DOCUMENTATION

ASSIGNMENT NUMBER 3:

**— ORDER MANAGEMENT —**

NAME: Chețe Doru-Gabriel

GROUP NUMBER: 30423

1. Objective of the assignment

The main object of the assignment is to build an application that manages the products, clients and orders of a warehouse. In order to do so, the application will have 3 views: a view for the products, a view for the clients and one for the orders. To store the clients, products and orders, a database will be used. The users should be able do edit the lists of clients, products and orders with adding/editing/deleting/finding options. When an order is added, the stock of the ordered product is checked in order to see if the quantity asked for is available. Messages should be displayed in a message box.

The secondary objectives are:

1. Analyze the problem and identifying the requirements
2. Design the application that can manage the clients, products and orders
3. Implement the application
4. Test the built order management application

With these objectives clearly set, the start of the analysis of the problem will be presented:

2. Analysis of the problem, modeling, scenarios and use cases

Analysis:

The problem can be viewed as a matter of modeling the entities that are being stored and relating them to a database. These entities are quite obvious: there is a client entity, a product entity and an order entity. The database should have one table for each of these 3 entities. The columns of the tables in the database should match the fields of the defined client, product and order objects. We also need a mechanism for communicating with the database, a DAO, that will have reflexive implementations for creating queries and creating objects from data retrieved from the DB.

When it comes to UI, there should be buttons for all options such as add/find and a table for each of the modeled objects where the data from the database can be displayed.

Modeling:

In order to model this problem, we can consider five packages: Business Logic, Model, Data Access, Presentation and Connection. The model is rather simple.

The Client object only has a few fields, such as id, name and address and as previously mentioned, they match the columns in the database. This needs to be done considering that reflection will be used within the Data Access package.

The Product and Orders objects also have just a few fields, that also match the columns in their corresponding tables in the database. As for methods, all the objects in the Model package only have getters and setters and constructors.

For the Data Access package, an Abstract DAO object is defined, that handles the communication between the database and the application. This is defined using a generic, so all objects within the model can use the same implementation for communicating with the database. This is done using a reflexion technique. There are DAO objects for each of the three modeled entities, that simply extend the Abstract DAO.

Within the Business Logic package, some layers are modeled. We need a business logic layer for each of the 3 modeled objects. So there are three classes: ClientBLL, ProductBLL and OrderBLL. These will make use of their respective DAO classes in order to communicate with the database. For this, they contain an instance of an object of their respective DAO type that gets initialized when the BLL object is constructed. Then there are methods which use said data access object to retrieve data from the database such as columns or table data, or to edit/remove/add data to the database.

