Order Management

Nadu Laura Andreea
Group 30421

Table of Contents

Assignment Objective

Main Objective

The main objective is to propose, design and implement an "Order Management" application for processing client orders for a warehouse. The application should use a relational database schema to store the information about the clients, products and orders.

Sub-Objectives

O1. Analyse the problem and identify the requirements
O2. Create the database for the application
O3. Design the order management application
O4. Implement the order management application
O5. Structure the application using a layered architecture
O6. Implement the User Interface for the main menu, the client menu, the product menu and the order menu
O7. Test the order management application
O8. For each order, create a bill.

Problem analysis, modelling and use cases

Analysing the problem

The task has been interpreted in the form of a managerial system of a warehouse. An employee should be able to manage all the data corresponding to the warehouse management: the products, the clients and the orders made by the client.

Usually, the information about the warehouse products, services and historic was stored into records. Nowadays, the information is stored into databases. The data for this application is stored inside a relational database which has minimum 3 tables: Client, Product and Order.

The relations between these tables should follow the relationship constraints:

- An order is specific to on and only one client, but a client may place more orders. This means that between Client and Order, there should be a ONE-TO-MANY relationship
- A client may does not have any direct association with the products
- An order may contain more products, but also a product may be associated to more than one order. This means that between Order and Product, there is a MANY-TO-MANY relationship. To solve this kind of relationship a new table will be used. The newly added table will be called OrderDetail and it will transform the MANY-TO-MANY relationship into two ONE-TO-MANY relationships

The Order table should contain details about:

- The client which made the order.
- The shipping address.
- The total amount spent on the order.

The Client table should contain details about:

- The first and last name of the client.
- An email address.
- A phone number of the client.

The Product table should contain details about:

- The name of the product.
- The quantity currently in stock.
- The current price of a unit piece of the product.

The OrderDetail should contain:

- The id referencing to an existing order.

- The id referencing to an existing product which was ordered.
- The quantity ordered.

Modelling the solution

The main objective of the modelling stage is to be able to create a representation of an Order Management System which would be able to fulfil the needs of managing a real-life warehouse.

As highlighted in the problem analysis, the main components which are included in the general term „Order Management" refer to the concepts of products, clients, orders and order details. All of these components will be represented as model classes, each of them having the appropriate attributes to mirror the columns of the tables.

Furthermore, it is important to take into consideration the fact that the entire system is created to be changed according to the current status and needs of the warehouse. These changes would be registered as CRUD operations on the database models, meaning addition/removal/update of products or client or addition of orders. The CRUD acronym stands for: Create, Read, Update, Delete. In order to apply to the database these operations, we would use SQL queries. For creating the queries, special Data Access Classes will be created.

The database will be modelled using the MySQL SQL Language with the help of the MySQL Workbench.

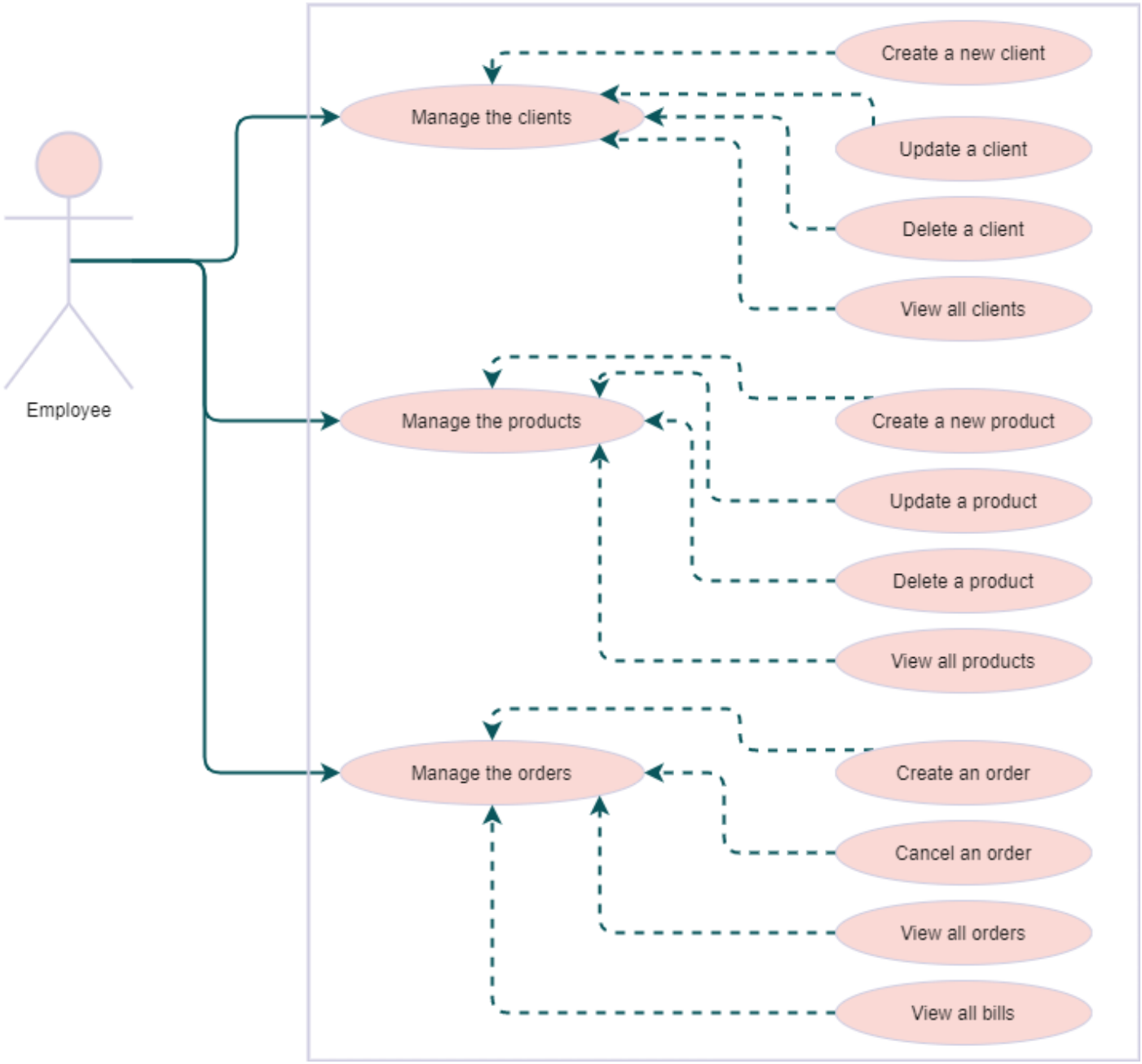
Functional requirements:

