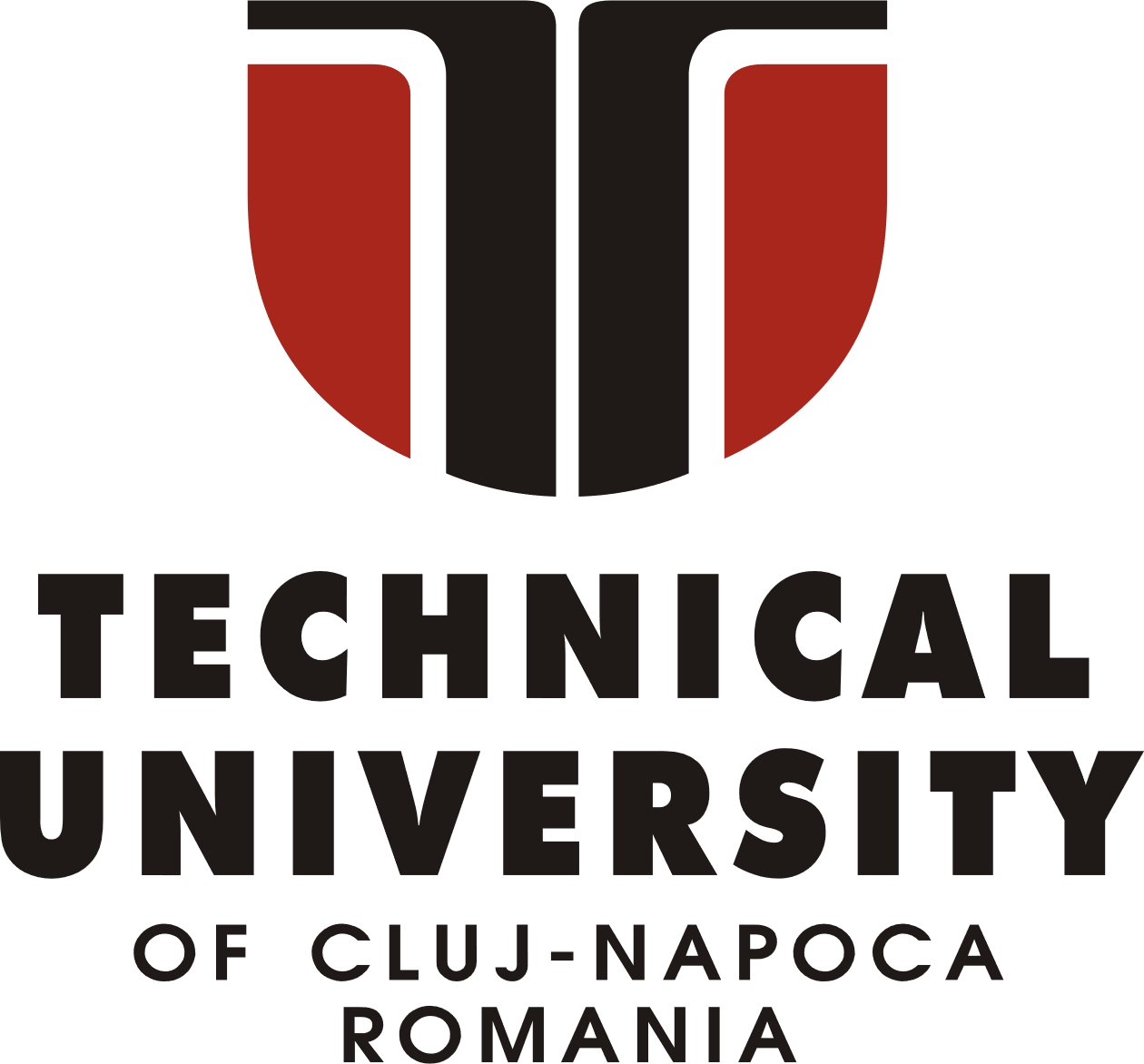
****

**\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_**

**DOCUMENTATION**

**Assignment 3. Orders Management**

**Application**

Fundamental Programming Techniques

Student: **Radu-Augustin Vele**

Group: **30422**

Table of Contents

1. Project Objectives3

2. Problem Analysis. Modelling. Scenarios. Use cases3

3. Design7

4. Implementation9

5. Results11

6. Conclusions11

7. Bibliography11

**1. Project objectives**

The **main goal** of the project is to provide the users (employees) with a management application destined for any kind of warehouse. The application deals with client, product, and orders management.

