设计模式: 工厂（Factory）模式

实验编号 exp04

0.1 知识拓展：设计模式概述

设计模式是一套被反复使用、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。

0.1.1 设计模式体现的六大原则

1. 开闭原则（Open Close Principle）
   开闭原则即是对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类。

2. 里氏代换原则（Liskov Substitution Principle）
   里氏代换原则（LSP）面向对象设计的基本原则之一，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充，实现开闭原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

3. 依赖倒转原则（Dependence Inversion Principle）
   依赖倒转原则是开闭原则的基础，具体内容包括：针对接口编程，依赖于抽象而不依赖于具体。

4. 接口隔离原则（Interface Segregation Principle）
   接口隔离原则是使用多个隔离的接口，比使用单个接口要好。

5. 迪米特法则（最少知道原则）（Demeter Principle）
   迪米特法则（最少知道原则）就是说一个实体应当尽量少的与其他实体之间发生相互作用，

使得系统功能模块相对独立。

6. 合成复用原则（Composite Reuse Principle）
   合成复用原则是尽量使用合成/聚合的方式，而不是使用继承。

0.1.2 设计模式的分类

Java 设计模式可以分为以下几个大类：

创建型模式 共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式 共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式 共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其他 并发型模式和线程池模式。

0.2 实验目的

通过本实验章节的学习，需要完成：

1. 理解设计模式之工厂（Factory）模式。
2. 理解工厂模式中所包含的 Java 设计原则。
3. 学习使用工厂（Factory）模式编写程序。
4. 为后续持续学习掌握 Java 其他设计模式打下一定的基础。

0.3 问题分析

我们首先来看一个不够合理的披萨店设计：

```
1  public class OldStore {
2
3    Pizza orderPizza() {
4      Pizza pizza = new Pizza();
5
6      pizza.prepare();
7      pizza.bake();
8      pizza.cut();
9      pizza.box();
10     return pizza;
11   }
12 }
```

在上述代码中，为了让系统有弹性，我们希望 Pizza 类是个抽象类或者接口。但如果是这样，这个类或接口就无法直接实例化。

当我们需要更多种类的 Pizza 时，我们必须增加代码来确定适合的 Pizza 类型，然后再制造 Pizza。所以，披萨店设计可能变成了如下的样子：

```
public class OldStore {

  Pizza OrderPizza(String type) {
    Pizza pizza;
    // ---------  不断变化的部分 ----------
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("greek")) {
      pizza = new GreekPizza();
    } else if (type.equals("other")) {
      // other type.
    }
    //     ---------------------------------
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

但是，如果需要不断的增加 Pizza 的种类，我们需要不断的修改 orderPizza() 方法的代码，这使得orderPizza() 无法对修改关闭，所以我们需要进一步使用封装。

接下来，我们将分析三种类型的工厂模式，严格的说第一种称不上是设计模式，而可以理解为一种编程习惯。

0.3.1 简单工厂

❶ 封装创建对象的代码 将创建 Pizza 对象的代码移到 orderPizza() 方法之外的另一个对象 SimplePizzaFactory 中，由这个新对象专职创建披萨。我们称这个新对象为“工厂”。这样，orderPizza() 只需要关心从工厂得到一个披萨，然后进行后续的操作。

```
public class OldStore {

  Pizza OrderPizza() {
    Pizza pizza;

    // 披萨工厂，应该如何实现？

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
```

```
  }
}
```

❷ 建立简单的披萨工厂　建立工厂类，为所有披萨封装创建对象的代码。

```
public class SimplePizzaFactroy {

  public Pizza createPizza(String type) {
    Pizza pizza = null;

    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("greek")) {
      pizza = new GreekPizza();
    } else if (type.equals("other")) {
      // other type.
    }

    return pizza;
  }
}
```

当然，上述实现并不完美。

❸ 重做 PizzaStore 类　代码如下。

```
public class PizzaStore {
  SimplePizzaFactory factory; // 加入一个对 SimplePizzaFactory 的引用

  // 构造器需要一个工厂作为参数
  public PizzaStore(SimplePizzaFactory factory) {
    this.factory = factory;
  }

  public Pizza orderPizza(String type) {
    Pizza pizza;

    // 该方法通过简单传入订单类型来使用工厂创建披萨
    pizza = factory.createPizza(type); // new 操作符被替换为工厂对象创建方法

