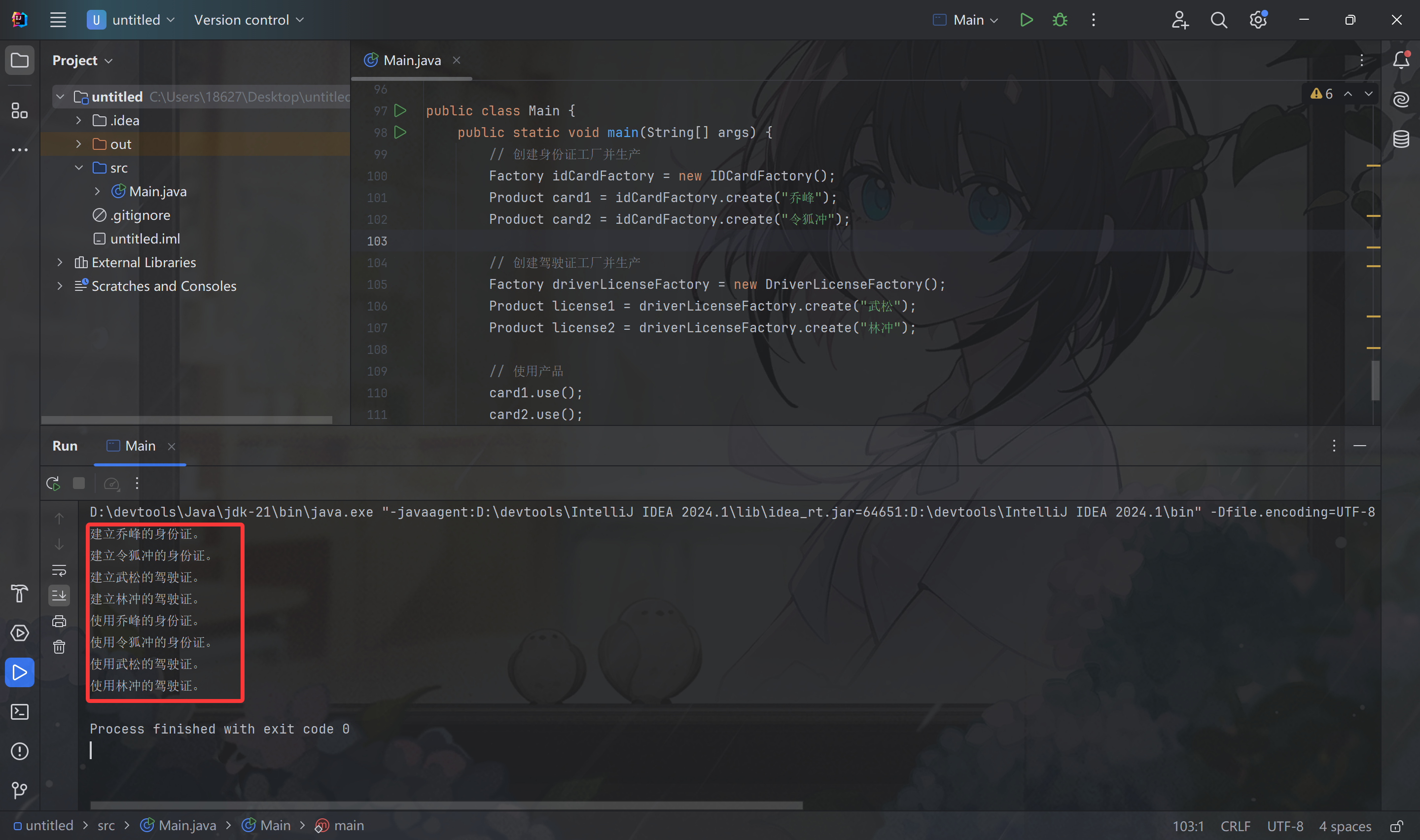
1. 改写本例，用于添加另一个具体工厂和具体产品

Code：

import java.util.Vector;  
  
// 框架层：抽象工厂  
abstract class Factory {  
 public final Product create(String owner) {  
 Product p = createProduct(owner);  
 registerProduct(p);  
 return p;  
 }  
  
 protected abstract Product createProduct(String owner);  
  
 protected abstract void registerProduct(Product product);  
}  
  
// 框架层：抽象产品  
abstract class Product {  
 public abstract void use();  
}  
  
// 原有身份证模块：具体产品  
class IDCard extends Product {  
 private String owner;  
  
