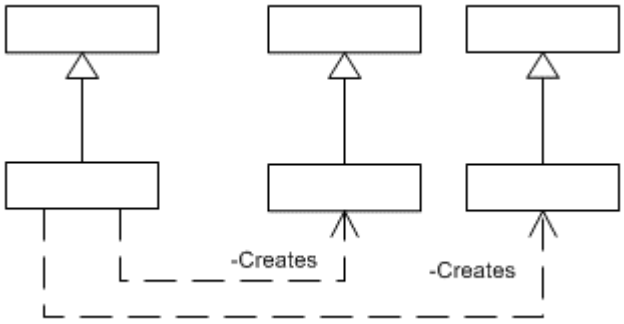
第 14 章　抽象工厂（Abstract Factory）模式

在阅读本章之前，请首先阅读本书的“简单工厂（Simple Factory）模式”以及“工厂方法（Factory Method）模式”两章。

14.1　引　　言

抽象工厂模式的用意

抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式的简略类图如下所示。



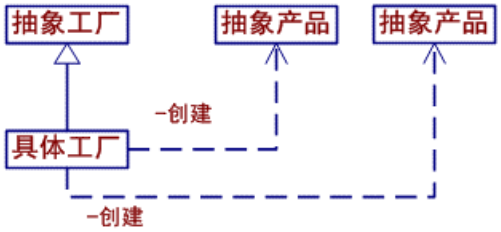
左边的等级结构代表工厂等级结构，右边的两个等级结构分别代表两个不同的产品的等级结构。

抽象工厂模式可以向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，创建多个产品族中的产品对象。这就是抽象工厂模式的用意。

这是什么意思？相信很多读者会有这样的问题。为了说明抽象工厂模式的用意，不妨把它分成三段理解。

第一段

一个系统需要消费多个抽象产品角色，这些抽象产品角色可以用 Java 接口或者抽象 Java 类实现。读过本书的“工厂方法（Factory Method）模式”一章的读者可能会建议，既然客户端需要这些抽象产品角色的实例，为什么不使用一个工厂类负责创建这些角色的实例呢？工厂类负责创建抽象产品的实例描述如下图所示。

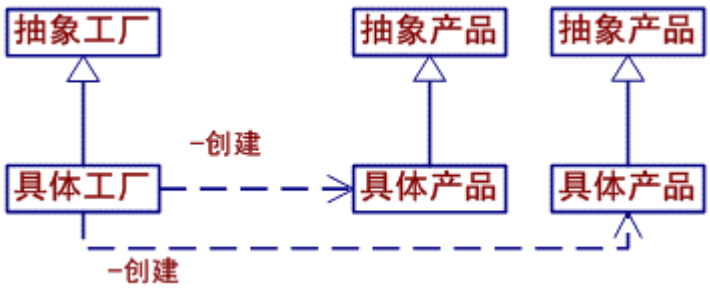


但是，正如上面所指出的，这些抽象产品角色是由 Java 接口或者抽象 Java 类实现的，而一个 Java 接口或者抽象 Java 类是不能实例化的。也就是说，上面的设计是不能成立的。

第二段

那么怎么满足系统的需求呢？

根据里氏代换原则，任何接收父类型的地方，都应当能够接收子类型。因此，实际上系统所需要的，仅仅是类型与这些抽象产品角色相同的一些实例，而不是这些抽象产品的实例。换言之，也就是这些抽象产品的具体子类的实例。工厂类负责创建抽象产品的具体子类的实例如下图所示。



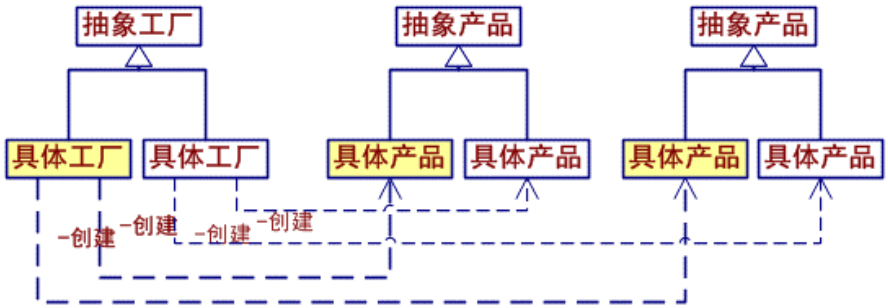
这就是抽象工厂模式用意的基本含义。

第三段

那么接下来的一个问题就是，如果每个抽象产品都有多于一个的具体子类的话，工厂角色怎么知道实例化哪一个子类呢？比如下面的类图中就给出了两个抽象产品，而每一个抽象产品都有两个具体产品。

抽象工厂模式提供两个具体工厂角色，分别对应于这两个具体产品角色。每一个具体工厂角色仅负责某一个具体产品角色的实例化。每一个具体工厂类负责创建抽象产品的某

一个具体子类的实例如下图所示。



涂有阴影的两个具体产品属于同一个产品族，关于产品族的概念，请见后面的讲解。

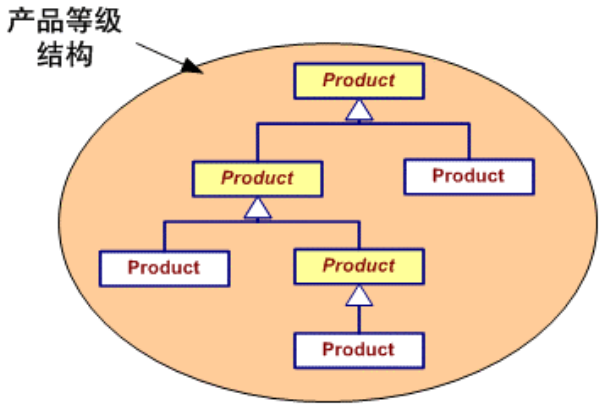
理解了这三个步骤，就不难理解“抽象工厂”这个名字的来源了。“抽象”来自“抽象产品角色”，而“抽象工厂”就是抽象产品角色的工厂。

14.2　问　　题

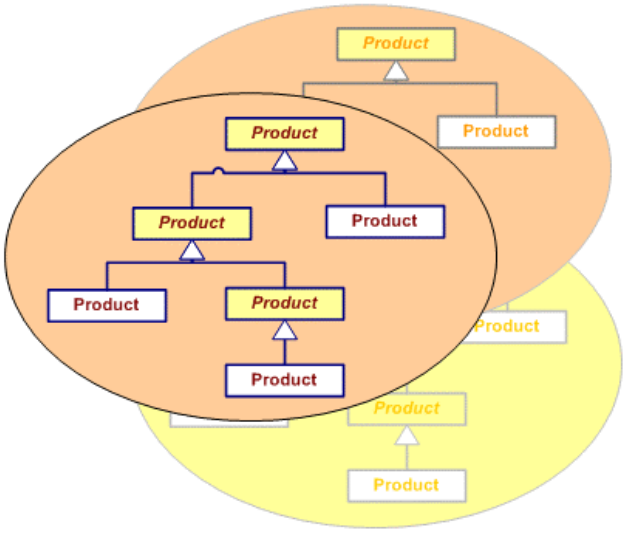
每一个模式都是针对一定问题的解决方案。正如前面所提到的，抽象工厂模式面对的问题是多个产品等级结构的系统设计。下面就从所面对的问题开始，将抽象工厂模式引进到系统设计中。

多个产品等级结构

抽象工厂模式与工厂方法模式的最大区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构。下图给出了一个产品等级结构。

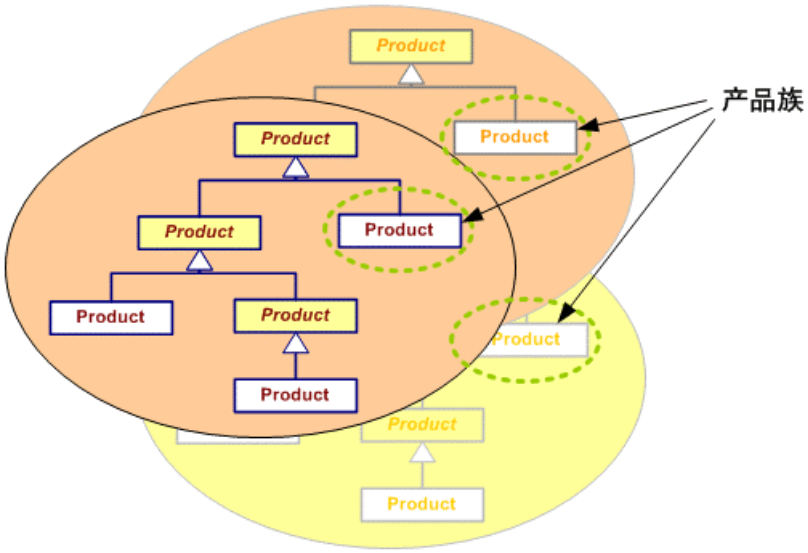


下图则给出了多个相平行的产品等级结构的例子。

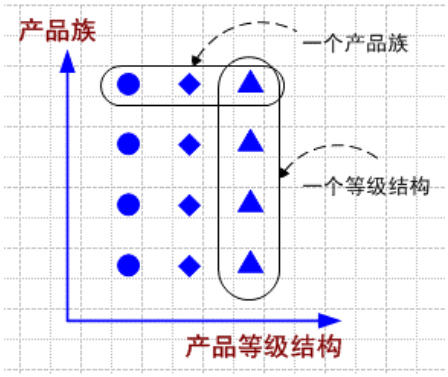


产品族

为了方便引进抽象工厂模式，特地引进一个新的概念：产品族（Product Family）。所谓产品族，是指位于不同产品等级结构中，功能相关联的产品组成的家族。比如在下图中，箭头所指就是三个功能相关联的产品，它们位于三个不同的等级结构中的相同位置上，组成一个产品族。



