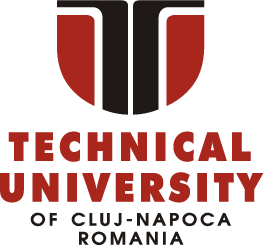
Technical University of Cluj-Napoca

Programming Techniques

Laboratory – Assignment 4

Food Delivery Management System

**

Teacher: prof. Ioan Salomie

Teacher Assistant: Cristina Bianca Pop

Student: Fleser Mihai

Group: E\_30424

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:

1. 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.
2. the products ordered more than a specified number of times so far.
3. the clients that have ordered more than a specified number of times and the value of the order was higher than a specified amount.
4. 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.

Secondary objectives:

* Analyze the problem and identify requirements ………………………...………………. Chapter 2
* Design the Food Delivery Management System App …..………………………………. Chapter 3
* Implement the Food Delivery Management System App …………………….………….. Chapter 4
* Test the Food Delivery Management System App ……………………………..……….. Chapter 5

1. Problem analysis, modeling, scenarios, use cases
   1. *Analyzing the problem*

PROBLEM: Design an application to simulate a Food Delivery Management System

Furthermore, we must consider the following:

1) Define the interface IDeliveryServiceProcessing containing the main operations that can be executed by the administrator and client, as follows:

• Administrator: import products, manage the products from the menu, generate reports

• Client: create new order which implies computing the price for an order and generating a bill in .txt format, searching for products based on several criteria.

2) Define and implement the classes from the class diagram shown above:

• 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.

3) Implement the class DeliveryService using a predefined JCF collection which uses a hashtable data structure. The hashtable key will be generated based on the class Order, which can have associated several MenuItems.

• Define a structure of type Map<Order, Collection<MenuItem>> for storing the order related information in the DeliveryService class. The key of the Map will be formed of objects of type Order, for which the hashCode() method will be overwritten to compute the hash value within the Map from the attributes of the Order (OrderID, date, etc.).

• Define a structure of type Collection<MenuItem> which will save the menu (i.e. all the products) provided by the catering company. Choose the appropriate collection type for your implementation.

• Define a method of type “well formed” for the class DeliveryService.

• Implement the class using Design by Contract method (involving pre, post conditions, invariants, and assertions).

4) The base products used initially for populating the DeliveryService object can be loaded from the products.csv file (adapted from Link) using lambda expressions and stream processing. Note: the administrator can manually add other base products as well.

5) 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. *Modelling the problem*

There are three types of users in the application: the client, the administrator and the employee. Each type of user will have a separate GUI in which he will work. The application will be organized in Layers, the main ones being the Presentation Layer, The Business Layer and the Data Layer.

The Administrator has control over the whole system, meaning that he will be able to modify, add, delete, compose products and also to generate reports. If the administrator closes the application, then all other users won’t be able to access it.

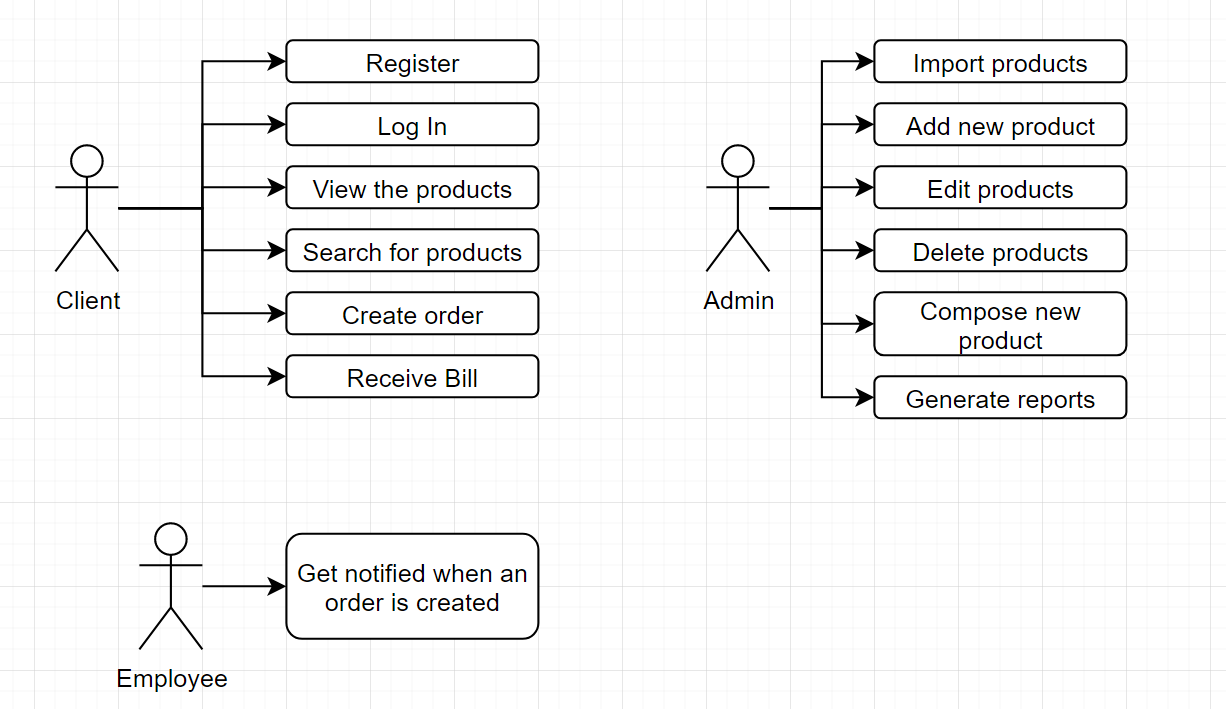
The client has control only over what he wants to order. He can search products, add them to his cart, remove them from the cart and finally perform the order. He will need an account to access the system and he can register or log in from his GUI.

