抽象工厂模式

1.抽象工厂模式介绍

抽象工厂模式（Abstract Factory Pattern），也是创建型设计模式之一。大家联想一下现实生活中的工厂肯定都是具体的，那么抽象工厂意味着生产出来的产品是不是确定的，那这岂不是很奇怪?抽象工厂模式起源于以前对不同操作系统的图形化解决方案，如不同的操作系统中的按钮和文本框控件其实现不同，展示效果也不一样，对于每一个操作系统，其本身就构成一个产品类，而按钮与文本框控件也构成一个产品类，两种产品类两种变化，各自有自己的特性，如Android中的Button和TextView 、ios中的Button和TextView等等。

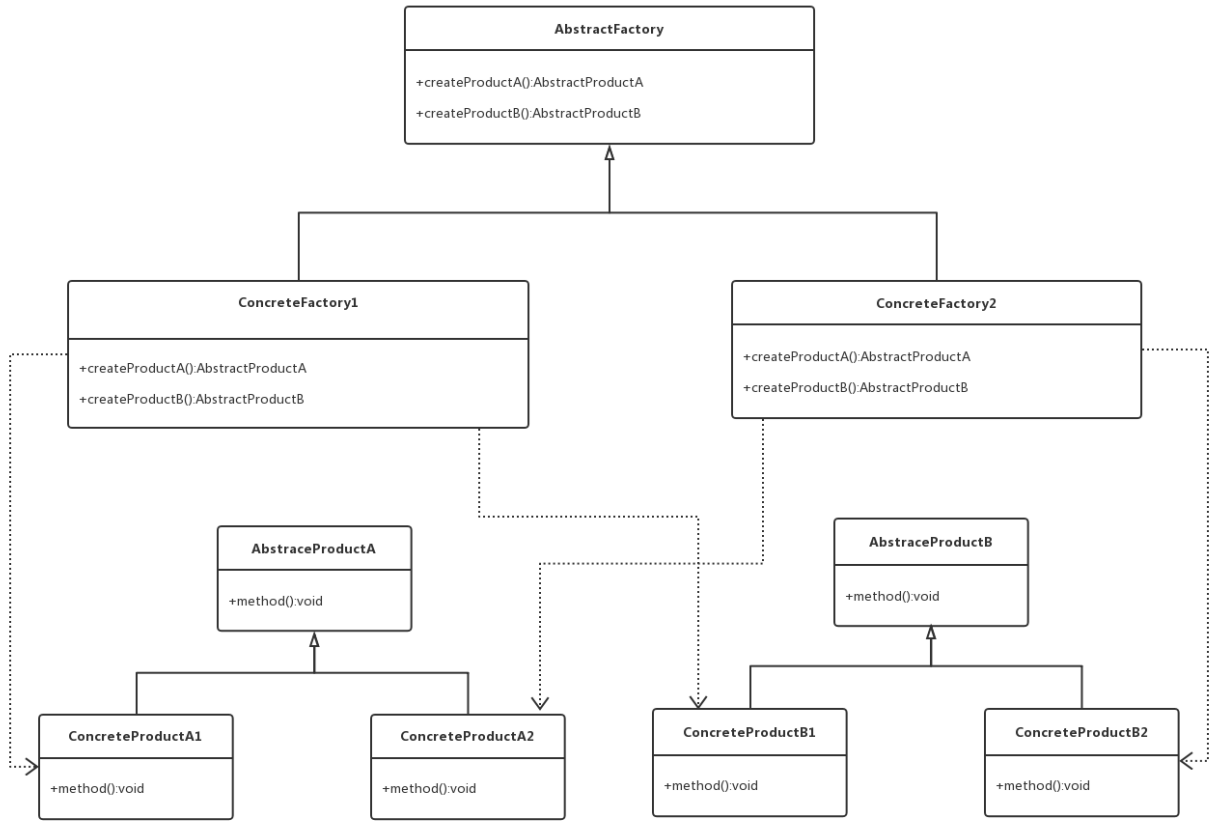
2.抽象工厂模式的定义

为创建一组相关或者是相互依赖的对象提供一个接口，而不需要制定它们的具体类。

3,抽象工厂模式的使用场景

一个对象族有相同的约束时可以使用抽象工厂模式。是不是听起来很抽象?举个例子，Android、ios、Window Phone下都有短信软件和拨号软件，两者都属于Software软件的范畴，但是，它们梭子啊的操作系统平台不一样，即便是同一家公司出品的软件，其代码的实现逻辑也是不同的，这时候就可以考虑使用抽象工厂方法模式来产生Android、ios、Window Phone下的短信软件和拨号软件。

