**FOOD DELIVERY MANAGEMENT**

**SYSTEM DOCUMENTATION**

1. Assignment objective

Design and implement a food delivery management system for a catering company. The client can order products from the company’s menu. The system should have three types of users that log in using a username and a password: **administrator**, **regular employee**, and **client**.

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.

Secondary objectives:

* 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. A log of events
* Use the Composite Design Pattern for defining the classes MenuItem, BaseProduct and CompositeProduct
* Use the Observer Design Pattern to notify the employee each time a new order is created.
* The menu items, performed orders and user information will be persisted using serialization so as to be available at future system executions by means of deserialization

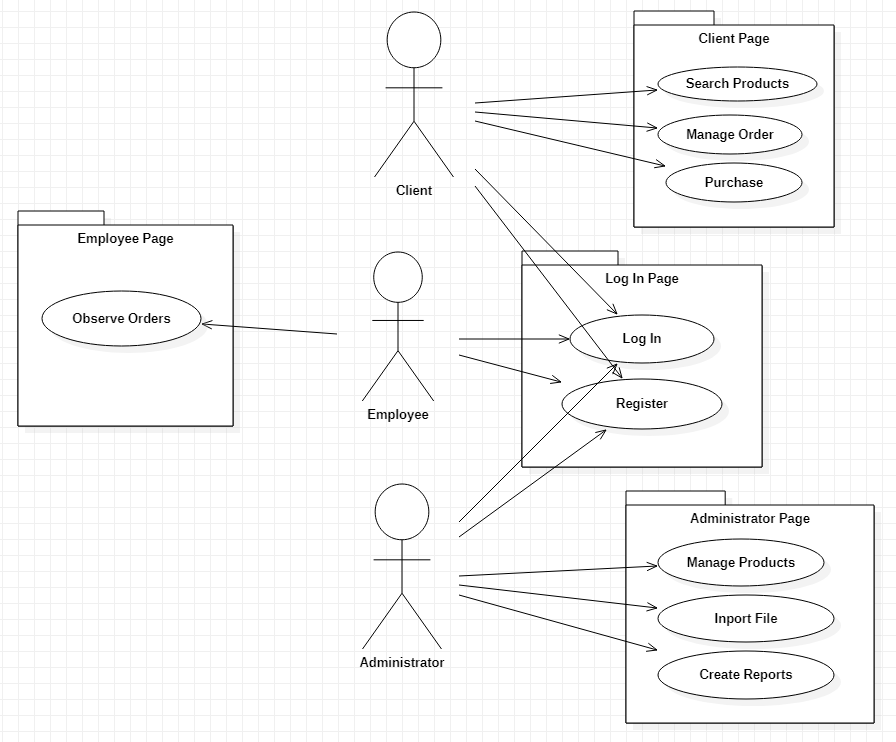
1. Problem analysis
2. Problem analysis

All users will have to log in or register. After this they will be considered one of 3 types: client, admin or employee. The clients will be able to inspect the menu and use search options to navigate easier. When they decide on what they want they will add any number of products to their order and send it, receiving a bill. The Administrator will be able to modify the data in the menu, by adding, deleting, modifying, importing or creating combinations. They will also be able to receive reports about the system (mentioned at Section 1). The employees will only be able to see orders. They will be notified when new orders appear, and I will also save the older orders for them to see.

1. Modelling of the problem

The user will be able to input 13 constraints for the search and will be able to add products to a container for their order. This container will be modifiable until they are ready to send the order. This will generate a bill and be saved by the system. The employee will be notified about this new order. The administrator will be able to change the data as they see fit, with input fields, and give constrains for the reports that they can generate. If an order is eligible for a report, it will be added and all of the order’s information is available to the employees, but not freely to the administrator who can see them only in reports.

1. Use cases & scenarios



All three actors have to log in first. There are 2 choices here, to log in with a known account, or to register a new account and then log in with it.

Register – Success: The name is not taken

-Failure: The chosen name is already taken.

Log in – Success: The username and password combination is correct.

-Failure: The username does not exist, or the username exists and the password is incorrect.

After this, each actor will be given a role. The client is taken to the client page where he can Inspect the menu. He has 3 choices to make that don’t block one another:

Search – Success: The search data is in the correct format and there are results that match

-Indecisive: The search data has the correct format, however there are no products that match it

-Failure: The search data is in the incorrect format

Manage Order – Success: The user adds or removes products from their order and no exceptions occurs

-Failure: The user does not select the product correctly

Purchase – Success: The user filled their order and bought it, receiving a bill

-Failure: The user had an empty order or products that do not exist

The employee is taken to his separate page, where he can do only one thing, inspect the orders, while being notified of new orders. There is no failure/success condition.

The administrator is taken to his own page, where he can do multiple things.