The employee does not need an account to access the system, but he can only see when certain products were ordered and by who.

* 1. *Different scenarios and use cases*

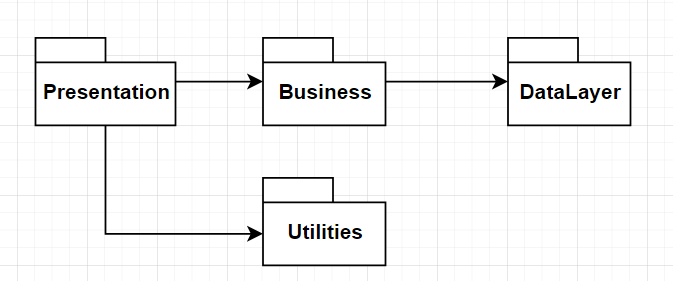
Use cases are a technique for capturing, modelling and specifying the requirements of a system. A use case corresponds to a set of behaviors that the system may perform in interaction with its actors, and which produces an observable result that contribute to its goals. Actors represent the role that human users or other systems have in the interaction.

As I have said before, there are 3 types of users in my application: the admin, the client and the employee. Each one of them have a dedicated GUI that lets them perform specific actions. I will detail these actions in the use case diagram below:

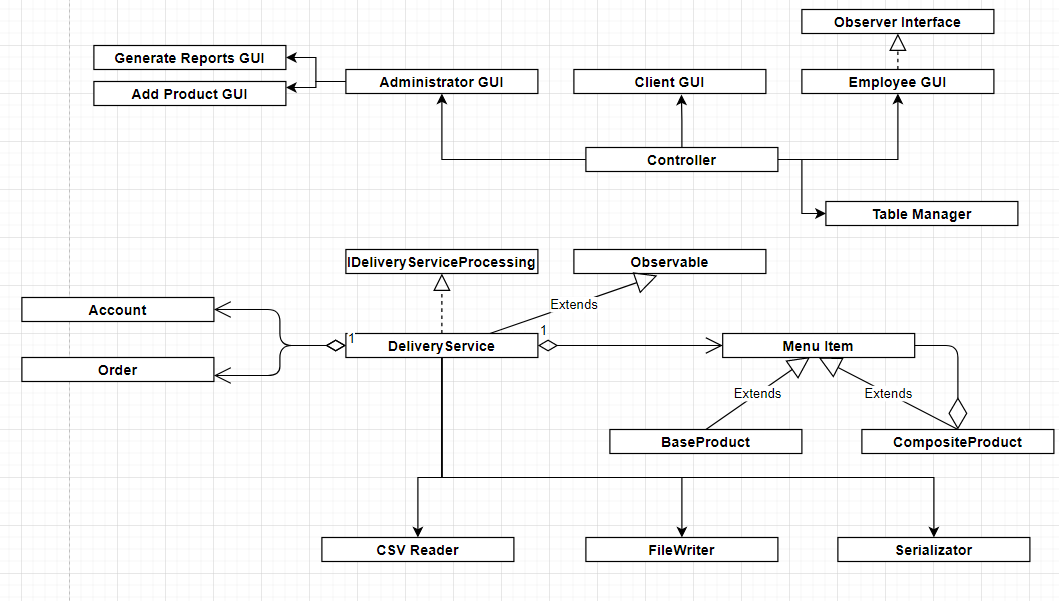


The use case diagram presents all the actors that interact with the application.

