Assignment 4 Documentation

Food Delivery Management System

Grama Mălina Bianca

Group 30423

Teaching Assistant: Antal Marcel

Table of Contents

1. Assignment Objectives

The objective of the food delivery management system is to process client orders for a catering company. The application will work by using databases and serialization to store the information for products available in the catering company, users and details regarding the orders.

a. Primary Objective

The primary objective of this fourth assignment is to propose, design and implement a system that is able to simulate an application for a catering company with three types of users: administrator, client and employee. The users can perform various operations:

The **administrator** can:

- Import the initial set of products which will populate the menu from a .csv file.
- Manage the products from the menu: add/delete/modify products and create new products composed of several products (an example of composed product could be named “daily menu 1” composed of a soup, a steak, a garnish, and a dessert).
- Generate reports about the performed orders considering the following criteria:
  - *time interval of the orders* – a report should be generated with the orders performed between a given start hour and a given end hour regardless the date.
  - *the products ordered more than a specified number of times so far*.
  - *the clients that have ordered more than a specified number of times and the value of the order was higher than a specified amount*.
  - *the products ordered within a specified day with the number of times they have been ordered*.

The **client** can:
- Register and use the registered username and password to log in within the system.
- View the list of products from the menu.
- Search for products based on one or multiple criteria such as keyword (e.g. “soup”), rating, number of calories/proteins/fats/sodium/price.
- Create an order consisting of several products – for each order the date and time will be persisted and a bill will be generated that will list the ordered products and the total price of the order.

The **employee** is notified each time a new order is performed by a client so that it can prepare the delivery of the ordered dishes.

b. Secondary Objectives

The secondary objectives of the assignment that I followed while implementing my solution would be the following:

- **Development of use cases and scenarios** – we need to think about how the users will interact with the application and what scenarios could arise. This will be described in detail in the **problem analysis** section of this documentation.
- **Choosing the data structures** – we need to think about how the user input will be stored and managed by our application, so that it will generate the required result. This will be briefly mentioned in the **design** section of the documentation.
- **Division of the solution into classes** – we need to think about how we will model the real-life “objects” that are the users, products and orders into an object-oriented programming object. This will be mentioned in the **design** section of this documentation and detailed in the **implementation** section.
- **Declaration of variables and methods** into aforementioned classes – after we figure out the classes that we need, we also need to think about the variables and methods that will go into these classes. Most of the times we start from the project specification, looking for nouns, which become possible

candidate classes, and verbs that could play the role of class methods. This will be detailed in the **implementation** section.
- **Development of the algorithms** necessary for the simulation – we need to think about how we will implement the functionality of the application. This objective will be detailed in the **implementation** section of the documentation.
- **Implementation of the solution** – we will need to describe more specifically how each class used in the project. Also, we will describe the implementation of the user interface. This will be detailed in the **implementation** section of this documentation.
- **Testing** – detailed in the **testing and results** section.
- **Debugging**.
- **More testing**.

2. Problem Analysis

By analyzing the problem, we refer to a first abstract set of operations and properties through which we try to detect possible features and behaviors of unknown processes. Object-oriented programming offers us a clear advantage here, precisely because it allows us to tackle the problem from a higher level, without being constrained, to such an extent, by the technical characteristics.

Below we will exemplify three use cases for the order management system:

***Use Case Name*: Food Delivery Management System – Client placing an order Action**

Primary Actor: Client User

Triggers: The client user logged in with his credentials (or registered and then logged in with the credentials).

Preconditions: The user has typed in correctly his username and password.

Normal Flow:

- The client selects which products he wants to see: the base products or the composite product
- The client searches for the desired products by browsing the tables or by searching products by a selected criteria
- The client clicks on the products that he wants on the table, and then clicks the “Select Base Product For Order” button, or the “Select Composite Product For Order” button; the products selected will appear in the text area, and a text containing the total of the order will be updated each time a new product is selected
- The client clicks on the “Place Order” button, and the Food Delivery Management System will add this order to the list of orders. A bill will be generated in a .txt file with information about the order placed
- The client can click on the “Open Bill” button to open the generated .txt file

Alternate Flows:

- The user types in an incorrect username/password combination. The application will display a message, and the user can try to log in again.
- The client adds incorrect products to the order. In this case, he can click the “Delete Products Selected” button, and all of the products selected will be discarded

***Use Case Name*: Food Delivery Management System – Administrator Action**

Primary Actor: Administrator

Triggers: The administrator user logged in with his credentials.

Preconditions: The user has typed in correctly his username and password in the login window.

Normal Flow:

- The administrator can see a table with the base products available for order in the catering company. He can import these products from a CSV file by clicking on the "Import Products" button.
- If the admin clicks on a product from the table, all the available data will be completed in the text fields, so that it will be easier to modify an existing product. If the user whishes to empty these text fields fast, he can click on the "Clear Fields" button.
- The administrator can choose one of the three available operations for modifying the base products table: add a product, modify a product or delete a product. For the modification and deletion of a product, the user has to click on the desired product in the table, before clicking on the action button.
- The administrator can choose to see the composite product available for purchase by clicking on the "View Composite Products" button
- The administrator can choose to add a composite product by clicking on the respective button. A new window will open up, showing a table with the base products. The admin can then select the wanted items and add them to the composite product, type a name for the said product, and create the composite product.
- The admin can also generate various reports by clicking the "Generate Reports" button. A new window will appear, where the user can type in the required information and generate said reports.

Alternate Flows:

- The user types in an incorrect username/password combination. The application will display a message, and the user can try to log in again.

***Use Case Name*: Food Delivery Management System – Orders Action**

Primary Actor: Employee

Triggers: The employee user logged in with his credentials.

Normal Flow:

- The employee presses the "See Orders" button, and the most recent orders will be displayed in the text area.

Alternate Flows:

- The user types in an incorrect username/password combination. The application will display a message, and the user can try to log in again.

We need to define a set of functional and non-functional requirements, to ensure that we cover all of the cases and provide the functionality of the application.

Some of the functional requirements could be:

- The food delivery system should perform correctly each of the selected operations for all the types of users.

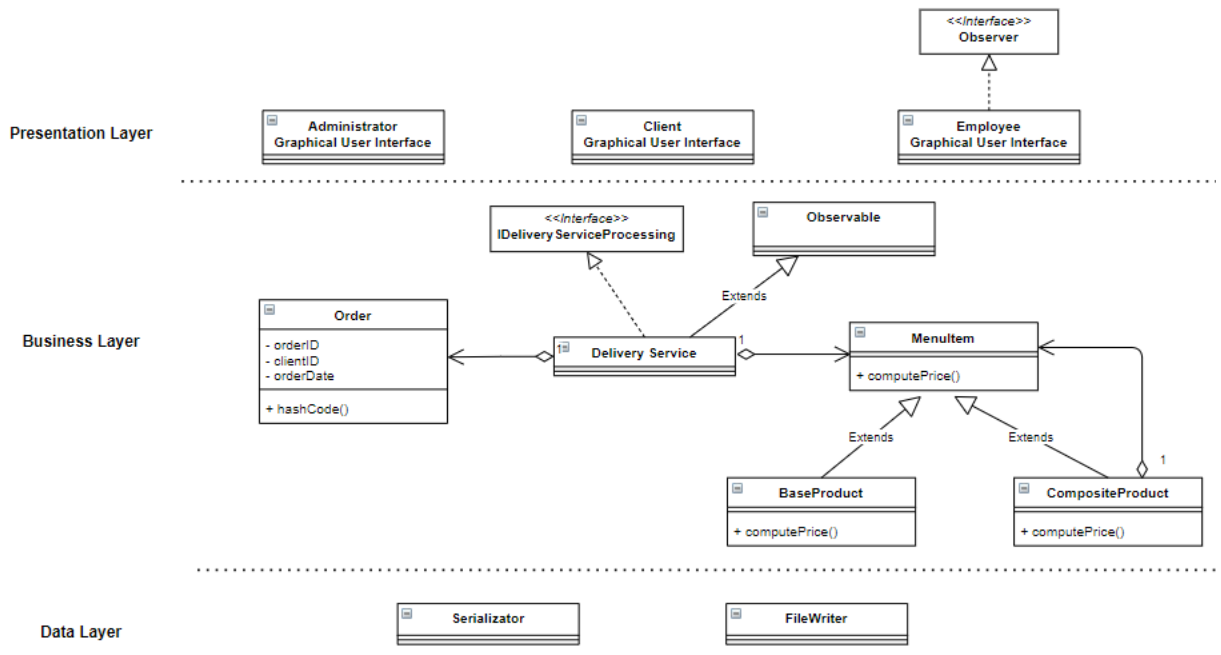
Some of the non-functional requirements of the application:

- The food delivery management system should be easy to use and intuitive.
- The results should be visible for the user after each selected operation.

3. Design

We try to think of the solution in a bottom-up manner, dividing the components of the problem into smaller pieces. As we advance to lower levels, the complexity of the problem increases.

In the project specification we have a skeleton of an UML, meant to help us with the implementation of our application.



Working with a database requires a class that will handle the connection. This class is called **DatabaseConnection** and is placed in the "**BusinessLayer**" package.

Because we are working with real-life objects, we will need several classes for these objects. One of these classes is the **MenuItem** abstract class, that represents a product from the menu. Two more classes will extend this one, based on whether the product from the menu is a Base one or a Composite one: **BaseProduct** and **CompositeProduct**, that both extend the MenuItem class. Another class that mirrors a real-life object is the **Order** class, that holds information about an order placed by a client. I added another class that will help me with the employee operations, namely the **Employee** class.

Other two classes placed in the "**BusinessLayer**" package are the **IDeliveryServiceProcessing** interface and the **DeliveryService** class, that represents the "heart" of the application and that holds all the necessary information needed for running the application. Because of this, to have the updated information available every time we run the simulation, we will serialize and deserialize the DeliveryService class with the help of the **Serializator** class, placed in the "**DataLayer**" package. To get the products from the CSV file provided, we also use a class named **FileWriter**, placed in the same package as the Serializator class.

We also need classes that will control the GUI windows, and an App class that will contain the main method that will launch the application. These classes will be placed in the "**PresentationLayer**" package.