Manage products – Success: The user managed to add/remove/modify the data by choosing free names and existing products, with input data that respects the format

-Failure: The user chose taken names, inexistent products or sent input data that did not respect the required format

Import file – Success: The given file exists and has usable data that was parsed without problems

-Failure: The given file did not exist or contained unparseable data that most likely did not respect the needed format

Generate report-Success: The requested report was created using valid input data and there was matching data

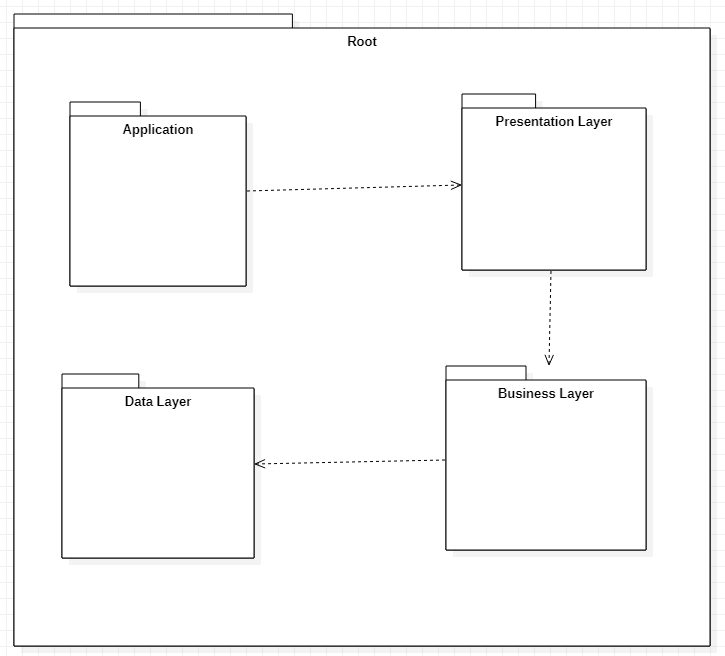
-Indecisive: The requested report was created using valid input data, however it is empty because there were no matches

-Failure: The report was not created, most likely because of invalid input data

1. Design
2. Packages

There are 3 main packages: the Data Layer, that will work with data by reading or writing it in files and other related operations; the Business Layer, that will process data, having the container classes for the data and classes that work with that data, parsing it, storing it and sending it; the Presentation Layer, which represents the front end of the application, containing the GUI classes and their respective controllers.

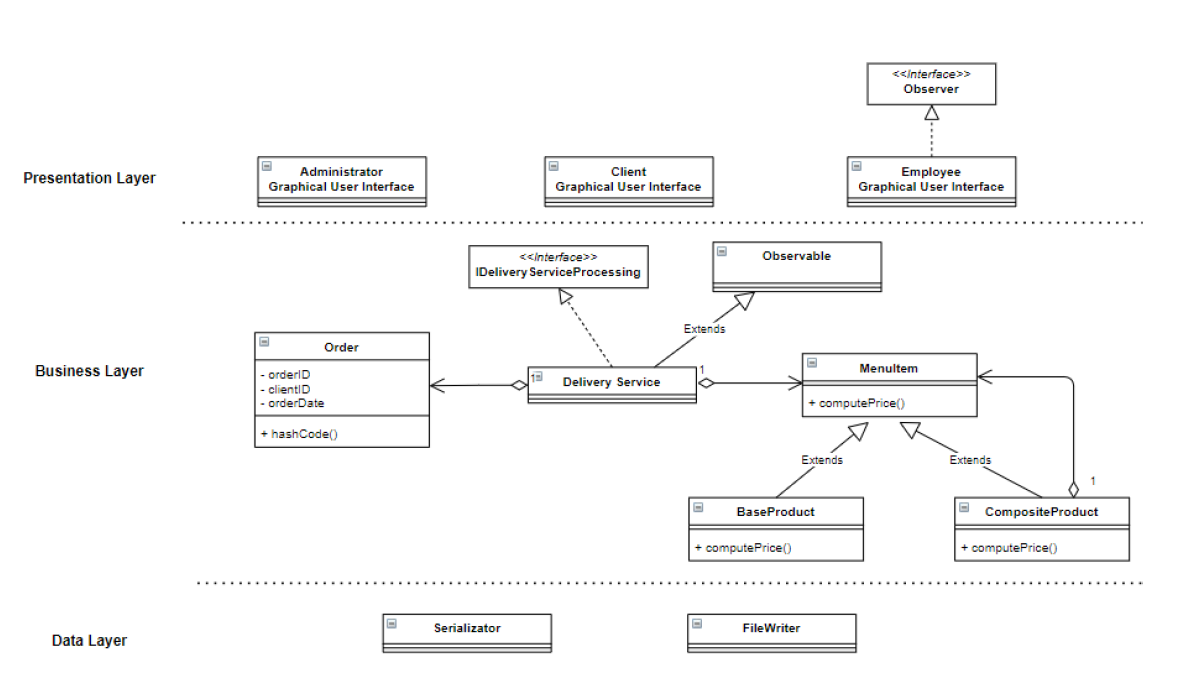
Apart from those we have the application package that holds the main method and class, whose only job is to start the application; and the root package, which is the project’s folder that contains all java source code.