* 1. Design
     + 1. *UML package diagram*



* + - 1. *UML class Diagram*



* + - 1. *Data Structures*

For my project, I have used the following data Structures: ArrayList, String and HashMap.

I decided to use ArrayList instead of classic arrays because ArrayList does not have a fixed length as normal arrays do, thus making them more memory efficient. Also, I do not have to worry about exceeding the length of the array.

I used String to store the some of input from the user in the model Classes.

I used hashMap to store the order related information in the DeliverySercice class. The key of the map is formed by objects of type Order, for which the hashCode() method is overwritten.

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

* + - 1. *Design Decisions and Packages*

I have done my project using a Layered Arhitecture.

In a standard enterprise application which has a database and graphical UI (web or desktop), there are some typical layers which constitute the application:

* Presentation Layer: Every enterprise application has a UI, in fact graphical UI. This UI can be web or desktop based, which doesn't matter. The rule is simple. UI takes user action and sends it to the controller. And at the end it shows result taken from controller to the user. UI can be implemented according to MVC, MVP, MVVM or another approach.
* Business Layer: Handles business logic of the application. Takes info from user and sends it to DB layer (DAO or ORM framework) and vice versa. Abstraction type of controller may vary (a separate control and business layers for example)  according to the application parameters or development patterns (MVC, MVP, MVVM, ...) but the main idea remains the same.
* Data Layer: Database handling layer of the application. It may contain entity definitions, ORM framework or DB connection codes having SQL sentences, according to the abstraction decision. Its role is getting data from controller, performing data operation on database and sending results again to controller (if result exists). Database independence is a very important plus for this layer, which brings flexibility.Model Layer:
* Utilities Packages: Contains a class that the controller uses to update the information from the tables on the GUI.
  + - 1. *Class Design*

As I said earlier, my program consists of 4 packages, following the Layered Architecture design pattern:

The Presentation Package:

* AdminGUI – it’s the class used by the administrator to control the application. From here he can manage the list of products and he can compose new products.
* AddProductGUI – it is used by the Administrator to insert a new product in the database.
* GenerateReportsGUI – it is also used by the admin to generate the specific reports he needs.
* ClientGUI – it’s the class used by the client to make new orders. He can see all the products from the menu, he can search for a specific product, he can add or remove products from the cart and finally he can make a new order. Before making orders, he has to log in first. He can also use this GUI to register and log in into the application.
* EmployeeGUI – it’s the class used by the employee to receive a notification each time a new order has been created.
* Controller - This class communicates between the GUIs and Bussiness Logic. It handles the events that happens on the GUI. It takes the data introduced by the users and transfers it further to the Bussiness Package to manage it.

The Utilities Package:

* TableManager – a class used by the Controller to update the tables from the GUIs.

Business Package:

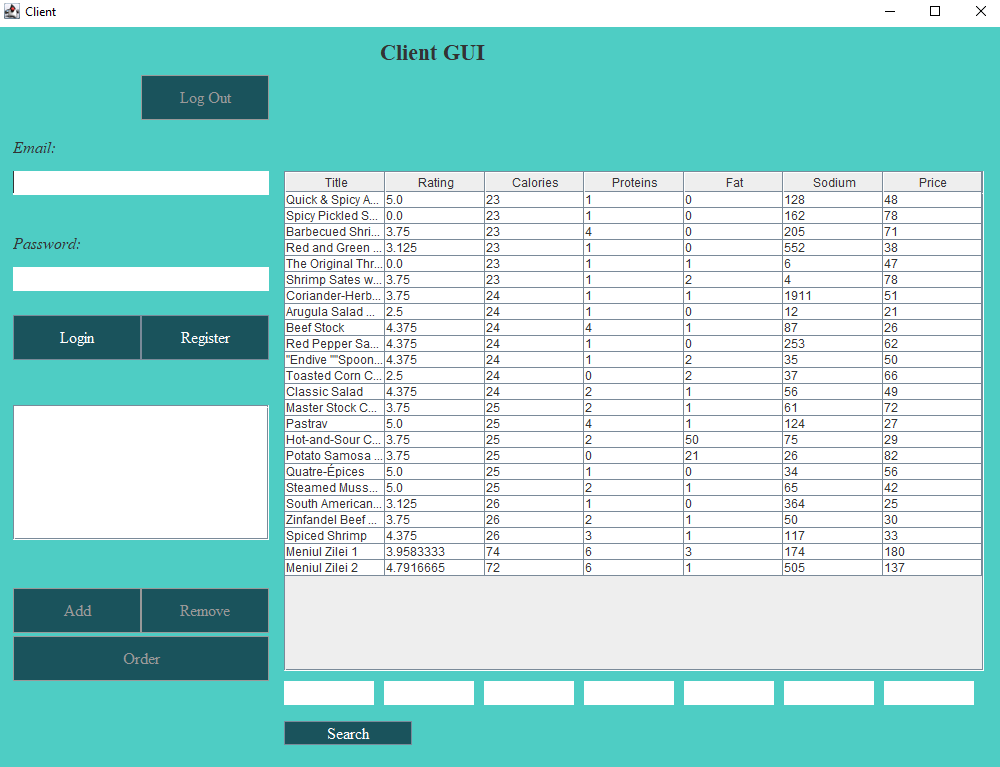
* Account – class used to store information about the accounts of the clients.
* MenuItem – abstract class used to store information about the items from the menu.
* BaseProduct – extends menuItem - class used when importing the product from the CSV file.
* CompositeProduct – extends MenuItem – class used to compose a new manu item from other existing items.
* Order – class used to store information about the orders done by the clients.
* IDeliveryServiceProcessing – interface containing the main operation that can be executed by the administrator and client
* DeliveryService – implements IDeliveryServiceProcessing – used to implement by contract the methods declared in the interface.

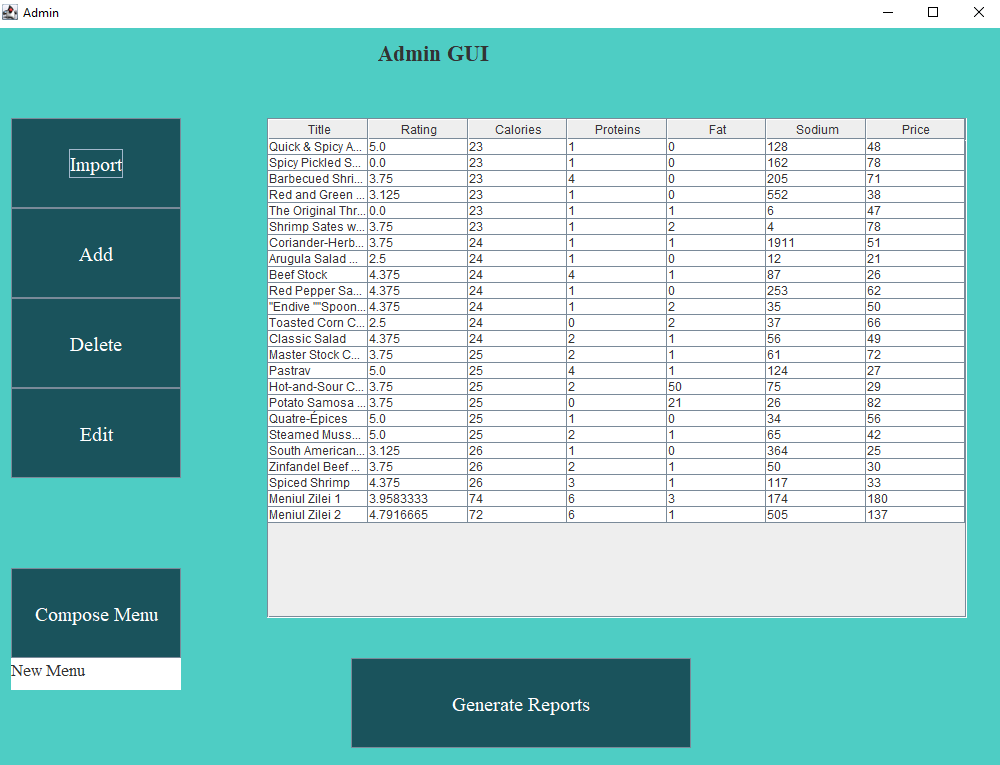
Data Layer Package:

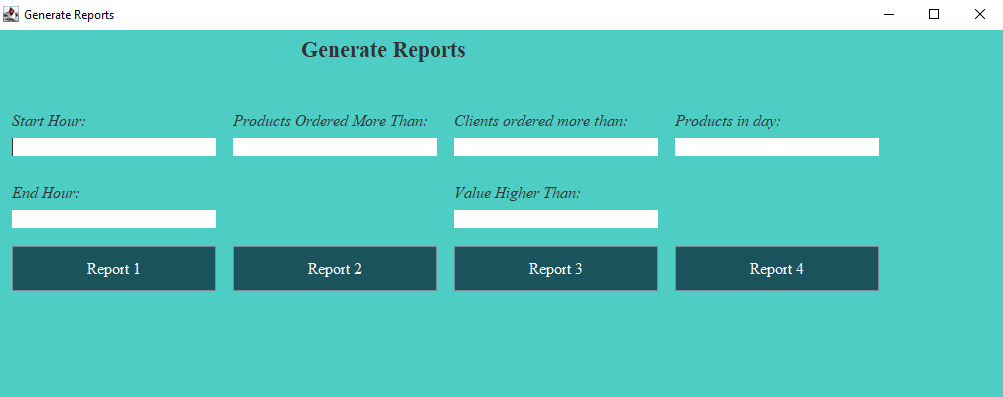
* CSVReader – class used to import the initial set of products from the CSV file.
* Serializator – class used to serialize and deserialize the information from the DeliveryService class when the application starts or ends. It is used to save the information for the next time the application opens.
* FileWriterClass – class used to write the reports or the bills in certain files.

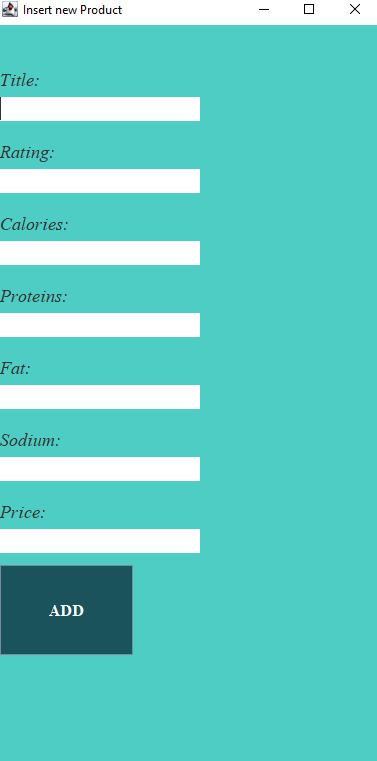
App class – the entry point of the application, just creates a new Controller.

* + - 1. *User Interface*











* 1. Implementation

/\*\*

\* Interface for the operations that the client and the administrator can do in the application

\*/

public interface IDeliveryServiceProcessing {

/\*\*

\* Imports a set of products from a file, namely products.csv

\*/

void importProducts();

/\*\*

\* Adds a new product to the items from the menu.

\* @pre NewItem has to be not null

\* @param newItem The new item that will pe introduced

\*/

void addProduct(MenuItem newItem);

/\*\*

\* @pre The id must exist

\* @post The new item exists in the database

\* @param id

\* @param title

\* @param rating

\* @param calories

\* @param proteins

\* @param fat

\* @param sodium

\* @param price

\*/

void editProduct(Integer id, String title,Float rating, Integer calories, Integer proteins, Integer fat, Integer sodium, Integer price);

/\*\*

\* Deletes a product using it's title

\* @pre item with title exists in the database

\* @param title

\*/

void deleteProduct(String title);

/\*\*

\* Adds a new account into the database.

\* @pre account not null

\* @post the new account exists in the database

\* @param account

\*/

void addAccount(Account account);

/\*\*Log in with the client

\* @pre the account is not null

\* @post the logged in account updates

\* @param account

\*/

void logIn(Account account);

/\*\*

\* Logs out from the application

\* @post the logged in account updates

\*/

void logOut();

ArrayList<MenuItem> searchForTitle(ArrayList<MenuItem> items, String title);

ArrayList<MenuItem> searchForRating(ArrayList<MenuItem> items, Float rating);

ArrayList<MenuItem> searchForCalories(ArrayList<MenuItem> items, Integer calories);

ArrayList<MenuItem> searchForProteins(ArrayList<MenuItem> items, Integer proteins);

ArrayList<MenuItem> searchForFat(ArrayList<MenuItem> items, Integer fat);

ArrayList<MenuItem> searchForSodium(ArrayList<MenuItem> items, Integer sodium);

ArrayList<MenuItem> searchForPrice(ArrayList<MenuItem> items, Integer price);

/\*\*

\* Generates a report with the orders performed between a start hour and an end our regardless the date

\* @param startHour

\* @param endHour

\*/

void generateReport1(Integer startHour, Integer endHour);

/\*\*

\* Generates a report with the products ordered more than a specified number of times so far

\* @param minNumber

\*/

void generateReport2(Integer minNumber);

/\*\*

\* Generates a report with 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 minClients

\* @param valueHigher

\*/

void generateReport3(Integer minClients, Integer valueHigher);

/\*\*

\* Generates a report with the products ordered within a specified day with the number of times they have been ordered.