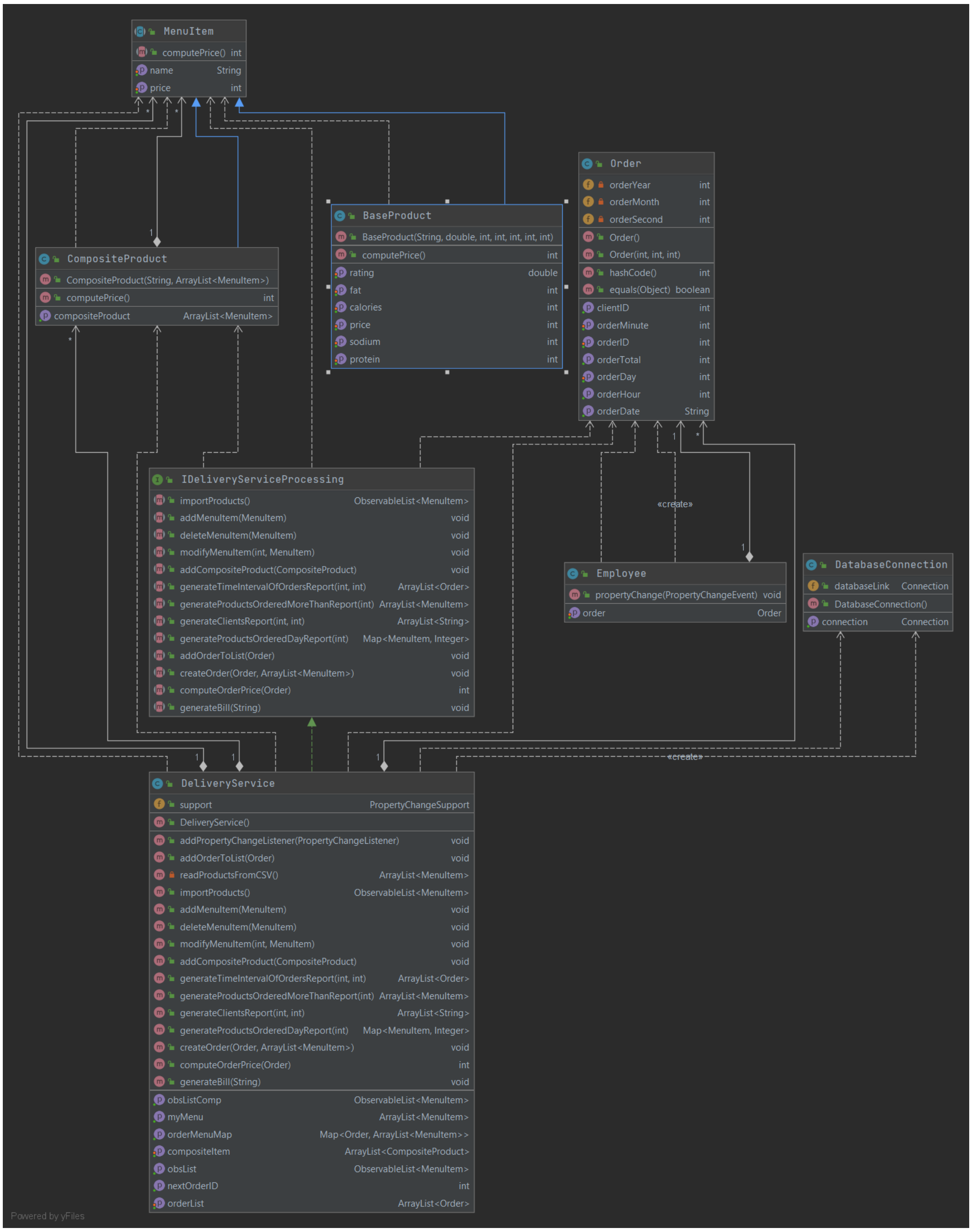
BusinessLayer	**DataLayer**	**PresentationLayer**
BaseProduct.java	FileWriter.java	AdminController,java
CompositeProduct.java	Serializator.java	App.java
DatabaseConnection.java		ClientController.java
DeliveryService.java		CompositeProductCreateController.java
Employee.java		CompositeProductViewController.java
IDeliveryServiceProcessing.java		EmployeeController.java
MenuItem.java		GenerateReportsController.java
Order,java		LoginController.java
		RegisterController.java

4. Implementation

a. Back-End

Now that we have decided what classes are needed for our project, let us talk briefly about the main methods of these Java Classes.

We will start with the classes in the **BusinessLayer** package.

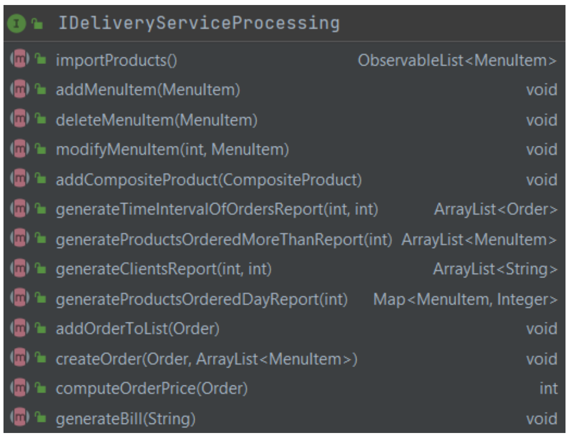

MenuItem
computePrice() int
name String
price int
CompositeProduct
CompositeProduct(String, ArrayList<MenuItem>)
computePrice() int
compositeProduct ArrayList<MenuItem>
BaseProduct
BaseProduct(String, double, int, int, int, int, int)
computePrice() int
rating double
fat int
calories int
price int
sodium int
protein int
Order
orderYear int
orderMonth int
orderSecond int
Order()
Order(int, int, int)
hashCode() int
equals(Object) boolean
clientID int
orderMinute int
orderID int
orderTotal int
orderDay int
orderHour int
orderDate String
«create»
IDeliveryServiceProcessing
importProducts() ObservableList<MenuItem>
addMenuItem(MenuItem) void
deleteMenuItem(MenuItem) void
modifyMenuItem(int, MenuItem) void
addCompositeProduct(CompositeProduct) void
generateTimeIntervalOfOrdersReport(int, int) ArrayList<Order>
generateProductsOrderedMoreThanReport(int) ArrayList<MenuItem>
generateClientsReport(int, int) ArrayList<String>
generateProductsOrderedDayReport(int) Map<MenuItem, Integer>
addOrderToList(Order) void
createOrder(Order, ArrayList<MenuItem>) void
computeOrderPrice(Order) int
generateBill(String) void
Employee
propertyChange(PropertyChangeEvent) void
order Order
DatabaseConnection
databaseLink Connection
DatabaseConnection()
connection Connection
«create»
DeliveryService
support PropertyChangeSupport
DeliveryService()
addPropertyChangeListener(PropertyChangeListener) void
addOrderToList(Order) void
readProductsFromCSV() ArrayList<MenuItem>
importProducts() ObservableList<MenuItem>
addMenuItem(MenuItem) void
deleteMenuItem(MenuItem) void
modifyMenuItem(int, MenuItem) void
addCompositeProduct(CompositeProduct) void
generateTimeIntervalOfOrdersReport(int, int) ArrayList<Order>
generateProductsOrderedMoreThanReport(int) ArrayList<MenuItem>
generateClientsReport(int, int) ArrayList<String>
generateProductsOrderedDayReport(int) Map<MenuItem, Integer>
createOrder(Order, ArrayList<MenuItem>) void
computeOrderPrice(Order) int
generateBill(String) void
obsListComp ObservableList<MenuItem>
myMenu ArrayList<MenuItem>
orderMenuMap Map<Order, ArrayList<MenuItem>>
compositeItem ArrayList<CompositeProduct>
obsList ObservableList<MenuItem>
nextOrderID int
orderList ArrayList<Order>
Powered by yFiles

The classes **BaseProduct** and **CompositeProduct** are simple classes containing variables representing the fields of a base/composite product, a constructor, getters and setters. They both extend the **MenuItem** abstract class, that contains a String representing the name of the product, an int representing the price, and a method int *computePrice()* that calculates the price of a composite product.

The class **Order** contains variables that represent information regarding a placed order, like int orderID, int clientID, String orderDate, int orderTotal etc. The constructor of this class has as parameters the id of the order (that is computed as the next available index in the list of the orders that is placed in the DeliveryService class), the id of the client that placed the order, and the total value of the products ordered, and the rest of the variables are computed inside the constructor. The class has the methods *int hashCode()* and *boolean equals(Object)* overridden.

The class **Employee** is responsible with notifying the employee when a new order is placed. This is possible with the help of the PropertyChangeListener interface.

