

# Assignment 3 Documentation

## Orders Management System

Grama Mălina Bianca

Group 30423

Teaching Assistant: Antal Marcel

## Table of Contents

1. Assignment Objectives.....	3
a. Primary Objective.....	3
b. Secondary Objectives .....	3
2. Problem Analysis .....	3
3. Design.....	6
4. Implementation.....	7
a. Back-End .....	7
b. Front-End.....	9
5. Conclusions and Further Development .....	15
6. Bibliography .....	15

## 1. Assignment Objectives

The objective of the order management system is to process client orders for a warehouse. The application will work by using relational databases to store the information for products available in the warehouse, clients and details regarding the orders.

### a. Primary Objective

The primary objective of this third assignment is to propose, design and implement a system that is able to simulate an application that can store and edit the information regarding the clients, products and orders.

### b. Secondary Objectives

The secondary objectives of the assignment that I followed while implementing my solution would be the following:

- **Development of use cases and scenarios** – we need to think about how the user will interact with the application and what scenarios could arise. This will be described in detail in the **problem analysis** section of this documentation.
- **Choosing the data structures** – we need to think about how the user input will be stored and managed by our application, so that it will generate the required result. This will be briefly mentioned in the **design** section of the documentation.
- **Division of the solution into classes** – we need to think about how we will model the real-life “objects” that are the clients, products and orders into an object-oriented programming object. This will be mentioned in the **design** section of this documentation and detailed in the **implementation** section.
- **Declaration of variables and methods** into aforementioned classes – after we figure out the classes that we need, we also need to think about the variables and methods that will go into these classes. Most of the times we start from the project specification, looking for nouns, which become possible candidate classes, and verbs that could play the role of class methods. This will be detailed in the **implementation** section.
- **Development of the algorithms** necessary for the simulation – we need to think about how we will implement the functionality of the application. This objective will be detailed in the **implementation** section of the documentation.
- **Implementation of the solution** – we will need to describe more specifically how each class used in the project. Also, we will describe the implementation of the user interface. This will be detailed in the **implementation** section of this documentation.
- **Testing** – detailed in the **testing and results** section.
- **Debugging.**
- **More testing.**

## 2. Problem Analysis

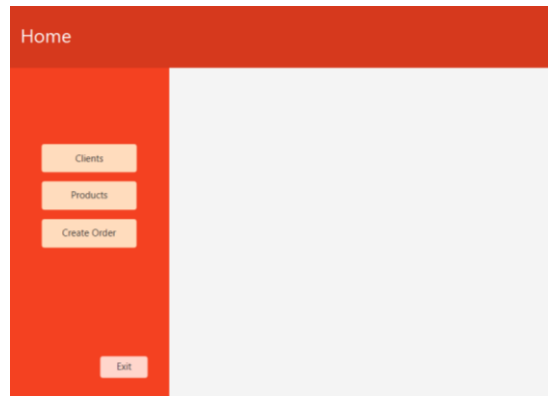
By analyzing the problem, we refer to a first abstract set of operations and properties through which we try to detect possible features and behaviors of unknown processes. Object-oriented programming offers us a clear advantage here, precisely because it allows us to tackle the problem from a higher level, without being constrained, to such an extent, by the technical characteristics.

Below we will exemplify three use cases for the order management system:

### **Use Case Name: Order Management System – Clients Action**

*Primary Actor:* User

*Triggers:* The user indicated that he wants to perform one of the three available operations in the Order Management System: to see and edit the data available for the clients, by clicking the “Clients” button on the homepage.



*Preconditions:* The user has clicked the “Refresh Table” button so that the information for the existing clients will be displayed on the table.

*Normal Flow:*

- The user selects an existing client by clicking on the data on the table.
- The user inputs the information that he wants to add/insert/delete from the database.
- The user clicks on one of the three buttons available to perform the operations on the database: Add, Edit or Delete.
- The Order Management System performs the selected operation, modifies the database, and the updated table will be visible on the screen.
- The user can also choose to Search for an existing client by typing its ID in the search bar and pressing the “Search” button. A new window will appear, showing all of the information available for the searched client.

*Alternate Flows:*

- The user inserts text that does not pass the validation tests: an incorrect e-mail address, an incorrect telephone number etc.
- The user presses one of the available buttons.
- The Order Management System will not let incorrect data to be added into the database, so nothing will happen.
- The scenario returns to the first step.

### **Use Case Name: Order Management System – Products Action**

*Primary Actor:* User

*Triggers:* The user indicated that he wants to perform one of the three available operations in the Order Management System: to see and edit the data available for the products, by clicking the “Products” button on the homepage.

*Preconditions:* The user has clicked the “Refresh Table” button so that the information for the existing products will be displayed on the table.

*Normal Flow:*

- The user selects an existing product by clicking on the data on the table.
- The user inputs the information that he wants to add/insert/delete from the database.
- The user clicks on one of the three buttons available to perform the operations on the database: Add, Edit or Delete.
- The Order Management System performs the selected operation, modifies the database, and the updated table will be visible on the screen.