- The order management system should allow the user to select on which kind of item should he/she like to operate.
- The order management system should allow the user to select between the operations to be performed to an item
- The order management system should update the database each time a modification to an item is applied
- The order management system should automatically generate the bills for each system
- The order management system should display messages containing the successful or unsuccessful status of the operation performed

Non-functional requirements:

- The order management system should be intuitive and easy to use by any user, regardless of their knowledge about databases systems
- The order management system should verify the correctness of the input inserted by the user before modifying the contents of the database
- The order management system should offer the possibility to open the bills record folder

Use Case Diagram

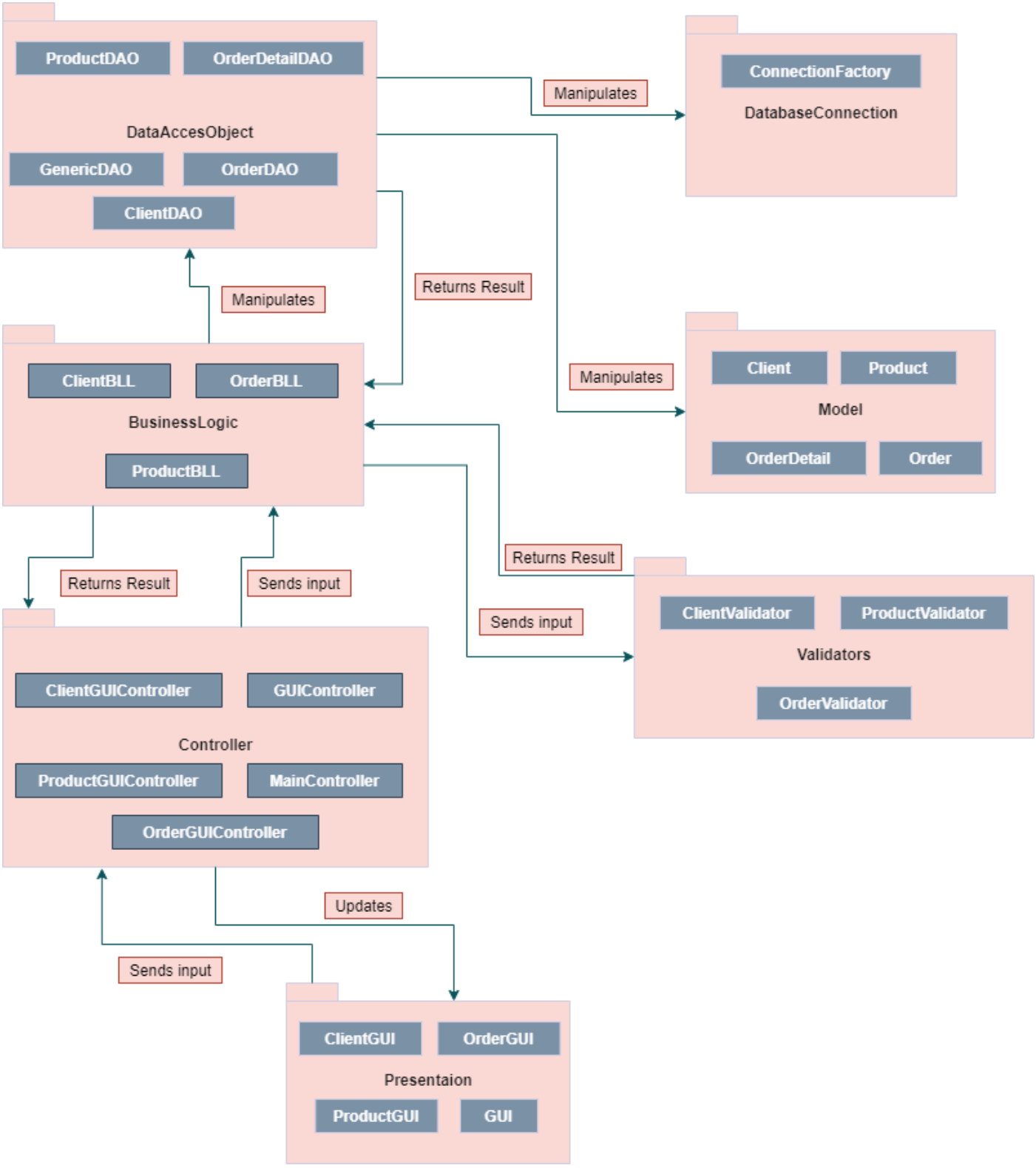


As shown in the diagram, on a successfully sequence of steps, the user should be able to manage the whole database system, including updating/creating/removing items as well as viewing all the contents of the tables.

