**Order Management**



Teacher: Ioan Salomie

Teacher Assistant: Ciprian Stan

Student: Bodea Razvan-Marius

Group: English\_30422

Year: 2021

1. **Assignment Objective**

Design and implement a warehouse management system. Use relational databases and store information about Clients, Orders, Products.

Sub-Objectives:

* + Have a window for Client, Product, Order using a TableView.
  + Implement reflexively the following operations : Add, Edit, Delete, View all, Select by id.
  + A reflexive method to populate the tables and generate the headers
  + Have a layered architecture
  + Create a bill for the orders

Why create a warehouse manager?

This project simulates working with the order, clients and products of a warehouse. This kind of app could be used by a real warehouse for an easier way to work with all it’s data in a more organized and automated manner.

1. **Problem analysis, modeling, scenarios, use cases**
   1. Analysis

The Database for the warehouse has 3 tables: Client, Product, Order.

We want to create classes that reflect these tables (client, product, order) and create methods to work with this data.

We want to separate the problem in more layers:

A connection layer where we have methods for connecting and working with the database.

A Business logic layer where we have methods that call functions form the Data Access Layer.

In the Data Access Layer we ant to implement some reflexive methods to work with the database. (like selecting all clients, deleting a product etc).

We want to create a bill for each order.

For creating an order we want to be able to select an existing client, an existing product and specify a quantity to be ordered. This quantity should be subtracted from the available product stock. If there are not enough items in stock, a message should be displayed.

2.2 Modeling

I created 3 different interfaces for an easier presentation. One for each object (client, table, operation).

Each interface contains a TableView where the data from the table or the data that results from the operations is displayed.

Each interface has some buttons for different functions (add, delete, edit, select, show all).

For the order interface, I have 2 choiceBoxes where the user can select an existing client, an existing product and type a quantity to be ordered. After the button add order is pressed, the data from these fields is processed and verified in order to obtain a valid operation / error message.

2.3 Scenarios & Use Cases

The usage of this app is a bit more complex due to the fact that there are a lot of text fields and buttons and there is a lot of information on the screen.

When the app starts, the Client interface is presented. The user can freely swap between the 3 available interfaces for client, order and product.

Each interface contains buttons and text fields for the user to use.

Some examples of usages:

Switch to product/order/client buttons change the current interface to the selected one.

Show/Refresh button displays all the data in the interface table.

Remove button takes the id from the Id text field and deletes the item in the table with that id.

The add button takes the information from all the fields except the id (which is generated automatically in the database) and tries to create a new item in that table with the given information.

The update client needs an id to find the item we want to edit and will take the information from the rest of the text fields and replace the current information of the item with the new one.

The Add order button will take the selected client and product from the choice boxes (these boxes displays and lets the user choose one client/product) and the quantity and will try to create a new order with the given information.

Calculate bill button will write in a text file the bills for all the orders if no id is specified. If id is specified, the bill will be created only for the order with that id.

1. **Design (design decisions, UML Diagrams, data structures, class design, interfaces, relationship packages, algorithms, user interface)**

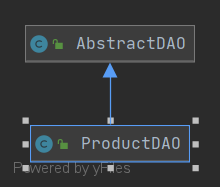
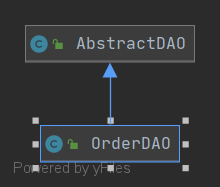
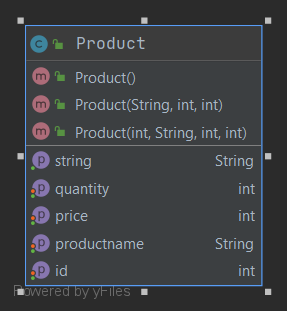
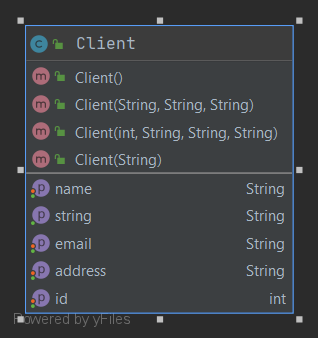
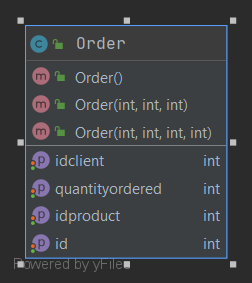
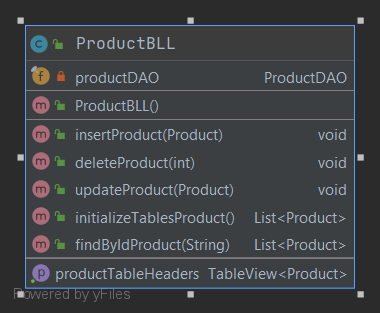
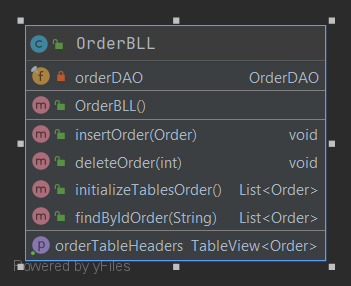
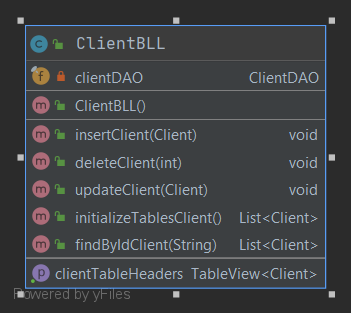
3.1 Design decisions

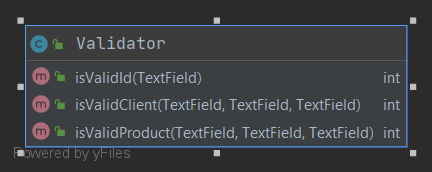
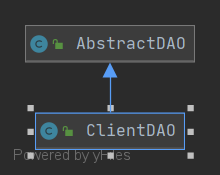
The Layered structure given in the presentation was a big inspiration for my design. I have the following layers: BLL, Business Layer, Connection, Data Access, Model, Validator, App.

