**ASSIGNMENT 3 DOCUMENTATION**

**ORDER MANAGEMENT**

**Name: Zarnescu Radu**

**Group: 30423**

**Laboratory Assistant: Antal Marcel**

Contents

[1. Assignment Objective 3](#_Toc476131445)

[2. Problem Analysis 3](#_Toc476131446)

[3. Design 7](#_Toc476131449)

[3.1. Data Structures 7](#_Toc476131451)

[3.2. UML Diagram 8](#_Toc476131452)

[4. Implementation 8](#_Toc476131454)

[4.1. Class Description 8](#_Toc476131453)

[5. Testing 12](#_Toc476131454)

[6. Conclusions and Further Improvements 12](#_Toc476131455)

[7. Bibliography 13](#_Toc476131456)

1. **Assignment Objective**

Consider an application OrderManagement for processing client orders for a warehouse. Relational databases are used to store the products, the clients and the orders.

Create a graphical user interface including:

* A window for client operations: add new client, edit client, delete client, view all clients in a table
* A window for product operations: add new product, edit product, delete product, view all product in a table
* A window for creating product orders - the user will be able to select an existing product, select an existing client, and insert a desired quantity for the product to create a valid order. In case there are not enough products, an under stock message will be displayed. After the order is finalized, the product stock is decremented.

Use reflection techniques to create a method that receives a list of objects and generates the header of the table by extracting through reflection the object properties and then populates the table with the values of the elements from the list.

1. **Problem Analysis**

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

* The required classes for reflection are provided under java.lang.reflect package.
* Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
* Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.

Reflection can be used to get information about –

1. **Class** The getClass() method is used to get the name of the class to which an object belongs.
2. **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
3. **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.

A use case is a written description of how users will perform tasks on your website.  It outlines, from a user’s point of view, a system’s behavior as it responds to a request. Each use case is represented as a sequence of simple steps, beginning with a user's goal and ending when that goal is fulfilled.

Basic Flow: Adding a new client:

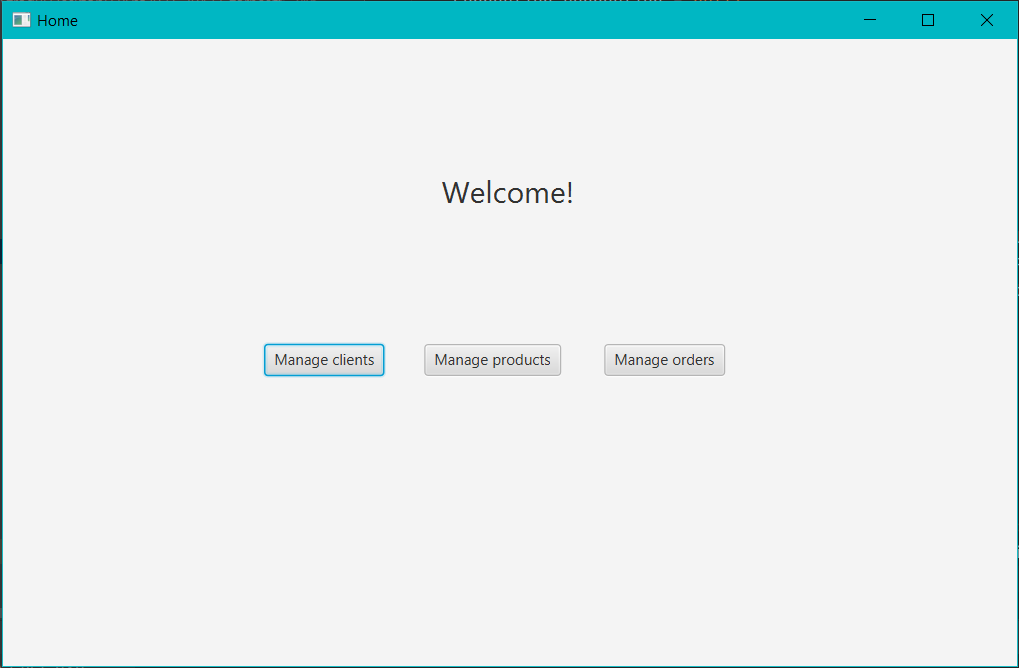
* Actor: User
* Description: The main success scenario is the situation where the user inserts a new client to the database using the client operations window

