



LANZHOU UNIVERSITY OF TECHNOLOGY

结课作业

题 目 设计模式期末测评作业

学生姓名 李广清

学 号 1817700424

专业班级 软件四班

学 院 软件学院

目录

[一、设计模式的分类 1](#_Toc42677437)

[二、介绍工厂模式以及例子 1](#_Toc42677438)

[2.1简单工厂模式 1](#_Toc42677439)

[2.2工厂方法模式 5](#_Toc42677440)

[2.3抽象工厂模式 9](#_Toc42677441)

[2.4工厂模式优缺点 12](#_Toc42677442)

## 一、设计模式的分类

总体来说设计模式分为三大类：

1.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。

2.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。

3.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。

## 二、介绍工厂模式以及例子

### 2.1简单工厂模式

一个抽象产品类，派生出多个具体产品类

一个工厂类，通过逻辑判断，实例化所需的产品类

具体的描述

(1) 抽象产品类：为各种具体产品声明抽象类或者接口

(2) 具体产品类：抽象产品的具体实现，对应具体的产品

(3) 工厂类：用户只关注产品，无需关注具体实现，生产出具体的产品

工厂方法模式

一个抽象产品类，派生出多个具体产品类

一个抽象工厂类，派生出多个具体工厂类

每个具体工厂类只能生产出一个具体的产品类

具体的描述

(1) 抽象产品类：为各种具体产品声明抽象类或者接口

(2) 具体产品类：抽象产品的具体实现

(3) 抽象工厂类：声明了一个方法，只对应一种产品

(4) 具体工厂类：抽象工厂的实现，生成一个具体产品 (一组产品)

抽象工厂模式

多个抽象产品类，每个抽象产品类都可以派生出多个具体产品类

一个抽象工厂类，可以派生出多个具体工厂类

每个具体工厂类都能生成出多个具体产品类 （后续都有相应代码，看了就会懂）

具体的描述

(1) 抽象产品类：为各种具体产品声明抽象类或者接口

(2) 具体产品类：抽象产品的具体实现

(3) 抽象工厂类：声明了一组方法，每个方法对应一种类型产品

(4) 具体工厂类：抽象工厂的实现，生成一组具体产品 (一组产品)

代码实现

简单工厂模式

抽象产品类

package com.nicecui.design.factory.simple;

/\*\*

\*

\*

\*/

public abstract class Fruits {

public abstract void print();

}

具体产品类

// 苹果

public class Apple extends Fruits {

@Override

public void print() {

System.out.println("This is a Apple");

}

}

//橙子

public class Orange extends Fruits {

@Override

public void print() {

// TODO Auto-generated method stub

System.out.println("This is a Orange");

}

}

工厂类

package com.nicecui.design.factory.simple;

/\*\*

\*

\* FruitsFactory

\* @author NiceCui

\* @date 2018年1月23日上午11:33:20

\*

\*/

public class FruitsFactory {

public Fruits getFruit(String name) {

Fruits fruit = null;

if(""==name||name == null) {

return fruit;

}

switch (name) {

case "Apple":

fruit = new Apple();

break;

case "Orange":

fruit = new Orange();

break;

}

return fruit;

}

}

### 2.2工厂方法模式

实现一个加减的计算机

抽象产品类

package com.qimi;

public abstract class Operation {

public int A;

public int B;

public int getA() {

return A;

}

public void setA(int a) {

A = a;

}

public int getB() {

return B;

}

public void setB(int b) {

B = b;

}

public abstract void getResult() ;

}

具体产品类

//加法

public class OperationAdd extends Operation{

public int a;

public int b;

public OperationAdd() {

}

@Override

public void getResult() {

System.out.println("a+b="+(super.A+super.B));

}

}

// 减法

public class OperationSub extends Operation{

public int a;

public int b;

@Override

public void getResult() {

// TODO Auto-generated method stub

System.out.println("a-b="+(super.A - super.B));

