**Documentation Homework 4**

**Food Delivery Management System**

First Name: Marius

Last Name: Măcean

Group: 30423/2

Laboratory Professor: Marcel Antal

**Table of contents**

1. Functional Requirements……………………………………………………………………………..……. 3
2. Objectives…………………………………………………………………………………………….….… 3
   1. Main Objective…………………………………………………………………………………..… 3
   2. Secundary Objectives………………………………………………………………………...…..... 3

.

1. Analysing the Problem………………………………………………………………………………...….… 4
2. Projection……………………………………………………………………………………………..….….. 5
   1. Data Structures…………………………………………………………………………….…..…... 5
   2. Main Class Diagrams..…...………………………………………………………………….....….. 5
   3. Algorithms……………………………………………………………………………………….… 7
3. Implementation…………………………………………………………………………………………..… 10
4. Testing……………………………………………………………………………………………….….…. 15
5. Conclusions and Future Developments……………………………………………………………….…… 16
6. Bibliography………………………………………………………………………………………………. 16
7. **Functional Requirements**

* Use an Object-oriented programming design, classes with maximum 300 lines, methods with maximum 30 lines, Java naming conventions.
* Implement the class diagram and choose appropriate data structures for saving the “Orders” and the “MenuItems” classes.
* Define the class “BaseProduct” with the following fields: title, rating, calories, proteins, fats, sodium and price. The data is read from a csv file (“products.csv”) by using streams and split each line in 7 parts: title, rating, calories, proteins, fats, sodium and price, and create a list of objects of type “BaseProduct”. The file contains duplicate dishes so we have to make sure to select only one dish.
* Create a graphical user interface including:
* A log in window: it will give the user the possibility to log with an existing account in or to create a new one.
* A window for administrator operations; the administrator should be able to: import the initial set of product which will populate the menu from a .csv file, add a new product, delete a product, modify a product, create products composed of several products and 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*.
* A window for client operations; the client should be able to: 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 in the menu based on one or multiple criteria such as keyword, rating, number of calories/proteins/fats/sodium/price and create an order consisting of several products – each order needs to have a date and a time for the order. A bill will also be generated that will list the products that were ordered and the total price of the placed order.
* Use lambda expressions and stream processing for generating the administrator specific reports such as:
* *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*.
* Use the Lambda expressions and stream processing to implements the search functionalities available to the client.
* The documentation should have a good and consistent quality.

1. **Objectives**

**2.1 Main Objectives**

The main objective of the project is to come up with a solution of an implementation of an application for a food delivery management system. The application should be structured in packages using a layered architecture and should use serialization, streams and lambda expresions.

**2.2 Secondary Objectives**

|  |  |  |
| --- | --- | --- |
| Objective | Description | Chapter in the documentation |
| Development of the use cases and different scenarios. | Use of behavioral diagrams in order to describe the goals of the application, the users and other systems that interact with the developed program. This part explain the best how the application has to work. Also, here they are presented several particular cases when interacting with the program (that can ruin it if not handled properly). | 3 |
| Choosing the data structures | Use appropriate data to store date types: String types and int types, Map (abstract), HashMap, HashSet. | 4 |
| Class distribution | Every object in the system will form a class. Moreover, there will be needed the main class, a UI class (a View class) and also a class (a Controller class) that will tell the UI class how to interact with the classes of objects (the Models classes). | 4 |
| Development of the algorithms | Each operation (the insertion of an object – set of products/ product in the serialized files, the deletion of an object from the serialized files, the update of an object in the serialized files, the Filering of the products) consits of a certain algorithm that has to be properly developed, in order to reach efficiency and also not to get unexpected errors (bugs). | 4 |
| Solution Implementation | In order to implement the solution, it is required to declare the classes, the attributes for each class, the data structures. The next step is to implement the methods in every class. Using the methods, the algorithms will be implemented, translated from the pseudocode written when developing them. | 5 |
| Testing | It is required to test every single case in order not to omit any particular one. This is the part where the bugs are fixed. For some logs of events (varying the data input), it is good to test at least 3 different scenarios and see the result. When a result is not the correct one or an error occurs, the code will be debugged on that exact scenario (reviewing it from the class where the result is displayed on the GUI to the class where the main methods are implemented). | 6 |

1. **Analyzing the Problem**

When analysing the problem, use cases for every operation on the queues will be provided. For example, an analysis of the operation of adding a composite product is discribed below.

Use case name: Add a composite product based on the input from the user.

