Fundamental Programming Techniques

Assignment 3- Documentation

Order Management

Student: Moisa Oana Miruna

Group: 30422

1. Assignment objective

The objective of this assignment is to develop a Java application for managing client orders for a warehouse. This application must use a relational database for storing data related to clients, products and orders. The secondary objectives of this project are: understanding the requirements and how the application is supposed to work, designing the application, implementing the application (both the actual functionality and the graphical user interface). Each of these sub-objectives will be described in detail in the following paragraphs.

2. Problem analysis, modeling, scenarios, use cases

The functional requirements for this assignment are the following:

- The users should be allowed to choose what kind of operations they want to perform from a menu.
- The users should be allowed to perform the following operations both on the table of clients and on the table of products from the database: add element, delete element, edit element, view all elements.
- The user should be able to choose a client, select desired products and place an order.
- The application should update the tables from the relational database with any changes made by the user.
- The application should write in a text file the bill of the order.
- The application should handle bad inputs from the user.

The non-functional requirements of this problem are:

- The application should be easy to use, with a friendly interface.
- Except for the UI classes, the rest of the classes should contain at most 300 lines.
- The methods must be at most 30 lines long.
- The application should be structured in packages using a layered architecture.
- The classes must contain Javadoc comments.
- The application should use reflection techniques.

Use cases:

•**Use Case 1**: add a new client/product

Primary Actor: user

Main Success Scenario:

-The user inserts all the required information in the graphical user interface.

-The user clicks on the “Add client/product” button.

- The new client/product is added to the corresponding table in the database.

Alternative Sequence 1: Missing fields

-The user doesn’t fill in all the text fields

-The user is warned about the mistake

-The process starts again

Alternative Sequence 2: Incorrect email/quantity/price

-The user inserts an email address that is not valid or a negative quantity/price.

-The user is warned about the mistake

-The process starts again.

•**Use Case 2**: delete a client/product

Primary Actor: user

Main Success Scenario:

-The user inserts the name of the client/product in the graphical user interface.

-The user clicks on the “Delete client/product” button.

- The client/product is removed from the corresponding table in the database.

Alternative Sequence 1: Missing field

-The user doesn’t fill in the required field

-The user is warned about the mistake

-The process starts again

Alternative Sequence 2: Client/product doesn’t exist

-The user wants to delete a client/product that does not exist in the database

-The user is warned about the mistake

-The process starts again.

•**Use Case 3**: update a client/product

Primary Actor: user

Main Success Scenario:

-The user inserts all the required information in the graphical user interface.

-The user clicks on the “Update client/product” button.

- The client/product is updated in the corresponding table in the database.

Alternative Sequence 1: Missing fields

-The user doesn’t fill in all the text fields

-The user is warned about the mistake

-The process starts again

Alternative Sequence 2: Incorrect email/quantity/price

-The user inserts an email address that is not valid or a negative quantity/price.

-The user is warned about the mistake

-The process starts again.

Alternative Sequence 3: The client/product with that id doesn’t exist in the database

-The user wants to update a client/product which does not exist

-The user is warned about the mistake

-The process starts again.

•**Use Case 4**: place an order

Primary Actor: user

Main Success Scenario:

-The user selects a client and adds to the cart the desired products.

-The user clicks on the "Order" button.

- The order is stored in the database and a bill is generated.

