DOCUMENTATION

ASSIGNMENT 3 – ORDERS MANAGEMENT APPLICATION

NUME STUDENT: Bumbuc Ioana

GRUPA: 30421

# CONTENTS

[1. Assignment’s objective 3](#_Toc95297885)

[*2.* Problem analysis, modelling, scenarios, use-case*s* 3](#_Toc95297886)

[3. Design 7](#_Toc95297887)

[4. Implementation 9](#_Toc95297888)

[5. Results 15](#_Toc95297889)

[6. Conclusions 15](#_Toc95297890)

[7. Bibliography 16](#_Toc95297891)

# Assignment’s Objective

The objective of this assignment is designing and implementing an application that manages orders in a warehouse. It should create a mechanism for the user to interact with data defining clients, products, and orders. The application should be able to perform operations on each set of data, that is inserting, deleting, viewing, and updating entries in the corresponding tables.

The sub-objectives of this assignment are the following:

* Analyzing the problem and identifying its requirements (detailed in chapter 2)
* Designing the orders management application (chapter 3)
* Implementing the orders management application (chapter 4)
* Testing the orders management application (chapter 5)

# Problem analysis, modelling, scenarios, use-cases

This application wishes to solve the problem created by managing orders using handwritten registers. The idea is to implement an application that manages products, clients, and orders in the form of tables stored in a database, that the user can modify through the user application.

* 1. **Functional Requirements**

The application must perform operations on three types of data: client, product, order. For each one, the user must be able to visualize the table containing all objects. The app must also implement operations to modify existing data: to add a new object into the database, to delete an existing object or to update an existing object by modifying the data already existent.

* 1. **Non-Functional Requirements**

Non-functional requirements refer to how the application behaves and what limits there are on its functionality. Thus, the application should be intuitive, easy to use by the user, while the output of the simulation must be concise and relevant.

* 1. **Use Cases**

****

**2.3.1. Client operations use cases**

* + - **Add Client**

Primary actor: Employee

Main success scenario:

1. Employee selects “Client” table to perform operations
2. Employee inserts client data into the fields: id, name, email, phone number
3. Clicks “Add Client” button
4. Application stores new client in the data base.

Alternative sequence: invalid values for client data (i.e., Id that already exists, invalid e-mail address or phone number). In this case, an exception appears, and the scenario returns to step 2.

* **View Clients**

Primary actor: Employee

Main success scenario:

1. Employee selects “Client” table
2. Employee clicks on “View Clients” button
3. A table appears with all clients in the table.

* **Update Client**

Primary actor: Employee

Main success scenario:

1. Employee selects “Client” table
2. Employee enters the id of the existing client they want to update
3. Employee enters name, phone number, email into the fields
4. Employee clicks on “Update Client” button
5. The entry is modified inside the database

Alternative scenario: The id of the client that should be updated doesn’t already exist in the database. An exception appears and the scenario returns to step 2.

* **Delete Client**

Primary actor: Employee

Main success scenario:

1. Employee selects “Client” table
2. Employee enters the id of the existing client they want to delete
3. Employee clicks on “Delete Client” button
4. The entry is deleted from the database

Alternative scenario: The id of the client that should be deleted doesn’t already exist in the database. An exception appears and the scenario returns to step 2.

**2.3.2. Product operations use cases**

* + - **Add Product**

Primary actor: Employee

Main success scenario:

1. Employee selects “Product” table to perform operations
2. Employee inserts product data into the fields: id, name, stock, price
3. Clicks “Add Product” button
4. Application stores new product in the data base.

Alternative sequence: invalid values for product data (i.e., Id that already exists, negative values for stock or price). In this case, an exception appears, and the scenario returns to step 2.

* **View Products**

Primary actor: Employee

Main success scenario:

1. Employee selects “Products” table
2. Employee clicks on “View Products” button
3. A table appears with all products in the table.

* **Update Product**

Primary actor: Employee

Main success scenario:

1. Employee selects “Product” table
2. Employee enters the id of the existing product they want to update
3. Employee enters name, price, and stock into existing fields
4. Employee clicks on “Update Product” button
5. The entry is modified inside the database

Alternative scenario: The id of the product that should be updated doesn’t already exist in the database. An exception appears and the scenario returns to step 2.

* **Delete Product**

Primary actor: Employee

Main success scenario:

1. Employee selects “Product” table
2. Employee enters the id of the existing product they want to delete
3. Employee clicks on “Delete Product” button
4. The entry is deleted from the database

Alternative scenario: The id of the product that should be deleted doesn’t already exist in the database. An exception appears and the scenario returns to step 2.