Actors:

* <Finding the item in the list of products> system (based on the input “Title”, added by the user - administrator in the GUI, the program will search for the product with that exact unique name - Title);
* <Adding the item to the product to composite list> system (deserialize the itemsToComposite list, add this item and serialize if back modified);
* <Composite the items in the list> system (when the user - administrator triggers the Composite button, all the items in the toComposite list are added to be one single object of type product/ MenuItem);
* <Add the composited item into the products list> system (deserialize the list of items, add the new item – the composited one and serialize it back);

Triggers:

* The buttons “add item to composite” and “composite”.

Pre-conditions:

* The title field has to be introduced into the GUI, before the adding of the item.

Post-conditions:

* The items list will be updated.

Normal Flow:

* The system will read the input from the GUI, when the user clicks the “add to composite” button.
* The <Finding the item in the list of products> system will check if the title field is filled.
* Then, it will deserealize the item list.
* It will search for an item with that title in the list.
* The <Adding the item to the product to composite list> system will deserialize the toComposite items list.
* It will add the new item.
* Eventually, it will serialize the list back updated.
* The steps above will repeat untill the user clicks the “composite” button.
* The <Composite the items in the list> system will be notified and will deserialize the items toComposite list.
* It will compute the fields of the new composited item (based on the items of the list – the concatenation/ average/ sum of the fields).
* The <Add the composited item into the products list> system will deserialize the actual item list.
* Then, it will add the new composited item to the list and serialize it back.

Alternate Flow:

* From 4: The system will not find any item with that title in the list. Therefore, it will throw an exception and return to step 1.
* From 9: The system will find out that the toComposite list is null and will throw an exeption. Then, it will return to step 1.

1. **Projection**

**4.1 Data Structures**

There will be used some Data Structures in order to store and to serialize/ deserialize the data efficiently. The used structures are:

HashSet<Object>, HashSet<String>, Map<Object, HashSet<Object>>, Hashmap<String, String>,

HashMap<String, Integer>

**4.2 Main Class Diagrams**

|  |
| --- |
| Client |
| ~clients : HashMap<String, String>  ~panel : ClientGUI |
| + actionPerformed(e: ActionEvent) : void  + addInOrder (menuItem: MenuItem, username : String) : void  + modifyItems(menuItem: MenuItem) : void  + findByTitle(items : HashSet<MenuItem>, title : String) : MenuItem  + showTableOnFilter() : void  + populateTableOnFilter(items : HashSet<MenuItem>) : String[][]  + populateTable(items : HashSet<MenuItem>) : String[][]  + assignValuesToRow(clientData : String[][], index : int, menuItem) : void  + Client(panel : ClientGUI) |

|  |
| --- |
| Employee |
| ~ serializer : Serializer  ~ employees : HashMap<String, String>  ~ panel : EmployeeGUI |
| **+** actionPerformed(e: ActionEvent) : void  **+** Employee (panel : Employee GUI)  **+** populateTable(clientOrdering : HashMap<String, Order>) : String[][] |

|  |
| --- |
| Administrator |
| ~ serializer : Serializer  ~ administrator : HashMap<String, String>  ~ panel : AdministratorGUI |
| + actionPerformed(e: ActionEvent) : void  + Administrator (panel : AdministratorGUI)  + compositeItems( items : HashSet<MenuItem>) : MenuItems  + findrByTitle ( items : HashSet<MenuItem>, title : String) : MenuItems  + filterByProducts( date : Date) : HashMap<MenuItem, Integer>  + filterClients( oftenProducts : int, orderValue : int) : : HashSet<String>  + getKey( clientsOrdering : HashMap<String, Order>, order : Order) : String  + computeBill(menuItems : HashSet<MenuItems>) : int  + populateTable(items : HashSet<MenuItem>) : String[][]  + assignValuesToRow(clientData : String[][], index : int, menuItem : MenuItems) : void |

|  |
| --- |
| <<utility>> Serializer |
|  |
| + deserializeUsername(filename : String) : String  + serializeUsername(username : String, filename : String) : void  + deserializeClientToOrder(filename : String) : HashMap<String, Order>  + serializeClientToOrder (ClientToOrder: : HashMap<String, Order>, filename : String) : void  + deserializeOrders(filename : String) : Map< Order, HashSet<MenuItem>>  + serializeOrders(orders: Map< Order, HashSet<MenuItem>>, filename : String) : void  + deserializeItems(filename : String) : HashSet<MenuItem>  + serializeItems(items : HashSet<MenuItem>, filename : String) : void  +deserializeHash(filename : String) : HashMap<String, String>  +serializeHash(map : HashMap<String, String> , filename : String) : void |

