Catering Management System

NUME : Denisa Bianca Deac

CONTENTS TABLE

1. Objectiv

The main objective of this assignment is to design and implement an application for managing the food orders for a catering company. It can be used by the administrator, by any client or employees. The application uses several design patterns, lambda expressions and streams to process the information.

The sub-objectives of the project are to analyze the problem and identify requirements, design, implement and test the orders management application.

2. Problem analysis, modelling, scenarios, use cases

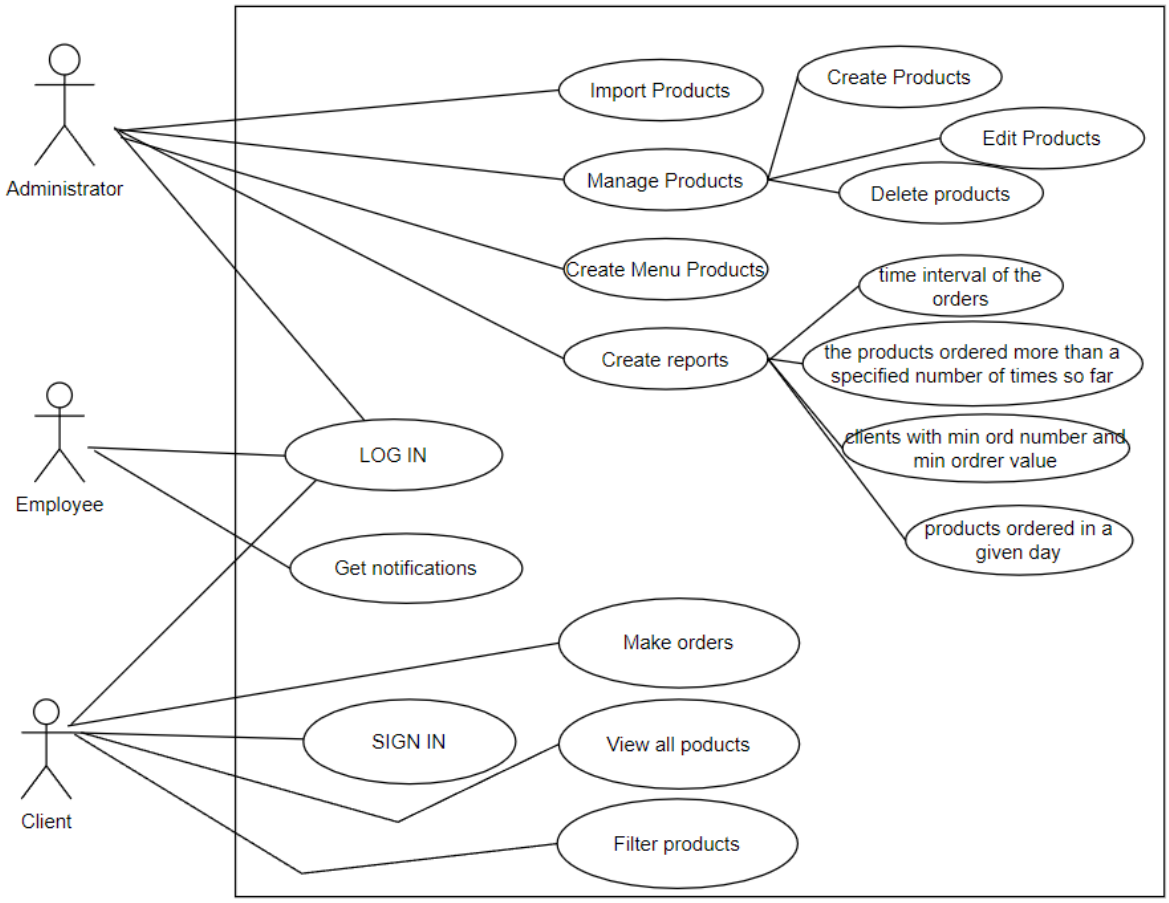
Problem analysis

The catering management system should give to the user the possibility to perform a different set of operations depending on the type of person it is a client, administrator or employee. Each category has a different set of operations.

The operations will be done through a user-friendly graphical interface from which the user will log as a client, administrator or employee or, in case the client doesn't have an account, he can create it. After that, depending on the category, a new window will replace the previous one with the necessary fields to input the data that will be used to perform the requested operation.

Scenarios and use cases

Each user has a different set of possible operations. All of them have the possibility to log in the system, although the client can also SIGN IN.



