工厂方法模式

亦称：多态工厂(Polymorphic Factory)模式，虚拟构造器(Virtual Constructor)模式

前言

因为简单工厂模式存在一系列的问题

- 工厂类集中了所有实例（产品）的创建逻辑，一旦这个工厂不能正常工作，整个系统都会受到影响；
- 违背“开放 - 关闭原则”，一旦添加新产品就不得不修改工厂类的逻辑，这样就会造成工厂逻辑过于复杂。
- 简单工厂模式由于使用了静态工厂方法，静态方法不能被继承和重写，会造成工厂角色无法形成基于继承的等级结构。

为了解决上述的问题，我们又使用了一种新的设计模式：工厂方法模式。

介绍

定义

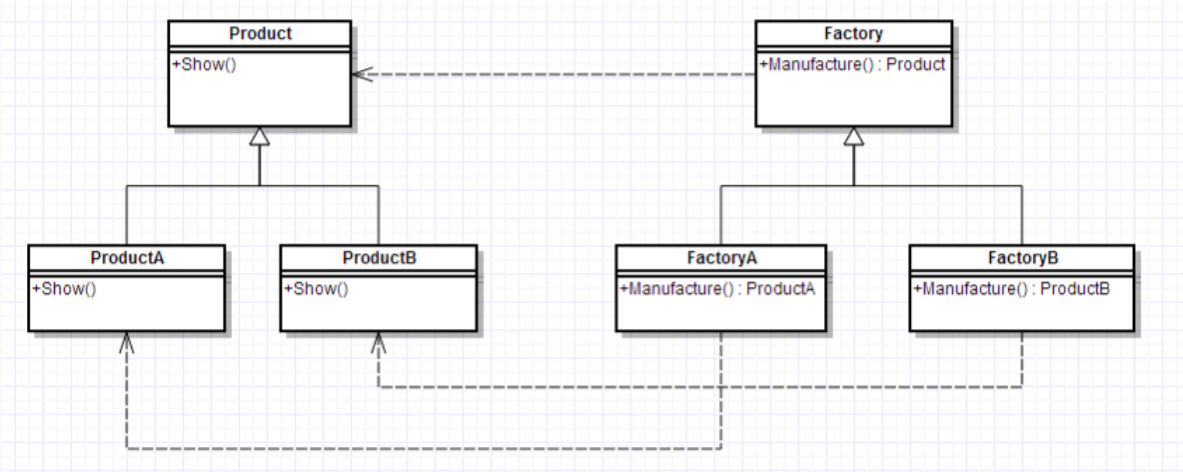
工厂方法模式通过定义工厂抽象父类(或接口)负责定义创建对象的公共接口，而工厂子类(或实现类)则负责生成具体的对象。

主要作用

将类的实例化（具体产品的创建）延迟到工厂类的子类（具体工厂）中完成，即由子类来决定应该实例化（创建）哪一个类。

模式原理

UML类图



模式组成

组成（角色）	关系	作用
抽象产品（Product）	具体产品的父类	描述具体产品的公共接口
具体产品（Concrete Product）	抽象产品的子类；工厂类创建的目标类	描述生产的具体产品
抽象工厂（Creator）	具体工厂的父类	描述具体工厂的公共接口
具体工厂（Concrete Creator）	抽象工厂的子类；被外界调用	描述具体工厂；实现FactoryMethod工厂方法创建产品的实例

使用步骤

步骤1： 创建**抽象工厂类**，定义具体工厂的公共接口；
步骤2： 创建**抽象产品类**，定义具体产品的公共接口；
步骤3： 创建**具体产品类**（继承抽象产品类） & 定义生产的具体产品；
步骤4：创建**具体工厂类**（继承抽象工厂类），定义创建对应具体产品实例的方法；
步骤5：外界通过调用具体工厂类的方法，从而创建不同**具体产品类的实例**

实例演示

- 背景：小成有一间塑料加工厂（仅生产A类产品）；随着客户需求的变化，客户需要生产B类产品；
- 冲突：改变原有塑料加工厂的配置和变化非常困难，假设下一次客户需要再发生变化，再次改变将增大非常大的成本；
- 解决方案：小成决定置办**塑料分厂B**来生产B类产品；

代码实现

步骤1： 创建**抽象工厂类**，定义具体工厂的公共接口

```
abstract class Factory{  // 也可定义为接口，扩展性会更好
    public abstract Product Manufacture();
}
```

步骤2： 创建**抽象产品类**，定义具体产品的公共接口；

```
abstract class Product{
    public abstract void Show();
}
```