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

❸ 简单工厂小结　简单工厂类图如图 1所示。

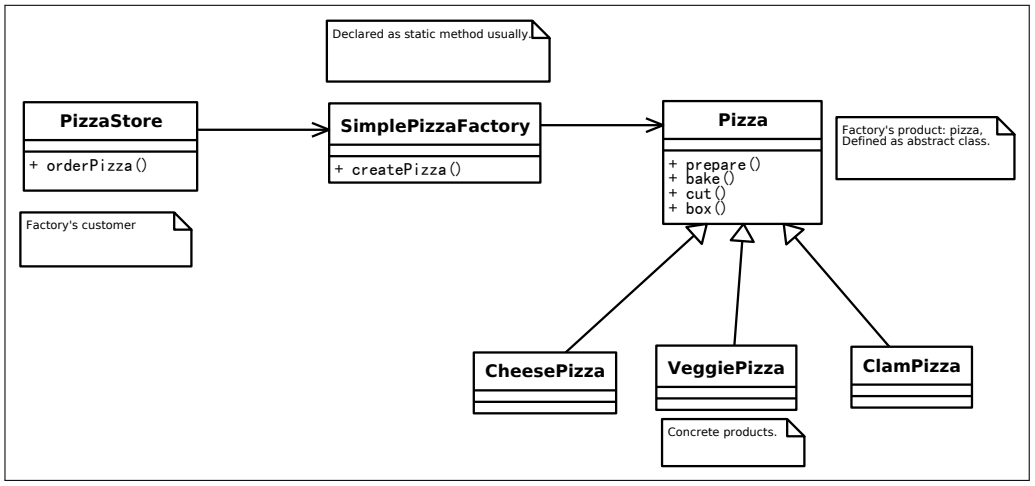

图 1: 简单工厂类图

0.3.2 工厂方法

❶ 加盟披萨店 要求披萨店的各个加盟店能够提供不同风味的披萨，并复用代码，以使得披萨的流程能够一致不变。利用 SimplePizzaFactory 的一般实现如下：

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("Veggie") ;

ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore nyStore = new PizzaStore(chicagoFactory);
nyStore.orderPizza("Veggie") ;
```

希望能够建立一个框架，把加盟店和创建披萨捆绑在一起，保持披萨的质量控制，同时使得代码具有一定的弹性。

❷ 给披萨店使用的框架 采用如下框架可以使得披萨制作活动局限于 PizzaStore 类，而同时又能让这些加盟店依然可以自由的制作该区域的风味。

```
public abstract class PizzaStore {

  public Pizza OrderPizza(String type) {
    Pizza pizza;

    // createPizza() 方法从工厂对象中回到 PizzaStore
    pizza = createPizza(type);

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
```

```
14    }
16    // PizzaStore 里的工厂方法是抽象的
17    abstract Pizza createPizza(String type);
18  }
```

现在，我们将 PizzaStore 作为超类，让每个区域类型（NYPizzaStore 等）都继承这个 PizzaStore，每个子类各自决定如何制作披萨。

❸ 允许子类做决定 orderPizza() 方法负责处理订单，使得所有加盟店对于订单的处理保持一致，如图 2所示。

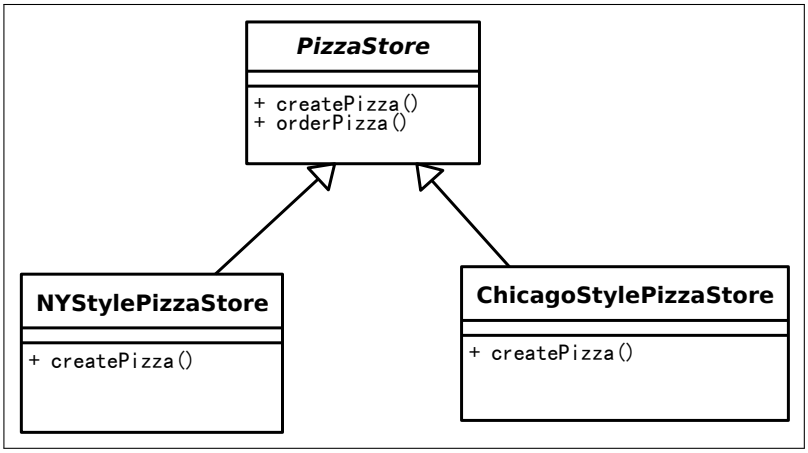

图 2: 加盟店实现超类抽象方法

createPizza() 方法是个抽象方法，所有任何具体的加盟店必须实现这个方法，从而个性化加盟店的披萨风味。例如，我们创建 NYPizzaStore。

```
1   public class NYPizzaStore extends PizzaStore {
3     @Override
4     Pizza createPizza(String type) {
6       if (type.equals("cheese")) {
7         return new NYStyleCheesePizza();
8       } else if (type.equals("veggie")) {
9         return new NYStyleVeggiePizza();
10      } else if (type.equals("other")) {
11        // other type.
12      } else
13        return null;
14    }
15  }
```

❹ 声明一个工厂方法 原来是在简单工厂中是由一个对象负责所有具体类的实例化，现在通过对 PizzaStore 作改变，使得由一群子类来负责实例化。

工厂方法用来处理对象的创建，并将这样的行为封装在子类中，这样客户程序中关于超类的代码就和子类对象创建代码解耦。

- 工厂方法是抽象；
- 工厂方法必须返回一个产品，超类中定义的方法，通常会用到工厂方法的返回值；
- 工厂方法将客户（即超类中的代码，如 orderPizza()）和实际创建具体产品的代码分隔开。

```
abstract Product factoryMethod(String type);
```

❺ 制作一些披萨用于出售 声明为抽象类，所有的具体披萨都必须派生自这个类。

