软件体系结构

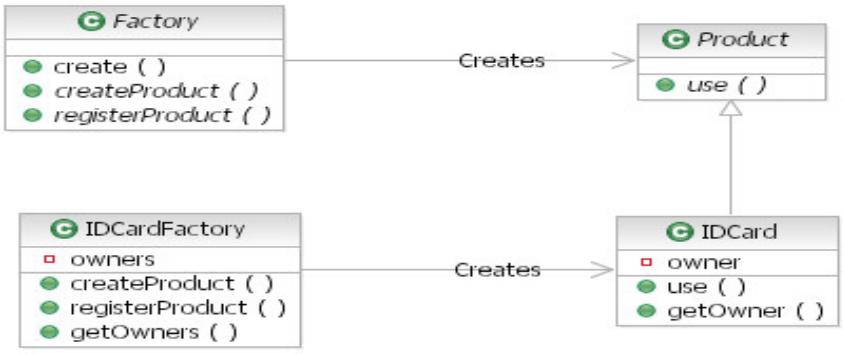
《软件体系结构作业十》

学　号 22920212204396

姓　名 黄子安

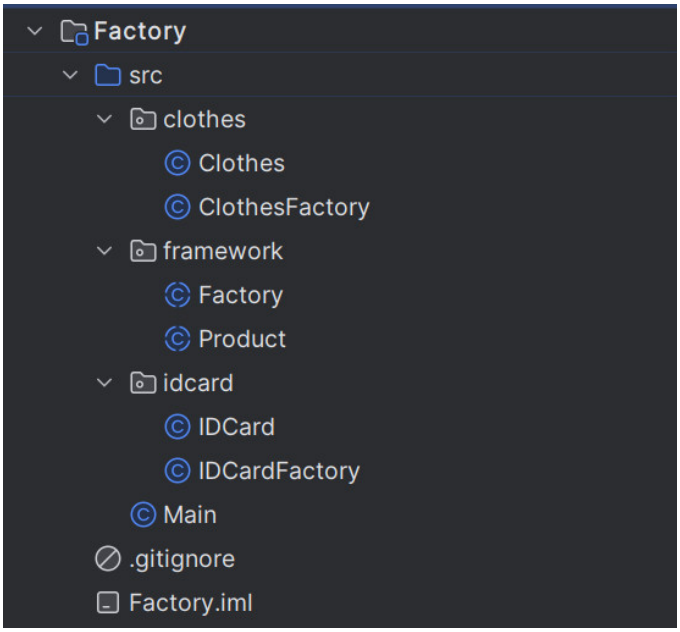
2024 年 4 月 27 日

1、改写本例，用于添加另一个具体工厂和具体产品。



根据工厂方法模式，Factory 是产生 Product 的抽象类，具体内容由 ConcreteFactory 决定，Factory 对于实际产生的 ConcreteProduct 一无所知，唯一知道的就是调用 Product 和产生新对象的方法。Product 规定了此 Pattern 所产生的对象实例应有的接口，具体内容则由子类 ConcreteProduct 参与者决定，该类描述的是框架。

因此要新增具体的工厂和产品的时候只需要新增具体工厂类继承 Factory，具体的产品继承 Product 即可，符合 **Liskov 原则和开闭原则**



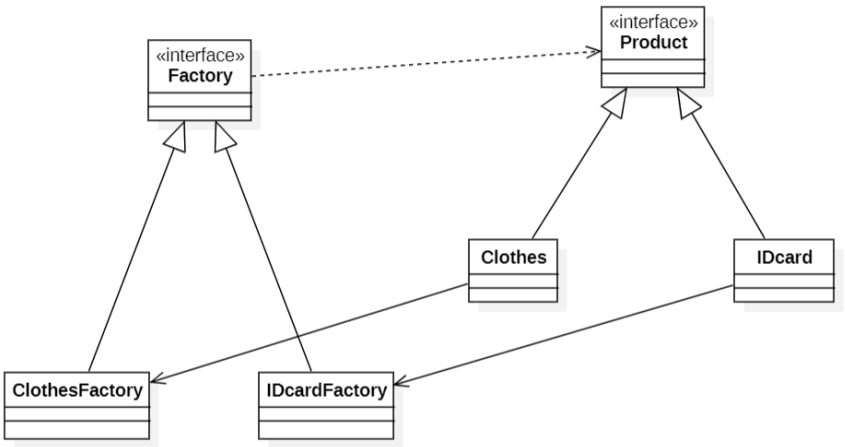
修改后的包如上所示，直接新增一个 clothes 包用于放 clothes 相关的具体产品和工厂