LOG-IN USE CASE

Use Case 1: log-in
Primary Actor: administrator, employee, client
Main Success Scenario:
1. Introduce the credentials
2. A new window corresponding to the needed category will appear

Alternative Sequence: Wrong credentials/Wrong category
1. Error message informing the credentials are not registered

FOR ADMINISTRATOR

Use Case 1: import products
Primary Actor: administrator
Main Success Scenario:
1. The administrator presses the “Import Products” button
2. A message will appear on the screen transmitting that the operation was succesfull

Alternative Sequence: Error while importing
1. The user left one or many fields empty
2. Dialog error message will be displayed

Use Case 2: create product
Primary Actor: administrator
Main Success Scenario:
1. The administrator inserts the information about the new product
2. Presses the create button

Alternative Sequence: Error while creating
1. The new products won’t appear on the table

Use Case 3: delete products
Primary Actor: administrator
Main Success Scenario:
1. The administrator inserts the id of the new product
2. Presses the delete button

Alternative Sequence: Error while deleting
1. The item won’t be deleted from the table

Use Case 4 : edit products
Primary Actor: administrator
Main Success Scenario:
1. The administrator inserts the id of the new product and the new data
2. Presses the edit button

Alternative Sequence: Error while editing
1. The item won’t be edited in the table

Use Case 5 : create report
Primary Actor: administrator
Main Success Scenario:
1. The administrator inserts the needed information
2. Presses the report button
3. Displays a message to notify the success of the operation

Alternative Sequence: Error while creating report
1. Error message on console

FOR CLIENT

Use Case 1: view products
Primary Actor: client
Main Success Scenario:
1. The client presses the "View Products" button
2. A table will appear on the window

Alternative Sequence:
1. The table will not appear

Use Case 2: filter products
Primary Actor: client
Main Success Scenario:
1. The client inserts all the filter criteria he wants
2. The client presses the "Filter Products" button
3. A table will appear on the window

Alternative Sequence:
1. The table will not appear

Use Case 3: make order
Primary Actor: client
Main Success Scenario:
1. Input the id of the order
2. The item will be added to cart by pressing the corresponding button as many times as the desired number of products
3. Press the "Order" button to create the new order
4. A message will appear on the screen to notify that the bill was created

Alternative Sequence:
1. The order did not register the products

Use Case 4: sign-in
Primary Actor: client
Main Success Scenario:
1. Input the new credentials
2. Press button
3. Window for the client category will appear

FOR EMPLOYEE
Use Case 1: view notification
Primary Actor: client
Main Success Scenario:
1. After log-in, a window with all the notifications is displayed

Alternative Sequence:
1. No notifications on the screen

Functional requirements:
1. The application should allow the user log to their account corresponding to their category
2. The application should allow the user to input data for all the needed operations
4. The application should generate a bill for every order created
5. The application should display some "success messages:

Non-Functional requirements:
1. The application should be intuitive for the user
2. The application should be easy to use by the user
3. The application should inform the user in case of any errors or input mistakes

3. Design

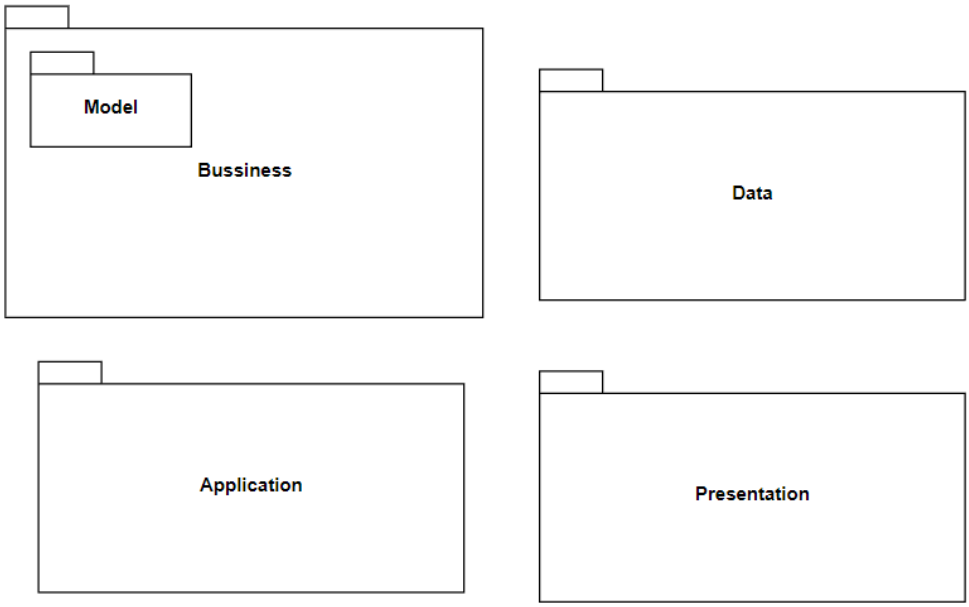
My application is divided on 5 packages, one of them having a package as a "sub-package".

