第5章

抽象工厂模式

本章导学

抽象工厂模式是常用的创建型设计模式之一，它比工厂方法模式的抽象程度更高。在工厂方法模式中，每一个具体工厂只需要生产一种具体产品，但是在抽象工厂模式中，一个具体工厂可以生产一组相关的具体产品，这样的一组产品称为产品族，产品族中的每一个产品都分属于某一个产品继承等级结构。

本章将通过实例来学习抽象工厂模式，分析抽象工厂模式的结构及特点，并学习如何在实际软件项目开发中合理地使用抽象工厂模式。

本章知识点

- 产品等级结构与产品族。
- 抽象工厂模式的定义。
- 抽象工厂模式的结构。
- 抽象工厂模式的实现。
- 抽象工厂模式的应用。
- 抽象工厂模式的优缺点。
- 抽象工厂模式的适用环境。
- 开闭原则的倾斜性。

5.1 产品等级结构与产品族

工厂方法模式通过引入工厂等级结构，解决了简单工厂模式中工厂类职责太重的问题，但由于工厂方法模式中的每个具体工厂只有一个或者一组重载的工厂方法，只能生产一种产品，可能会导致系统中存在大量的工厂类，势必会增加系统的开销。有时可能需要一个工厂能够提供多种产品对象，而不是单一的产品对象，例如一个电器工厂，它可以生产电视机、电冰箱、空调等多种电器，而不是只生产某一种电器。此时，可以考虑将一些相关的产品组

成一个“产品族”，由同一个工厂来统一生产，这就是本章将要学习的抽象工厂模式的基本思想。

为了更好地理解抽象工厂模式，先引入以下两个概念。

（1）产品等级结构：产品等级结构即产品的继承结构，例如一个抽象类是电视机，其子类包括海尔电视机、海信电视机、TCL 电视机，则抽象电视机与具体品牌的电视机之间构成了一个产品等级结构，抽象电视机是父类，而具体品牌的电视机是其子类。

（2）产品族：在抽象工厂模式中，产品族是指由同一个工厂生产的，位于不同产品等级结构中的一组产品。例如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品等级结构中，海尔电冰箱位于电冰箱产品等级结构中，海尔电视机、海尔电冰箱构成了一个产品族。

产品等级结构与产品族示意图如图 5-1 所示。

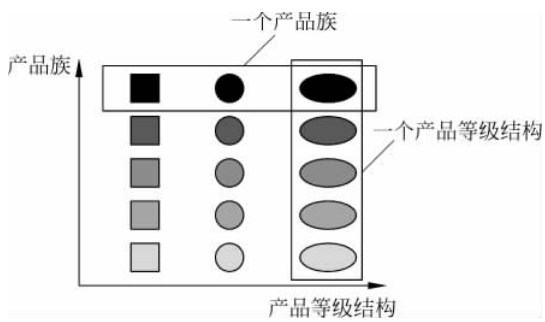

图 5-1　产品等级结构与产品族示意图

在图 5-1 中，不同颜色的多个正方形、圆形和椭圆形分别构成了 3 个不同的产品等级结构，而相同颜色的正方形、圆形和椭圆形构成了一个产品族，每一个形状对象都位于某个产品族，并属于某个产品等级结构。在图 5-1 中共有 5 个产品族，分属于 3 个不同的产品等级结构，只要指明一个产品所处的产品族以及它所属的等级结构，就可以唯一地确定这个产品。

5.2　抽象工厂模式概述

当系统所提供的工厂生产的具体产品并不是一个简单的对象，而是多个位于不同产品等级结构、属于不同类型的具体产品时就可以使用抽象工厂模式。

抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形式。抽象工厂模式与工厂方法模式最大的区别在于，工厂方法模式针对的是一个产品等级结构，而抽象工厂模式需要面对多个产品等级结构，一个工厂等级结构可以负责多个不同产品等级结构中的产品对象的创建。当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时，抽象工厂模式比工厂方法模式更为简单、更有效率。抽象工厂模式示意图如图 5-2 所示。

