**设计模式 实验3-工厂模式**

一.简答题（共1题）

某连锁超市为留住顾客，决定发行会员卡。会员卡分为金卡、银卡、普通卡等多种类型，卡的类型可能还会增加。不论是什么卡，其编号都是统一编制的，即使用同一个编号序列。某软件公司承接了该项目，设计人员觉得应该将系统设计的具有一定的通用性，不仅可以较方便的应对该连锁店的需求变化（增减卡的类型），而且可以较容易的修改扩展，为其他公司提供发卡（卡的风格可能不同）服务。假定你是该软件公司的设计师，请你运用学过的设计模式设计该系统的架构。编写实验报告，要求画出相应的类图，提供java源程序。

1. 类图

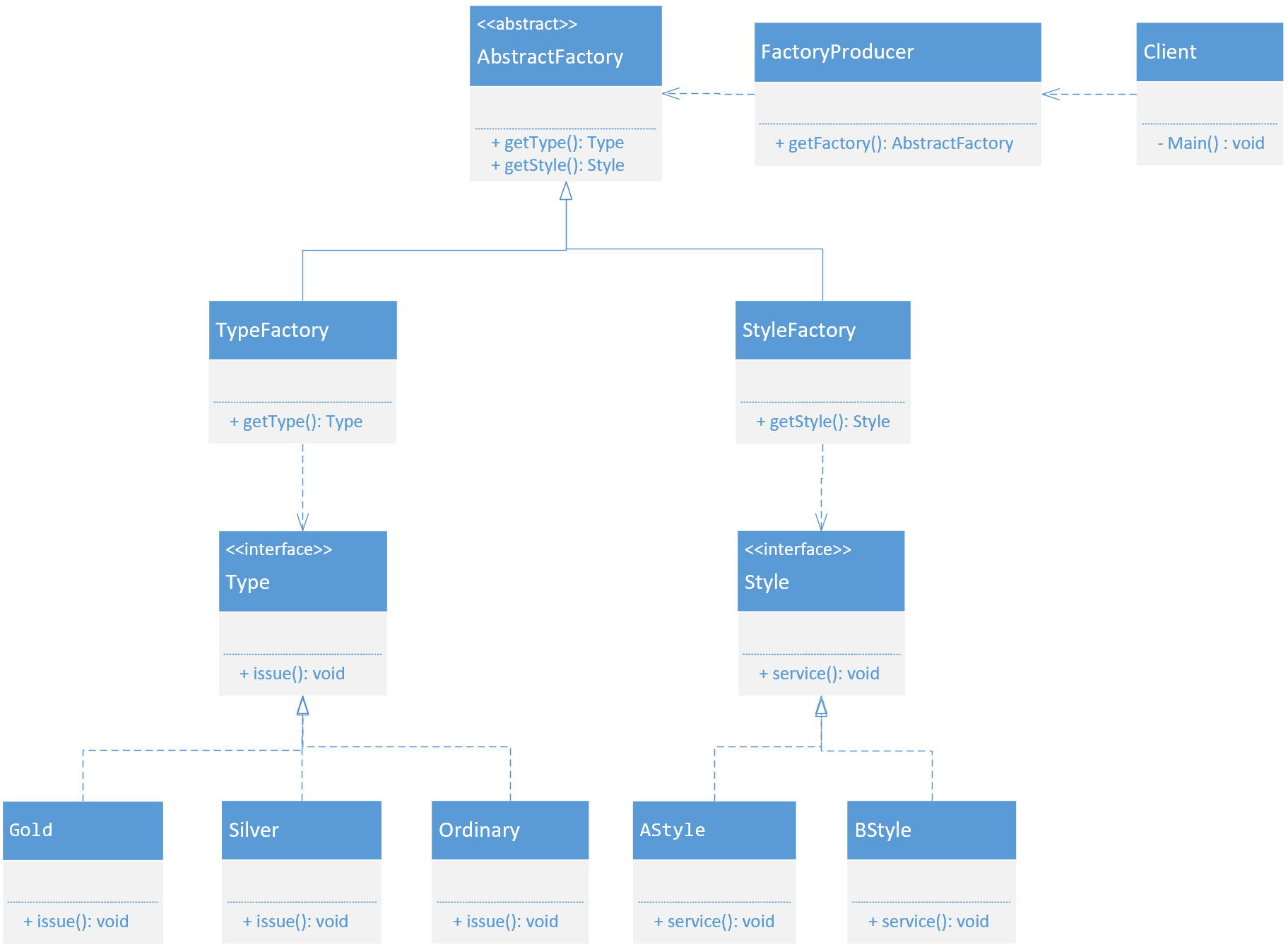
该系统涉及到会员卡的类型和风格，运用抽象工厂模式设计架构较为合适。

会员卡类型(Type)和发卡服务风格(Style)作为产品接口，派生出金卡(Gold)、银卡(Silver)、普通卡(Ordinary)等多种类型的Type接口实体类，和AStyle、BStyle等多种风格的Style接口实体类。

AbstractFactory作为抽象工厂类，派生出创建卡类型的工厂(TypeFactory)和创建发卡服务风格的工厂(StyleFactory)等多种具体工厂类。

FactoryProducer作为工厂生成器类，通过传递卡的类型或服务的信息来获取工厂。

Client作为客户端模拟系统运作。



1. 源程序



①创建会员卡类型接口，声明会员卡发行方法issue()

/\*\*  
 \* 接口 会员卡类型  
 \*/  
public interface Type {  
 void issue();  
}

②创建实现接口的实体类，实现不同卡的发行方法  
/\*\*  
 \* 接口实现类 金卡  
 \*/  
public class Gold implements Type{  
 @Override  
 public void issue() {  
 System.out.println("发行金卡");  
 }  
}

