MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

Order management application

- Project #2 -

Student: Dariana Lupea

Teacher: Delia Balaj

Group: 30425

Deadline: March 25, 2016

Table of contents

1. Introduction

1.1 Problem specification

The application I developed is based on the following requirements:

Consider an application OrderManagement for processing customer orders. The application uses (minimally) the following classes: Order, OPDept (Order Processing Department), Customer, Product, and Warehouse. The classes OPDept and Warehouse use a BinarySearchTree for storing orders.

a. Analyze the application domain, determine the structure and behavior of its classes, identify use cases.

b. Generate use case diagrams, an extended UML class diagram, two sequence diagrams and an activity diagram.

c. Implement and test the application classes. Use javadoc for documenting the classes.

d. Design, write and test a Java program for order management using the classes designed at question c). The program should include a set of utility operations such as under-stock, over-stock, totals, filters, etc.

1.2 Personal interpretation

First of all, from my point of view, this program represents a chance to learn how to deal with applications that require several use-cases, which means a more complex design and also more logical thinking.

There are some classes which must be used: Order, OPDept, Customer, Product and Warehouse. Of course these classes are not sufficient, that is why it is necessary to create others. One can approach this problem in different manners, according to the use-cases identified.

2. General aspects

2.1 Problem analysis

In order to obtain a good model, it is essential to start by understanding the basic concepts implied by the given problem and to try to find the best solution to it. Therefore, I started by identifying the situations when this application could be useful. In real life, such a program is useful in managing an online shop, from the warehouse point of view, but also as a customer. This was the main idea which guided me in developing the use-cases and the UML diagrams.

2.2 Modeling

As I mentioned before, I chose to implement an "Online Shop", on which the requested operations could be executed, by processing some orders. I started by analyzing how the given classes should be used, what attributes and methods should be present in each and what type of relationships could be established between them.

This part was not easy at all, but after getting everything at its right place, the problem seemed easier.

2.3 Scenarios and use cases

Before getting to the final version, it is obvious that I tried different scenarios and my program passed through different stages. I created first the POJO classes: *Order*, *OPD (Order Processing Department)*, *Product*, *Customer* and *Warehouse*, which were very important in developing the other classes. By considering two users (an **admin** and a **customer**) I designed a GUI which is different for each user, because the admin is responsible with the stock adjustment, while the regular user, which is the client, can only place orders, without having access to the background operations.

3. Projection

3.1. UML Diagrams

3.1.1 Use – case diagrams

A **use case diagram** at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved.

Mainly, the users of this application would be an admin, who can add products on stock, remove them or just update products and the customer, who can place an order and see order history. So, there are 2 use-case diagrams:

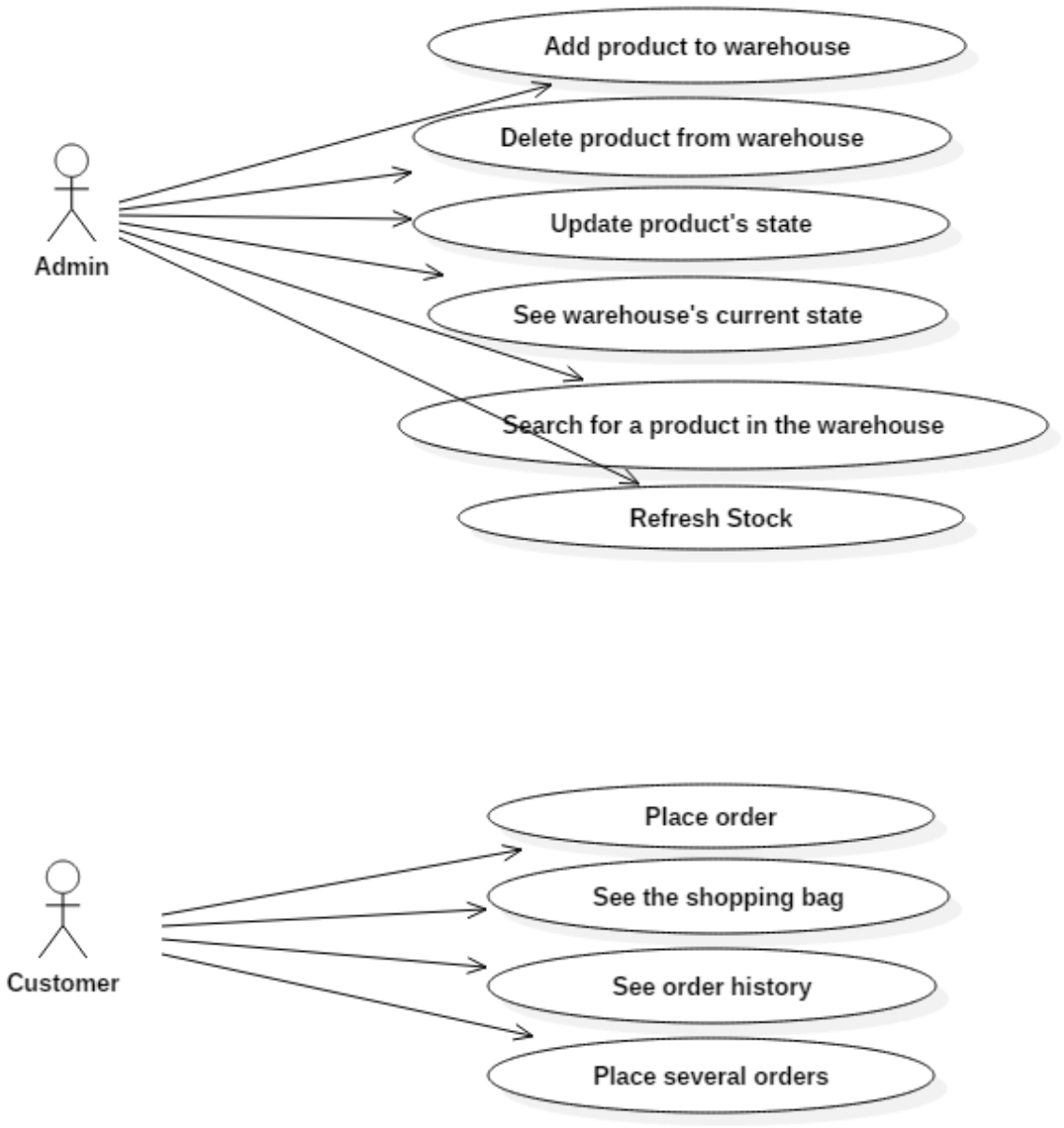


Fig. 1: Use-case diagrams

3.1.2. Class diagram

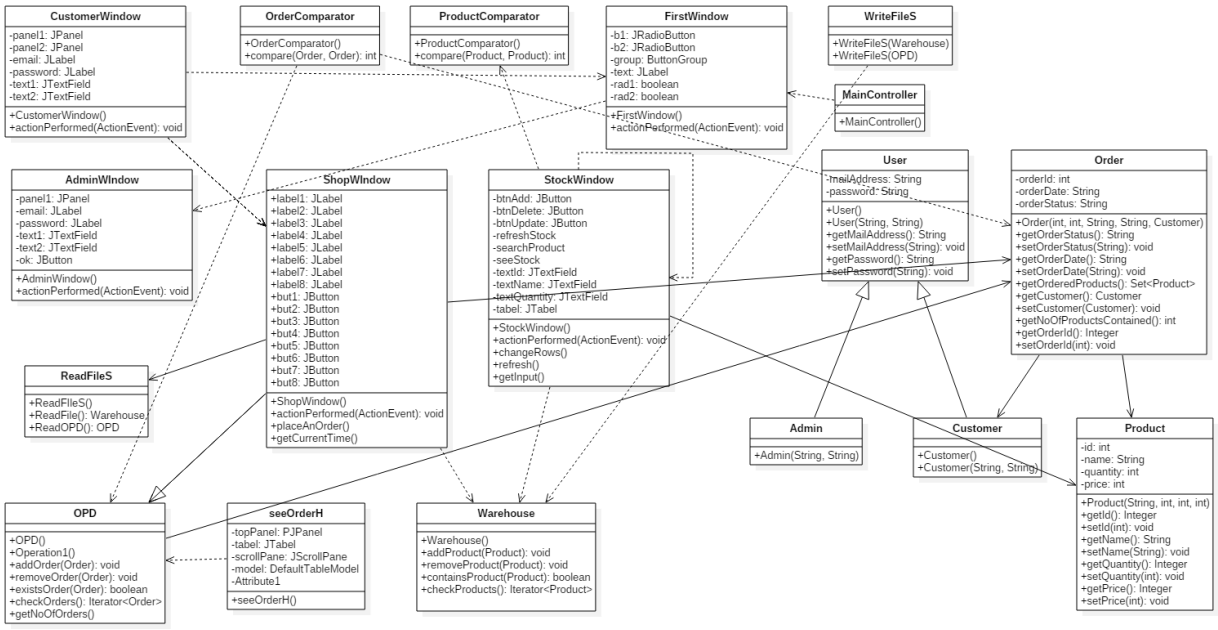


Fig. 2: Class diagram

3.1.3. Sequence Diagrams

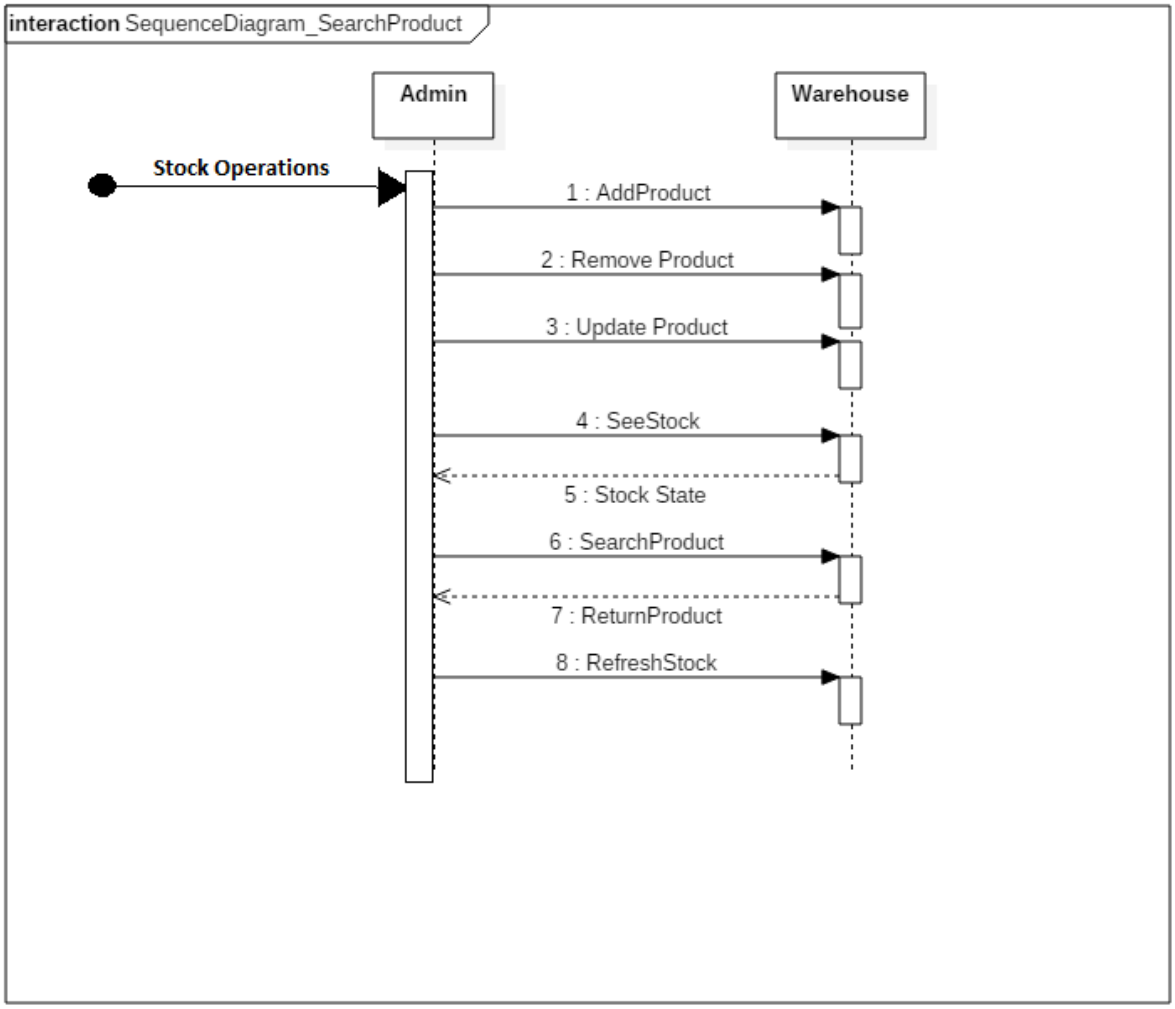


Fig. 3: Sequence diagram: Stock Operations

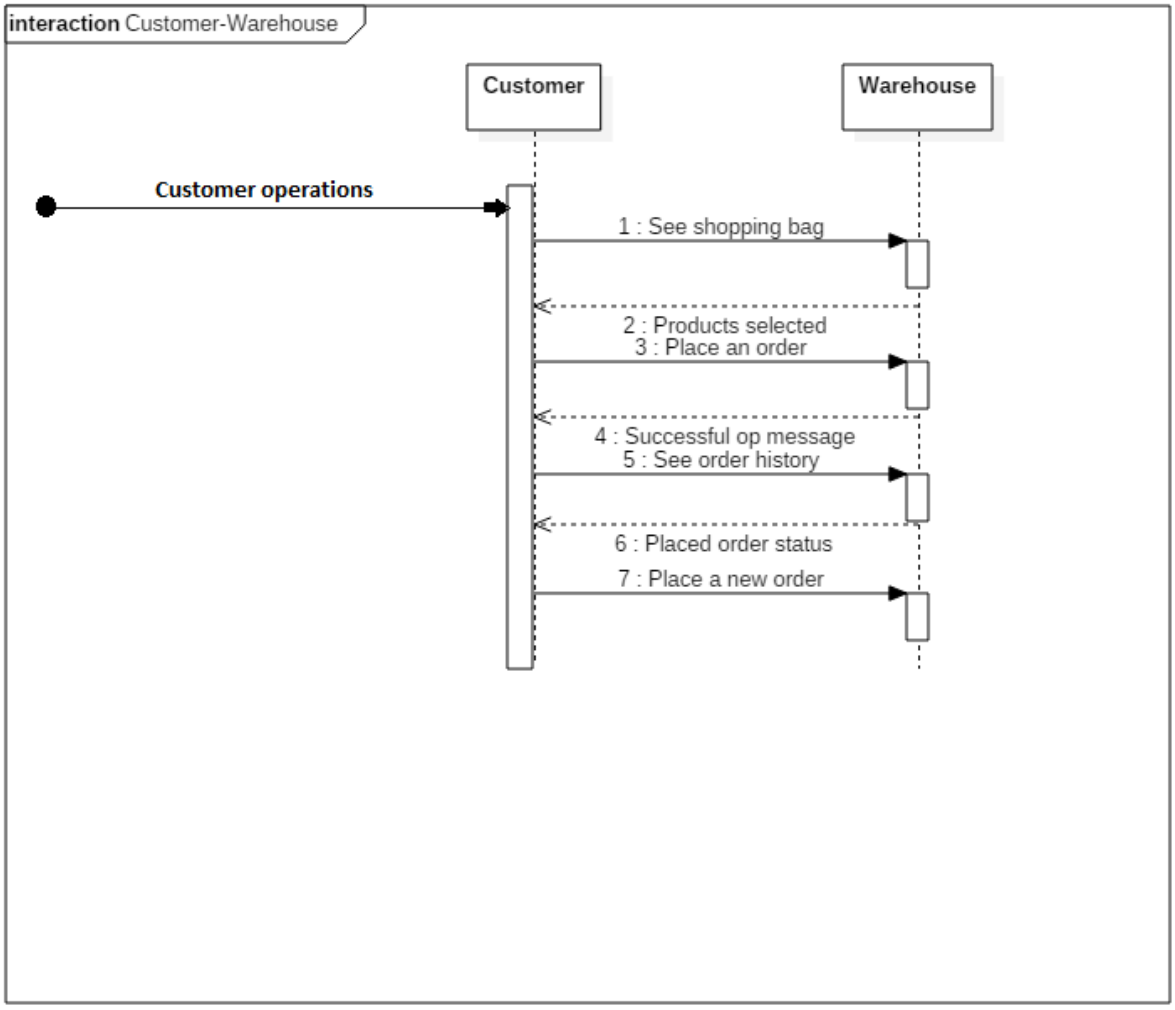


Fig. 4: Sequence diagram: Shop Operations

3.1.4. Activity Diagram