显然，每一个产品族中含有产品的数目，与产品等级结构的数目是相等的。产品的等级结构和产品族将产品按照不同方向划分，形成一个二维的坐标系，如下图所示。



在坐标图中有四个产品族，分布于三个产品等级结构中。

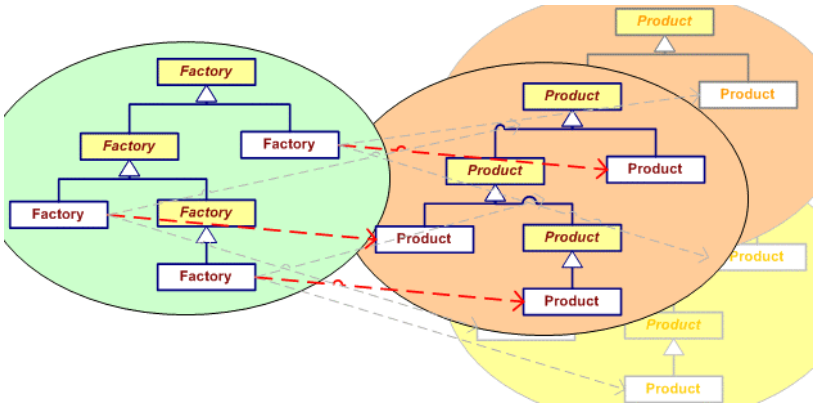
在上面的坐标图中，横轴表示产品等级结构，纵轴表示产品族。可以看出，图中一共有四个产品族，分布于三个不同的产品等级结构中。只要指明一个产品所处的产品族以及它所属的等级结构，就可以惟一地确定这个产品。

这样的坐标图，叫做相图。在一个相图中，坐标轴代表抽象的自由度，相图中两个坐标点之间的绝对距离并没有意义，有意义的是点与点的相对位置。

引进抽象工厂模式

上面所给出的三个不同的等级结构具有平行的结构。因此，如果采用工厂方法模式，就势必要使用三个独立的工厂等级结构来对付这三个产品等级结构。由于这三个产品等级结构的相似性，会导致三个平行的工厂等级结构。随着产品等级结构的数目增加，工厂方法模式所给出的工厂等级结构的数目也会随之增加。

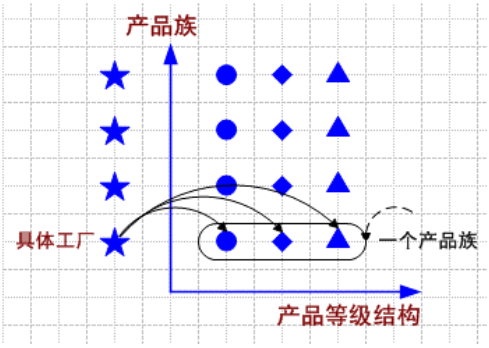
那么，是否可以使用同一个工厂等级结构来对付这些相同或者极为相似的产品等级结构呢？当然是可以的，而这就是抽象工厂模式的好处。同一个工厂等级结构负责三个不同产品等级结构中的产品对象的创建，如下图所示，图中的虚线代表创建关系。



可以看出，一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象；显然，这时候抽象工厂模式比工厂方法模式更有效率。

应当指出的是，虽然大多数的文献都以一个含有两个层次（抽象和具体层次）的产品族作为讲解的例子，但在真实的系统中，产品族往往具有复杂的等级结构，就如同上面的图中所描述的一样，可以具有多于一个的抽象产品和很多的具体产品。

如果使用相图来描述的话，就如下面的类图所示。



在上面的相图中加入了具体工厂角色。可以看出，对应于每一个产品族都有一个具体工厂。而每一个具体工厂负责创建属于同一个产品族、但是分属于不同等级结构的产品。

14.3　抽象工厂模式的结构

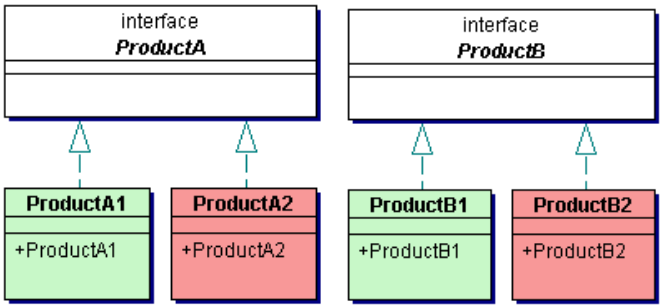
抽象工厂模式[GOF95]是对象的创建模式，它是工厂方法模式的进一步推广。

假设一个子系统需要一些产品对象，而这些产品又属于一个以上的产品等级结构。那么为了将消费这些产品对象的责任和创建这些产品对象的责任分割开来，可以引进抽象工厂模式。这样的话，消费产品的一方不需要直接参与产品的创建工作，而只需要向一个公用的工厂接口请求所需要的产品。

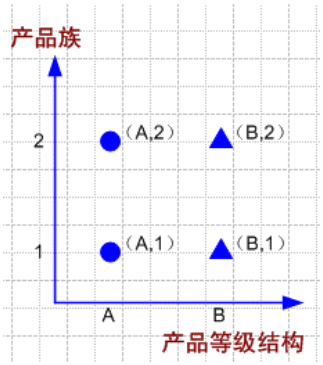
下面就以一个示意性的系统为例，说明这个模式的结构。

产品对象的创建问题

通过使用抽象工厂模式，可以处理具有相同（或者相似）等级结构的多个产品族中的产品对象创建问题。比如下面就是两个具有相同等级结构的产品族 A 和产品等级结构 B 的结构图。

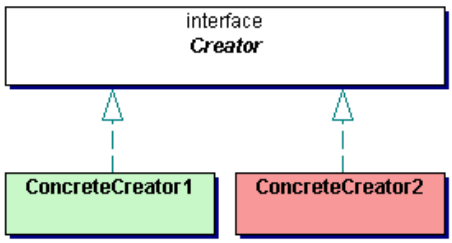


如果使用相图描述的话，会看到在相图上出现两个等级结构 A 和 B，以及两个产品族 1 和 2。如下图所示。

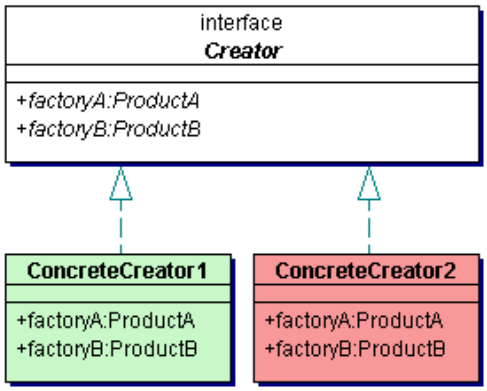


在上面的相图中，每一个坐标点都代表一个具体产品角色。可以看出，坐标点(A,1)，(A,2)，(B,1)和(B,2)分别对应于具体产品角色 ProductA1，ProductA2，ProductB1，ProductB2 等。

就像本章前面所谈到的一样，如果使用工厂方法模式处理的话，就必须要有两个独立的工厂族。由于这两个产品族的等级结构相同，因此，使用同一个工厂族也可以处理这两个产品族的创建问题。后者就是抽象工厂模式，这样根据产品角色的结构图，就不难给出工厂角色的结构设计图，如下图所示。



