DOCUMENTATION

ASSIGNMENT *3*

NAME: STOICA IRINA

GRUPA: 30422

**TABLE OF CONTENTS**

1. Assignment Objective……………………………………………………………………………………3
2. Problem analysis, modelling, scenarios, use cases………………………………………………………3
3. Design……………………………………………………………………………………………………4
4. Implementation…………………………………………………………………………………………..9
5. Conclusions……………………………………………………………………………………………..10
6. **Assignment Objective**

The main objective of the assignment is to design and implement a Java application which is suitable for holding the evidence of the client orders in a warehouse. In order to achieve this goal, a database is useful and a lot more suitable than a file system, because our data is constantly interacting with other data and changing. The user of the application should be able to add, modify the data or delete the client or the product he chooses. One should also be able to make an order by selecting a client, a product, and the quantity of the desired product. The application should load the data from the database and show it to the user, but also update and make modification in the database when the user requests. After creating an order, a bill with the most important details of the order should be generated in form of a text file.

As a general objective, we should implement a conceptual architecture which is suitable and makes sense for our application. In our case, a good architecture option is the layered architecture which consists of the presentation, the business logic, the data base access and the SQL database layers.

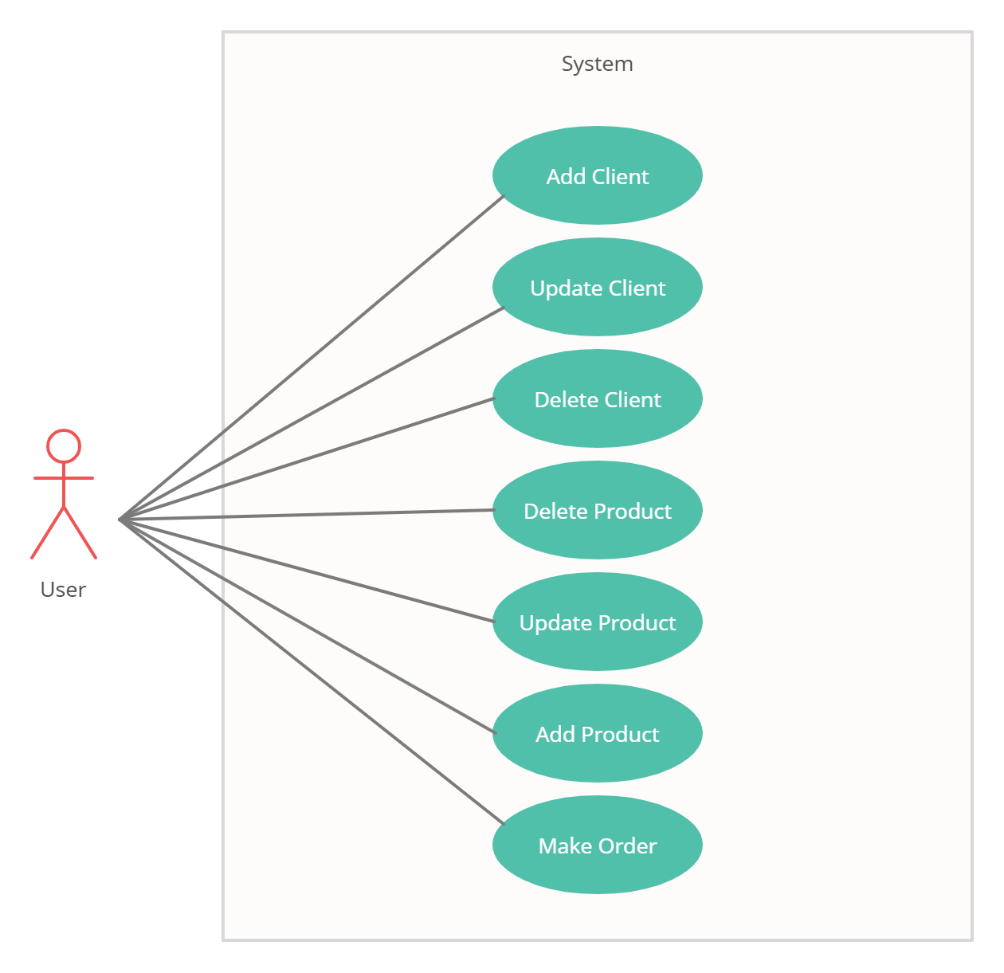
1. **Problem analysis, modelling, scenarios, use cases.**

