实验报告

实验设计名称： 工厂方法模式

专 业： 软 件 工技术

班 级： 2018级

姓 名： 林志龙

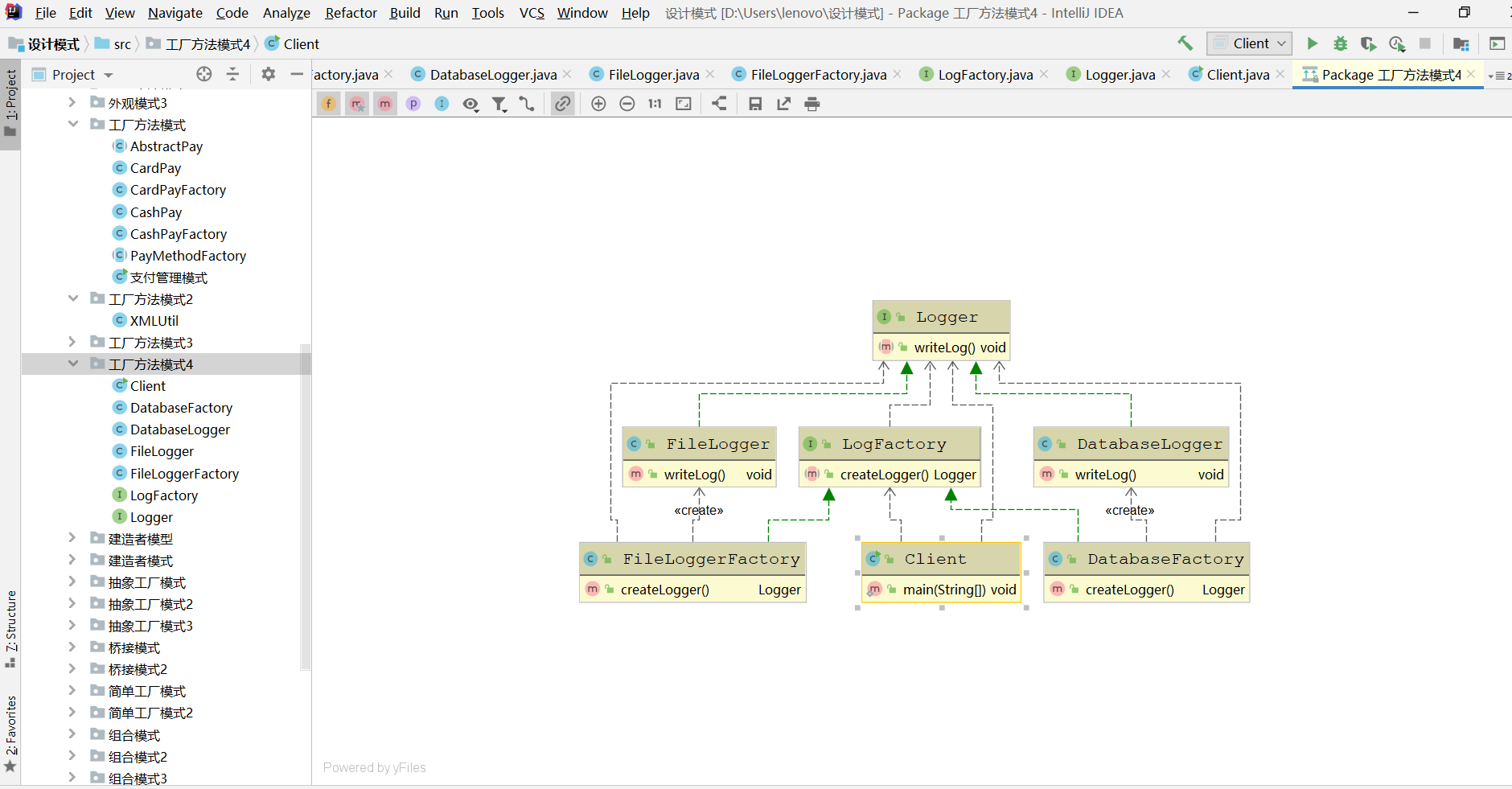
学 号： 1817700302

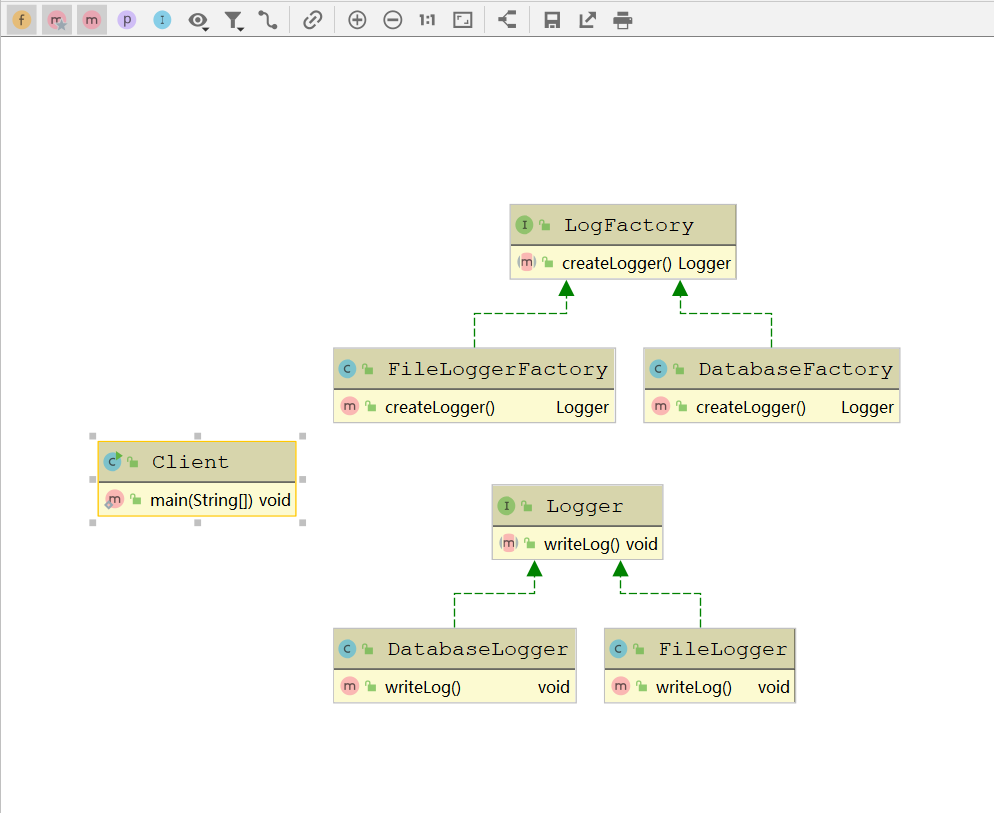
2020 年 4 月 10 日

1. 实验内容

某系统日志记录器要求支持多种日志记录方式，如文件记录、数据库等，且用户可以根据要求动态选择日志记录方式，现使用工厂方法模式涉及该系统。

1. 实验设计类图(UML)





1. 实验代码
2. 抽象日志工厂类代码实现：

**package** 工厂方法模式4;

*// 抽象工厂*

**public interface** LogFactory {  
 *//抽象工厂方法* **public** Logger createLogger();  
 }

1. 抽象日志产品类代码实现：

**package** 工厂方法模式4;  
*//抽象产品***public interface** Logger {  
 **public void** writeLog();  
}

1. 数据库日志具体工厂类代码实现：

**package** 工厂方法模式4;  
*// 具体工厂***public class** DatabaseFactory **implements** LogFactory {  
 @Override  
 **public** Logger createLogger() {System.***out***.println(**"数据库日志工厂记录数据库日志"**);  
 *//创建数据库日志记录器对象  
 //初始化数据库日志记录器,代码省略* **return new** DatabaseLogger();  
 }  
}

1. 文件日志具体工厂类代码实现：

**package** 工厂方法模式4;  
*//具体工厂* **public class** FileLoggerFactory **implements** LogFactory {  
 @Override  
 **public** Logger createLogger() {  
 System.***out***.println(**"文件日志工厂记录文件日志"**);  
 **return new** FileLogger();  
 }  