在图 5-2 中，每一个具体工厂可以生产属于一个产品族的所有产品，例如生产颜色相同的正方形、圆形和椭圆形，所生产的产品又位于不同的产品等级结构中。如果使用工厂方法

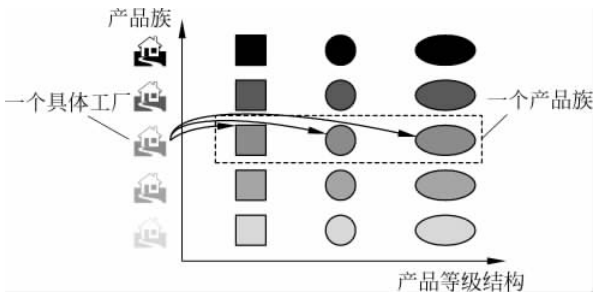

图 5-2 抽象工厂模式示意图

模式,图 5-2 所示的结构需要提供 15 个具体工厂,而使用抽象工厂模式只需要提供 5 个具体工厂,极大地减少了系统中类的个数。

抽象工厂模式为创建一组对象提供了一种解决方案。与工厂方法模式相比,抽象工厂模式中的具体工厂不只是创建一种产品,而是负责创建一族产品。

抽象工厂模式的定义如下:

> **抽象工厂模式**:提供一个创建一系列相关或相互依赖对象的接口,而无须指定它们具体的类。
>
> **Abstract Factory Pattern**: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

抽象工厂模式又称为工具(Kit)模式,它是一种对象创建型模式。

5.3 抽象工厂模式的结构与实现

5.3.1 抽象工厂模式的结构

在抽象工厂模式中,每一个具体工厂都提供了多个工厂方法用于产生多种不同类型的产品,这些产品构成了一个产品族。抽象工厂模式的结构如图 5-3 所示。

由图 5-3 可知,抽象工厂模式包含以下 4 个角色。

(1) **AbstractFactory(抽象工厂)**:它声明了一组用于创建一族产品的方法,每一个方法对应一种产品。

(2) **ConcreteFactory(具体工厂)**:它实现了在抽象工厂中声明的创建产品的方法,生成一组具体产品,这些产品构成了一个产品族,每一个产品都位于某个产品等级结构中。