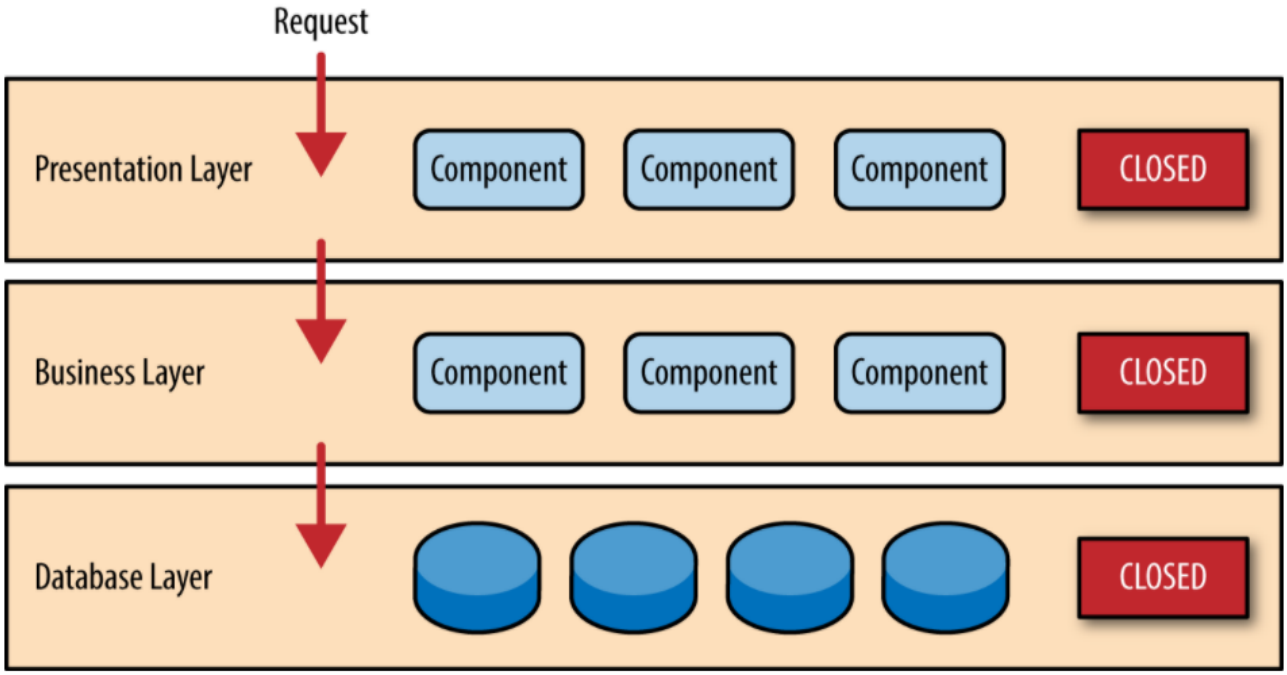
As an alternative case scenario, the user may also encounter error messages at any step of the usage displaying information about the error occurrence.

Design Decisions

UML Package Diagram

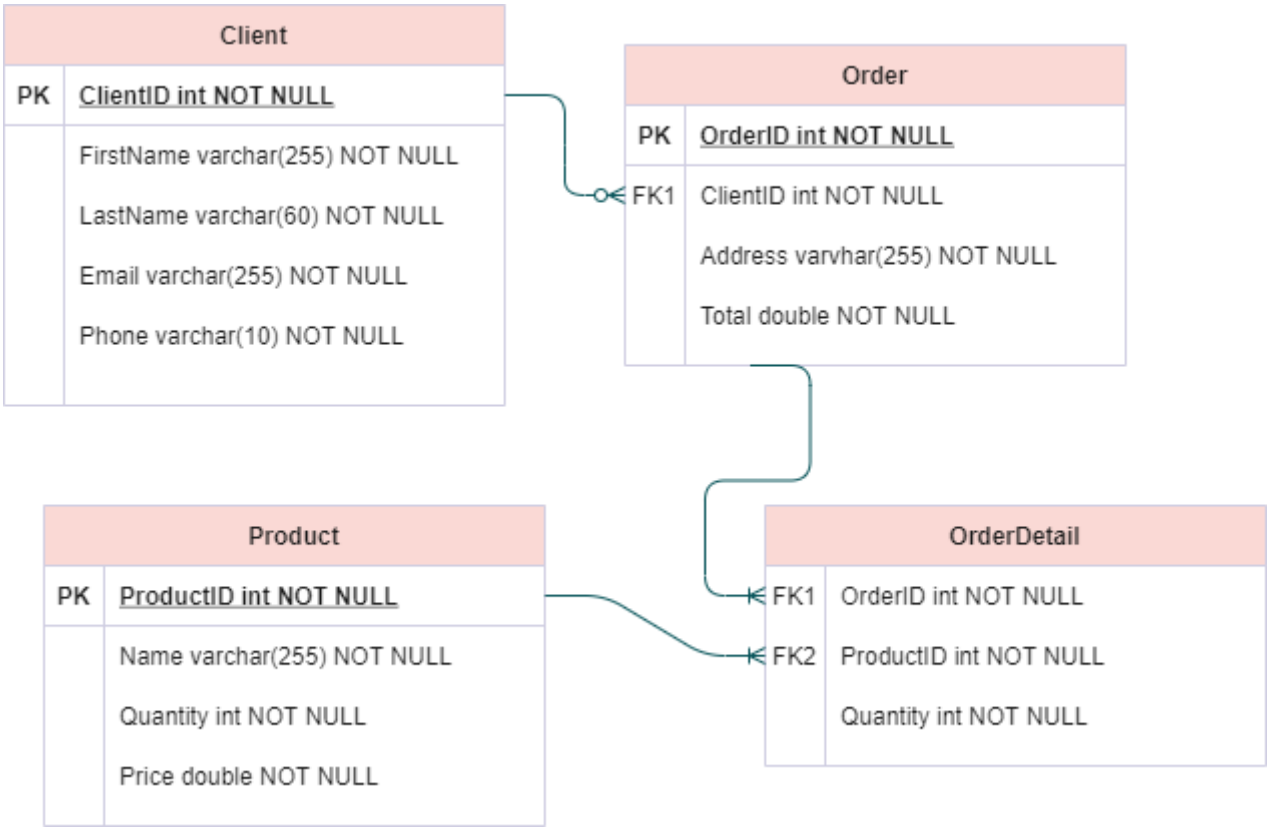


For this package design I tried to plan a layered architectural design. This design is based on building up information necessary to fulfil a request from the first layer to the last one, as described in the picture below. As we can see, the user request is sent to the presentation layer, through the business layer, ending up into the database layer.