}

1. 数据库日志具体产品类代码实现：

**package** 工厂方法模式4;

*//具体产品*

**public class** DatabaseLogger **implements** Logger {  
 @Override  
 **public void** writeLog() {  
 System.***out***.println(**"数据库日志记录"**);  
 }  
}

1. 文件日志具体产品类代码实现：

**package** 工厂方法模式4;

*//具体产品* **public class** FileLogger **implements** Logger {  
 @Override  
 **public void** writeLog() {  
 System.***out***.println(**"文件日志记录"**);  
 }  
}

1. 客户端类检测代码：

**package** 工厂方法模式4;  
**import** 工厂方法模式2.XMLUtil;**public class** Client{

**public static void** main(String[] args) {  
 LogFactory factory;  
 Logger logger;

*// factory = new FileLoggerFactory();  
// logger = factory.createLogger();* factory = (LogFactory) XMLUtil.*getBean*();  
 logger =factory.createLogger();  
 logger.writeLog();  
 }  
}

1. XMLUtil工具类代码：

**package** 工厂方法模式2;  
  
  
**import** org.w3c.dom.Document;  
**import** org.w3c.dom.Node;  
**import** org.w3c.dom.NodeList;  
**import** javax.xml.parsers.DocumentBuilder;  
**import** javax.xml.parsers.DocumentBuilderFactory;  
**import** java.io.File;  
  