The interface **IDeliveryServiceProcessing** and the class **DeliveryService** are the most important classes in the application. In the interface we have defined a few methods with the actions that can be performed by the administrator and the client, namely:



IDeliveryServiceProcessing

importProducts()	ObservableList<MenuItem>
addMenuItem(MenuItem)	void
deleteMenuItem(MenuItem)	void
modifyMenuItem(int, MenuItem)	void
addCompositeProduct(CompositeProduct)	void
generateTimeIntervalOfOrdersReport(int, int)	ArrayList<Order>
generateProductsOrderedMoreThanReport(int)	ArrayList<MenuItem>
generateClientsReport(int, int)	ArrayList<String>
generateProductsOrderedDayReport(int)	Map<MenuItem, Integer>
addOrderToList(Order)	void
createOrder(Order, ArrayList<MenuItem>)	void
computeOrderPrice(Order)	int
generateBill(String)	void

The DeliveryService class has the following variables:

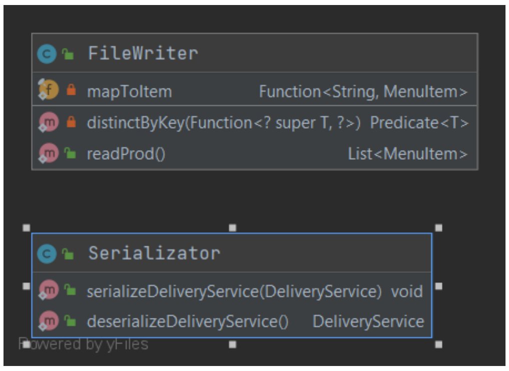
- ArrayList<MenuItem> myMenu – holding the base products that form the menu of the catering company
- ArrayList<CompositeProduct> compositeItem - holding the composite products that form the menu of the catering company
- HashMap<Order, ArrayList<MenuItem>> orderMenuMap – linking the orders with the products that have been ordered
- ArrayList<Order> orderList – a list with the orders that have been placed

The methods of the class described shortly are the following:

- *int getNextOrderID()* – returns the next available index of the orderList, used for an id for the next order
- *void addOrderToList(Order)* – adds an order transmitted as a parameter to the orderList
- *ArrayList<MenuItem> readProductsFromCSV()* – calls the readProd() method from the FileWriter class and returns an array of products from the CSV file
- *ObservableList<MenuItem> importProducts()* – imports products from the CSV file (by calling the upper method) and places them in the myMenu list
- *ObservableList<MenuItem> getObsList()* – returns an observable list made up of the objects in the myMenu list, used for displaying the data from the menu in the tables of the Graphical User Interface
- *ObservableList<MenuItem> getObsListComp()* - returns an observable list made up of the objects in the compositeItem list, used for displaying the data from the menu in the tables of the Graphical User Interface
- *void addMenuItem(MenuItem)* – adds the MenuItem in the myMenu list
- *void deleteMenuItem(MenuItem)* – looks for and deletes the MenuItem from the myMenu list
- *void modifyMenuItem(int, MenuItem)* – modifies the MenuItem from the myMenu list located at index transmitted as a parameter with the MenuItem transmitted
- *void addCompositeProduct(CompositeProduct)* – adds the CompositeProduct to the compositeItem list
- *ArrayList<Order> generateTimeIntervalOfOrdersReport(int, int)* – traverses the list orderList and adds the orders that have the orderHour in the specified range to another ArrayList, and returns it

- *ArrayList<MenuItem> generateProductsOrderedMoreThanReport(int)* – the method works like this: first we get all of the products that have been ordered from the orderMenuMap and we add them to an ArrayList; then we declare an HashMap and we count the occurrences of all the items in the ArrayList generated above; we traverse this HashMap, and if the number of occurrences is bigger than the integer transmitted as a parameter, we add the product in the ArrayList that will be returned
- *ArrayList<String> generateClientsReport(int, int)* – the method returns an ArrayList wih the usernames of the clients that have ordered more than a specified amount of times, and the value of the order was bigger than a specified value transmitted as a parameter; the algorithm works like this: first we traverse the orderList and we form an ArrayList with the ids of those clients that have the value of the order bigger than the specified value; then, we count the occurrences of these ids with the help of a HashMap; we traverse this HashMap and we remove the entries that have the number of occurrences less than the specified number of orders; then, for every id left, we execute a query on the database to get the user's username based on its id, and we add that username to the ArrayList of Strings that will be returned
- *Map<MenuItem, Integer> generateProductsOrderedDayReport(int)* – the method returns a HashMap containing all of the products ordered in a specified day, with the number of times they have been ordered; the method works like this: we traverse the orderMenuMap and we add the products ordered in the specified day in an ArrayList; then, with the help of a HashMap, we count the occurrences of these products, and we return the HashMap
- *void createOrder(Order, ArrayList<MenuItem>)* – adds the order and the ArrayList of products ordered in the orderMenuMap
- *int computeOrderPrice(Order)* – we get the products ordered from the orderMenuMap, we traverse them and we compute thee total price of the order
- *void generateBill(String)* – the method writes the String transmitted in a .txt file

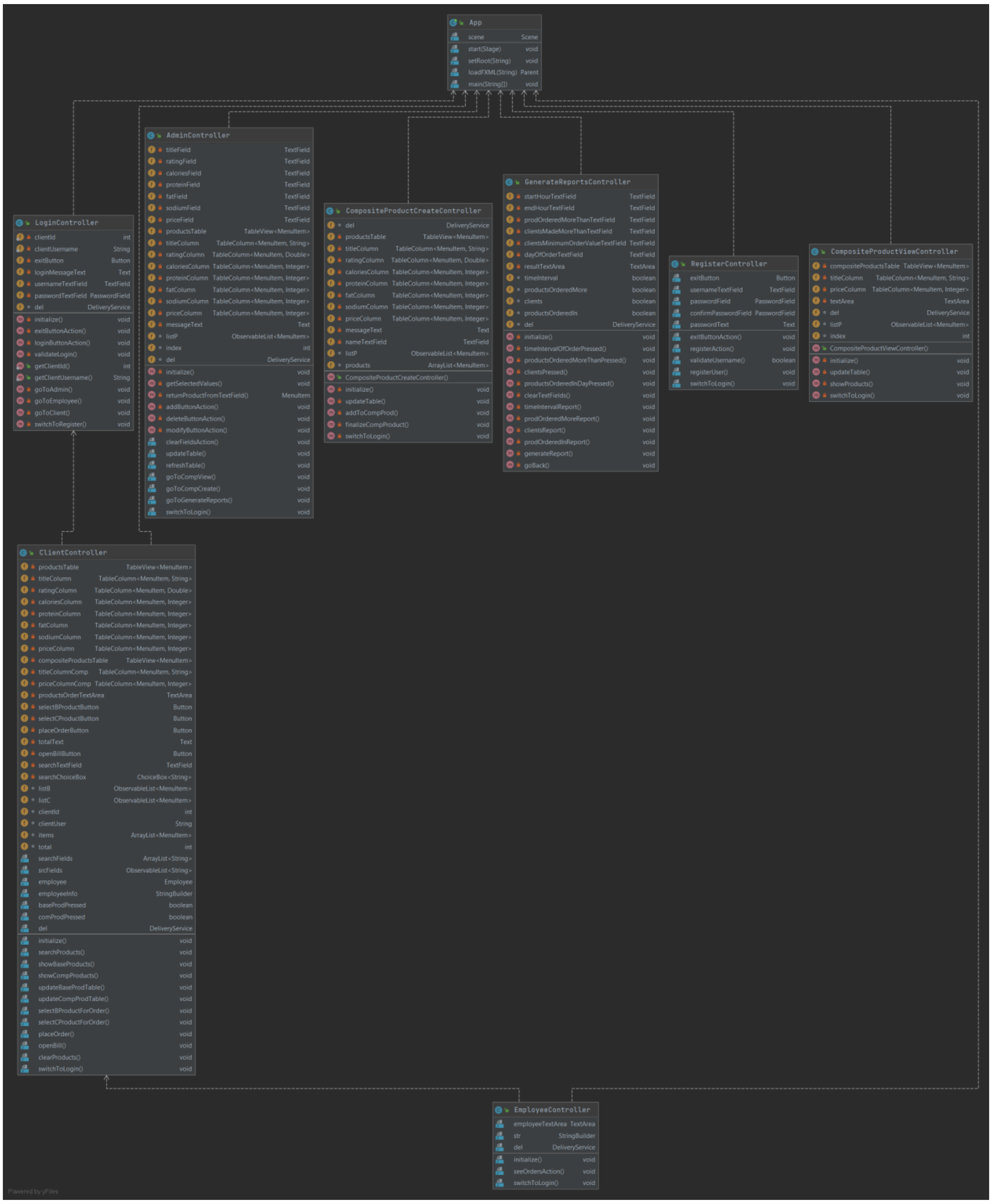
The classes in the **DataLayer** package:


FileWriter
mapToItem Function<String, MenuItem>
distinctByKey(Function<? super T, ?>) Predicate<T>
readProd() List<MenuItem>
Serializator
serializeDeliveryService(DeliveryService) void
deserializeDeliveryService() DeliveryService
Powered by yFiles

The class **FileWriter** uses streams to read from the given .csv file the base products that will be imported in the menu. It filters the products that are distinct by key, it forms BaseProduct objects with the information from a line, and it returns an ArrayList of BaseProducts.

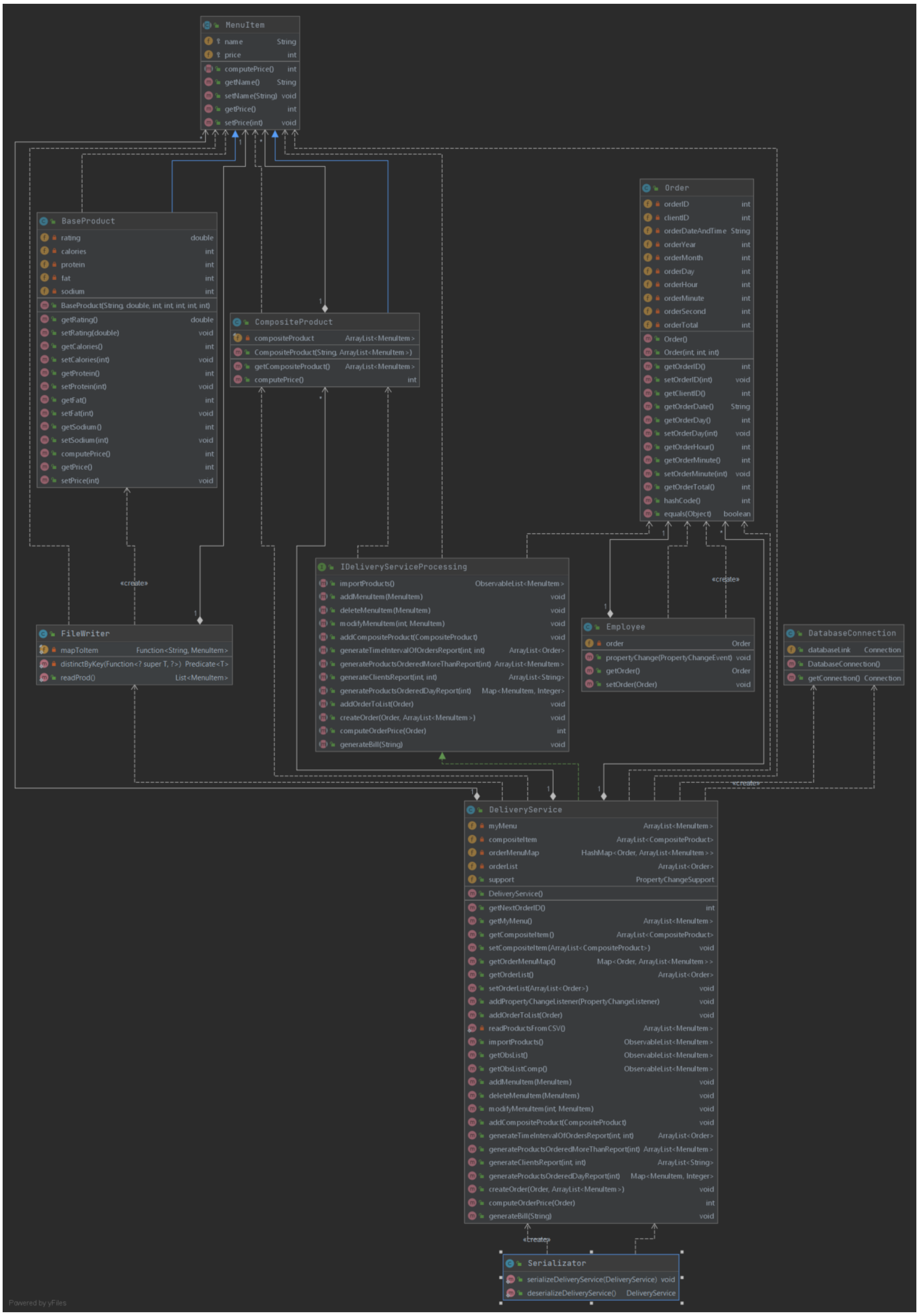
The class **Serializator** serializes and deserializes the class DeliveryService in a .ser file. The serialization/deserialization is done every time a new window is closed/open, so that the data written in the DeliveryService class is not lost, even if the application is closed and run again later.