(3) **AbstractProduct(抽象产品)**:它为每种产品声明接口,在抽象产品中声明了产品所具有的业务方法。

(4) **ConcreteProduct(具体产品)**:它定义具体工厂生产的具体产品对象,实现抽象产品接口中声明的业务方法。

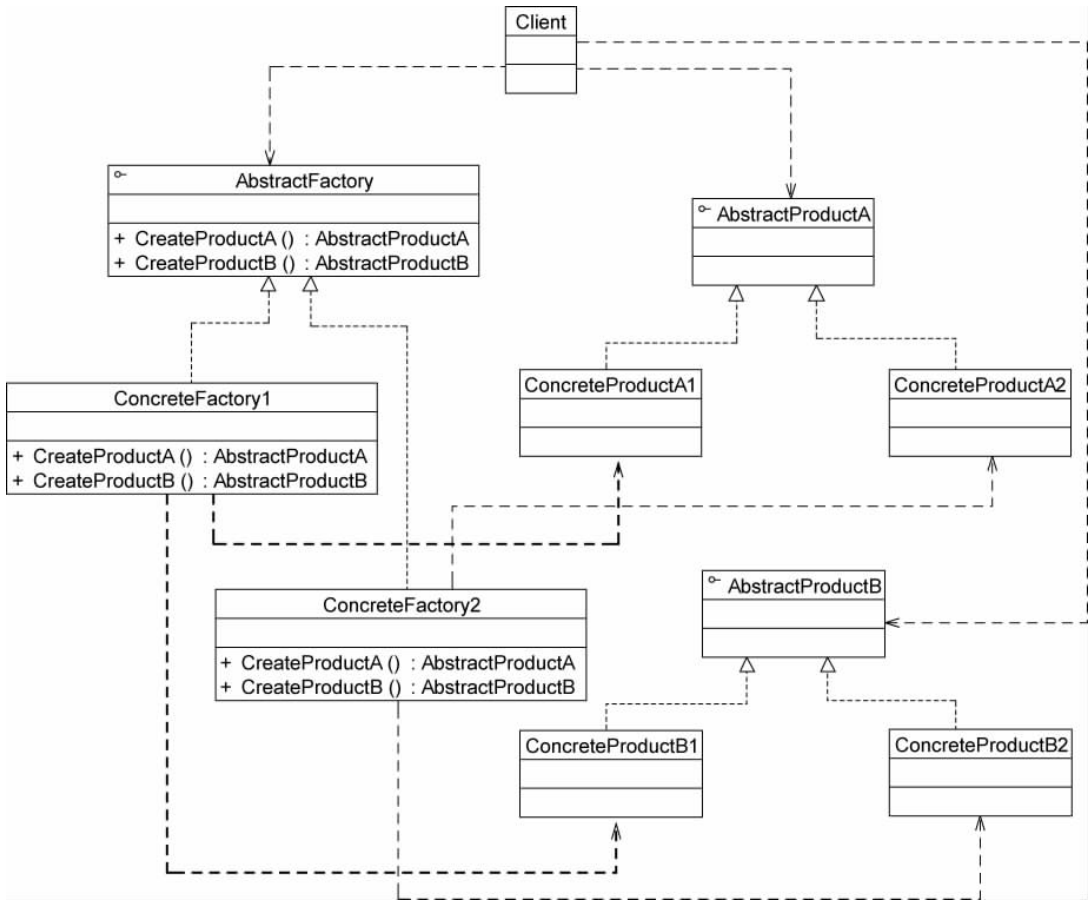

图 5-3　抽象工厂模式结构图

5.3.2　抽象工厂模式的实现

在抽象工厂中声明了多个工厂方法，用于创建不同类型的产品，抽象工厂可以是接口，也可以是抽象类或者具体类。其典型代码如下：

```
abstract class AbstractFactory
{
    public abstract AbstractProductA CreateProductA();    //工厂方法一
    public abstract AbstractProductB CreateProductB();    //工厂方法二
    …
}
```

具体工厂实现了抽象工厂，每一个具体的工厂方法创建一个特定的产品对象，而同一个具体工厂所创建的产品对象构成了一个产品族。对于每一个具体工厂类，其典型代码如下：

```
class ConcreteFactory1 : AbstractFactory
{
```

```
        //工厂方法一
        public override AbstractProductA CreateProductA()
        {
            return new ConcreteProductA1();
        }

        //工厂方法二
        public override AbstractProductB CreateProductB()
        {
            return new ConcreteProductB1();
        }

        …
    }
```

与工厂方法模式一样，抽象工厂模式也可为每一种产品提供一组重载的工厂方法，以不同的方式来创建产品对象。

5.4 抽象工厂模式的应用实例

下面通过一个应用实例来进一步学习和理解抽象工厂模式。

1. 实例说明

某软件公司要开发一套界面皮肤库，可以对基于.NET平台的桌面软件进行界面美化。用户在使用时可以通过菜单来选择皮肤，不同的皮肤将提供视觉效果不同的按钮、文本框、组合框等界面元素，例如春天(Spring)风格的皮肤将提供浅绿色的按钮、绿色边框的文本框和绿色边框的组合框，而夏天(Summer)风格的皮肤则提供浅蓝色的按钮、蓝色边框的文本框和蓝色边框的组合框，其结构示意图如图5-4所示。