\* @param day

\*/

void generateReport4(Integer day);

/\*\*

\* Client creates a new order.

\*/

void createOrder();

}

public class DeliveryService extends Observable implements IDeliveryServiceProcessing{

private ArrayList<MenuItem> menuItems;

private ArrayList<Account> accounts;

private Account loggedInAccount;

private ArrayList<MenuItem> currentItemsInOrder;

private HashMap<Order, ArrayList<MenuItem>> allOrders;

public DeliveryService()

{

menuItems = new ArrayList<>();

accounts = new ArrayList<>();

currentItemsInOrder = new ArrayList<>();

allOrders = new HashMap<>();

loggedInAccount = null;

}

@Override

public void createOrder() {

assert checkNotNull();

Integer price = 0;

for(MenuItem item: currentItemsInOrder)

price += item.getPrice();

Order order = new Order(allOrders.size(),loggedInAccount.getEmail(), (GregorianCalendar) Calendar.getInstance(), price);

FileWriterClass.writeOrder(order,currentItemsInOrder);

setChanged();

notifyObservers(FileWriterClass.orderToString(order, currentItemsInOrder));

allOrders.put(order,new ArrayList<>(currentItemsInOrder));

currentItemsInOrder.clear();

}

public void addItemInOrder(MenuItem item)

{

assert checkNotNull();

currentItemsInOrder.add(item);

}

public void removeItemFromOrder()

{

assert checkNotNull();

currentItemsInOrder.remove(currentItemsInOrder.size() - 1);

}

public String itemsFromOrderToString()

{

assert checkNotNull();

String output = "";

for(MenuItem item: currentItemsInOrder)

{

output = output + item.getTitle() + "\n";

}

return output;

}

@Override

public void importProducts() {

assert checkNotNull();

menuItems = CSVReader.readData();

}

public ArrayList<MenuItem> getMenuItems() {

assert checkNotNull();

return menuItems;

}

public void importSerialisedProducts()

{

assert checkNotNull();

this.menuItems = new Serializator<MenuItem, Object>().deserialize("Products");

}

public void importSerialisedAccounts()

{

assert checkNotNull();

this.accounts = new Serializator<Account, Object>().deserialize("Accounts");

}

public void importSerialisedOrders()

{

assert checkNotNull();

this.allOrders = new Serializator<Order, MenuItem>().deserializeHashMap("Orders");

}

public void exportSerialisedOrders()

{

assert checkNotNull();

new Serializator<Order, MenuItem>().serializeHashMap(allOrders, "Orders");

}

public void exportAccounts()

{

assert checkNotNull();

new Serializator<Account, Object>().serialize(accounts,"Accounts");

}

public void exportProducts()

{

assert checkNotNull();

new Serializator<MenuItem, Object>().serialize(menuItems,"Products");

}

@Override

public void addProduct(MenuItem newItem) {

assert checkNotNull();

assert newItem != null;

menuItems.add(newItem);

assert getItemWithTitle(newItem.getTitle()) != null;

}

@Override

public void editProduct(Integer id ,String title, Float rating, Integer calories, Integer proteins, Integer fat, Integer sodium, Integer price) {

assert checkNotNull();

assert id < menuItems.size();

MenuItem item = menuItems.get(id);

item.setTitle(title);

item.setRating(rating);

item.setCalories(calories);

item.setProteins(proteins);

item.setFat(fat);

item.setSodium(sodium);

item.setPrice(price);

}

public MenuItem getItemWithTitle(String title)

{

assert checkNotNull();

for(MenuItem item: menuItems)

{

if(item.getTitle().equals(title))

return item;

}

return null;

}

@Override

public void deleteProduct(String title) {

assert checkNotNull();

assert getItemWithTitle(title) != null;

menuItems.remove(getItemWithTitle(title));

}

@Override

public void addAccount(Account newAccount) {

assert checkNotNull();

assert newAccount != null;

accounts.add(newAccount);

assert getAccountWithEmail(newAccount.getEmail()) != null;