Presentation is the package containing the GUI classes, all for classes for Administrator, Client, Employee, Sign-In and Log-In windows.

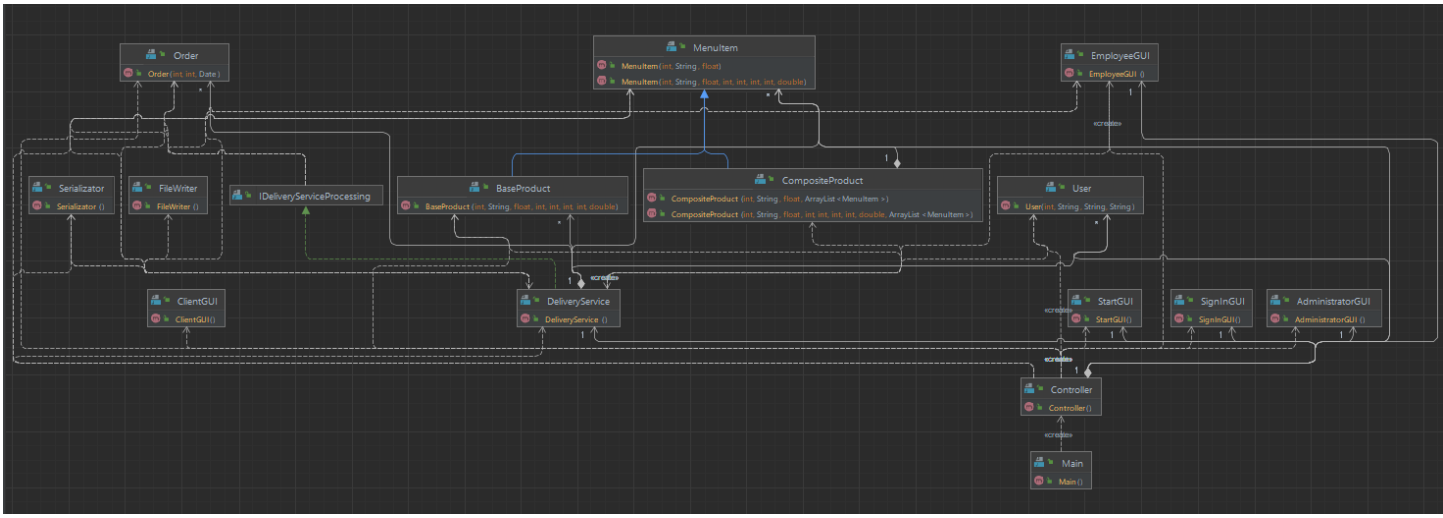
Application: represents the control of the application, having the main class and the Controller in which the action listeners are created and the linkage between the GUI and the logic operations is done.

Data is represented by the 2 classes that work as "auxiliars": The fileWritter that has one method for creating .txt files and the Serializator class which performs the serialization of the data.

The last main package, Business, contains a small package having the classes representing the entities, called Model. The client, user, order and product are present here. In the business package are created the most important classes: the one performing the logical operations for each category of user. The details about these classes will be presented in the following paragraphs.



The concept behind designing this order management system is the following: each type of user will choose their category by entering their credentials and logging into their account. From each new window, they can perform the wanted operation. This process is created by having the following dependencies between classes:



My design for this project is made by combining the Composite Design Pattern, the Observer Design Pattern and Design by contract. By using them, every class work in their best way possible. Each design is appropriate for a certain part of the project.

For the data to be stored even after the end of the execution, I used serialization.

ArrayLists and HahsMaps are used to store the necessary data. Since each order has plenty products ordered, this way of storing the order details is really efficient: a really easy way to access data.