4.抽象工厂模式的UML类图



根据类图可以得出如下一个抽象工厂模式的通用模式代码：
抽象产品类A:

```
public abstract class AhstractProductA {
        /*
         * 每个具体的产品子类需要实现的方法
         */
        public abstract void method();
 }
```

抽象产品类B：

```
public abstract class AhstractProductB {
        /*
         * 每个具体的产品子类需要实现的方法
         */
        public abstract void method();
 }

```

具体产品：

```
//具体产品类A1
public   class ConcreteProductA1 extends AhstractProductA{
```

```
3
4           @Override
5           public void method() {
6                   System.out.println("具体产品A1 的方法");
7           }
8
9    }
10
11  //具体产品类A2
12  public    class ConcreteProductA2 extends AhstractProductA{
13
14          @Override
15          public void method() {
16                  System.out.println("具体产品A2 的方法");
17          }
18
19   }
20
21
22  //具体产品类B1
23  public    class ConcreteProductB1 extends AhstractProductB{
24
25          @Override
26          public void method() {
27                  System.out.println("具体产品B1 的方法");
28          }
29
30
31   }
32
33  //具体产品类B2
34  public class ConcreteProductB2 extends AhstractProductB {
35
36          @Override
37          public void method() {
38                  System.out.println("具体产品B2 的方法");
39          }
40
41  }
```

抽象工厂类：

```
1   public abstract class AbstractFactory {
2           /*
3            * 创建产品A 的方法
4            *
5            * @return 产品A对象
6            */
7           public abstract AhstractProductA createProductA();
8
9           /*
10           * 创建产品B 的方法
11           *
12           * @return 产品B对象
13           */
```

```
        public abstract AhstractProductB createProductB();
}
```

具体工厂类：

```
//具体工厂类1
public class ConcreteFactory1 extends AbstractFactory {

        @Override
        public AhstractProductA createProductA() {
                return new ConcreteProductA1();
        }

        @Override
        public AhstractProductB createProductB() {
                return new ConcreteProductB1();
        }

}

//具体工厂类2
public class ConcreteFactory2 extends AbstractFactory {

        @Override
        public AhstractProductA createProductA() {
                return new ConcreteProductA2();
        }

        @Override
        public AhstractProductB createProductB() {
                return new ConcreteProductB2();
        }

}
```

虽然抽象工厂方法模式的类繁多，但是，主要还是分为4类：

- AbstractFactory： 抽象工厂角色，它声明了一组用于创建一种产品的方法，每一个方法对应一种产品，如上述类图中的AbstrctFactory中就定义了两个方法，分别创建产品A和B.
- ConcreteFactory：具体工厂角色，它实现了在抽象工厂中定义的创建产品的方法，生成一组具体产品，这些产品构成了一个产品类，每一个产品都位于某个产品等级结构中，如上述类图中ConcreteFactory1和ConcreteFactory2。
- AbstractProduct：抽象产品角色，它为每种产品声明接口，比如上述类图中的AbstractProductA和AbstractProductB.
- ConcreteProduct：具体产品角色，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，如上述图中ConcreteProductA1、ConcreteProductA2、ConcreteProductB1、ConcreteProductB2。