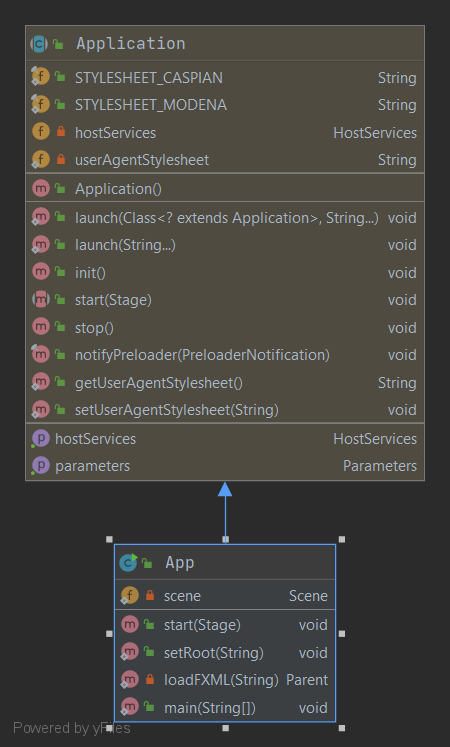
The user interface should be as easy to use as possible and have all the required functionalities.

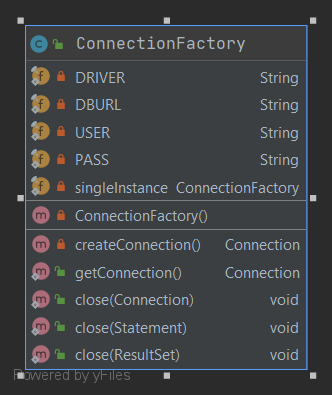
A text field is used only for displaying error logs so that the user can see what was the mistake if the selected action did not occur.

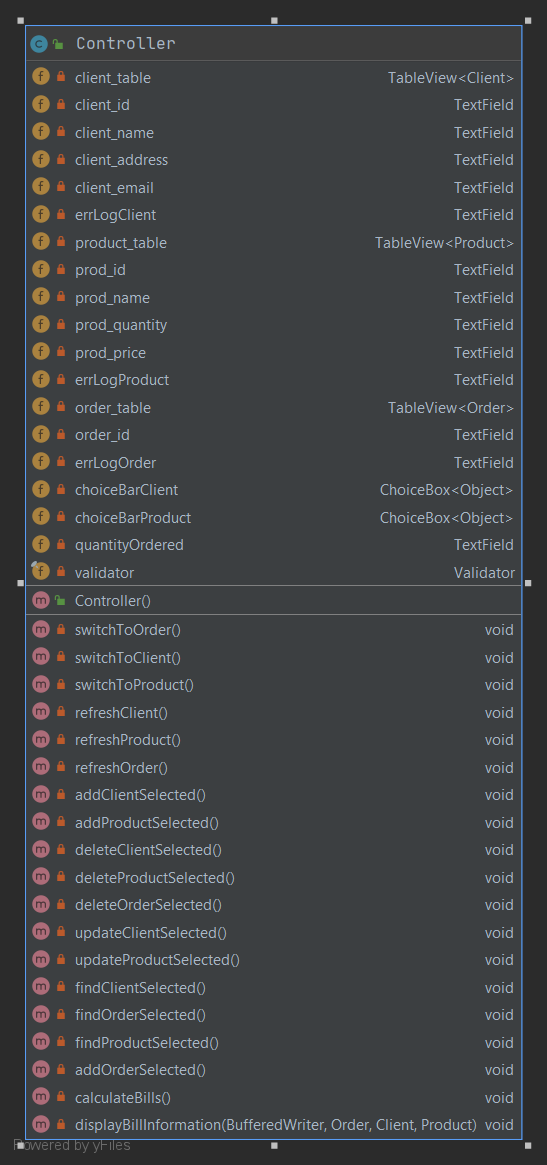
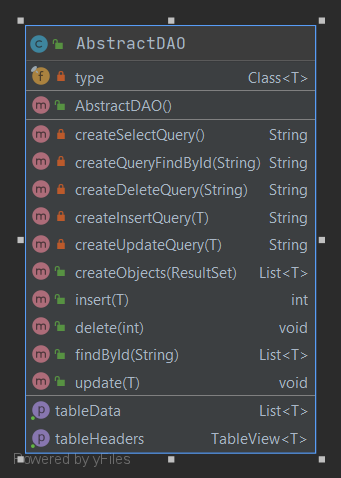
3.2 UML Diagrams & Classes











3.3 Data Structures/data types & other elements

Observable Lists are used for the objects we want to display in the Table View.

Booleans are used for validation of operations.

