***Programming Techniques***

***Assignment 3***

***Order Management***

Student : Scutea Emilia

Group: 30423

Laboratory Assistant: Antal Marcel

Teacher: Ioan Salomie

Technical University of Cluj Napoca

2021

Contents

[**1.** **Objectives** 3](#_Toc69986934)

[**2.** **Problem Analysis, Modelling, Scenarios, Use Cases** 4](#_Toc69986935)

[**I.** **Problem Analysis** 4](#_Toc69986936)

[**II.** **Modelling** 4](#_Toc69986937)

[**III.** **Scenarios and Use Cases** 5](#_Toc69986938)

[**3.** **Design** 8](#_Toc69986939)

[**I.** **Class Design and UML Diagram** 8](#_Toc69986940)

[**II.** **Packages and Relationships** 10](#_Toc69986941)

[**III.** **Algorithms and Data Structures** 11](#_Toc69986942)

[**IV.** **Graphical User Interface** 12](#_Toc69986943)

[**4.** **Implementation** 14](#_Toc69986944)

[**I.** **Class Client** 14](#_Toc69986945)

[**II.** **Class Product** 14](#_Toc69986946)

[**III.** **Class OrderTable** 15](#_Toc69986947)

[**IV.** **Class AbstractDAO** 15](#_Toc69986948)

[**V.** **Class ClientDAO** 16](#_Toc69986949)

[**VI.** **Class ProductDAO** 16](#_Toc69986950)

[**VII.** **Class OrderDAO** 16](#_Toc69986951)

[**VIII.** **Class ConnectionFactory** 16](#_Toc69986952)

[**IX.** **Class ClientBLL** 17](#_Toc69986953)

[**X.** **Class ProductBLL** 17](#_Toc69986954)

[**XI.** **Class OrderBLL** 18](#_Toc69986955)

[**XII.** **Class EmailValidator** 18](#_Toc69986956)

[**XIII.** **Class OrderValidator** 18](#_Toc69986957)

[**XIIII. Class PriceValidator** 18](#_Toc69986958)

[**XV. Class Alert** 19](#_Toc69986959)

[**XVI. Class FirstPageDesign** 19](#_Toc69986960)

[**XVII. Class ClientDesign** 20](#_Toc69986961)

[**XVIII. Class ClientDesign** 20](#_Toc69986962)

[**XIX. Class OrderDesign** 21](#_Toc69986963)

[**XX.** **Main** 22](#_Toc69986964)

[**5.** **Results** 22](#_Toc69986965)

[**6.** **Conclusions** 22](#_Toc69986966)

[**7.** **Bibliography** 22](#_Toc69986967)

# **Objectives**

The main purpose of this project is to design an order management application which performs operations on database tables: client, ordertable and product.

In order to implement a functional application, we make use of the following operations:

1. Adding a client in the database
2. Updating a client in the database
3. Deleting a client from the database
4. Viewing all clients from the database
5. Adding a product in the database
6. Updating a product in the database
7. Deleting a product from the database
8. Viewing all products from the database
9. Adding an order in the database
10. Viewing all orders from the database

For a user-friendly approach we implement a graphical user interface in order to perform the above mentioned operations.

# **Problem Analysis, Modelling, Scenarios, Use Cases**

## **Problem Analysis**

A system which performs operations on database tables can be used in any domain mainly because of the great importance of the database management in any specific field of work. This management operations are commonly used in any company which works with several products and has to keep track of clients, product and the orders.

The application provides a user-friendly interface where the user can easily introduce the input and select the desired operation because of the buttons and textfields displayed on the application. There are also given some instructions on how to use the application, such that the usage of the calculator gets easy with no errors encountered.

## **Modelling**

A client consists of an ID, a name, an email and an address of shipping.

A product consists of an ID, a name, a price and the available quantity in the stock.

An order consists of an ID, an ID of the client who made the order, an ID of the ordered product, the quantity of the product and the total price of the order.

## **Scenarios and Use Cases**

In order to perform operations, the user must select the desired operation and then introduce the requested input.