The classes in the **PresentationLayer**:



Every class controls one window of the application.

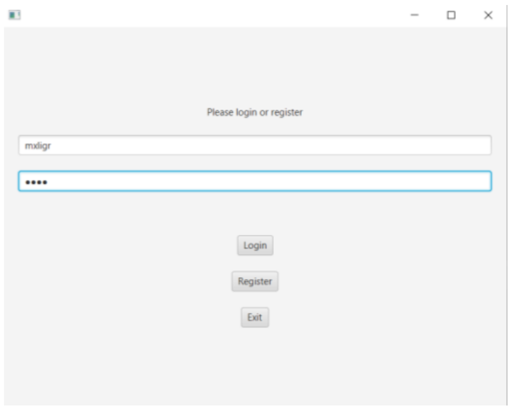
The UML diagram for the back-end of the application:


MenuItem
name String
price int
computePrice() int
getName() String
setName(String) void
getPrice() int
setPrice(int) void
BaseProduct
rating double
calories int
protein int
fat int
sodium int
BaseProduct(String, double, int, int, int, int)
getRating() double
setRating(double) void
getCalories() int
setCalories(int) void
getProtein() int
setProtein(int) void
getFat() int
setFat(int) void
getSodium() int
setSodium(int) void
computePrice() int
getPrice() int
setPrice(int) void
CompositeProduct
compositeProduct ArrayList<MenuItem>
CompositeProduct(String, ArrayList<MenuItem>)
getCompositeProduct() ArrayList<MenuItem>
computePrice() int
Order
orderID int
clientID int
orderDateAndTime String
orderYear int
orderMonth int
orderDay int
orderHour int
orderMinute int
orderSecond int
orderTotal int
Order()
Order(int, int, int)
getOrderID() int
setOrderID(int) void
getClientID() int
getOrderDate() String
getOrderDay() int
setOrderDay(int) void
getOrderHour() int
getOrderMinute() int
setOrderMinute(int) void
getOrderTotal() int
hashCode() int
equals(Object) boolean
IDeliveryServiceProcessing
importProducts() ObservableList<MenuItem>
addMenuItem(MenuItem) void
deleteMenuItem(MenuItem) void
modifyMenuItem(int, MenuItem) void
addCompositeProduct(CompositeProduct) void
generateTimeIntervalOfOrdersReport(int, int) ArrayList<Order>
generateProductsOrderedMoreThanReport(int) ArrayList<MenuItem>
generateClientsReport(int, int) ArrayList<String>
generateProductsOrderedDayReport(int) Map<MenuItem, Integer>
addOrderToList(Order) void
createOrder(Order, ArrayList<MenuItem>) void
computeOrderPrice(Order) int
generateBill(String) void
FileWriter
mapToItem Function<String, MenuItem>
distinctByKey(Function<? super T, ?>) Predicate<T>
readProd() List<MenuItem>
«create»
Employee
order Order
propertyChange(PropertyChangeEvent) void
getOrder() Order
setOrder(Order) void
«create»
DatabaseConnection
databaseLink Connection
DatabaseConnection()
getConnection() Connection
«create»
DeliveryService
myMenu ArrayList<MenuItem>
compositeItem ArrayList<CompositeProduct>
orderMenuMap HashMap<Order, ArrayList<MenuItem>>
orderList ArrayList<Order>
support PropertyChangeSupport
DeliveryService()
getNextOrderID() int
getMyMenu() ArrayList<MenuItem>
getCompositeItem() ArrayList<CompositeProduct>
setCompositeItem(ArrayList<CompositeProduct>) void
getOrderMenuMap() Map<Order, ArrayList<MenuItem>>
getOrderList() ArrayList<Order>
setOrderList(ArrayList<Order>) void
addPropertyChangeListener(PropertyChangeListener) void
addOrderToList(Order) void
readProductsFromCSV() ArrayList<MenuItem>
importProducts() ObservableList<MenuItem>
getObsList() ObservableList<MenuItem>
getObsListComp() ObservableList<MenuItem>
addMenuItem(MenuItem) void
deleteMenuItem(MenuItem) void
modifyMenuItem(int, MenuItem) void
addCompositeProduct(CompositeProduct) void
generateTimeIntervalOfOrdersReport(int, int) ArrayList<Order>
generateProductsOrderedMoreThanReport(int) ArrayList<MenuItem>
generateClientsReport(int, int) ArrayList<String>
generateProductsOrderedDayReport(int) Map<MenuItem, Integer>
createOrder(Order, ArrayList<MenuItem>) void
computeOrderPrice(Order) int
generateBill(String) void
«create»
Serializator
serializeDeliveryService(DeliveryService) void
deserializeDeliveryService() DeliveryService
Powered by yFiles

b. Front-End

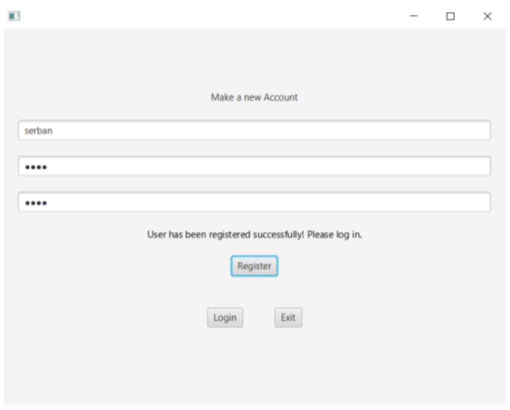
I implemented the Graphical User Interface using JavaFX and SceneBuilder.

When the user launches the application, he is met with the login window that looks like this:



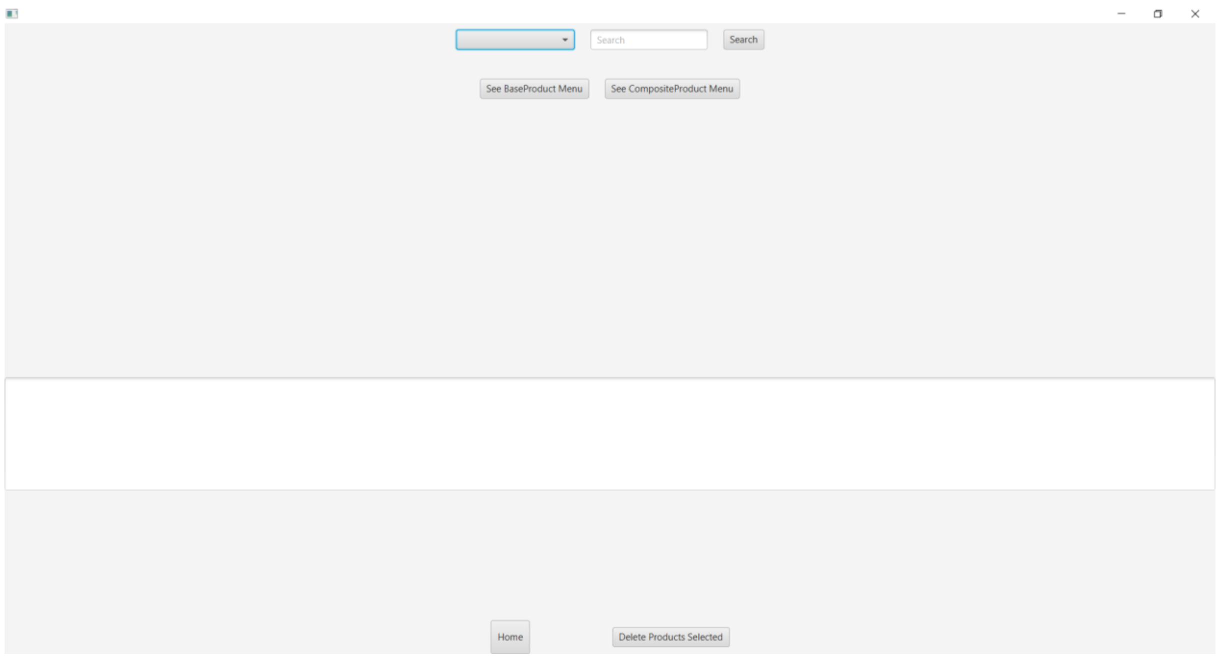
After the user enters the username and password, a new window will open. The window that will open depends on the type of user that is linked with the username and password typed in.