**2.3.3. Order operations use cases**

* + - **Add Order**

Primary actor: Employee

Main success scenario:

1. Employee selects “Order” table to perform operations
2. Employee inserts client data into the fields: order id, product id, client id, quantity
3. Clicks “Add Order” button
4. Application stores new order in the data base and the stock of the corresponding product is decremented with the quantity of the order

Alternative sequence: invalid values for order data (i.e., Id that already exists, client or product id that do not exist in the data base, negative quantity). In this case, an exception appears, and the scenario returns to step 2.

Alternative sequence 2: the quantity entered for the order is larger than the amount of products in stock. In this case, an out-of-stock exception appears, and the scenario returns to step 2.

* **View Orders**

Primary actor: Employee

Main success scenario:

1. Employee selects “Orders” table
2. Employee clicks on “View Orders” button
3. A table appears with all orders in the table.

* **Update Order**

Primary actor: Employee

Main success scenario:

1. Employee selects “Orders” table
2. Employee enters the id of the existing order they want to update
3. Employee enters product id, client id and quantity
4. Employee clicks on “Update Order” button
5. The entry is modified inside the database

Alternative scenario: The id of the order that should be updated doesn’t already exist in the database. An exception appears and the scenario returns to step 2.

* **Delete Order**

Primary actor: Employee

Main success scenario:

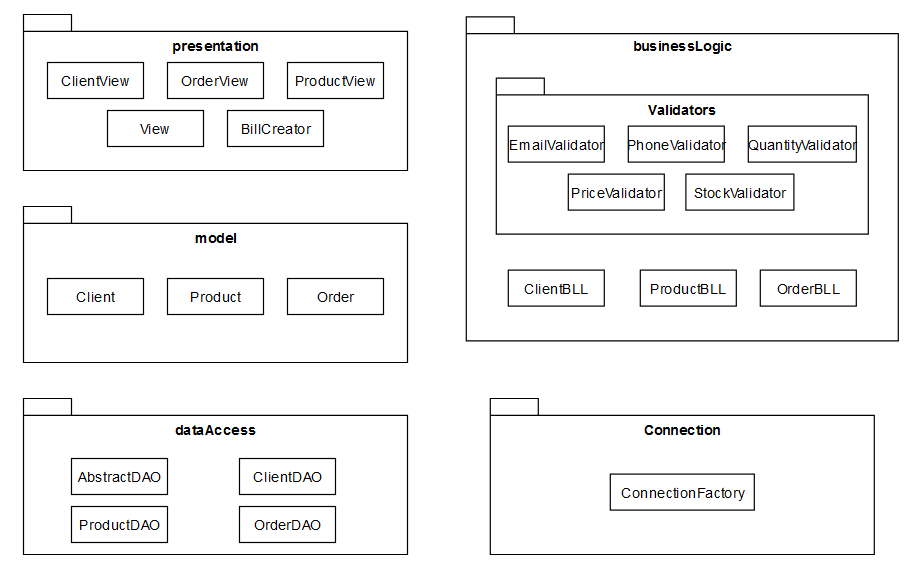
1. Employee selects “Orders” table
2. Employee enters the id of the existing order they want to delete
3. Employee clicks on “Delete Order” button
4. The entry is deleted from the database

Alternative scenario: The id of the order that should be deleted doesn’t already exist in the database. An exception appears and the scenario returns to step 2.

# Problem design

The design of this problem is realized using layered architecture. The layers are presentation layer, which contains the classes that implement the user interface, business logic layer that encapsulate the logic behind the operations, and data access layer that realizes the connection to the data base and creates queries for manipulating data inside tables.