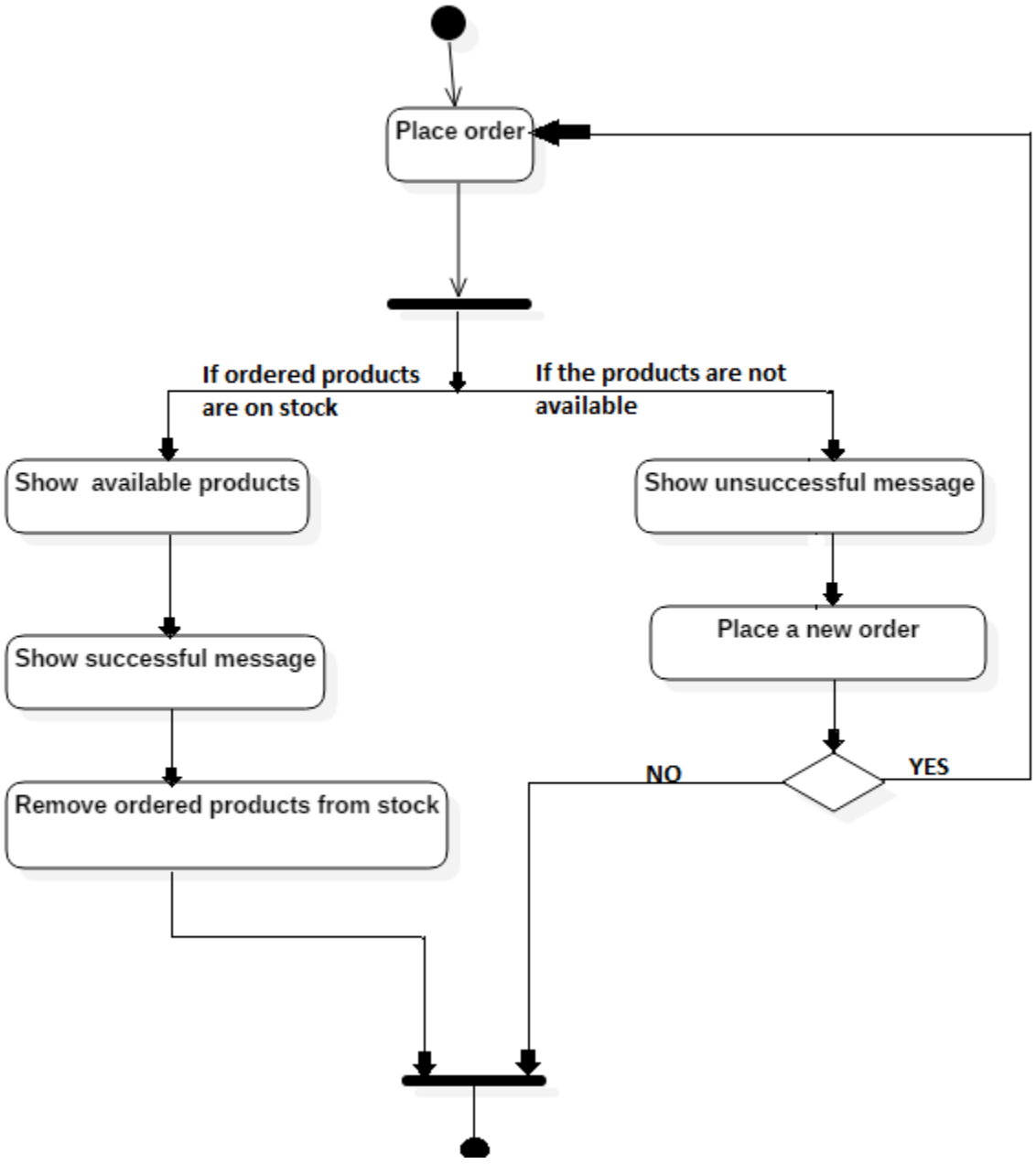


Fig. 5: Activity Diagram: Place order

3.2 Data Structures

The data structure my program is based on is the **TreeSet**. In the problem specification it is required that the classes *OPD* and *Warehouse* use a **BinarySearchTree** for storing orders. So, I decided to implement this behavior with the TreeSets. The property which is very useful in this situation is the fact that the objects are always stored in sorted, ascending order. This fact provides an easier and faster access to data.

In the Warehouse class, I used a TreeSet of products, for which I chose as an ordering criteria the name. In the other class, OPD, the TreeSet contains orders, which are sorted by their ID.