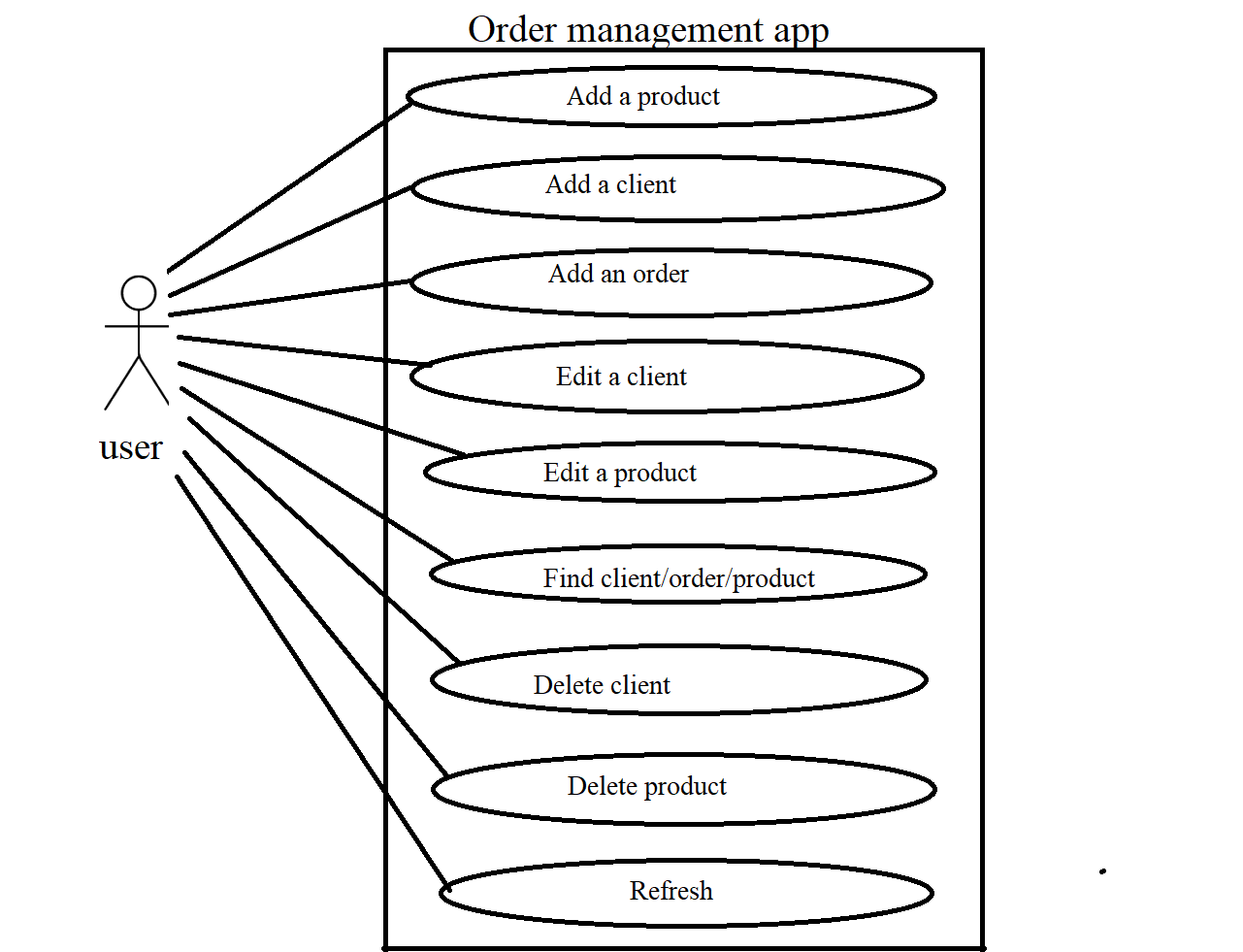
Scenarios:

The use scenarios for this application are a bit more varied. Once the application is launched a user can expect to be able to view three windows: the client view, the product view and the orders view. These windows can be switched with buttons. On any of the 3 windows, there are a couple of options the user has. One simple scenario would be that the user wants to add a client to the database. Or add a product. In order to do so, he may choose the respective options using the UI. Once a client is added, perhaps a user wants to see the new list of clients, to check if the client was indeed added to the list. For updating the UI table, the user can choose the refresh option. In the case of the orders view, the refresh option also updates the options available within the choice boxes of the clients and the products. The scenarios of editing or deleting a user or a product are also very similar to adding. User presses the button in the UI, the required actions are performed in the back-end of the application and if the action was successful, a message confirming this will be displayed in the message box below the table.

For adding an order, once the available products or clients are selected and an order id is assigned, the same process will happen. However, if stock is not enough to satisfy the ordered quantity, then a message conveying this issue will be displayed an the order will not go through.

There is also the scenario when a user simply wants to find a certain client/product/order. In this case, he may choose this option in the UI, and if a client/product/order is found, then its corresponding data will be displayed in the message box.

Use cases:



The use cases for this program are in conclusion quite simple, and refer to actions performed in the database. To sum up, if an insertion, update or deletion is to be performed, a check is being made in order to see if this action would be valid within the database and if this is the case, a message is displayed and once refreshed, the table will reflect the change that was made. If an object is searched by ID, the database will simply be searched for it. In any scenario, if there is a problem with the provided data or lack thereof, an error message is displayed and no change is performed in the database.

3. Design of the simulator

For the design of the management application, I started with the ideas presented in the modeling part of the documentation and followed them faithfully.

The model package classes are very simple, with just a couple of fields describing them and their respective gettter and setter methods.

The Data Access package is the heart of the application. Here, within the AbstractDAO class, the methods for interacting with the database are implemented. There is no need for designing a separate DAO class for each of the modeled objects. Using a reflexion technique, the abstract object can be used for all of them. The Client, Product and Order DAOs simply extend the Abstract object, which is designed with a generic.