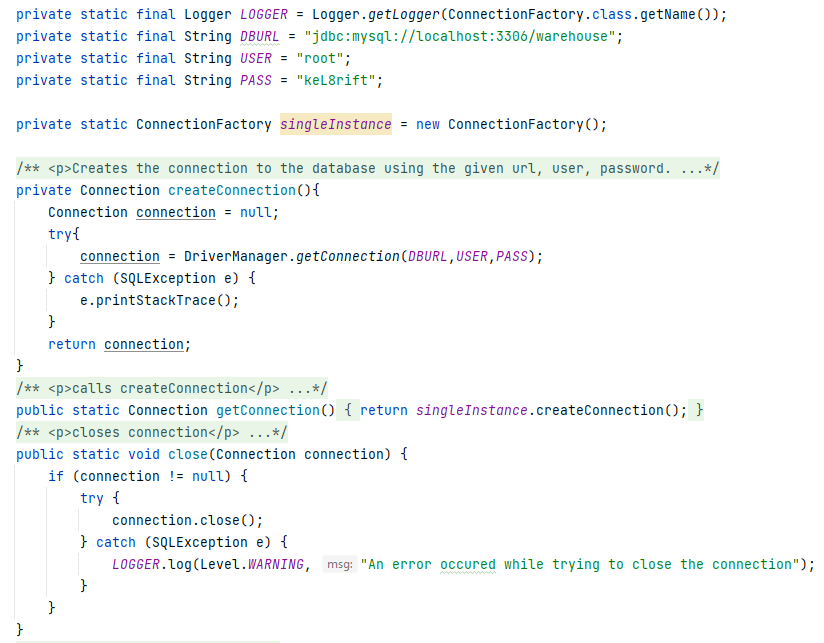
* 1. **Package Diagram**



* Presentation package – contains classes that define the user interfaces. ClientView, OrderView, ProductView are classes that implement the interface for each type of operations. BillCreator creates a txt file with order details each time an employee adds a new order to the database.
* BusinessLogic – contains a validator package with various classes used for validating sets of data entered by user. Price, quantity, and stock validators check that those fields are positive. Phone validators checks if the phone number contains only digits and is longer than 5 digits but no more than 10 digits. E-mail validator uses regex to check if the e-mail entered by the user is valid.
* Model – contains the classes for clients, products, and orders. An order contains a client and a product.
* DataAccess – contains classes that realize the interaction between the application and the data base. It has an AbstractDAO class which defines abstract methods for creating and executing queries, using reflection technique. ClientDAO, ProductDAO and OrderDAO contain methods that are specific to each type of data to manipulate.
* Connection – contains ConnectionFactory class that realizes the connection to the database.

# Implementation

* 1. **Business logic**
     1. **Connection Factory**



This class creates the connection to the database. The method createConnection() realizes the connection using the database URL, user and password defined in the class. The method close (Connection connection) closes the connection. This method is overridden and can be given as parameter a Statement or ResultSet.

* + 1. **Validators**

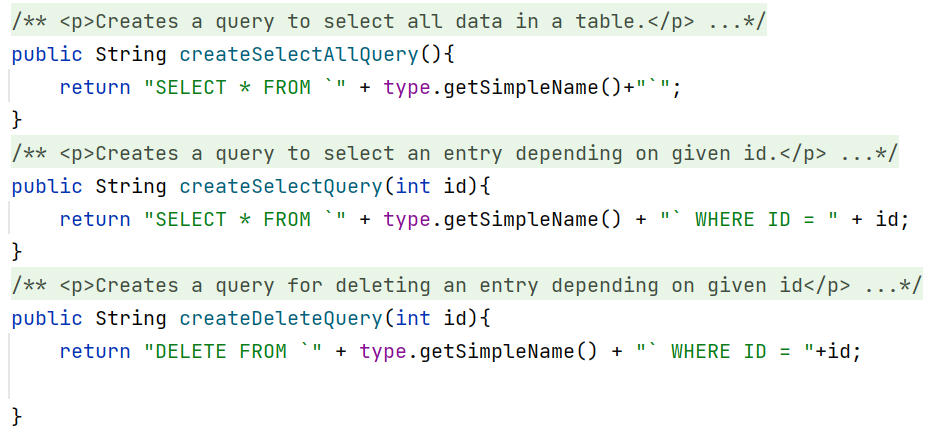
The validators package contains the Validator interface, which has the method validate (T t) on an abstract type of object. This interface is then implemented by other Validator classes, for checking e-mail address, phone number, price, stock, or quantity.

* + 1. **ClientBLL, ProductBLL, OrderBLL**

These classes implement the methods for finding, inserting, deleting, and updating. To exemplify, in ClientBLL, the methods either take an object of type Client as parameter or they return a client object. Those methods call for other methods inside the ClientDAO class.