**Secondary objectives**:

* analyze the problem while identifying requirements (see Problem Analysis)
* design the project using object-oriented programming (see Design section)
* organize and implement the application using the Layered Architecture approach and store the data in relational databases (see Implementation section)
* use reflection and generics for the database access.
* test the application for various scenarios. (See Results section)

**2. Problem Analysis. Modelling. Scenarios. Use cases.**

The problem imposes both functional and non-functional requirements.

**Functional requirements:**

a. Client-related:

* the employees need to be able to add a new client to the database (uniquely identified with an id and with their email address).
* the employees need to be able to delete or edit any kind of information about an existing client.
* the employees need to be able to get information about all the clients in the database in a tabular organization.
* the employees must be able to duplicate the clients table for a specific purpose.

b. Product-related:

* the employees need to be able to add a new product to the database. (Uniquely identified with its id)
* the employees need to be able to delete or edit any kind of information about an existing product.
* the employees need to be able to get information about all the products in the database in a table.

c. Order-related:

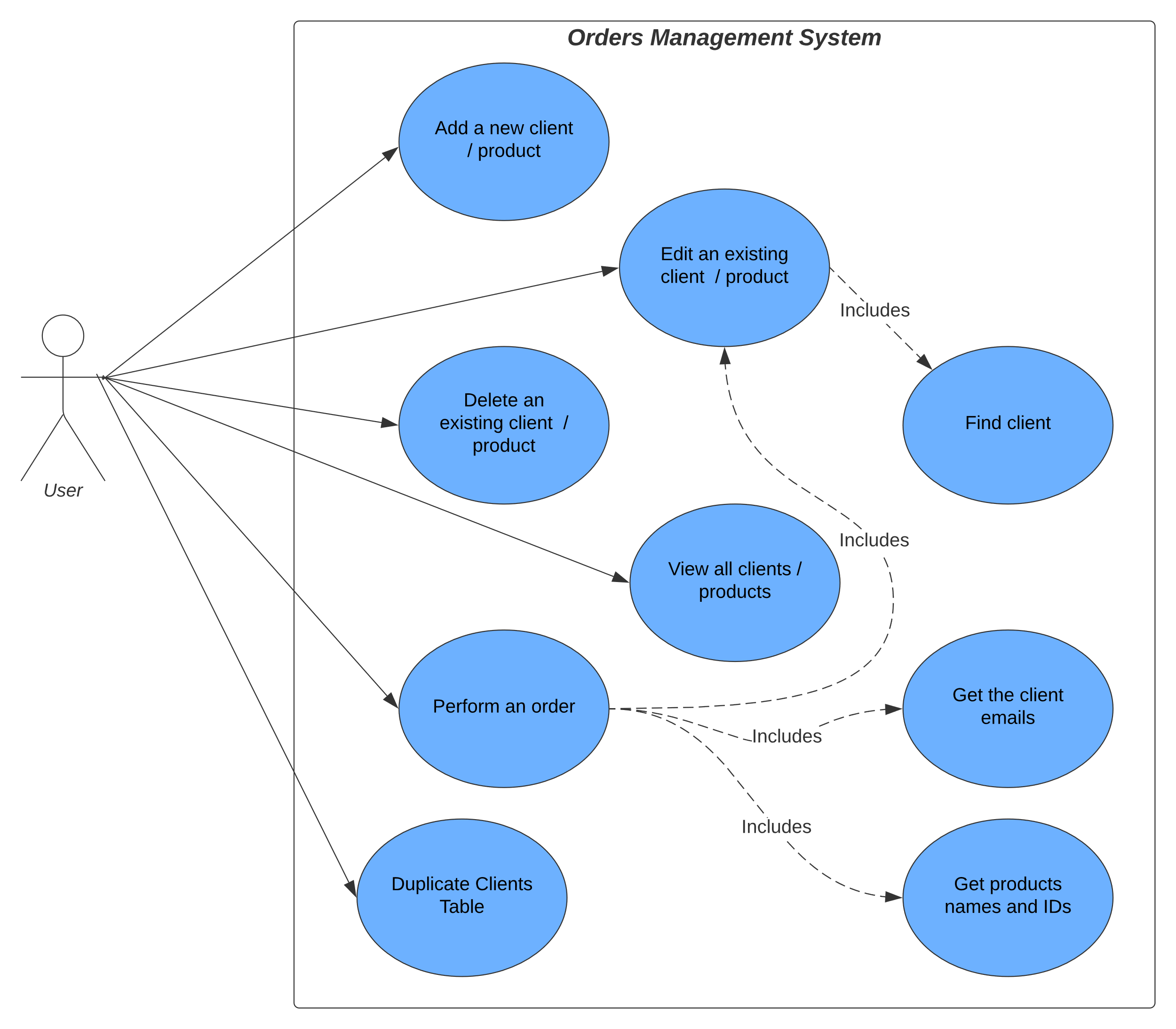
* the employees must be able to perform an order. That means selecting an existing client, an existing product, and the wanted number of items to be ordered.