The table view is used for displaying the objects on the user interface. This is the main element of the user interface.

The buttons are used to select different operations.

The client, order and product classes are created to imitate the fields in the database.

StringBuilders are used for creating queries and other messages.

ResultSet is used to store the data we get form executing a query on the database.

3.4 Class Design, Interfaces, Packages

The idea I started with here:

Have Classes that are compact, well organized, that have methods that perform specific operations and are named accordingly.

I will start explaining some of the classes:

AbstractDAO :

The most important class of this project. It is used for the reflexive methods of the project.

This class works with a placeholder class named T that can be either one of the other 3 classes we want to work with (Client, Order, Product).

In this class I implemented the create query methods (add item, delete item, find item, select all items). And also the methods that use these query creating methods. (The delete client method uses the delete query etc).

Methods from this class are called through the classes that extend this main class (ClientDAO, ProductDAO, OrderDAO). The only role of these classes is to extend the AbstractDAO so that we can call through them the method on the Classes we actually want to use the functions on (Client, Product, Order).

These calls are made in the Business Level (ClientBll, OrderBll, ProductBll). These classes create an object of type DAO and use it to call the methods from AbstractDAO on the object the name represents. Thus replacing the “T” in the AbstractDAO with the class we want to use.

The connectionFactory class handles the connection/disconnection to/from the database.

The Client, Order and Product classes have the same fields as the tables in the database. Items of this type will represent an entry in the database. The tostring method creates a string representation of the objects of this type.

The Validator Class contains methods for validating the data (like having the id as a natural number >0, or no empty text fields etc).

The App Class is the “main” of the app. It starts the app and sets the default user interface.

The Controller Class is the “brain” of my app. Here I handle all the events that happen on button presses (the interaction between the user and the app). Each button press calls a different method that triggers a cascading effect on all the other methods from the other classes.

The UI elements are stored in the resources file.

Packages represent the layered architecture.

The database is the lowest layer in this architecture. Then it is followed by the database access layer, then business logic level, and on top of this hierarchy is the Controller and the User interface elements.

3.5 Algorithms

The main “algorithm“ for working with a database is creating a query based on what you need (slect \* from table etc) and using that query to get a result set that corresponds to the data obtained by using that query. With the resulset, you can go further and create objects in your program that can be further used for all the other operations.

The idea of reflexivity comes from the desire to use the same code to work with multiple sets of data/ multiple objects. If you have 3 different table, it is a lot more efficient to write a single method that can extract the data from all 3 tables instead of writing similar methods for all 3 different tables. Reflexive code can also be used on more projects that work with similar ideas.