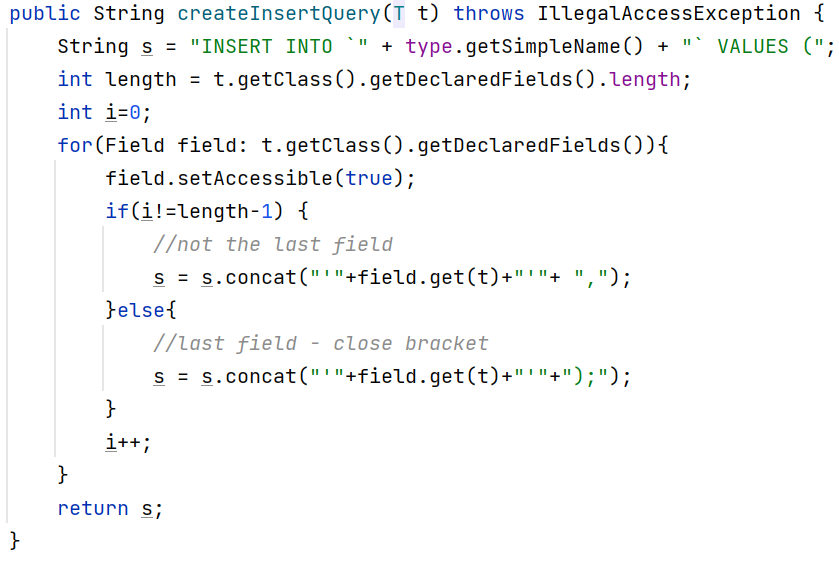
* 1. **Data access**
     1. **AbstractDAO**

AbstractDAO implement methods for an abstract object. It has methods for selecting all, selecting by ID, deleting, updating, inserting. The three methods above create a simple String where the name of the table is given by calling type.getSimpleName( ).

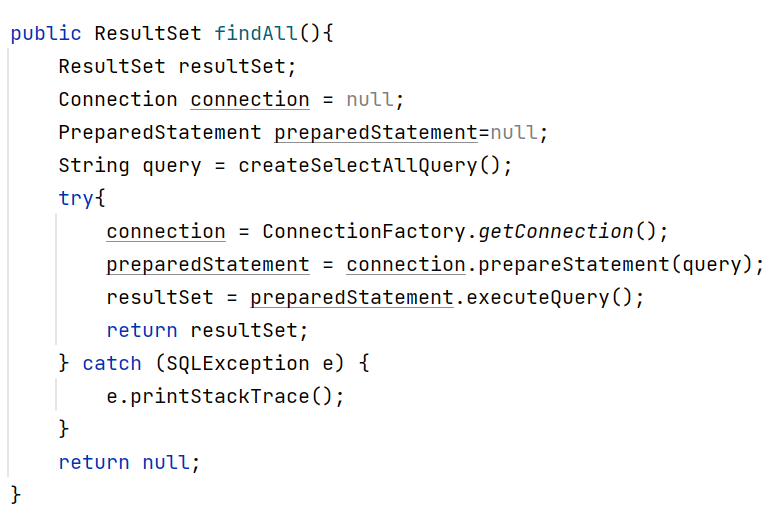


The method for creating an insert query iterates through every field of the object t. The values stored in those fields are the values we wish to insert into the table. For each field, the values stored in it is added to the string, the method returning the correct query.

The same idea is used for updating a query but using the name of the field as well as the data stored in that field.

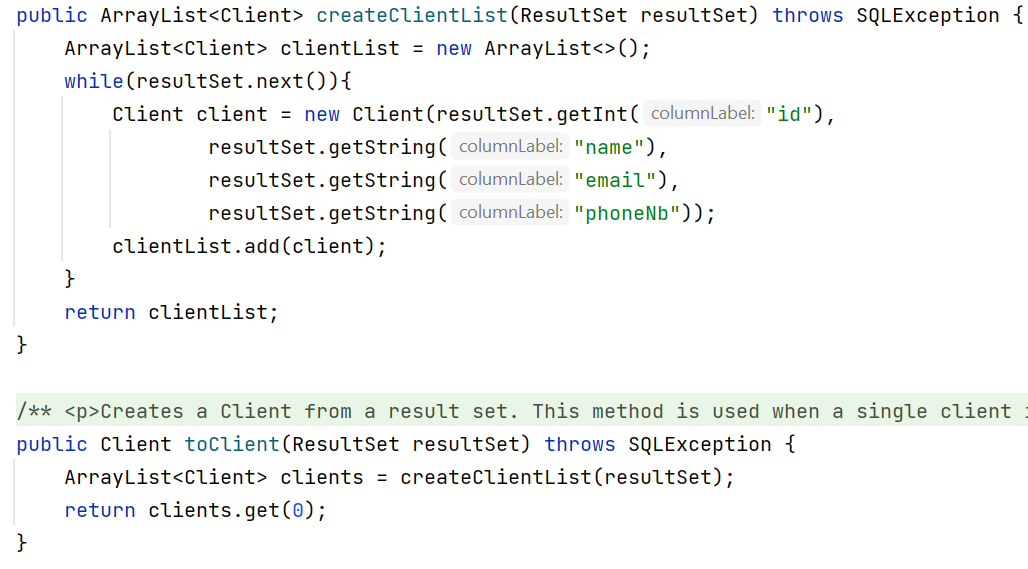


The method “find all” calls the method that creates a query and stores int the query String. Then, inside a try-catch block, the method tries to connect to the data base, creates a prepared statement, then executes that query. For other operations that do not use select queries, the execution of them is done using executeUpdate() method.