5.抽象工厂方法模式的简单实现

在工厂方法模式中，以小明的车厂为例阐述了工厂方法模式，但是，后来小明发现一个问题，虽然Q3、Q5、Q7都是一个车系，但是三者之间的零部件差别却是很多，就拿Q3和Q7说，Q3使用的发动机是国产的，而Q7则原装进口的；Q3轮胎时普通轮胎，而Q7是全尺寸越野轮胎；还有Q3使用的是比较普通的制动系统，而Q7则使用的制动性能更好的制动系统。Q3、Q7对应的是一系列车，而发动机、轮胎、制动系统则对应得一系列零部件，两者是两种不同的产品类型，这时候就可以将抽象工厂模式应用到其中，首先，汽车工厂需要生产轮胎、发动机、制动系统这3种部件。
抽象车厂类：

```
public abstract class CarFactory {
        /*
         * 生产轮胎
         *
         * @return 轮胎
         */
        public abstract ITire createTire();

        /*
         * 生产发动机
         *
         * @return 发动机
         */
        public abstract IEngine createEngine();

        /*
         * 生产制动系统
         *
         * @return 制动系统
         */
        public abstract IBrake createBrake();
}
```

轮胎相关类：

```
public interface ITire {
        /*
         * 轮胎
         */
        void tire();
}

public class NormalTire implements ITire {

        public void tire() {
                System.out.println("普通轮胎");
        }

}

public class SUVTire implements ITire {

        public void tire() {
                System.out.println("越野轮胎");
        }

}
```

发动机相关类：

```
public interface IEngine {
        /*
         * 发动机
         */
        void engine();
}

public class DomesticEngine implements IEngine{

        public void engine() {
                System.out.println("国产发动机");
        }

}

public class ImportEngine implements IEngine{

        public void engine() {
                System.out.println("进口发动机");
        }

}
```

制动系统类：

```
public interface IBrake {
        /*
         * 制动系统
         */
        void brake();
}

public class NormalBrake implements IBrake{

        public void brake() {
                System.out.println("普通制动系统");
        }

}

public class SeniorBrake implements IBrake{

        public void brake() {
                System.out.println("高级制动系统");
        }

}
```

对于生产Q3的工厂，其使用的零部件不同，而生产Q7的工厂，其使用的零部件也不同。
Q3工厂类：

```
 public class Q3Factory extends CarFactory{

        @Override
        public ITire createTire() {
                return new NormalTire();
        }

        @Override
        public IEngine createEngine() {
                return new DomesticEngine();
        }

        @Override
        public IBrake createBrake() {
                return new NormalBrake();
        }

}
```

Q7工厂类：

```
public class Q7Factory extends CarFactory{

        @Override
        public ITire createTire() {
                return new SUVTire();
        }

        @Override
        public IEngine createEngine() {
                return new ImportEngine();
        }

        @Override
        public IBrake createBrake() {
                return new SeniorBrake();
        }

}
```

最后，我们在Client客户端中模拟：

```
public class Client {
        public static void main(String[] args) {
                // 构造一个生产Q3的工厂
                CarFactory factoryQ3 = new Q3Factory();
                factoryQ3.createTire().tire();
                factoryQ3.createEngine().engine();
                factoryQ3.createBrake().brake();

                System.out.println("--------------------");
                // 构造一个生产Q7的工厂

```

```
                CarFactory factoryQ7 = new Q7Factory();
                factoryQ7.createTire().tire();
                factoryQ7.createEngine().engine();
                factoryQ7.createBrake().brake();

        }
    }
```

输出：

```
 普通轮胎
国产发动机
普通制动系统
--------------------
越野轮胎
进口发动机
高级制动系统

```

上面我们只是模拟了两个车系Q3和Q7的工厂，如何此时我们需求增加Q5的工厂呢？那么对应的轮胎、制动系统和发动机类又要增加，这里就可以看到抽象工厂方法模式的一个弊端，就是类的突增，如果工厂类过多，势必导致类文件非常多，因此，在实际开发中一定要权衡慎用。

6.总结

优点：
一个显著的优点是分离接口与实现，客户端使用抽象工厂来创建需要的对象，而客户端根本就就不知道具体的实现是谁，客户端只是面向产品的接口编程而已，使其从具体的产品实现中解耦，同时基于接口与实现的分离，使抽象该工厂模式在切换产品类时更加灵活、容易。
缺点：
一是类文件的爆炸性增加，而是不太容易扩展新的产品类，因为每当我们增加一个产品类就需要修改抽象工厂，那么所有的具体工厂类均会被修改。

xyzso1z
原创文章 80　获赞 40　访问量 2万+