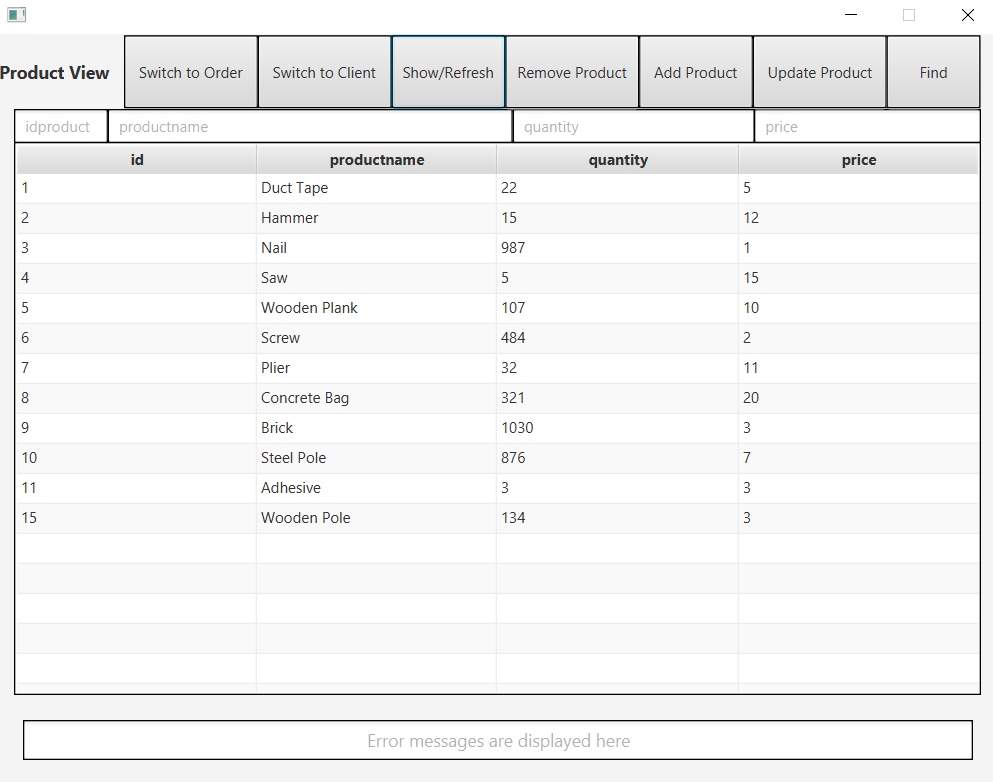
3.6 User Interface

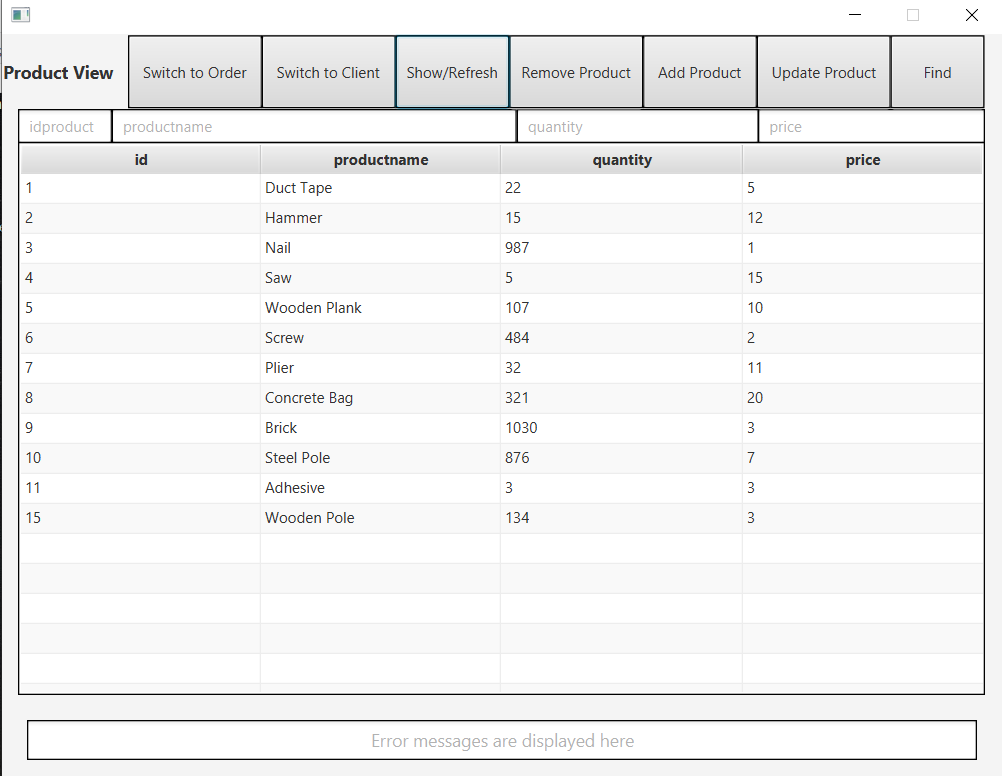
For creating the user interface, I used Scene Builder (link in Bibliography) and also some details I added with code.

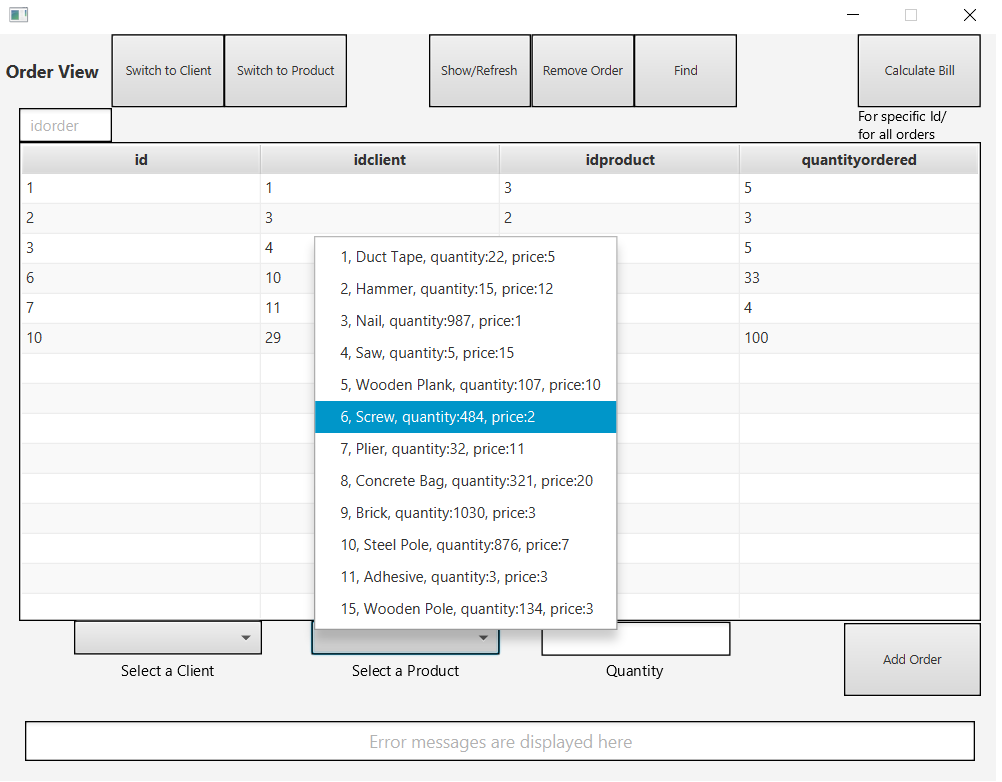
The UI is implemented with javafx.

The User Interface should be simple and accessible for all users.

The UI calls the operation to be executed when the user presses the button for the operation (dynamical).