- The user can also choose to Search for an existing product by typing its ID in the search bar and pressing the “Search” button. A new window will appear, showing all of the information available for the searched product.

*Alternate Flows:*

- The user inserts incorrect text in the available TextFields.
- The user presses one of the available buttons.
- The Order Management System will not let incorrect data to be added into the database, so nothing will happen.
- The scenario returns to the first step.

**Use Case Name: Order Management System – Orders Action**

*Primary Actor:* User

*Triggers:* The user indicated that he wants to perform one of the three available operations in the Order Management System: to place an order for a product, by clicking the “Place Order” button on the homepage.

*Normal Flow:*

- The user selects an existing product by clicking on the ChoiceBox on the interface and clicking again on the name of the desired product.
- The user selects an existing client by clicking on the ChoiceBox on the interface and clicking again on the name of the desired client.
- The user inputs the quantity that he wants to order in the specified TextField.
- The user clicks on the “Checkout” button. The total price of the order will be shown.
- The user clicks on the “Place Order” button.
- The Order Management System will add the information regarding the order in the database.
- The user clicks on the “Open Bill” button that has showed up after the placing of the order, and the Order Management System will open a pdf file with the bill that contains information about the order.

*Alternate Flows:*

- The user inserts text that does not pass the validation tests: an incorrect value for the quantity of the product to be ordered.
- The user presses the “Place Order” button.
- The Order Management System will not let incorrect data to be added into the database, so nothing will happen.
- The scenario returns to the first step.

We need to define a set of functional and non-functional requirements, to ensure that we cover all of the cases and provide the functionality of the application.

Some of the functional requirements could be:

- The order management system should allow users to pick one of the actions available in the app.
- The order management system should allow users to insert, modify and delete data for the clients and products.
- The order management system should allow users to place orders, and provide the specific bill.
- The order management system should check if the input of the user has the correct syntax, displaying an error message in the eventuality that the user typed illegal inputs.
- The order management system should perform correctly the selected operations with the provided input.
- The order management system should update the database.

Some of the non-functional requirements of the application:

- The order management system should be easy to use and intuitive.
- The results should be visible for the user after each selected operation.

### 3. Design

We try to think of the solution in a bottom-up manner, dividing the components of the problem into smaller pieces. As we advance to lower levels, the complexity of the problem increases.

Because we are working with a database in the implementation of this application, it is clear from the start that we will need a class that will model the information for each table in the database. We will place these classes in the “**Model**” package, and define the variables with the same name and type as the table columns. These classes are **Client**, **Product** and **Orders**.

Working with a database requires another class that will handle the connection. This class is called **ConnectionFactory** and is placed in the “**DataAccess**” package.

For isolating the application/business layer from the persistence layer (relational database), we will need specific classes called Data Access Objects (DAO). The functionality of these classes is to hide from the application all the complexities involved in performing CRUD operations in the underlying storage mechanism. We will use an abstract DAO class called **AbstractDAO** that will perform the insertion, update and deletion in all the tables of the database. We declare then specific DAO classes for each table: **ClientDAO**, **ProductDAO** and **OrderDAO**, that will extend the parent class AbstractDAO with added functionalities, like finding a record by id or name. These classes can be found in the “**DAO**” package.

To encapsulate the application logic, we will declare a “**BLL**” package. This package contains validation classes that are used for checking if the data typed in by the user in the user interface is correct, and a BLL class for all of the database tables, that will perform the necessary queries: **ClientBLL**, **ProductBLL** and **OrderBLL**. The validation classes contained in the “**Vals**” package are as follows: an interface named **Val** and classes: **EmailValidator**, **PhoneNumberValidator**, **PriceValidator**, **StockValidator**, **QuantityValidator**.

Using an architectural pattern, we know that we need four types of classes in our design: **Presentation Layer** that contains the classes defining the user interface, **Business Logic** that contains the classes implementing the functionality of the application, the **Data Access** layer that contains the classes that contain the queries and the database connection, and finally, the **Model** that contains the classes mapped to the database table.

Regarding the Graphical User Interface, we will use JavaFX to implement the Order Management System. Therefore, we will need Controller classes that will handle the functionality of the elements placed in the user interface, an **App** class that will be the driver of our application and will hold the main method, and fxml files that will contain information about our GUI and will link the GUI with the Controller classes. Because the User Interface contains 4 windows (a homepage window, a clients window, a products window and an orders window), we need 4 Controller classes (HomeController, ClientController, ProductController and OrderController) and 4 fxml files (home.fxml, client.fxml, product.fxml, order.fxml).

