

# Warehouse Management System

Miu Daria

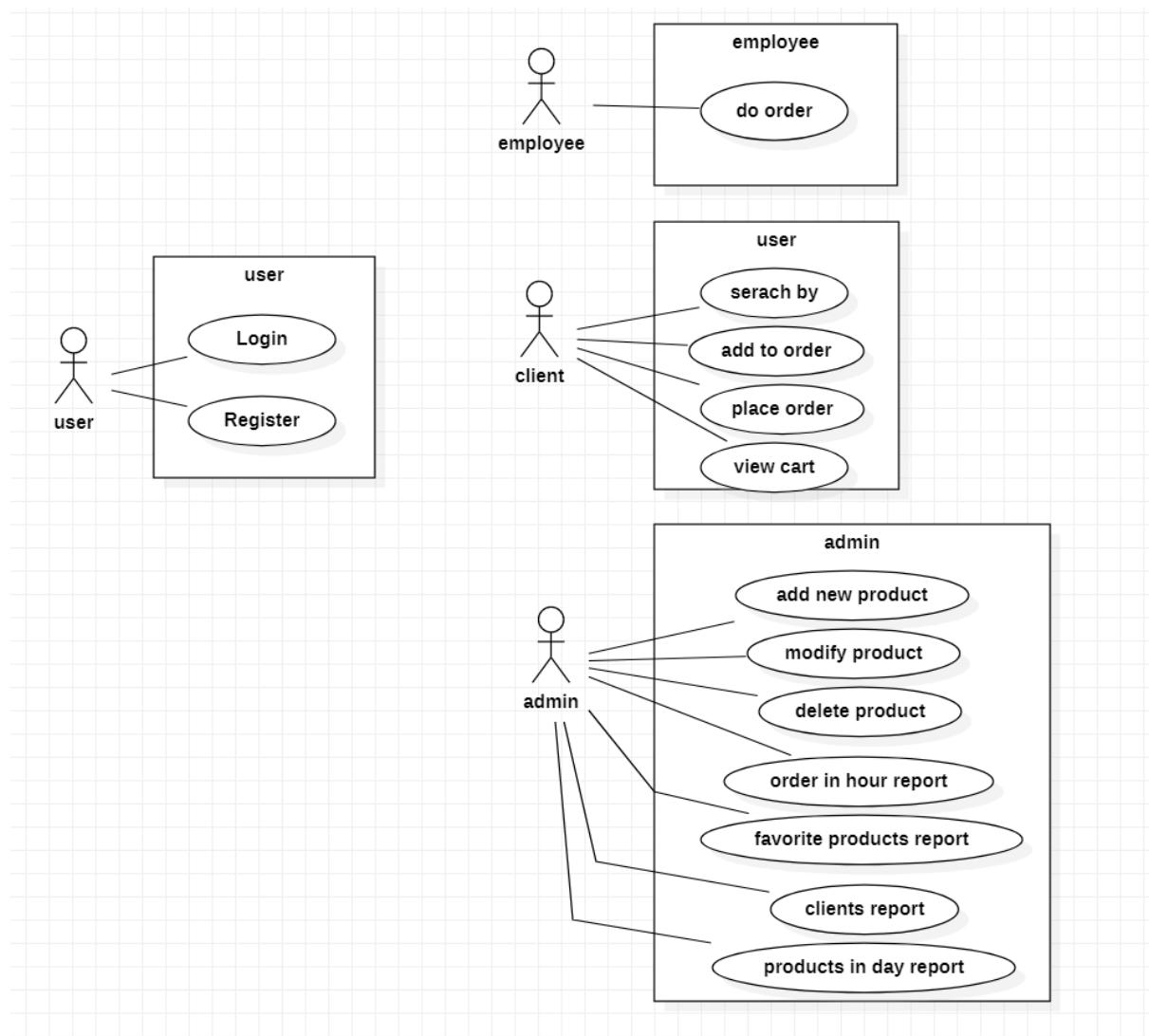
# Assignment Objectives

## Main objective

The main objective of this project is to design and implement an application for processing and managing client orders for a warehouse.

## Sub-objectives

- Analyze the problem and identify the functional and non-functional requirements
- Create the design of order management system
- Implement the order management system
- Test the order management in multiple scenarios



# Problem Analysis, Modeling, Scenarios, Use-cases

## Problem Analysis

In real-world circumstances, one of the main functions of a Warehouse Management System is to improve all stock control and tracking. The producer will be able to assist clients with goods and products according to the available possibilities.

In order to fulfill the properties of a warehouse management system, the application will be connected to a relational MySQL database, in which the data about the customers, the products and the orders will be stored. For each real-life entity, a table will be created (Customer, Order, Product). By storing data in a database, retrieving, updating, deleting data from the system transform into basic operations in database tables.