If a client user wishes to make a new account, he can do so by pressing on the register button. A new window will appear, prompting the user to enter a username, a password, and to confirm the password by typing it again. The application then checks if the username is not yet taken and if the password matches the confirm password field, and if all is well, the user is registered. A message will pop up, telling the user that he needs to log in with the newly made account in order to place an order.

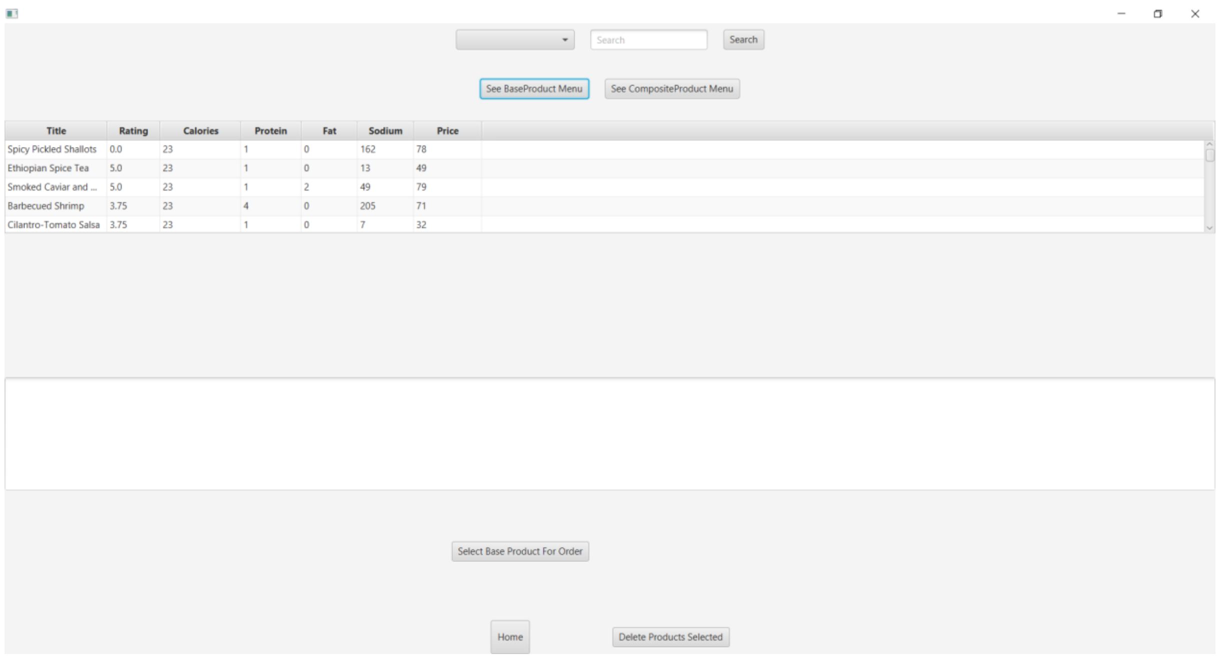


User has been registered successfully! Please log in.

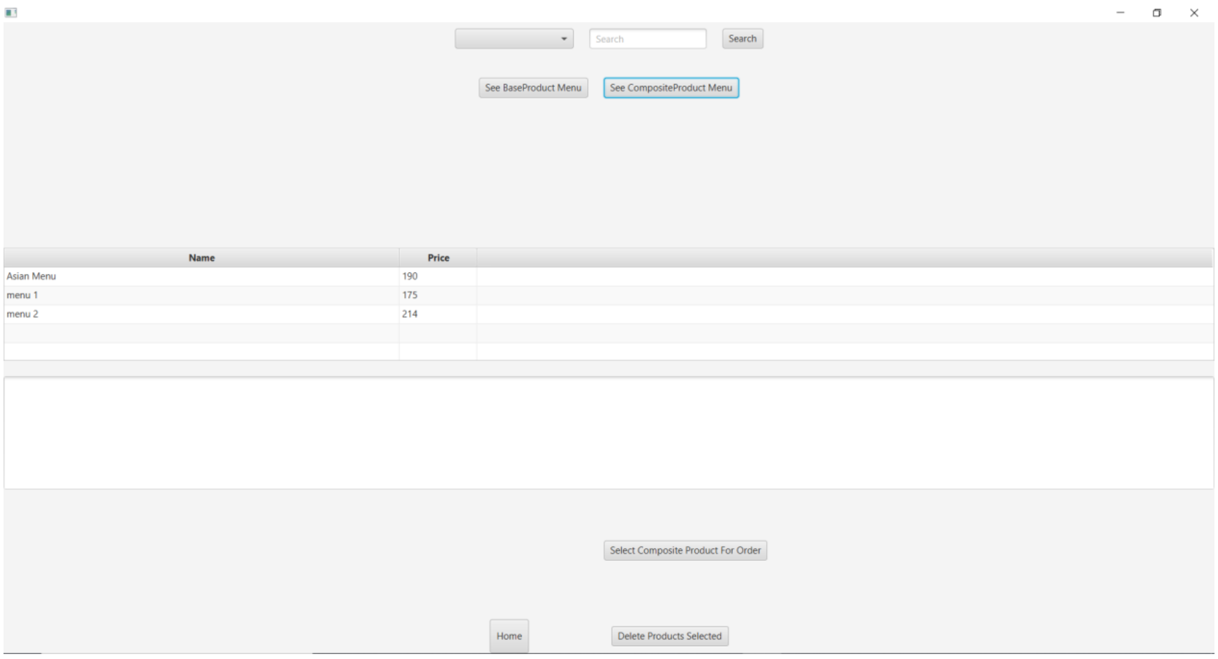
After a client logs in with the correct credentials, he is met with the client window, that looks like this:



If the user presses the "See BaseProduct Menu" button situated above, a table with the base products will appear. Respectively, if the "See CompositeProduct Menu" button is pressed, a table with the composite products in the menu will appear.

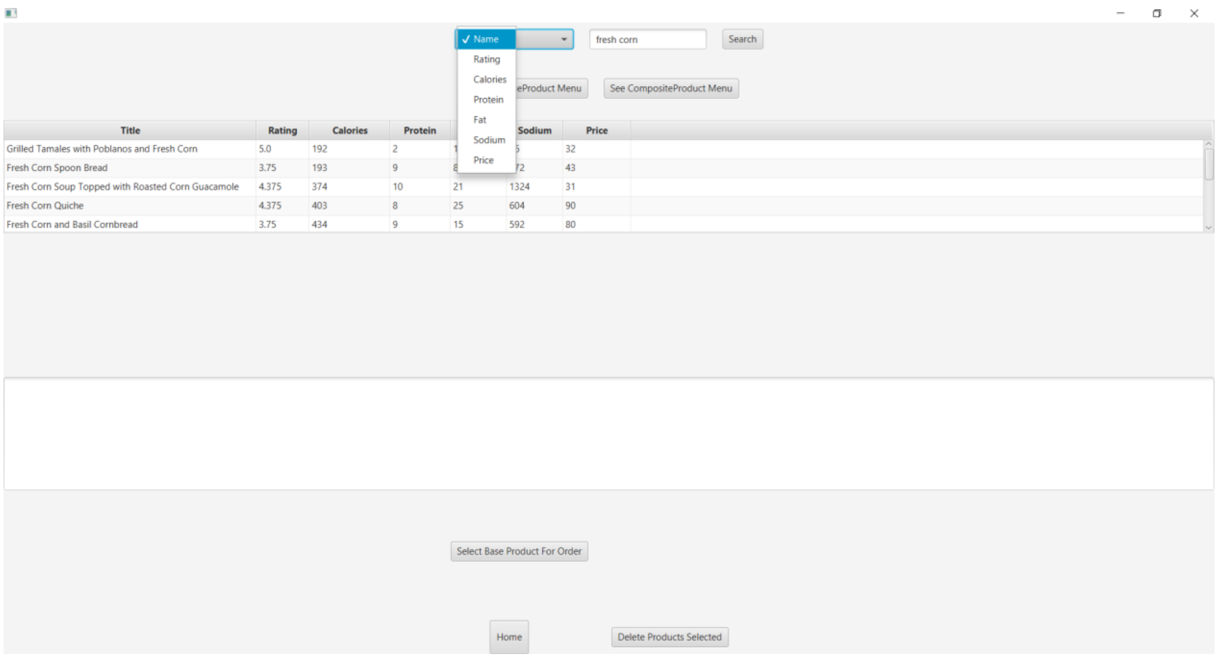


Title	Rating	Calories	Protein	Fat	Sodium	Price
Spicy Pickled Shallots	0.0	23	1	0	162	78
Ethiopian Spice Tea	5.0	23	1	0	13	49
Smoked Caviar and ...	5.0	23	1	2	49	79
Barbecued Shrimp	3.75	23	4	0	205	71
Cilantro-Tomato Salsa	3.75	23	1	0	7	32