```
public abstract class Pizza {
  String name; // 披萨名字
  String dough; // 面团类型
  String sauce;  // 酱料类型
  ArrayList toppings = new ArrayList(); // 一套佐料
  public void prepare() {
    // 具体实现
  }
  public void bake() {
    // 具体实现
  }
  public void cut() {
    // 具体实现
  }
  public void box() {
    // 具体实现
  }
}
```

❻ 工厂方法订购和生产披萨的过程（main() 方法）

1. 需要一个纽约披萨店。

```
    PizzaStore nyPizzaStore = new NYPizzaStore();
```

2. 下订单。

```
    nyPizzaStore.orderPizza("cheese"); // 该方法在 PizzaStore 中定义
```

3. orderPizza() 方法调用 createPizza() 方法

```
Pizza pizza = createPizza("chesse"); // 工厂方法在具体子类中实现
pizza.prepare();      // orderPizza() 方法并不知道披萨的具体类
pizza.bake();
pizza.cut();
pizza.box();
```

❼ 工厂方法模式小结 工厂方法模式（Factory Method）包含的组成元素：

- 创建者类，定义一个抽象工厂方法，让子类实现此方法制造产品。
- 产品类，抽象产品类，子类实现具体的产品。
- 工厂生产产品。

☛ 工厂方法模式定义

工厂方法模式定义了一个创建对象的接口，但由子类决定要实例化的类是那一个，工厂方法让类把实例化推迟到子类。

0.3.3 抽象工厂

工厂方法模式从设计角度考虑存在一定的问题：类的创建依赖工厂类。也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则。

所以，可以使用抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。参考示例的3类图，该示例包含一个消息发送者接口(Sender) 和一个抽象工厂接口（Provider）。

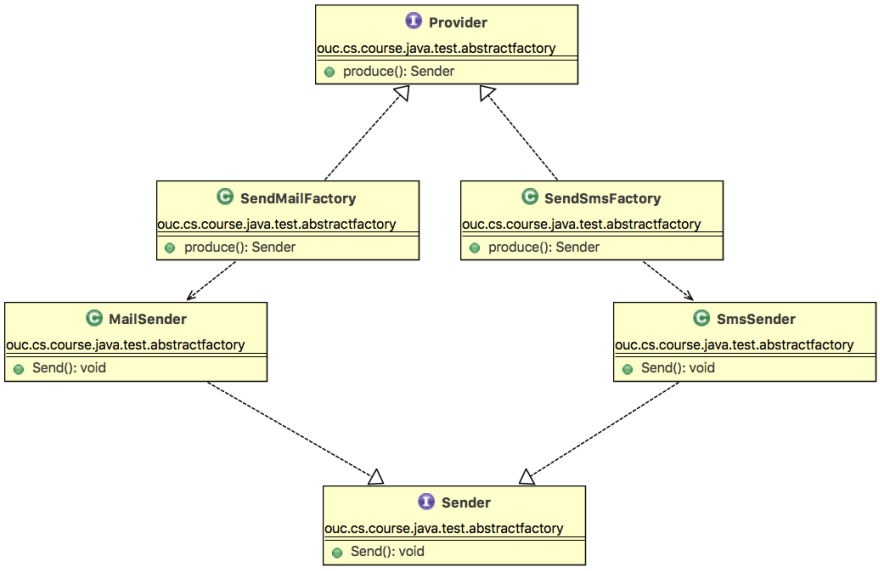

图 3: 抽象工厂示例类图

相关代码如下：

File: Sender.java

```
package ouc.cs.course.java.test.abstractfactory;

public interface Sender {
  public void Send();
}
```

File: Provider.java

```
package ouc.cs.course.java.test.abstractfactory;

public interface Provider {
  public Sender produce();
}
```

File: MailSender.java

```
package ouc.cs.course.java.test.abstractfactory;

public class MailSender implements Sender {

  public void Send() {
    System.out.println("this is mailsender!");
  }
}
```

File: SmsSender.java

```
package ouc.cs.course.java.test.abstractfactory;

public class SmsSender implements Sender {

  public void Send() {
    System.out.println("this is smssender!");
  }
}
```

File: SendMailFactory.java

```
package ouc.cs.course.java.test.abstractfactory;

public class SendMailFactory implements Provider {

  @Override
  public Sender produce() {
    return new MailSender();
  }
}
```

File: SendSmsFactory.java

```
package ouc.cs.course.java.test.abstractfactory;

public class SendSmsFactory implements Provider {

  @Override
  public Sender produce() {
    return new SmsSender();
  }
}
```

File: Test.java

```
package test.abstractfactory;

import ouc.cs.course.java.test.abstractfactory.Provider;
import ouc.cs.course.java.test.abstractfactory.SendMailFactory;
import ouc.cs.course.java.test.abstractfactory.Sender;

public class Test {
  public static void main(String[] args) {
    Provider provider = new SendMailFactory();
    Sender sender = provider.produce();
    sender.Send();
  }
}
```

0.4 实验要求

学习本章内容并参考问题分析章节 0.3，编写工厂模式的程序实例。

0.5 实验过程、步骤及原始记录

实验过程和代码如下：

```
.
.
.
.
.
.
.
.
.
```

0.6 参考程序

本章节参考前述代码示例即可。