## Modeling

To reach its purpose, such an application should meet certain requirements like:

Functional requirements:

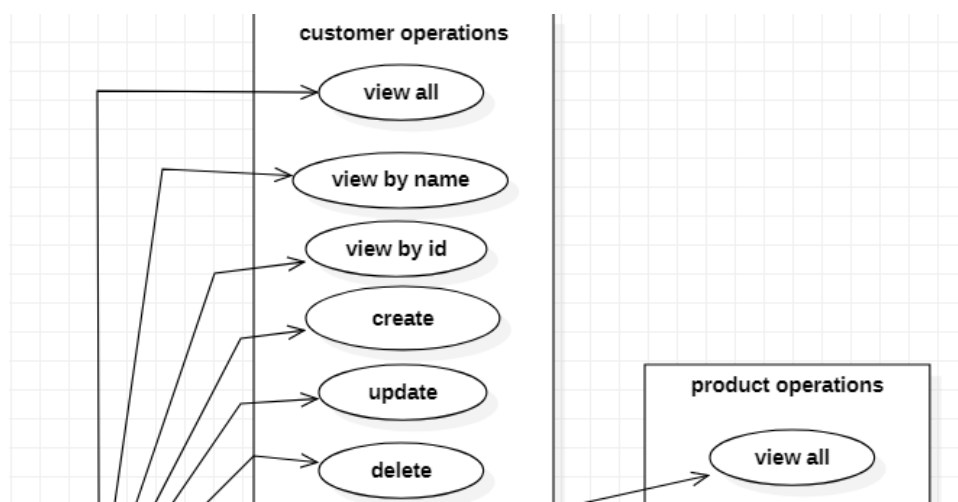
- The warehouse management application should allow the user to insert, update, delete all sorts of entities from the database (Order, Product, Customer)
- The warehouse management system should be started by the user, with the press of a button
- The warehouse management system should have the option to show the user data from tables.

Non-functional requirements:

- The simulator should be intuitive and easy to use.

## Use-cases and Scenarios

### Use Case Diagram



## Use Case Scenarios

### Customer

#### view all

Successful scenario

step 1: user presses the “viewAll” button.

step 2: the data from the table is shown in the GUI.

Unsuccessful scenario

step 1: user presses the “viewAll” button.

step 2: error might occur in the application ( connection problems with the database) and nothing shows up.

#### view by name

Successful scenario

step 1 : user introduces data in the “introduce name” filed.

step 2: user presses the “search” button.

step 3: the searched client is shown.

#### Unsuccessful scenario

step 1 : user introduces data in the “introduce name” filed.

step 2: user presses the “search” button.

step 3: the searched client name doesn't exist in the database, and an error is shown to the user.

step 4: go back to step 1.

#### create

##### Successful scenario

step 1: user introduces data in all the fields (id, name, phone, email,city, country).

step 2: user presses the “create” button.

step 3: client is created and the message of successful creation is shown on the screen.

##### Unsuccessful scenario

step 1: user introduces data in all the fields (id, name, phone, email,city, country).

step 2: user presses the “create” button.

step 3: data is wrong and a message about the wrong input is shown.

step 4: back to step 1.

#### update

##### Successful scenario

step 1: user introduces the “id”.

step 2: user presses the “show” button.

step 3: the data about the client is shown in the text fileds, and the user can edit the data.

step 4: the user presses the “update” button.

step 5 : the client is updated and a message of successful updating is displayed.

##### Unsuccessful scenario

step 1: user introduces the “id”.

step 2: user presses the “show” button.

step 3: the id is wrong and an error message is displayed.

step 4: back to step1.

step 1: user introduces the “id”.

step 2: user presses the “show” button.

step 3: the data about the client is shown in the text fields, and the user can edit the data.

step 4: the user presses the “update” button.

step 5: data is wrong and a message about the wrong input is shown.

step 6: back to step 1.

#### delete

##### Successful scenario

step 1: user introduces the “id”.

step 2: user presses the “delete” button.

step 3: the client is deleted and a message of successful deletion is shown.

Unsuccessful scenario

step 1: user introduces the “id”.

step 2: user presses the “delete” button.

step 3: the id is wrong and an error message is displayed.

step 4: back to step1.

## Order

### create

Successful scenario

step 1: user introduces id or selects the name of the customer.

step 2: user presses the “ok” button.

step 3: the order is created in the database, and the client can begin to add products.

step 4: the user begins to add products and quantity, presses the “add” button.

step 5: the user presses the “place order” button and the bill is generated.

Unsuccessful scenario

step 1: user introduces id or selects the name of the customer.

step 2: user presses the “ok” button.

step 3: the order is created in the database, and the client can begin to add products.

step 4: the user begins to add products and quantity, presses the “add” button.

step 5: the selected product is out of stock or is not enough for the client’s needs, and a message informing the user about the error is shown.

step 6: back to step 4.