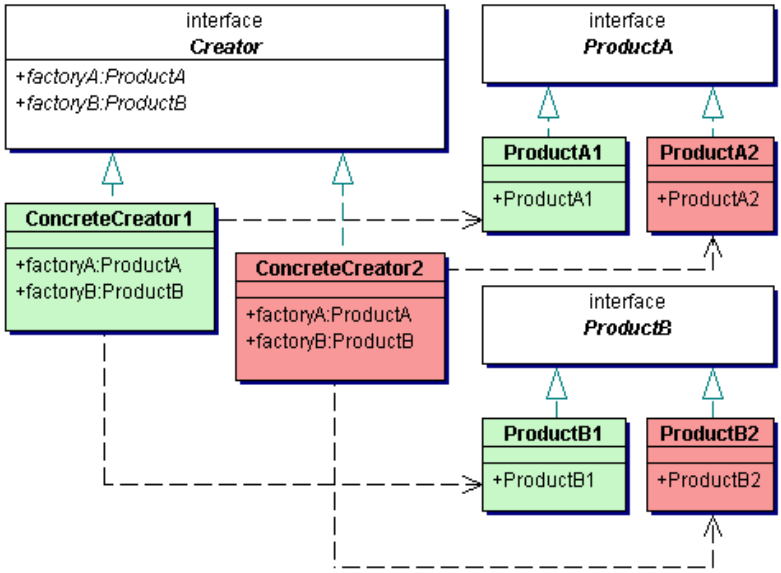
由于每个具体工厂角色都需要负责两个不同等级结构的产品对象的创建，因此每个工厂角色都需要提供两个工厂方法，分别用于创建两个等级结构的产品。既然每个具体工厂角色都需要实现这两个工厂方法，所以这种情况就具有一般性，不妨抽象出来，移动到抽象工厂角色 Creator 中加以声明。产品等级结构 A 和产品等级结构 B 的结构图如下所示。



可以看出，每一个工厂角色都有两个工厂方法，分别负责创建分属不同产品等级结构的产品对象。

系统的设计

采用抽象工厂模式设计出的系统类图如下所示。



从上图可以看出，抽象工厂模式涉及到以下的角色。

- 抽象工厂（AbstractFactory）角色：担任这个角色的是工厂方法模式的核心，它是与应用系统的商业逻辑无关的。通常使用 Java 接口或者抽象 Java 类实现，而所有的具体工厂类必须实现这个 Java 接口或继承这个抽象 Java 类。
- 具体工厂类（Conrete Factory）角色：这个角色直接在客户端的调用下创建产品的实例。这个角色含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的商业逻辑紧密相关的。通常使用具体 Java 类实现这个角色。

- 抽象产品（Abstract 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。通常使用 Java 接口或者抽象 Java 类实现这一角色。
- 具体产品（Concrete Product）角色：抽象工厂模式所创建的任何产品对象都是某一个具体产品类的实例。这是客户端最终需要的东西，其内部一定充满了应用系统的商业逻辑。通常使用具体 Java 类实现这个角色。

源代码

下面给出这个系统所有的源代码。

首先给出工厂角色的源代码，可以看出，抽象工厂角色规定出两个工厂方法，分别提供两个不同等级结构的产品对象。

代码清单 1：抽象产品角色的源代

```
package com.javapatterns.abstractfactory;
public interface Creator
{
    /**
     *   产品等级结构 A 的工厂方法
     */
     public ProductA factoryA();
    /**
     *   产品等级结构 B 的工厂方法
     */
     public ProductB factoryB();
}
```

下面给出具体工厂角色 ConcreteCreator1 的源代码。这个具体工厂类实现了抽象工厂角色所要求的两个工厂方法，分别提供两个产品等级结构中的某一个产品对象。

代码清单 2：具体工厂类 ConcreteCreator1 的源代码

```
package com.javapatterns.abstractfactory;
public class ConcreteCreator1 implements Creator
{
    /**
     *   产品等级结构 A 的工厂方法
     */
     public ProductA factoryA()
     {
          return new ProductA1();
     }
    /**
     *   产品等级结构 B 的工厂方法
     */
     public ProductB factoryB()
```

```
        {
            return new ProductB1();
        }
    }
```

一般而言，有多少个产品等级结构，就会在工厂角色中发现多少个工厂方法。每一个产品等级结构中有多少具体产品，就有多少个产品族，也就会在工厂等级结构中发现多少个具体工厂。

下面给出具体工厂角色 ConcreteCreator2 的源代码。这个具体工厂类实现了抽象工厂角色所要求的两个工厂方法，分别提供两个产品等级结构中的另一个产品对象。

代码清单 3：具体工厂类 ConcreteCreator2 的源代码

```
package com.javapatterns.abstractfactory;
public class ConcreteCreator2 implements Creator
{
    /**
     *  产品等级结构 A 的工厂方法
     */
     public ProductA factoryA()
     {
         return new ProductA1();
     }
    /**
     *  产品等级结构 B 的工厂方法
     */
     public ProductB factoryB()
     {
         return new ProductB1();
     }
}
```

客户端需要的是产品，而不是工厂.在真实的系统中，产品类应当与应用系统的商业逻辑有密切关系。下面是产品等级结构 A 的抽象产品角色，在这个示意性的系统中，这个抽象产品角色是由一个 Java 接口实现的。

代码清单 4：具体产品类 ProductA 的源代码

```
package com.javapatterns.abstractfactory;
public interface ProductA
{
}
```

下面是属于产品等级结构 A 的具体产品类 ProductA1 的源代码。这个具体产品实现了产品等级结构 A 的抽象产品接口。

代码清单 5：具体产品类 ProductA1 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductA1 implements ProductA
```

```
{
    public ProductA1()
    {
    }
}
```

下面是同样属于产品等级结构 A 的具体产品类 ProductA2 的源代码。这个具体产品也实现了产品等级结构 A 的抽象产品接口。

代码清单 6：具体产品类 ProductA2 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductA2 implements ProductA
{
    public ProductA2()
    {
    }
}
```

下面是产品等级结构 B 的抽象产品角色，这个抽象产品角色也是由一个 Java 接口实现的。

代码清单 7：抽象产品角色 ProductB 的源代码

```
package com.javapatterns.abstractfactory;
public interface ProductB
{
}
```

可以看出这是一个标识接口（也就是没有声明任何方法的空接口，请参见本书的“专题：Java 接口”一章）。

下面是属于产品等级结构 B 的具体产品类 ProductB1 的源代码。这个具体产品实现了产品等级结构 B 的抽象产品接口。

代码清单 8：具体产品类 ProductB1 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductB1 implements ProductB
{
   /**
    *  构造子
    */
    public ProductB1()
    {
    }
}
```

下面是属于产品等级结构 B 的具体产品类 ProductB2 的源代码。这个具体产品实现了产品等级结构 B 的抽象产品接口。

代码清单 9：具体产品类 ProductB2 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductB2 implements ProductB
{
    /**
     *  构造子
     */
     public ProductB2()
     {
     }
}
```

在本例中有两个产品等级结构，而每个产品等级结构中又恰好有两个产品，也就是有两个产品族。因此，工厂等级结构中就会出现两个具体工厂（对应于两个产品族）；而每个工厂类中又有两个工厂方法（对应于两个产品等级结构）。

在真实的系统中，产品等级结构的数目与每个产品等级结构中产品的数目（也就是产品族的数目）一般是不相等的。

14.4　在什么情形下应当使用抽象工厂模式

文献[GOF95]指出，在以下情况下应当考虑使用抽象工厂模式：

（1）一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节。这对于所有形态的工厂模式都是重要的；

（2）这个系统的产品有多于一个的产品族，而系统只消费其中某一族的产品；

（上面这一条叫做抽象工厂模式的**原始用意**。）

（3）同属于同一个产品族的产品是在一起使用的，这一约束必须要在系统的设计中体现出来；

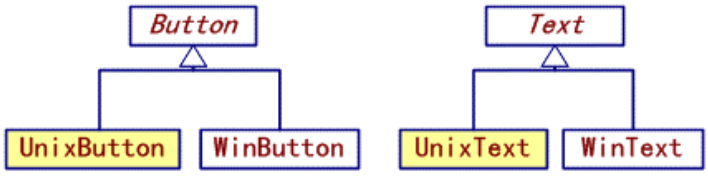
（4）系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

仔细思考一下，很多人都会问这样一个问题：为什么在第二条中说“系统只消费其中某一族的产品”呢？这实际上与抽象工厂模式的起源有关。

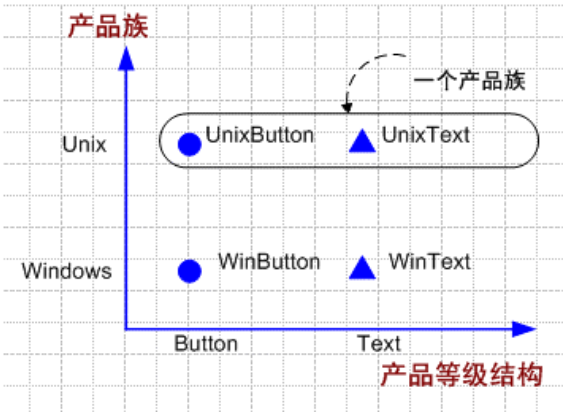
抽象工厂模式的起源

抽象工厂模式的起源或者说最早的应用，是用于创建分属于不同操作系统的视窗构件。比如，命令按键（Button）与文字框（Text）都是视窗构件，在 UNIX 操作系统的视窗环境和 Windows 操作系统的视窗环境中，这两个构件有不同的本地实现，它们的细节也有所不同。