步骤3： 创建**具体产品类**（继承抽象产品类），定义生产的具体产品；

```
//具体产品A类
class  ProductA extends  Product{
    @Override
    public void Show() {
        System.out.println("生产出了产品A");
```

```
    }
}

//具体产品B类
class  ProductB extends  Product{

    @Override
    public void Show() {
        System.out.println("生产出了产品B");
    }
}
```

步骤4：创建具体工厂类（继承抽象工厂类），定义创建对应具体产品实例的方法；

```
//工厂A类 - 生产A类产品
class  FactoryA extends Factory{
    @Override
    public Product Manufacture() {
        return new ProductA();
    }
}

//工厂B类 - 生产B类产品
class  FactoryB extends Factory{
    @Override
    public Product Manufacture() {
        return new ProductB();
    }
}
```

步骤5：外界通过调用具体工厂类的方法，从而创建不同具体产品类的实例

```
//生产工作流程
public class FactoryPattern {
    public static void main(String[] args){
        //客户要产品A
        FactoryA mFactoryA = new
FactoryA();
        mFactoryA.Manufacture().Show();

        //客户要产品B
        FactoryB mFactoryB = new
FactoryB();
        mFactoryB.Manufacture().Show();
    }
}
```

结果

生产出了产品A
生产出了产品B

模式优缺点

工厂方法模式的优点如下：

- 客户端只依赖产品的抽象，符合依赖倒置原则；无需关注具体产品，符合迪米特(最少知识)原则。

- 工厂方法模式符合开放封闭原则，新增产品时只需增加相应的产品和工厂类，而无需修改现有代码。
- 工厂方法模式符合单一职责原则，每个具体的工厂类只负责创建对应的产品。
- 工厂方法模式不使用静态工厂方法，可以形成基于继承的等级结构。

缺点如下：

- 新增产品时除增加新产品类外，还要增加对应的具体工厂类，没有简单工厂代码简洁、高效。
- 为了扩展性而进一步引入抽象层，增加了系统的抽象性和理解难度。

扩展

迪米特法则

定义

迪米特法则(Law of Demeter, LoD)是1987年秋天由lan holland在美国东北大学一个叫做迪米特的项目设计提出的，它要求**一个对象应该对其他对象有最少的了解**，所以迪米特法则又叫做最少知识原则（Least Knowledge Principle, LKP）。

意义

迪米特法则的意义在于降低类之间的耦合。由于每个对象尽量减少对其他对象的了解，因此，很容易使得系统的功能模块功能独立，相互之间不存在（或很少有）依赖关系。

值得一提的是，这一法则却不仅仅局限于计算机领域，在其他领域也同样适用。比如，美国人就在航天系统的设计中采用这一法则。

六大设计原则

- Single Responsibility Principle：单一职责原则
- Open Closed Principle：开闭原则
- Liskov Substitution Principle：里氏替换原则
- Law of Demeter：迪米特法则
- Interface Segregation Principle：接口隔离原则
- Dependence Inversion Principle：依赖倒置原则

应用场景

- 当一个类不知道它所需要的对象的类时
  在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可；
- 当一个类希望通过其子类来指定创建对象时
  在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序

运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

- 将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

与其他模式的关系

- 在许多设计工作的初期都会使用工厂方法模式 （较为简单， 而且可以更方便地通过子类进行定制）， 随后演化为使用抽象工厂模式、 原型模式或生成器模式 （更灵活但更加复杂）。
- 抽象工厂模式通常基于一组工厂方法， 但你也可以使用原型模式来生成这些类的方法。
- 你可以同时使用工厂方法和迭代器模式来让子类集合返回不同类型的迭代器， 并使得迭代器与集合相匹配。
- 原型并不基于继承， 因此没有继承的缺点。 另一方面， 原型需要对被复制对象进行复杂的初始化。 工厂方法基于继承， 但是它不需要初始化步骤。
- 工厂方法是模板方法模式的一种特殊形式。 同时，工厂方法可以作为一个大型模板方法中的一个步骤。

总结

工厂方法模式对简单工厂模式中的工厂类进一步抽象。核心工厂类不再负责产品的创建，而是演变为一个抽象工厂角色，仅负责定义具体工厂子类必须实现的接口。同时，针对不同的产品提供不同的工厂。即**每个产品都有一个与之对应的工厂**。这样，系统在增加新产品时就不会修改工厂类逻辑而是添加新的工厂子类，从而弥补简单工厂模式对修改开放的缺陷。

在实际项目中，工厂方法模式是使用较多的工厂模式。