4. Implementation

MODEL CLASSES

These classes are "modelling" the entities needed for this project. For the menuItem was used the Composition Design Pattern. The menuItem is an abstract class that is extended by both BaseProduct and compositeProduct.

```
public class CompositeProduct extends MenuItem {

    private ArrayList<MenuItem> menuProducts;

    public CompositeProduct( int id, String title, float rating, ArrayList<MenuItem> menuProducts) {
        super(id, title, rating);
        this.menuProducts = menuProducts;
    }
```

```
public double compuntePrice() {

    double price=0;
     Iterator<MenuItem> myIterator = menuProducts.iterator();
     while(myIterator.hasNext())
     {
         MenuItem curentItem = myIterator.next();
         price+=curentItem.getPrice();
     }
     setPrice(price);
     return price;

}

public void computeMenuDetails(){

    int calories=0, protein=0, fat=0, sodium=0;
    int[] values = new int[4];

    for(MenuItem m: this.menuProducts){
        if(m != null) {
            calories += m.getCalories();
            protein += m.getProtein();
            fat += m.getFat();
            sodium += m.getSodium();
        }
    }
    setCalories(values[0]);
    setProtein(values[1]);
    setFat(values[2]);
    setSodium(values[3]);
```

```
public abstract class MenuItem implements Serializable {

    private int id;
    private String  title;
    private float rating;
    private int calories;
    private int protein;
    private int fat;
    private int sodium;
    private double price;

    public abstract double compuntePrice();
```

The other entities are Order and User in which the info about the orders and the user are kept. The user keeps the credentials of each exixtent account and the category in which it stands.

The order entity stores only the id of the order, client and the date. The information about the products ordered within that order are stored in the DeliveryService class in a hash map (the key being the order and the value being a list of products).

```java
public class Order implements Serializable {
    private int orderID;
    private int clientID;
    private Date orderDate;

public class User implements Serializable {
    private int id;
    private String username;
    private String password;
    private String userType;
```

BUSINESS

IDeliveryServiceProcessing is the interface that defines the methods that the DeliveryService class will inherit.

The design by contract was use for these classes: the conditions and preconditions were made in the interface and the assertions in the DeliveryService class.

```java
public interface IDeliveryServiceProcessing {

    /**
     * @pre menuList.isEmpty()
     * @pre productList.isEmpty()
     * @post !menuList.isEmpty()
     * @post !productList.isEmpty()
     */
    public  void importProducts();

    /**
     * @pre list.isEmpty()
     *
     * @post !list.isEmpty()
     */
```

The DeliveryService class stores all the important data about the products and the orders. In this class are implemented all the operation that needed to be performed by the user.

Most of the methods, as import, create reports and filter products were implemented using streams and lambdas.

Here are found all the methods for:importing products, creating, editing, deliting a product, creating a composite product, making an order, make all 4 types of reports, filtering the products by some criteria, and viewing all the products existend.

Beside them, I made some other helpfull methods like: finding the next available id, fining by name an item, by id, importing users.

The initial set of users are stored in a csv file from which the data was importing in a similar way as for the products importation. This method is used only when first “running” the application, because after that the users will be deserialized in order to also use the new created accounds.

```java
@Override
public void importProducts() {

    assert menuList.isEmpty();
    assert productsList.isEmpty();

    Set<String> uniqueSet = new HashSet<>();
    List<String[]> lines = null;

    try {
        lines = Files.lines(Paths.get( first: "products.csv")) .skip(1) .map(line -> line.split( regex: ",")) .toList();

        List<String[]> finalLines = lines;
        productsList = (ArrayList<BaseProduct>) lines.stream() Stream<String[]>
                .filter(m -> uniqueSet.add(m[0]))
                .map(m -> new BaseProduct( id: finalLines.indexOf(m)+1, m[0],Float.parseFloat(m[1]), Integer.parseInt(m[2]),
                        Integer.parseInt(m[3]), Integer.parseInt(m[4]), Integer.parseInt(m[5]), Integer.parseInt(m[6])) )
                .collect(toList());
        menuList.addAll(productsList);
    } catch (IOException e) {
        e.printStackTrace();
    }
    Serializator.serialize( deliveryService: this);
    assert !menuList.isEmpty();
    assert !productsList.isEmpty();
    JOptionPane.showMessageDialog( parentComponent: null,  message: "All products were imported");
}
```

```java
@Override
public void filter3( int ord, double value){
    assert ord >= 0;
    assert value >= 0;
    Map<Integer,Long> map = new HashMap<>();
    StringBuilder sb = new StringBuilder();
    sb.append("Clients that have ordered more than "+ ord+" times so far with the value of the order grea


    map = orderList.stream()
            .filter(o -> getTotalOrderPrice(o, ordersMap.get(o)) > value)
            .collect(Collectors.groupingBy(Order::getClientID,
            Collectors.counting()));

    map.entrySet().stream().forEach( pair -> {
        System.out.println("client id: "+pair.getKey() + "   products: "+ pair.getValue()+"\n");
            if( pair.getValue() > ord){
                sb.append("\nClient with id "+ pair.getKey()+ " has ordered "+pair.getValue()+"\n");
            }
    });
    reportId++;
    String reportTitle = "Report_3_"+reportId+".txt";
    FileWriter.write(String.valueOf(sb),reportTitle);
    Serializator.serialize( deliveryService: this);
}
```

APPLICATION

The Controller class manipulates the data from the GUI and performes the operations on them by adding the Action Listeners for the buttons.