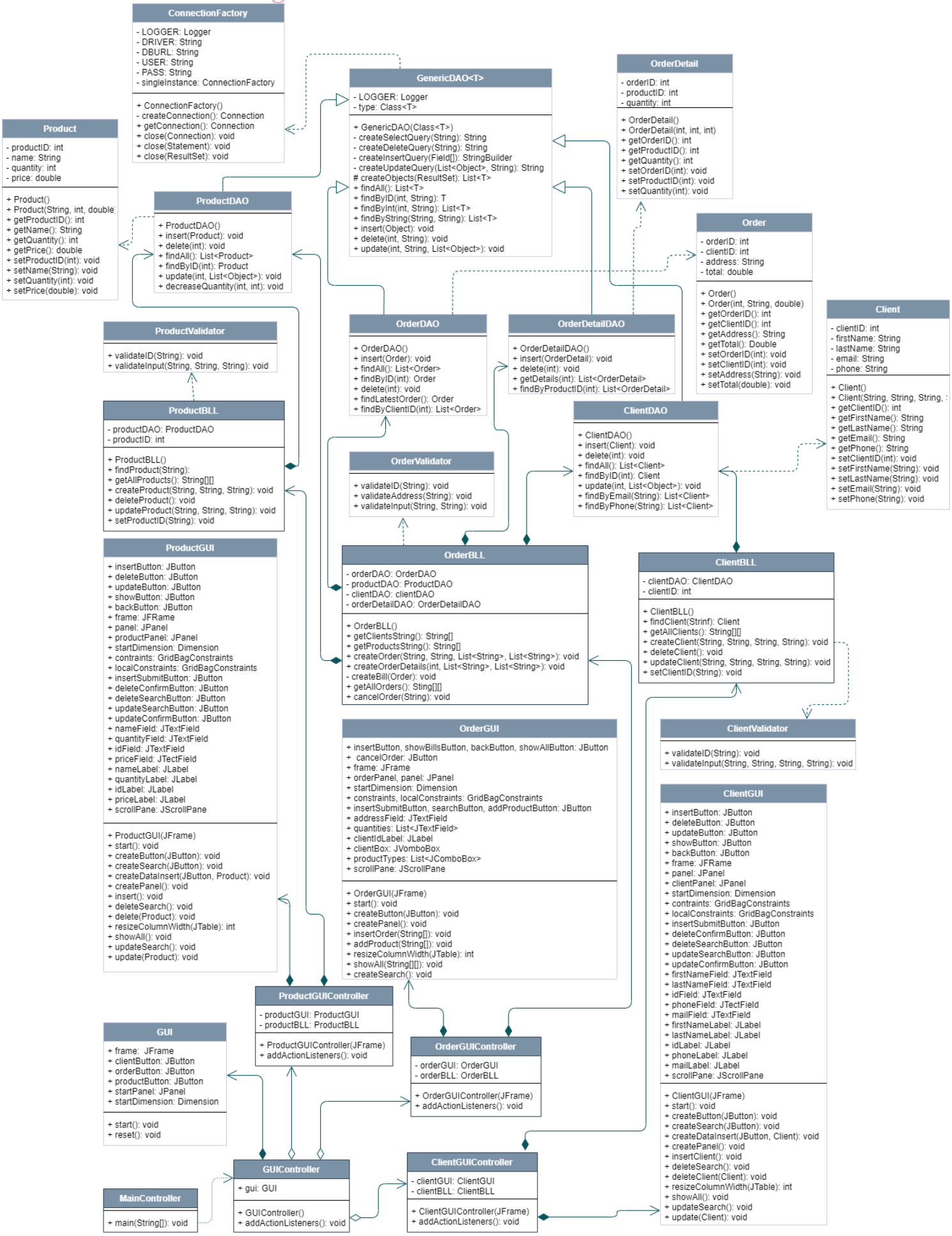
For the order management system the request corresponds to a query request reveived first by the Presentation package. It then send the information received to the Controller package which then transmits it to one of the corresponding components of the BusinessLogicLayer package. This packages adds its information about the request (for example, whether the request is valid or not) and sends it further to the DataAccessObject package which implements the requested query.

Database ER Diagram



Order Managements

UML Class Diagram



Data Structures

The main data structure used is the **ArrayList**. ArrayList, a member of the Java Collection Framework, was used mostly to store the list of servers or the randomly generated tasks. It is a dynamic data structure whose size changes based on the number of elements stored. By using these data structures, access was granted to methods that allow access to any element in the list, which can be just returned or removed, or to methods that return the size of the array so we won't have to store it in a special variable.

Class Design

Non-user Interface Classes

The classes will follow the Layered Architecture style on three branches: the client branch, the order branch and the product branch. Each will keep to its branch, but there may also be intersections

The Client Branch

When the user chooses to manage the clients of the warehouse, he/she will enter the client branch. Here, based on the specified architecture, the request made by the user is analysed, and at each layer, the response becomes more clearly shaped. The first layer is the presentation layer. Here is the user interface which takes in the request from the user. This request is sent to the Controller layer which decides to call the Business Layer class corresponding to the clients.

In this layer, the input will be validated by a "client" validator class and, if everything fits the constraints, then the request enters the database access layer, responsible for connecting to the database to execute the request of the user.

The Product Branch

When the user chooses to manage the products of the warehouse, he/she will enter the product branch. Here, based on the specified architecture, the request made by the user is analysed, and at each layer, the response becomes more clearly shaped. The first layer is the presentation layer. Here is the user interface which takes in the request from the user. This request is sent to the Controller layer which decides to call the Business Layer class corresponding to the products.

In this layer, the input will be validated by a "product" validator class and, if everything fits the constraints, then the request enters the database access layer, responsible for connecting to the database to execute the request of the user.