Diagram, schematic

Description automatically generated

*Use case: Add a client*

Performed by: user

Best case scenario:

1. User presses the button „Perform clients operations”
2. User presses the button ”Add new client”
3. The program reads and validates the introduced name, address and email
4. The program performs the insert in the database
5. A confirmation alert is displayed on the screen

*Use case: Edit client*

Performed by: user

Best case scenario:

1. User presses the button „Perform clients operations”
2. User presses the button ”Edit existing client”
3. The program reads and validates the introduced ID, name, address and email
4. The program performs the update in the database
5. A confirmation alert is displayed on the screen

*Use case: Delete client*

Performed by: user

Best case scenario:

1. User presses the button „Perform clients operations”
2. User presses the button ”Delete existing client”
3. The program reads and validates the introduced ID
4. The program performs the delete in the database
5. A confirmation alert is displayed on the screen

*Use case: View all clients*

Performed by: user

Best case scenario:

1. User presses the button „Perform clients operations”
2. User presses the button ”View all clients” and the button „Show the clients in the database”
3. A table with the clients in the database is displayed on the screen

*Use case: Add a product*

Performed by: user

Best case scenario:

1. User presses the button „Perform product operations”
2. User presses the button ”Add new product”
3. The program reads and validates the introduced name, price and quantity
4. The program performs the insert in the database
5. A confirmation alert is displayed on the screen

*Use case: Edit product*

Performed by: user

Best case scenario:

1. User presses the button „Perform product operations”
2. User presses the button ”Edit existing product”
3. The program reads and validates the introduced ID, name, price and quantity
4. The program performs the update in the database
5. A confirmation alert is displayed on the screen

*Use case: Delete product*

Performed by: user

Best case scenario:

1. User presses the button „Perform product operations”
2. User presses the button ”Delete existing product”
3. The program reads and validates the introduced ID
4. The program performs the delete in the database
5. A confirmation alert is displayed on the screen

*Use case: View all products*

Performed by: user

Best case scenario:

1. User presses the button „Perform product operations”
2. User presses the button ”View all products” and the button „Show the products in the database”
3. A table with the products in the database is displayed on the screen

*Use case: Add an order*

Performed by: user

Best case scenario:

1. User presses the button „Perform order operations”
2. User presses the button ”Create a new order” and the button „Create order”
3. The program reads and validates the selected product, the price, quantity, the total price and the ID of the client
4. The program performs the insert in the database
5. A confirmation alert is displayed on the screen

*Use case: View all clients*

Performed by: user

Best case scenario:

1. User presses the button „Perform order operations”
2. User presses the button ”View all orders” and the button „Show the orders in the database”
3. A table with the orders in the database is displayed on the screen

# **Design**

## **Class Design and UML Diagram**

The design of this project is based in the Model-View-Controller design, thus there are 3 main packages in which we are interested in.

Diagram

Description automatically generated

The class *ConnectionFactory* creates and closes the connection with the database.

The class *AbstractDAO* implements the operations defined in 1. Objectives. It creates general queries: SELECT, INSERT, UPDATE, DELETE and it is a class having a generic type T.

The class *ClientDAO* inherits all methods from class AbstractDAO of generic type Client.

The class *ProductDAO* inherits all methods from class AbstractDAO of generic type Product.

The class *OrderDAO* inherits all methods from class AbstractDAO of generic type OrderTable.

The class *Client* represents a model of the table Client from the database and has the same attributes as in that table.

The class *Product* represents a model of the table Product from the database and has the same attributes as in that table.

The class *OrderTable* represents a model of the table OrderTable from the database and has the same attributes as in that table.

The class *ClientBLL* uses the methods from the ClientDAO class in order to check the results of the Client related queries.

The class *ProductBLL* uses the methods from the ProductDAO class in order to check the results of the Product related queries.

The class *OrderBLL* uses the methods from the OrderDAO class in order to check the results of the OrderTable related queries.

The class *EmailValidator* validates the email of a client.

The class *OrderValidator* validates the price of an order.

The class *PriceValidator* validates the price of a product.

The class *ClientDesign* is a controller class for the windows regarding operations on clients: add, edit, view, delete.

The class *ProductDesign* is a controller class for the windows regarding operations on products: add, edit, view, delete.

The class *OrderDesign* is a controller class for the windows regarding operations on orders: add, view.

The class *FirstPageDesign* is a controller class for the windows regarding operations on each item: client, product and order.

