A close up of a logo

Description automatically generated

Student: Mitrica Livia-Maria

Group: 30424

Techer Assistant: Chifu Viorica

Assignment 3  
Order management

Contents

[Objective 2](#_Toc37757672)

[The main purpose of this application is processing customer orders for a warehouse. In order to store the products, the clients and the orders relational databases need to be used. 2](#_Toc37757673)

[The application should allow the user to insert new clients, products and orders using a txt file given as command line input and receive afterwards pdf reports. 2](#_Toc37757674)

[Problem analysis, modelling, scenarios, use cases 2](#_Toc37757675)

[Problem design 3](#_Toc37757676)

[Implementation 6](#_Toc37757677)

[Conclusions 16](#_Toc37757682)

[Bibliography 16](#_Toc37757683)

# Objective

# The main purpose of this application is processing customer orders for a warehouse. In order to store the products, the clients and the orders relational databases need to be used.

# The application should allow the user to insert new clients, products and orders using a txt file given as command line input and receive afterwards pdf reports.

The application should use minimally the following classes

•Model classes -the data models of the application

•Business Logic classes –implement the application logic

•Presentation classes –implement the user input/output

•Data access classes -implement the access to the database

# Problem analysis, modelling, scenarios, use cases

A relational database is a type of database that stores and provides access to data points that are related to one another. Relational databases are based on the relational model, an intuitive, straightforward way of representing data in tables.

I will use MySql database in order fulfill the requested tasks.

The project needs to provide the following functionality:

* Insert product
* Insert client
* Order
* Delete product
* Delete client
* Report order
* Report client
* Report product

A close up of a logo

Description automatically generatedUse case scenario

Use case scenarios

Insert client

In case the client exists in the database, it will not be inserted again, otherwise it will be added to the clients’ table

Insert product

In case the product exists in database, the stock (quantity) will be updated, keeping the old price, otherwise it will be inserted as a new item

Delete client

If client exists in database it will deleted and all the orders associated to it too Delete product

If product exists in database it will be deleted, otherwise no action

Order

If client and the product exist in database and the quantity requested is available, the order will be placed and a pdf bill will be generated. In case of insufficient stock, a pdf will be generated specifying this. In case the product or client does not exist, a console message will be displayed.

Report client/order/product

A pdf containing all the clients/products/orders in the database arranged in a table will be generated.

# Problem design

I have decided to divide the solution for this problem in 7 packages, which are the following model, dataBaseAccess, bll, launcher, validator, presentation and connection. They are illustrated in the picture below. Further on I will detail each package.

* model

This package contains 3 classes, Client, product and Order representing the data models of the application.

* connection

This package is composed of only one class, which is responsible for communicating with the database.

* dataBaseAccess

This package is responsible for implementing access queries for the database. It contains an abstract class, AbstractDAO and classes for each model which extend the abstract class.

* bll

This package is used for implementing the logic in the project. It contains an abstract class from which we extend classes according to our model and implement specific methods.

* Validator

This package is meant for validating inserted field in the product table.