It can be seen from the image that the model view controller design pattern is respected. The dependencies are one way, because, the lower the level, the higher the autonomy of the classes and methods is.

1. Classes

Here is the UML diagram of the classes:



// Placeholder //

The 3 product classes: Menuitem, BaseProduct and CompositeProduct are using the Composite Design Pattern, having a parent class (MenuItem) that seems to be an abstract class, the BaseProduct which is the most simple implementation of menuItem and the CompositeProduct (that gives the name of the design pattern), extending the MenuItem class and also having an internal collection of MenuItem, therefore being a complex class formed out of other similar classes.

The DeliveryService and Employee classes are using the Observer Design Pattern to comunicate, DeliveryService being the Observable and employee the Observer. Because of the

1. Data structures

There are 2 main data structures Order and MenuItem.

* Order: has an orderId an clientId and a DateTime orderDate that are used to identify an order, This is also used as the key in a hashMap by overwriting it’s hashCode() function
* MenuItem has 7 attributes, name, rating, calories, proteins, fat, sodium and price. This is an abstract class and thererfore its 2 children will be used for actually storing the information.
* BaseProduct is a simple class that extends MenuItem and dosen’t add much to it
* CompositeProduct is a more complex one, which, apart from extending MenuItem, also has a collection of MenuItems, storing a lot more than its counterpart, but also being more difficult to manage.

1. Interfaces

There will be only one interface, the IDeliveryServiceProcessing interface that will be implemented by the DeliveryServiceProcessing class. It will contain all its methods in order to be able to implement the Design by Contract design model.

1. Algorithms

There are no noteworthy algorithms in this project, however lambda functions and streams are a common occurrence. Finding the number of occurrences in a Map<SomeClass ,Collection<OtherClass>> using streams is an interesting challenge.

Apart from that, we mostly work with collections and data management for which I am not too stressed since I‘ve gotten a lot of experience from the past laboratories.

1. Design decisions

For the log in screen there will be 2 text boxes and 2 labels, for the username respectively the password. Additionally, there will also be 2 buttons, one for registration and one for logging in.

For the client tab we will have a main table, a panel with the search boxes their labels and a button, and an additional panel for orders that will contain a textbox 3 buttons and a smaller table.

For the administrator we will have again a main table, a panel with 7 textboxes and 3 buttons for working with base products, a panel with a textbox 4 buttons and a smaller table for working with composite products and a panel with reports buttons and a few textboxes. Somewhere in there a import button with a textbox will be thrown in.

For the employee we will have 2 tables, one with old orders and one with new ones.

1. Implementation
2. Classes and methods

Here are the classes and their methods, divided by packages. Of course, all of them have their needed getters, setters and constructors that I did not include here.

* Application:

public static void main(String[] args) {  
 SwingUtilities.*invokeLater*(new Runnable() {  
 */\*\*  
 \* Starts a thread for the Swing graphical interface and completes the setup for the frame.  
 \*/* @Override  
 public void run() {  
 IntegerMutable id = new IntegerMutable();  
 DeliveryService deliveryService = new DeliveryService();  
  
 JFrame clientFrame = new ClientFrame("Client", id, deliveryService);  
 clientFrame.setSize(1400, 800);  
 clientFrame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 clientFrame.setVisible(false);  
  
 JFrame adminFrame = new AdminFrame("Administrator", deliveryService);  
 adminFrame.setSize(1400, 800);  
 adminFrame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 adminFrame.setVisible(false);  
  
 JFrame employeeFrame = new EmployeeFrame("Employee", deliveryService);  
 employeeFrame.setSize(1400, 800);  
 employeeFrame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 employeeFrame.setVisible(false);  
  
 JFrame frame = new MainFrame("Food delivery management system", clientFrame, adminFrame,  
 employeeFrame, id);  
 frame.setSize(500, 300);  
 frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 frame.setVisible(true);  
 }  
 });  
}

* Business

*/\*\*  
 \* Class for base products.  
 \*/*public class BaseProduct extends MenuItem

*/\*\*  
 \* A constructor that parses a string  
 \*  
 \** ***@param*** *str A string containing the attributes separated by ","  
 \** ***@throws*** *WrongSizeOfCollectionException The string does not have the right number of attributes  
 \*/*public BaseProduct(String str) throws WrongSizeOfCollectionException

*/\*\*  
 \* Calculates the price  
 \*  
 \** ***@return*** *The price  
 \*/*@Override  
public Integer computePrice() {  
 return getPrice();  
}

*/\*\*  
 \* Class for composite products.  
 \* Using setters for anything but the title on this one is not recommended.  
 \*  
 \*/*public class CompositeProduct extends MenuItem {  
 private final List<MenuItem> products;