1. User presses the “Manage clients” button
2. User inputs the desired name, address and email in the text fields
3. User presses the “Insert client” button

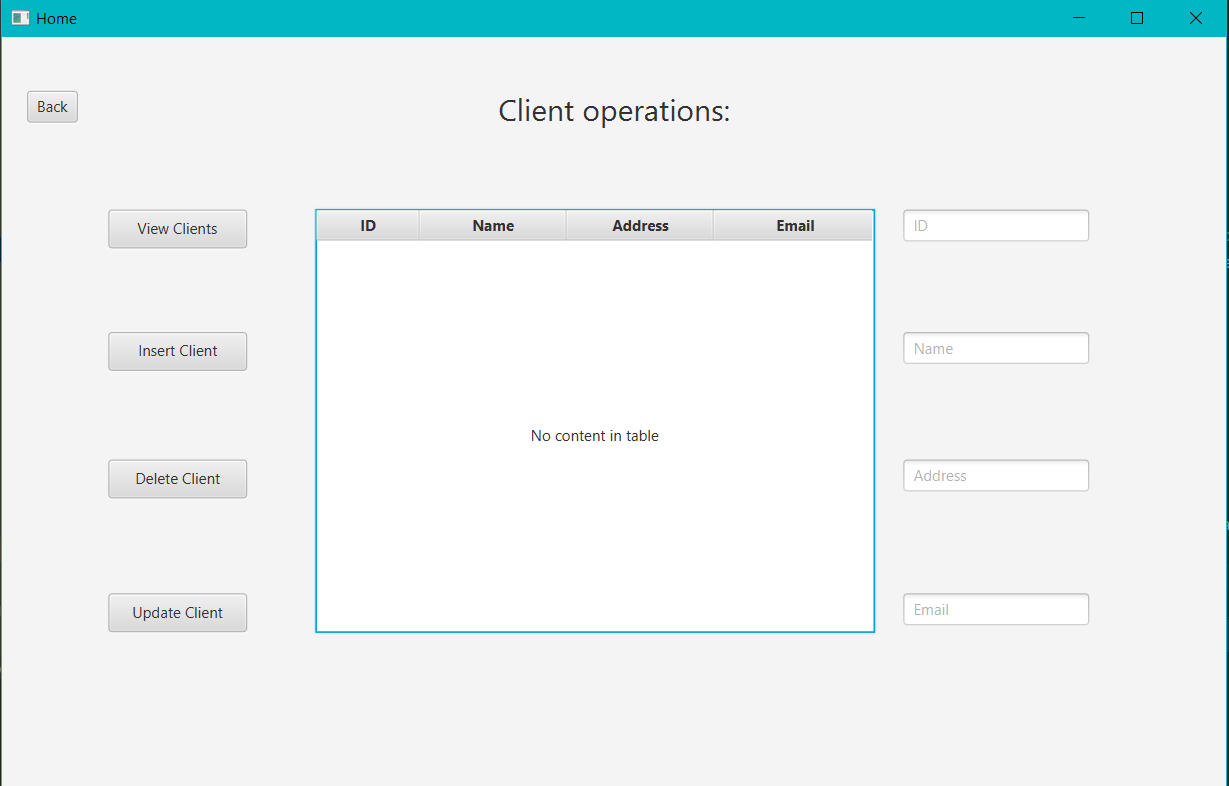
* Termination outcome: the input client is added to the database successfully.