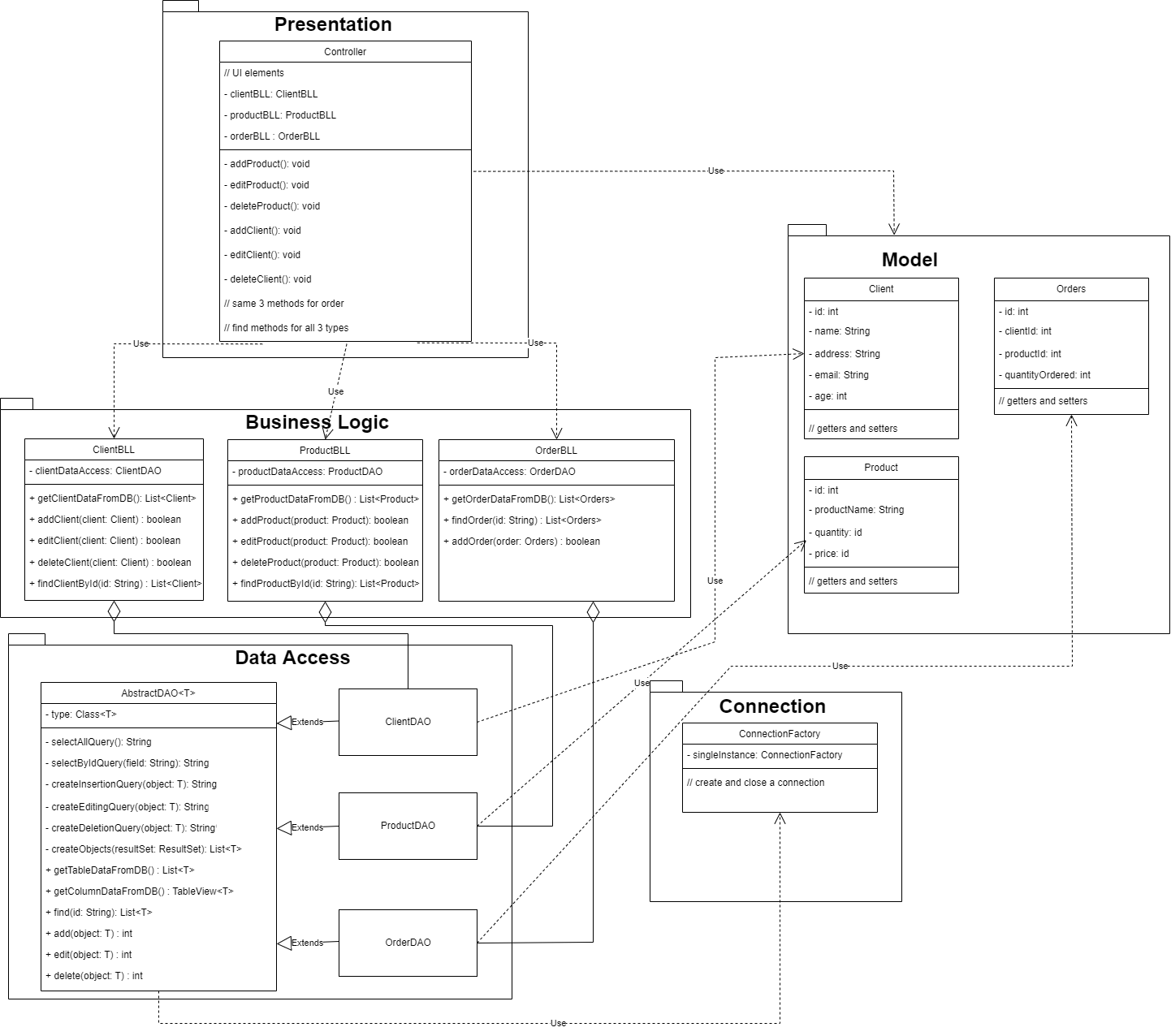
Within the Presentation package, the Controller class servers as the controller for all of the views of the application. Since only one is running at a time, the same controller can be used, all UI elements being defined within it, but only being used while their corresponding view is active. It’s obviously important to avoid situations when a UI element from one view tries to interfere with a UI element from another view. Since the scene of the FX application is changed every time the view is switched, this will generate errors.

Every “view” UI has its design within its own .fxml file. When a view is “switched”, its corresponding fxml file is loaded and the scene is changed to it.