*/\*\*  
 \* Adds a MenuItem to the internal list, then updates the base attributes  
 \*  
 \** ***@param*** *product A MenuItem to be added  
 \*/*public void addProduct(MenuItem product) {  
 products.add(product);  
 update();  
}  
  
*/\*\*  
 \* Removes a MenuItem from the internal list, then updates the base attributes  
 \*  
 \** ***@param*** *product A MenuItem to be removed  
 \*/*public void removeProduct(MenuItem product) {  
 products.remove(product);  
 update();  
}

*/\*\*  
 \* Recomputes the base attributes by adding them from the internal product list  
 \*/*private void update()

*/\*\*  
 \* Computes the price of the product.  
 \*  
 \** ***@return*** *The price, an Integer.  
 \*/*@Override  
public Integer computePrice() {  
 Integer finalPrice = 0;  
 for (MenuItem product : products) {  
 finalPrice += product.getPrice();  
 }  
 return finalPrice;  
}  
  
*/\*\*  
 \* A static method that transforms a CompositeProducts' ingredients names in a string, separated by ","  
 \*  
 \** ***@param*** *menuItem The MenuItem to be decomposed  
 \** ***@return*** *CompositeProducts' ingredients names in a string, separated by ",", "EMPTY" if it has  
 \* no products or "" if it is not a CompositeProduct  
 \*/*public static String decomposeProduct(MenuItem menuItem)

*/\*\*  
 \* Abstract Class for the products to be ordered  
 \*/*public abstract class MenuItem implements Serializable {  
 private String title;  
 private Double rating;  
 private Integer calories;  
 private Integer protein;  
 private Integer fat;  
 private Integer sodium;  
 private Integer price;

*/\*\*  
 \* Computes the price  
 \*  
 \** ***@return*** *The price, Integer  
 \*/*public abstract Integer computePrice();

*/\*\*  
 \* Class for the order data  
 \*/*public class Order implements Serializable {  
 private final Integer orderID;  
 private final Integer clientID;  
 private final LocalDateTime orderDate;

@Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Order order = (Order) o;  
 return Objects.*equals*(orderID, order.orderID) && Objects.*equals*(clientID, order.clientID) &&  
 Objects.*equals*(orderDate, order.orderDate);  
 }  
  
 */\*\*  
 \* Overwritten hashCode for the hashMap encription  
 \*  
 \** ***@return*** *The result of the hash function  
 \*/* @Override  
 public int hashCode() {  
 return Objects.*hash*(orderID, clientID, orderDate);  
 }  
}

*/\*\*  
 \* Class for user credentials  
 \*/*public class UserData implements Serializable {  
 private final Integer id;  
 private final String username;  
 private final String password;  
 private final UserType type;

*/\*\*  
 \* Enum for user types  
 \*/*public enum UserType {  
 *client*,  
 *administrator*,  
 *employee*}

*/\*\*  
 \* Interface for DeliveryServiceProcessing.  
 \* Contains all its methods.  
 \* I was supposed to implement Design by Contract, however I didn't manage to do it  
 \*/*

public interface IDeliveryServiceProcessing {  
  
 */\*\*  
 \* Force serialization of internal values  
 \*/* void save();  
  
 */\*\*  
 \* Imports the MenuItems from a file  
 \*  
 \** ***@param*** *filePath The path of the file  
 \** ***@throws*** *IOException Error when working with a file  
 \** ***@throws*** *WrongSizeOfCollectionException The file has an improper data format  
 \*/* void importCSV(String filePath) throws IOException, WrongSizeOfCollectionException;  
  
 */\*\*  
 \* Creates a base product and serializes the list  
 \*  
 \** ***@param*** *title A name  
 \** ***@param*** *rating A rating  
 \** ***@param*** *calories A nb of calories  
 \** ***@param*** *protein A nb of proteins  
 \** ***@param*** *fat A nb of fats  
 \** ***@param*** *sodium A nb of sodium  
 \** ***@param*** *price A price  
 \** ***@throws*** *ExistentUniqueFieldException A unique field (name) already contains the value  
 \*/* void createBaseProduct(String title, Double rating, Integer calories, Integer protein, Integer fat,  
 Integer sodium, Integer price) throws ExistentUniqueFieldException;  
  
 */\*\*  
 \* Removes a product from the list  
 \*  
 \** ***@param*** *toRemove The name of the product to be removed  
 \** ***@throws*** *ObjectNotFoundException The searched object has not been found  
 \*/* void removeProduct(String toRemove) throws ObjectNotFoundException;  
  
 */\*\*  
 \* Modifies a base product  
 \*  
 \** ***@param*** *toModify The name of the product to be modified  
 \** ***@param*** *newTitle A new name  
 \** ***@param*** *newRating A new rating  
 \** ***@param*** *newCalories A new nb of calories  
 \** ***@param*** *newProtein A new nb of proteins  
 \** ***@param*** *newFat A new nb of fats  
 \** ***@param*** *newSodium A new nb of sodium  
 \** ***@param*** *newPrice A new price  
 \** ***@throws*** *ObjectNotFoundException The searched object has not been found  
 \** ***@throws*** *ExistentUniqueFieldException A unique field (name) already contains the value  
 \*/* void modifyProduct(String toModify, String newTitle, Double newRating, Integer newCalories,  
 Integer newProtein, Integer newFat, Integer newSodium, Integer newPrice)  
 throws ObjectNotFoundException, ExistentUniqueFieldException;  
  