Alternative Sequence 1: Missing client/product

-The user doesn't select a client and at least one product

-The user is warned about the mistake

-The process starts again

3. Design (design decisions, UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interfaces)

The application is designed using a layered architecture. It contains 6 packages: start, presentation, model, data access layer, connection, business layer. The start package consists of only one class, Main, which is used for starting the application. The connection package also consists of only one class called Connection Factory, which implements the connection of the java application to the database. The model package is the one that contains the classes mapped to the database tables: Client, Product and Orders. The dataAccessLayer package contains the classes containing the queries and the access to the database. All operations that the user wants to perform on the database have to be made through the classes from this package, being the lowest level in the hierarchy, just above the database connection. The package bussinessLayer contains the classes that encapsulate the application logic and use the methods defined in the DAO classes. And last but not least, the presentation package contains the UI related classes. This kind of layered architecture is a very good way of structuring an application that communicates with a database. The layers are placed as follows: at the lowest level we have the actual database, with the 3 tables, then above it we have the Connection Factory class that makes sure the connection is successful in order for the next layers to function. Next, we have the data access layer composed of the classes: AbstractDAO, ClientDAO, ProductDAO, OrderDAO and right over it the business logic layer consisting of the classes: ClientBLL, ProductBLL, OrderBLL. The last layer is the presentation layer composed of the classes: ClientsWindow, ProductWindow, Controller, OrdersWindow, ErrorWindow, TableSceneClient, TableSceneProduct.

In this application I have used reflection techniques to create a generic class that contains the methods for accessing the DB, called AbstractDAO. The classes ClientDAO, ProductDAO and OrderDAO inherit the methods from this generic class. The operations are defined on the specified generic type <T>. T can be any of the Client, Product or Orders classes, which are mapped to the database tables(the name of the table coincides with the name of the class and the name of the class fields are the same as the name of the table columns headers.

The user interface is simple and easy to use. The main window of the application is the menu, which lest the users decide if they want to perform operations in the Client Table, the Product Table or place an order. There is a separate window for all of these 3 options. The Client and Product windows are similar, offering the same operations: add, delete, update, find by id and view all. For the first 4 operations there are specific text fields that the user has to fill in in order to get the desired result, while for the view all operation the user has to click a button which opens another window that contains a TableView with all the rows from the corresponding table. As for the Order Window, it consists of 3 TableView objects, one for displaying all clients, one for displaying all products and one for displaying the products in the cart, selected by the user. Each row from the Clients and Products table have an additional column of + buttons, used to select the client/add the products to the cart. The Cart TableView has also an additional row of – buttons, if the user decides that they don't want a product anymore. As the user selects the products, the total sum is displayed in a text field.

4. Implementation

Client Class

This class is a model class and has the same name as the corresponding table from the database and the same fields as the columns of the table: a unique id, a name, an address and an email. In the table, this id is auto-incrementing. The methods are only getters and setters.

Product Class

This class is a model class and has the same name as the corresponding table from the database and the same fields as the columns of the table: a unique id, a name, a brand, a type, a price and a quantity. In the table, this id is auto-incrementing. The methods are only getters and setters. The fields were chosen based on the kind of products that will be stored in the database: make-up products.

Orders Class

This class is a model class and has the same name as the corresponding table from the database and the same fields as the columns of the table: a unique id of the row, an order id, client's id, the id of the product and the quantity of the ordered product. The methods are only getters and setters. In the Orders table from the database, the client id and product id are foreign keys from the Client and Product tables. All products selected in an order have the same order id. But the order id and product id tuple has to be unique, hence why I added an additional unique id for the whole row, which is auto-incrementing.

ConnectionFactory Class

This class handles the connection to the database. In this case, the connection to the database has to be unique, so I have used the Singleton Design Pattern. There are methods for creating the connection and 3 methods for closing the statement, the result set and the connection, which will be used after every operation on the database.

AbstractDAO Class

This is a generic class of type T, T being any of the classes Client, Product or Order. Here we have a few methods for generating the queries for specific operations such as select all query, select based on id query, delete query, insert query and update query. Then there is the findById() method which receives an id as a parameter and returns an object of type T if that object exists or null otherwise. The method insertElement() receives as a parameter an object of type T and returns the id of the inserted element. The deleteElement() method receives the id of the element to be deleted and deletes that row from the table if it exists. The updateElement() method receives an object of type T as parameter and updates all the fields of the row with that specific id. The findAll() method returns a list of objects of type T, which are all the elements from the table. The displayTable() method receives a list of objects of type T 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. In this method we define the columns of the table and modify the observable list of the table with the given objects. The createObjects() function transforms the resultSet generated by the findAll() or findById() functions into a list of objects.