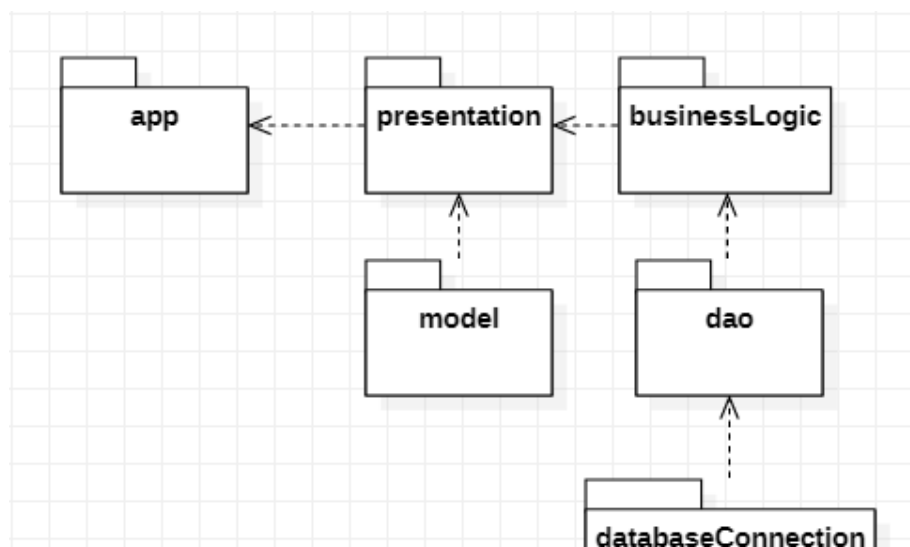
## Design

The application is structured in packages using a layered architecture. The purpose of using a layered architecture is to split the application into different layers. Each layer has a special purpose and calls functions of the layers below it.

To clearly represent this pattern, the application is divided into multiple packages.

## Packages

### Package Diagram



The app Package: contains only the **Main** class, which is run to start the application.

The model Package: contains the **Customer, Order, Product, OrderProduct** classes, the fundamental objects of the app. (one class for each database table)

The databaseConnection Package: contains one class **ConnectionFactory**, used to connect the application to the created MySQL database: “warehouse3”

The dao Package: contains the classes that manage the direct contact of the app with the database: **AbstractDao, CustomerDao, OrderDao, OrderProductDao** (one class for each database table)

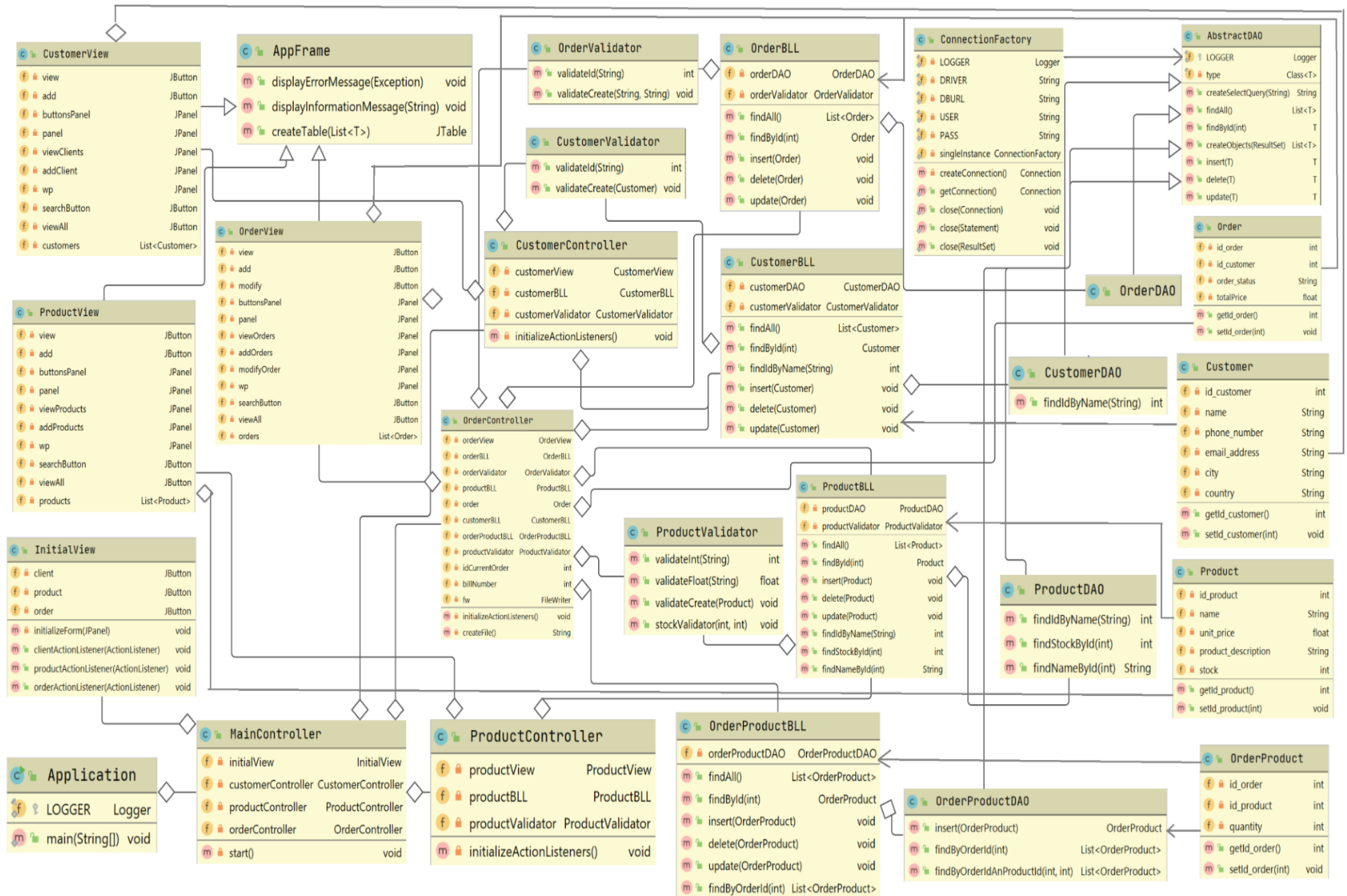
The businessLogic Package: contains the classes that implement the application logic, making use of the methods in the classes from the dao package: **CustomerBLL, OrderBLL, ProductBLL, OrderProductBLL**. And the validators for the actions performed in the app : **CustomerValidator, OrderValidator, ProductValidator**.