 */\*\*  
 \* Creates a composite product  
 \*  
 \** ***@param*** *title The name  
 \** ***@param*** *menuItems The ingredients  
 \** ***@throws*** *ExistentUniqueFieldException A unique field (name) already contains the value  
 \*/* void createCompositeProduct(String title, List<MenuItem> menuItems) throws ExistentUniqueFieldException;  
  
 */\*\*  
 \* Creates an order and adds it to the hashTable  
 \*  
 \** ***@param*** *clientID The id of the client  
 \** ***@param*** *menuItems The products ordered  
 \** ***@return*** *The total price  
 \*/* Integer createOrder(Integer clientID, List<MenuItem> menuItems);  
  
 */\*\*  
 \* Searches for a MenuItem in the menu with the given name  
 \*  
 \** ***@param*** *name The name to search for  
 \** ***@return*** *The MenuItem if found, else null  
 \*/* MenuItem searchByName(String name);  
  
 */\*\*  
 \* Searches for all products that respect the given condittions.  
 \* A condition not specified will be taken as true.  
 \*  
 \** ***@param*** *titleLike A substring of the name  
 \** ***@param*** *ratingMin A min for rating  
 \** ***@param*** *ratingMax A max for rating  
 \** ***@param*** *caloriesMin A min for calories  
 \** ***@param*** *caloriesMax A max for calories  
 \** ***@param*** *proteinMin A min for protein  
 \** ***@param*** *proteinMax A max for protein  
 \** ***@param*** *fatMin A min for fat  
 \** ***@param*** *fatMax A max for fat  
 \** ***@param*** *sodiumMin A min for sodium  
 \** ***@param*** *sodiumMax A max for sodium  
 \** ***@param*** *priceMin A min for price  
 \** ***@param*** *priceMax A max for price  
 \** ***@return*** *A list of products that respect the condittions  
 \*/* List<MenuItem> searchProduct(String titleLike, Double ratingMin, Double ratingMax, Integer caloriesMin,  
 Integer caloriesMax, Integer proteinMin, Integer proteinMax, Integer fatMin,  
 Integer fatMax, Integer sodiumMin, Integer sodiumMax, Integer priceMin,  
 Integer priceMax);  
}

public class DeliveryService extends Observable implements IDeliveryServiceProcessing {  
 private static final String *menuSaveFile* = "menu.ser";  
 private static final String *ordersSaveFile* = "orders.ser";  
  
 private final Map<Order, Collection<MenuItem>> orders;  
 private final Collection<MenuItem> menu;

* Data

*/\*\*  
 \* Class for managing writing to files  
 \*/*public class MyFileWriter

*/\*\*  
 \* Creates a bill for a order  
 \*  
 \** ***@param*** *filePath The file path  
 \** ***@param*** *list The ordered items  
 \** ***@param*** *price The total price  
 \** ***@param*** *id The id of the client  
 \*/*public static void createBill(String filePath, List<MenuItem> list, Integer price, Integer id)

*/\*\*  
 \* 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.  
 \*  
 \** ***@param*** *filePath The file path  
 \** ***@param*** *hashMap The hashMap of the orders  
 \** ***@param*** *hourMin The minimum hour  
 \** ***@param*** *hourMax The maximum hour  
 \*/*public static void intervalReport(String filePath, Map<Order, Collection<MenuItem>> hashMap, Integer hourMin,  
 Integer hourMax)

*/\*\*  
 \* The products ordered more than a specified number of times so far.  
 \*  
 \** ***@param*** *filePath The file path  
 \** ***@param*** *hashMap The hashMap of the orders  
 \** ***@param*** *nbOrderedMin The minimum number of orders done  
 \*/*public static void orderedReport(String filePath, Map<Order, Collection<MenuItem>> hashMap, Integer nbOrderedMin)

*/\*\*  
 \* The clients that have ordered more than a specified number of times and the value of  
 \* the order was higher than a specified amount.  
 \*  
 \** ***@param*** *filePath The file path  
 \** ***@param*** *hashMap The hashMap of the orders  
 \** ***@param*** *times The min times a client ordered  
 \** ***@param*** *value The min value of the orders  
 \*/*public static void clientsReport(String filePath, Map<Order, Collection<MenuItem>> hashMap, Integer times,  
 Integer value)

*/\*\*  
 \* The products ordered within a specified day with the number of times they have been ordered.  
 \*  
 \** ***@param*** *filePath The file path  
 \** ***@param*** *hashMap The hashMap of the orders  
 \** ***@param*** *date The date of the day  
 \*/*public static void dayReport(String filePath, Map<Order, Collection<MenuItem>> hashMap, LocalDate date)