* Presentation

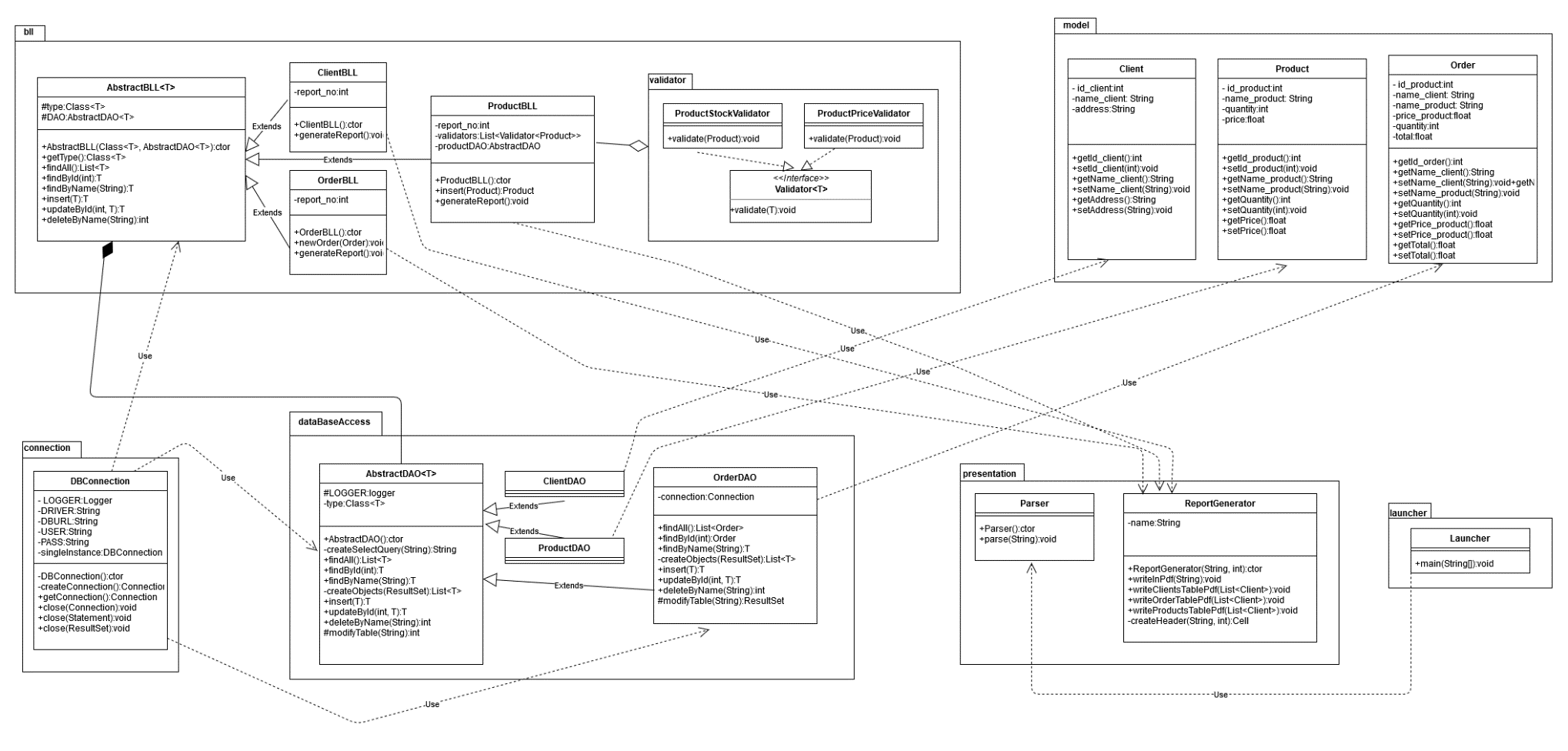
This package is used for “interacting” with the user. The classes that it has are meant for reading and writing files.

* launcher

This package contains only one class, the class designed for starting the application.

Although the laucher is used for starting the application the Parser in the presentation is responsible for deciding the opertions to be performed according to the given commands.

The project is structured using a layered architecture, as it is described in the following picture.



# Implementation

In this part I will describe the classes according to their packages, their fields and important methods.

A screenshot of a cell phone

Description automatically generatedModel

Client

* class used for representing a client
* a client should have an id, a name and an address

A screenshot of a cell phone

Description automatically generated

Product

* class used for representing a product
* a product should have id, name, price and quantity

A screenshot of a cell phone

Description automatically generated

Order

* class used for representing an order
* an order should have an id, the name of the client, the name of the product, the price of the product, the quantity desired by the client and the total price a client would have to pay

A screenshot of a cell phone

Description automatically generated

Connection

DBConnection

* class used to get a connection with the database
* singleton design pattern should be used
* attributes: static final Strings containing the information about the database needed in order to connect to it
* methods include constructor, method for creating the connection to the database and methods for closing Connection, Statement and ResultSet

A screenshot of a cell phone

Description automatically generateddataBaseAccess

AbstractDAO

* generic class used to access the database for an object
* abstract class
* contains methods for adding, deleting, updating objects
* also contains methods for getting all existing objects, as well as objects which respect some filtering condition
* the methods are implemented using java reflection, supposing there is 1-1 mapping between an object fields and a table in a database

A screenshot of a cell phone

Description automatically generatedClientDAO

* extends AbstractDAO, no overridden methods

A screenshot of a cell phone

Description automatically generated

Product DAO

* extends AbstractDAO, no overridden methods

A screenshot of a cell phone

Description automatically generated

OrderDAO

* extends AbstractDAO
* overrides 3 methods, because in my table information about orders is split in two tables

@Override  
public List<Order> findAll() throws SQLException {  
 List<Order> listOrders = new ArrayList<>();  
 connection = DBConnection.*getConnection*();  
 PreparedStatement statement = connection.prepareStatement("SELECT \* FROM `order`");  
 ResultSet rs = statement.executeQuery();  
 try {  
 while (rs.next()){  
 Order order = new Order();  
 ClientBLL clientBLL = new ClientBLL();  
 Client client = clientBLL.findById(rs.getInt("id\_client"));  
 ProductBLL productBLL = new ProductBLL();  
 Product product = productBLL.findById(rs.getInt("id\_product"));  
  
 order.setId\_order(rs.getInt("id\_order"));  
 order.setName\_client(client.getName\_client());  
 order.setName\_product(product.getName\_product());  
 order.setPrice\_product(product.getPrice());  
 order.setTotal(rs.getFloat("total"));  
  