具体 clothes 产品和工厂代码如下所示

```
package clothes;

import framework.Product;

public class Clothes extends Product {
    private String owner;
    Clothes(String owner) {
        System.out.println("给" + owner + "衣服。");
        this.owner = owner;
    }
    public void use () {
        System.out.println(owner + "穿衣服。");
    }
    public String getOwner() {
        return owner;
    }
}
```

```
package clothes;

import framework.Factory;
import framework.Product;

import java.util.Vector;

public class ClothesFactory extends Factory {
    private Vector owners = new Vector();
    protected Product createProduct(String owner) {
        return new Clothes(owner);
    }
    protected void registerProduct(Product product) {
        owners.add(((Clothes)product).getOwner());
    }
    public Vector getOwners() {
        return owners;
    }
}
```

此时对应的类图如下所示



最后修改下测试方法，在其中新增对 clothes 产品和工厂的测试

```
import clothes.ClothesFactory;
import framework.*;
import idcard.*;

public class Main {
    public static void main(String[] args) {
        Factory factory = new IDCardFactory();
        Product card1 = factory.create("乔峰");
        Product card2 = factory.create("令狐冲");
        Product card3 = factory.create("武松");
        card1.use();
        card2.use();
        card3.use();

        System.out.println();

        Factory factory2 = new ClothesFactory();
        Product clothes1= factory2.create("乔峰");
        Product clothes2 = factory2.create("令狐冲");
        Product clothes3 = factory2.create("武松");
        clothes1.use();
        clothes2.use();
        clothes3.use();
    }
}
```

运行结果如下图所示：

```
D:\Java17\bin\java.exe "-javaagent:D:\IdeaU\IntelliJ IDEA 2023.1.3\lib\idea_rt.jar=
建立乔峰的卡。
建立令狐冲的卡。
建立武松的卡。
使用乔峰的卡。
使用令狐冲的卡。
使用武松的卡。

给乔峰衣服。
给令狐冲衣服。
给武松衣服。
乔峰穿衣服。
令狐冲穿衣服。
武松穿衣服。

Process finished with exit code 0
```

2、请举例说明其他的工厂模式的应用。

工厂模式事实上有三种，分别为简单工厂模式、工厂方法模式和抽象工厂模式

1. 简单工厂模式

简单工厂模式通过一个中心化的工厂类来决定创建哪种类型的对象，这种模式通常用于创建类似的对象，只是配置不同。

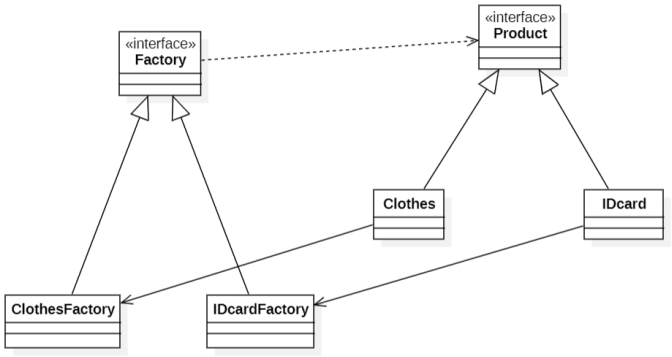
先将产品类抽象出来，比如苹果和梨都属于水果，抽象出来一个水果类Fruit，苹果和梨就是具体的产品类，然后在需要的时候创建（new）一个水果工厂，分别用来创建苹果和梨，代码如下：

```
public interface Fruit {

}

public class Apple implements Fruit {

}

public class Pear implements Fruit {

}

public class FruitFactory {
    public Fruit createFruit(String type) {
        if (type.equals("apple")) {
            return new Apple();
        } else if (type.equals("pear")) {
            return new Pear();
        }
        return null;
    }
}

//使用工厂创建产品
FruitFactory mFactory = new FruitFactory();
Apple apple = (Apple) mFactory.createFruit("apple");
Pear pear = (Pear) mFactory.createFruit("pear");
```

简单工厂模式有一定问题，比如想添加一种水果，就必然要修改工厂类，这显然违反了开闭原则；所以简单工厂只适合于产品对象较少，且产品固定的需求，对于产品变化无常的需求来说不合适