**public class** XMLUtil {  
 **public static** Object getBean() {  
 **try** {  
 *//创建DOM文档对象* DocumentBuilderFactory dfactory = DocumentBuilderFactory.*newInstance*();  
 DocumentBuilder builder = dfactory.newDocumentBuilder();  
 Document doc;  
 doc = builder.parse(**new** File(**"src/design/config.xml"**));  
  
 *//获取包含类名的文本节点* NodeList nl = doc.getElementsByTagName(**"className"**);  
 Node classNode = nl.item(0).getFirstChild();  
 String cName = classNode.getNodeValue();  
  
 *// 通过类名生成实例对象并将其返回* Class c = Class.*forName*(cName);  
 Object obj = c.newInstance();  
 **return** obj;  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 **return null**;  
 }  
 }  
}

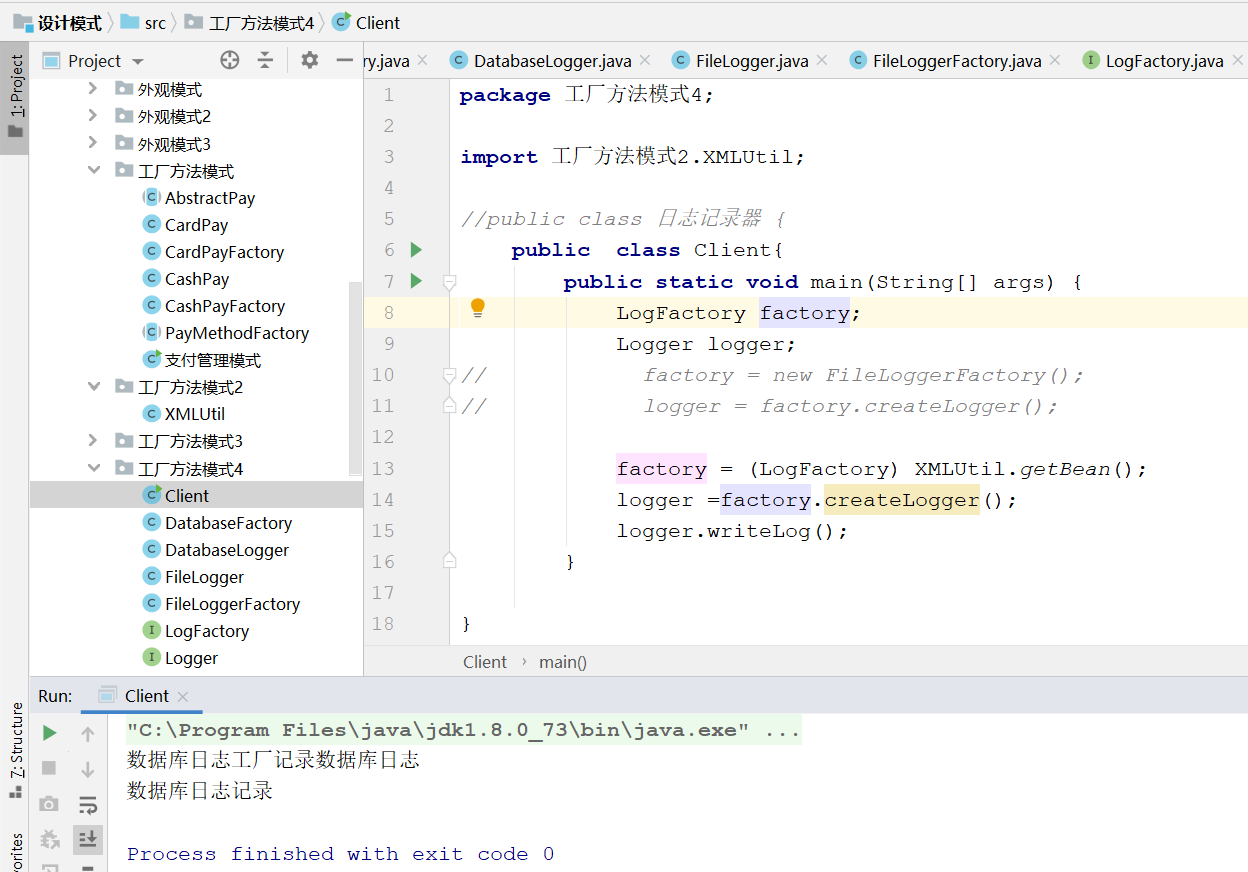
1. Config.xml文件配置：

通过XMLUtil.getBean()方法直接创建并实例化相应所需对象。减少代码冗余。

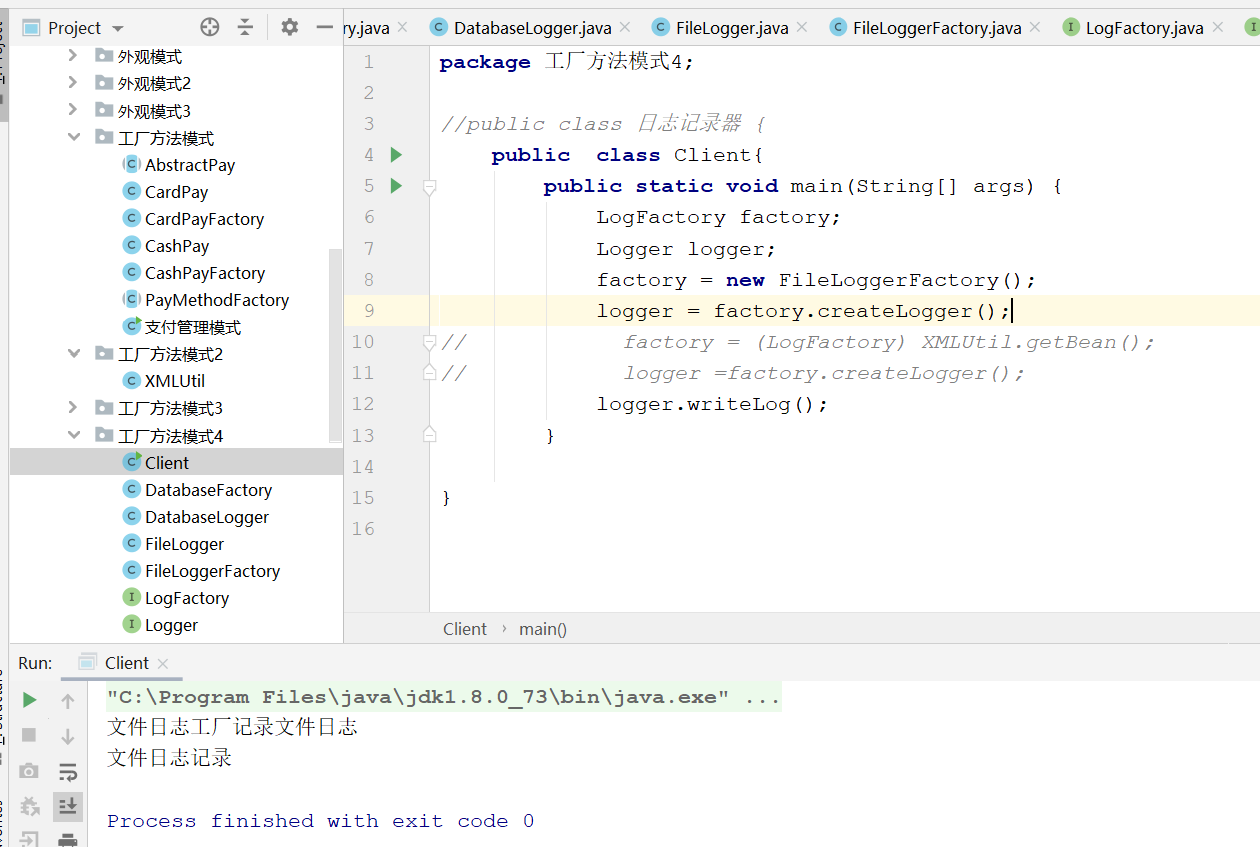
*<?***xml version="1.0"***?>*<**config**>  
<**className**>工厂方法模式4.DatabaseFactory</**className**>  
</**config**>

1. 实验结果

运行结果截图：



同样，也可以直接通过new FileLoggerFactory()，再调用其createLogger()方法创建对象并实例化。也可以配置在config.xml文件中，结果相同，展示结果如下所示：



1. 实验分析与总结

工厂方法模式(Factory Method Pattern)又称为工厂模式，也叫虚拟构造器(Virtual Constructor)模式或者多态工厂(Polymorphic Factory)模式，它属于类创建型模式。在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。

工厂方法模式包含四个角色，分别是(Product)抽象产品角色、(ConcreteProduct)具体产品角色、(Factory)抽象工厂角色和(ConcreteFactory)具体工厂。

工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了面向对象的多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责哪一个产品类被实例化这种细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品，很好的符合了“开闭原则”。

**工厂方法模式的优点**：

①用户只需要关心所需产品对应的工厂，无须关心创建细节，甚至无须知道具体产品类的类名；

②工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部；

③系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了。这样，系统的可扩展性也就变得非常好，完全符合“开闭原则”。

**工厂方法模式的缺点**：

①在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销；

②由于考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度，且在实现时可能需要用到DOM、反射等技术，增加了系统的实现难度。

**工厂方法模式的使用环境：**

①一个类不知道它所需要的对象的类：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，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类；

②一个类通过其子类来指定创建哪个对象：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。

③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