The presentation Package: contains the graphical user interface classes: **AppFrame, CustomerView, ProductView, OrderView, InitialView** and the controllers for each graphical user interface: **CustomerController, ProductController, OrderController** and **MainController**.

## Classes

### Class Diagram

The relationships between classes are represented in the following diagram on the next page.





# Class design

## Customer Class

The class is a simplified representation of a real customer, the fields of the class representing relevant data for completing an order placement, such as the name of the customer, a phone number, an email address, the country, and the city where the customer is from.

## Product Class

Is an entity that imitates a real life product, containing fields that are necessary for identifying a product in a warehouse, such as: the product name, a description of the product, the product price, the available stock of the product.

## Order Class

The class is a simplified representation of a real order made at a warehouse, the fields of the class representing relevant data for completing an order placement, such as the id of the customer, the status of the order, the total price of the order.

## OrderProduct Class

The class represents the connection between the order and the product tables, it is mainly used to solve the many to many relationship between the order and the product table in the database. It contains fields as the id of the order, the product and the quantity of that product in the order.

## Dao Classes

The Dao Classes manage the direct contact of the app with the database.

The AbstractDAO class is a generic class with methods that model the data from the database. Then, there are classes for each object of the app that might contain more specialized methods for that specific object.

## BLL Classes

The businessLogic classes implement the application logic, making use of the methods in the classes from the dao package. There is one class for each table from the database with the necessary implementation logic for the working of that object.

## ConnectionFactory Class

This class simply models the connection of the application with the database.

## View and Controller Classes

This set of classes will be used to implement the functionality of the user interface, the user interface connection with the logic of the application.

# Graphical User Interface - design

The graphical user interface should be intuitive and easy to understand and to use. The idea of the design is the following. When started the graphical user interface will present 3 buttons, each of them getting the access to one of the tables from the database.

Window Name

customer

product


order

For each interface, customer, product and order there will be different panels for modifying the table or viewing data from the table. Some examples are shown below.

## Mockup for the view section from each GUI

Window Name

name

Name (job title) ▲	Age ▼	Nickname	Employee ▼
Giacomo Guilizzoni Founder & CEO	40	Peldi	<input type="radio"/>
Marco Botton Tuttofare	38		<input checked="" type="checkbox"/>
Mariah Maclachlan Better Half	41	Patata	<input type="checkbox"/>
Valerie Liberty Head Chef	:)	Val	<input checked="" type="checkbox"/>
<a href="#">Data Grid Docs</a> 			<input type="checkbox"/>

Mockup for the modify client panel(very similar to the product view)

modify Customer

Some text

Button

Some text

Some text

Some text

Some text

Some text

Some text

Button

Button

Button

Mockup for the panel for adding orders

add order

Button

Button

Button

id

name

ComboBox

show

products

ComboBox

show

quantity

add

place order

Mockup for the panel of modifying orders

---

modify order

---

delete

id

modify

id

status

## Data Structures

Data Structure represents a way in which the data can be stored and organized so that it can be used efficiently and easily. The data structures used in this project are Lists.

The choice for the List data structure, more precisely, ArrayList, was made based on the fact that it comes with multiple helpful features and methods that make the process of writing code simpler and faster. Specific methods such as add, isEmpty or get(index) were used. For iterating through lists the iterator and list iterator were used and also the foreach instead of the simple for.

## Implementation

### Model Classes

Each class contains the exact same fields as the database table is represented, in the exact same order, this specific order is needed for the use of reflection techniques later in the implementation. Besides the attributes each class contains constructor (with parameters and without parameters) and getters and setters for all its attributes.