**Non-functional requirements:**

* the application should be intuitive and easy to use.
* the interface should be organized in a familiar way.
* the switch between the functionalities of the application should be "smooth" and easy (for instance, the users should easily check the current content of the database tables while editing a client).

**Use cases:**

The use cases of the orders management application are depicted in the figure below.



The steps of execution in each use case are presented below:

**Use case: add a new client.** (Similar for products)

Actor: employee

Steps:

a. Success scenario steps:

1. the user presses on the client operations button on the home screen.
2. the user selects the "Add" tab in the client window.
3. the user inserts all the information for each field shown.
4. the user presses the "Insert Client" button.
5. several insert functions are called in the program (top-down from the presentation layer to the database)
6. the email and phone number fields are validated
7. the client is inserted in the database table and a success label is visible on the client window.

b. Alternative sequence steps:

1. the user presses on the client operations button on the home screen.
2. the user selects the "Add" tab in the client window.
3. the user inserts information in some of the shown fields.
4. the user presses the "Insert Client" button.
5. an error pop-up window appears on the screen communicating the user that they need to complete all fields.

c. Alternative sequence steps:

1. the user presses on the client operations button on the home screen.
2. the user selects the "Add" tab in the client window.
3. the user inserts information in all the fields but the email or phone number are not valid.
4. the user presses the "Insert Client" button.
5. an error pop-up window appears on the screen communicating the user that the phone number and/or the email address don't have the required format.

d. Alternative sequence steps:

1. the user presses on the client operations button on the home screen.
2. the user selects the "Add" tab in the client window.
3. the user inserts information in all the fields but use an email that has already been inserted.
4. the user presses the "Insert Client" button.
5. an error pop-up window appears on the screen communicating the user that the insertion failed because there is already a user with that email address (the email address must be unique).

**Use case: delete an existing client**. (Similar for products)

Actor: employee

a. Success scenario steps:

1. the user presses on the client operations button on the home screen.
2. the user selects the delete tab in the client window.
3. the user inputs the email of the client they want to delete from the database.
4. the user presses the delete button.
5. the application deletes the client having that email from the database and a success label is visible on the client window.

b. Alternative sequence steps:

1. the user presses on the client operations button on the home screen.
2. the user selects the delete tab in the client window.
3. the user inserts an email that doesn't exist in the database.
4. an error pop-up window appears on the screen communicating the user that there isn't a client with that email in the database.