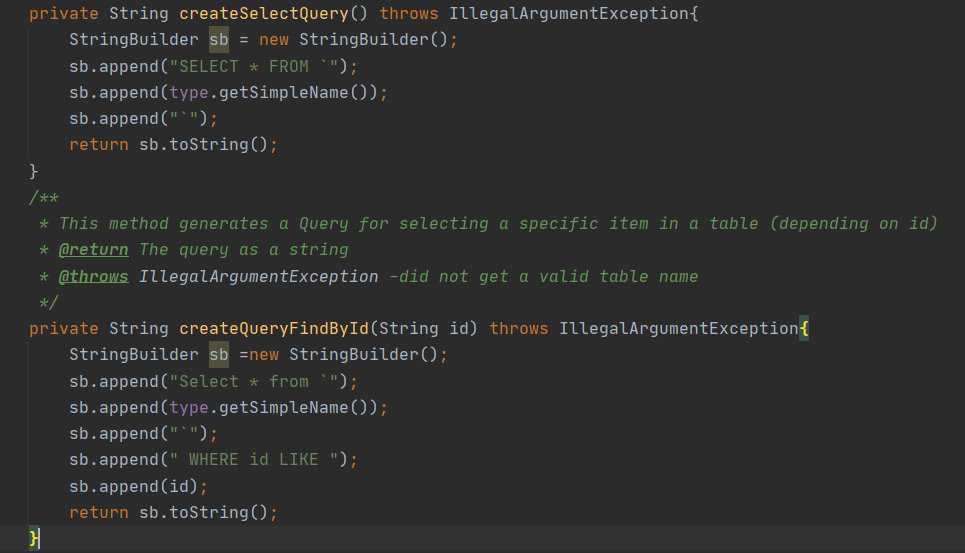
The user interface is easy to read and understand by every user. The buttons and fields are self-explanatory and intuitive. 



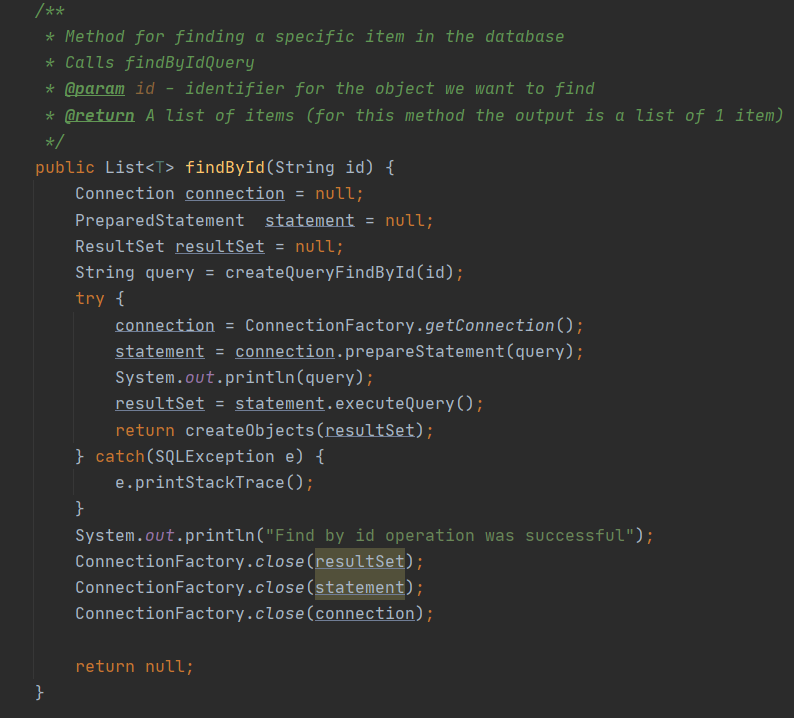


1. **Implementation**

In this section, I will talk about some methods I implemented.

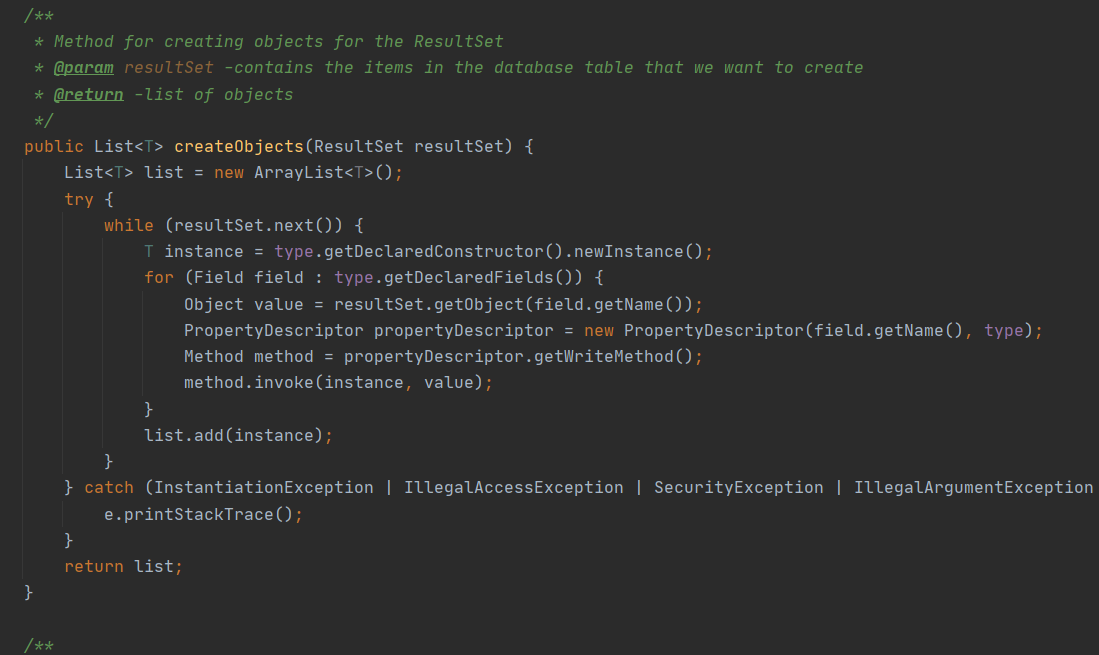


The method used for creating queries are very important in order to obtain the data we actually want form the table. The type.getSimpleName() returns the name of the table we want to select from. The id is given as a parameter in the findById method and we select the item with that id from the database.