Data structures that are used:

With respect to the data structures that are used, we can talk mostly about lists. Lists are used as the data structure in which all of the created objects from the retrieved rows from a query are stored. When a query is performed, a number of rows will be return and as previously discussed, their fields match the fields of our modeled objects. So for every row, an objects is created. A list is used to collect all of these objects and return it to the controller, through the BLL objects. A specific type of List is used in order to match the fields of the objects with their columns within the JavaFX TableView object, the ObservableList

UML diagram



For the UML diagram, the packages described above are illustrated. The AbstractDAO object has all the necessary methods for interacting with the database, obviously being defined with a generic. The ClientDAO, ProductDAO and OrderDAO all extend this class. The relationship between BLL objects and the DAOs is one of aggregation, not quite of composition. The BLL objects use their corresponding DAOs for communicating with the database and then providing their methods within the controller. Also the AbstractDAO uses the ConnectionFactory object, which is used for establishing a connection to the database and closing the respective connection. The DAO objects also use, quite obviously, the modeled objects, as the given type for the generic of the AbstractDAO. The Controller uses the modeled objects as types for returned lists and such, but mostly deals with the BLL objects.

Algorithms:

For algorithms, since this is an application that is heavily based on the interaction with the database, there isn’t really that much in the way of algorithms. However, some simple algorithms are used in order to build the necessary queries to interact with the database. For example, while building the insertion query, we consider all of the fields of the object using reflexion, and for each of them we append the string that will be used as the query. For insert we need to enumerate all of the fields that are to be inserted and their values, with the “,” separator. Initially this separator is an empty string, and before every time a field name is appended, the separator is appended firstly. At the end of the for, the separator is changed to “,” such as only the first time the loop is executed the “,” separator is not appended firstly. This may seem convoluted, but the idea is that this algorithm builds the string query without having a trailing comma, which would be a syntax error in SQL. For building an editing query, I also considered the case when some of the fields aren’t to be changed and so their corresponding UI fields are left empty. In such a case, before adding that field in the built query, the program will check to see if the field actually contains something and if not, it won’t add it to the update query.

4. Implementation

The implementation, considering the constraints of the design is quite straightforward. The modeled structures can be easily implemented in Java. As previously mentioned, the classes in the model package are very simple. Primitive types are used for their fields and there are only getters and setters.

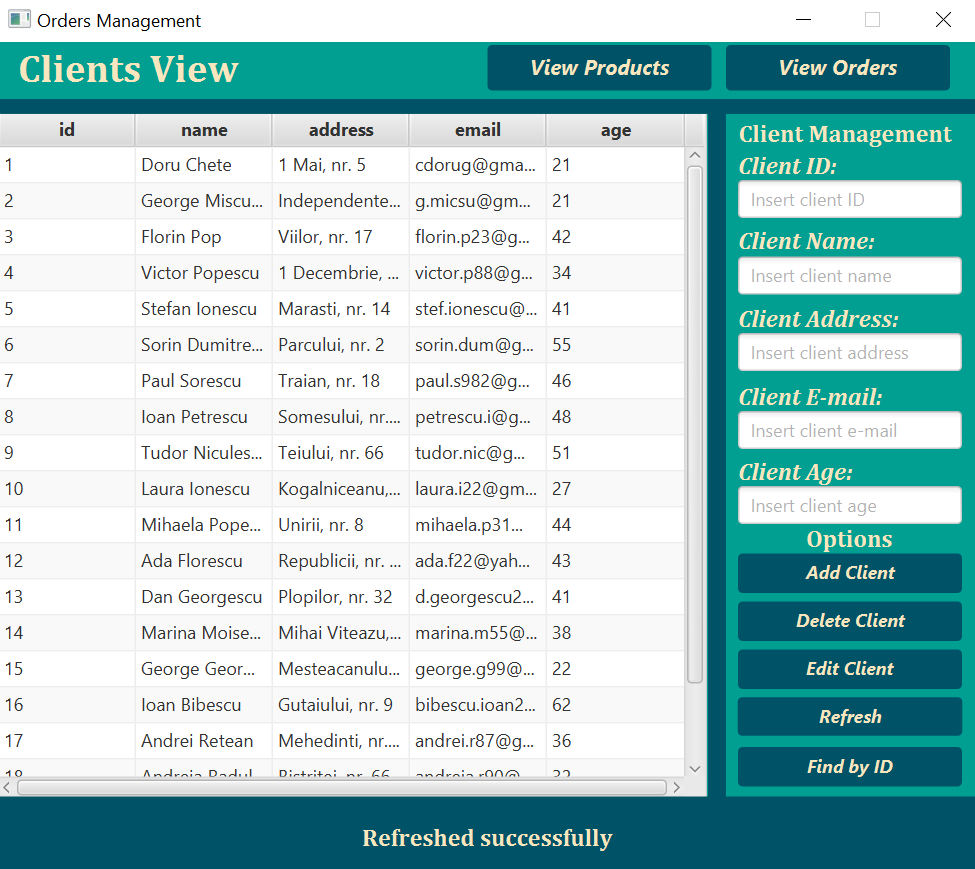
For the BL package, a data access object instance is used, being initialized within the constructed and is declared as final. For modification operations, methods such as addClient() return a Boolean, being true if the operation was successful or false otherwise. This is used within the controller to check if the operation was successful and update the UI accordingly.