Data Access	Business Logic	Presentation Layer	Model
ConnectionFactory.java	ClientBLL.java	HomeController.java	Client.java
AbstractDAO.java	ProductBLL.java	ClientController.java	Product.java
ClientDAO.java	OrderBLL.java	ProductController.java	Orders.java
ProductDAO.java	Val.java	OrderController.java	
OrderDAO.java	EmailValidator.java	App.java	
	PhoneNumberValidator.java	home.fxml	
	PriceValidator.java	client.fxml	
	StockValidator.java	product.fxml	
	QuantityValidator.java	order.fxml	

## 4. Implementation

### a. Back-End

Now that we have decided what classes are needed for our project, let's talk briefly about the main methods of these Java Classes.

We start with the classes that model the tables in the database: **Client**, **Product** and **Order**. These classes only require constructors, getters and setters for the variables, and the variables should match with the columns of the tables. With that being said, the **Client** class has the following variables: **int idClient**, representing the primary key of the table, **String name** with the client's name, **String email** with the client's e-mail address, and **String phoneNumber**, holding the telephone number of the client. The **Products** class is similar, having the variables **int idProduct** representing the primary key of the table, **String name**, **int price** and **int stock**, representing the number of product available in the warehouse for purchase. The **Orders** class has the variables: **int idOrder** (primary key), **int idClient** (foreign key representing the client that placed the order), **int idProduct** (foreign key representing the product that was ordered), and **int quantity**.

The **DataFactory** class has a few static final variables that help with the database connection:

- **Logger LOGGER**
- **String DRIVER**
- **String DBURL** - containing the url of the database
- **String USER**
- **String PASSWORD**

The methods of the **DataFactory** class described briefly:

- **Connection createConnection()** – creates the connection to the database holding the necessary information
- **static Connection getConnection()** – returns the connection
- **static void close(Connection connection)** – closing the connection to the database
- **static void close(Statement statement)** – closing the statement
- **static void close(ResultSet resultSet)** – closing the result set
- **static ObservableList<Client> getDataClients()** - returns an observable list made of the records from the client table, used for updating the tableView in the GUI
- **public static ObservableList<Product> getDataProducts()** – returns an observable list made of the records from the product table, used for updating the tableView in the GUI

The DAO methods have the Logger as a variable, and the specific DAO classes have some SQL query statements strings specific for the tables.

The **AbstractDAO** methods are briefly explained bellow:

- **`static int insert(Object object)`** – inserts an object in its specific table in the database, and returns an integer representing the generated id of the inserted object. The steps of executing the ADD in a table are the following: we get the database connection, the name of the table (from the name of the object's class), and the fields of the table. We declare a `StringBuilder` with the String representing the SQL query, and we append the fields declared in the object's class to the query. Finally, we execute the resulted query and we close the connection and the insert Statement. The process is similar to the update and delete methods, so I will not explain it again. The SQL statement generated by the method has the following syntax: *INSERT INTO assignment3.(table name) (list with declared fields) VALUES (list with the variables of the object parameter)*
- **`static int update(Object object)`** – updates the table with the fields of the specified object and returns an integer representing the id of the updated object. The SQL query looks something like this: *UPDATE assignment3.(table name) SET (declared fields) = (object fields) WHERE idObject = (id of the object parameter)*. We can only update the columns that are not primary keys
- **`static void delete(Object object)`** – deletes a record in the table specified by the object. The SQL query generated by the method looks like this: *DELETE FROM assignment3.(table name) WHERE idObject = (id of the object parameter)*.

The **ClientDAO** class has the methods:

- **`static ArrayList<Client> returnClients()`** – returns an array list of Client objects made of the records from the client table
- **`static Client findById(int clientID)`** - searches for the client with the specified ID and returns a Client object with the specific ID
- **`static Client findByName(String name)`** - searches for the client with the specified name and returns a Client object

The **ProductDAO** class has the methods:

- **`static ArrayList<Product> returnProducts()`** – returns an array list of Product objects made of the records from the product table
- **`static Product findById(int productID)`** - searches for the product with the specified ID and returns a Product object with the specific ID
- **`static Product findByName(String name)`** - searches for the product with the specified name and returns a Product object

The **OrderDAO** class has the methods:

- **`static Orders findById(int orderID)`** - searches for the order with the specified ID and returns an Orders object with the specific ID

The validator classes are responsible with validating the user input so that the information inserted in the database is correct. They all implement the interface **Val** with the method **`void validate(T t)`**. The **EmailValidator** class uses a regex pattern for validating the e-mail string. The same with the **PhoneNumberValidator** class. The classes **PriceValidator** and **StockValidator** verify if the integer to be inserted into the database is bigger or equal to 0, and the **QuantityValidator** class verifies if the quantity to be inserted into the database is strictly bigger than zero.