|  |
| --- |
| ClientGUI |
| + clientLabel : JLabel  + addItemText : JTextField  + priceText : JTextField  + sodiumText : JTextField  + fatText : JTextField  + proteinText : JTextField  + caloriesText : JTextField  + ratingText : JTextField  + titleText : JTextField  + passwordText : JTextField  + usernameText : JTextField  + addItem : JButton  + placeOrder : JButton  + filter : JButton  + signUp : JButton  + logIn : JButton  + clientAccessOrderPanel : JPanel  + clientAccessFilterPanel : JPanel  + clientAccessPanel : JPanel  + clientFirstPanel : JPanel  + clientMainPanel : JPanel  + clientAccessFrame : JFrame  + clientFrame : JFrame |
| + showClientPanel() : void  + showClientRegistrationPanel() : void  - createSimpleButton(text : String) : JButton  - GUI() : void  + ClientGUI() |

|  |
| --- |
| EmployeeGUI |
| + employeeLabel : JLabel  + passwordText : JTextField  + usernameText : JTextField  + signUp : JButton  + logIn : JButton  + employeeAccessPanel : JPanel  + employeeFirstPanel : JPanel  + employeeMainPanel : JPanel  + employeeAccessFrame : JFrame  + employeeFrame : JFrame |
| + showEmployeePanel() : void  + showEmployeeRegistrationPanel() : void  - createSimpleButton(text : String) : JButton  - GUI() : void  + EmployeeGUI() |

|  |
| --- |
| AdministratorGUI |
| + adminLabel : JLabel  + day : JTextField  + highOrders : JTextField  + oftenProducts : JTextField  + endHour : JTextField  + startHour : JTextField  + productPriceText : JTextField  + productSodiumText : JTextField  + productFatText : JTextField  + productProteinText : JTextField  + productCaloriesText : JTextField  + productRatingText : JTextField  + productTitleText : JTextField  + passwordText : JTextField  + usernameText : JTextField  + compositeProducts : JButton  + addProductToComposite : JButton  + forthReport : JButton  + thirdReport : JButton  + secondReport : JButton  + firstReport : JButton  + viewProducts : JButton  + deleteProducts : JButton  + editProducts : JButton  + addProducts : JButton  + signUp : JButton  + logIn : JButton  + adminAccessReportsPanel : JPanel  + adminAccessTablePanel : JPanel  + adminAccessPanel : JPanel  + adminFirstPanel : JPanel  + adminMainPanel : JPanel  + adminAccessFrame : JFrame  + adminFrame : JFrame |
| + showAdministratorPanel() : void  + showAdministratorRegistrationPanel() : void  - createSimpleButton(text : String) : JButton  - GUI() : void  + AdministratorGUI() |

|  |
| --- |
| Order |
| - orderDate : Date  - clientID : int  - orderID : int |
| + hashCode() : int  + equals(object : Object) : Boolean  + setOrderDate(orderDate : Date) : void  + getOrderDate() : Date  + setClientID( clientID : int) : void  + getClientID() : int  + setOrderID(orderID : int) : void  + getOrderID() : int  +Order(orderID : int, vlientID : int, orderDate: Date) |

|  |
| --- |
| MenuItem |
| # nrOfOrders : int  # price : int  # sodium : int  # fat : int  # protein : int  # calories : int  # rating : float  # title : String  # productID : int |
| + setNrOfOrders( nrOfOrders : int) : void  + getNrOfOrders() : int  + setPrice (price : int) : void  + getPrice () : int  + setSodium (sodium: int) : void  + getSodium () : int  + setFat (fat: int) : void  + getFat () : int  + setProtein (protein: int) : void  + getProtein () : int  + setCalories (calories : int) : void  + getCalories () : int  + setRating (rating: float) : void  + getRating () : float  + setTitle (title: String) : void  + getTitle () : String  + setProductID (productID: int) : void  + getProductID () : int |

|  |
| --- |
| <<utility>> CompositeProduct |
|  |
| + compositePrice(products : HashSet <BaseProducts>) : int |

|  |
| --- |
| <<utility>> BaseProduct |
|  |
| + compositePrice() : int  + BaseProduct(title : String, rating : float, calories : int, fat : int, sodium : int, price : int, nrOfOrders : int) |

|  |
| --- |
| DeliveryService |
| ~items : HashSet<MenuItem>  ~orders : Map<Order, HashSet<MenuItem>> |
| + getNewItems() : void  + get(Order : Object) : HashSet<MenuItem>  + DeliveryService(items : HashSet<MenuItem>) |

Class relationships:

Diagram

Description automatically generated

**4.3 Algorithms**

The most complex algorithm of the project are those applied when handling the input from the user and diplaying the result on the GUI. All of them basically consist of deserialising data from file and serialize it back if changed then applying one of the operations (add, update, delete, show all, add composite, add order, depending on the type of the user) on the retrieved object / objects.