## ConnectionFactory Class

This class contains the name of the driver (initialized through reflection), the database location (DBURL), and the user and the password for accessing the MySQL Server .The class contains methods for creating a connection, getting an active connection and closing a connection, a Statement or a ResultSet.

initializing the driver

```
private static final Logger LOGGER = Logger.getLogger(ConnectionFactory.class.getName());
private static final String DRIVER = "com.mysql.cj.jdbc.Driver";
private static final String DBURL = "jdbc:mysql://localhost:3306/warehouse3";
private static final String USER = "root";
private static final String PASS = "root";
```

creating a connection

```
private ConnectionFactory() {
    try {
        Class.forName(DRIVER);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

## AbstractDao Class

This class contains methods that send queries to the server using Statements and it receives the results of the queries as ResultSet.

Standard format for query with 1 parameter is created

```
public String createSelectQuery(String field) {
    StringBuilder sb = new StringBuilder();
    sb.append("SELECT ");
    sb.append(" * ");
    sb.append(" FROM warehouse3.");
    sb.append(type.getSimpleName());
    sb.append(" WHERE " + field + " =?");
    return sb.toString();
}
```

The findById() and findAll() methods implement such queries.

Connecting to the database and constructing the query.

```
Connection connection = null;
PreparedStatement statement = null;
ResultSet resultSet = null;
StringBuilder query = new StringBuilder();
query.append("SELECT * FROM warehouse3." + type.getSimpleName());
```

Retrieving the result

```
connection = ConnectionFactory.getConnection();
statement = connection.prepareStatement(query.toString());
resultSet = statement.executeQuery();
return createObjects(resultSet);
```

Closing the connection after the query

```
ConnectionFactory.close(resultSet);
ConnectionFactory.close(statement);
ConnectionFactory.close(connection);
```

The method createObjects() is parsing aResult set into a list of objects using reflection.

```
while (resultSet.next()) {
    T instance = type.getDeclaredConstructor().newInstance();
    for (Field field : type.getDeclaredFields()) {
        Object value = resultSet.getObject(field.getName());
        System.out.println(field.getName());
        PropertyDescriptor propertyDescriptor = new PropertyDescriptor(field.getName(), type);
        Method method = propertyDescriptor.getWriteMethod();
        method.invoke(instance, value);
    }
    list.add(instance);
}
```

The methods insert(), update(), delete() also perform queries that are constructed with reflection techniques. The insert operation will be exemplified.

```
StringBuilder query = new StringBuilder();
query.append("INSERT INTO warehouse3." + type.getSimpleName() + " (");
```

Appending the field names

```
Field[] fields = type.getDeclaredFields();
for (int j = 1; j < fields.length ; j++) {
    Field field = fields[j];
    field.setAccessible(true);
    i++;
    query.append(field.getName());

    if( i < type.getDeclaredFields().length){
        query.append(",");
    }
}
```

## Appending the field values

```
for (int j = 1; j < fields.length; j++) {  
    Field field = fields[j];  
    field.setAccessible(true);  
    i++;  
    if(field.getType().isAssignableFrom(String.class)){  
        query.append("'" + field.get(t) + "'");  
    }else{  
        query.append(field.get(t));  
    }  
    if (i < type.getDeclaredFields().length){  
        query.append(",");  
    }  
}  
query.append(")");
```

## CustomerDao Class

Contains 1 method for querying the database findIdByName() implemented as explained above.

## ProductDao Class

Contains 3 methods for querying the database findIdByName(), findStockById(), findNameById() implemented as explained above.

## OrderProductDao

Contains a special insert function. For the other tables, when inserting a new entity the id is generated automatically in the database so the general insert() function from AbstractDao starts introducing data only from the second field of the table. For this table, the primary key consists of the id\_product and id\_order together, so that is why it also needs the first field to be inserted by the app, not by the database.

Contains 2 methods for querying the database findByIdAnProductId() and findById() implemented as explained above.

## CustomerBLL, ProductBLL, OrderBLL,

## OrderProductBLL

Contain methods that connect the DAO classes with the Controllers, methods are mainly named exactly like the methods from the DAO classes and only some validations are added.

Example of function with validation from ProductBLL

```
public int findStockById(int id){  
    if(productDAO.findStockById(id) == 0){  
        throw new RuntimeException("Out of stock!");  
    }  
    return productDAO.findStockById(id);  
}
```

## CustomerValidator, ProductValidator, OrderValidator Classes

These classes contain methods used for validating the data used in the logic of the application. Their methods throw errors when something wrong is encountered.

Method that checks if a given input is integer

```
public int validateInt(String s){
    int number;
    try{
        number = Integer.parseInt(s);
    }catch (NumberFormatException e){
        throw new RuntimeException("field must be an integer");
    }
    return number;
}
```

Method to check the stock of a product when an order is made

```
public void stockValidator(int stock, int quantity){
    if(stock < quantity) {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append("Not enough stock, only ");
        stringBuilder.append(stock + " available");
        throw new RuntimeException(stringBuilder.toString());
    }
}
```

Checking if all the fields are specified when creating or updating an object.