* + 1. **ClientDAO, ProductDAO**

In ClientDAO and ProductDAO I implemented methods that only apply on these data. For example, in createClientList method I take a resultSet (the result after doing a query) and I create Client objects to add to the array list of clients. The toClient method takes a resultSet and returns one single object of type Client, instead of an array list. This method is useful when performing a select by id query. ProductDAO class has the same methods but applied on objects of type Product.



* + 1. **OrderDAO**

OrderDAO contains methods that are specific only to the Order class. Besides the methods for creating an array list of orders, this class also overrides the methods for creating insert and update queries. I did this because in the Order class there are fields of type Client and Product, whereas in the database I have client\_id and product\_id columns.

There is also a method for decrementing the stock of a product. After inserting a new order, this method is called to subtract the quantity from the stock of the corresponding product.

* 1. **Model**
     1. **Client**

Client class has the following fields:

* + - id – int
    - name – String
    - email – String
    - phoneNb – String

Methods are constructors and getters.

* + 1. **Product**
  + id – int
  + name – String
  + stock – int
  + price – int

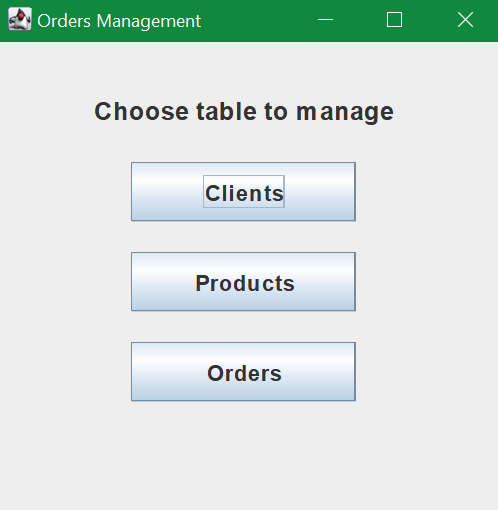
Product class also has a method for decrementing the stock with a given amount.

* + 1. **Order**
* id – int
* client – Client
* product – Product
* quantity – int

Order class contains a method that calculates the total price of an order. It does that by multiplying the order’s quantity with the price of a product.

* 1. **Presentation**
     1. **View**

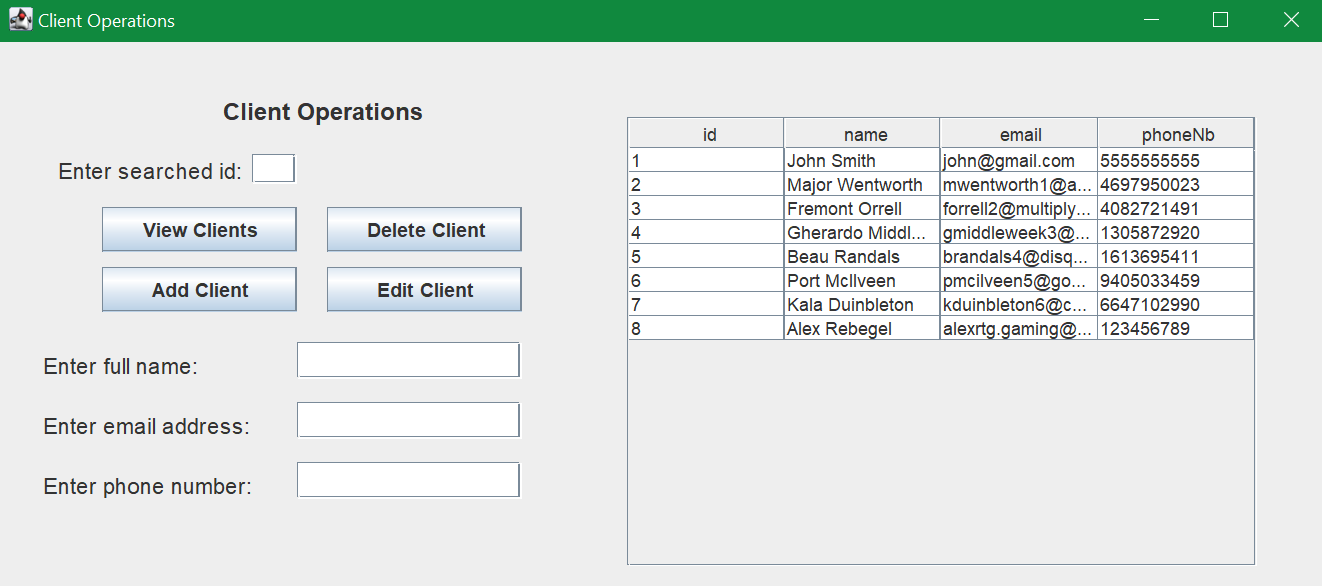
When running the application, a window appears where the user can select what operation they wish to perform. When pressing one button, the corresponding window opens for that category: client, product, order.