The **ClientBLL** class implements the functionality of the operations made on the client table. Its methods are:

- **`ArrayList<Client> listClients()`** - returns an array list of Client objects made of the records from the client table by calling the `returnClients()` method in the ClientDAO class
- **`Client findClientById(int id)`** – calls the `findById(id)` method in the ClientDAO class and throws an exception if the client was not found in the table
- **`int insertClient(Client client)`** – calls the `insert()` method in the ClientDAO class
- **`int updateClient(Client client)`** - calls the `update()` method in the ClientDAO class
- **`void deleteClient(Client client)`** - calls the `delete()` method in the ClientDAO class



The **ProductBLL** class implements the functionality of the operations made on the product table. Its methods are:

- **`ArrayList<Product> listProducts()`** - returns an array list of Product objects made of the records from the product table by calling the `returnProducts()` method in the ProductDAO class
- **`Product findProductById(int id)`** - calls the `findById(id)` method in the ProductDAO class and throws an exception if the product was not found in the table
- **`int insertProduct(Product product)`** - calls the `insert()` method in the ProductDAO class
- **`int updateProduct(Product product)`** - calls the `update()` method in the ProductDAO class
- **`void deleteProduct(Product product)`** - calls the `delete()` method in the ProductDAO class

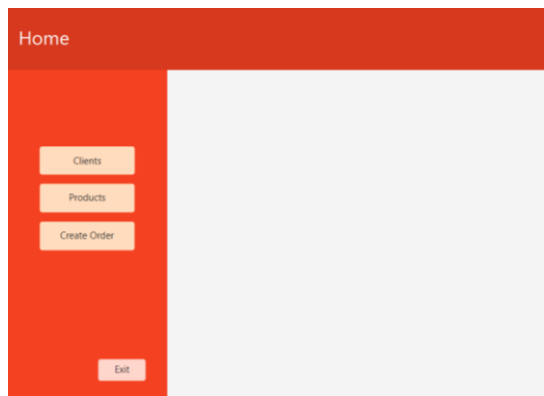
The **OrderBLL** class implements the functionality of the operations made on the orders table. Its methods are:

- **`Orders findOrderById(int id)`** - calls the `findById(id)` method in the OrderDAO class and throws an exception if the order was not found in the table
- **`void createOrder(Orders orders)`** - calls the `insert()` method in the OrderDAO class

### b. Front-End

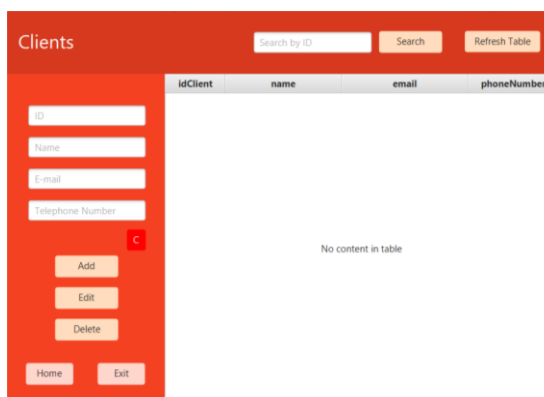
I implemented the Graphical User Interface using JavaFX and SceneBuilder.

The first page of the interface is a homepage, letting the user pick which window he wants to see next: Clients, Products or Create Order window. The homepage looks like this:

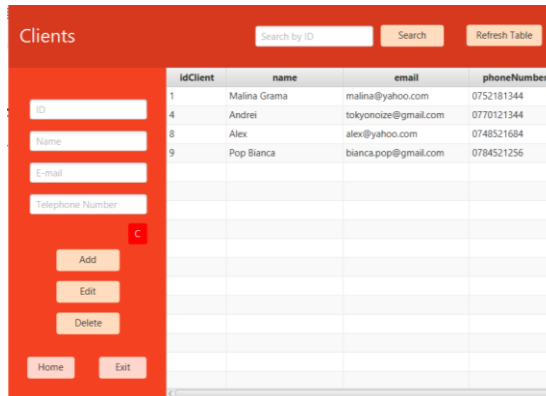


The Exit button terminates the application, and is available in all windows of the application.

If the user clicks the Clients button, a window in which the user can perform user operations will appear. The page look like this:

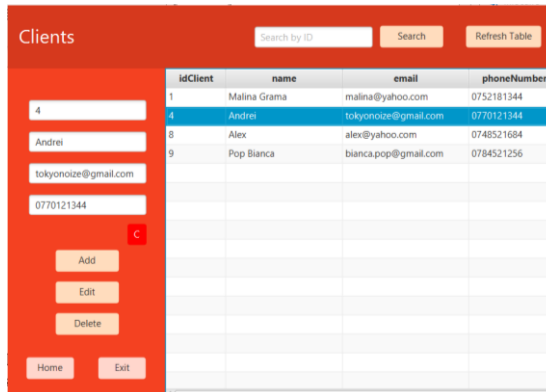


If the user clicks on the Refresh Table button, the table with the clients will be filled with the data from the database table client.



idClient	name	email	phoneNumber
1	Malina Grama	malina@yahoo.com	0752181344
4	Andrei	tokyonoize@gmail.com	0770121344
8	Alex	alex@yahoo.com	0748521684
9	Pop Bianca	bianca.pop@gmail.com	0784521256