This method is uses the selectByIdQuery to get the object we are searching for. Before any operation, we have to connect to the database and at the end we have to disconnect.

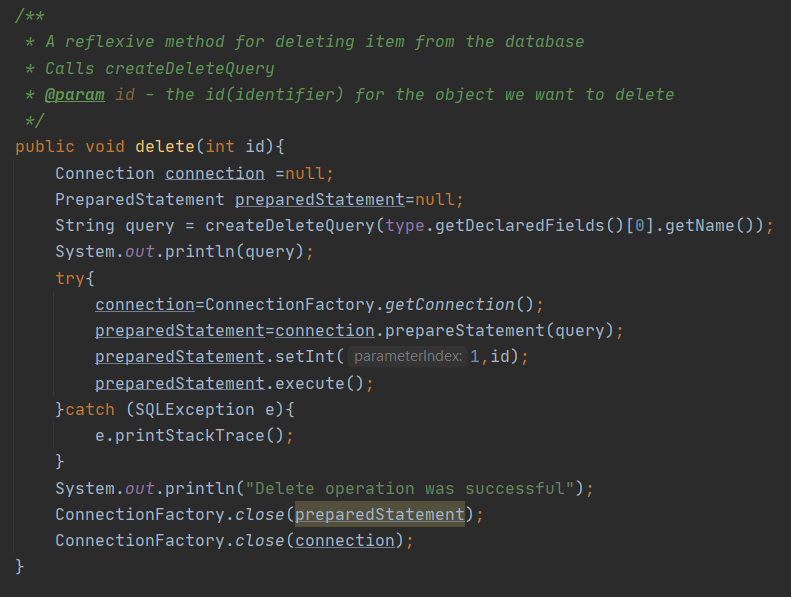
The query is viewed as a string and the .executeQuery() method executes the query we have prepared before.



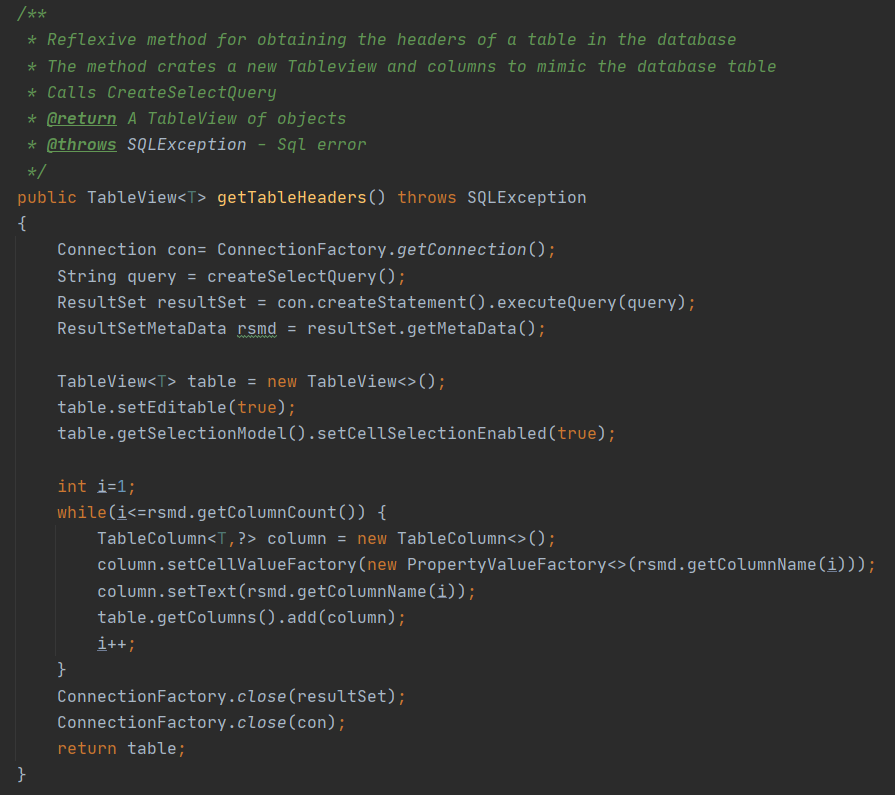
This method was given in the presentation support. It creates objects corresponding to the entries in the database and returns a list of items of that type.



In these methods we take the data form different textFields and check if the data is valid and can be used by our app or not. The return value 1 means the data is correct and the other values each represent a different type of error. The return values can be used to display a specific error depending on that return value.