Each button represents an operation and each class have its own listener.

In the controller the serialization os the application is started. At the firs “run” of the program, the users must be impored using the method presented above. After that, only the serialisation - deserialisation (that willl be described) next will be used to persist the data.

Replacing importUsers, the method getUsers will take place. It will get the new deserialisez data from the business package (DeliveryService class) and will store it for later usage. For the log-in listener the list with all the users is neded to check for the category of each person that want to log in.

```
public void addSignInListener(SignInGUI signInGUI){
    signInGUI.addListeners(e-> {
        if(e.getSource() == signInGUI.getSignIn()){
            int id = users.size()+1;
            User p = new User(id, signInGUI.getUsername(), signInGUI.getPassword(), userType: "client");
            users.add(p);
            deliveryService.addNewUser(p);
            JOptionPane.showMessageDialog( parentComponent: null, message: "Account Created with Success");

            ClientGUI clientGUI = new ClientGUI();
            clientGUI.setVisible(true);
            addClientListeners(clientGUI);
            signInGUI.setVisible(false);
        }
        if(e.getSource() == signInGUI.getBack()){
            startGUI.setVisible(true);
            signInGUI.setVisible(false);
        }
    });
}
```

PRESENTATION

The interaction with the user is made through a friendly, explicit, graphical user interface. Each user will log-in and will be sent to the next window containing the possible operations to be computed. For the Employee to receive notifications, the Observer designed pattern was used, where the GUI is the observer and the DeliveryService the observable.

```
public class EmployeeGUI extends JFrame implements Observer {

    private final JLabel titleLabel = new JLabel( text: " EMPLOYEE ");
    private static JTextArea result = new JTextArea();
    private static JScrollPane scrollBar;
    private final JButton back = new JButton( text: "BACK");
```

DATA

The data package contains the classes that are used to save the data from the project. The Serializator class has one method for serializing the data from the DeliveryService and one for the opposite operation = derealization.

Each time a modification is made, the data from the class is serialized. The information is stored in a .ser file from which, at the beginning of the program the data will be deserialized. For this to happen without errors, each entity-type class must implement java.io.Serializable.

```java
public class Serializator {

    public static void serialize(DeliveryService deliveryService) {
        try {
            FileOutputStream fileOutput = new FileOutputStream( name: "Info.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOutput);
            out.writeObject(deliveryService);
            out.close();
            fileOutput.close();
            System.out.printf("Serialized data is saved in Info.ser");
        } catch (IOException i) {
            i.printStackTrace();
        }
    }
```

```java
public static DeliveryService deserialize() {
    DeliveryService deliveryService = null;

    try {
        FileInputStream fileIn = new FileInputStream( name: "Info.ser");
        ObjectInputStream in = new ObjectInputStream(fileIn);
        deliveryService = (DeliveryService) in.readObject();
        in.close();
        fileIn.close();
        return deliveryService;
    } catch (IOException i) {
        i.printStackTrace();
        return null;
    } catch (ClassNotFoundException c) {
        System.out.println("Employee class not found");
        c.printStackTrace();
        return null;
    }
}
```

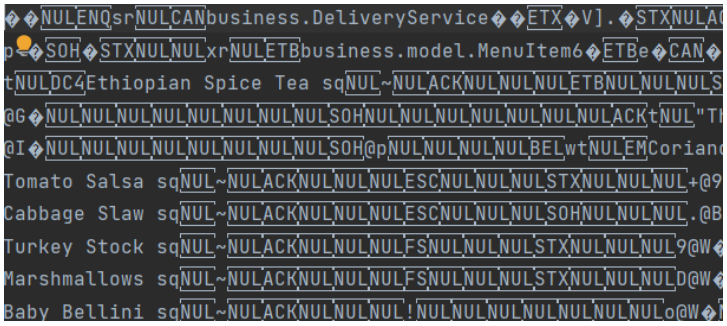
The method in the FileWriter takes a string as parameter and the name of the file to be opened/created and write ethere the given string. This method is used togenerate the bills and all the reports.

```java
public static void write(String s, String fileName){

    try {
        java.io.FileWriter file = new java.io.FileWriter(fileName);
        file.write(s);
        file.close();
        JOptionPane.showMessageDialog( parentComponent: null,  message: fileName+" Generated With Success!");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
```