Illustration of the use case presented above:

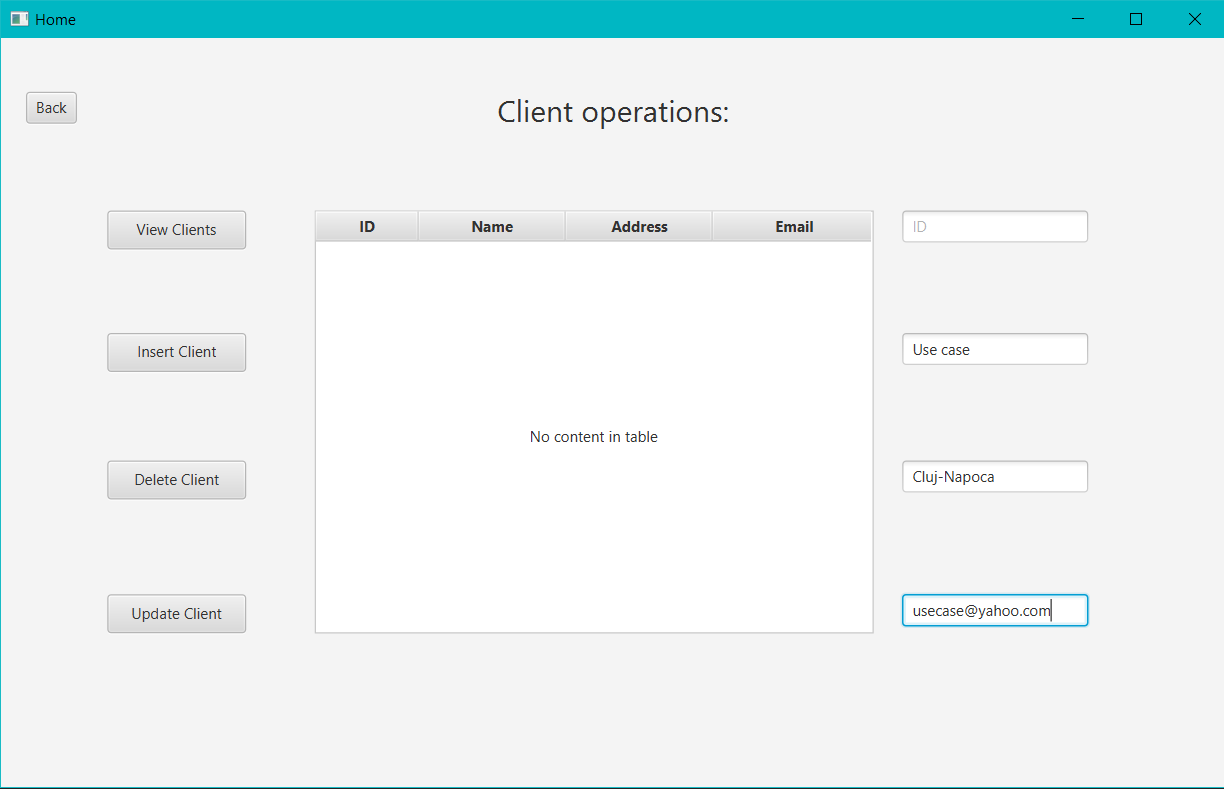
The user interacts with the program through a dedicated UI found in the screenshot below. It is very simple to use and it indicates the user what every field represents.