 PreparedStatement statement\_ = connection.prepareStatement("SELECT quantity FROM `order\_details` WHERE id\_order="+order.getId\_order());  
 ResultSet RS = statement\_.executeQuery();  
  
 if(RS.next())  
 order.setQuantity(RS.getInt("quantity"));  
  
 listOrders.add(order);  
 }  
 return listOrders;  
 } catch (SQLException e) {  
 e.printStackTrace();  
 } finally {  
 DBConnection.*close*(connection);  
 }  
 return null;  
}

@Override  
public Order findById(int id) {  
 Order order = new Order(null, null, 0, 0);  
 order.setId\_order(id);  
  
 try {  
 if (connection.isClosed()) connection = DBConnection.*getConnection*();  
 String query = "SELECT \* FROM order WHERE id\_order=" + id;  
  
 ResultSet rs = modify(query);  
 while (rs.next()) {  
 int idCustomer = rs.getInt("id\_client");  
 int idProduct = rs.getInt("id\_product");  
 order.setName\_client(new ClientDAO().findById(idCustomer).getName\_client());  
 order.setName\_product(new ProductDAO().findById(idProduct).getName\_product());  
 }  
  
 rs = modify("SELECT \* FROM order\_details where id\_order=" + id);  
 rs.next();  
 order.setQuantity(rs.getInt("quantity"));  
 return order;  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return null;  
}

public Order insert(Order order) {  
 try {  
 Connection connection = null;  
 ResultSet rs = null;  
 ClientBLL clientBLL = new ClientBLL();  
 Client currentClient = clientBLL.findByName(order.getName\_client());  
  
 ProductBLL productBLL = new ProductBLL();  
 Product currentProduct = productBLL.findByName(order.getName\_Product());  
 float total = currentProduct.getPrice()\*order.getQuantity();  
  
 String query1 = "INSERT INTO `order` (id\_client, id\_product, total) VALUES ('"+currentClient.getId\_client()+"', '"+currentProduct.getId\_product()+"', '"+total+"')";  
 rs = modify(query1);  
  
 if(rs.next()) {  
  
 order.setId\_order(rs.getInt(1));  
 order.setTotal(total);  
 order.setPrice\_product(currentProduct.getPrice());  
 String query2 = "INSERT INTO `order\_details` (id\_order, quantity) VALUES ('" + order.getId\_order() + "', '" + order.getQuantity() + "');";  
 modifyTable(query2);  
  
 }  
  
 } catch (Exception e) {  
 System.*out*.println("Cannot insert order.\n");  
 e.printStackTrace();  
 }  
 return order;  
}

A screenshot of a cell phone

Description automatically generatedbll

AbstractBLL

* generic class used to represent the business logic for the application
* provides methods which directly go to the data access level

A picture containing bird

Description automatically generated

ClientBLL

* extends AbstractBLL
* adds a new method for generating the clients’ report

public void genearteReport() throws Exception {  
 ClientBLL clientBLL= new ClientBLL();  
 List<Client> clients = new ArrayList<>();  
 clients = clientBLL.findAll();  
 ReportGenerator reportGenerator = new ReportGenerator("ReportClients", *report\_no*);  
 *report\_no* ++;  
 reportGenerator.writeClientsTablePdf(clients);  
}

A screenshot of a cell phone

Description automatically generated

ProductBLL

* extends AbstractBLL
* adds a new method for generating the products’ report
* overrides insert method because if the user tries to insert an existing product, the stock of that product should be updated, not inserted as a duplicate

public Product insert(Product product) {  
 for (Validator<Product> v : validators) {  
 v.validate(product);  
 }  
 ProductDAO productDAO = new ProductDAO();  
 Product productInDB = null;  
  
 productInDB = productDAO.findByName(product.getName\_product());  
  
 //update product because it already exists in the table  
 if( productInDB != null) {  
 productInDB.setQuantity( productInDB.getQuantity()+product.getQuantity());  
 System.*out*.println(productInDB.toString());  
 return productDAO.updateById(productInDB.getId\_product(), productInDB);  
 }  
  
 // otherwise we must insert new product  
 return productDAO.insert(product);  
}

public void genearteReport() throws Exception {  
 ProductBLL productBLL= new ProductBLL();  
 List<Product> products = new ArrayList<>();  
 products = productBLL.findAll();  
 ReportGenerator reportGenerator = new ReportGenerator("reportProducts", *report\_no*);  
 *report\_no*++;  
 reportGenerator.writeProductsTablePdf(products);  
}