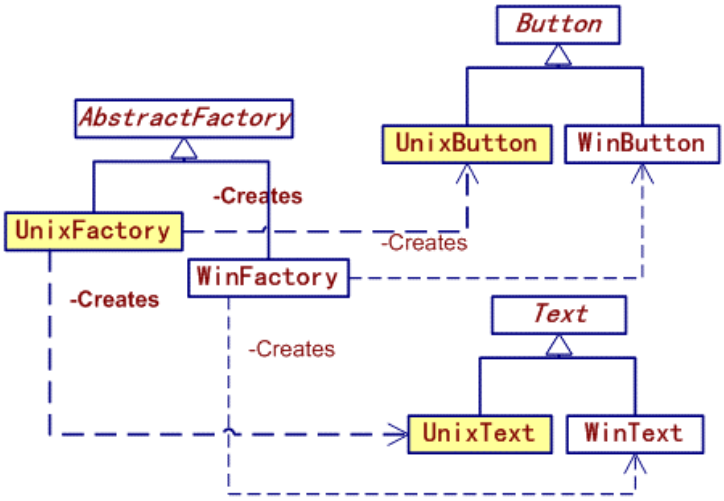
在每一个操作系统中，都有一个视窗构件组成的构件家族。在这里就是 Button 和 Text 组成的产品族。而每一个视窗构件都构成自己的等级结构，由一个抽象角色给出抽象的功能描述，而由具体子类给出不同操作系统下的具体实现，如下图所示。



可以发现在上面的产品类图中，有两个产品的等级结构，分别是 Button 等级结构和 Text 等级结构。同时有两个产品族，也就是 UNIX 产品族和 Windows 产品族。UNIX 产品族由 UnixButton 和 UnixText 产品构成；而 Windows 产品族由 WinButton 和 WinText 产品构成。相图描述如下所示。



系统对产品对象的创建需求由一个工厂的等级结构满足；其中有两个具体工厂角色，即 UnixFactory 和 WinFactory。其中 UnixFactory 对象负责创建 Unix 产品族中的产品，而 WinFactory 对象负责创建 Windows 产品族中的产品。这就是抽象工厂模式的应用，抽象工厂模式的解决方案如下图所示。



显然，一个系统只能够在某一个操作系统的视窗环境下运行，而不能够同时在不同的

操作系统上运行。所以，系统实际上只能消费属于同一个产品族的产品。

在现代的应用中，抽象工厂模式的使用范围已经大大扩大了，不再要求系统只能消费某一个产品族了；因此读者可以不理会前面所提到的原始用意。

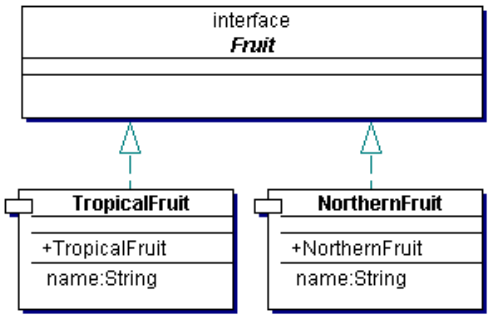
14.5　抽象工厂模式在农场系统中的实现

本节就考查一下如何扩展本书在“简单工厂（Simple Factory）模式”以及“工厂方法（Factory Method）模式”两章中所讨论过的农场系统。

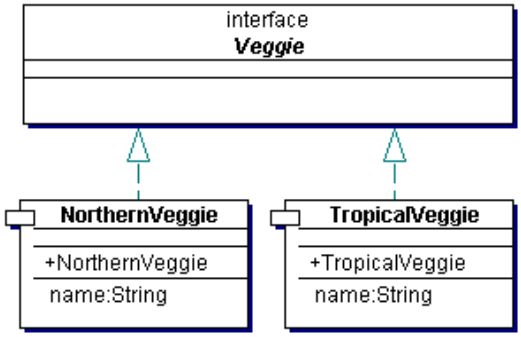
问题

本书在“简单工厂（Simple Factory）模式”与“工厂方法（Factory Method）模式”两章中曾经仔细讨论过一个农场公司从小到大的发展过程。而如今，农场公司再次面临新的大发展，一项重要的工作，就是引进塑料大棚技术，在大棚里种植热带（Tropical）和亚热带的水果和蔬菜。

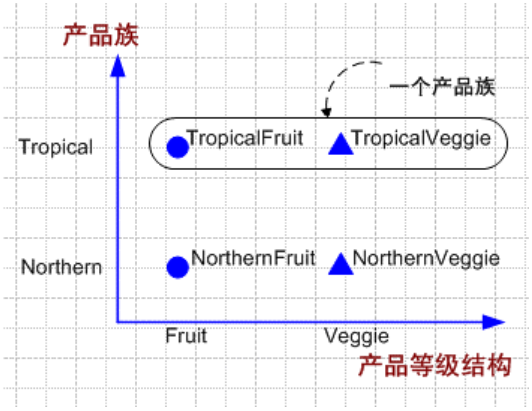
因此，在这个系统里面，产品分成两个等级结构：水果（Fruit）和蔬菜（Veggie）。下面就是水果（Fruit）的类图。



下面则是蔬菜（Veggie）的类图。

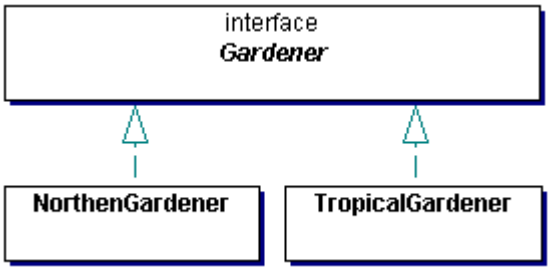


下面是描述这个系统的产品角色的相图。



可以看出，这个系统的产品可以分成两个等级结构：Fruit 和 Veggie，以及两个产品族：Tropical 和 Northern。坐标图上出现了四个坐标点，分别代表 TropicalFruit（热带水果）、TropicalVeggie（热带蔬菜）、NorthernFruit（北方水果）以及 NorthernVeggie（北方蔬菜）等四个产品。

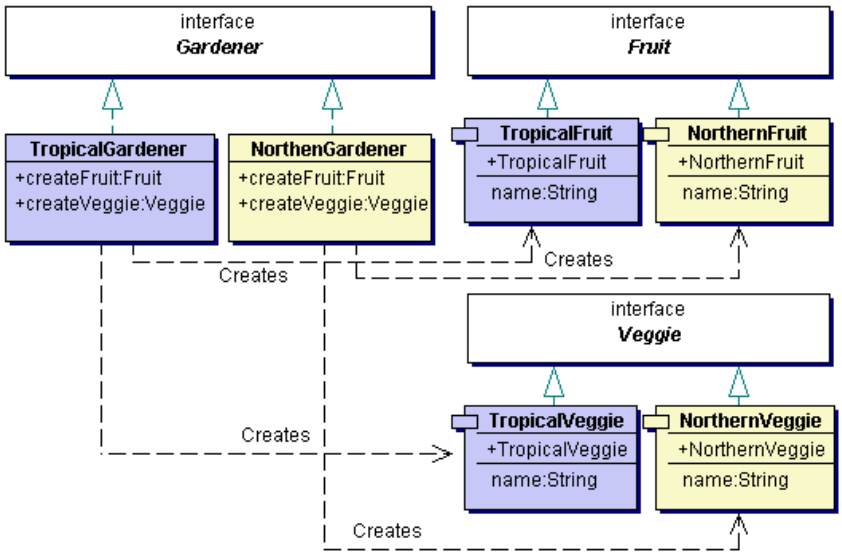
显然可以使用一个工厂族来封装它们的创建过程。这个工厂族的等级结构应当与产品族的等级结构完全平行，园丁等级结构的类图如下图所示。



系统所需要的是产品的实例，而工厂则是对产品创建过程的封装。

系统设计

与抽象工厂模式的各个角色相对照，不难发现，所谓各个园丁其实就是各个工厂角色，而蔬菜和水果角色则是产品角色。将抽象工厂模式应用于农场系统中，系统的设计图如下所示。



种在田间的北方作物与种在大棚的热带作物都是系统的产品，它们分属于两个产品族。显然，北方作物是要种植在一起的，而大棚作物是要另外种植在一起的。这些分别体现在系统的设计上，就正好满足了使用抽象工厂模式的第三个条件。

首先，Gardener 接口是一个没有任何方法的 Java 接口。读过本书的“专题：Java 接口”一章的读者可以看出，这是一个标识接口。

代码清单 10：接口 Gardener

```
public interface Gardener {}
```

NorthernGardener 和 TropicalGardener 均是抽象工厂类 Gardener 的具体子类，也就是说它们全都是具体工厂类。

代码清单 11：具体工厂类 NorthernGardener

```
package com.javapatterns.abstractfactory.farm;
public class NorthernGardener implements Gardener
{
    /**
     *  水果的工厂方法
     */
     public Fruit createFruit(String name)
     {
          return new NorthernFruit(name);
     }
    /**
     *  蔬菜的工厂方法
     */
     public Veggie createVeggie(String name)
     {
```

```
            return new NorthernVeggie(name);
        }
}
```