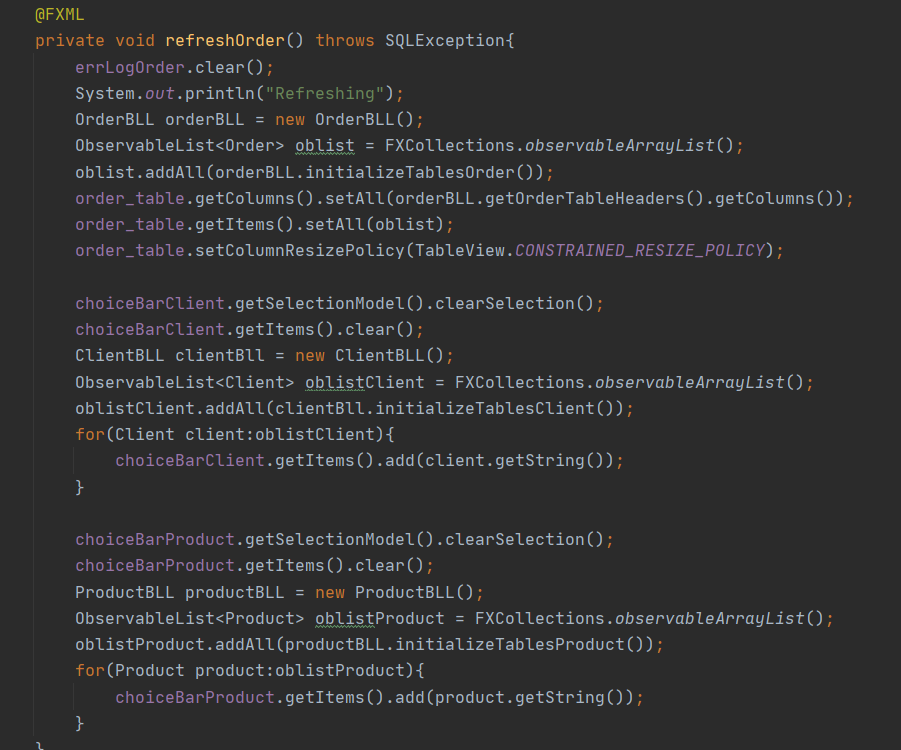
A method for deleting item with a specific id. It is similar with the other methods on strings, the main difference is the query we execute before.



A method to get the table headers from the database. We create a new TableView and a number of columns equal to the nr of columns in the database. We add these columns to the tableView and return the newly created TableView to work with it in a different method.

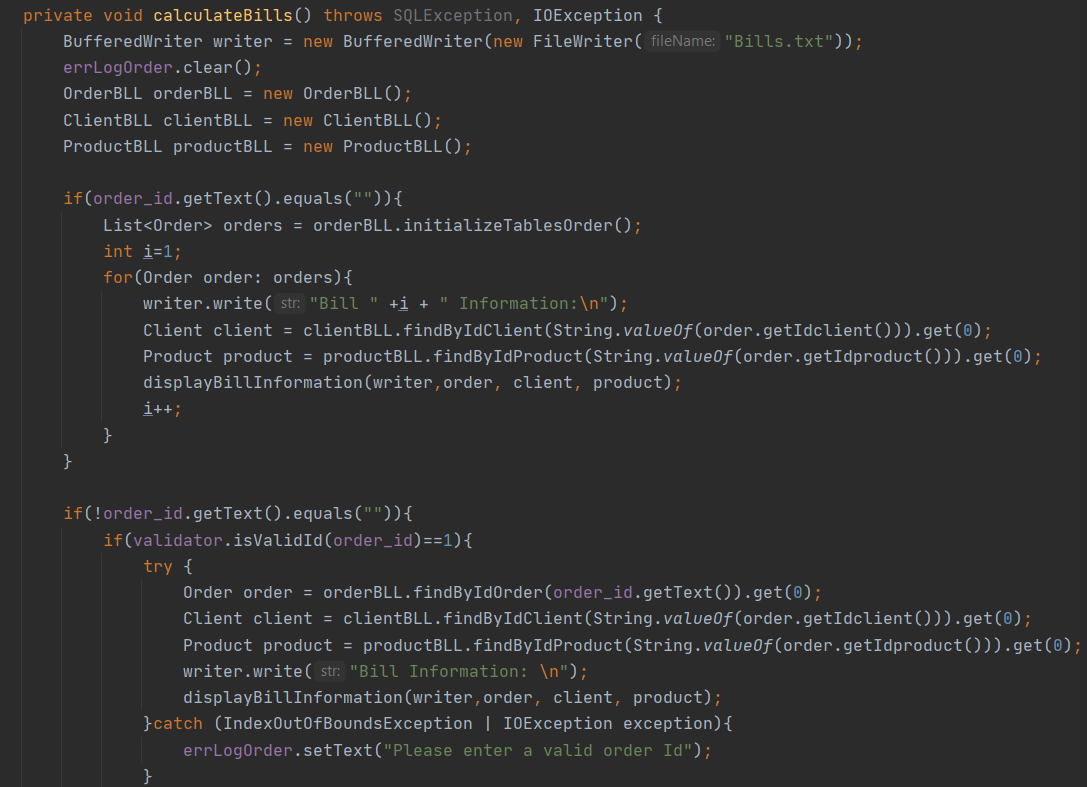


Methods for working with the database (Connection).