Drawback: the success label is shown even if there is not client with the email in the database

**Use case: edit a client in a table**. (Similar for products)

Actor: employee

a. Success scenario steps:

1. the user presses on the client operations button on the home screen.
2. the user selects the edit tab in the client window.
3. the user checks the checkbox and inserts new data for the fields they want to change.
4. the user presses on the edit client button.
5. the application searches for that client in the database, the client is retrieved, and the needed fields are changed with the new input data.
6. additional validations are performed, and the client is edited in the database.
7. a success label is made visible in the client window.

b. Possible scenarios when the operation doesn't work:

1. the user doesn't check and input text for the fields they want to edit.
2. the user inserts a new email address or phone number that are not valid.
3. In these cases, an error pop-up window with the proper message appears on the screen.

**Use case: view all clients in a table**. (Similar for products)

Actor: employee

a. Success scenario steps:

1. the user presses on the client operations button on the home screen.
2. the user selects the view all tab.
3. the user presses on the "Show all clients in table" button.
4. the data is printed on the client window inside a table.

**Use case: perform an order.**

Actor: employee

a. Success scenario steps:

1. the user presses on the order operations button on the home screen.
2. the user selects the desired client email address.
3. the user selects the desired product.
4. the user inserts the number of items wanted in the order
5. the user checks if they want a bill or not.
6. the order is performed and information about it is inserted in the database.
7. the receipt is generated as a pdf file.

b. Possible scenarios when the operation doesn't work:

1. the user didn't insert the number of items correctly.

2. there aren't enough items in stock.

In these cases, an error pop-up window with the proper message appears on the screen.