The class *Alert* defines several methods used to generates alerts. Alerts are used to inform the user on input mistakes (invalid format of the input) or inform the user about successfully or unsuccessfully operations.

The class *Main* launches a new window and starts the application.

## **Packages and Relationships**

The application contains 6 packages:

1. Bussines Layer Logic

This package contains the classes ClientBLL, ProductBLL, OrderBLL, which use methods from class AbstractDAO to check the results of the queries.

1. Connection

This package contains the class *ConnectionFactory*, which creates the connection with the database.

1. Data Access Object

As the name suggests, this package is used for classes defining the queries performed to make operations on the database: SELECT, INSERT, UPDATE, DELETE.

1. Model

This package contains classes representing models of the tables in the database.

1. Presentation

This package containes the UI related classes which are controllers used to manage the operations performed when pressing buttons/ inserting data in the provided text fields.

1. Start

This package contains the class Main, which starts the application.

## **Algorithms and Data Structures**

In this implementation all algorithms used to implement the operations are using queries which are executed in the database.

Firstly, a specific query is created: a SELECT/INSERT/UPDATE/DELETE query, with the requested fields i.e.: „SELECT \* FROM client WHERE ID=?”. The „?” will be replaced by the value of the given ID.

The second step is to create a connection with the database, fill in all the ? in the statement with the specific values (the number of fields differs from a query to another) and then execute the statement. The methods either return a list/ an object representing the returned result set or true/false if the statement was successfully executed in the database.

The last step is to close the connection with the database.