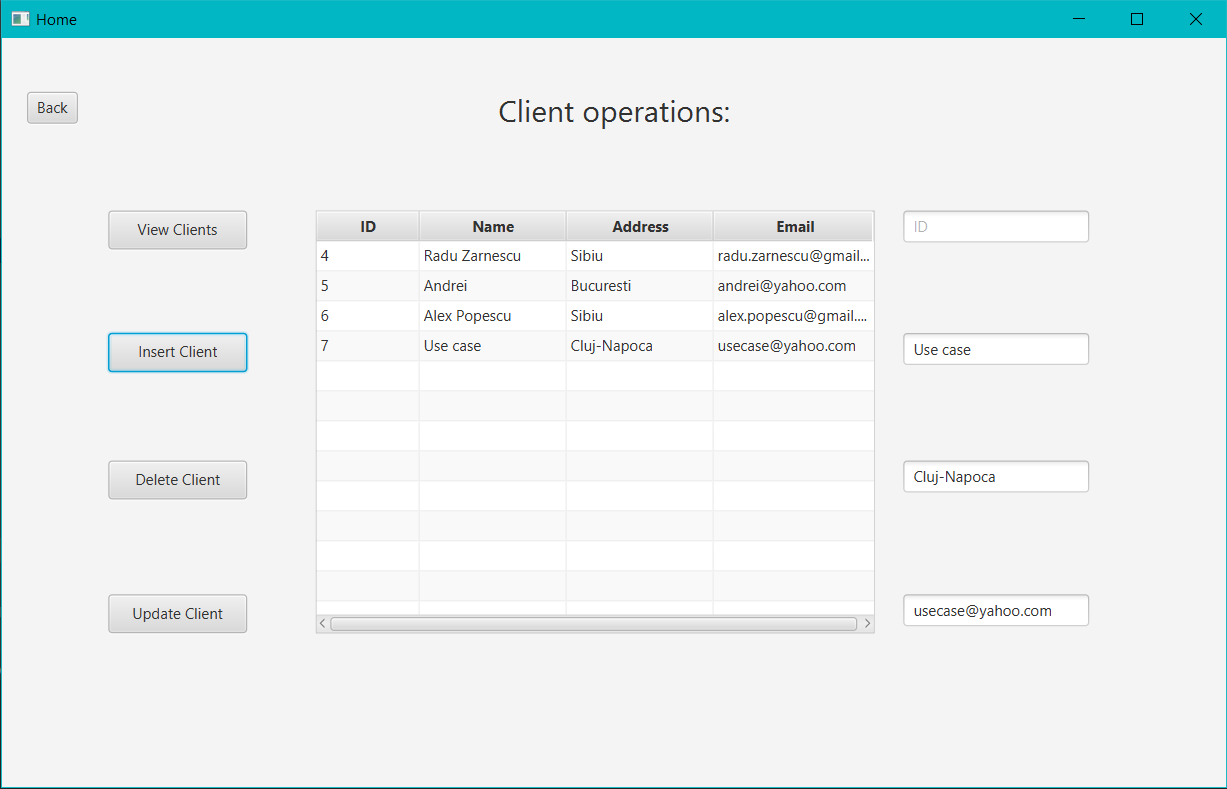
To insert a new client, the user has to press the “Manage clients” button. Then, the following window will be shown.



The next step is to insert the desired values to the text fields on the right hand side (i.e. Name, Address, Email). An ID is not required because for new clients it is automatically created. If desired, before this step, the user can press “View Clients” to visualize all the clients in the table.

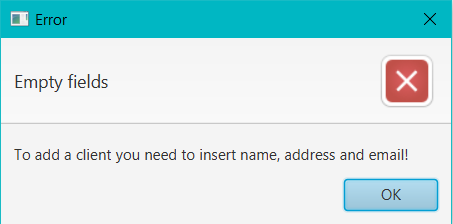


The remaining step is to press the “Insert client button”. The client will successfully be inserted into the database and all the clients will be displayed on the table.

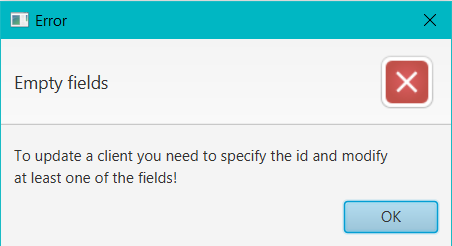


Cases when the program cannot function properly (error cases):

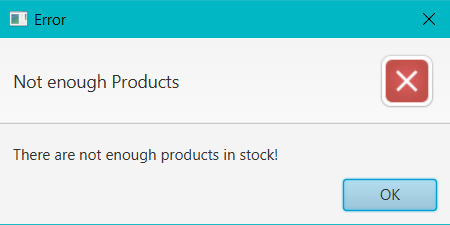
If the insert button is pressed and any of the fields is empty the following error will pop up.



If the “Update client” button is pressed and the id or one of the other fields are empty the following error will be displayed.



There are other possible errors, but I will not go into any further depth. However, one important error is the next one, which is the error displayed when an inputted quantity in the orders window is greater than the stock of the selected product.