另一个具体工厂类 TropicalGardener 的源代码如下。

代码清单 12：具体工厂类 TropicalGardener

```
package com.javapatterns.abstractfactory.farm;
public class TropicalGardener implements Gardener
{
    /**
     *   水果的工厂方法
     */
     public Fruit createFruit(String name)
     {
          return new TropicalFruit(name);
     }
    /**
     *   蔬菜的工厂方法
     */
     public Veggie createVeggie(String name)
     {
          return new TropicalVeggie(name);
     }
}
```

显然 Veggie 是一个标识接口。

代码清单 13：接口 Veggie

```
public interface Veggie {}
```

北方的蔬菜 NorthernVeggie 应当实现 Veggie 接口。

代码清单 14：具体产品类 NorthernVeggie

```
package com.javapatterns.abstractfactory.farm;
public class NorthernVeggie implements Veggie
{
     private String name;
     public NorthernVeggie(String name)
     {
     }
     public String getName()
     {
          return name;
     }
     public void setName(String name)
     {
          this.name = name;
```

```
    }
}
```

热带蔬菜 TropicalVeggie 也应当实现 Veggie 接口。
代码清单 15：具体产品类 TropicalVeggie

```
public class TropicalVeggie implements Veggie
{
    private String name;

   /**
    *  构造子
    */
    public TropicalVeggie(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
}
```

抽象产品角色 Fruit 有一个 Java 接口实现，它的源代码如下所示。
代码清单 16：抽象产品角色 Fruit 的源代码

```
package com.javapatterns.abstractfactory.farm;
public interface Fruit
{
}
```

而北方水果 NorthernFruit 类则实现了抽象水果接口 Fruit。
代码清单 17：抽象产品角色 NorthernFruit 的源代码

```
package com.javapatterns.abstractfactory.farm;
public class NorthernFruit implements Fruit
{
    private String name;
    public NorthernFruit(String name)
    {
    }
    public String getName()
    {
        return name;
```

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

同样，热带水果 TropicalFruit 类也实现了抽象水果接口 Fruit。

代码清单 18：抽象产品角色 TropicalFruit 的源代码

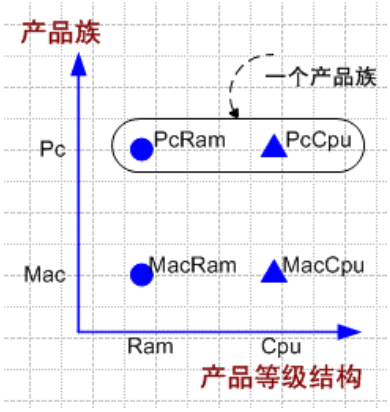
```
package com.javapatterns.abstractfactory.farm;
public class TropicalFruit implements Fruit
{
    private String name;
    public TropicalFruit(String name)
    {
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
}
```

在使用时，客户端只需要创建具体工厂的实例，然后调用工厂对象的工厂方法，就可以得到所需要的产品对象。

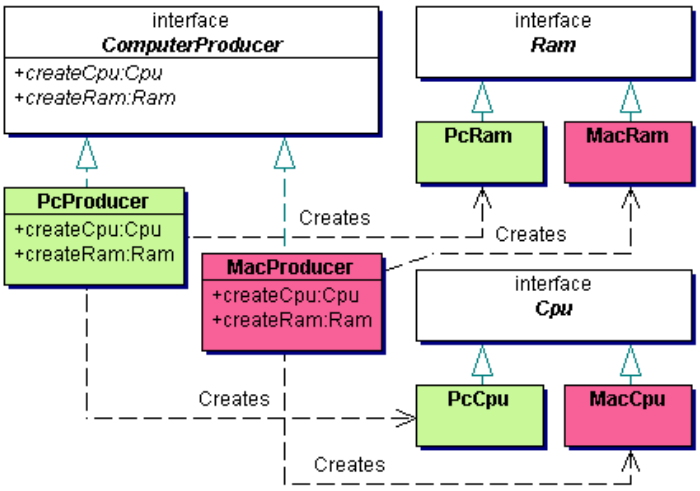
14.6　抽象工厂模式的另一个例子

这个例子描述微型计算机配件的生产。这个系统所需要的产品族有两个，一个系列是 PC，或称 IBM 及 IBM 克隆机系列；另一个系列是 MAC，或称 MacIntosh 系列。产品等级结构也有两个，一个是 RAM，另一个是 CPU。

如果使用相图描述的话，应当是下图所示的样子。



显然，这个系统应该使用抽象工厂模式，而不是工厂方法模式，因为后者适合于处理只有一个产品等级结构的情形。抽象工厂模式应用于微型计算机配件的生产系统中，如下图所示。



两种不同的背景颜色可以区分两个不同的产品族，及其每一个产品族所对应的具体工厂类。

从图中可以看出，每一个工厂角色都提供两个工厂方法，分别对应于两个不同的抽象产品。有多少抽象产品，就会有多少的工厂方法。限于篇幅，不再提供这个系统的详细讨论。但是，读者可以在本章后面的问答题中找到更多的关于这个系统的内容。

14.7　“开-闭”原则

“开-闭”原则要求一个软件系统可以在不修改原有代码的情况下，通过扩展达到增强其功能的目的。对于一个涉及到多个产品等级结构和多个产品族的系统，其功能的增强不

外乎两个方面：

（1）增加新的产品族；

（2）增加新的产品等级结构。

那么抽象工厂模式是怎样支持这两方面功能增强的呢？

增加新的产品族

在产品等级结构的数目不变的情况下，增加新的产品族，就意味着在每一个产品等级结构中增加一个（或者多个）新的具体（或者抽象和具体）产品角色。

由于工厂等级结构是与产品等级结构平行的登记机构，因此，当产品等级结构有所调整时，需要将工厂等级结构做相应的调整。现在产品等级结构中出现了新的元素，因此，需要向工厂等级结构中加入相应的新元素就可以了。

换言之，设计师只需要向系统中加入新的具体工厂类就可以了，没有必要修改已有的工厂角色或者产品角色。因此，在系统中的产品族增加时，抽象工厂模式是支持“开-闭”原则的。

增加新的产品等级结构

在产品族的数目不变的情况下，增加新的产品等级结构。换言之，所有的产品等级结构中的产品数目不会改变，但是现在多出一个与现有的产品等级结构平行的新的产品等级结构。

要做到这一点，就需要修改所有的工厂角色，给每一个工厂类都增加一个新的工厂方法，而这显然是违背“开–闭”原则的。换言之，对于产品等级结构的增加，抽象工厂模式是不支持“开–闭”原则的。

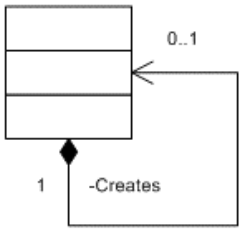
综合起来，抽象工厂模式以一种倾斜的方式支持增加新的产品，它为新产品族的增加提供方便，而不能为新的产品等级结构的增加提供这样的方便。

14.8　相关的模式与模式的实现

抽象工厂模式与一些其它的设计模式有密切的关系；因而在 Java 语言中实现抽象工厂模式时，有下面一些值得注意的地方。

具体工厂类与单例模式

具体工厂类可以设计成单例类。单例模式的简略类图如下所示。

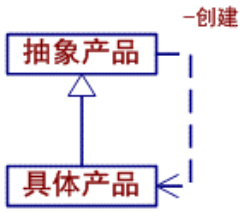


一个单例类只有一个实例，它自己向外界提供自己惟一的实例。关于单例类的知识，请见本书的“单例（Singleton）模式”一章。

很显然，在农场系统中，只需要 NorthernGardener 和 TropicalGardener 的一个实例就可以了。在计算机生产的例子中，PcProducer 和 MacProducer 也分别只需要一个实例。因此，这两个具体工厂角色都可以设计成单例类。

工厂的工厂（之一）

在本书的“简单工厂（Simple Factory）模式”一章中，曾经谈到在简单工厂模式中，工厂角色可以与抽象产品角色合并，并以 java.util.DateFormat 为例子讲解这一做法。如下图所示。



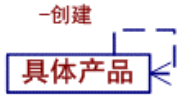
这种做法可以应用到工厂等级结构中去，只是注意在这里“产品”不是产品角色，而是具体工厂角色。

在抽象工厂模式中，抽象工厂类可以配备静态方法，以返还具体工厂的实例。具体地

讲，抽象工厂角色可以配备一个静态方法，这个方法按照参量的值，返回所对应的具体工厂的实例。静态方法的返还类型是抽象工厂类型，这样可以在多态性的保证之下，允许静态工厂方法自行决定哪一个具体工厂符合要求。

工厂的工厂（之二）

在本书的“简单工厂（Simple Factory）模式”一章中，还曾经谈到在简单工厂模式中，工厂角色可以与具体工厂角色合并，如下图所示。



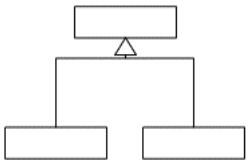
这种做法也可以应用到工厂等级结构中去，只是注意在这里“产品”不是产品角色，而是具体工厂角色。