*/\*\*  
 \* Class for managing serialization and deserialization from files  
 \*/*public class Serialization

*/\*\*  
 \* Serializes any object  
 \*  
 \** ***@param*** *filePath The path of the file  
 \** ***@param*** *object The serializable object  
 \*/*public static void serialize(String filePath, Serializable object)

*/\*\*  
 \* Deserializes a List<MenuItem> representing the menu  
 \*  
 \** ***@param*** *filePath The path of the file  
 \** ***@return*** *The menu  
 \*/*public static List<MenuItem> deserializeMenu(String filePath)

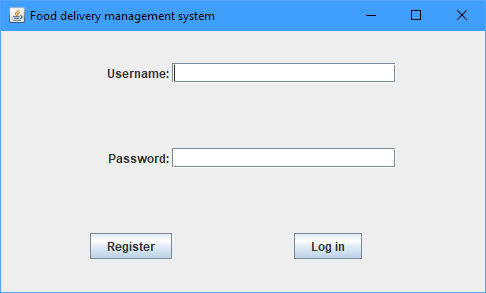
*/\*\*  
 \* Deserializes a List<UserData> representing the user credentials  
 \*  
 \** ***@param*** *filePath The path of the file  
 \** ***@return*** *The credentials list  
 \*/*public static List<UserData> deserializeUsers(String filePath)

*/\*\*  
 \* Deserializes a Map<Order, Collection<MenuItem>> representing the hashmap of the orders  
 \*  
 \** ***@param*** *filePath The path of the file  
 \** ***@return*** *The orders  
 \*/*public static Map<Order, Collection<MenuItem>> deserializeHashMap(String filePath)

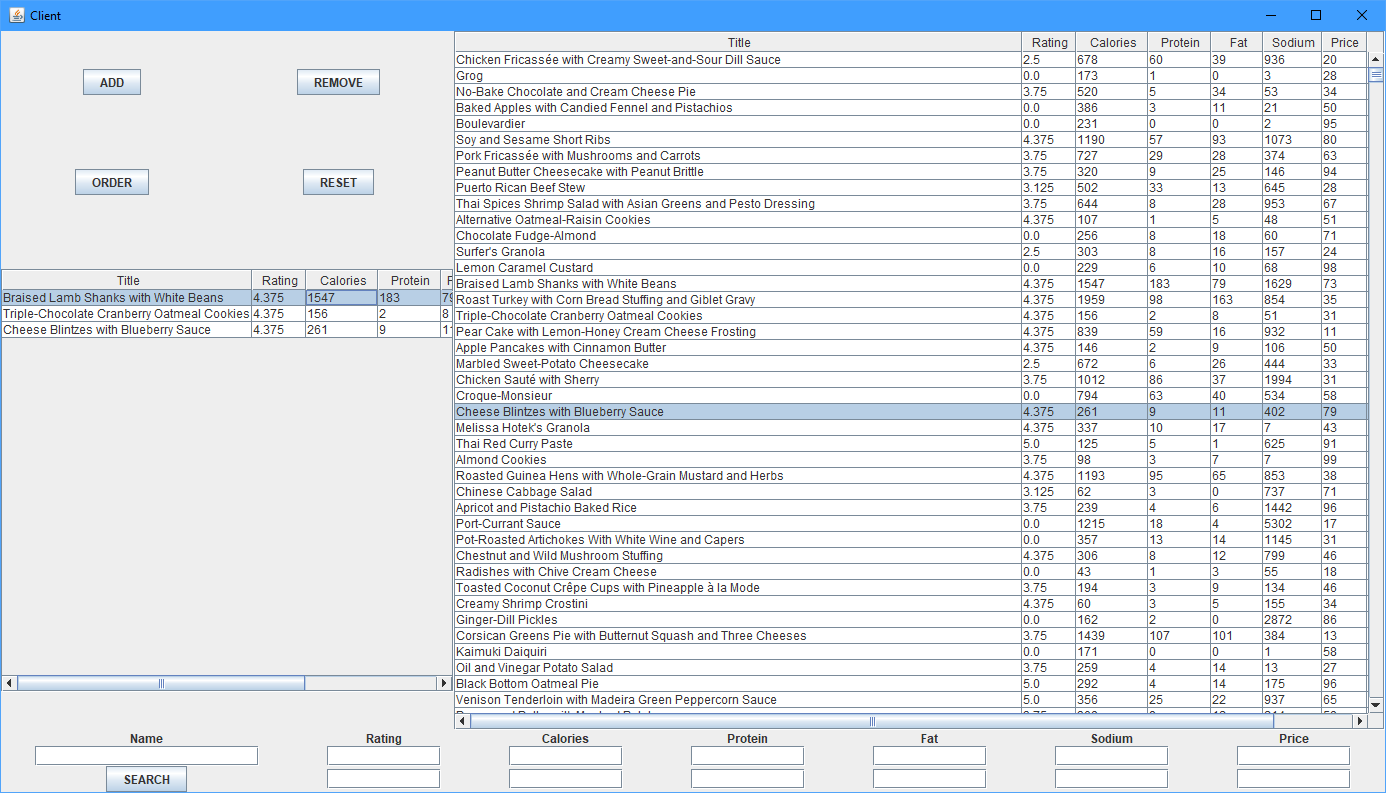
*/\*\*  
 \* Deserializes a List<Order> representing the orders to be viewed by the employees  
 \*  
 \** ***@param*** *filePath The path of the file  
 \** ***@return*** *The orders  
 \*/*public static List<Order> deserializeEmployeeOrders(String filePath)