A screenshot of a cell phone

Description automatically generated

OrderBLL

* extends AbstractBLL
* adds a new method for generating the orders’ report
* adds a new method for inserting a new order, because this process requires inserting in 2 tables and updating the quantity of the product if the order can be processed

public Order newOrder(Order order) throws Exception {  
 ClientBLL clientBLL = new ClientBLL();  
 Client currentClient = clientBLL.findByName(order.getName\_client());  
  
 ProductBLL productBLL = new ProductBLL();  
 Product currentDBProduct = productBLL.findByName(order.getName\_Product());//get the product from database  
  
 if(currentDBProduct!=null && currentClient!=null) {  
  
 if( currentDBProduct.getQuantity() >= order.getQuantity() ) {  
 OrderDAO orderDAO = new OrderDAO();  
 orderDAO.insert(order);//add the order in the database  
 currentDBProduct.setQuantity(currentDBProduct.getQuantity() - order.getQuantity());//set product new quantity  
 productBLL.updateById(currentDBProduct.getId\_product(), currentDBProduct);//update product new quantity  
  
 ReportGenerator file = new ReportGenerator("bill", order.getId\_order());//file.writeInPDF(order.toString());  
 List<Order> orders = new ArrayList<Order>();  
 orders.add(order);  
 file.writeOrderTablePdf(orders);  
 }  
 else{  
 ReportGenerator file = new ReportGenerator("bill", 110);  
 file.writeInPDF("Cannot process the following order:\n\n"+ order.toString()+"\nInsufficient stock.");  
 }  
 }  
 return order;  
}

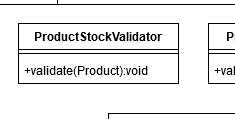
public void genearteReport() throws Exception {  
 OrderBLL orderBLL= new OrderBLL();  
 List<Order> orders = new ArrayList<>();  
 orders = orderBLL.findAll();  
 ReportGenerator reportGenerator = new ReportGenerator("reportOrders", *report\_no*);  
 *report\_no*++;  
  
 reportGenerator.writeOrderTablePdf(orders);  
}

A screenshot of a cell phone

Description automatically generated

Validator

Product StockValidator

* implements method in Validate interface
* checks if price is a positive number, in case it is not it throws an exception

ProductPriceValidator

* implements method in Validate interface
* checks if stock is a positive number, in case it is not it throws an exception

A screenshot of a cell phone

Description automatically generatedPresentation

Parser

* contains several methods for dealing with the input commands
* parse() method receives a file and reads its contents, according to the user command one of the functions insert\_command, report\_command, delete\_command or order\_command is called

A screenshot of a cell phone

Description automatically generatedReportGenerator

* attribute name represents the name of the generated pdf
* constructor receives a string with the actual name, which can be bill, report client, report order or report product and a number, which is the id of the bill/report
* different methods for models exist, since the table is creating according to the particular fields
* method createHeader returns the header of the table, receives as parameters the title an the number of columns to span

A screenshot of a cell phone

Description automatically generatedLauncher

* contains only one class, the main class which starts the application

I have used the following MySQL tables:

A screenshot of a cell phone

Description automatically generated

Regarding bill generation, I have decided to generate a new bill for each order as specified in the assignment. Altough grouping consecutive orders by client is a possibility, I have not considered that. In real life, even if a client might place two orders one after another, he might choose for example different delivery addresses or different methods to pay. This is the reason why I have chosen to implement my program this way.

A screenshot of a social media post

Description automatically generated

# A picture containing screenshot Description automatically generated

# A screenshot of a cell phone Description automatically generated

# A screenshot of a social media post Description automatically generatedA screenshot of a cell phone Description automatically generated

# A screenshot of a computer screen Description automatically generated

A screenshot of a cell phone

Description automatically generated

# Conclusions

This assignment has tought me how to work with reflection techniques. Moreover, I was able to get more familiar with database access from java and how to make java doc comments. Apart from this I have learn how to generate pdf files.

First the project could be improved by adding a graphical interface. Moreover, another improvement would be letting the client to order multiple items. Also it could generate reports of all the orders made by client given as input ( Report orders X). Also more information about clients/ products/ orders could be introduced, like phone number, email address, product manufacturer, delivery method etc.

# Bibliography

1. https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html
2. https://www.javatpoint.com/example-to-connect-to-the-mysql-database
3. https://www.jrebel.com/blog/sql-cheat-sheet
4. https://www.geeksforgeeks.org/reflection-in-java/
5. https://itextpdf.com/en/resources/examples/itext-7/cell-and-table-widths
6. https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html
7. https://bitbucket.org/utcn\_dsrl/pt-reflection-example.git
8. https://bitbucket.org/utcn\_dsrl/pt-layered-architecture.git
9. Lecture materials