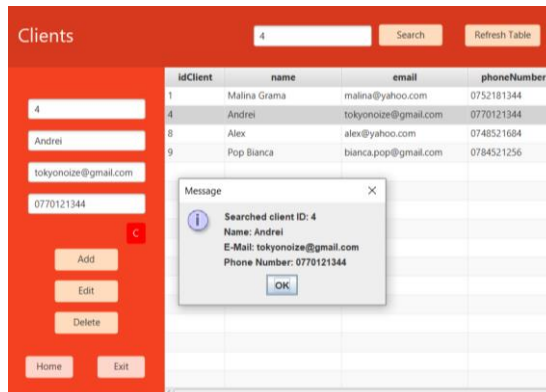
The table will be updated every time an operation is made, but I implemented the Refresh Table button to be there in case it does not refresh automatically. The user can then click on a record from the table, and the TextFields on the left of the page will be filled with the data from the table. I implemented this so that it will be easier for the user to edit an already existing record. If the C red button is pressed, the TextFields will be cleared, so that the user doesn't have to delete them "by hand".



idClient	name	email	phoneNumber
1	Malina Grama	malina@yahoo.com	0752181344
4	Andrei	tokyonoize@gmail.com	0770121344
8	Alex	alex@yahoo.com	0748521684
9	Pop Bianca	bianca.pop@gmail.com	0784521256

If the user wants to Add, Edit or Delete from the table, he simply has to either input the text in the TextFields for a client to be added, or click an existing user and modify its data in the TextFields to Edit or Delete the client.

The user can also search a client by its ID, by typing the ID in the "Search by ID" TextField and pressing the Search button. A new window will pop up, containing all of the information for the respective client.



idClient	name	email	phoneNumber
1	Malina Grama	malina@yahoo.com	0752181344
4	Andrei	tokyonoize@gmail.com	0770121344
8	Alex	alex@yahoo.com	0748521684
9	Pop Bianca	bianca.pop@gmail.com	0784521256

Message

Searched client ID: 4

Name: Andrei

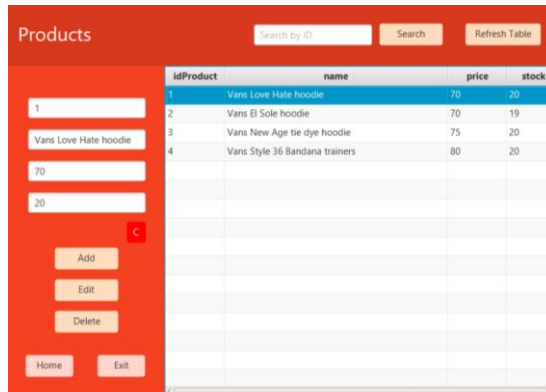
E-Mail: tokyonoize@gmail.com

Phone Number: 0770121344

OK

The Home button will take the user to the homepage window, where he can choose to see another window to perform more operations. This is implemented in all of the windows of the application except, of course, the homepage.

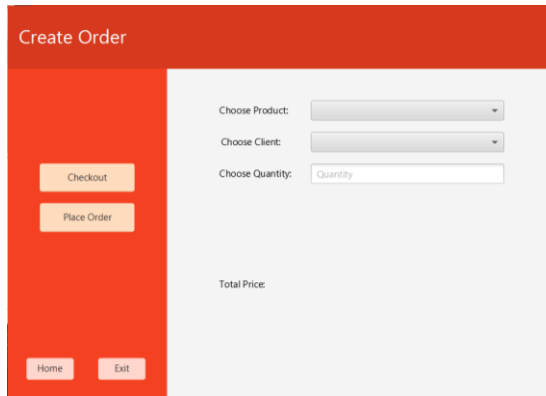
On the homepage, if the user clicks the Products button, a window similar to the Clients window will appear, containing the data from the database product table and the options of adding, editing or deleting a product from the database.



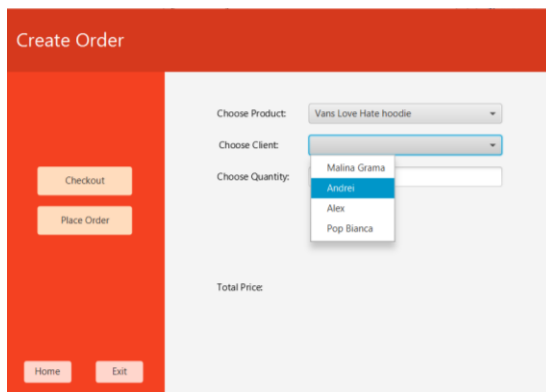
IdProduct	name	price	stock
1	Vans Love Hate hoodie	70	20
2	Vans B Sole hoodie	70	19
3	Vans New Age tie dye hoodie	75	20
4	Vans Style 36 Bandana trainers	80	20

The functionality is exactly the same for the Products page as it is for the Clients page.