**Use case: duplicate client table.**

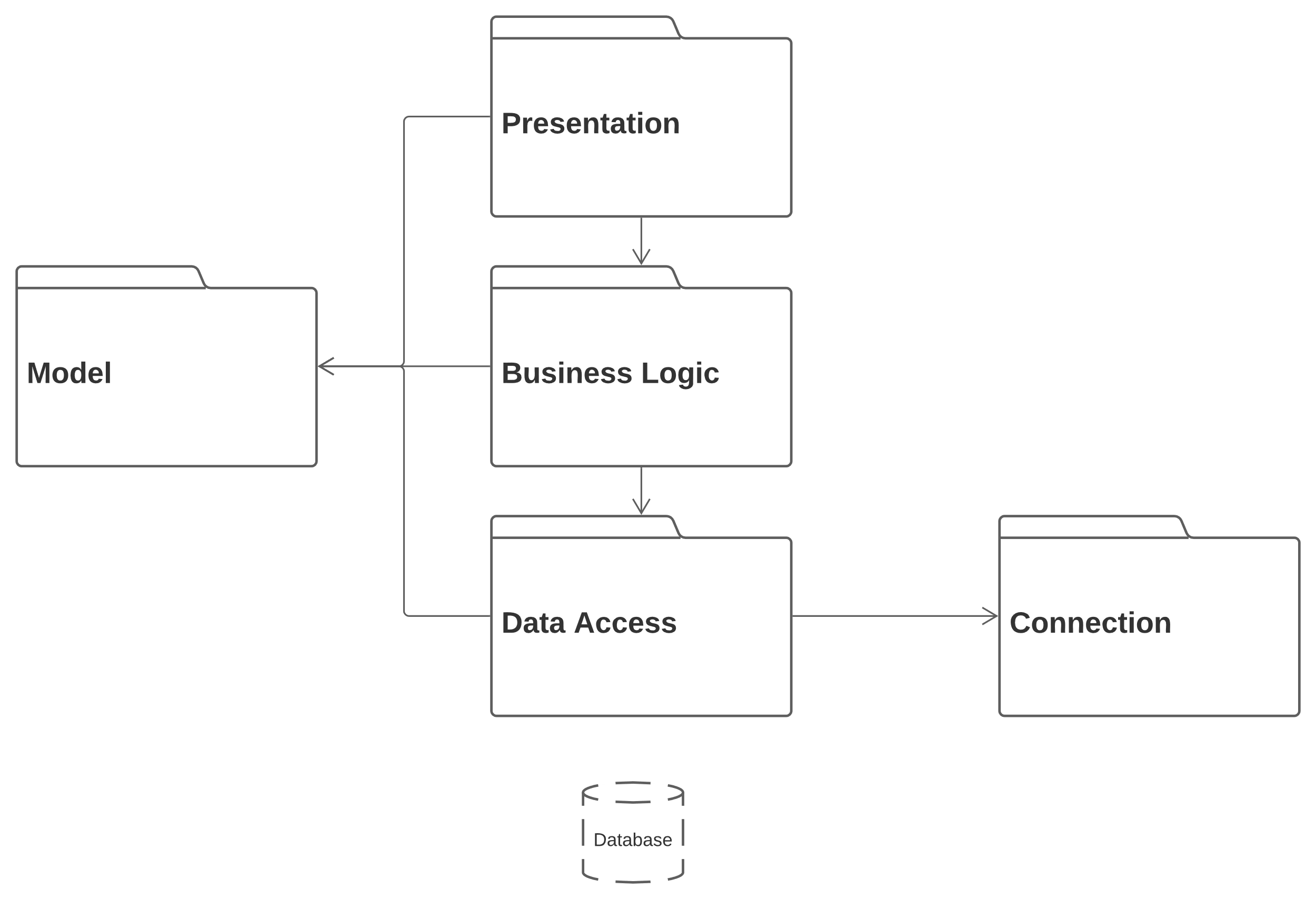
Actor: employee

a. Success scenario steps:

1. the user presses on the client operations button on the home screen.
2. the user selects the view all tab.
3. the user presses on the "Duplicate table button".
4. the table is duplicated in the database schema receiving the original name of the table concatenated with a number of identification.
5. a success label is visible on the screen.