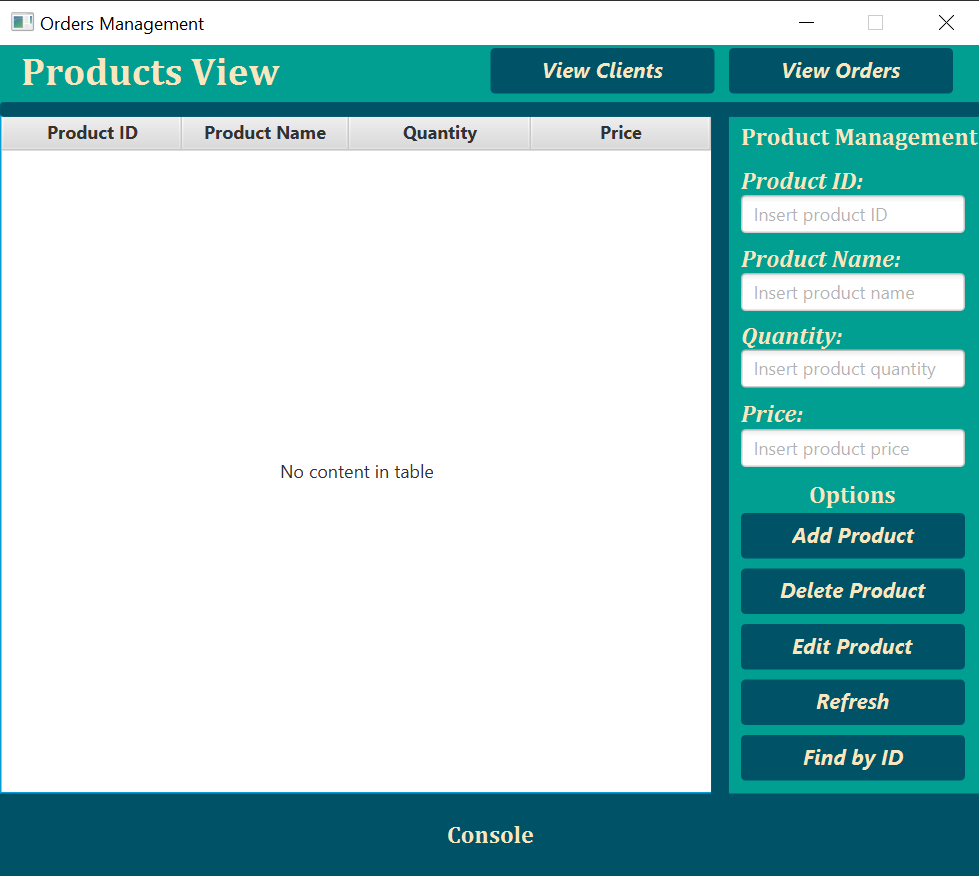
The Connection Factory contains data for connecting to the DB, and is initialized as a singleton object. It is implemented having methods to connect to the DB then to return the connection with the getConnection() method and also a method for closing the connection.