The third and final window represents the window for creating an order, called Create Order. It looks like this:



The user can choose a product he wishes to create an order for by clicking on the DropDown field next to the Choose Product text. The names of all the products present in the database will then appear, and the user can choose one. The same can be said for choosing the client from the list of clients' names.



Then the quantity can be inputted from the keyboard in the Quantity TextField.

If the user presses the Checkout button, the total price of the order will be displayed, and the user can then press the Place Order button to add the order in the database. After this action is complete, a window showing a confirmation message will appear, and a button with the text "Open Bill" will appear. If the user presses it, a pdf file with information about the purchase will be opened.

### Create Order

Checkout

Place Order

Open Bill

Home

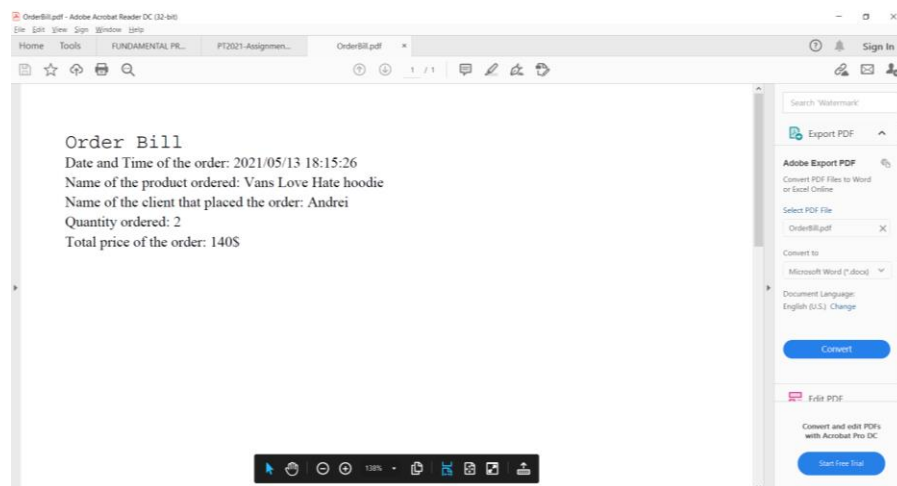
Exit

Choose Product: Vans Love Hate hoodie

Choose Client: Andrei

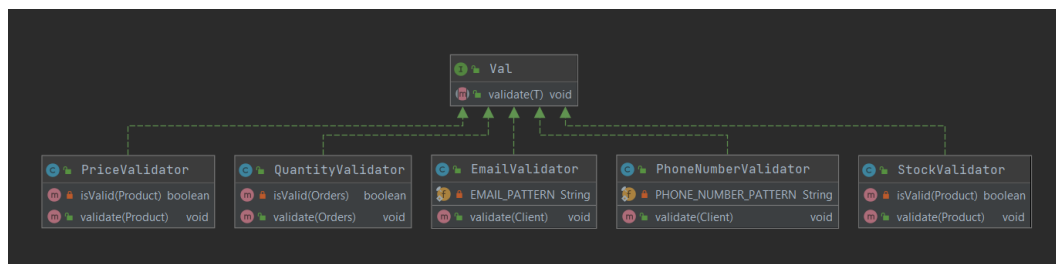
Choose Quantity: 2

Total Price: 140

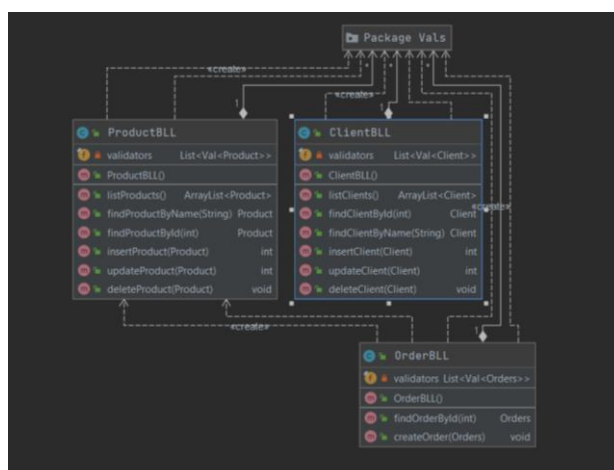


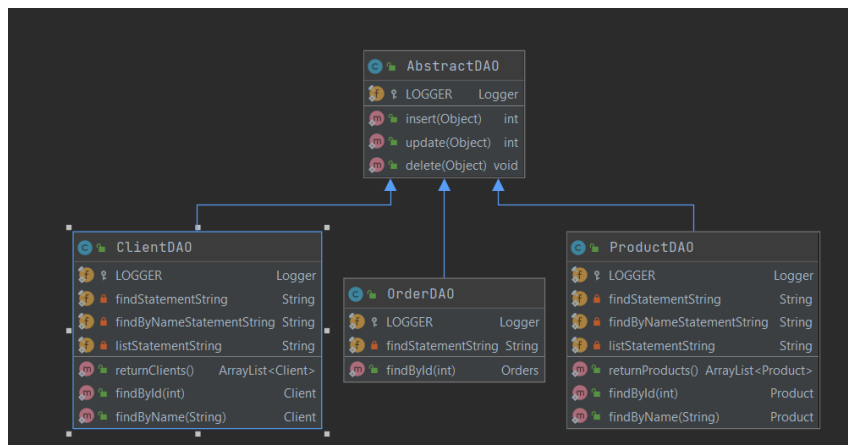
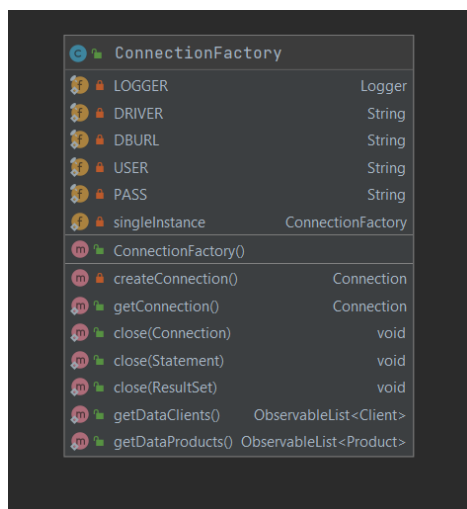
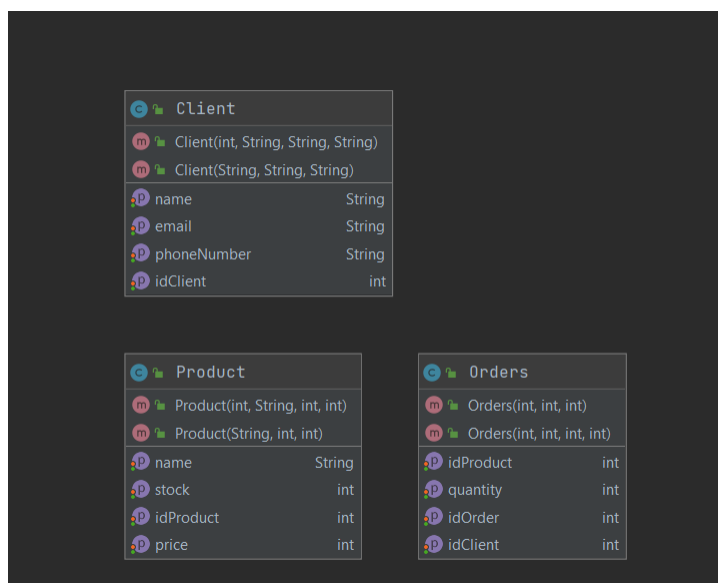
Next, I will include the UML Diagrams for each package in my project