}

@Override

public void logIn(Account account) {

assert checkNotNull();

assert account!= null;

loggedInAccount = account;

assert loggedInAccount != null;

}

public Account getLoggedInAccount()

{

assert checkNotNull();

return loggedInAccount;

}

public ArrayList<Account> getAccounts() {

assert checkNotNull();

return accounts;

}

public Account getAccountWithEmail(String email)

{

assert checkNotNull();

for(Account account : accounts)

{

if(account.getEmail().equals(email))

return account;

}

return null;

}

@Override

public void logOut() {

assert checkNotNull();

loggedInAccount = null;

assert loggedInAccount == null;

}

@Override

public ArrayList<MenuItem> searchForTitle(ArrayList<MenuItem> items, String title) {

assert checkNotNull();

return (ArrayList<MenuItem>) items.stream().filter(p -> p.getTitle().toLowerCase().contains(title.toLowerCase())).collect(Collectors.toList());

}

@Override

public ArrayList<MenuItem> searchForRating(ArrayList<MenuItem> items, Float rating) {

assert checkNotNull();

return (ArrayList<MenuItem>) items.stream().filter(p -> p.getRating().equals(rating)).collect(Collectors.toList());

}

@Override

public ArrayList<MenuItem> searchForCalories(ArrayList<MenuItem> items, Integer calories) {

assert checkNotNull();

return (ArrayList<MenuItem>) items.stream().filter(p -> p.getCalories().equals(calories)).collect(Collectors.toList());

}

@Override

public ArrayList<MenuItem> searchForProteins(ArrayList<MenuItem> items, Integer proteins) {

assert checkNotNull();

return (ArrayList<MenuItem>) items.stream().filter(p -> p.getProteins().equals(proteins)).collect(Collectors.toList());

}

@Override

public ArrayList<MenuItem> searchForFat(ArrayList<MenuItem> items, Integer fat) {

assert checkNotNull();

return (ArrayList<MenuItem>) items.stream().filter(p -> p.getFat().equals(fat)).collect(Collectors.toList());

}

@Override

public ArrayList<MenuItem> searchForSodium(ArrayList<MenuItem> items, Integer sodium) {

assert checkNotNull();

return (ArrayList<MenuItem>) items.stream().filter(p -> p.getSodium().equals(sodium)).collect(Collectors.toList());

}

@Override

public ArrayList<MenuItem> searchForPrice(ArrayList<MenuItem> items, Integer price) {

assert checkNotNull();

return (ArrayList<MenuItem>) items.stream().filter(p -> p.getPrice().equals(price)).collect(Collectors.toList());

}

@Override

public void generateReport1(Integer startHour, Integer endHour) {

assert checkNotNull();

ArrayList<Order>localOrders =(ArrayList) allOrders.keySet().stream().filter(o->o.getDate().get(Calendar.HOUR\_OF\_DAY) <= endHour). collect(Collectors.toList());

localOrders =(ArrayList) localOrders.stream().filter(o->o.getDate().get(Calendar.HOUR\_OF\_DAY) >= startHour). collect(Collectors.toList());

FileWriterClass.writeReport1(localOrders);

}

@Override

public void generateReport2(Integer minNumber)

{

assert checkNotNull();

HashMap<MenuItem, Integer> occurence = new HashMap<>();

HashSet<MenuItem> products = new HashSet<>();

for(Order order: allOrders.keySet())

{

for(MenuItem menuItem: allOrders.get(order))

{

products.add(menuItem);

}

}

for(MenuItem menuItem: products)

{

occurence.put(menuItem, 0);

}

for(Order order: allOrders.keySet())

{

for(MenuItem menuItem: allOrders.get(order))

{

occurence.put(menuItem, occurence.get(menuItem) + 1);

}

}

HashMap<MenuItem, Integer> finalResult = new HashMap<>();

occurence.entrySet().stream().filter(entry->entry.getValue() > minNumber).forEach(entry-> finalResult.put(entry.getKey(), entry.getValue()));

FileWriterClass.writeReport2(finalResult);

}

@Override

public void generateReport3(Integer minClients, Integer valueHigher) {

assert checkNotNull();

HashMap<String, Integer> occurence = new HashMap<>();

HashSet<String> accounts = new HashSet<>();

for(Order order: allOrders.keySet())

{

accounts.add(order.getClientId());

}

for(String account: accounts)

{

occurence.put(account, 0);

}

for(Order order: allOrders.keySet())

{

if(order.getTotalPrice() > valueHigher)

occurence.put(order.getClientId(), occurence.get(order.getClientId()) + 1);