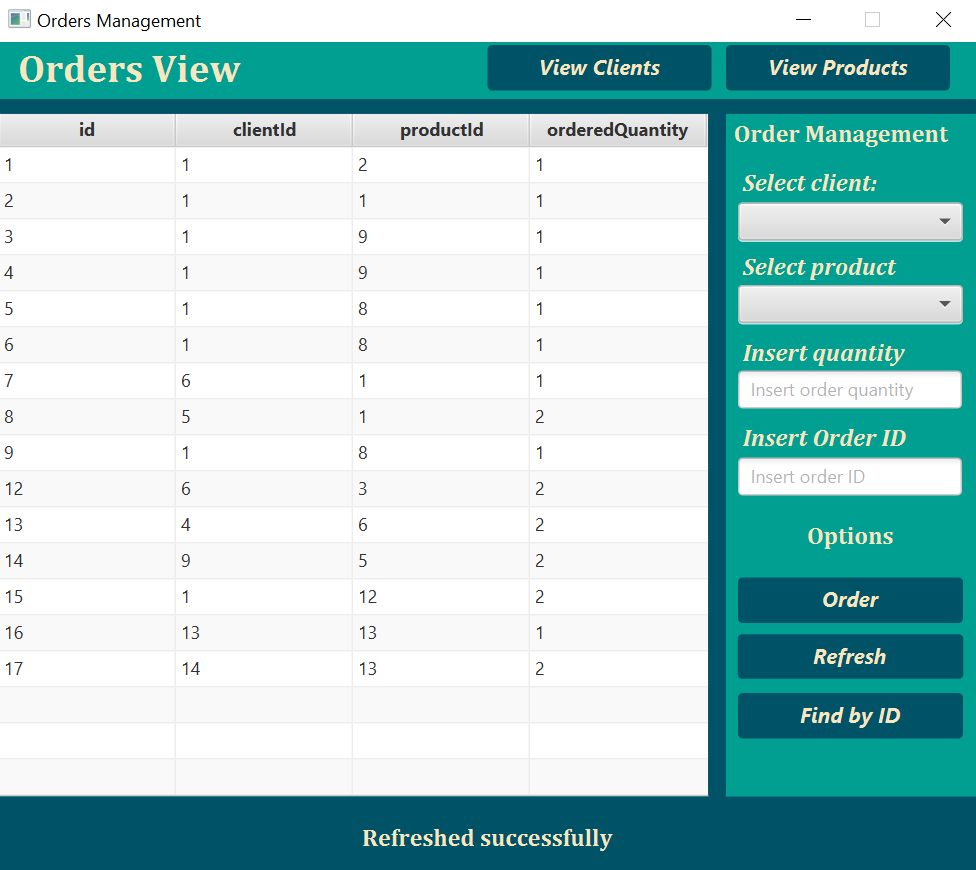
The AbstractDAO is declared using a generic, and has a field named type. This is used for the implementation of the methods using reflexion. Methods for creating queries for adding, updating deleting, selecting all or selecting by id are implemented by using a StringBuilder object that is used repeatedly by appending necessary strings based on the fields of the object given as parameter and obtained using reflexion. These are private methods and they return the built query as a string. They are used within the add, edit, delete or find methods which are public and used within the implementation of the BLL objects. These return an integer, which would be -1 if the operation wasn’t successful or positive if it was. The find() method returns a list containing just one object, the found object.

The application is built using JavaFX, so within the controller there are a lot of FX UI elements. The UI elements with fx:ids that have to be injected have a corresponding field, however they are all separated in their use. No label, for example, from the ProductsView gets updated from any of the other views. The methods for adding an object to the list are different for the product, client and orders. There is also the bill generation, which happens within the addOrder method. Once an order is added successfully, a bill containing all relevant data is built using a FileWriter object, so the bill will be generated as a .txt file.

Finally, the .fxml files with the GUI:







This was made using JavaFX library. It is rather simple. A user can use the text fields in the right section of the UI for inserting data and using the options, and viewing the table data in the left part. On the top, the user can switch between the windows.

5. Results

For testing the implementation of the program, I tried to consider all scenarios, including some scenarios that generate errors. For example, I considered scenarios where the user only wants to edit certain fields of a client or order. Or scenarios when a user erroneously tries to insert a negative value for the ordered quantity. Such scenarios will generate an error, which will be displayed in the message box at the bottom of the UI, as previously mentioned. Otherwise, for correct scenarios of use, the database updates correctly and the results are as expected.

6. Conclusions

During the analysis and development of this project I learned a lot about development of applications using a database. I learned a lot about creating and using DAOs and the way they may interact with a database. I also learned how to adapt the UI of a JavaFX application to suit the requirements of such an application, namely using the TableView object.

7. Bibliography

Helped me with finding RGB colour codes for the UI: <https://www.color-hex.com/color/5d1049>

Helped me learn more about the FileWriter, used for bill generation: https://www.geeksforgeeks.org/filewriter-class-in-java/

Made me understand there was an issue with naming a class ‘Order’ https://stackoverflow.com/questions/23446377/syntax-error-due-to-using-a-reserved-word-as-a-table-or-column-name-in-mysql

Helped me understand how to use TableView in JavaFX: <https://jenkov.com/tutorials>

Helped me understand JavaDox: https://www.baeldung.com/javadoc