3.3 Class Projections

In the project design implementation I tried to use the **MVC** pattern. After several changes, I obtained the final version of the system's structure. It contains 5 packages, namely: **Comparators**, **Controller**, **Model, Serializing** and **View**. Each of these consists of other classes, which perform specific tasks. I will present them below:

- **Comparators** package:
  - **OrderComparator** class – used for comparing the orders by their id
  - **ProductComparator** class – used for comparing the orders by their names

- **Controller** package:
  - **MainController** class – creates and starts the application

- **Serializing** package:
  - **ReadFileS** class – for deserialize an object
  - **WriteFileS** class - for serialize an object

- **Model** package:
  - **Admin** class – POJO class for admin
  - **Customer** class – POJO class for customer
  - **OPD** class – Order Process Department, contains different operations on orders
  - **Order** class – POJO class for an order
  - **Product** class - POJO class for a product

- **User** class – a class that is extended by both **Admin** and **Customer**
- **Warehouse** class – contains operations performed on the stock

- **View** package:
  - **FirstWindow** class – the "welcome" interface, requires a selection between admin and customer
  - **AdminWindow** class – login frame for admins
  - **CustomerWindow** class - login frame for customers
  - **StockWindow** class – contains the warehouse state
  - **ShopWindow** class – only for customer, to place orders

3.4 Packages

- **3.4.1 Comparators** package:

It contains 2 classes, which implement *Comparator* interface: *OrderComparator* and *ProductComparator*, where the ordering criteria is established.

a) OrderComparator class

It contains only one method, the one that was overridden: *compare*, which takes 2 arguments of type *Order* and compares them by the order:

```
public class OrderComparator implements Comparator<Order>, Serializable {
    private static final long serialVersionUID = 1L;

    @Override
    public int compare(Order o1, Order o2) {
        return o1.getOrderId().compareTo(o2.getOrderId());
    }
```

b) ProductComparator class

In this class, it is performed the comparison between two products, by their names. The method used is the same as above.

3.4.2 Controller package

It contains only a class, which is responsible with starting the application: *MainController* class.

3.4.3 **Serializing** package

This package contains two classes, for serializing and deserializing objects.

a) **ReadFileS** class: this class is used for deserializing two types of object: *Warehouse* and *OPD*. It contains two methods, specific for each purpose:

- **public Warehouse ReadFile()**
- **public OPD ReadOPD ()**

b) **WriteFileS** class: it contains two constructors, defined by overloading. They are used for storing the information in the file.

- **public WriteFileS(Warehouse o)**
- **public WriteFileS(OPD o)**

3.4.4 **Model** package

This package contains several classes, where the entities are modeled: **Admin, Customer, OPD, Order, Product, User, Warehouse.**

a) **User** class**:** it is a POJO class, which is extended by Customer and User; it defines the common attributes of those classes. It has the following attributes:

- **private String mailAddress**
- **private String password**

It also contains some getters and setters for these attributes and a constructor, which is called from the child classes.

```
public User(String mailAddress, String password) {
    this.mailAddress = mailAddress;
    this.password = password;
}
```

b) **Admin** class: is a **User** (extend User) and it contains only a constructor, in order to call the constructor of the extended class by using **super** keyword.

c) **Customer** class**:** is a **User** (extend User) and it contains only a constructor, in order to call the constructor of the extended class by using **super** keyword.

d) **Product** class: it is a POJO class, which describes the basic properties of the object and it also contains a constructor and getters and setters for the given attributes.

```
public class Product implements Serializable{

    private static final long serialVersionUID = 1L;
    private int id;
    private String name;
    private int quantity;
    private int price;

    public Product(String name, int id, int quantity, int price){
        this.name = name;
        this.id = id;
        this.quantity = quantity;
        this.price = price;
```

e) Order class**:** an order has an ID, a date of order, a status, a number of ordered products and, of course, a set of ordered products:

- **private int orderId;**
- **private String orderDate**
- **private String orderStatus**
- **private Set<Product> orderedProducts**
- **private Customer customer**

There are some other methods defined in this class, but only for setting and getting the attributes of the Product.

f) **Warehouse** class**:**

It is the place where the products are added to the stock, removed or update. This is done by the admin, because only the admin has access to the warehouse. The admin can also search a certain product in the stock.
This class contains a set of Products:

- **private TreeSet<Product> products**

As I already mentioned the operations that can be performed on the stock, so these are the methods used to implement them:

- **public void addProduct(Product p)**
- **public void removeProduct(Product p)**

- **public Iterator<Product> checkProducts()**
- **public boolean contains(Product p)**

```
public class Warehouse implements Serializable {

    private static final long serialVersionUID = 1L;
    private TreeSet<Product> products;

    public Warehouse() {
        products = new TreeSet<Product>(new ProductComparator());
    }
```

g) OPD class: is the Order Process Department, where the placed orders are organized. It contains a **TreeSet** of Orders, which arranges the orders by their IDs. The orders can be added, removed and the customer can see the history of its orders.

3.4.5 **View** package

It is a very important package, because it builds up the **Graphical User Interface**, which allows a better understanding of the program and offers users the possibility to visualize how the applications works.
It contains several classes: *FirstWindow*, *AdminWindow*, *CustomerWindow*, *ShopWindow*, *StockWindow*.

a) FirstWindow class**:** the main aspect here is the fact that the user that starts the application has the possibility of login as an *admin* or as a *customer*. Both of them already have an account to login. The **top level container** is the frame, which extends **JFrame**.
This aspect is implemented by using some Swing components:

- JRadioButtons
- ButtonGroup

Also, some other components were used here:

- private JLabel **text:** used for writing the welcome text

- private JButton **ok:** when this button is pressed, the validity of the introduced data is checked