2. 工厂方法模式

第一问的例子就是工厂方法模式，通过把工厂类也抽象出来，生产什么样的产品由子类来决定，实现了**解耦合与开闭原则**，需要新增的时候只要新增具体工厂类和具体产品类即可，不会修改原有的代码



但是依旧会有问题，如果需要的产品很多的话，则需要创建非常多的工厂，所以这种方式的缺点也很明显

3. 抽象工厂模式

为创建一组相关或者是相互依赖的对象提供的一个接口，而不需要指定它们的具体类，抽象工厂和工厂方法的模式基本一样，区别在于工厂方法是生产一个具体的产品，而抽象工厂可以用来生产一组相同，有相对关系的产品，重点在于一组，一批，一系列；

比如生产小米手机，小米手机有很多系列，小米 note、红米 note 等，小米 note 生产需要的配件有 825 的处理器，6 英寸屏幕，而红米只需要 650 的处理器和 5 寸的屏幕

使用抽象工厂模式的代码实现如下：

```
public interface Cpu {
    void run();

    class Cpu650 implements Cpu {
        @Override
        public void run() {

        }
    }

    class Cpu825 implements Cpu {
        @Override
        public void run() {

        }
    }
}
```

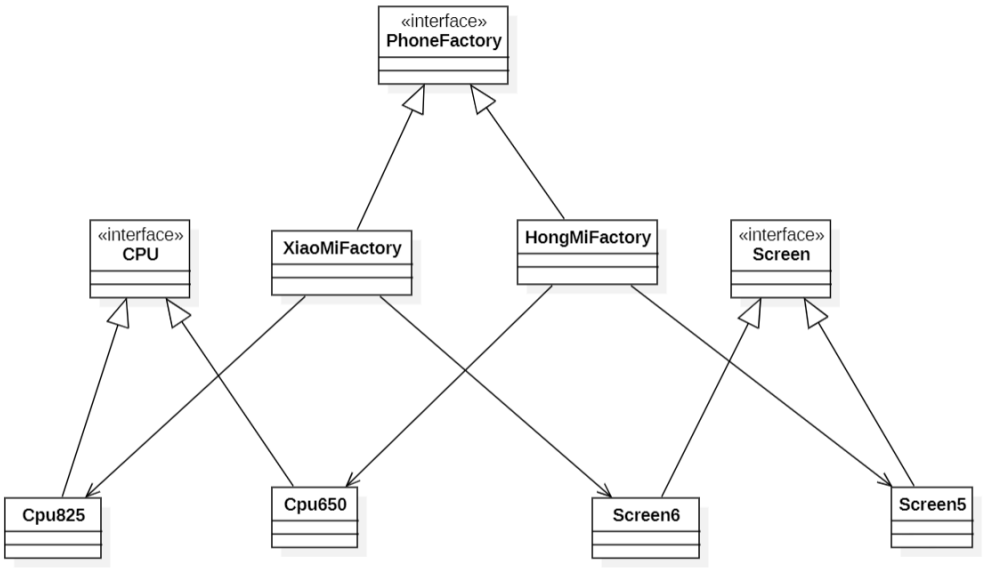
```
public interface Screen {

    void size();

    class Screen5 implements Screen {

        @Override
        public void size() {
            //5寸
        }
    }

    class Screen6 implements Screen {

        @Override
        public void size() {
            //6寸
        }
    }
}
```

```
public interface PhoneFactory {

    Cpu getCpu();//使用的cpu

    Screen getScreen();//使用的屏幕
}
```

```java
public class XiaoMiFactory implements PhoneFactory {
    @Override
    public Cpu getCpu() {
        return new Cpu.Cpu825();//高性能处理器
    }

    @Override
    public Screen getScreen() {
        return new Screen.Screen6();//6寸大屏
    }
}
```

```java
public class HongMiFactory implements PhoneFactory {

    @Override
    public Cpu getCpu() {
        return new Cpu.Cpu650();//高效处理器
    }

    @Override
    public Screen getScreen() {
        return new Screen.Screen5();//小屏手机
    }
}
```

抽象工厂可以解决一系列的产品生产的需求，对于大批量、多系列的产品，用抽象工厂可以更好的管理和扩展，对应的类图如下：



本质上来说就是对工厂方法的每一个产品寻找了共同特征进行打包处理，避免工厂的数量过多