* + 1. **ClientView, ProductView, OrderView**

This class implements the interface for operations on the client table. There are text fields for the user to write input data for a new client to be added or for an existing client to be updated. If the data the user inserts is invalid for different reasons, an exception appears telling the user what is incorrect, and the application doesn’t stop.

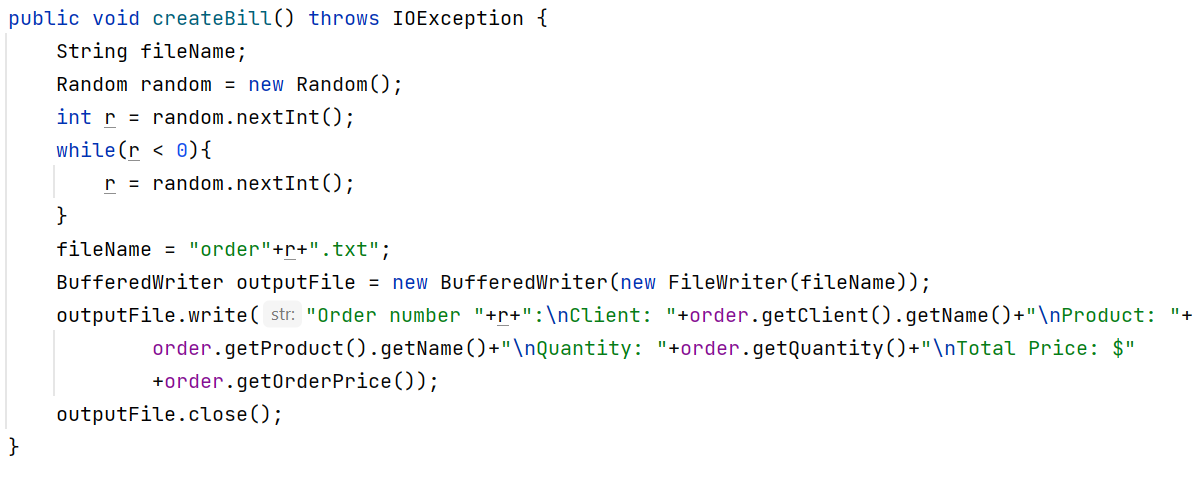
The buttons provide the functionality of the application. When pressing “View Clients” the table in the right side appears or is refreshed with the new information added/deleted by the user.



ProductView and OrderView look and function in the same manner, but for their specific objects.

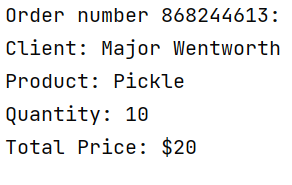
* + 1. **CreateBill**

CreateBill class contains one field of type Order. It takes the data from that object and creates a text file of the bill, containing the client, product, amount, and price.



# Results

The results of this application are the orders resulted between a client and a product. Each time an order is added to the database, a bill is created in a text file.



# Conclusions

The purpose of this application is to create an order management system that employees can efficiently use to keep track of orders inside a warehouse. The aim is to solve the problem created by handwritten registers, that are difficult to keep track of.

# Bibliography

* <https://www.baeldung.com/java-jdbc>
* <http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>
* <https://dzone.com/articles/layers-standard-enterprise>
* <http://tutorials.jenkov.com/java-reflection/index.html>
* <https://www.baeldung.com/java-pdf-creation>
* <https://www.baeldung.com/javadoc>
* <https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html>