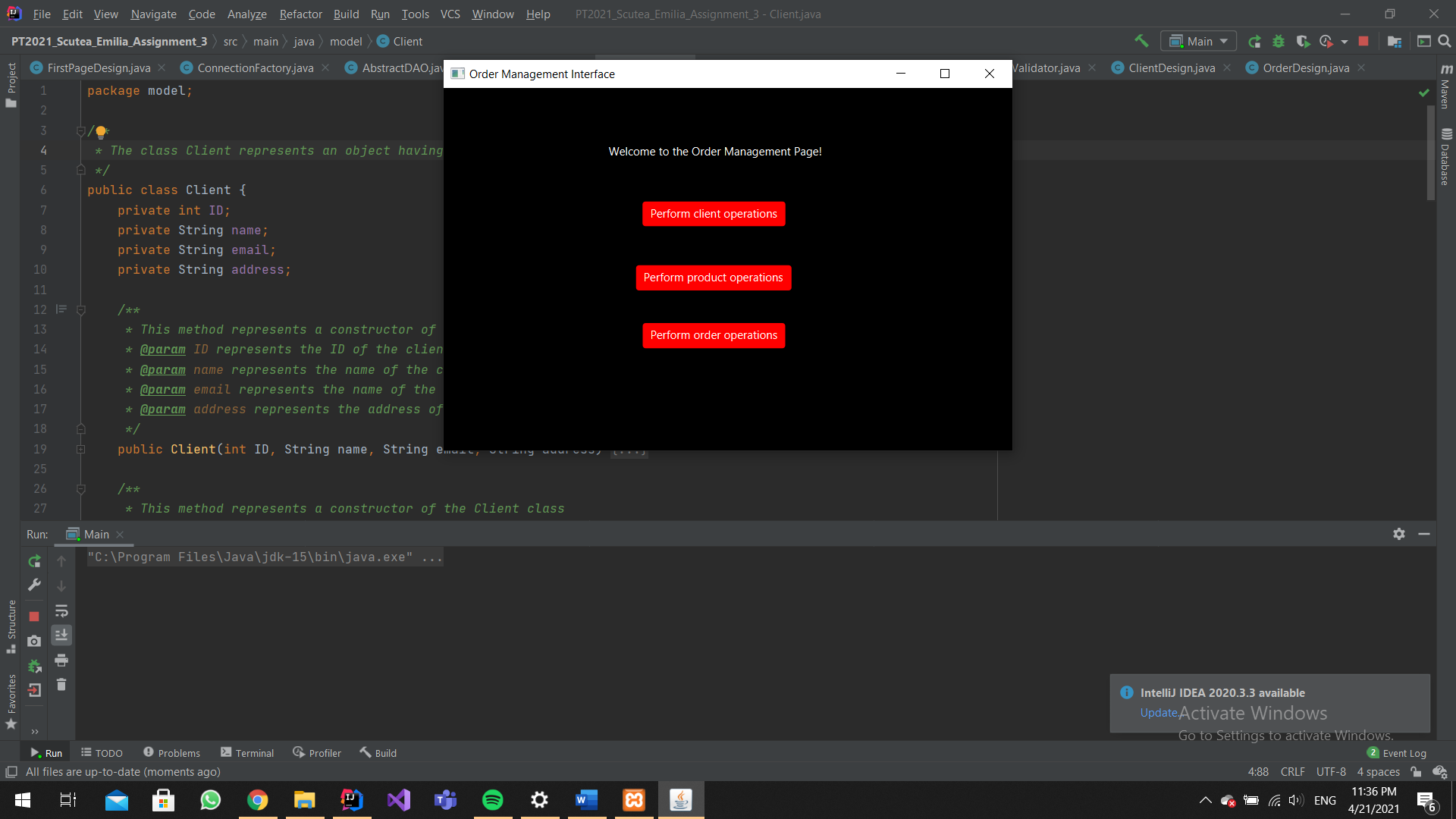
## **Graphical User Interface**

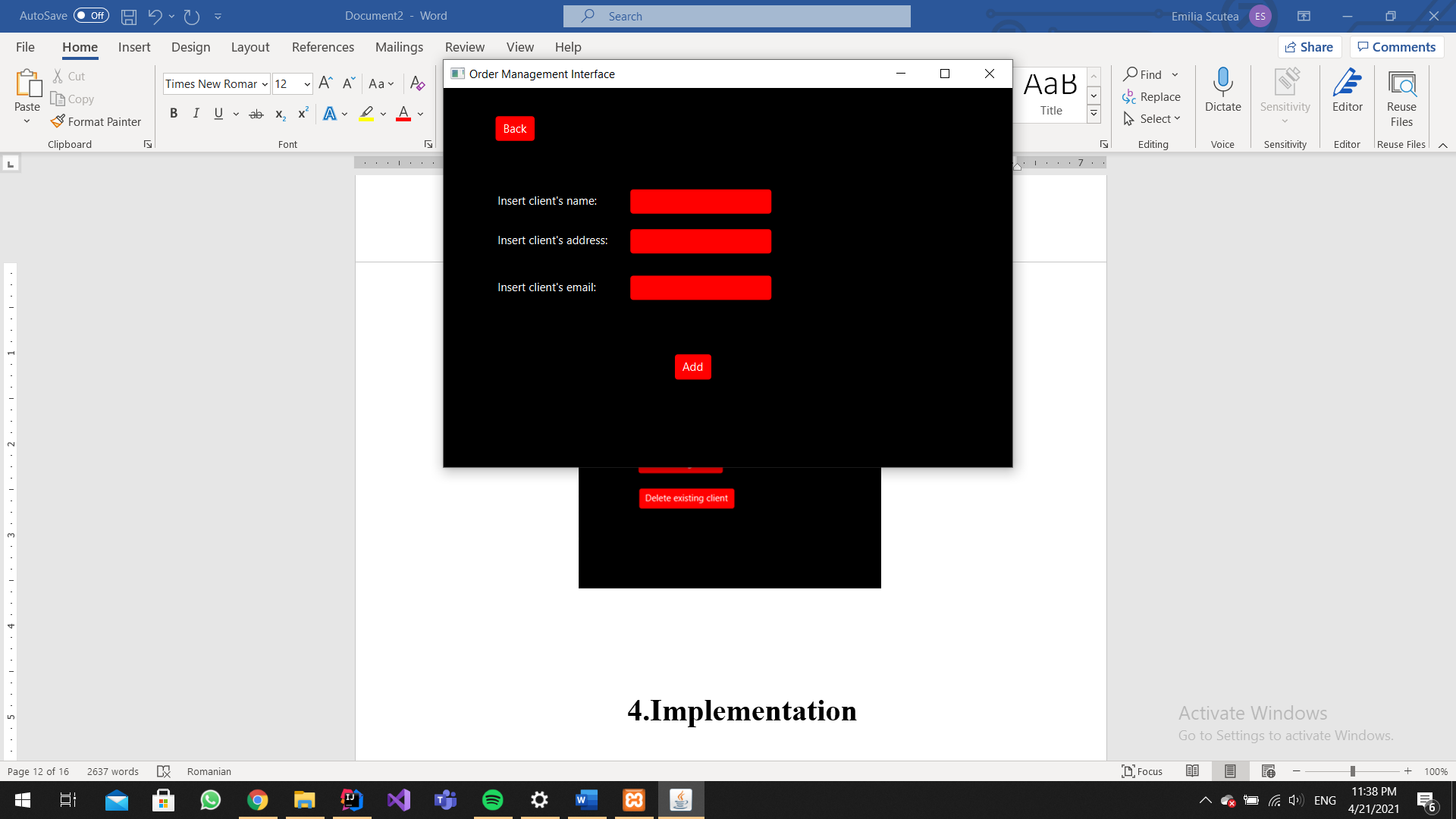
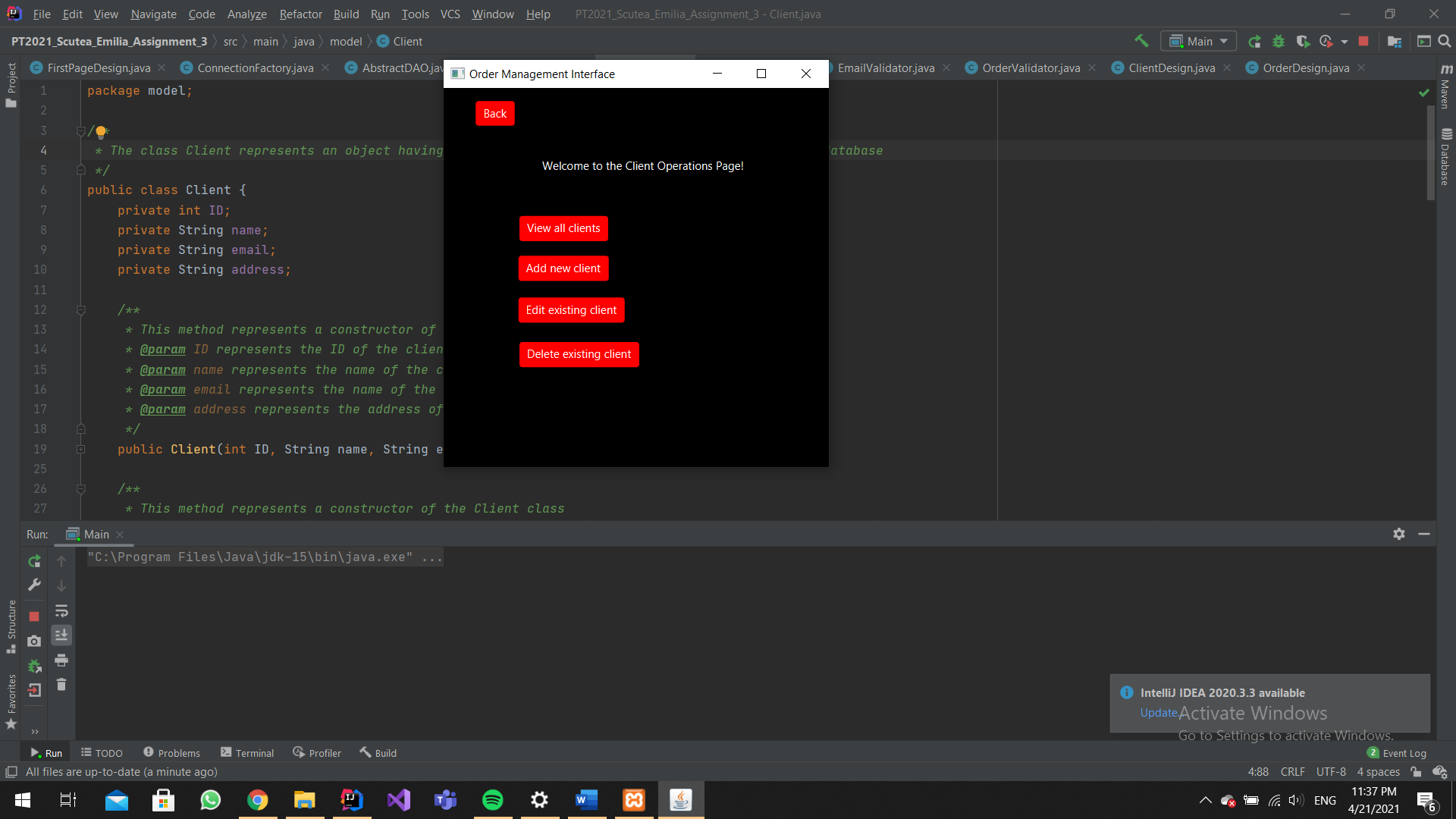
The graphical user interface is a layered interface with several windows. The buttons are used to select with desired operation to perform and they have labels which describe the operation. There also exist a button used to go back to the previus page.