/\*\*  
 \* 接口实现类 银卡  
 \*/  
public class Silver implements Type{  
 @Override  
 public void issue() {  
 System.out.println("发行银卡");  
 }  
}

/\*\*  
 \* 接口实现类 普通卡  
 \*/  
public class Ordinary implements Type{  
 @Override  
 public void issue() {  
 System.out.println("发行普通卡");  
 }  
}

③创建会员卡风格接口，声明发卡服务方法service()  
/\*\*  
 \* 接口 会员卡风格  
 \*/  
public interface Style {  
 void service();  
}

④创建实现接口的实体类，实现不同风格会员卡的发卡服务方法

/\*\*  
 \* 接口实现类 A风格  
 \*/  
public class AStyle implements Style{  
 @Override  
 public void service() {  
 System.out.println("提供A风格服务");  
 }  
}

/\*\*  
 \* 接口实现类 B风格  
 \*/  
public class BStyle implements Style{  
 @Override  
 public void service() {  
 System.out.println("提供B风格服务");  
 }  
}

⑤为 Type 和 Style 对象创建抽象类来获取工厂

/\*\*  
 \* 抽象工厂类  
 \*/  
public abstract class AbstractFactory {  
 public abstract Type getType(String type) ;  
 public abstract Style getStyle(String style);  
}

⑥创建扩展了 AbstractFactory 的工厂类，基于给定的信息生成实体类的对象

/\*\*  
 \* 具体工厂类 会员卡类型工厂  
 \*/  
public class TypeFactory extends AbstractFactory{  
 final class Const {  
 static final String TYPE\_A = "gold";  
 static final String TYPE\_B = "silver";  
 static final String TYPE\_C = "ordinary";  
 }  
 @Override  
 public Type getType(String type) {  
 switch (type) {  
 case Const.TYPE\_A:  
 return new Gold();  
 case Const.TYPE\_B:  
 return new Silver();  
 case Const.TYPE\_C:  
 return new Ordinary();  
 }  
 return null;  
 }  
 @Override  
 public Style getStyle(String style) {  
 return null;  
 }  
}

/\*\*  
 \* 具体工厂类 会员卡风格工厂  
 \*/  
public class StyleFactory extends AbstractFactory{  
 final class Const {  
 static final String Style\_A = "A";  
 static final String Style\_B = "B";  
 }  
 @Override  
 public Type getType(String type) {  
 return null;  
 }  
 @Override  
 public Style getStyle(String style) {  
 switch (style) {  
 case Const.Style\_A:  
 return new AStyle();  
 case Const.Style\_B:  
 return new BStyle();  
 }  
 return null;  
 }  
}

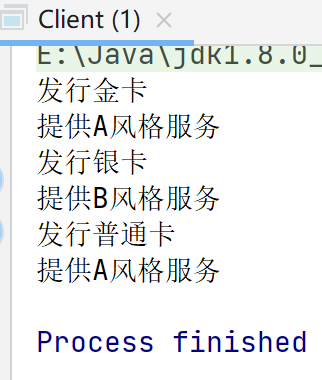
⑦创建一个工厂生成器类，通过传递卡的类型或服务的信息来获取工厂

/\*\*  
 \* 工厂生成器类  
 \*/  
public class FactoryProducer {  
 public static AbstractFactory getFactory(String choice){  
 if(choice.equalsIgnoreCase("type")){  
 return new TypeFactory();  
 } else if(choice.equalsIgnoreCase("style")){  
 return new StyleFactory();  
 }  
 return null;  
 }  
}

⑧使用 FactoryProducer 来获取 AbstractFactory，通过传递类型信息来获取实体类的对象

/\*\*  
 \* 客户端  
 \*/  
public class Client {  
 public static void main(String[] args) {  
 //获取卡类型工厂  
 AbstractFactory typeFactory = FactoryProducer.getFactory("type");  
 //获取卡风格工厂  
 AbstractFactory styleFactory = FactoryProducer.getFactory("style");  
  
 //获取金卡类型的对象  
 Type typeA = typeFactory.getType("gold");  
 //获取银卡类型的对象  
 Type typeB = typeFactory.getType("silver");  
 //获取普通卡类型的对象  
 Type typeC = typeFactory.getType("ordinary");  
  
 //获取A风格的对象  
 Style styleA = styleFactory.getStyle("A");  
 //获取B风格的对象  
 Style styleB = styleFactory.getStyle("B");  
  
  
 //发行金卡  
 typeA.issue();  
 //提供A风格发卡服务  
 styleA.service();  
  
 //发行银卡  
 typeB.issue();  
 //提供B风格发卡服务  
 styleB.service();  
  
 //发行普通卡  
 typeC.issue();  
 //提供A风格发卡服务  
 styleA.service();  
 }  
}

1. 程序运行结果



1. 总结

由于对于什么情况下使用哪种工厂模式比较模糊，所以罗列一下工厂模式的使用场景比较。

* 简单工厂模式：

产品种类相对较少，只要一个工厂类即可完成；

* 工厂方法模式：

①客户只知道创建产品的工厂名，而不知道具体的产品名（TCL 电视工厂、海信电视工厂）

②创建对象的任务由多个具体子工厂中的某一个完成，而抽象工厂只提供创建产品的接口

③客户不关心创建产品的细节，只关心产品的品牌

* 抽象工厂模式：

①创建对象是一系列相互关联或相互依赖的产品族时（电器工厂中的电视机、洗衣机、空调等）

②系统中有多个产品族，但每次只使用其中的某一族产品（有人只喜欢穿某一个品牌的衣服和鞋）

③系统中提供了产品的类库，且所有产品的接口相同，客户端不依赖产品实例的创建细节和内部结构