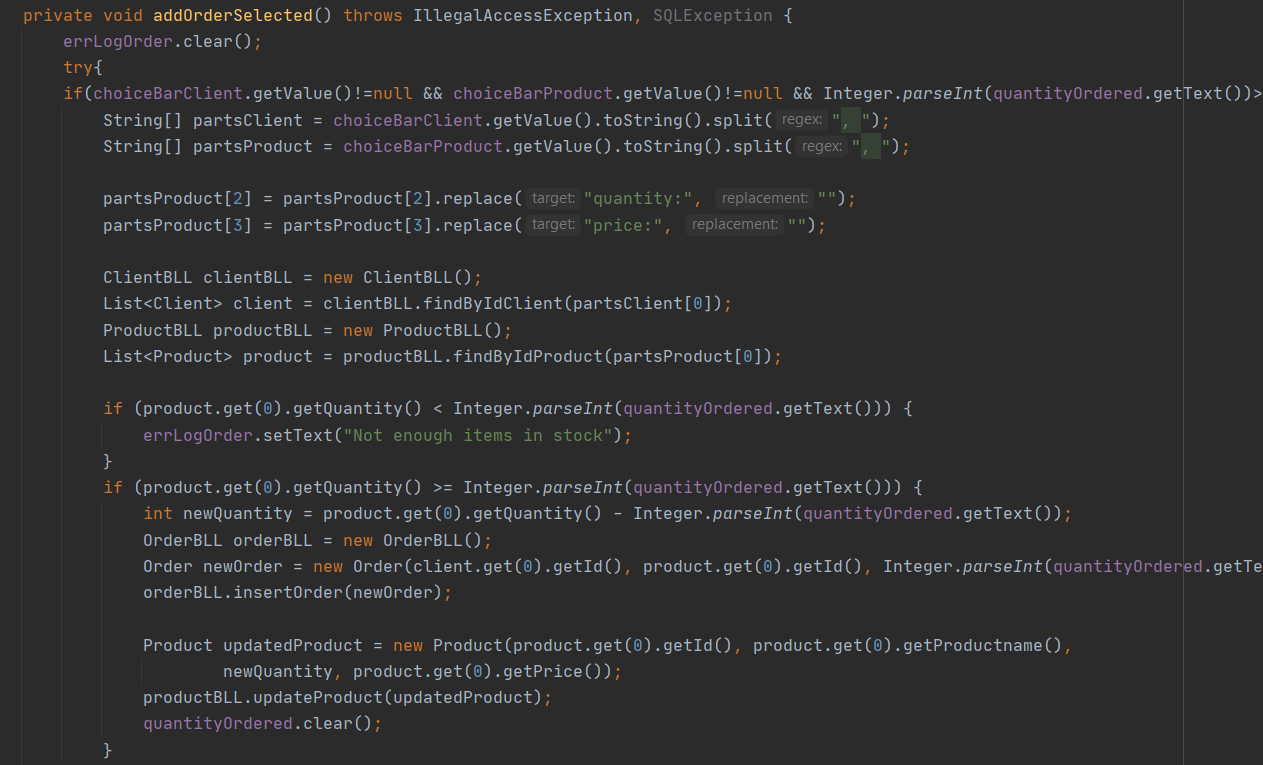


A method from the controller class that is executed when the refreshOrder button is pressed.

We update the TableView with the data in the tables and also add the objects we have in the choiceboxes available in the interface.



Method that calculates the bills. For each order, we create a string representation of the client and product and calculate the total payment amount = price \* quantity.



For adding a product we have to check if there are enough products left of that type and we also have to subtract the quantity ordered from the amount we currently have. For this method I used method I already declared for other purposes. Like updating for setting a new price. We also have to check if the user entered a valid quantity to be ordered. We only have to check if the choiceboxes are not empty because with the way I declared them, they will only let the user select clients and products that are existing inside the database.

1. **Results**

The app works as required for all the required functionalities and some extra ones I thought about (like having the option to create a bill for all the orders or just for a specific one).

I tested every method by pretending to be a bad user. The app wouldn’t let you enter wrong data that could mess up the database/flow of instructions. All text fields that require numbers should contain numbers and so on.

One error that I think could happen is when we delete data that is tied together by a order (like deleting a client that ordered something). The order will remain in the database util it is deleted but it will contain the id of a client that is not existing.

1. **Conclusions**

This project was a great learning experience because I worked for the first time in java with a database. Learning how to connect the database with my app was a bit difficult but I managed to do it.

Javadoc was an interesting and useful way of documenting my methods through comments before the actual code of the method.

I learned a lot about the observable lists and how to work with the TableView in javafx in order to display the data I wanted.

Reflexivity was another great concept I learned and I tried to create methods that I could use for the next project.

The layer structure helped me create a more readable and cohesive code.

1. **Bibliography**

--The documentation for the project, and the course

Connect to MySql from a Java application

[Introduction to JDBC | Baeldung](https://www.baeldung.com/java-jdbc)

[Connect to MySQL with JDBC driver - Mkyong.com](https://mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/)

Connect to MySql from a Java application

[Layers of a Standard Enterprise Application - DZone Java](https://dzone.com/articles/layers-standard-enterprise)

Reflection in Java

[Java Reflection Tutorial (jenkov.com)](http://tutorials.jenkov.com/java-reflection/index.html)

Creating PDF files in Java

[Creating PDF Files in Java | Baeldung](https://www.baeldung.com/java-pdf-creation)

JAVADOC

[Introduction to JavaDoc | Baeldung](https://www.baeldung.com/javadoc)

SQL dump file generation

[MySQL :: MySQL Workbench Manual :: 6.5.2 SQL Data Export and Import Wizard](https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html)