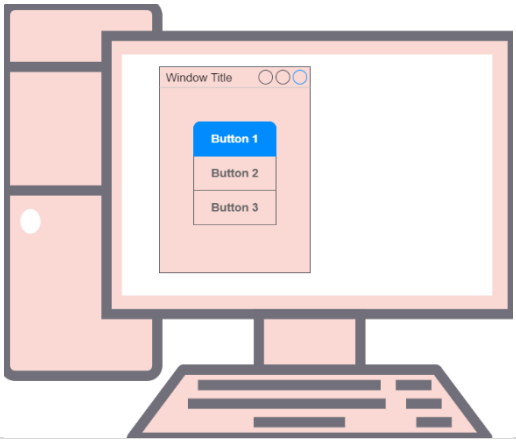
The Order Branch

When the user chooses to manage the orders of the warehouse, he/she will enter the order branch. Here, based on the specified architecture, the request made by the user is analysed, and at each layer, the response becomes more clearly shaped. The first layer is the presentation layer. Here is the user interface which takes in the request from the user. This request is sent to the Controller layer which decides to call the Business Layer class corresponding to the orders.

In this layer, the input will be validated by an "order" validator class and, if everything fits the constraints, then the request enters the database access layer, responsible for connecting to the database to execute the request of the user.

Of course, even if the three branches are more or less independent, as we have seen from both the database diagram and the class diagram, the clients, products and orders are all related to each other, meaning that we may encounter, for example, a Product in an Order, because this is specific for the definition of an order.

Until the database access layer is met is harder for the three branches to converge to a more general method that could apply to all of them. But the database access consists of defining SQL queries and executing them. As the SQL queries have a general format that should be followed, creating an intersection point of the three branches is possible by applying correctly the reflection technique.



User Interface Classes

The User interface will be made out of 4 main components.

The main menu having three buttons to choose between the types of items to manage.

Then, the other 3 components should contain buttons and fields specific to each type of item and to each type of operation the user would like to apply.

Implementation

Even though from the class diagram there seem to be many classes involved in the project, many of them are the same and have the same scopes so I will skip eaxplaining each class individually where there is no particular reason to do so.

ConnectionFactory

When speaking about a database application, the main question that rises is “How do we connect to the database?”. This is the class that handles this step. The application is connected to a MySQL database named “schooldb” through the root user. The root account of the database suggested does not have a password.

The class implementation is the one proposed by the teachers in the Assignment Presentation.

Client, Product, Order and OrderDetail

These are the Model classes. They are the classes that mimic the database table perfectly. The attribute fields of each class correspond to the columns of the tables with the same name. They have the same name and the same primitive type.

Besides these, the classes contain only getters and setters for the class attribute fields.

GenericDAO<T>

This is a generic class that implements through reflection the general form of the queries. Because the queries are almost all the same, this class implements the common queries. The generic parameter takes the place of a specific class Model so that, the methods implemented by this class may be used by any other subclass or by any instantiation of type T.

The queries are firstly constructed as a string, then the connection to the database is applied and the query is executed.

The SELECT query

```
private String createSelectQuery(String field) {
    String query = "SELECT * FROM schooldb." + type.getSimpleName();
    if (field != null)
        return query + " WHERE " + field + " =?";
    return query;
}
```

The query takes in a String, but it may be optional. If the field given is null, then the query has the effect of selecting all of the elements of the given field. In case a condition for the query exists, the field will contain the name of the field to which the condition is applied. The value for the field condition is given by the “?” sign. This value will be set in the method that calls the SELECT query creating method.

The DELETE query

```
private String createDeleteQuery(String field) {
    return "DELETE FROM schooldb." + type.getSimpleName() + " WHERE " + field + " =?";
}
```

The elements to be deleted are seleced based on a condition imposed on the field column of the database table. This condition is mandatory for the query to be executed correctly.

The INSERT query

The insert query is also built in a separate method, but only half of it is created. The query creation stops at the moment the values should be inserted. These will be added in the insert method that called the contruction.

The UPDATE query

The method to create the update query makes use of both the values and the field list. A query updates the fields with the new values only for those items that respects the set condition upon the given field. This method is a good example for the use of reflection to generalize the update query for all types of models.

```
private String createUpdateQuery(List<Object> values, String idField) {
    Field[] fields = type.getDeclaredFields();

    String query = "UPDATE schooldb." + type.getSimpleName() + " SET ";
    try {
        for (int i = 1; i < fields.length - 1; i++) {
            fields[i].setAccessible(true);
            query = query + fields[i].getName() + " = ";
            if (values.get(i - 1).getClass().getSimpleName().equals("String"))
                query = query + "\"" + values.get(i - 1) + "\", ";
            else query = query + values.get(i - 1) + ", ";
        }
        fields[fields.length - 1].setAccessible(true);
        query = query + fields[fields.length - 1].getName() + " = ";
        if (values.get(values.size() - 1).getClass().getSimpleName().equals("String"))
            query = query + "\"" + values.get(values.size() - 1) + "\"";
        else query = query + values.get(values.size() - 1);
        query = query + " WHERE " + idField + " =?";
    } catch (Exception e) {
        e.printStackTrace();
    }
    return query;
}
```

The other methods are:

- findAll which calls a SELECT query without condition
- findByID, findByInt and findByString which call a SELECT query with a specific field condition
- update, delete and insert which call the specific types of queries

ClientDAO, ProductDAO, OrderDAO and OrderDetailDAO

These classes extend the GenericDAO class and are called to perform the CRUD operations on each table specified in the class name. They are more of a filter between the business logic and the genericDAO. They only call the super methods in the parent class with the specific fields of the child class.

I chose to implement them to make the code easier to understand. That is because it is easier to understand a clientDAO rather than a genericDAO<Client>.

ClientBLL, ProductBLL and OrderBLL

These classes form the business logic of the application. Here the application verifies the input of the request and decides which CRUD operation should be applied.

