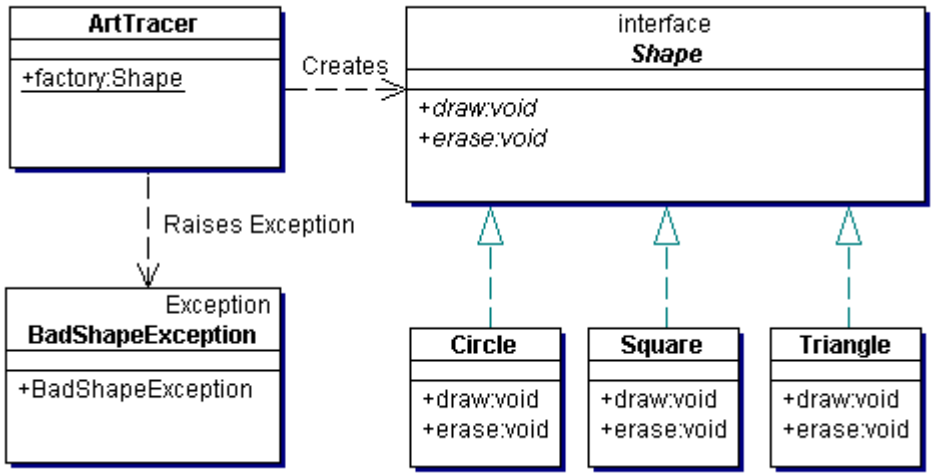
3. 请给出上一题的源代码。

4. 请简单举例说明描图员系统怎样使用。

问答题答案

1. 对整个系统而言，工厂模式把具体使用 new 操作符的细节包装和隐藏起来。当然只要程序是用 Java 语言写的， Java 语言的特征在细节里一定会出现的。

2. 这里给出问题的完整答案。描图员（Art Tracer）系统的 UML 如下图所示。



3. 描图员系统包括一个接口扮演抽象产品角色，三个具体产品类，一个工厂类，和一个 Exception 类。工厂类便是 ArtTracer 类，它的源代码如下。

代码清单 18：ArtTracer 类的源代码

```
public class ArtTracer
{
    /**
     *  静态工厂方法
     */
     public static Shape factory(String which) throws BadShapeException
     {
          if (which.equalsIgnoreCase("circle"))
          {
               return new Circle();
          }
          else if (which.equalsIgnoreCase("square"))
          {
```

```
            return new Square();
        }
        else if (which.equalsIgnoreCase("triangle"))
        {
            return new Triangle();
        }
        else
        {
            throw new BadShapeException(which);
        }
    }
}
```

Shape 是一个 Java 接口，规范出所有的产品类必须实现的方法。它的源代码如下。

代码清单 19：Shape 接口的源代码

```
public interface Shape
{
    void draw();
    void erase();
}
```

Square 是一个具体产品类，实现抽象产品角色，即 Shape 接口。

代码清单 20：Square 类的源代码

```
public class Square implements Shape
{
    public void draw()
    {
        System.out.println("Square.draw()");
    }
    public void erase()
    {
        System.out.println("Square.erase()");
    }
}
```

与上面的 Square 一样，Circle 也是一个具体产品类，实现 Shape 接口。

代码清单 21：Circle 类的源代码

```
public class Circle implements Shape
{
    public void draw()
    {
        System.out.println("Circle.draw()");
    }
    public void erase()
    {
        System.out.println("Circle.erase()");
    }
}
```

Triangle 是一个具体产品类。与上面的两个具体产品类一样，也实现了 Shape 接口。

代码清单 22：Triangle 类的源代码

```
public class Triangle implements Shape
{
    public void draw()
    {
        System.out.println("Triangle.draw()");
    }
    public void erase()
    {
        System.out.println("Triangle.erase()");
    }
}
```

为了提供出错管理，这里特别提供一个 BadShapeException 异常类。如果客户端请求一个并不存在的 Shape 的话，工厂就应当抛出这个异常。

代码清单 23：BadShapeException 类的源代码

```
public class BadShapeException extends Exception
{
    public BadShapeException(String msg)
    {
        super(msg);
    }
}
```

4．描图员(Art Tracer)系统的使用方法如下。

代码清单 24：描图员(Art Tracer)系统的代码

```
try
{
    ArtTracer art = new ArtTracer();
    art.factory("circle");
    art.factory("square");
    art.factory("triangle");
    art.factory("diamond");
}
catch(BadShapeException e)
{
    ...
}
```

对 ArtTracer 类提出菱形（diamond）请求时，会收到 BadShapeException 异常。

参考文献

[SINTES00] Tony Sintes, Polymorphism in its Purest Form - The Nature of Abstract Classes and Polymorphism, JavaWorld (www.javaworld.com), December 2000.