这就意味着要为每一个具体工厂类配备一个静态方法，而其返还类型是该具体工厂类自己。

通过工厂方法模式或者原始模型模式实现

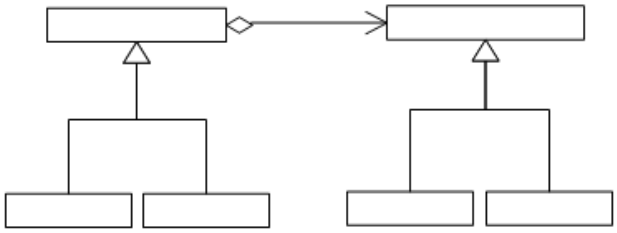
在本章的所有例子中，抽象工厂模式都是通过工厂方法模式实现的。换言之，每一个工厂角色都配有一个工厂方法，这个方法负责调用产品的构造子，将产品角色实例化。

抽象工厂模式完全可以使用原始模型模式而不是工厂方法模式实现。读者可以参阅本书的“原始模型（Prototype）模式”一章，这里不拟赘述。原始模型模式的简略类图如下图所示。



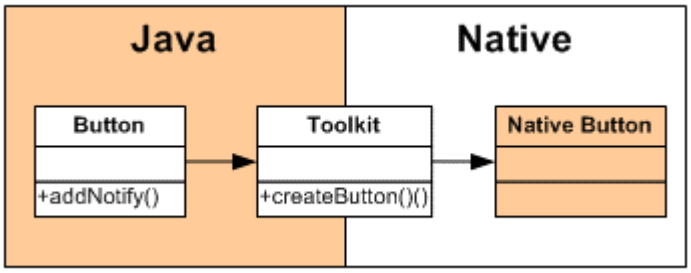
桥梁模式

抽象工厂模式可以为桥梁模式提供某一个等级结构的创建功能；抽象模式可以与桥梁模式一同使用。下面就是桥梁模式的简略类图。



在 Java 的 AWT 库中，定义了两套平行的等级结构；一套是 Java 的构件，以 Component 为超类，另一套是所谓的 peer 构件，以 ComponentPeer 为超类。Java 构件向 Java 程序提供一套与操作系统无关的、统一的构件接口；而 peer 构件则处理底层的、与操作系统密切相关的功能。

在这两个等级结构之间的，是 java.awt.Toolkit 类；这个抽象类在不同的操作系统中有不同的具体子类，并为每一个 peer 构件提供了相应的工厂方法，以创建并且返还一个 peer 构件的实例。下面的图显示了 Java 构件 Button 通过调用 Toolkit 工厂对象创建一个 ButtonPeer 对象的情况。



上面是 Button 和 ButtonPeer 的通讯图。Button 是一个 Java 构件，通过 Toolkit 对象与本地 peer 构件通讯。在这个结构中，Toolkit 就是抽象工厂角色，它的具体实现是具体工厂角色；而各个 peer 对象就是具体产品角色。peer 对象根据自己的分类形成继承的等级结构；而根据操作系统划分成不同的产品族。

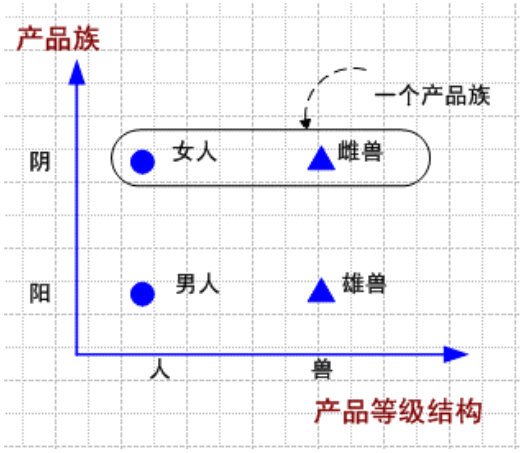
这两个等级结构之间通过委派进行通讯，形成桥梁模式的结构。详细的讨论，请参见本章后面的附录“附录：Java AWT 的 Peer 架构与抽象工厂模式”，以及本书的“桥梁（Bridge）模式”一章。

14.9　女娲造万物的故事

《说文解字》在解释“娲”时，云：“古之神圣女，化万物者”。换言之，女娲不仅仅造了人，而且造了世间万物，这也包括各种动物。因此，女娲一定是把举绳造人的方法推广应用到了创造各种动物身上。可以想到，女娲把绳子搅到泥水里，然后把沾满泥水的绳

子凭空一甩，甩出的泥点，像人的变成了人，像各种其他动物的泥点则变成了其他的动物。女娲的阴绳造出的是女人和雌动物，阳绳造出的是男人和雄动物。

读者可以看出，女娲造物用的是抽象工厂模式。在这个故事里面，女娲的“产品”有两个划分方法：一是按照“产品”是人还是兽来划分，二是按照“产品”是男女、雌雄来划分。女娲的绳子按照阴、阳划分，产品则按照人、兽划分。女娲造万物系统里阴、阳两个等级结构和人、兽两个产品族的类图如下所示。



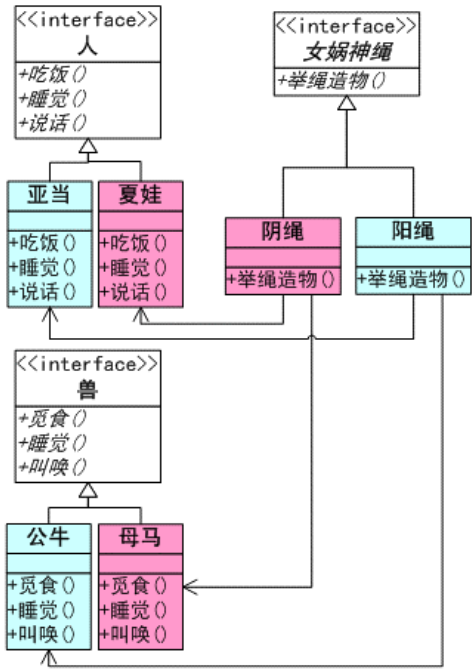
换言之，女娲的故事里面有如下的抽象角色：

- “神绳”接口作为抽象工厂角色；
- “人”接口作为人类抽象角色。在女娲造人之前，一定在头脑里有了一个对人的样子的想像，这个想像就是对人的抽象；
- “兽”接口作为兽类抽象角色。在女娲造各种野兽之前，一定在头脑里有了一个对野兽样子的想像，这个想像就是对野兽的抽象。

以及继承自抽象角色的具体角色：

- 阴绳、阳绳继承自“神绳”接口，是具体的绳子类；
- 亚当、夏娃继承自“人”接口，是具体的人类。本书选用亚当和夏娃为例，是为了张显中西合璧的妙处；
- 公牛、母马继承自“兽”接口，是具体的野兽类，它必须符合女娲对野兽的设想。

将抽象工厂模式应用于女娲造万物的模拟系统设计中。如下图所示，两种颜色代表阴、阳两种系列。



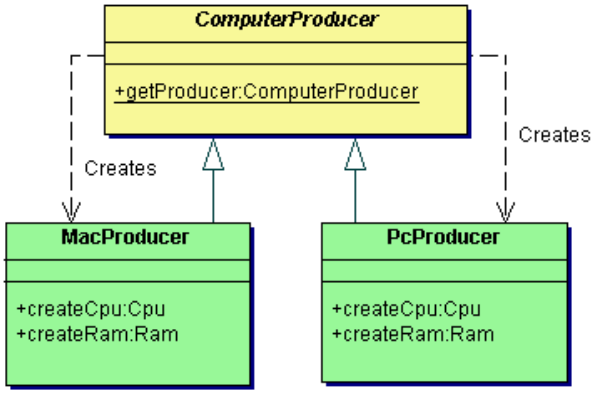
显然，产品类有两个族，为阴和阳（男和女或公和母）；以及两个产品等级结构，为人和兽。这种多于一个产品等级结构的情况，是需要抽象工厂模式的关键原因。

问答题

1. 如上面的讨论，抽象工厂类可以配备一个静态方法，按照参量的值，返回所对应的具体工厂。请把微型计算机生产系统的抽象工厂类按照这一方案改造，给出类图和源代码。
2. 如上面的讨论，具体工厂类可以设计成单例类。请在第 1 题的基础上把微型计算机生产系统的具体工厂类按照这一方案改造，给出 UML 类图和源代码。
3. 请问相图与 UML 图是什么关系？
4. 请使用相图描述一下工厂方法模式。
5. 请使用相图描述一下简单工厂模式。

问答题答案

1. 微型计算机生产系统的抽象工厂原本是接口，现在需要改造成抽象类。如下图所示，其 ComputerProducer 的类名为斜体，表明该类是抽象的；而 getProducer()的下划线，表明该方法是静态的。