```
public void validateCreate(Customer customer){
    if (customer == null) {
        throw new RuntimeException("Client must not be null!");
    }else if(StringUtils.isBlank(customer.getName())){
        throw new RuntimeException("Name field is empty");
    }else if(StringUtils.isBlank(customer.getPhone_number())){
        throw new RuntimeException("Phone number field is empty");
    }else if(StringUtils.isBlank(customer.getCity())){
        throw new RuntimeException("City field is empty");
    }else if(StringUtils.isBlank(customer.getCountry())){
        throw new RuntimeException("Country field is empty");
    }else if(StringUtils.isBlank(customer.getEmail_address())){
        throw new RuntimeException("Email field is empty");
    }
}
```



# Graphical User Interface Implementation

The graphical user interface was implemented using Java Swing. The interface was designed according to the previously designed mockups.

The whole user interface is composed of 5 classes.

## The AppFrame Class

This class is a class that extends JFrame and will later be inherited by all the other interfaces that represent tables from the database. The class implements the following methods common to all the interfaces for displaying messages to the user.

```
public void displayErrorMessage(Exception exception) {
    if (exception != null) {
        String message = exception.getMessage();
        JOptionPane.showMessageDialog( parentComponent: this, message, title: "Error", JOptionPane.ERROR_MESSAGE);
    }
}

public void displayInformationMessage(String message) {
    if (!StringUtils.isBlank(message)) {
        JOptionPane.showMessageDialog( parentComponent: this, message, title: "Info", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

It also contains a method that receives a list of objects and generates the header of the table by extracting through reflection the object properties and then populates the table with the values of the elements from the list.

```
public JTable createTable(List<T> objects) {
```

setting the columns names

```
for(Field field : objects.get(0).getClass().getDeclaredFields()){
    field.setAccessible(true);
    columnNames[i] = field.getName();
    i++;
}
```

setting the tables rows

```

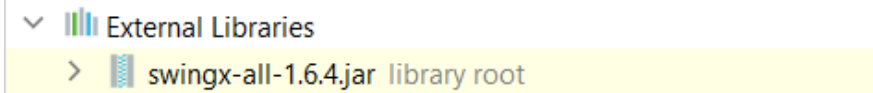
DefaultTableModel model = new DefaultTableModel(columnNames, rowCount: 0);
for (Object object : objects) {
    String[] result = new String[size];
    int j = 0;
    for (Field field: object.getClass().getDeclaredFields()) {
        field.setAccessible(true);
        try {
            result[j] = field.get(object).toString();
        } catch (Exception e) {
            e.printStackTrace();
        }
        j++;
    }
    model.addRow(result);
}

```

### The View Classes

These 3 classes use components from java.awt and javax.swing to implement the aspect of the interfaces. Components used are : JTexFields, JButtons, JLabel, JPanels, JComboBoxes, JScrollPane, JTable.

In the ProductView class for creating a JComboBox with auto search when writing an external jar downloaded from the internet was used(see bibliography).



```

cb =new JComboBox(init);
cb.setBounds( x: 55, y: 230, width: 200, height: 30);
cb.setEditable(true);
AutoCompleteDecorator.decorate(cb);

```

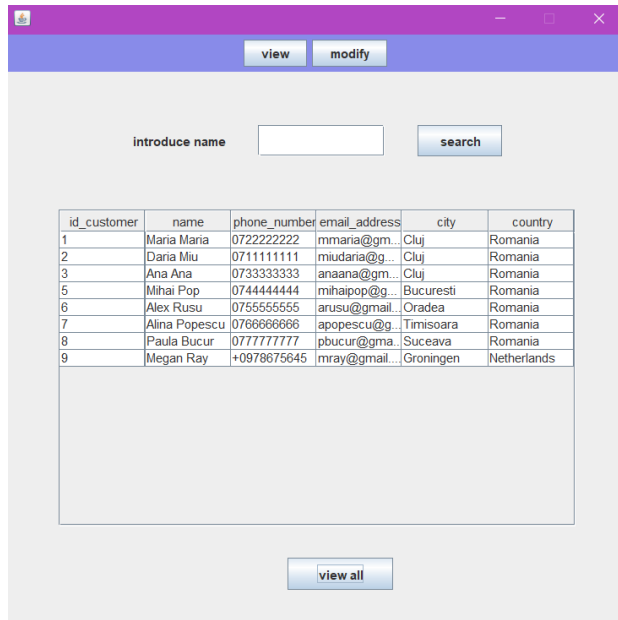
## CustomerController, ProductController, OrderController

The controller classes contain a method initializeActionListeners() that sets the action listeners to the button from the interface and retrieves the data from the text fields in the interface and sends the data to the BLL to be processed.