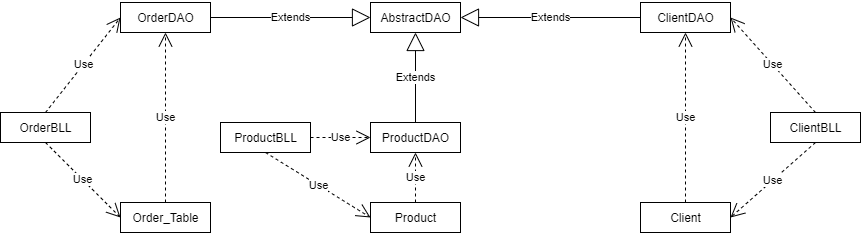
1. **Design**
   1. **Data Structures**

In order to implement the required problem, I have chosen to use three main data structures:

* Client: the class used to store clients, containing an id, name, address and email
* Product: class used to store products, containing an id, name, price and stock
* Order\_Table: class used to store the orders, consisting of id, clientID, productID and quantity

I have chosen to use ArrayLists instead of simple Arrays because they are more efficient for memory management and also, they do not have a fixed number of elements.

* 1. **UML Diagram (Class Diagram)**



1. **Implementation**
   1. **Class Description**

I based my class structure on the following packages:

1. Model: includes the classes which mostly deal with the back end of the project, the algorithms and the operations which allow the program to function. It is the part which the user does not see at all when using the app. Classes: Client, Product, Order\_Table
2. Data Access Layer (DAO): package that includes the classes that create and execute queries on the database. Classes: AbstractDAO, ClientDAO, ProductDAO, OrderDAO
3. Business Logic Layer (BLL): includes the classes which call the methods in the data access layer. These classes are used to perform operations at the highest level. Classes: ClientBLL, ProductBLL, OrderBLL
4. DatabaseConnection: contains the class responsible for establishing the connection between the application and the database, ConnectionFactory
5. View: represents the classes that form the user interface, the part with which the user interacts. Classes: Alert, Main
6. Controller: the collection of classes which make the connection between back end and front end, model and view. For example, when a button is pressed, the controller is responsible for calling the appropriate methods and algorithms that apply to that action. Classes: Controller, Writer.

* Client Class

The client class is the data structure used to store all the information from the Clients table.

Methods:

* public Client(int id, String name, String address, String email): the constructor which overloads the default one. It creates a new object of the Client class.
* Setters and Getters which are needed to access private attributes.
* Product class

The Product class is used to store the entries in the Product table

Methods:

* public Product(int product\_id, String product\_name, int product\_price, int product\_stock): the constructor which overloads the default one. It creates a new object of the Product class.
* Setters and Getters which are needed to access private attributes.
* Order\_Table Class

The class used to store and create the order entries from the database.

Methods:

* public Order\_Table(int order\_id, int order\_clientID, int productID, int quantity): the constructor which overloads the default one. It creates a new object of the Order\_Table class.
* Setters and Getters which are needed to access private attributes.
* AbstractDAO Class

The abstract data access object class. This is the main class which executes and sets up queries for the database. It is used for reflection, meaning that the methods in this class should work for every table in the database.

Methods:

* deleteById​(int id): method used to delete from the table represented by T the entry with the id specified.public void writeToFile(String s): this method writes the string s to the output file.
* findAll(): this method reads all the data from the table, and transforms it into multiple objects of the T type.
* insert​(T t): method that inserts into the required table an object of type T.
* update​(T t): method used to update the object t from the table.
* ClientDAO Class

The class which extends AbstractDAO in order to perform it’s methods on the Client class.

* ProductDAO Class

The class which extends AbstractDAO in order to perform it’s methods on the Product class.

* OrderDAO Class

The class which extends AbstractDAO in order to perform it’s methods on the Order\_Table class.

* EmailValidator Class

This class is used to validate the emails of the Clients. If they do not match the default email pattern, an exception is thrown.