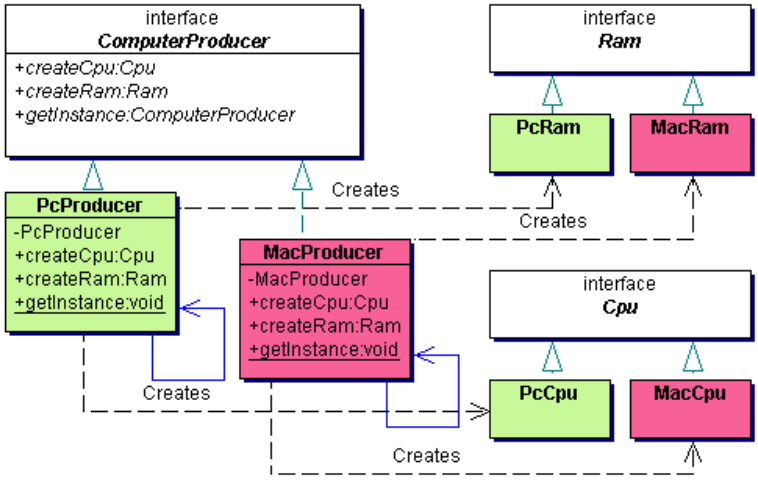


经过单例模式改造的 ComputerProducer 的源代码如下所示。

代码清单 19：抽象类 ComputerProducer 的方法 getProducer(String which)

```
public class ComputerProducer
{
    public static ComputerProducer getProducer(String which)
    {
        if (which.equalsIgnoreCase("Pc"))
        {
            return new PcProducer();
        }
        else if (which.equalsIgnoreCase("Mac"))
        {
            return new MacProducer();
        }
        else
        {
            return null;
        }
    }
}
```

2. 本题答案是建立在第 1 题基础之上的。如下图所示，三种不同的背景颜色可以区分抽象工厂类，两类产品族，及其对应的具体工厂类。



ComputerProducer 类为斜体，表明该类是抽象的，而 getInstance()的下划线表明该方法是静态的。MacProducer 和 PcProducer 的构造子是私有的，因此，这两个类必须自己将自己实例化。

抽象工厂类 ComputeProducer 的源代码如下所示。

代码清单 20：抽象工厂类 ComputerProducer

```
abstract public class ComputerProducer
{
    public static ComputerProducer getInstance(String which)
    {
        if (which.equalsIgnoreCase("Pc"))
        {
            return PcProducer.getInstance();
        }
        else if (which.equalsIgnoreCase("Mac"))
        {
            return MacProducer.getInstance();
        }
    }
}
```

具体工厂类 MacProducer 的源代码如下所示。

代码清单 21：具体工厂类 MacProducer 是单例类

```
public class MacProducer extends ComputerProducer
{
    private static MacProducer m_MacProducer =
        new MacProducer();
   /**
    *  私有的构造子，保证外界不能直接实例化
    */
    private MacProducer()
```

```
    {
    }
   /**
    *   工厂方法，返还产品实例
    */
    public Cpu createCpu()
    {
        return new MacCpu();
    }
   /**
    *   工厂方法，返还产品实例
    */
    public Ram createRam()
    {
        return new MacRam();
    }
   /**
    *   静态工厂方法，返还单例实例
    */
    public static MacProducer getInstance()
    {
        return m_MacProducer;
    }
}
```

读过本书“单例（Singleton）模式”一章的读者应当知道， 这里使用的单例类实现方法是饿汉式方法。

具体工厂类 PcProducer 的源代码如下所示。

代码清单 21：具体工厂类 MacProducer 是单例类

```
public class PcProducer extends ComputerProducer
{
    private static PcProducer m_PcProducer =
        new PcProducer();
   /**
    *   私有的构造子，保证外界不能直接实例化
    */
    private PcProducer()
    {
    }
   /**
    *   工厂方法，返还产品实例
    */
    public Cpu createCpu()
    {
        return new PcCpu();
    }
```

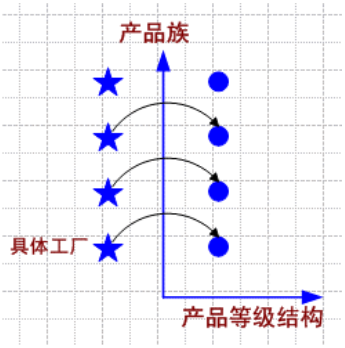
```
    /**
     *   工厂方法，返还产品实例
     */
     public Ram createRam()
     {
          return new PcRam();
     }

    /**
     *   静态工厂方法，返还单例实例
     */
     public static PcProducer getInstance()
     {
          return m_PcProducer;
     }
}
```

使用的单例类实现方法是饿汉式方法。 各产品类没有变化，因此不在此重复。

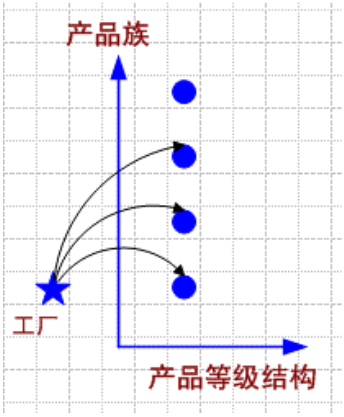
3. 相图与 UML 图没有关系。

4. 在工厂方法模式中，只有一个产品的等级结构，工厂的等级结构与产品的等级结构是平行的。因此，不难给出这个模式的相图，如下图所示。



图中的左侧有一列具体工厂，右侧有一列具体产品，从工厂到产品是创建关系。

5. 在简单工厂模式中，只有一个工厂角色和一个产品的等级结构。因此。不难给出这个模式的相图，如下图所示。



参考文献

[ZUKOWSKI97] John Zukowski, *Java AWT Reference*, published by O'Reilly, 1997.

19．10 附录：Java AWT 的 Peer 架构与抽象工厂模式

在“抽象工厂（Abstract Factory）模式”一章中的“抽象工厂模式的起源”一节中，本书使用示意性的结构讲解了怎样将抽象工厂模式应用到多个操作系统的视窗构件的创建上。实际上，在 Java 语言的 AWT 库里，确实使用了抽象工厂模式创建分属于不同操作系统的 peer 构件。

本节就对 AWT 这个库，特别是库中的 Toolkit 类及其子类作一个考察，以说明抽象工厂模式是怎么应用到 AWT 库中的。

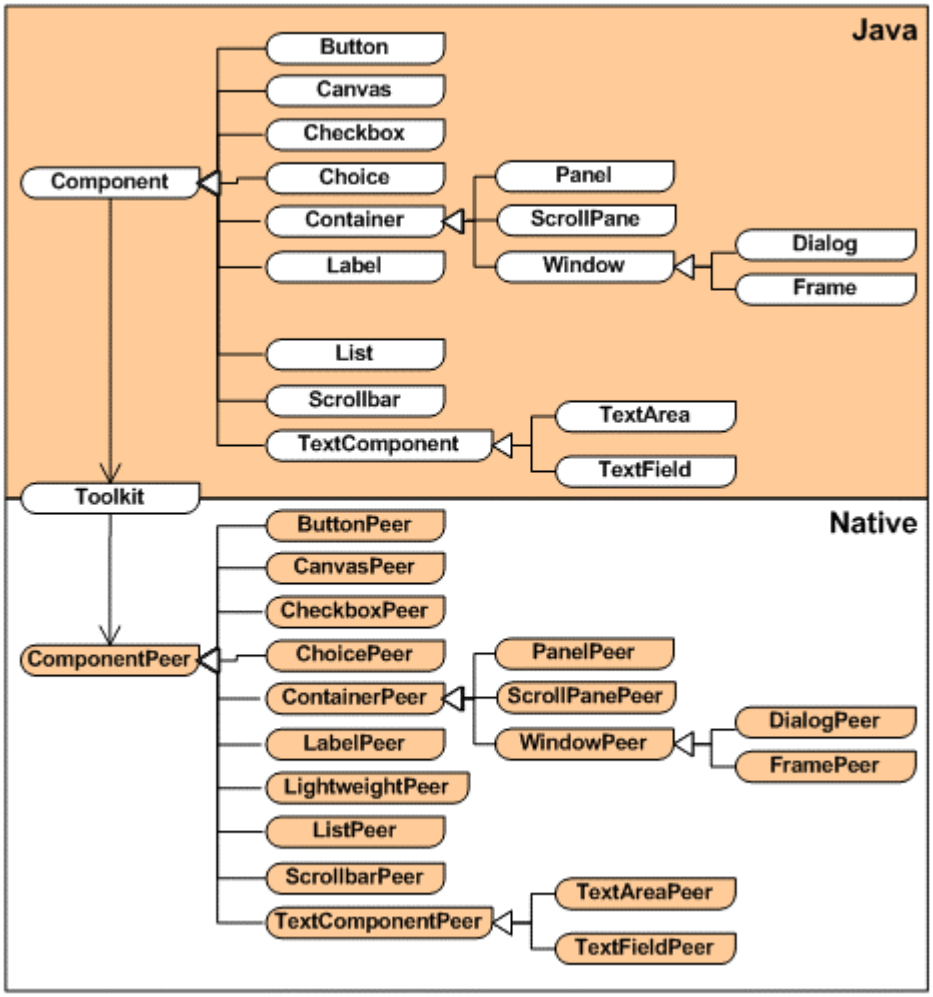
Peer 架构

就从 Java 用户界面的视感谈起吧。

一个由用户界面的 Java 程序总是显示所在操作系统的视感（Look and Feel）；而 Java 程序内部却总是对一个统一的构件接口编程。这是由于在这个同一个构件接口后面，还存在着一层软件结构，叫做 Peer 接口；在这个接口和 Java 构件接口之间，存在着 Toolkit 接口，负责 peer 对象的创建。这种三重接口的架构，叫做 Peer 架构；显然，Peer 架构是桥梁模式的应用。