 IDCard(String owner) {  
 System.*out*.println("建立" + owner + "的身份证。");  
 this.owner = owner;  
 }  
  
 @Override  
 public void use() {  
 System.*out*.println("使用" + owner + "的身份证。");  
 }  
  
 public String getOwner() {  
 return owner;  
 }  
}  
  
// 原有身份证模块：具体工厂  
class IDCardFactory extends Factory {  
 private Vector<String> owners = new Vector<>();  
  
 @Override  
 protected Product createProduct(String owner) {  
 return new IDCard(owner);  
 }  
  
 @Override  
 protected void registerProduct(Product product) {  
 owners.add(((IDCard) product).getOwner());  
 }  
  
 public Vector<String> getOwners() {  
 return owners;  
 }  
}  
  
// 新增驾驶证模块：具体产品  
class DriverLicense extends Product {  
 private String owner;  
  
 DriverLicense(String owner) {  
 System.*out*.println("建立" + owner + "的驾驶证。");  
 this.owner = owner;  
 }  
  
 @Override  
 public void use() {  
 System.*out*.println("使用" + owner + "的驾驶证。");  
 }  
  
 public String getOwner() {  
 return owner;  
 }  
}  
  
// 新增驾驶证模块：具体工厂  
class DriverLicenseFactory extends Factory {  
 private Vector<String> owners = new Vector<>();  
  
 @Override  
 protected Product createProduct(String owner) {  
 return new DriverLicense(owner);  
 }  
  
 @Override  
 protected void registerProduct(Product product) {  
 owners.add(((DriverLicense) product).getOwner());  
 }  
  
 public Vector<String> getOwners() {  
 return owners;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 // 创建身份证工厂并生产  
 Factory idCardFactory = new IDCardFactory();  
 Product card1 = idCardFactory.create("乔峰");  
 Product card2 = idCardFactory.create("令狐冲");  
  
 // 创建驾驶证工厂并生产  
 Factory driverLicenseFactory = new DriverLicenseFactory();  
 Product license1 = driverLicenseFactory.create("武松");  
 Product license2 = driverLicenseFactory.create("林冲");  
  
 // 使用产品  
 card1.use();  
 card2.use();  
 license1.use();  
 license2.use();  
 }  
}

运行结果：



1. 请举例说明其他的工厂模式的应用

比如实现一个**日志记录器**，可以使用工厂模式

1. 抽象层

抽象层定义日志接口Log和日志工厂接口LogFactory

Code：  
package log;  
  
public abstract class Log {  
 public abstract void writeLog();  
}  
  
// log/LogFactory.java  
package log;  
  
public abstract class LogFactory {  
 public abstract Log createLog();  
}

1. 具体实现类

具体实现文件日志和数据库日志的工厂及产品

Code：

package log;

public class FileLog extends Log {

@Override

public void writeLog() {

System.out.println("写入文件日志");

}

}

// log/DatabaseLog.java

package log;

public class DatabaseLog extends Log {

@Override

public void writeLog() {

System.out.println("写入数据库日志");

}

}

// log/FileLogFactory.java

package log;

public class FileLogFactory extends LogFactory {

@Override

public Log createLog() {

return new FileLog();

}

}

// log/DatabaseLogFactory.java

package log;

public class DatabaseLogFactory extends LogFactory {

@Override

public Log createLog() {

return new DatabaseLog();

}

}

1. 客户端调用

客户端通过工厂获取具体日志对象，符合开闭原则

Code：

import log.Log;

import log.LogFactory;

import log.FileLogFactory;

import log.DatabaseLogFactory;

public class LogClient {

public static void main(String[] args) {

// 使用文件日志工厂

LogFactory fileFactory = new FileLogFactory();

Log fileLog = fileFactory.createLog();

fileLog.writeLog(); // 输出：写入文件日志

// 使用数据库日志工厂

LogFactory dbFactory = new DatabaseLogFactory();

Log dbLog = dbFactory.createLog();

dbLog.writeLog(); // 输出：写入数据库日志

}

}