**3. Design**

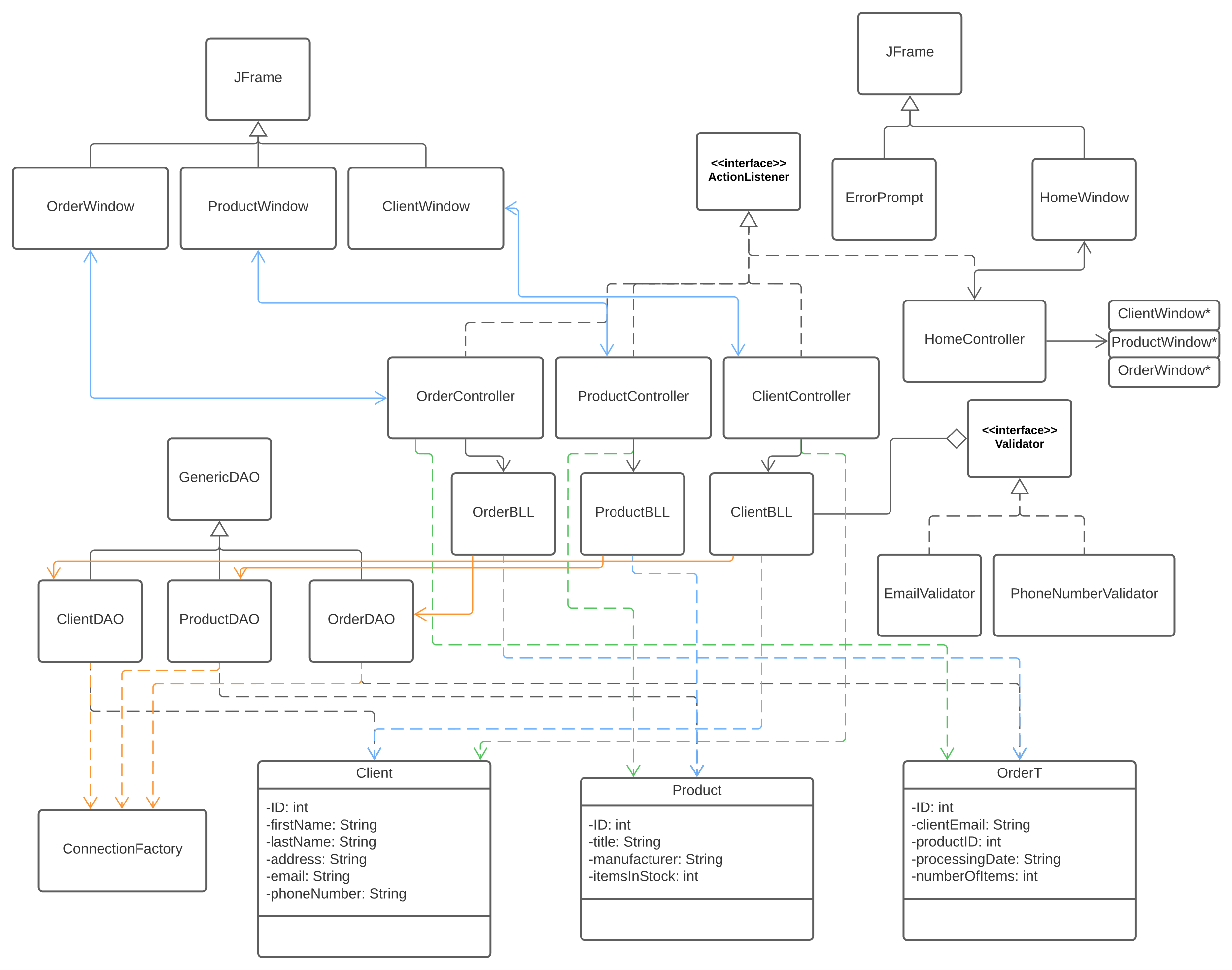
**Package Diagram**

The solution was designed using the OOP principles, following the layered architecture. The package diagram below brings a better understanding about the interaction between the classes that are grouped in layers.

This kind of architecture imposes a "vertical" communication based on requests and returned data. For instance, when a client is inserted, the ClientController whose action listener is called, calls a method in the ClientBLL. To follow, the ClientBLL validates the inserted field if needed and calls a method from the ClientDAO that accesses the database schema for updates. Subsequently a returned id is sent "upwards" to the business and presentation layers. As seen, the model classes: Client, Product, and OrderT are featured as local variables and parameters in classes belonging to all the layers. The connection package that only contains the Connection Factory class is linked to the data access layer. As its name states, it is responsible with the creation of the connection to the database.

The belonging of a class in a package is indicated by the class name. This way the classes ending in DAO belong to the Data Access level, the BLL classes belong to the Business Logic Layer, and the classes ending in Window and Controller belong to the Presentation Level.

**OOP Design**

The UML diagram below, shows the overall organization of the program classes and the relationships between them. There are multiple association and dependency relationships between the classes belonging to the different levels. An important aspect is that the data access classes inherit from a GenericDAO class that contains generalized CRUD methods for interacting with the database. Also, all the controllers implement ActionListener as they define behaviors that happen when buttons are pressed. Another idea is that for the validation mechanism, a validator interface (that can be implemented by validators for various fields - in our case email and phone number) has been created. This way, in the business logic classes there could be collections of different validators.

In the presentation layer, the classes Window and Controller are both having a field of each other, imposing double association.

**4. Implementation**

The model classes implementation consists of the fields visible in the UML class diagram while the methods are only getters and setters. Also, the BLL and DAO classes mostly call lower level and to the super class methods. The implementation of some of the more “interesting” classes is described as follows.

**GenericDAO**

This is a class that uses generics and reflection to perform CRUD (create, remove, update, delete) operations on the database tables.

public class GenericDAO<T> {...}

It is inherited by DAO classes that are destined to work with classes from the model. This way, the T parameter is destined to be replaced by either OrderT, Client, or Product. Each CRUD method first gets a connection to the database, through the ConnectionFactory instance. Then a query is prepared using one of the private methods that return the raw query string. Afterwards, through reflection (Class, Field, Method classes) the ‘?’s in the raw query are replaced by the proper values.