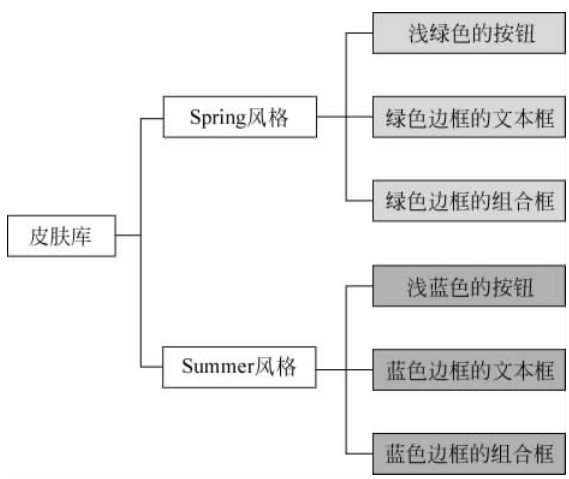

图5-4 界面皮肤库结构示意图

该皮肤库需要具备良好的灵活性和可扩展性，用户可以自由选择不同的皮肤，开发人员可以在不修改既有代码的基础上增加新的皮肤。

现使用抽象工厂模式来设计该界面皮肤库。

2. 实例类图

通过分析，本实例的结构如图 5-5 所示。

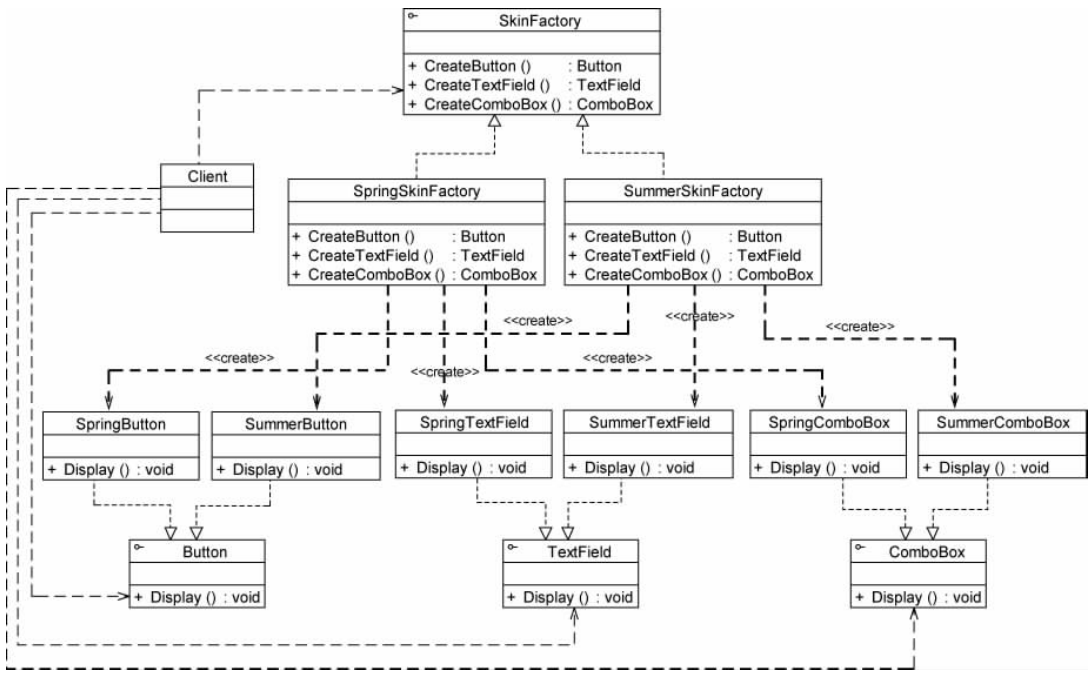

图 5-5　界面皮肤库结构图

在图 5-5 中，SkinFactory 接口充当抽象工厂，其子类 SpringSkinFactory 和 SummerSkinFactory 充当具体工厂，接口 Button、TextField 和 ComboBox 充当抽象产品，其子类 SpringButton、SpringTextField、SpringComboBox 和 SummerButton、SummerTextField、SummerComboBox 充当具体产品。

3. 实例代码

(1) Button：按钮接口，充当抽象产品。

```
//Button.cs
namespace AbstractFactorySample
{
    interface Button
    {
        void Display();
    }
}
```