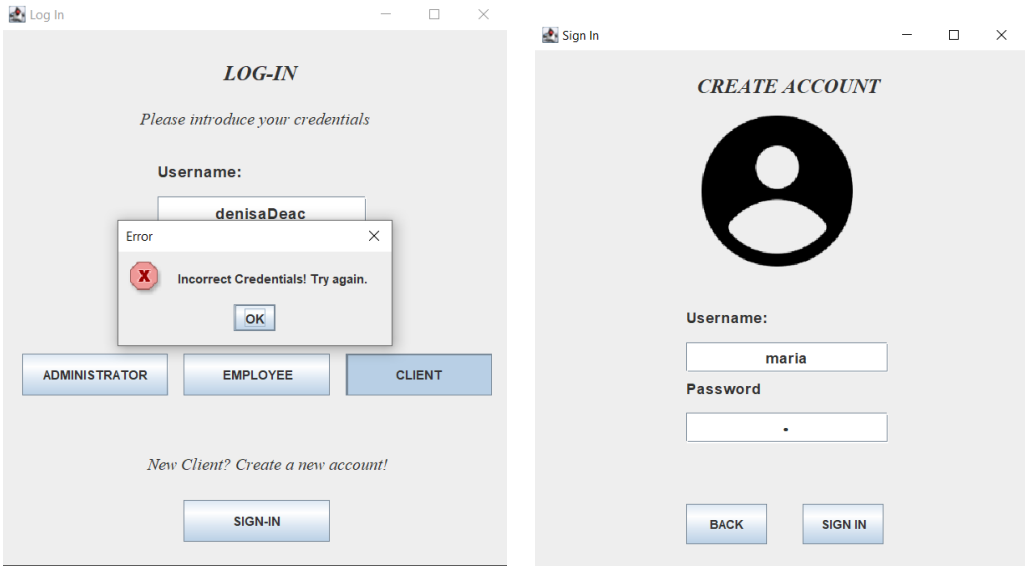
5. Results

The testing of my catering management system is done via the graphical user interface. The serialization is done by first not deserializing anything and only in the second running of the program to let the controller deserialize the data. You will see that the orders, products, and accounts done in the previous test were still on the memory. The Info.ser file in which the information from the application is stocked should look something like this:

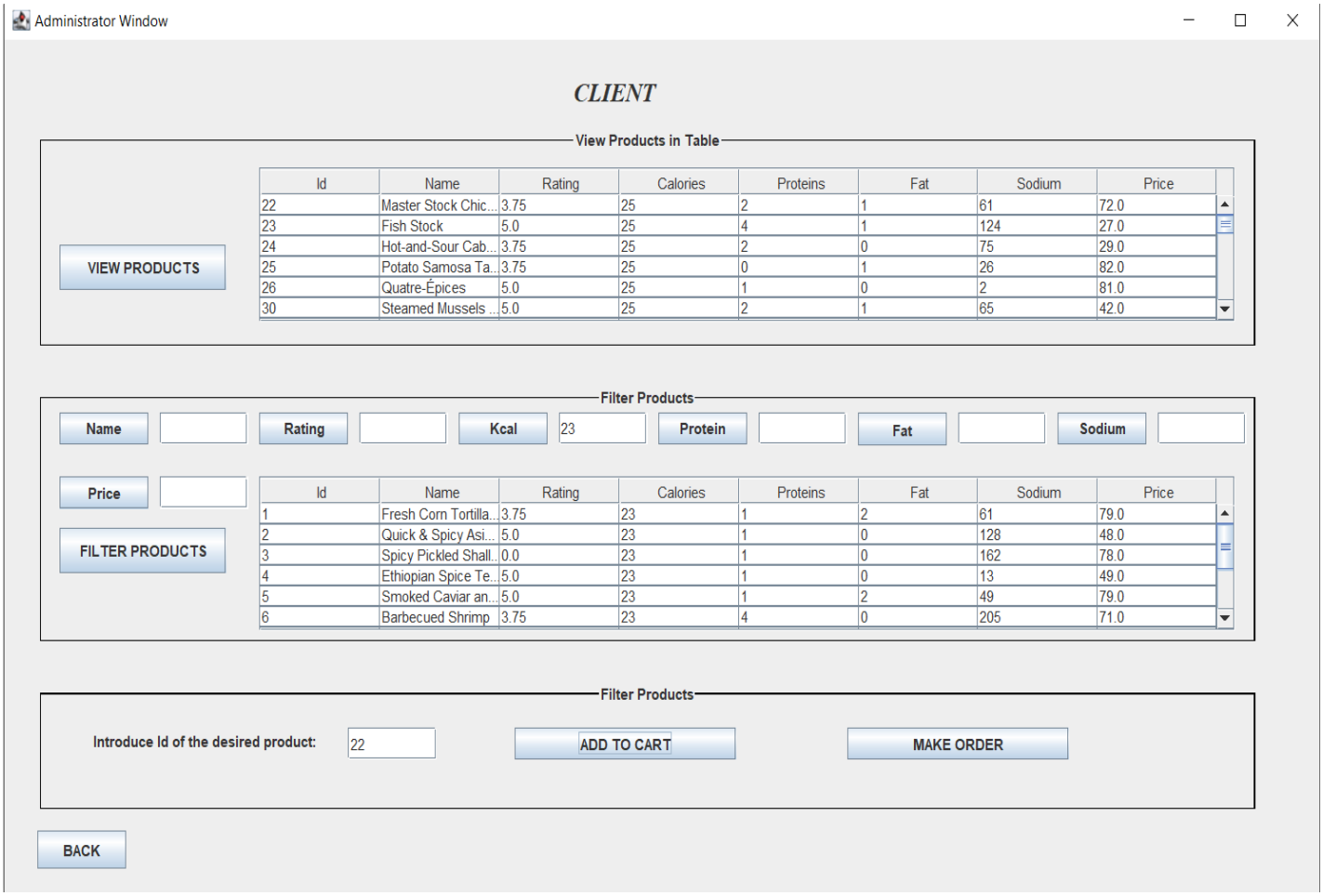


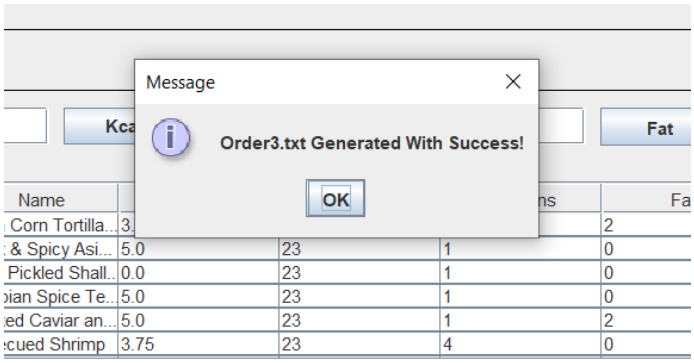
First, the account credentials must be entered in order to log in. In case the credentials are wrong or you don't have access to that category, meaning that if you try to log as administrator and you are an employee, you will get an error message on the screen to inform you(first photo). If you are a client and you don't have an account,

You have the posibility to create one, and after you done it, you will be automatically logged.

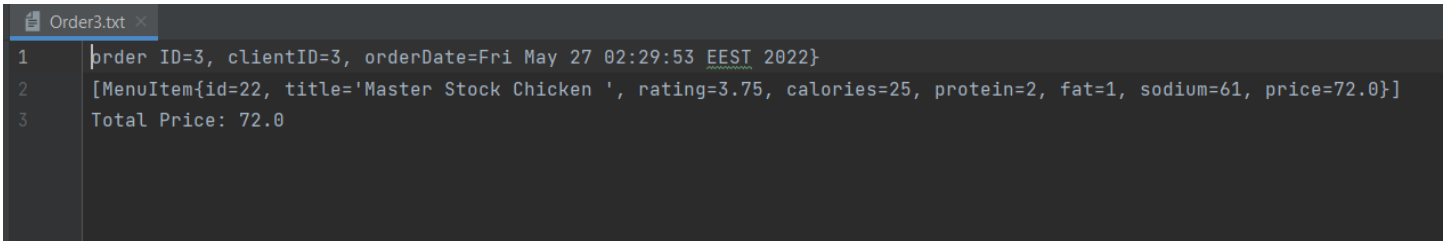


In the client window the resul of the first2 operation are displayed on the tables. For creating a new order, when pushing the “Make Order” button, a dialog message will pop on the screen.