Requirements

1. The employee using the application should be able to add a new client.
2. The employee using the application should be able to edit the information of an existing client.
3. The employee using the application should be able to delete a client he chooses.
4. The employee using the application should be able to add a new product.
5. The employee using the application should be able to edit the information of an existing product
6. The employee using the application should be able to delete a product he chooses.
7. The employee using the application should be able to choose a client, choose a product and the quantity of the product and create an order.
8. The application should generate a bill for each order created by the employee.
9. The fields with user entered data should be checked for possible incorrect data or missing data.
10. The application should let the user know when errors occur or when the entered data is not valid.
11. The employee using the application should be able to see all the clients in the database.
12. The employee using the application should be able to see all the products in the database.
13. All the modifications happening in the application should also happen in the database

Use cases

1. The user chooses the window he wants to view client, product, or order.
2. The user enters the needed data for a product and adds the product.
3. The user enters new data for an existing product and modifies it in the database.
4. The user chooses a product he wants to delete from the database and deletes it.
5. The user enters the needed data for a client and adds the client.
6. The user enters new data for an existing client and modifies it in the database.
7. The user chooses a product he wants to delete from the database and deletes it.
8. The user chooses a client, a product, the quantity of the product and places an order.



1. **Design**

Class Design

The chosen architecture for this application is the layered architecture. The first level is the SQL database level which consists of the schema itself. We will have three tables in our schema. One for the clients, one for the products and one for the orders. The next level is the data access layer. This layer creates and executes queries in relation to the database and transforms the result sets of the queries in objects. These objects must have the same name as the table they are created from, and all the columns of the table must have a corresponding field name in the class of the object. The data access layer is the only layer that has access to the database. The business logic layer gets the data from the data access layer and performs the computations and logic part of the application. The last level is the presentation level which contains the user interface and calls methods from the business layer.

For each layer we have a separate package containing the needed classes. In addition, we have the model package which contains the classes Client, Product and Orders. These classes are used by all the layers, but don’t contain any logic or computations. They only contain fields, their corresponding getters and setters and various constructors. The class containing methods regarding the database connection is placed in a different package and is accessed only by the AbstractDAO<T> class placed in the dao package.

The model package contains three classes: Client, Model and Orders. These classes correspond to the database tables and their fields correspond to the columns of the table. All their fields need to have getters and setters in order to perform the database queries, inserts, updates and deletes.

The bll package contains another package called validators and contains the interface Validator<T> which requires the implementation of the validate function. The three classes that implement this interface are EmailValidator<Client>, NumberValidator<Product> and QuantityValidator<Orders>. The interface receives an Object T and validates it. Besides the validators package, the bll also contains the classes ClientBLL, ProductBLL and OrderBLL. These classes contain methods that check if the data received from the user is valid and can be used for the queries. If not, exceptions are thrown.

Graphical user interface, application

Description automatically generated

Text

Description automatically generated

The dao package contains methods that create and execute some basic queries for the database, such as insert, delete, update, select a row given an id and select all rows of a table.

The presentation package holds the responsibility of the user interface. It contains the controller for each window and their corresponding .fxml files generated with the SceneBuilder. In addition, there is a helper class called TableGenerator<T> which, given an object in the place of T, will generate through the reflection technique the header of the table and add a given list of objects to it. There is also a PopUpWindow class that creates and displays a new window in case the application needs to send an important message to the user. The class App loads the starting window .fxml file and contains methods for changing the stage from outside the class.

User interface

The user interface has a main menu, which is loaded at the start of the application and contains three buttons which can be clicked and will load the window concerning the clients, the window responsible with the products or the window for making an order. The last button closes the window and terminates the application.

Graphical user interface

Description automatically generated

When clicking “View Clients” the window changes and the window for clients is loaded. On the left side there is a TableView in a ScrollPane which contains all the entries of the client table. On the right side we can see text field boxes for entering data needed for adding, deleting and updating a client. There is also a button for returning to the main menu and select another window.