(2) SpringButton：Spring 按钮类，充当具体产品。

```
//SpringButton.cs
using System;

namespace AbstractFactorySample
{
    class SpringButton : Button
    {
        public void Display()
        {
            Console.WriteLine("显示浅绿色的按钮。");
        }
    }
}
```

(3) SummerButton：Summer 按钮类，充当具体产品。

```
//SummerButton.cs
using System;

namespace AbstractFactorySample
{
    class SummerButton : Button
    {
        public void Display()
        {
           Console.WriteLine("显示浅蓝色的按钮。");
        }
    }
}
```

(4) TextField：文本框接口，充当抽象产品。

```
//TextField.cs
namespace AbstractFactorySample
{
    interface TextField
    {
        void Display();
    }
}
```

(5) SpringTextField：Spring 文本框类，充当具体产品。

```
//SpringTextField.cs
using System;

namespace AbstractFactorySample
{
```

```
    class SpringTextField : TextField
    {
        public void Display()
        {
            Console.WriteLine("显示绿色边框的文本框。");
        }
    }
}
```

(6) SummerTextField：Summer 文本框类，充当具体产品。

```
//SummerTextField.cs
using System;

namespace AbstractFactorySample
{
    class SummerTextField : TextField
    {
        public void Display()
        {
            Console.WriteLine("显示蓝色边框的文本框。");
        }
    }
}
```

(7) ComboBox：组合框接口，充当抽象产品。

```
//ComboBox.cs
namespace AbstractFactorySample
{
    interface ComboBox
    {
        void Display();
    }
}
```

(8) SpringComboBox：Spring 组合框类，充当具体产品。

```
//SpringComboBox.cs
using System;

namespace AbstractFactorySample
{
    class SpringComboBox : ComboBox
    {
        public void Display()
        {
            Console.WriteLine("显示绿色边框的组合框。");
        }
    }
}
```

(9) SummerComboBox：Summer 组合框类，充当具体产品。

```
//SummerComboBox.cs
using System;

namespace AbstractFactorySample
{
    class SummerComboBox : ComboBox
    {
        public void Display()
        {
            Console.WriteLine("显示蓝色边框的组合框。");
        }
    }
}
```

(10) SkinFactory：界面皮肤工厂接口，充当抽象工厂。

```
//SkinFactory.cs
namespace AbstractFactorySample
{
    interface SkinFactory
    {
        Button CreateButton();
        TextField CreateTextField();
        ComboBox CreateComboBox();
    }
}
```

(11) SpringSkinFactory：Spring 皮肤工厂，充当具体工厂。

```
//SpringSkinFactory.cs
namespace AbstractFactorySample
{
    class SpringSkinFactory : SkinFactory
    {
        public Button CreateButton()
        {
            return new SpringButton();
        }

        public TextField CreateTextField()
        {
            return new SpringTextField();
        }

        public ComboBox CreateComboBox()
        {
            return new SpringComboBox();
        }
    }
}
```

(12) SummerSkinFactory：Summer 皮肤工厂，充当具体工厂。

```
//SummerSkinFactory.cs
namespace AbstractFactorySample
{
    class SummerSkinFactory : SkinFactory
    {
        public Button CreateButton()
        {
            return new SummerButton();
        }

        public TextField CreateTextField()
        {
            return new SummerTextField();
        }

        public ComboBox CreateComboBox()
        {
            return new SummerComboBox();
        }
    }
}
```

(13) 配置文件 App.config：在配置文件中存储了具体工厂类类名。

```
<?xml version = "1.0" encoding = "utf - 8" ?>
< configuration >
  < appSettings >
    < add key = "factory" value = "AbstractFactorySample.SpringSkinFactory"/>
  </appSettings >
</configuration >
```