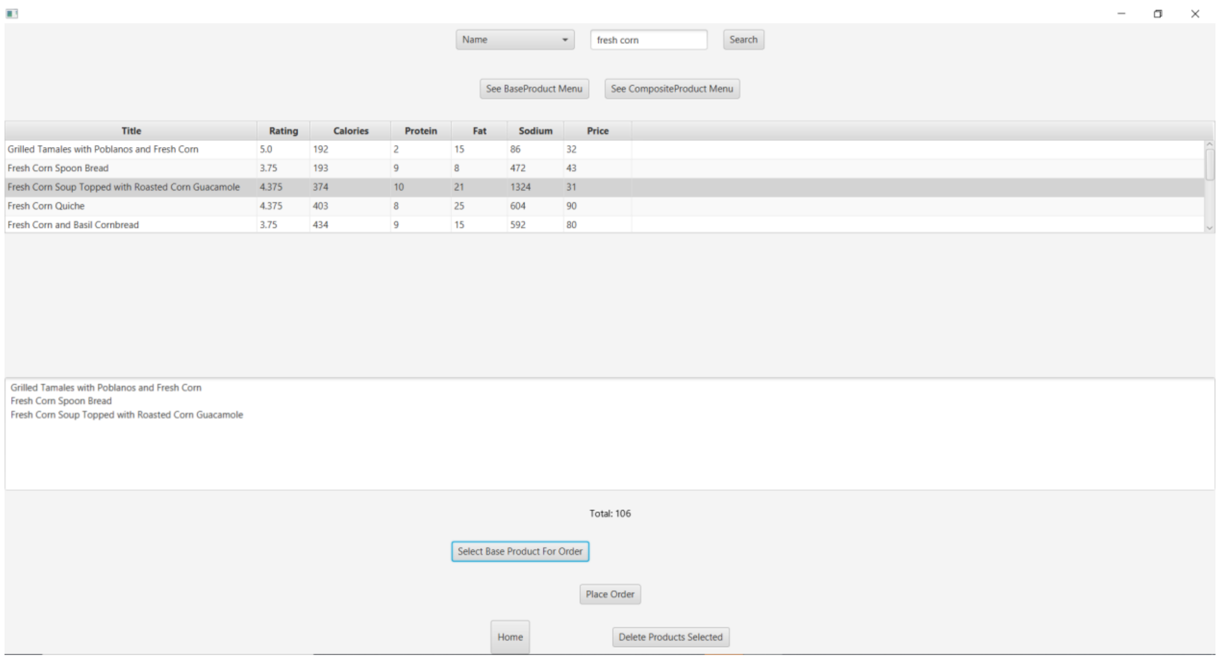
Name	Price
Asian Menu	190
menu 1	175
menu 2	214

The user can then search for products in the selected table based on criterions such as name, rating, calories etc. The criterion will be selected from the drop down, the required criterion typed in the search text bar, and the "Search" button will be pressed. The table will then be updated.



Title	Rating	Calories	Protein		Sodium	Price
Grilled Tamales with Poblanos and Fresh Corn	5.0	192	2	1	5	32
Fresh Corn Spoon Bread	3.75	193	9	8	72	43
Fresh Corn Soup Topped with Roasted Corn Guacamole	4.375	374	10	21	1324	31
Fresh Corn Quiche	4.375	403	8	25	604	90
Fresh Corn and Basil Cornbread	3.75	434	9	15	592	80

After browsing the tables, the user can select products he wishes to order by clicking on them in the table, and then pressing on the "Select Base Product For Order"/"Select Composite Product For Order" button. The selected item's name will appear in the text area below, and a text with the total sum of the order will appear.

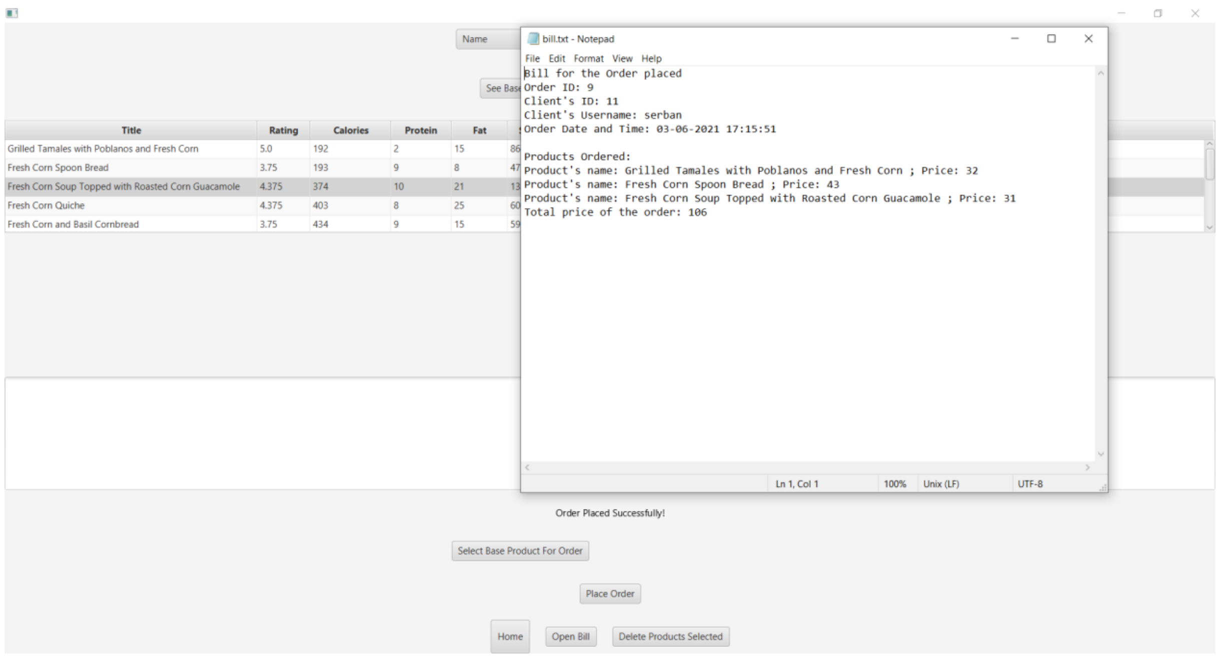


Title	Rating	Calories	Protein	Fat	Sodium	Price
Grilled Tamales with Poblanos and Fresh Corn	5.0	192	2	15	86	32
Fresh Corn Spoon Bread	3.75	193	9	8	472	43
Fresh Corn Soup Topped with Roasted Corn Guacamole	4.375	374	10	21	1324	31
Fresh Corn Quiche	4.375	403	8	25	604	90
Fresh Corn and Basil Cornbread	3.75	434	9	15	592	80

```
Grilled Tamales with Poblanos and Fresh Corn
Fresh Corn Spoon Bread
Fresh Corn Soup Topped with Roasted Corn Guacamole
```

Total: 106

If the client is satisfied with the products picked, he can press the “Place Order” button, and the bill will be generated.



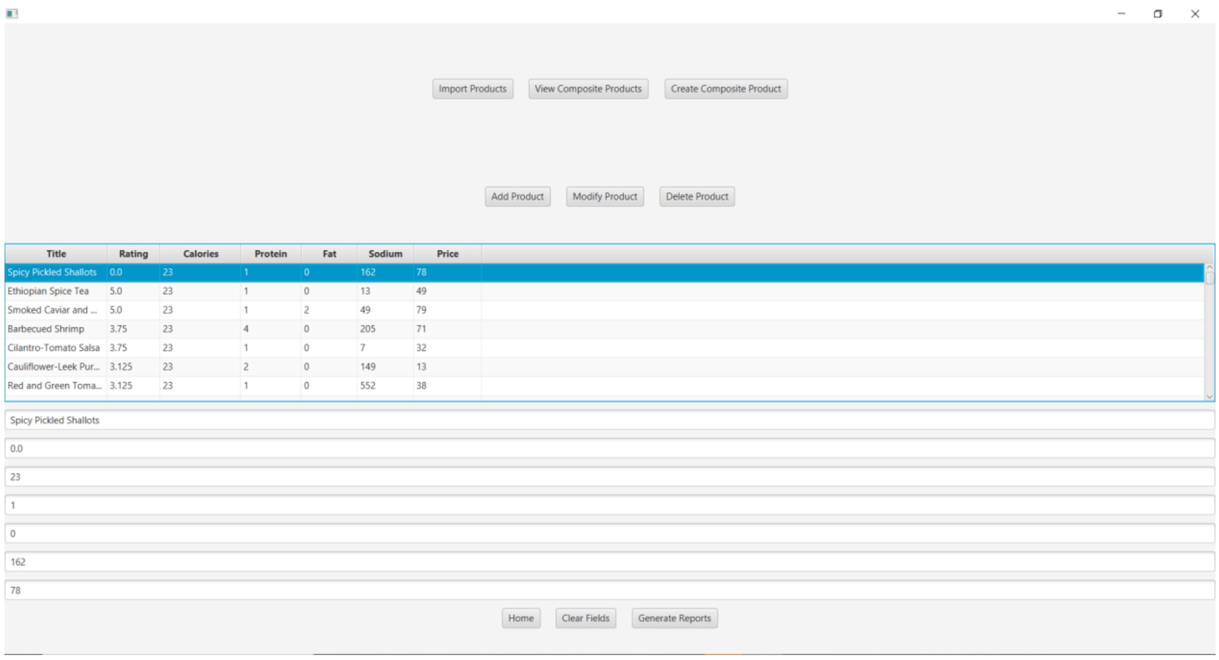
Title	Rating	Calories	Protein	Fat
Grilled Tamales with Poblanos and Fresh Corn	5.0	192	2	15
Fresh Corn Spoon Bread	3.75	193	9	8
Fresh Corn Soup Topped with Roasted Corn Guacamole	4.375	374	10	21
Fresh Corn Quiche	4.375	403	8	25
Fresh Corn and Basil Cornbread	3.75	434	9	15

```
Bill for the Order placed
Order ID: 9
Client's ID: 11
Client's Username: serban
Order Date and Time: 03-06-2021 17:15:51

Products Ordered:
Product's name: Grilled Tamales with Poblanos and Fresh Corn ; Price: 32
Product's name: Fresh Corn Spoon Bread ; Price: 43
Product's name: Fresh Corn Soup Topped with Roasted Corn Guacamole ; Price: 31
Total price of the order: 106
```

Order Placed Successfully!

If the client wishes to start again with the selection of items, he can press the “Delete Products Selected” button, and the items already selected will be deleted.

Logging in with the credentials linked to the administrator will open the administrator window.