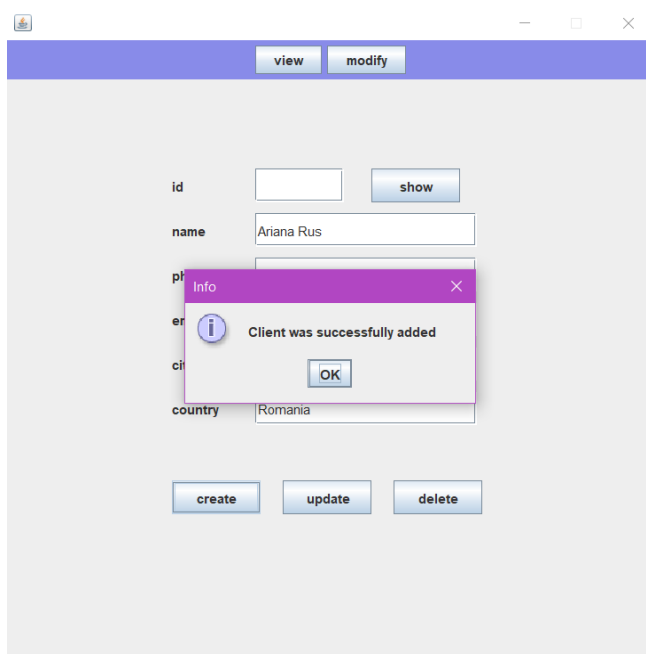
# Testing and Results

The application was tested for various scenarios while it was implemented by introducing data in the user interface and observing the produced result.

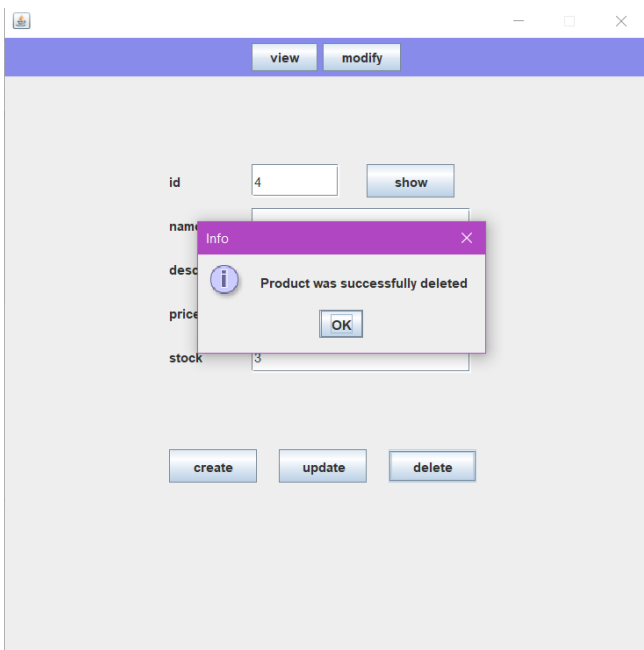
Result when pressing the “view All” button in customer window



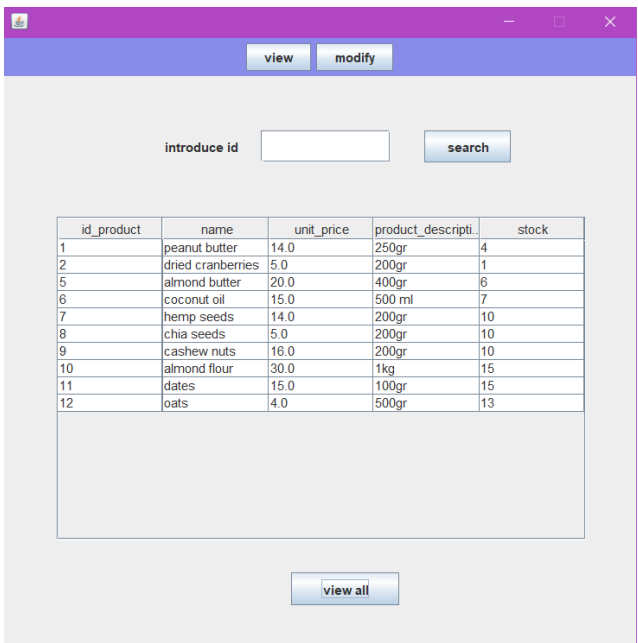
Results when creating a new client.



Results when deleting a product



Results for view all products.



Error resulted when the wanted quantity of a product is greater than the available stock.

