软件体系结构作业 10

姓　名：洪伟鑫

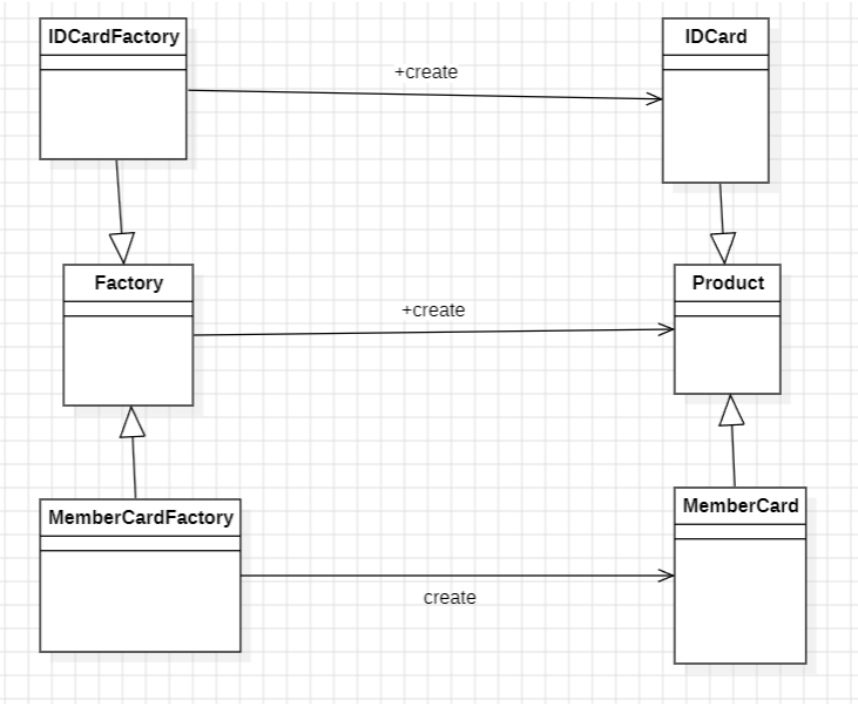
专　业：软件工程

年　级：2022 级

学　号：37220222203612

2025 年 4 月 20 日

1、改写本例，用于添加另一个具体工厂和具体产品。



具体的工厂类，例如 IDCardFactory 和 MemberCardFactory，继承或实现了抽象工厂。它们负责实现具体的对象创建逻辑，从而决定实例化哪种具体产品（如 IDCard 或 MemberCard）。这些具体产品类则继承或实现了抽象产品类。

该模式的一个主要特点是将具体产品的实例化过程推迟到子类（具体工厂）中进行。Factory 类中的 create 方法作为一个模板方法，它调用了在子类中实现的、用于实际创建对象的抽象方法（通常也称为工厂方法）。此外，该实现还提及了产品注册功能，这可能是指在工厂内部维护某种注册信息，以便于管理或追踪创建的产品。

在使用时，客户端代码会先实例化一个具体的工厂（例如 new IDCardFactory()），然后调用其 create 方法来获取所需的产品实例（Product card = factory.create("John");）。
工厂方法模式通过封装对象的创建过程，将创建逻辑与使用逻辑分离。这提高了系统的灵活性，便于扩展新的产品和对应的工厂，同时也为产品创建提供了一个统一的入口点。其核心思想是定义一个用于创建对象的接口，但让实现该接口的子类来决定具体要实例化的类是哪一个。

代码:

```java
package membercard;

import framework.*;

public class MemberCard extends Product {
    private String owner;
    private String level;

    MemberCard(String owner, String level) {
        System.out.println("Created " + owner + "'s " + level + " membership card.");
        this.owner = owner;
        this.level = level;
    }

    public void use() {
        System.out.println("Using " + owner + "'s " + level + " membership card.");
    }

    public String getOwner() {
        return owner;
    }

    public String getLevel() {
        return level;
    }
}
```

MemberCard 类

```java
package membercard;

import framework.*;
import java.util.*;

public class MemberCardFactory extends Factory {
    private HashMap<String, String> members = new HashMap<String, String>();

    protected Product createProduct(String owner) {
        // Create default regular membership card
        return new MemberCard(owner, level:"Regular");
    }

    // Overloaded method to specify membership level
    public Product create(String owner, String level) {
        Product p = new MemberCard(owner, level);
        registerProduct(p);
        return p;
    }

    protected void registerProduct(Product product) {
        MemberCard card = (MemberCard) product;
        members.put(card.getOwner(), card.getLevel());
    }

    public HashMap<String, String> getMembers() {
        return members;
    }
}
```

MemberCardFactory 类

1. 创建了一个新的包 membercard，包含两个类：
   MemberCard：会员卡类，继承自 Product 抽象类，实现了 use()方法
   MemberCardFactory：会员卡工厂类，继承自 Factory 抽象类，实现了创建产品和注册产品的方法
2. 会员卡与 ID 卡相比的不同点：
   会员卡增加了"级别"属性，可以创建不同级别的会员卡
   会员卡工厂使用 HashMap 存储会员信息，而不是简单的 Vector
   会员卡工厂提供了一个重载的 create 方法，可以指定会员卡级别

运行截图：

2、请举例说明其他的工厂模式的应用。

日志记录框架:

应用: 可以定义一个 ILogger 接口和一个 LoggerFactory 基类（或接口）。LoggerFactory 中有一个抽象的 CreateLogger() 方法（工厂方法）。具体的日志记录器创建逻辑由子类实现，例如 FileLoggerFactory 创建 FileLogger，ConsoleLoggerFactory 创建 ConsoleLogger。客户端代码通过配置选择合适的 LoggerFactory 子类来获取 ILogger 实例，实现了日志记录方式的灵活切换。

支付网关集成:

场景: 电商平台或服务需要对接多种支付渠道（如支付宝、微信支付、PayPal、Stripe 等）。
应用 ： 可以定义一个 IPaymentGateway 接口，包含 ProcessPayment(), Refund() 等方法。创建一个 PaymentGatewayFactory，根据用户选择的支付方式或订单类型，创建并返回相应的支付网关实现类（AlipayGateway, WechatPayGateway, PayPalGateway 等）。这使得添加新的支付渠道或切换支付逻辑更加容易，核心支付流程代码保持稳定。

依赖注入 (DI) / 控制反转 (IoC) 容器:

应用：虽然 DI/IoC 容器本身是更复杂的设计，但其内部核心就是负责对象的创建和组装。它们可以看作是高度配置化和自动化的“超级工厂”。开发者配置好对象之间的依赖关系以及如何创建对象（例如，单例、每次请求一个新实例等），容器在需要时会自动使用类似工厂的机制来实例化对象及其依赖项，并将它们注入到需要的地方。