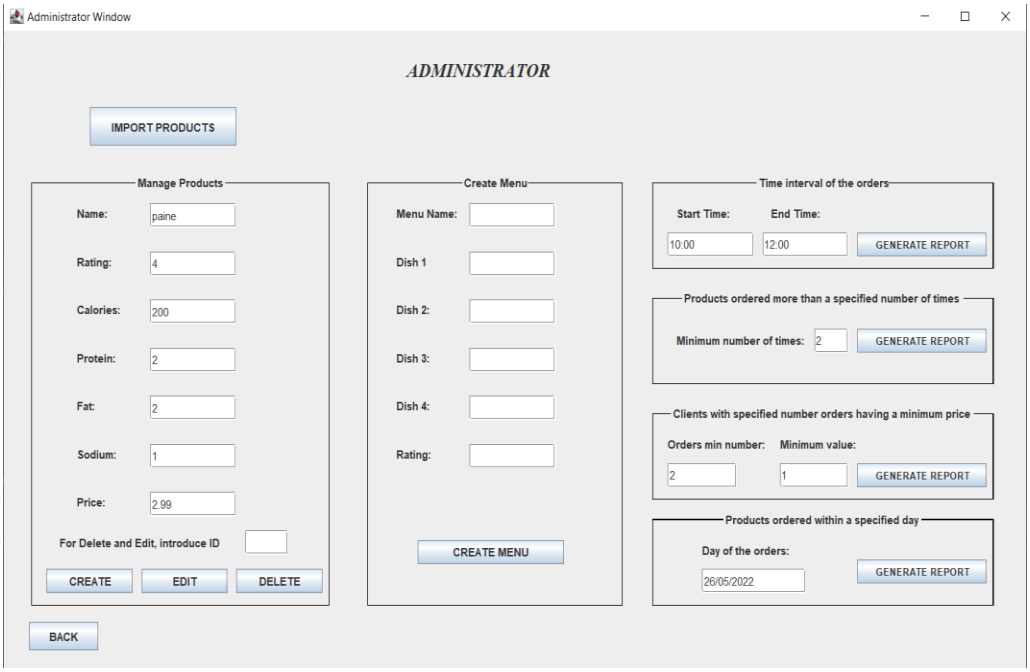




The order can be check from the generated bill.



In the administrator user interface, for creating menus and new products the corectitude of the code can be checked by locking in the client table. The reports will be generated as the order in a text file and a message will inform the user.



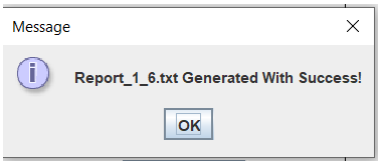
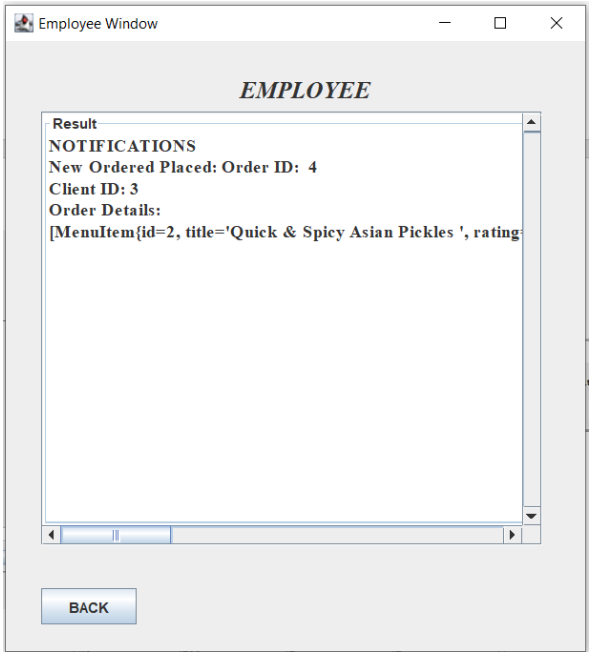


Table results:

14004	paine	4.0	200	2	2	1	2.99

The employee will get a notification in his window each time a new order has been placed.



6. Conclusion

In conclusion, this assignment was really challenging but extremely interesting. I enjoyed working with this king of application since is a good practice for a future job.

For future improvements, the id of the clients and for each report should also be serializes and used in such a way it won't restart at each running of the program.

7. Bibliography

- https://crunchify.com/how-to-remove-duplicate-elements-from-csv-or-any-other-file-in-java/
- https://stackoverflow.com/questions/8874545/java-observer-and-observable-not-working-properly-between-applications
- https://stackoverflow.com/questions/38021061/how-to-use-if-else-logic-in-java-8-stream-foreach
- https://www.baeldung.com/java-stream-filter-count
- https://stackoverflow.com/questions/49660669/parsing-csv-file-using-java-8-stream
- https://www.baeldung.com/java-stream-filter-lambda
- https://www.youtube.com/watch?v=tj5sLSFjVj4
- https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html
- https://www.javatpoint.com/java-get-current-date