The first window is used to select which operations to perform: client, product or order operations. After the selection, another window pops up, where the user must select which type of operation wants to perform: add, edit, delete or view. After selecting the desired operation, a new window appears, where the user must introduce the required data.

The text fields are used to introduce the input data.

If the introduced data is invalid, an alert will appear on the screen, specifying the mistake in the input.





# **Implementation**

## **Class Client**

The class *Client* represents a model of the table Client from the database and has the same attributes as in that table.

The client contains an ID, a name, an email and an address.

The defined methods in this class are:

* setID (int ID): sets the ID
* getID (): returns the ID
* setName (String name): sets the name
* getName (): returns the name
* setEmail (String email): sets the email
* getEmail (): returns the email
* setAddress (String address): sets the address
* getAddress (): returns the address

## **Class Product**

The class *Client* represents a model of the table Product from the database and has the same attributes as in that table.

The product contains an ID, a name, a price and a quantity.

The defined methods in this class are:

* setID (int ID): sets the ID
* getID (): returns the ID
* setName (String name): sets the name
* getName (): returns the name
* setPrice (float price): sets the price
* getPrice (): returns the price
* setQuantity (int quantity): sets the quantity
* getQuantity (): returns the quantity

## **Class OrderTable**

The class *OrderTable* represents a model of the table OrderTable from the database and has the same attributes as in that table.

The product contains an ID, a clientID, a product, a price and a quantity.

The defined methods in this class are:

* setID (int ID): sets the ID
* getID (): returns the ID
* setClientID (int ClientID): sets the ClientID
* getClientID (): returns the ClientID
* setProduct (String product): sets the product
* getProduct (): returns the product
* setPrice (float price): sets the price
* getPrice (): returns the price
* setQuantity (int quantity): sets the quantity
* getQuantity (): returns the quantity

## **Class AbstractDAO**

The class *AbstractDAO<T>* implements the operations defined in 1. Objectives. It creates general queries: SELECT, INSERT, UPDATE, DELETE and it is a class having a generic type T.

The defined methods in this class are:

* createObjects (ResultSet resultSet): creates the objects returned by the result set obtained fromexecuting a query
* createFindByIdQuery (String field): returns a string representing the SELECT query
* findById (int id): returns the element returned by the SELECT query
* createInsertQuery (T t): returns a string representing the INSERT query
* insert (T t): returns true if the INSERT was executed, otherwise false
* createUpdateQuery (): returns a string representing the UPDATE query
* update (T t): returns true if the UPDATE was executed, otherwise false
* createDeleteQuery (): returns a string representing the DELETE query
* delete (T t): returns true if the DELETE was executed, otherwise false
* createFindAllQuery (): returns a string representing the SELECT \* query
* findById (int id): returns the a list of elements returned by the SELECT \* query

## **Class ClientDAO**

The class *ClientDAO* inherits all methods from class AbstractDAO of generic type Client.

## **Class ProductDAO**

The class *ProductDAO* inherits all methods from class AbstractDAO of generic type Product.

## **Class OrderDAO**

The class *OrderDAO* inherits all methods from class AbstractDAO of generic type Order.

## **Class ConnectionFactory**

The class *ConnectionFactory* creates and closes the connection with the database.