1. **Implementation**

Start.java class. Here is the main method that starts the programs running.

EmployeeGUI.java class. The is a user interface class that manages the log in window for a Regular Employee and the window for which the have acces.

Atributes:

* JFrame: employeeFrame, employeeAccessFrame;
* JPanel employeeMainPanel, employeeFirstPanel, employeeAccessPanel;
* JButton logIn, signUp;
* JTextField usernameText, passwordText;
* JLabel employeeLabel;

ClientGUI.java class. The is a user interface class that manages the log in window for a Client and the window for which the have acces.

Atributes:

* + JFrame clientFrame, clientAccessFrame;
  + JPanel clientMainPanel, clientFirstPanel, clientAccessPanel, clientAccessFilerPanel, clientAccessOrderPanel;
  + JButton logIn, signUp, filter, placeOrder, addItem;
  + JTextField usernameText, passwordText, titleText, ratingText, caloriesText, proteinText, fatText, sodiumText, priceText, addItemText;
  + JLabel clientLabel;

AdministratorGUI.java class. The is a user interface class that manages the log in window for an Administrator and window the for which the have acces.

Atributes:

* JFrame adminFrame, adminAccessFrame;
* JPanel adminMainPanel, adminFirstPanel, adminAccessPanel, adminAccessTablePanel, adminAccessReportsPanel;
* JButton logIn, signUp, addProduct, editProduct, deleteProduct, viewProducts, firstReport, secondReport, thirdReport, forthReport, addProductToComposite, compositeProducts;
* JTextField usernameText, passwordText, productTitleText, productRatingText, productCaloriesText, productProteinText, productFatText, productSodiumText, productPriceText, startHour, endHour, oftenProducts, highOrders, day;
* JLabel adminLabel;

Employee.java class, Client.java class, Administrator.java class. These are controller classes that create the link between GUI classes and the class where all the operations take place: DeliverySevice class.

Order.java class. The class that represent the object of an order.

Atributes:

* int orderID: a unique identifier for the order;
* int clientID: : a unique identifier for the client ordering this order;
* Date orderDate: the current date and Time when the order was made;

Important methods:

These 2 metods override the Map methods (HashMap) in order to work right for the Order object.

@Override

public boolean equals(Object object) {

if (object == null) return false;

if (this == object) return true;

Order order = (Order) object;

return orderID == order.orderID;

}

@Override

public int hashCode() {

int value = 13;

value = 37 \* value + orderID;

value = 37 \* value + clientID;

if (orderDate != null) {

value = 37 \* value + orderDate.hashCode();

}

return value;

}

MenuItem.java class. Represent the object of product (base or composite) availabe to the clients.

Atributes:

* int productID;
* String title;
* float rating;
* int calories;
* int protein;
* int fat;
* int sodium;
* int price;
* int nrOfOrders: how many times this product has been ordered;

IDeliveryServiceProccesing.java interface. Contains the headers of the operation method from the next class.

DeliveryService.java class. Holds the essential methods of the program.

Important methods:

* adds a new product (by the administrator) to the item list:

public void addProduct(String title, float rating, int calories, int protein, int fat, int sodium, int price) {

Serializer serializerItems = new Serializer();

HashSet<MenuItem> items;

items = serializerItems.deserializeItems("serializeItems.txt");

items.add(new BaseProduct(title, rating, calories, protein, fat, sodium, price, 0));

serializerItems.serializeItems(items, "serializeItems.txt");

}

* deletes a product (by the administrator) from the item list:

public void deleteProduct(String title) {

Serializer serializerItems = new Serializer();

HashSet<MenuItem> items;

items = serializerItems.deserializeItems("serializeItems.txt");

MenuItem item = null;

for (MenuItem menuItem : items) {

if (menuItem.getTitle().equals(title)) {

item = menuItem;

break;

}

}

items.remove(item);

serializerItems.serializeItems(items, "serializeItems.txt");

}

* add a product to the composite list(by the administrator):

public void addProductToComposite(String title) {

HashSet<MenuItem> items;

HashSet<MenuItem> menuItems;

Serializer serializer = new Serializer();

items = serializer.deserializeItems("serializeComposite.txt");

menuItems = serializer.deserializeItems("serializeItems.txt");