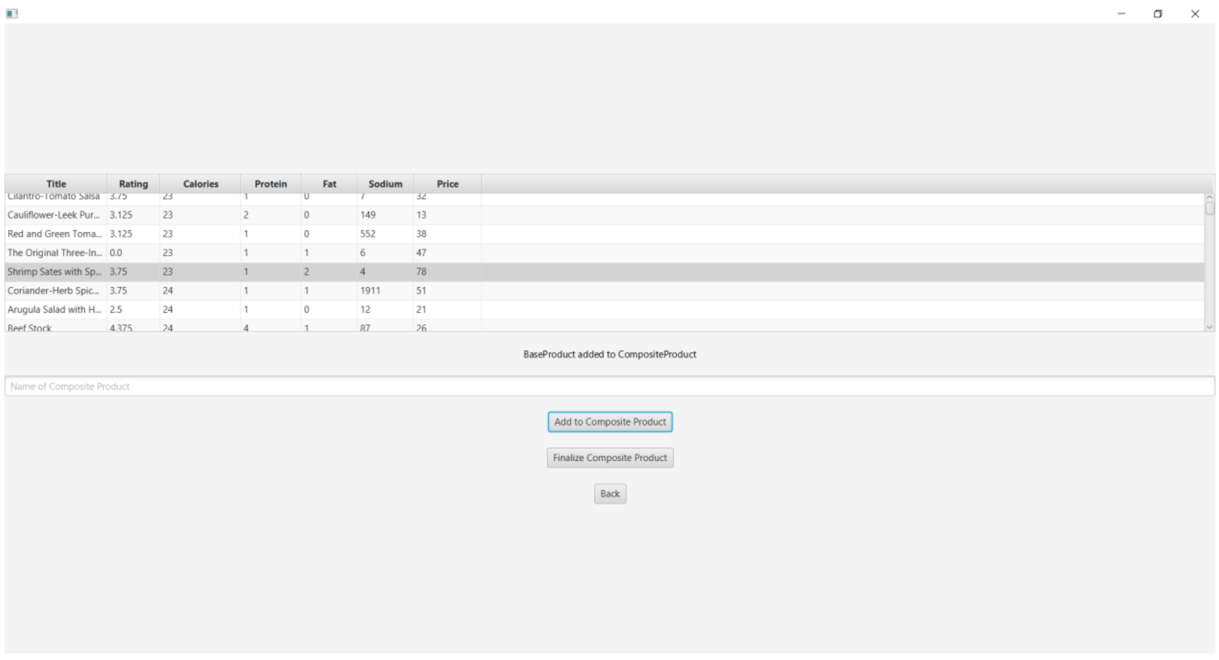
Title	Rating	Calories	Protein	Fat	Sodium	Price
Spicy Pickled Shallots	0.0	23	1	0	162	78
Ethiopian Spice Tea	5.0	23	1	0	13	49
Smoked Caviar and ...	5.0	23	1	2	49	79
Barbecued Shrimp	3.75	23	4	0	205	71
Cilantro-Tomato Salsa	3.75	23	1	0	7	32
Cauliflower-Leek Pur...	3.125	23	2	0	149	13
Red and Green Toma...	3.125	23	1	0	552	38

Spicy Pickled Shallots
0.0
23
1
0
162
78

The administrator can add/edit/delete products by selecting them in the table and filling up the necessary information in the text fields below. The text fields fill up automatically with the information of the product selected every time a product is clicked in the table. If the user wishes to delete the text in the text fields fast, he can click on the “Clear Fields” button.

The products that fill up the table can be generated from a .csv file by clicking on the “Import Products” button. The admin has to pay attention to the fact that if he presses on that button, all of the modifications made on the menu will be lost.

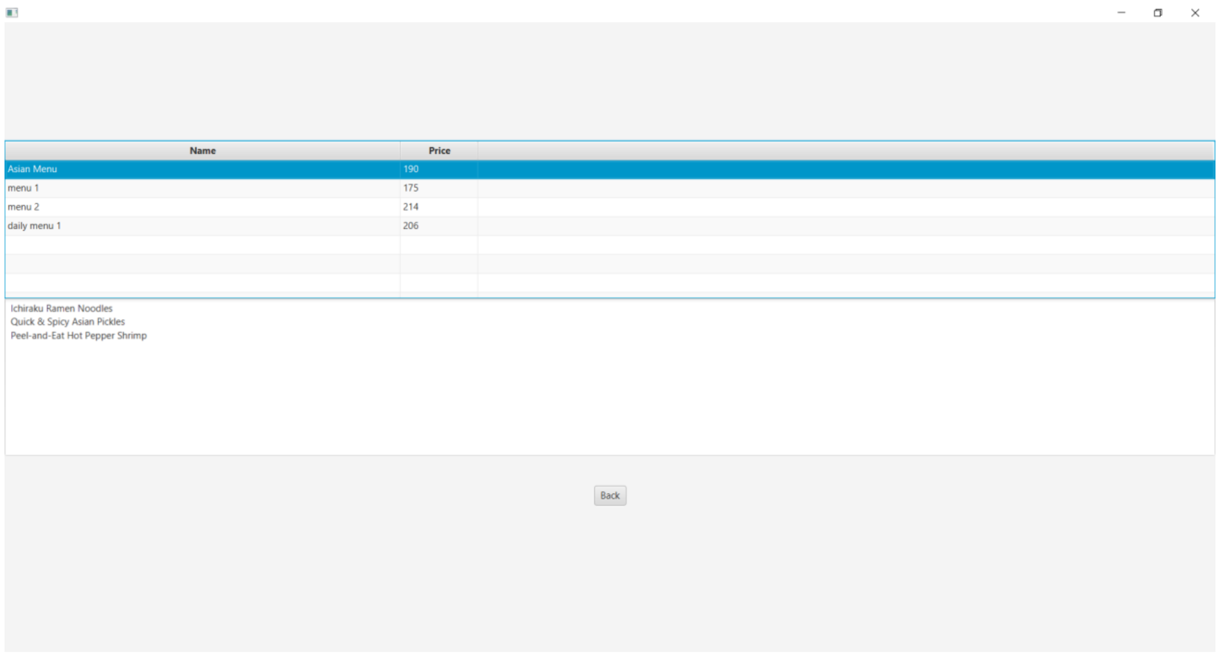
The administrator can also choose to see or add composite products. Composite products are made from a selection of base products, so when the “Create Composite Product” button is pressed, a window containing the same base products table will show up.



Title	Rating	Calories	Protein	Fat	Sodium	Price
Cilantro-Tomato Salsa	3.75	23	1	0	7	32
Cauliflower-Leek Pur...	3.125	23	2	0	149	13
Red and Green Toma...	3.125	23	1	0	552	38
The Original Three-In...	0.0	23	1	1	6	47
Shrimp Sates with Sp...	3.75	23	1	2	4	78
Coriander-Herb Spic...	3.75	24	1	1	1911	51
Arugula Salad with H...	2.5	24	1	0	12	21
Beef Stock	4.375	24	4	1	87	26

BaseProduct added to CompositeProduct

To create a composite product, the admin has to select the base products he wants from the table, click on the “Add to Composite Product” button after every selection, type in a name for the composite product, and finally, click on the “Finalize Composite Product” button. A message will be written saying that the composite product was successfully created, and the new composite product can be seen by going back to the main page of the administrator and clicking on the “View Composite Products” button.