ClientDAO Class

This class extends the AbstractDAO class and has all its methods specific for the Client.

ProductDAO Class

This class extends the AbstractDAO class and has all its methods specific for the Product.

OrderDAO Class

This class extends the AbstractDAO class and has all its methods specific for the Order.

ClientBLL Class

This is a business logic class that has all the methods that can be applied to the Client object, mainly calling methods from the ClientDAO and validating the data before inserting it into the database. For the client table, only the email address has to be validated and I have done that using a regex expression. The email address must end in "@yahoo.com" or "@gmail.com" and contain only letters from a to z, uppercase or lowercase, digits, dot or underscore.

ProductBLL Class

This is a business logic class that has all the methods that can be applied to the Product object, mainly calling methods from the ProductDAO and validating the data before inserting it into the database. The only validating that needs to be done in this case is making sure that the price is greater than zero and the quantity is greater or eqal than zero.

OrderBLL Class

This is a business logic class that has all the methods that can be applied to the Orders object, mainly calling methods from the OrderDAO and generating the id of the next order. The id of the next order is generated as follows: we call the findAll() method on the Orders table and get the last order id, to which we add 1, or if the table is empty the order id will be 1.

ClientsWindow Class

This is a UI class that allows the user to perform operations on the Clients table. The user can insert a new client in the table by filling in all the text fields required and clicking the "Add client" button. The user can also delete a client by entering its name in the text field and clicking the "Delete client button". Also, a client can be found (its information will be displayed in the available text fields that cannot be edited by the user) by its id, which is inserted by the user. Updating an existing client is also possible. The user has two options. He can introduce all fields of the client to be updated or he can introduce only the id, click on the "Show client" button, and the current information of that client will be displayed, which the user can modify and then click on "Update client" button". There is also a "View clients" button, which opens a new window that displays all the clients from the table.

ProductsWindow Class

This is also a UI class and is similar to the ClientsWindow class, but the operations are done on the Products table.

Controller Class

This is the main window of the application, which acts like a menu. It has 3 buttons, which open 3 different windows. All the other windows have "Back" buttons that take the user to this menu.

ErrorWindow Class

This window is used for displaying warnings to the user if some requirements are not met such as empty fields, invalid information, etc. The constructor of this class takes as a parameter a String, which is the message to be displayed on the window.

OrdersWindow Class

This UI class allows the user to place an order. It consists of 3 TableViews. The first one displays all clients from the Client table, using the generic method previously implemented. The second one displays all products from the Product table, using also the previously mentioned method. To both these tables, we add an extra column of buttons, using the addButtons() and addButtonsProductAdd() methods. When the user clicks a + button from the first table, the customer is selected and displayed in a text field. When the user selects products from the second table, they are added to the cart, so the 3rd Table view. To this table, we add an extra column of – buttons using the method addButtonsProductRemove(), which are used to remove products from the cart. When the user adds elements to the cart, the stock of that product is decremented, using the update method. When the "Order" button is clicked, the elements from the observable list of the cart table are processed and the order is entered into the database, a row for each product.

TableSceneClient Class

This UI class displays the Client table from the database, using a TableView.

TableSceneProductClass

This UI class displays the Product table from the database, using a TableView.

5. Conclusions

In conclusion, I believe that I have learned a lot of things from this assignment, as it was one that required a lot of work and introduced a few new concepts. I have worked with generics before but very

briefly, so I understood it a lot better now because this project relied heavily on it. Also, it was the first time that I have developed an application that works directly with a database and I have used the layered architecture design technique also for the first time, but I can clearly see its benefits. While working on this assignment, I have also learned how to write Javadoc comments. As for further improvements, I think I could make the user interface look more sophisticated. Also, I could maybe generate the bill as a pdf file instead of a text file.

6. **Bibliography**

ASSIGNMENT_3_SUPPORT_PRESENTATION.pptx