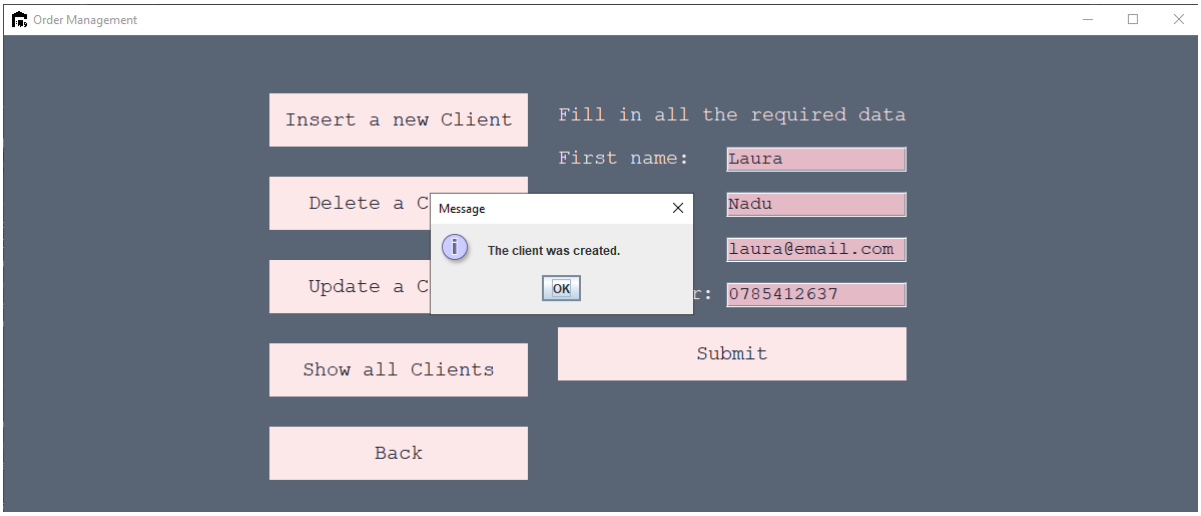
Controllers and GUIs

The controllers are there to control the action performed when pressing a button in the Graphical User Interfaces.

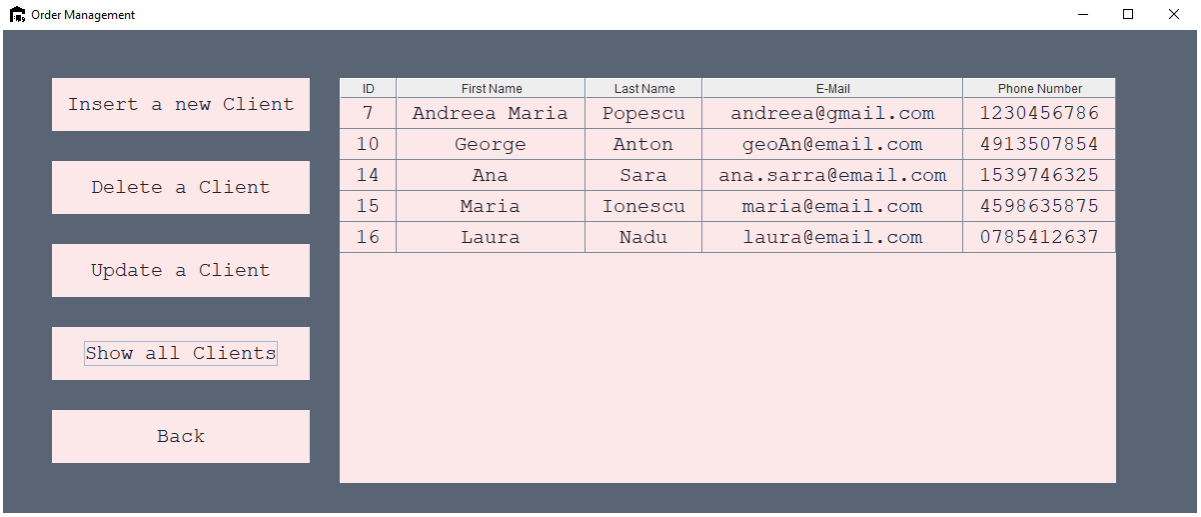
Testing

Client Management

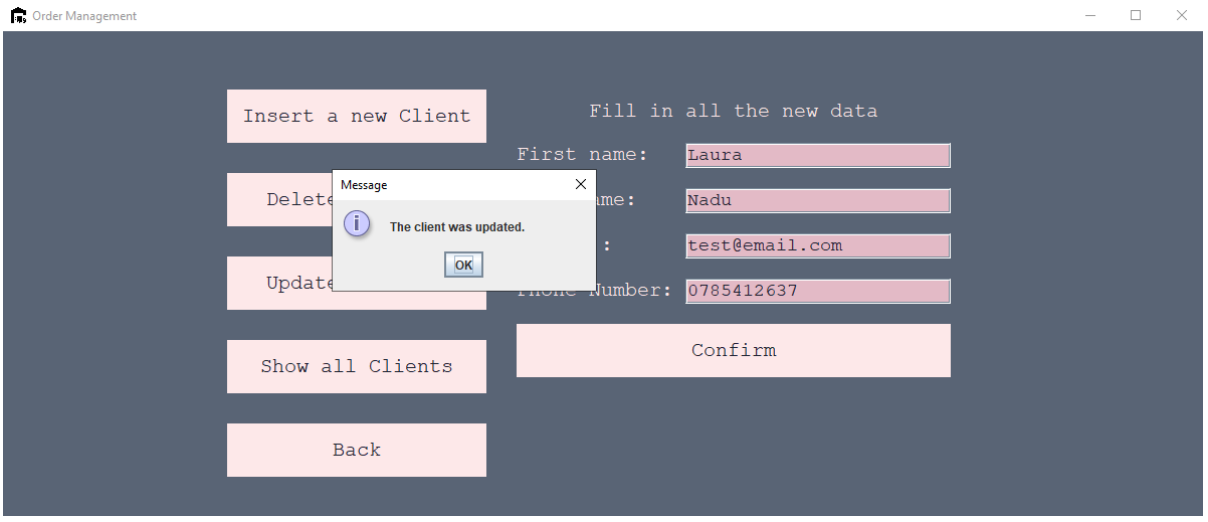
Inserting a new Client:



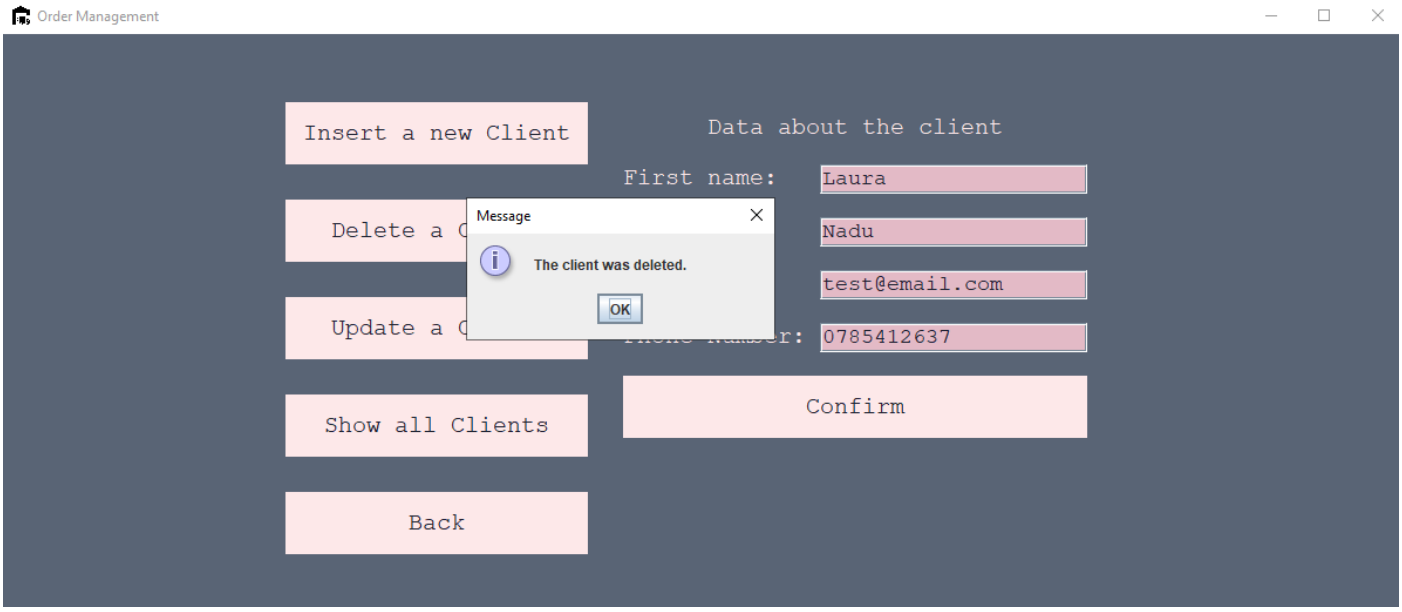
Displaying all the clients:



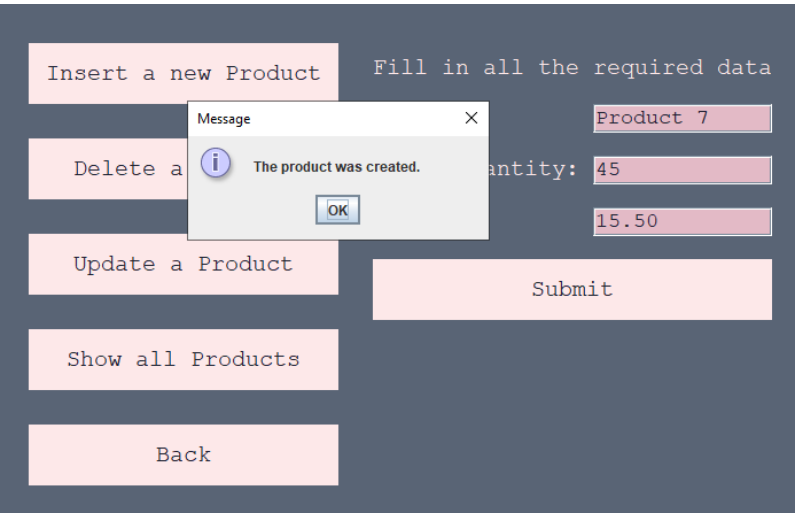
Update a client:

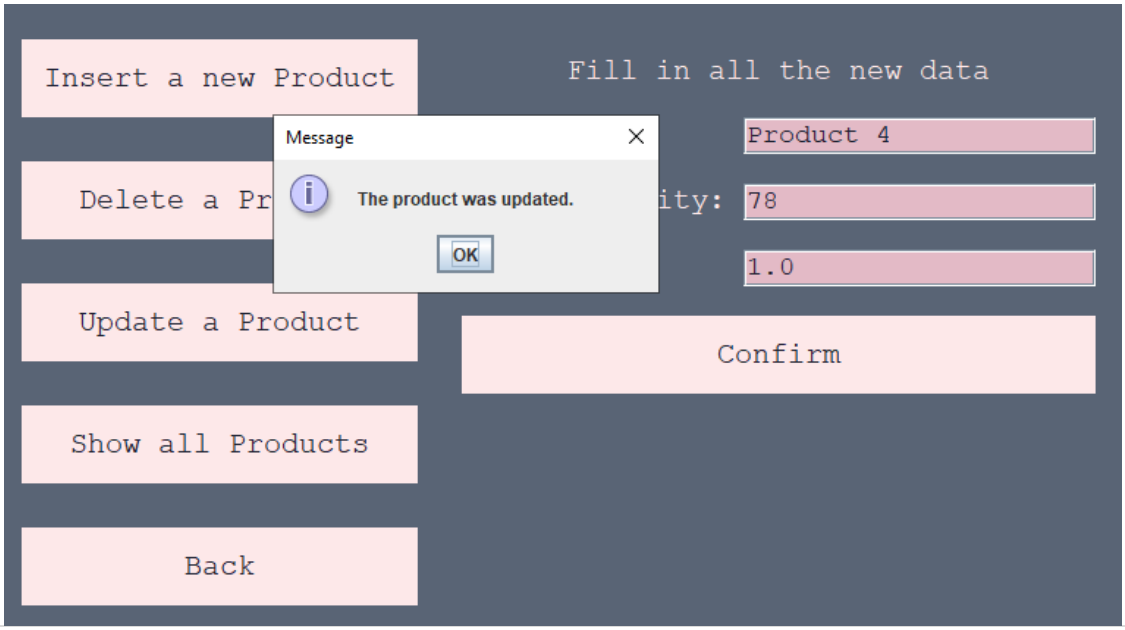


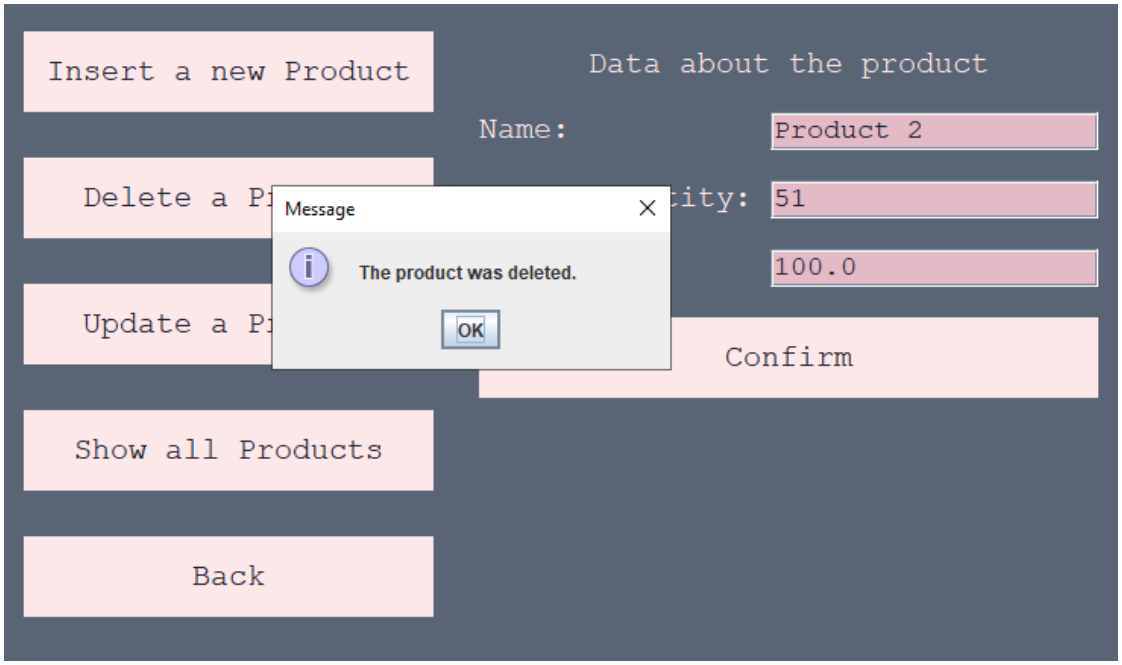
Deleting a Client:



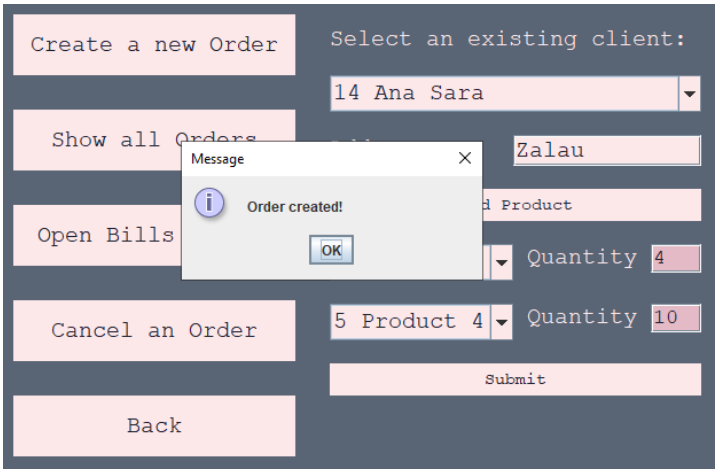
Product Management

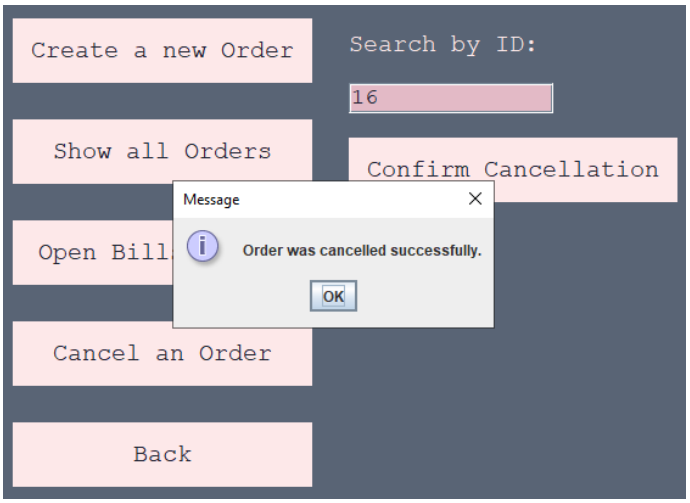






Order Management





Conclusions

As a final note for this assignment, I may say I found it a bit tricky at first but, I think I managed to solve the problems pretty well. I managed to connect to a database and implement the most basic CRUD operations.

The assignment specified to use the JTable for showing the items in the user interface so I couldn't try the JavaFX Collection, but there is another assignment coming so I hope I will get a chance to try it.

Regarding the further updates, I would add a scroll pane for the order creation as the user may enter how many orders he/she wishes.

Bibliography

Layered Architecture

- O'Reilly
- Layer, Wikipedia
- Kavin Kumar
- The Tech Cave

Reflection Technique

- Reflective Programming, Wikipedia
- Using Java Reflection, Oracle
- Reflection Basics, Kody Simpson
- Java Reflection Tutorial, Derek Banas

UML Diagrams

- Visibility in UML
- Package Diagram
- Use Case Diagram
- Draw.Io