Graphical user interface, table

Description automatically generated

When clicking “View Products” the window changes and the window for products is loaded. It is very similar to the client window because the functionalities for clients and products are very similar. On the left side there is a TableView in a ScrollPane which contains all the entries of the product table. On the right side we can see text field boxes for entering data needed for adding, deleting, and updating a product. There is also a button for returning to the main menu and select another window.

Graphical user interface

Description automatically generated with low confidence

When clicking the “Make Order” button, the window for orders is loaded. This window is a little different from the other two. On the upper left side we can see the client table from the data base in form of a TableView. On the upper right side we see the product table from the database. The user has to click on a client from the database and then click the “Add Client” button. The chosen client will be displayed once again on the screen. The user also has to click on a product from the right TableView and then click “Add Product”. After entering the desired quantity, he can click on the “Make Order” button and the bill will be generated in the background.

Graphical user interface, application, table

Description automatically generated

All fields for the operations are checked and if an error occurs, the application will create a pop up window with the error message if there is one, otherwise it displays a confirmation message that the operation was successfully executed.

Graphical user interface, application

Description automatically generated

1. Graphical user interface, application

   Description automatically generated**Implementations**

One of the most important and vast classes is the AbstractDAO<T> class. This class performs the basic abstract operations for a database lice insert, delete, update, selecting by id or selecting all entries. The class T is a class that corresponds to a table in the database. There is a method for creating each query. The method createObjects(ResultSet resultSet) converts a given result set into a list of objects with the same fields as the table columns. Each query tries to create a connection with the database and execute the specified query. Then it tries to convert the result set obtained to the Object. If the connection fails or any other exception occurs, the function throws the NoSuchElementException further in the application. The classes ClientDAO<Client>, ProductDAO<Product> and OrderDAO<Orders> only inherit the AbstractDAO<T> class and set the T object to a specific object, but do not contain any additional operational.

In the bll package, the ClientBLL and ProductBLL are very similar. A main difference is the validator. While the Client has the EmailValidator, which checks for the client email to be a valid one, the Product class has a NumberValidator which checks for the price and quantity to be positive numbers. These classes wrap the basic operations of the ClientDAO and ProductDAO and perform additional conversions and checks of the data received from the user. For example, the function *public Client findClientById(String idString)* receives an id in strind form from the presentation layer. If this String is an empty string, it throws a NumberFormatException which will be caught in the ClientController. If the idString is not empty, there is an attempt to parse it as an Integer. If this fails, there will be another NumberFormatException thrown, with another message to separate it from the first one. Last but not least, if a client with that id is not found, a NoSuchElement exception is thrown. All the methods work in a similar way but have slightly different field checks and exceptions thrown. The OrderBLL class has the method for generating the bill after the order is placed. The bill will contain the name, address and email of the client and the product, the quantity, the price per unit for the product and the total cost of the order. The OrderBLL also has the Quantity validator which checks for the quantity to be a positive value. The class contains an additional method which checks for the quantity in the order to be smaller than the quantity available for the product in the database, otherwise, an IllegalArgumentException will be thrown.

The presentation package contains the controllers for all the windows. All the controllers use the TableGenerator<T>. This class has some useful methods regarding the creation of the TableView. The function *public void generateTableHeader(List<T> objects, TableView<T> tableView)* receives a list of object of type T and a TableView<T>. It extracts the fields of the object using the reflection technique and creates columns with their names. After all columns are created, they are added to the tableView received in the function as parameter and the list of objects is added to the table wrapped in an ObservableList. There is also a function for reloading all the data in the table but without creating the columns. Another useful method is the *public T showSelectedRow(TableView<T> tableView)* which returns the selected row from a table in form of an object. The controller classes get the data from the text fields and when the specific buttons are pressed, call the business logic layer to do the computations and return the desired results. In case of a failure, they catch the exceptions and display the PopUpWindow with the error message. This class creates and displays a window which cannot be ignored by the user. The user must close it before trying to do anything else in the application.

1. **Conclusions**

There definitely is room for improvement. For example, the updating of a product or a client could be modified such that only certain fields are updated, not necessarily all of them. Aside from this, I think that the application does what it should do in a pretty user friendly and safe way. I liked this assignment because I learned more about the reflection technique which seems daunting at first but gets easier after a little practice and is quite powerful and convenient.