It also contains the **actionPerformed** method, which allows the connection between each swing component and the operations, which means

that the program behaves following the user's commands. As I already mentioned, the user needs to choose how to login and then press the "OK" button. Depending on this choice, the *Customer* window or the *Admin* window is opened.

```
@Override
public void actionPerformed(ActionEvent event) {
    Object c = (Object) event.getSource();

    if (c == b1) {
        rad1 = true;
        rad2 = false;
    }

    if (c == b2) {
        rad1 = false;
        rad2 = true;
    }
    if ((rad1 == true) && (event.getSource() == ok)) {
        dispose();
        new AdminWindow();
    } else if ((rad2 == true) && (event.getSource() == ok)) {
        dispose();
        new CustomerWindow();
    }
```

b) AdminWindow class**:** it also extend the **JFrame.** This window opens if the user wants to be an *Admin***. The following Swing components are used:**

- private **JPanel panel1**
- private **JLabel email, password**
- private **JTextField text1**
- private **JPasswordField text2**
- private **JButton ok**

If the user enters valid data (email and password), the warehouse frame appears and the access to the stock is allowed.

```
public class AdminWindow extends JFrame implements ActionListener {

    private static final long serialVersionUID = 1L;
    private JPanel panel1;
    private JLabel email, password;
    private JTextField text1;
    private JPasswordField text2;
    private JButton ok;
```

c) **CustomerWindow** class: only the customer has access to this class; it is similar to the the **AdminWindow** class, with the difference that it represents the way of entering to the "Shop" and place the order. It contains several fields:

- private **JPanel panel1, panel2**
- private **JLabel email, password**
- private **JTextField text1**
- private **JPasswordField text2**
- private **JButton ok**
- private **JLabel welcomeLabel**

As in the Admin login case, if the user enters a correct email and password, the ***ShopWindow*** is opened and the user has acces to the shop, in order to place new orders.

3.5 Algorithms

When designing this application, I did not use special algorithms. I have focused on satisfying the problem requirements and testing my own ideas in order to obtain the desired program. Anyway, some interesting and new feature for me that I have used for this application is the *Serializable* interface.

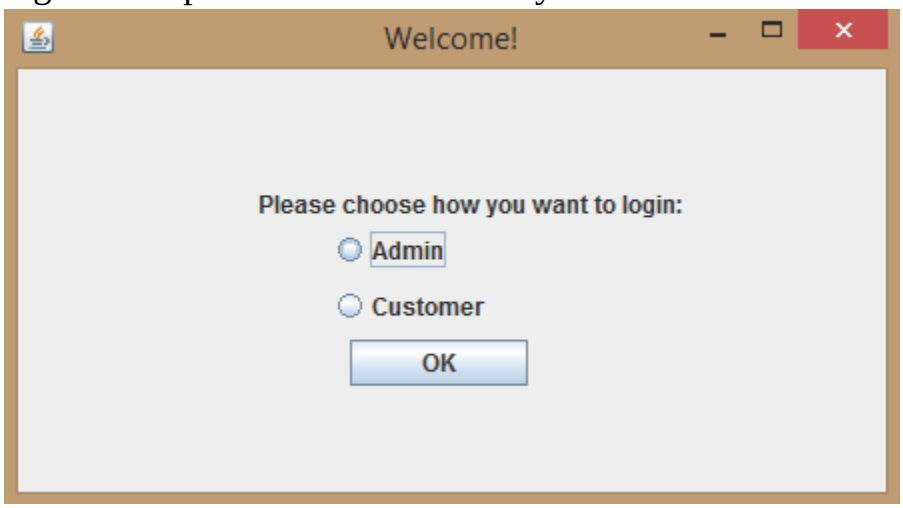
The classes that implement this interface are: OrderComparator, ProductComparator, *Customer*, *OPD*, *Product*, *Warehouse*, briefly the classes from which some piece of information needs to be stored.

3.6 Interface

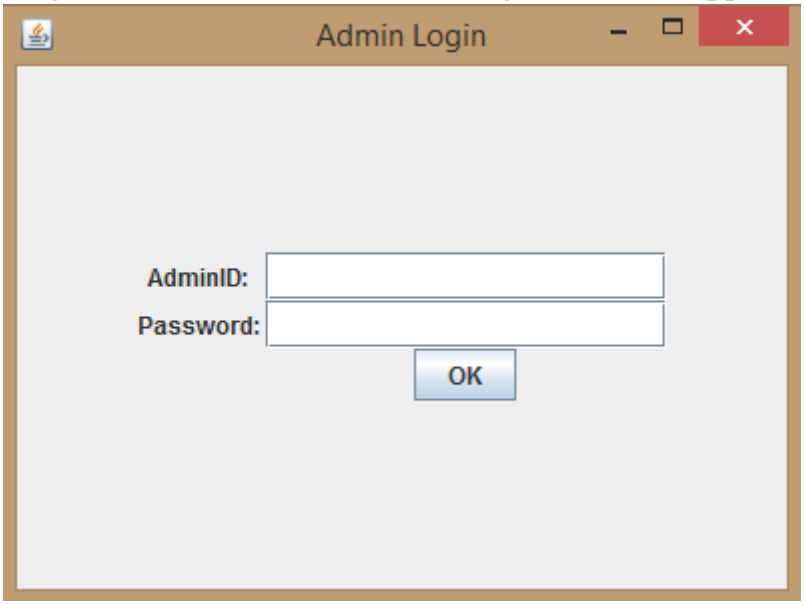
The Graphical User Interface I designed is user-friendly. Depending on user's type, there are two perspectives:

1) **admin** view
2) **customer** view

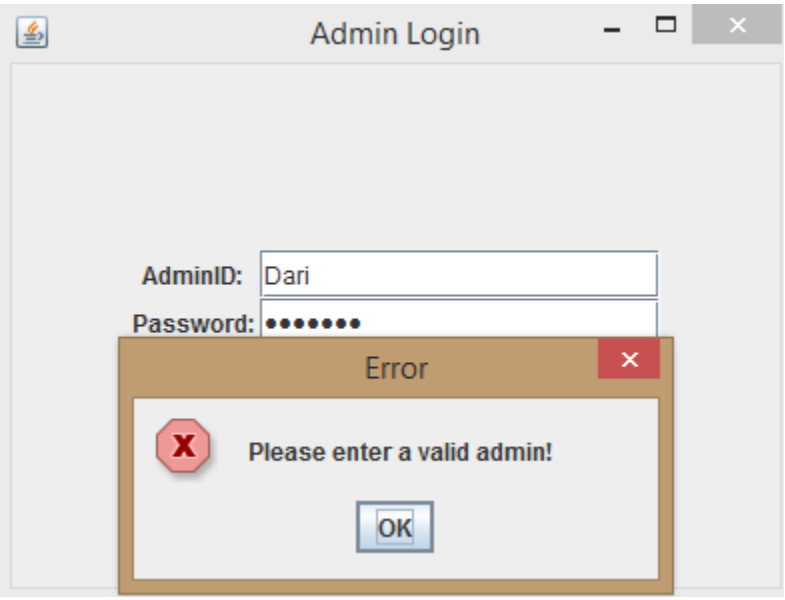
- The first frame is a "Welcome" window, which offers the possibility of login in the previous-mentioned ways:



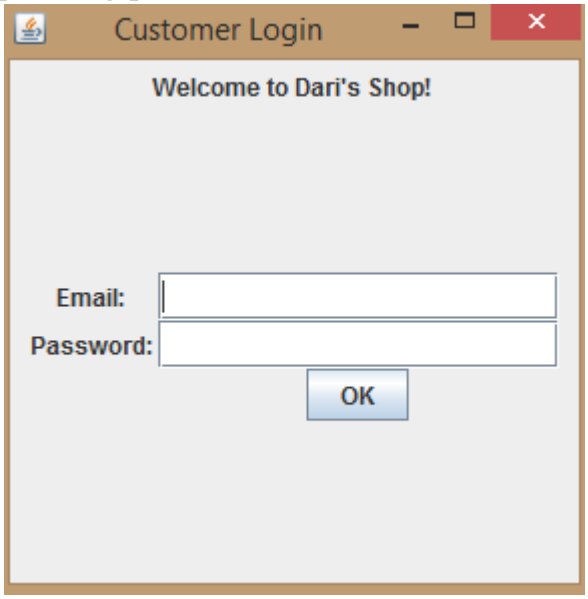
- If you choose to login as an **Admin**, the following frame will appear:



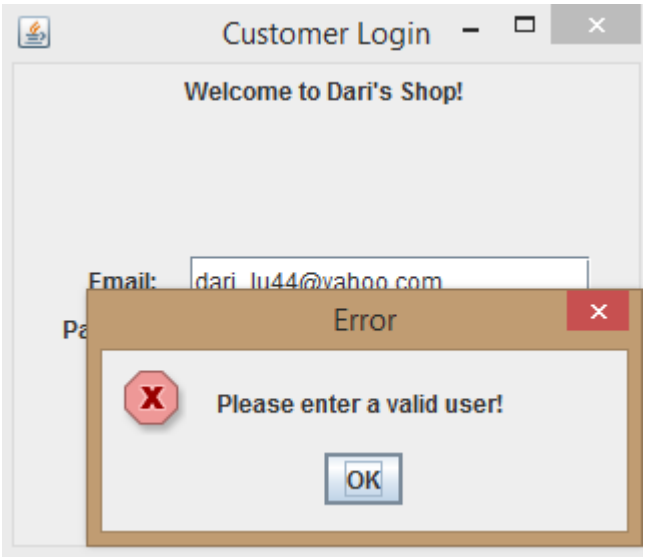
- User needs to enter a valid ID and a valid Password, in order to have access to the Warehouse. If one of the required fields is incorrect (the user does not exist or the password is typed wrong) the following error message shows up:



- Otherwise, if you choose to login as a **Customer**, you will be asked to enter your email and the corresponding password.



It is assumed that the user already has an account, so now he/she only needs to enter the email address and the password. If one of those is wrong, an error message appears:

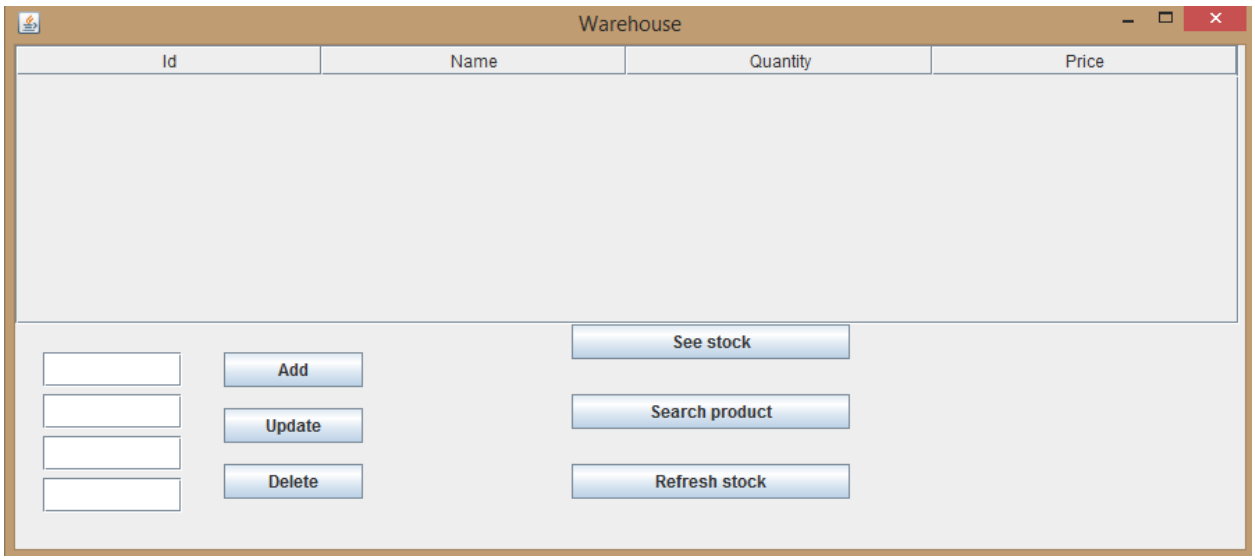


From now on, I will separately present both perspectives.

a) Admin

The admin is the only person who has access to the warehouse, where the products are stored. The following operations can be done by the admin:

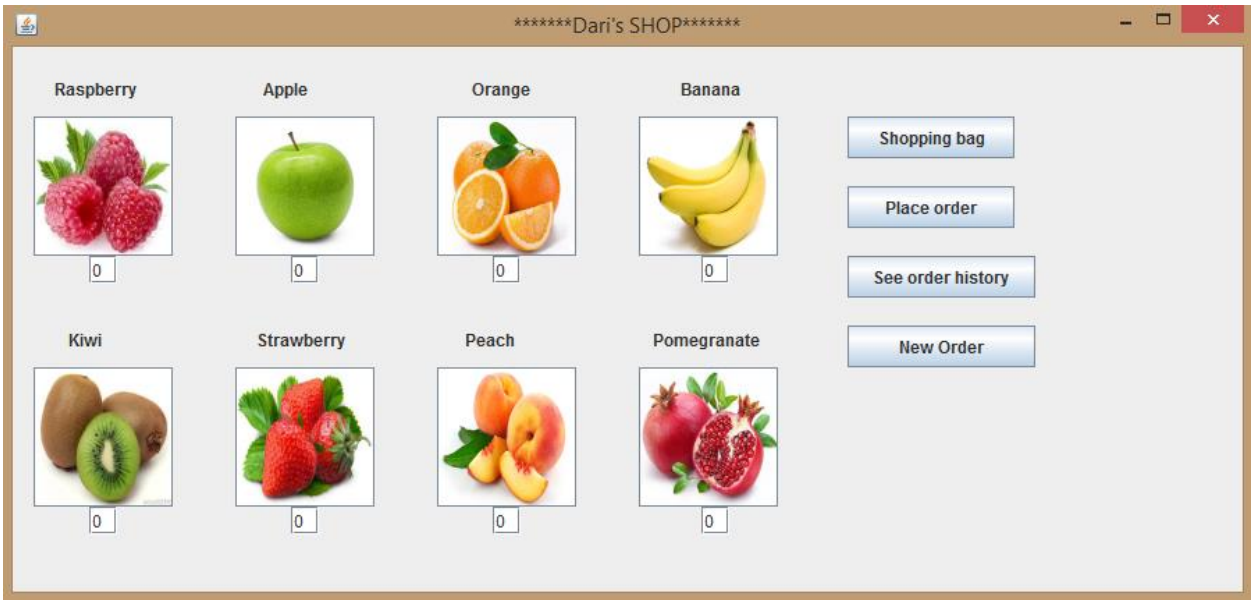
- Add a new product: **Add** button
- Remove an existing product: **Remove** button
- Update an existing product (change its id, name, quantity or price): **Update** button
- Search a certain product in stock: **Search** product
- Refresh the stock: **Refresh stock** button -> when this button is pressed, the warehouse state is saved



b) Customer

The customer only needs to choose the quantity of each product he/she wants and then press one of the buttons:

- **Shopping bag –** to see what product has added in the shopping list and what quantity of each
- **Place order –** to order the desired products, if they are in stock; otherwise, the lack of some products will be announced
- **See order history –** a customer can place many orders and he/she is able to see some order details of the processed orders at any time
- **New order –** it is actually a "refresh" of the shopping list



4. Implementation and testing

Regarding the implementation process, I used the **Eclipse IDE**. During the program development steps, many changes were made and as I started to have more and more methods I realized the importance of a good structure. I refer here to organization of the code in packages and classes. I consider that the code I wrote is understandable and reusable.

I have also tested each method, in the following way:

-> I used the console at first, for displaying the result
-> I initialized data with some appropriate values
-> I checked if the obtained result was the expected one
-> if YES, I continued the implementation process
-> if NO, I tried to figure out where is the error / problem coming from and solve it

5. Results

The results can be analyzed from different points of view. From my point of view, as developer of this application, I can say that this program fulfills all the requirements of the "client". I tried to implement the best version I could, considering the limited time and experience resources. From the user point of view, which has access only to the interface, I could assume that he/she would be satisfied by the functionalities offered by the Order Management Application.

6. Conclusion

To conclude, I can say that this project meant hard work, a lot of new things learned, focusing, development and creativity. Even if I encountered a lot of problems, I was able to fix them after all, by searching on the internet or asking a colleague for advice. I think that my application satisfies the requirements and the users will have at their disposal all its functionalities.

7. References:

- http://stackoverflow.com
- http://www.oracle.com/technetwork/articles/java/index-137868.html
- https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html
- https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html