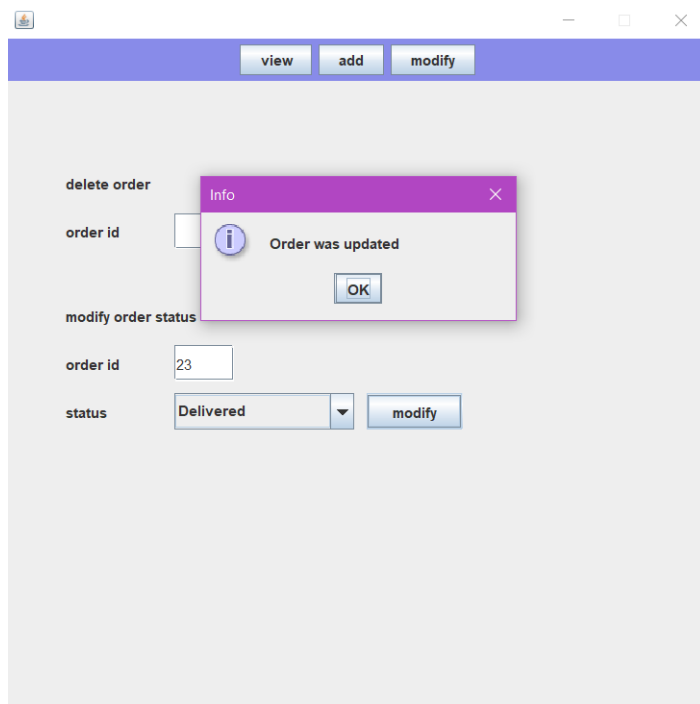
The screenshot shows a web application interface with a purple header bar containing 'view', 'add', and 'modify' buttons. Below the header, there are input fields for 'customer id' and 'customer name' (with a dropdown menu showing 'Megan Ray') and a 'show' button. An 'ok' button is also visible. A modal error dialog box is displayed in the center, with a red 'X' icon and the text 'Error' and 'Not enough stock, only 4 available'. Below the dialog, there is a 'place order' button. To the right of the dialog, there is a 'quantity' input field with the value '10' and an 'add' button.

Results for view all orders

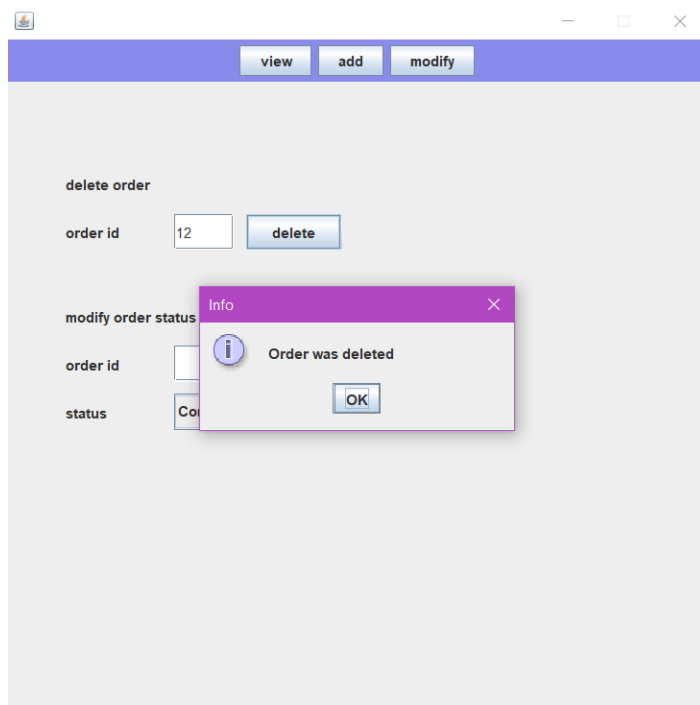
The screenshot shows a web application interface with a purple header bar containing 'view', 'add', and 'modify' buttons. Below the header, there is a search form with an 'introduce id' input field and a 'search' button. Below the search form, there is a table displaying order data. The table has four columns: 'id\_order', 'id\_customer', 'order\_status', and 'totalPrice'. Below the table, there is a 'view all' button.

id_order	id_customer	order_status	totalPrice
1	1	shipped	100.0
12	2	processing	0.0
13	1	processing	0.0
14	1	processing	0.0
15	2	processing	0.0
16	2	processing	0.0
23	1	processing	14.0
24	9	processing	90.0

Results when updating the status of on order.



Results when deleting an order.



## Conclusions

The task of implementing a warehouse management system application was successfully completed, since it can be seen that the application meets all the functional and non-functional requirements.

In conclusion, the warehouse management system application is a useful tool to manage the orders, clients and products from a warehouse database by implementing create, update, delete and view operations.

The modeling, design and implementation of this software are based on programming techniques and Object-Oriented-Programming principles, so the creation of this app was a great learning opportunity. I've learned about working with reflection techniques in java, since I've never used these concepts before. Also, implementing such a layered project was something new for me. I think that I have also improved my skills in writing OOP code. Moreover, facing and fixing all sorts of errors and bugs were an important step in learning.

## Further Developments

Some improvements that could make the software better would be:

- implementing the option to filter data from tables by all their fields.
- implementing a more friendly user interface containing pictures of the products.
- implementing the option to delete a product from order when creating the order.

## Bibliography

- Assignment 3 support presentation
- <https://drive.google.com/file/d/1ca5xxMsayog5fRgjFjvqVnuRDj6m-Hps/view>
- <https://www.jetbrains.com/help/idea/working-with-module-dependencies.html>
- <https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html>
- <https://www.baeldung.com/java-jdbc>
- <https://www.javatpoint.com/how-to-create-a-file-in-java>