(14) Program：客户端测试类。

```
//Program.cs
using System;
using System.Configuration;
using System.Reflection;

namespace AbstractFactorySample
{
    class Program
    {
        static void Main(string[] args)
        {
            //使用抽象层定义
            SkinFactory factory;
            Button bt;
            TextField tf;
            ComboBox cb;
```

```
            //读取配置文件
            string factoryType = ConfigurationManager.AppSettings["factory"];

            //反射生成对象
            factory = (SkinFactory)Assembly.Load("AbstractFactorySample").CreateInstance
(factoryType);
            bt = factory.CreateButton();
            tf = factory.CreateTextField();
            cb = factory.CreateComboBox();
            bt.Display();
            tf.Display();
            cb.Display();

            Console.Read();
        }
    }
}
```

4. 结果及分析

编译并运行程序,输出结果如下:

```
显示浅绿色的按钮。
显示绿色边框的文本框。
显示绿色边框的组合框。
```

如果需要更换皮肤,只需修改配置文件即可,例如将春天风格的皮肤改为夏天风格的皮肤,只需将存储在配置文件中的具体工厂类 SpringSkinFactory 改为 SummerSkinFactory 即可,代码如下:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="factory" value="AbstractFactorySample.SummerSkinFactory"/>
  </appSettings>
</configuration>
```

重新运行客户端程序,输出结果如下:

```
显示浅蓝色的按钮。
显示蓝色边框的文本框。
显示蓝色边框的组合框。
```

在实际环境中,可以提供一个可视化界面,例如菜单或者窗口来修改配置文件,用户无须直接修改配置文件。如果需要增加新的皮肤,只需增加一族新的具体组件并对应提供一个新的具体工厂,修改配置文件中的具体工厂类的类名即可使用新的皮肤,原有代码无须修改,符合开闭原则。

5.5 开闭原则的倾斜性

在5.4节设计的界面皮肤库中可以较为方便地增加新类型的皮肤，但是该设计方案存在一个非常严重的问题：如果在设计之初因为考虑不全面，忘记为某种类型的界面组件（以单选按钮RadioButton为例）提供不同皮肤下的风格化显示，那么在向系统中增加单选按钮时将非常麻烦，无法在满足开闭原则的前提下增加单选按钮，原因是抽象工厂SkinFactory中根本没有提供创建单选按钮的方法，如果需要增加单选按钮，首先要修改抽象工厂接口SkinFactory，在其中增加声明创建单选按钮的方法，然后逐个修改具体工厂类，增加相应方法，以便在不同的皮肤库中创建单选按钮，此外还需要修改客户端，否则单选按钮无法应用于现有系统。

抽象工厂模式无法很好地解决此类问题，这也是抽象工厂模式的最大缺点所在。在抽象工厂模式中，增加新的产品族很方便，但是增加新的产品等级结构很麻烦，抽象工厂模式的这种特性称为开闭原则的倾斜性。开闭原则要求系统对扩展开放，对修改关闭，通过扩展达到增强其功能的目的，对于涉及多个产品族与多个产品等级结构的系统，其功能增强包括以下两个方面。

（1）增加产品族：对于增加新的产品族，抽象工厂模式很好地支持了开闭原则，只需增加具体产品并对应增加一个新的具体工厂即可，对已有代码无须做任何修改。

（2）增加新的产品等级结构：对于增加新的产品等级结构，需要修改所有的工厂角色，包括抽象工厂类，在所有的工厂类中都需要增加生产新产品的方法，违背了开闭原则。

正因为抽象工厂模式存在开闭原则的倾斜性，它以一种倾斜的方式来满足开闭原则，为增加新产品族提供方便，但不能为增加新产品结构提供这样的方便，因此要求设计人员在设计之初就要全面考虑，不要在设计完成之后再向系统中增加新的产品等级结构，也不要删除已有的产品等级结构，否则将会导致系统出现较大的修改，为后续的维护工作带来诸多麻烦。

5.6 抽象工厂模式的优缺点与适用环境

抽象工厂模式是工厂方法模式的进一步延伸，由于它提供了功能更为强大的工厂类并且具备较好的可扩展性，在软件开发中得以广泛应用，尤其是在一些框架和API类库的设计中。抽象工厂模式是软件开发中最常用的设计模式之一。

5.6.1 抽象工厂模式的优点

抽象工厂模式的主要优点如下：

（1）抽象工厂模式隔离了具体类的生成，使得客户端并不需要知道什么被创建。由于这种隔离，更换一个具体工厂就变得相对容易，所有的具体工厂都实现了抽象工厂中定义的公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。

（2）当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象。

（3）抽象工厂模式增加新的产品族很方便，无须修改已有系统，符合开闭原则。

5.6.2 抽象工厂模式的缺点

抽象工厂模式的主要缺点如下：

增加新的产品等级结构麻烦，需要对原有系统进行较大的修改，甚至需要修改抽象层代码，这显然会带来较大的不便，违背了开闭原则。

5.6.3 抽象工厂模式的适用环境

在以下情况下可以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，这对于所有类型的工厂模式都是很重要的，用户无须关心对象的创建过程，将对象的创建和使用解耦。

(2) 系统中有多于一个的产品族，但每次只使用其中某一产品族，可以通过配置文件等方式使用户能够动态地改变产品族，也可以很方便地增加新的产品族。

(3) 属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来。同一个产品族中的产品可以是没有任何关系的对象，但是它们都具有一些共同的约束，如同一操作系统下的按钮和文本框，按钮与文本框之间没有直接关系，但它们都是属于某一操作系统的，此时具有一个共同的约束条件：操作系统的类型。

(4) 产品等级结构稳定，设计完成之后，不会向系统中增加新的产品等级结构或者删除已有的产品等级结构。

5.7 本章小结

(1) 在抽象工厂模式中，产品等级结构即产品的继承结构，产品族是指由同一个工厂生产的，位于不同产品等级结构中的一组产品。

(2) 抽象工厂模式提供了一个创建一系列相关或相互依赖对象的接口，开发人员无须指定具体的类。抽象工厂模式是一种对象创建型模式。

(3) 抽象工厂模式包含抽象工厂、具体工厂、抽象产品和具体产品 4 个角色。其中，抽象工厂声明了一组用于创建一族产品的方法，每一个方法对应一种产品；具体工厂实现了在抽象工厂中声明的创建产品的方法，生成一组具体产品，这些产品构成了一个产品族，每一个产品都位于某个产品等级结构中；抽象产品为每种产品声明接口，在抽象产品中声明了产品所具有的业务方法；具体产品定义具体工厂生产的具体产品对象，实现抽象产品接口中声明的业务方法。