The defined methods in this class are:

* createConnection (): creates the connection with the database and returns it
* getConnection (): returns the connection with the database
* close(Connection connection): closes the connection with the database
* close(Statement statement): closes a statement
* close(ResultSet resultSet): closes a result set

## **Class ClientBLL**

The class *ClientBLL* uses the methods from the ClientDAO class in order to check the results of the Client related queries.

The defined methods in this class are:

* findById (int id): returns the client found by the findById method in ClientDAO
* insert (Client client): checks if client was inserted by the insert method in ClientDAO, returns true if yes, false otherwise
* update (Client client): checks if client was updated by the update method in ClientDAO, returns true if yes, false otherwise
* findAll(): returns a list of clients returned by the findAll method in ClientDAO
* delete (Client client): checks if client was deleted by the delete method in ClientDAO, returns true if yes, false otherwise

## **Class ProductBLL**

The class *ProductBLL* uses the methods from the ProductDAO class in order to check the results of the Product related queries.

The defined methods in this class are:

* findById (int id): returns the product found by the findById method in ProductDAO
* findByName (String name): returns the product found by the findByName method in ProductDAO
* insert (Product product): checks if product was inserted by the insert method in ProductDAO, returns true if yes, false otherwise
* update (Product product): checks if product was updated by the update method in ProductDAO, returns true if yes, false otherwise
* findAll(): returns a list of products returned by the findAll method in ProductDAO
* delete (Product product): checks if product was deleted by the delete method in ProductDAO, returns true if yes, false otherwise

## **Class OrderBLL**

The class *OrderBLL* uses the methods from the OrderDAO class in order to check the results of the OrderTable related queries.

The defined methods in this class are:

* findById (int id): returns the order found by the findById method in OrderDAO
* insert (OrderTable order): checks if order was inserted by the insert method in OrderDAO, returns true if yes, false otherwise
* findAll(): returns a list of orders returned by the findAll method in OrderDAO

## **Class EmailValidator**

The class *EmailValidator* validates the email of a client.

The defined method in this class is:

* validate(Client t): returns an exception if the email of the client can not be validated by a specific email pattern

## **Class OrderValidator**

The class *OrderValidator* validates the price of an order.

The defined method in this class is:

* validate(OrderTable order): returns an exception if the price of the order is less or equal to 0

## **XIIII. Class PriceValidator**

The class *PriceValidator* validates the price of a product.

The defined method in this class is:

* validate(Product product): returns an exception if the price of the product can not be validated by a specific price pattern

## **XV. Class Alert**

The class *Alert* is used to generate different alerts for errors occurring in introducing the input in the UI or to confirm the successfull execution of operations on database tables.

The defined methods in this class are:

* alertInvalidID(): creates an alert which informs the user about an invalid ID introduced
* alertInvalidInput(): creates an alert which informs the user about an invalid input
* alertInvalidEmail(): creates an alert which informs the user about an invalid email introduced
* alertInvalidQuantity(String quantity): creates an alert which informs the user about a invalid quanitty introduced
* alertClientNotFound(): creates an alert which informs the user that a client was not found in the database
* alertErrorInsert(): creates an alert which informs the user about a failed insert query
* alertErrorEdit(): creates an alert which informs the user about a failed update query
* alertErrorDelete(): creates an alert which informs the user about a failed delete query
* alertInsertSuccessful(): creates an alert which informs the user about a successful insert query
* alertEditSuccessful(): creates an alert which informs the user about a successful update query
* alertDeleteSuccessful(): creates an alert which informs the user about a successful delete query
* alertBillSuccessful(): creates an alert which informs the user that the bill of the order was successfuly created
* alertBillUnsuccessful(): creates an alert which informs the user that the generation of the bill failed

## **XVI. Class FirstPageDesign**

The class *FirstPageDesign* is a controller class for the window/scene regarding operations on each item: client, product and order

The defined methods in this class are:

* setScene(): sets the window of the first page in the user interface
* performClientOperation(): creates a new instance of the ClientDesign class and sets the window of the client design in the user interface
* performProductOperation(): creates a new instance of the ProductDesign class and sets the window of the product design in the user interface
* performOrderOperation(): creates a new instance of the OrderDesign class and sets the window of the order design in the user interface

## **XVII. Class ClientDesign**

The class *ClientDesign* is a controller class for the windows/scenes regarding operations on clients: add, edit, view and delete.