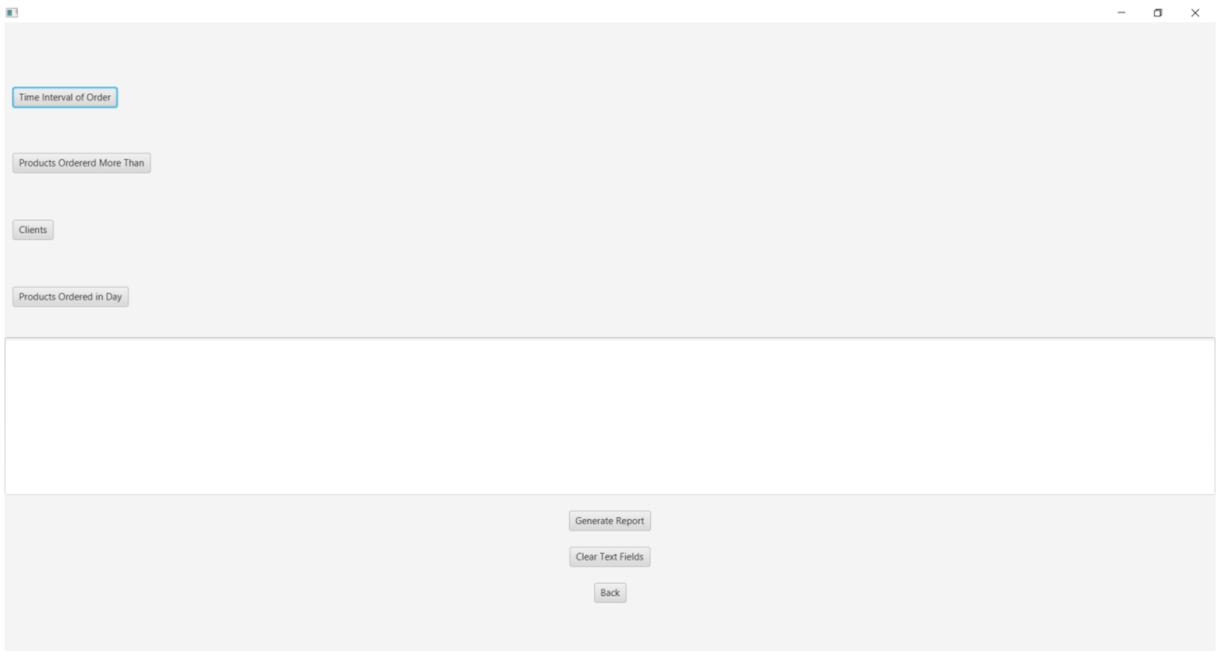


Name	Price
Asian Menu	190
menu 1	175
menu 2	214
daily menu 1	206

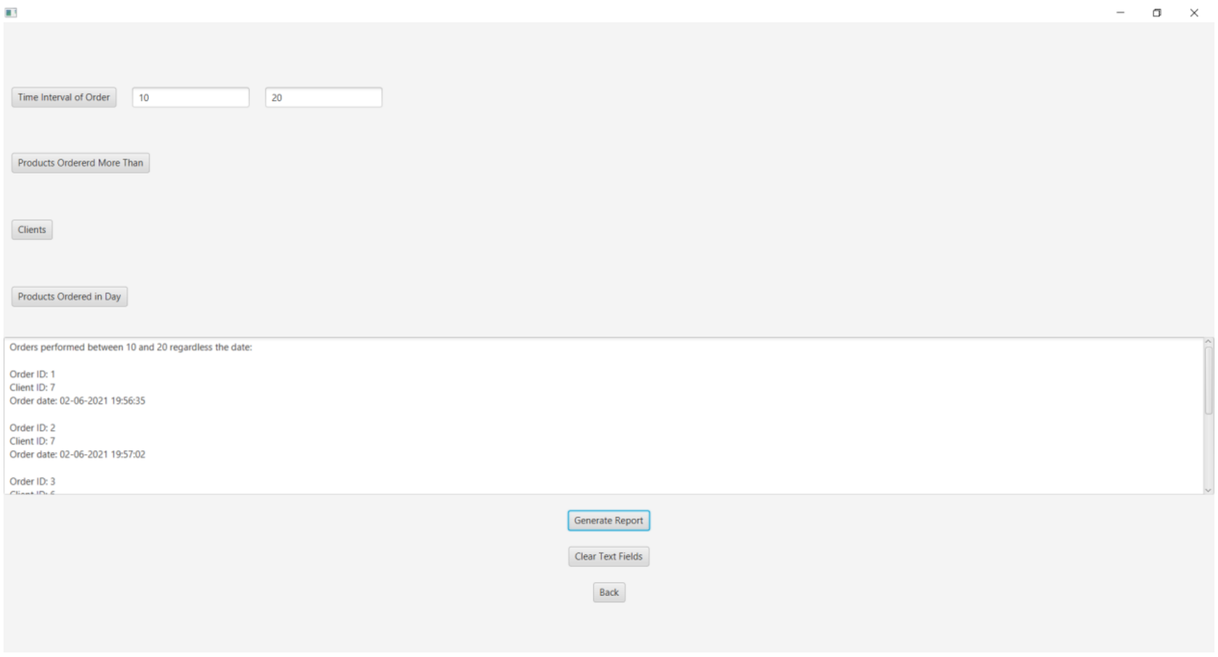
Ichiraku Ramen Noodles
Quick & Spicy Asian Pickles
Peel-and-Eat Hot Pepper Shrimp

When a composite product is pressed in the table, its base products will be shown in the text area below.

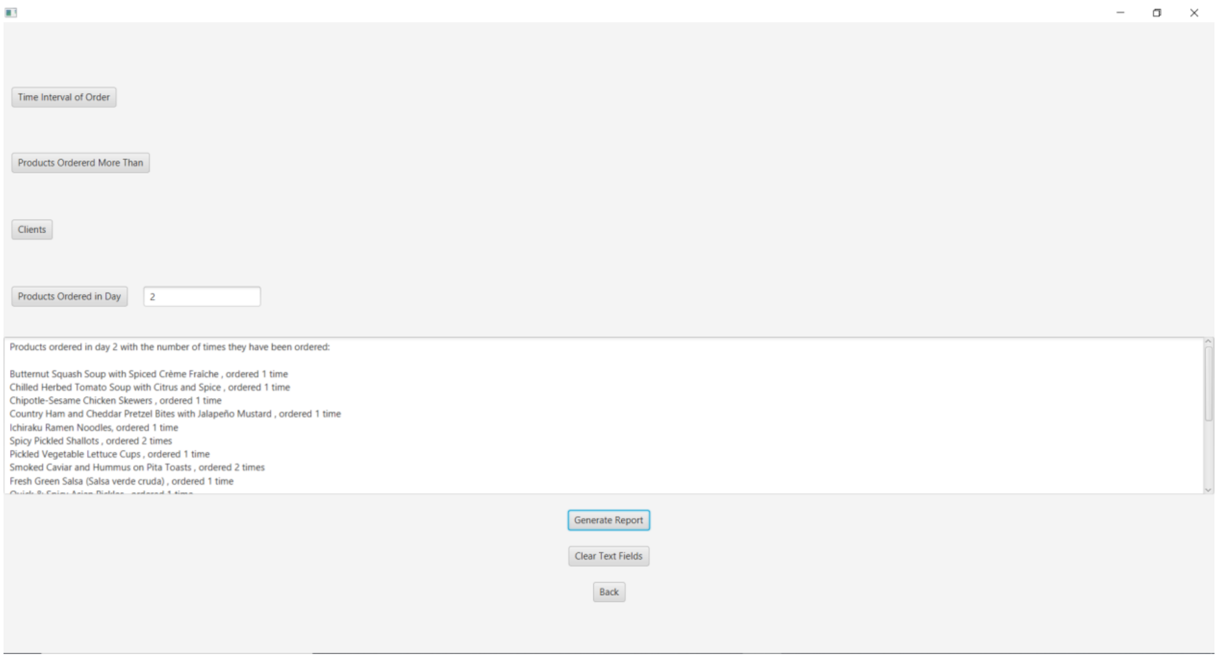
Finally, the administrator can generate various reports, by clicking on the “Generate Reports” button on the admin homepage. A new window will appear, with four buttons that represent the reports that can be generated.



If one button is pressed, text fields will appear, and the admin will have to input the necessary fields for the generation of a report, and press the “Generate Report” button. The results of the report will be displayed in the text area below.

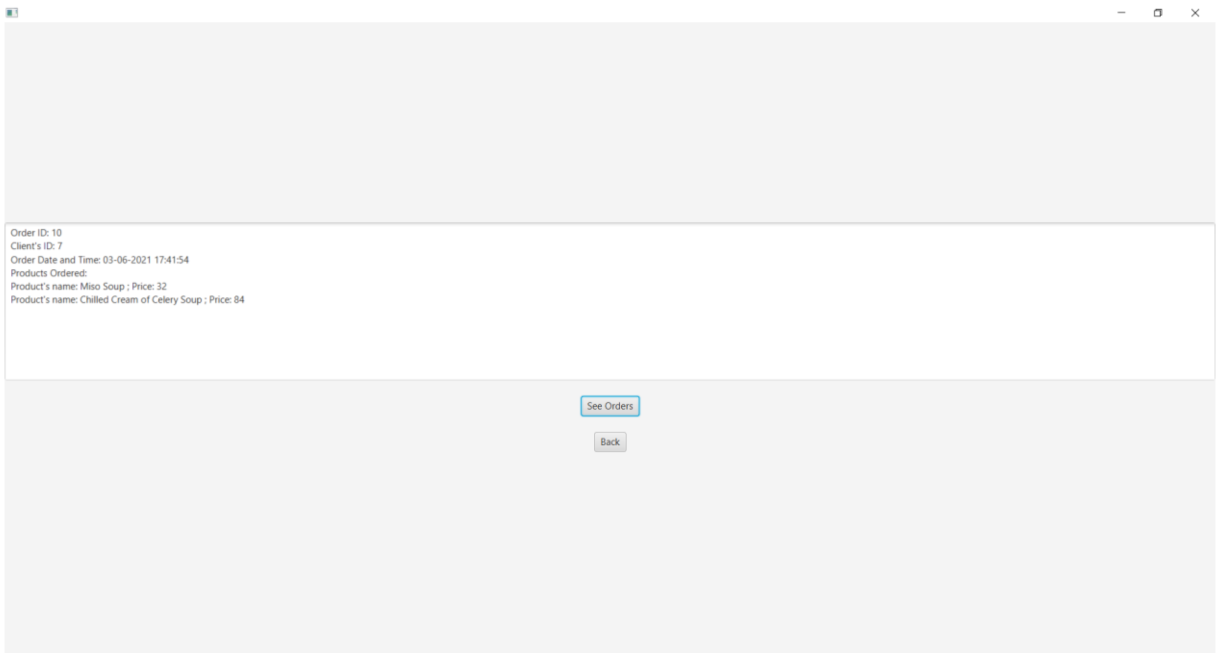


```
Orders performed between 10 and 20 regardless the date:

Order ID: 1
Client ID: 7
Order date: 02-06-2021 19:56:35

Order ID: 2
Client ID: 7
Order date: 02-06-2021 19:57:02

Order ID: 3
```



```
Products ordered in day 2 with the number of times they have been ordered:

Butternut Squash Soup with Spiced Crème Fraîche , ordered 1 time
Chilled Herbed Tomato Soup with Citrus and Spice , ordered 1 time
Chipotle-Sesame Chicken Skewers , ordered 1 time
Country Ham and Cheddar Pretzel Bites with Jalapeño Mustard , ordered 1 time
Ichiraku Ramen Noodles, ordered 1 time
Spicy Pickled Shallots , ordered 2 times
Pickled Vegetable Lettuce Cups , ordered 1 time
Smoked Caviar and Hummus on Pita Toasts , ordered 2 times
Fresh Green Salsa (Salsa verde cruda) , ordered 1 time
```

The last user type that can log in and use the application is the employee. The employee will be notified of every new order (a new order means an order that was placed in the same run of the application). If the employee presses on the “See Orders” button, the information regarding the new orders will be displayed in the text area.



```
Order ID: 10
Client's ID: 7
Order Date and Time: 03-06-2021 17:41:54
Products Ordered:
Product's name: Miso Soup ; Price: 32
Product's name: Chilled Cream of Celery Soup ; Price: 84
```

5. Conclusions and Further Development

A thing I have learned from this first task is the importance of dividing the objectives at the very beginning of the problem. A lack of rigor in the planning phase can set back the implementation, so the best approach is to "divide and conquer" the big problem into smaller tasks and implementing, testing and debugging them before going forward.

By the means of this project I managed to improve my knowledge about the Composite and Observer design patters, as well as the Serialization and Deserialization mechanisms.

As for further developments, the application could be extended in a lot of ways. A more thorough registration process could be implemented, where the client could enter his name, e-mail address, delivery information etc. The employee window could be improved, letting the server choose if he wants to deliver the order. The admin window could be improved, letting the administrator apply discounts.

6. Bibliography

- Class Materials provided by the professor and the teaching assistant.
- https://javahungry.blogspot.com/2014/03/hashmap-vs-hashtable-difference-with-example-java-interview-questions.html
- https://www.tutorialspoint.com/java/java_serialization.htm
- https://www.geeksforgeeks.org/serialization-in-java/
- https://stackabuse.com/how-to-get-current-date-and-time-in-java/
- https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html
- https://dzone.com/articles/how-to-read-a-big-csv-file-with-java-8-and-stream
- https://www.baeldung.com/java-observer-pattern