(4) 抽象工厂模式的主要优点是隔离了具体类的生成，使得客户端不需要知道什么被创建；当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象；增加新的产品族很方便，无须修改已有系统，符合开闭原则。其主要缺点是增加新的产品等级结构麻烦，需要对原有系统进行较大的修改，甚至需要修改抽象层代码，违背了开闭原则。

(5) 抽象工厂模式适用的环境：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；系统中有多于一个的产品族，但每次只使用其中某一产品族；属于同一个

产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来；产品等级结构稳定，在设计完成后，不会向系统中增加新的产品等级结构或者删除已有的产品等级结构。

（6）抽象工厂模式以一种倾斜的方式来满足开闭原则。对于增加新的产品族，抽象工厂模式很好地支持了开闭原则；对于增加新的产品等级结构，需要修改所有的工厂角色，违背了开闭原则。

5.8 习题

1. 某公司要开发一个图表显示系统，在该系统中，曲线图生成器可以创建曲线图、曲线图图例和曲线图数据标签，柱状图生成器可以创建柱状图、柱状图图例和柱状图数据标签。用户要求可以很方便地增加新类型的图形，系统需具备较好的可扩展能力。针对这种需求，公司采用（　　）最为恰当。

A. 桥接模式　　B. 适配器模式　　C. 策略模式　　D. 抽象工厂模式

2. 以下关于抽象工厂模式的叙述错误的是（　　）。

A. 抽象工厂模式提供了一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类

B. 当系统中有多于一个产品族时可以考虑使用抽象工厂模式

C. 当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时，抽象工厂模式比工厂方法模式更为简单、更有效率

D. 抽象工厂模式符合开闭原则，增加新的产品族和新的产品等级结构都很方便

3. 下列关于抽象工厂模式中的产品族和产品等级结构的叙述，错误的是（　　）。

A. 产品等级结构是从不同的产品族中任意选取产品组成的层次结构

B. 产品族是指由位于不同产品等级结构、功能相关的产品组成的家族

C. 抽象工厂是指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

D. 工厂方法模式对应唯一一个产品等级结构，而抽象工厂模式则需要对应多个产品等级结构

4. 计算机包含内存（RAM）、CPU 等硬件设备，根据图 5-6“产品等级结构-产品族”所示，使用抽象工厂模式实现计算机硬件设备的创建过程，绘制相应的类图并使用 C＃语言编程模拟实现。

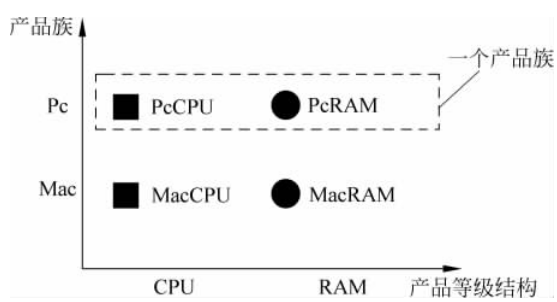

图 5-6 “产品等级结构-产品族”示意图

5. 一个电器工厂可以生产多种类型的电器，如海尔工厂可以生产海尔电视机、海尔空调等，TCL 工厂可以生产 TCL 电视机、TCL 空调等，相同品牌的电器构成了一个产品族，而相同类型的电器构成了一个产品等级结构，试使用抽象工厂模式模拟该场景。

6. 某基于.NET 平台的系统需改进数据库操作的性能，使用户可以自定义数据库连接对象 Connection 和数据命令对象 Command，针对不同类型的数据库提供不同的连接对象和命令对象，例如提供 Oracle 或 MySQL 专用连接类和命令类，而且用户可以通过配置文件等方式根据实际需要动态更换系统数据库以及增加新的数据连接类和数据命令类。使用抽象工厂模式设计该系统，要求绘制对应的类图并使用 C# 语言编程模拟实现。

7. 抽象工厂模式最早的应用之一是用来创建在不同操作系统的图形环境下都能够运行的应用程序，例如同时支持 Windows 与 Linux 操作系统。在每一个操作系统中，都有一个由图形构件组成的构件家族，可以通过一个抽象角色给出功能定义，而由具体子类给出不同操作系统下的具体实现，例如系统中包含两个产品等级结构，分别是 Button 与 Text，同时包含 3 个产品族，即 UNIX 产品族、Linux 产品族与 Windows 产品族，如图 5-7 所示。

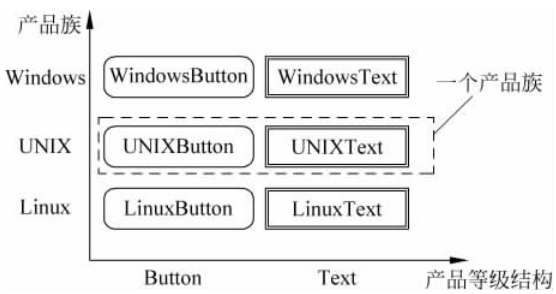

图 5-7　不同操作系统下不同构件的“产品等级结构-产品族”示意图

在图 5-7 中，Windows 中的 Button 和 Text 构成了一个 Windows 产品族，而不同操作系统下的 Button 构成了一个产品等级结构。试使用抽象工厂模式来设计并模拟实现该结构。

8. 某软件公司要推出一款新的手机游戏软件，该软件支持 iOS、Android 和 Windows Phone 等多个智能手机操作系统平台，针对不同的手机操作系统，该游戏软件提供了不同的游戏操作控制(OperationController)类和游戏界面控制(InterfaceController)类，并提供了相应的工厂类来封装这些类的初始化过程。该软件要求具有较好的扩展性，以支持新的操作系统平台，为了满足上述需求，试采用抽象工厂模式对其进行设计。