### Package Vals:

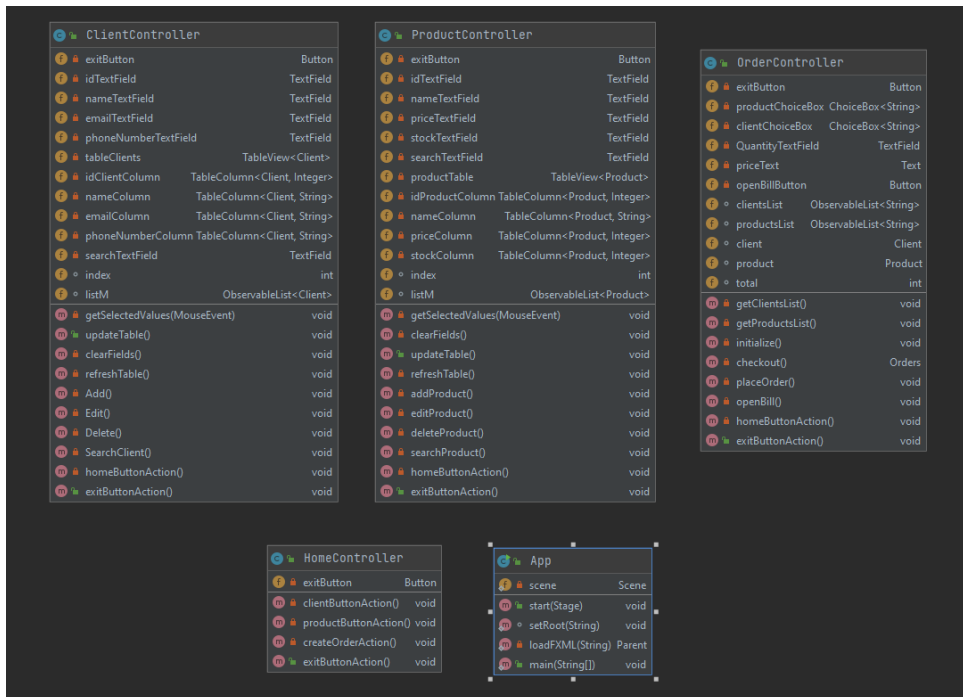


### Package BLL:

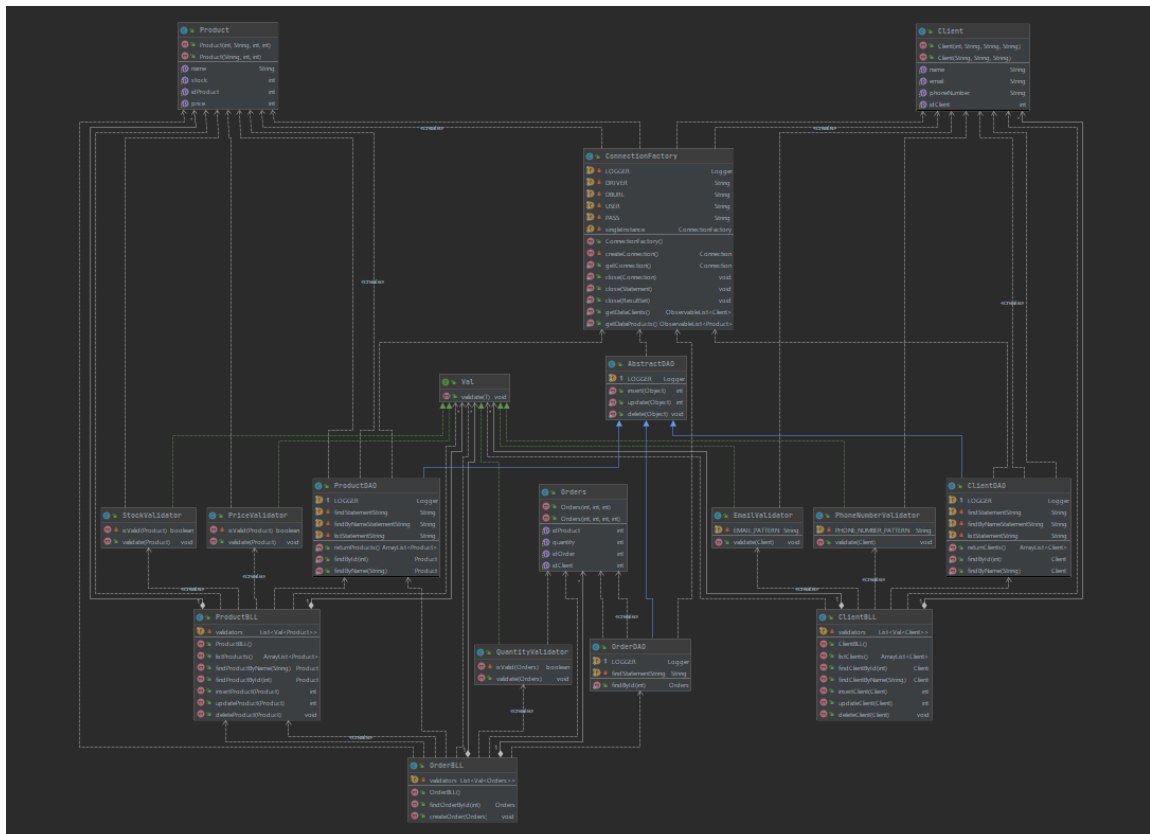


**Package DAO:****Package DataAccess:****Package Model:**

## Package PresentationLayer:



The generated UML Diagram for all of the classes that implement the backend:



## 5. Conclusions and Further Development

The task of working with databases in Java was interesting, and although it seemed challenging at first, after I understood the classes of connection, DAO and BLL it all went quite smoothly. I had to do some research regarding the opening of a database in IntelliJ IDEA, connecting to the database, generating a PDF file, generating JavaDoc files etc. Although it was a challenging task, I enjoyed it because of its applicability to the real world.

A thing I have learned from this first task is the importance of dividing the objectives at the very beginning of the problem. A lack of rigor in the planning phase can set back the implementation, so the best approach is to “divide and conquer” the big problem into smaller tasks and implementing, testing and debugging them before going forward.

As for further developments, the application could be extended in a lot of ways. A login page could be implemented, having two types of users: client and admin. The client could then browse through the products without being capable of modifying them, and place an order by viewing the products and inputting its credentials. The admin user would have more control over the application, being able to modify the database accordingly.

## 6. Bibliography

- Class Materials provided by the professor and the teaching assistant.
- <http://tynerblain.com/blog/2007/04/09/sample-use-case-example/>
- [https://bitbucket.org/utcn\\_dsrl/pt-layered-architecture/src/master/](https://bitbucket.org/utcn_dsrl/pt-layered-architecture/src/master/)
- <https://www.baeldung.com/java-pdf-creation>
- <https://www.baeldung.com/javadoc>