For example, the private function setFields is used for this purpose in the insert() method:

private int setFields(PreparedStatement queryStatement, Object object) throws Exception{  
 Field[] fields = type.getDeclaredFields();  
 int paramIndex = 1;  
 for(Field field : fields) {  
 if(!field.getName().equals("ID")) {  
 PropertyDescriptor propertyDescriptor = new PropertyDescriptor(field.getName(), type);  
 Method getter = propertyDescriptor.getReadMethod();  
 queryStatement.setObject(paramIndex, getter.invoke(object));  
 paramIndex++;  
 }  
 }  
  
 return paramIndex;  
}

**EmailValidator**

This class has a short implementation, but it is worth mentioning because it used regular expressions to validate whether an email is accepted in the database. The programmer could choose which kind of emails are available or not. In the initial implementation the regex is quite strict as it only accepts numbers and letters. The class implements the function validate() that uses the Pattern.matcher() function to check if a string is valid or not, returning the right information to the call in BLL.

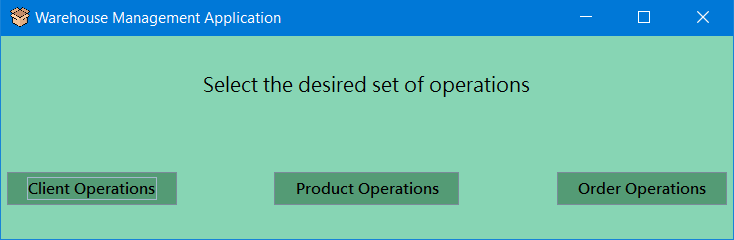
private static final String *email\_pattern* = "[\\w]+[@][\\w]+[.][\\w]{2,4}";

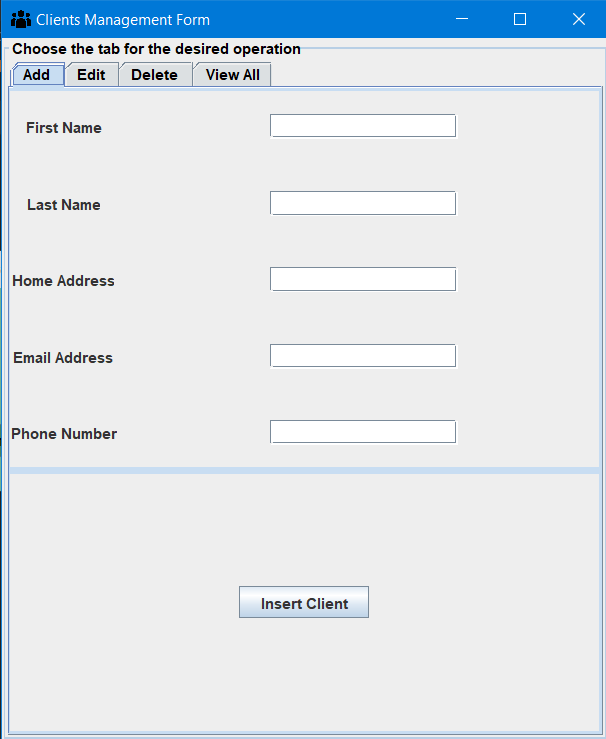
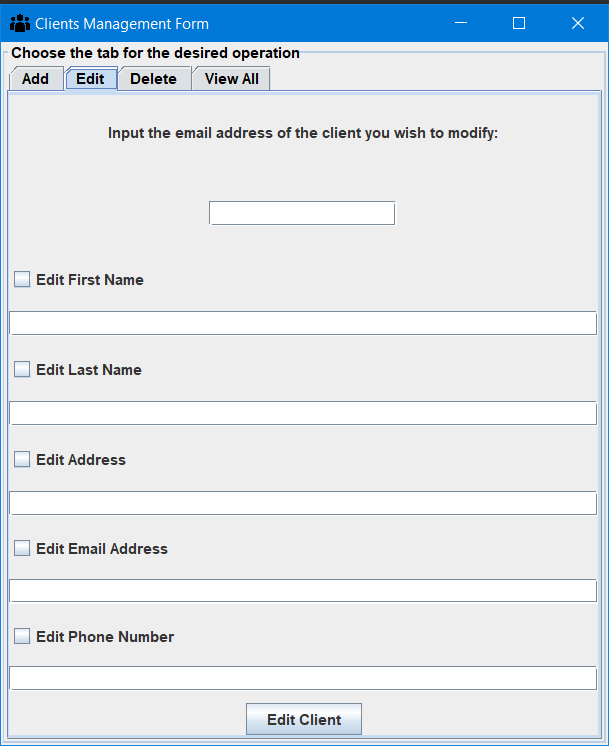
**ConnectionFactory**