* ClientBLL Class

The business logic layer for client operations. This class is the top layer used to execute the operations in ClientDAO and AbstractDAO.

Methods:

* public List<Client> findAll(): this method reads all the clients from the table and returns a List including all of them.
* public Client insert(Client c): this method inserts a new client, c and returns it if successful.
* public Client deleteById(int id): this method deletes the client with the id given as parameter and returns null.
* public Client update(Client c): this method updates the client given as parameter and returns it if successful.
* ProductBLL Class

The business logic layer for Product operations. This class is the top layer used to execute the operations in ProductDAO and AbstractDAO.

Methods:

* public List<Product> findAll(): this method reads all the products from the table and returns a List including all of them.
* public Product insert(Product p): this method inserts a new product, p and returns it if successful.
* public Product deleteById(int id): this method deletes the product with the id given as parameter and returns null.
* public Product update(Product p): this method updates the product given as parameter and returns it if successful.
* OrderBLL Class

The business logic layer for Product operations. This class is the top layer used to execute the operations in OrderDAO and AbstractDAO.

Methods:

* public List<Order\_Table> findAll(): this method reads all the orders from the table and returns a List including all of them.
* public Order\_Table insert(Order\_Table o): this method inserts a new order, o and returns it if successful.
* public Order\_Table deleteById(int id): this method deletes the order with the id given as parameter and returns null.
* public Order\_Table update(Order\_Table o): this method updates the order given as parameter and returns it if successful.
* ConnectionFactory Class

This class is used to establish the connection between the application and the database.

* MainController Class

The controller for the home page. It is used to navigate through the client, product, order windows.

Methods:

* public void changeSceneOnButtonAction(ActionEvent event, String fxml, Button btn): The method used to switch the windows of the GUI.
* public void btn1OnAction(ActionEvent event): The method which handles the press of the button “Manage Clients”.
* public void btn2OnAction(ActionEvent event): The method which handles the press of the button “Manage Products”.
* public void btn3OnAction(ActionEvent event): The method which handles the press of the button “Manage Orders”.
* ClientsController

Controller used for the client operations window.

Methods:

* getClients(): Method which gets all the clients from the database using ClientBLL
* insertClient(ActionEvent event): Method to insert a new client
* deleteClient​(javafx.event.ActionEvent event): method to delete the selected client
* updateClient​(javafx.event.ActionEvent event): method to update a client
* viewClients​(javafx.event.ActionEvent event): Method to display the clients on the table view
* ProductsController

Controller used for the product operations window.

Has the same methods as ClientsController, but with Product instead of Client.

* OrdersController

Controller used for the order operations window.

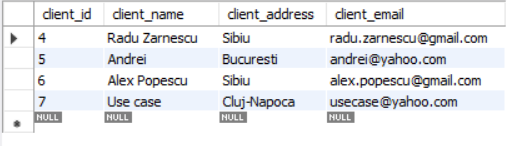
Methods:

* generateBill​(Order\_Table o, ObservableList<Client> clients, ObservableList<Product> products): method to generate bills for each order
* addOrder​(ActionEvent event): method to create a new order and insert it into the database

1. **Testing**

The most important part of the project is the testing. If the results are not correct, then the program is not doing it’s job properly. In order to test whether the app is working well, we can just run the program, follow the steps presented in section 2, and observe the output.

It can be seen in the database that the client added In section 2 was successfully inserted in the table.



1. **Conclusions and Further Improvements**

To conclude, this assignment was a very good exercise in developing an application that uses a database. I also learned a lot in terms of organizing the code into multiple classes (DAO, BLL) and also, how to use reflection.

I encountered some small problems, mostly with the operations used to generate and execute queries, but in the end I managed to solve them.

As for further improvements, it would be a good idea to implement a table where the data could be modified directly on it. Also, a better looking GUI wouldn’t hurt.

Overall, I am happy with my project and consider I did a good job implementing it.

1. **Bibliography**

* wikipedia.org
* [www.stackoverflow.com](http://www.stackoverflow.com)
* geeksforgeeks.org
* lecture courses and laboratory sessions