每有一个 Java 视窗构件，就有一个对应的 peer 接口；而这个接口在不同的操作系统中有不同的实现。在运行时，Peer 架构会自行产生一个对应于当前操作系统的 Toolkit 对象。在运行时，Peer 架构会自行调用这个 Toolkit 对象，创建出所需要的 peer 对象。

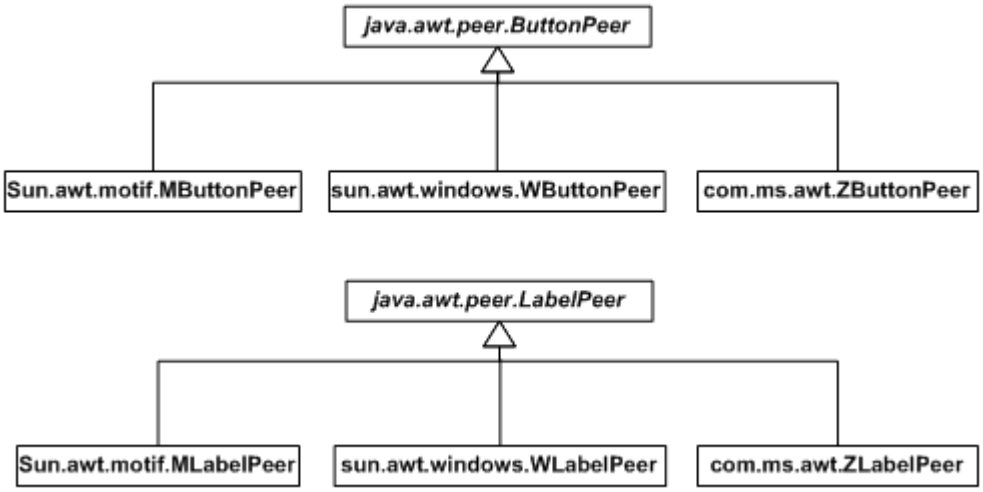
在下面的图中，Component 类型和 ComponentPeer 类型通过 Toolkit 对象相互通讯。



上面的结构图分成上下两个部分，下方的部分给出的就是 java.awt.peers 库中所有的 Peer 接口。关于 Peer 接口与 Peer 架构，读者可以参考阅读本书的“桥梁（Bridge）模式”一章。

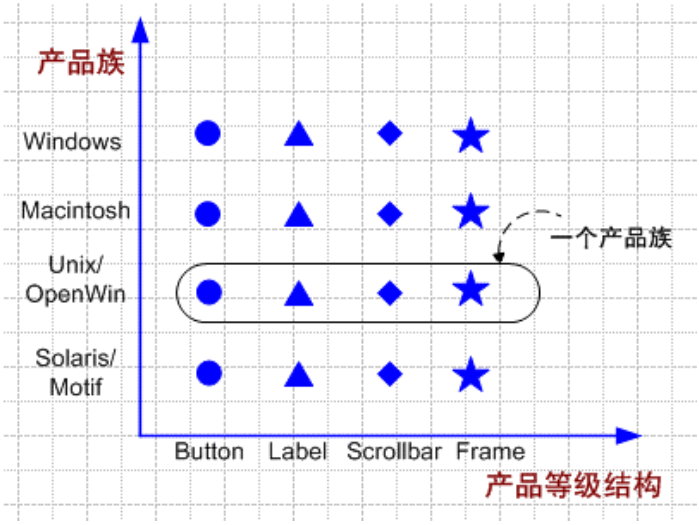
产品角色与 Peer 对象

在下图中，显示了两个产品等级结构，即 ButtonPeer 和 LabelPeer 等级结构；以及三个产品族，即 Windows 产品族，Solaris/Motif 产品族，以及 Unix/OpenWin 产品族。



上面的图显示了 Peer 构件形成产品的等级结构。

如果读者认真看一看 JDK 所带的源代码库的话，可以看到更多的产品等级结构，包括 Window，Frame，Scrollbar 等等；也会看到更多的产品族，比如 Macintosh 产品族等。换言之，每一个支持 Java 的操作系统都必须提供为所有的视窗构件的等级结构提供一个相应的实现。请参加下面的相图：

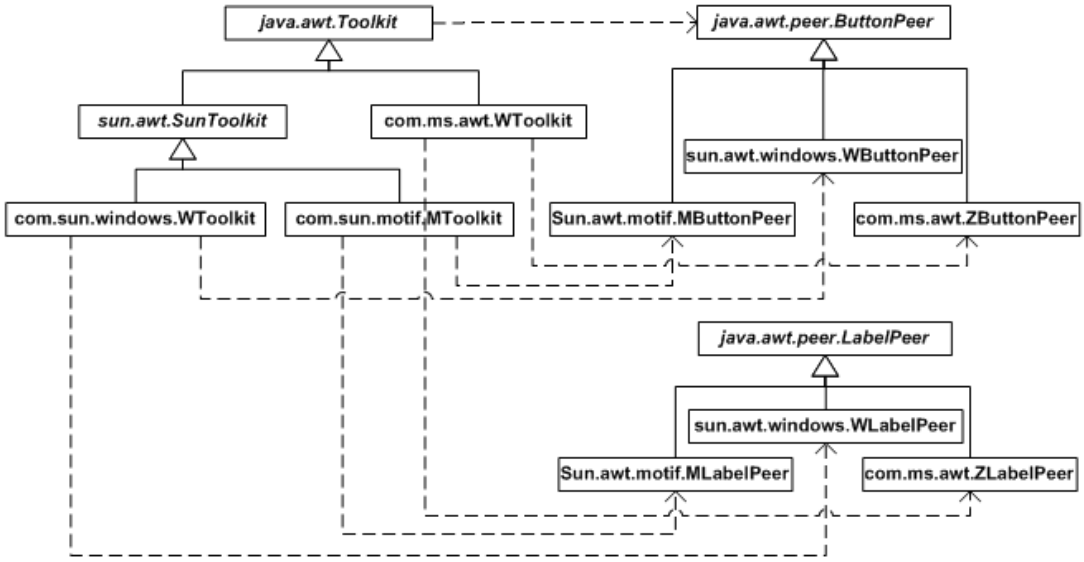


上面是 Peer 构件形成产品的相图。显然，由于一个程序不可能同时在两个操作系统中运行，因此在每一个时刻只可能有一个产品族的产品被创建和消费。这就是说，抽象工厂模式的原始用意在这里是被满足的。

实际上，要想收集到所有的产品族的产品（如 ZButton，WButton，以及 MButton 等分别是为 Windows，Unix/OpenWin，以及 Solaris/Motif 准备的 peer 构件）并不容易。因为 JDK 是分别为不同的操作系统准备的，所以在每一个版本的 JDK 中，只能找到相关操作系统的 peer 构件。

工厂角色与 Toolkit

对应地，图中还显示了一个工厂等级结构。这个等级结构的超类是 java.awt.Toolkit。每一个支持 Java 的操作系统都必须提供一个 Toolkit 的具体子类。读者可以看出，图中显示的三个 Toolkit 的具体子类分别对应于三个不同的操作环境；而如果添加上 Macintosh 的话，就应当再加上一个 Toolkit 的具体子类（应当是 sun.awt.macos.MToolkit）。



从上面的图可以看出各个 Toolkit 对象位于工厂等级结构中，负责创建 ButtonPeer 等级结构和 LabelPeer 等级结构中的对象。

虽然 Toolkit 是 Java API 中使用较为频繁的一个；但是在大多数情况下它是在背后运行的，普通的 Java 引用程序很少会需要直接创建一个 Toolkit 类的实例。

如果程序需要一个 Toolkit 对象，就应当调用它的静态工厂方法 getDefaultToolkit()，或者 Component.getToolkit()方法，以得到一个正确的 Toolkit 对象。

桥梁模式的应用

在 Peer 架构中还使用了桥梁模式。请参见本书的“桥梁（Bridge）模式”一章。

参考文献

[ZUKOWSKI97] John Zukowski, *Java AWT Reference*, published by O'Reilly, 1997.