}

}

抽象工厂类和具体工厂

package com.qimi;

public interface IFactory {

Operation createOperation();

}

class AddFactory implements IFactory{

@Override

public Operation createOperation() {

// TODO Auto-generated method stub

return new OperationAdd();

}

}

class SubFactory implements IFactory{

@Override

public Operation createOperation() {

// TODO Auto-generated method stub

return new OperationSub();

}

}

### 2.3抽象工厂模式

生产一辆汽车的工厂，配置不一样，有进口和本地发动机和在淘宝和京东购买的座椅

抽象产品类

//座椅

public interface Chair {

public void getChair();

}

//发动机

public interface Engine {

public void getEngine ();

}

具体产品类

/\*\*座椅\*\*/

//京东买的

public class JD\_Chair implements Chair{

@Override

public void getChair() {

// TODO Auto-generated method stub

System.out.println("从京东购买的座椅....");

}

}

//淘宝买的

public class TB\_Chair implements Chair {

@Override

public void getChair() {

// TODO Auto-generated method stub

System.out.println("从淘宝买的座椅.....");

}

}

/\*\*发动机\*\*/

//进口的

public class Import\_Engine implements Engine{

@Override

public void getEngine() {

// TODO Auto-generated method stub

System.out.println("使用进口发动机....");

}

}

//本地的

public class Local\_Engine implements Engine{

@Override

public void getEngine() {

// TODO Auto-generated method stub

System.out.println("使用本地发动机....");

}

}

抽象工厂

public interface CarFactory {

//一组方法 对应 一组产品 上面有讲过

public Chair getChair();

public Engine getEngine();

}

具体工厂

生产具体产品

//生产A型号的汽车

public class AcarFactory implements CarFactory{

@Override

public Chair getChair() {

// TODO Auto-generated method stub

return new JD\_Chair();

}

@Override

public Engine getEngine() {

// TODO Auto-generated method stub

return new Local\_Engine();

}

}

//生产B型号的汽车

public class BcarFactory implements CarFactory{

@Override

public Chair getChair() {

// TODO Auto-generated method stub

return new TB\_Chair();

}

@Override

public Engine getEngine() {

// TODO Auto-generated method stub

return new Import\_Engine();

}

}

### 2.4工厂模式优缺点

**简单工厂模式**

**优点：**

第一点当然是简单了，当耦合不是很大时候，简单工厂很适合

最大的优点就是将类的实例化交给工厂类，工厂类中包含了必要的逻辑判断。

客户端角度分析客户选择条件动态的实例化相关的类，去除了与具体产品的依赖。

**缺点：**

简单工厂模式不利于拓展，违背了开放封闭原则，每次添加一个类，都要修改工厂类。

**工厂方法模式**

**优点：**

工厂方法模式对简单工厂模式进行了升级，将类的实例化延迟到了其子类(也就是具体工厂类)。

拓展添加具体类，无需对抽象工厂类修改，只需要添加想要的具体类和具体工厂即可。

**缺点：**

面对再复杂的有点力不从心，例如：如若增加另一个产品体系，刚才我们的工厂方法是实现计算机，若果我们要来一台电视机呢，还是得再去抽象工厂添加修改，这样也会违背开放封闭原则。

**抽象工厂模式**

**优点：**

可以针对多个产品系，很容易组合产品系列，只需要去实现具体的工厂类即可；例如：我们再添加一个C型号的车 需要 进口发动机和京东座椅 那我们只需要去添加相应的具体工厂就可以了。

具有工厂方法模式解耦的优点。

产品系 进口发动机和本地发动机为两个不同的等级结构，搭载进口发动机的车子就是一个产品系。

**缺点：**

对于产品系的扩展将非常费劲，例如我们要生产的所有车子，需要配备 进口轮胎或者本地轮胎，产品系中需要增加一个新的产品，则几乎所有的工厂类都需要进行修改。