The defined methods in this class are:

* setScene(String FXMLFile): sets the window given by an FXML file in the user interface
* closeButtonClient(): sets the client design window when the close button is pressed in the user interface
* closeButtonMain(): sets the first page design window when the close button is pressed in the user interface
* addNewClient(): sets the window of „Add client” in the user interface
* validateAddInput(): validates the input introduced by the user in the window „Add client”
* addClientButton(): adds the introduced client in the database
* editClient(): sets the window of „Edit client” in the user interface
* validateEditInput(): validates the input introduced by the user in the window „Edit client”
* editClientButton(): edits the introduced client in the database
* deleteClient(): sets the window of „Delete client” in the user interface
* validateDeleteInput(): validates the input introduced by the user in the window „Delete client”
* deleteClientButton(): deletes the introduced client in the database
* viewClients(): sets the windows of „View clients” in the user interface
* setTableView(): populates the table with all the clients from the database

## **XVIII. Class ClientDesign**

The class *ProductDesign* is a controller class for the windows/scenes regarding operations on products: add, edit, view and delete.

The defined methods in this class are:

* setScene(String FXMLFile): sets the window given by an FXML file in the user interface
* closeButtonProduct(): sets the product design window when the close button is pressed in the user interface
* closeButtonMain(): sets the first page design window when the close button is pressed in the user interface
* addProduct(): sets the window of „Add product” in the user interface
* validateAddInput(): validates the input introduced by the user in the window „Add product”
* addProductButton(): adds the introduced product in the database
* editProduct (): sets the window of „Edit product” in the user interface
* validateEditInput(): validates the input introduced by the user in the window „Edit product”
* editProducttButton(): edits the introduced product in the database
* deleteProduct (): sets the window of „Delete product” in the user interface
* validateDeleteInput(): validates the input introduced by the user in the window „Delete product”
* deleteProductButton(): deletes the introduced product in the database
* viewProducts(): sets the windows of „View products” in the user interface
* setTableView(): populates the table with all the products from the database

## **XIX. Class OrderDesign**

The class *OrderDesign* is a controller class for the windows/scenes regarding operations on orders: add and view.

The defined methods in this class are:

* setScene(String FXMLFile): sets the window given by an FXML file in the user interface
* closeButtonOrder(): sets the order design window when the close button is pressed in the user interface
* closeButtonMain(): sets the first page design window when the close button is pressed in the user interface
* addNewOrder(): sets the window of „Add order” in the user interface
* getProductList(): gets all the products from the database and places them in an observable list
* showProducts(): sets the items of the products choice box and makes visible all labels, text fields and buttons in the „Add order” window in the user interface
* getChosenProduct(): gets the product selected by the user form the choicebox
* calculateOrderPrice(): calculates the total price of the order
* validateInput(): validates the input introduced by the user in the window „Add order”
* placeOrder(): adds the introduced order in the database
* createBill(): creates a bill of the order, with information about the product, the price and the client which made the order
* viewOrders(): sets the windows of „View orders” in the user interface
* setTableView(): populates the table with all the orders from the database

## **Main**

In the Main class, the method start creates a new stage which will be displayed on the screen, while in the main method the stage is launched and a new instance of the controller is created.

# **Results**

All database related operations which were performed by the user in the graphical user interfaced are validated by alerts which pop up on the screen to inform the user if the operation failed or if it was successfuly executed. Furthermore, the performed operations cand also be checked by inspecting the tables in the UI windows where all clients/product/orders are displayed or in the database tables of the application.

# **Conclusions**

I can affirm that during the development of this assignment I have got more familiar with reflection techniques, how this method works and how operations are implemented in this form. The assignment also requested a layered architecture of the project, which gave me a greater perspective of how an application should be organized.

Some future development ideas could be:

* A more interactive user interface
* More advanced operations: create order with several products

# **Bibliography**

<https://openjfx.io/>

<https://mvnrepository.com/>

<https://utcn_dsrl@bitbucket.org/utcn_dsrl/pt-layered-architecture/src/master/>

<https://utcn_dsrl@bitbucket.org/utcn_dsrl/pt-reflection-example/src/master/>