MenuItem menuItem = findByTitle(menuItems, title);

items.add(menuItem);

serializer.serializeItems(items, "serializeComposite.txt");

* composite and adds a the new product to the item list(by the administrator):

­­

public void compositeProducts() {

Serializer serializer = new Serializer();

HashSet<MenuItem> itemsToComposite;

itemsToComposite = serializer.deserializeItems("serializeComposite.txt");

MenuItem menuItem = compositeItems(itemsToComposite);

Serializer serializerItems = new Serializer();

HashSet<MenuItem> items;

items = serializerItems.deserializeItems("serializeItems.txt");

items.add(menuItem);

serializerItems.serializeItems(items, "serializeItems.txt");

}

* adds a product to the toOrder item list(by the client):

public void addItemByClient(String title) {

Serializer serializerItems = new Serializer();

HashSet<MenuItem> items;

items = serializerItems.deserializeItems("serializeItems.txt");

MenuItem menuItem = findByTitle(items, title);

Serializer serializerUsername = new Serializer();

String currentUsername = serializerUsername.deserializeUsername("serializeUsername.txt");

addInOrder(menuItem, currentUsername);

}

* make an Order based on the list of toOrder products and creates a bill (by the client):

public void placeOrder() {

FIleWriter billFileWriter = new FIleWriter();

Serializer serializerClientOrdering = new Serializer();

HashMap<String, Order> clientOrdering;

clientOrdering = serializerClientOrdering.deserializeClientToOrder("serializeClientOrdering.txt");

Serializer serializerAddNewOrder = new Serializer();

Map<Order, HashSet<MenuItem>> orders;

orders = serializerAddNewOrder.deserializeOrders("serializeOrders.txt");

Serializer serializerUsername = new Serializer();

String currentUsername = serializerUsername.deserializeUsername("serializeUsername.txt");

HashSet<MenuItem> menuItems = orders.get(clientOrdering.get(currentUsername));

billFileWriter.writeToFile(menuItems);

}

* load the initial set of items from the csv file:

public void getNewItems() {

Path path = Paths.get("C:\\Users\\Marius\\Desktop\\Products.csv");

if (Files.exists(path)) {

items = new HashSet<>();

try(Stream<String> stream = Files.lines(path)) {

stream

.skip(1)

.limit(10)

.distinct()

.forEach(line -> {

String[] currentLine = line.split(",");

String title;

int calories, protein, fat, sodium, price;

title = currentLine[0];

try {

float rating = Float.parseFloat(currentLine[1]);

calories = Integer.parseInt(currentLine[2]);

protein = Integer.parseInt(currentLine[3]);

fat = Integer.parseInt(currentLine[4]);

sodium = Integer.parseInt(currentLine[5]);

price = Integer.parseInt(currentLine[6]);

items.add(new BaseProduct(title, rating, calories, protein, fat, sodium, price, 0));

} catch(NumberFormatException e){

e.printStackTrace();

}

});

Serializer serializer = new Serializer();

serializer.serializeItems(items, "serializeItems.txt");

} catch (IOException e) {

e.printStackTrace();

}

} else {

System.out.println("File does not exist.");

}

BaseProduct.java class. Object from the csv file, of type MenuItem.

Compositeproduct.java class. Object composited from multiple objects from the csv file, also of type MenuType

Important methods:

public int computePrice(HashSet<BaseProduct> products) {

int price = 0;

for (BaseProduct product : products) {

price += product.computePrice();

}

return price;

}

1. **Testing**

The main (log in) windows:

Graphical user interface, text, application, Teams

Description automatically generated

The access windows for each type of user:

**Graphical user interface

Description automatically generated**

1. **Conclusion and future developments**

The algorithms used in the program code are quite simple to implement. The more compicated/ advanced part was working with streams and with HashTables. Also, handling the serialization tehnique was a little challanging. Nevertheless, building the GUI with more windows was a little time consuming. While testing and debugging (on particular cases), there were a lot of minor flaws that had to be fixed. Overall, I loved the fact that I had to work on an application that has huge possible implementation in the real life. I really feel that I gained a lot of knowledge about the coding in java using Serialization, Streams and HashMaps after finishing this project.

As future developments, the simulator might also be able to notify the Employee instantly when an order is up. Also, to track the delivery guys while transporting the food. Moreover, a more advanced GUI could be implemented.

1. **Bibliography**

**https://www.baeldung.com/java-serialization**

**https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/**

**https://www.geeksforgeeks.org/traverse-through-a-hashmap-in-java/**

**https://stackoverflow.com/questions/8198168/how-to-get-hours-from-a-date-using-java**

**http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html**

**https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html**