This class in built using the **Singleton Design Pattern** that states that only an instance of the class could exist at a time. This is achieved through a private constructor and a method that returns the instance of the class if that class doesn’t yet exist. The main methods provided are

**Graphical User Interface**

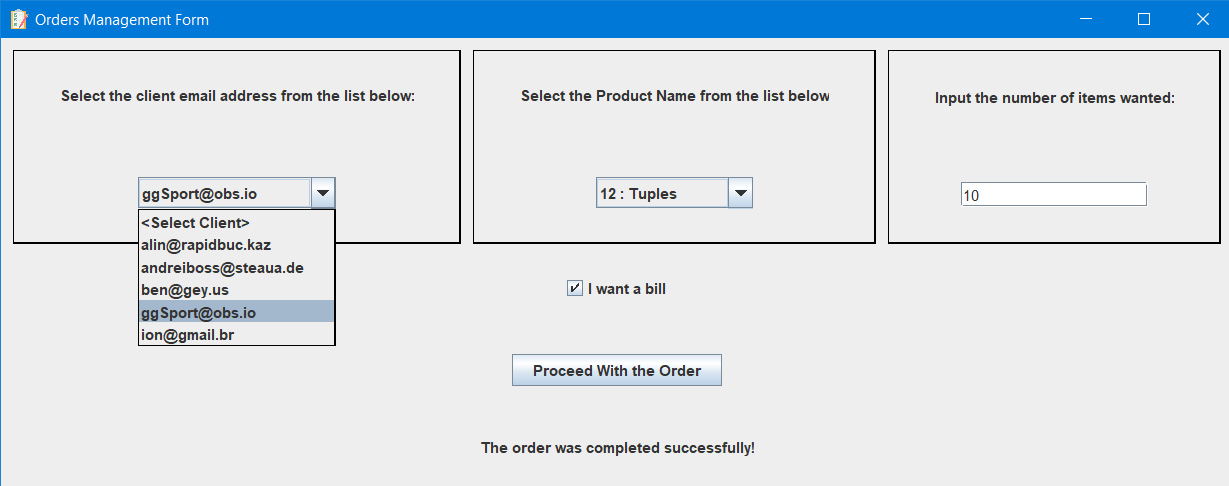
The Graphical User Interface was designed to allow the user to easily switch between the functionalities of the application. Initially a home window appears. After the user chooses the category of operations desired another window having multiple tabs appears. The design of the windows is presented below.





Graphical user interface, table

Description automatically generated



**5. Results.**

The application was tested with several inputs. Edge cases were taken into consideration as well. The obtained results proved to be as expected. To check the correspondence between the content of the database table and the information in the application, the MySQL Workbench application was used. The SQL Dump is also available in the project repository.

**6. Conclusions. Takeaways. Further Development**

The orders management application is a suitable choice for gaining a better understanding regarding the layered architecture and the communication with the database. As a complex project, it brought many challenges and a continuous learning journey. Moreover, it consists of a truly useful tool for managing any kind of warehouse. There are, of course many possible improvements such as more complex validators or a better and more industry-like GUI. Also, some methods could be further broken down into smaller one-purpose methods (to fully respect the single-responsibility principle).

**7. Bibliography**

1. Distributed systems research laboratories - TUCN, resources.

2. Stack overflow: Database errors,

3. W3Schools: SQL Course

4. Geeks for Geeks: Reflection in Java

5. Baeldung.com: Creating PDF files using pdfBox

6. Oracle Java Docs: JComboBox, Field Class