//The GUI classes are massive and I think that I showed the important methods already so your free trial has expired

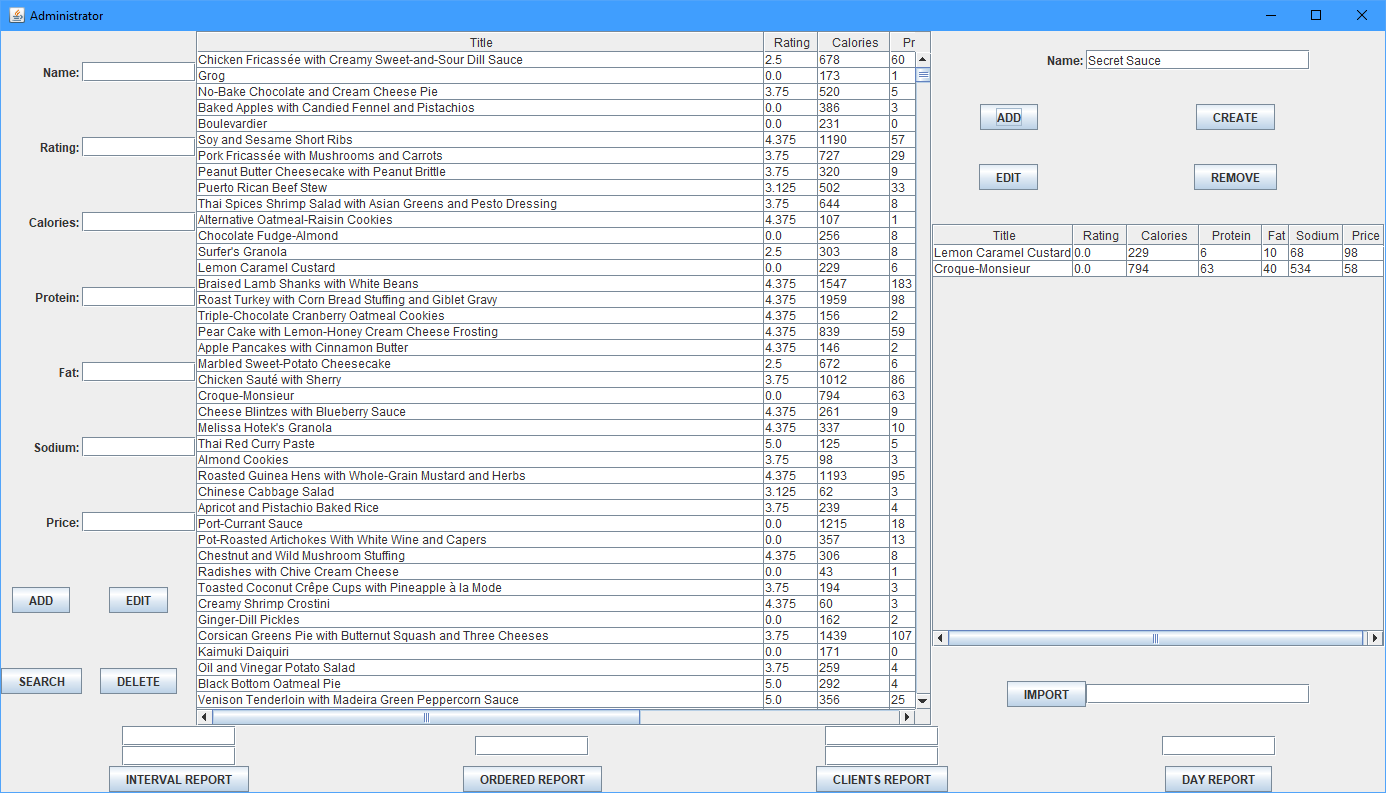
1. User interface



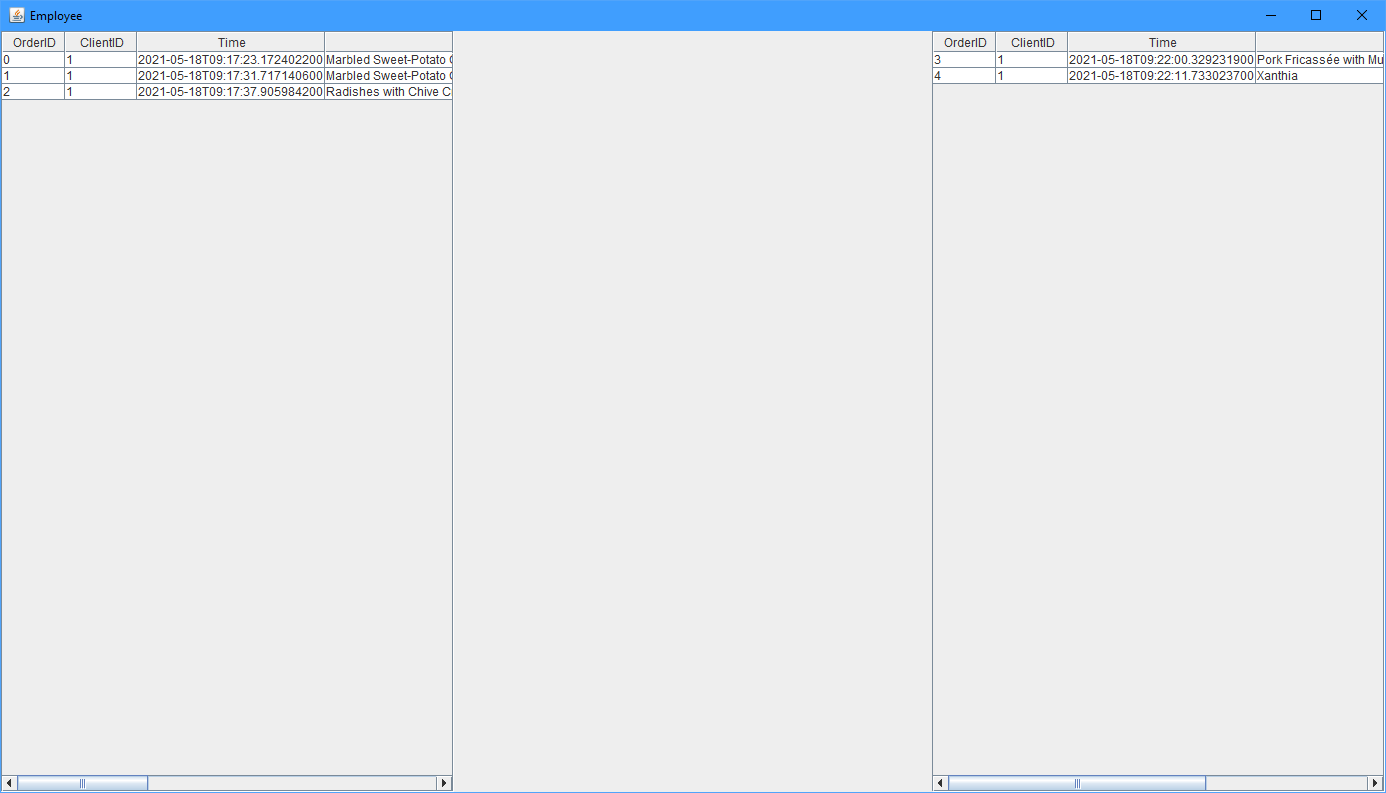
The log in screen is a classic one, everyone saw it at least once, so it is self-explanatory.



This is the client screen. On the right you have the table, on the left you have a panel for managing the order. To add you must select a row from the big table and to remove you must select one from the small table. Down you have the search panel, leaving empty fields will make that check to be passed as true. Where there are 2 textboxes, the upper one is the min value and the lower one is the maximum.



This is the administrator window. In the middle you have the main table, on the left a panel for managing BaseProducts and on the right one to handle the CompositeProducts and the extra import button with its textbox. The second table and the Down you have the reports pane. To be completely honest, I can’t remember which textbox means what, I had a paper around here, just read what the reports are supposed to do and you’ll figure it out.



The employee window contains only these 2 tables. The one on the left has old orders, while the one on the right are new orders, which will be moved to the other table at the next run of the application. Ignore the empty center, think of it like space for extra features.

1. Results

I did not make explicit tests with Junit or anything like it. However, I can’t say that I didn’t test it, because I did. A lot. Apart from checking that the number fields are numerical (I completely forgot about that), all textboxes are full proof with custom error messages. No empty orders, no nameless objects, no duplicates, even CompositeProducts with an empty list are handled.

1. Conclusion

This was an interesting project that helped me learn how to use streams and lambda functions. The observer and observable are an interesting concept, however they are deprecated, so making us use something newer wouldn’t have been bad. The composite design pattern is a bit weird and I don’t know how much I will use it, but I can see its usefulness in some situations.

This could be improved by adding a lot more functionality. More buttons, more methods, more everything. This is a good base and a good exercise for our careers and I feel that working more on this and expanding it would be nice.

1. Bibliography

<https://docs.oracle.com/en/>

<https://stackoverflow.com/>