}

HashMap<String, Integer> finalResult = new HashMap<>();

occurence.entrySet().stream().filter(entry->entry.getValue() > minClients).forEach(entry-> finalResult.put(entry.getKey(), entry.getValue()));

FileWriterClass.writeReport3(finalResult);

}

@Override

public void generateReport4(Integer day) {

assert checkNotNull();

HashMap<MenuItem, Integer> occurence = new HashMap<>();

HashSet<MenuItem> products = new HashSet<>();

ArrayList<Order>localOrders =(ArrayList) allOrders.keySet().stream().filter(o->o.getDate().get(Calendar.DAY\_OF\_MONTH) == day). collect(Collectors.toList());

for(Order order: localOrders)

{

for(MenuItem menuItem: allOrders.get(order))

{

products.add(menuItem);

}

}

for(MenuItem menuItem: products)

{

occurence.put(menuItem, 0);

}

for(Order order: localOrders)

{

for(MenuItem menuItem: allOrders.get(order))

{

occurence.put(menuItem, occurence.get(menuItem) + 1);

}

}

FileWriterClass.writeReport4(occurence);

}

/\*\*

\* @inv Invariant - check if the arrayList of menuItems is null.

\* @return

\*/

private boolean checkNotNull()

{

return menuItems != null;

}

}

/\*\*

\* Class used to serialize and deserialize data.

\* @param <T>

\* @param <V>

\*/

public class Serializator <T, V>{

/\*\*

\* Serialises an array

\* @param array

\* @param type

\*/

public void serialize(ArrayList<T> array, String type)

{

try {

FileOutputStream file = new FileOutputStream("ProgramData/" + type + ".txt");

ObjectOutputStream out = new ObjectOutputStream(file);

out.writeObject(array);

out.close();

file.close();

} catch (Exception e)

{

e.printStackTrace();

}

}

/\*\*

\* Deserialises an array

\* @param type

\* @return

\*/

public ArrayList<T> deserialize(String type)

{

ArrayList<T> result = new ArrayList<>();

try

{

FileInputStream file = new FileInputStream("ProgramData/" + type + ".txt");

ObjectInputStream in = new ObjectInputStream (file);

result = (ArrayList<T>) in.readObject (); // Method for deserialization of object

in.close();

file.close();

}

catch (Exception e)

{

e.printStackTrace();

}

return result;

}

/\*\*

\* Deserialises a Hash Map

\* @param type

\* @return

\*/

public HashMap<T,ArrayList<V>> deserializeHashMap(String type)

{

HashMap<T, ArrayList<V>> result = new HashMap<>();

try

{

FileInputStream file = new FileInputStream("ProgramData/" + type + ".txt");

ObjectInputStream in = new ObjectInputStream (file);

result = (HashMap<T, ArrayList<V>>) in.readObject (); // Method for deserialization of object

in.close();

file.close();

}

catch (Exception e)

{

e.printStackTrace();

}

return result;

}

/\*\*

\* Serialises a hash map.

\* @param hashMap

\* @param type

\*/

public void serializeHashMap(HashMap<T, ArrayList<V>> hashMap, String type)

{

try {

FileOutputStream file = new FileOutputStream("ProgramData/" + type + ".txt");

ObjectOutputStream out = new ObjectOutputStream(file);

out.writeObject(hashMap);

out.close();

file.close();

} catch (Exception e)

{

e.printStackTrace();

}

}

}

* 1. Results

I tested this project by introducing, editing or deleting products as the admin. I also tested the reports and the way the admin can compose new products. Furthermore, I registered as a new customer and I have successful manage to order what I wanted to eat. Besides that, the employee GUI notifies the employee each time I made a new order. If I try to introduce wrong data, like letters instead a number, a message will appear letting me know that the information I entered was wrong.

* 1. Conclusions

I liked this Project because it helped me to understand better how to work with design patterns and further develop my OOP knowledge. I have learned, again, that time management is quite important to not be stressed out and do the project in time. I also liked the fact that I have learned how to work with new data structures like HashMap which is a great deal.

Future improvements:

* One idea will be to make accounts for the admin and for the employees.
* Another one will be to let the client manage his account, like changing the password or deleting the account all together.
  1. Bibliography
* Object-Oriented Programming - Lecture Slides of prof. Marius JOLDOS
* Programming Techniques – Lectures of prof. Ioan SALOMIE
* <https://docs.oracle.com/javase/tutorial/uiswing/>
* <https://docs.oracle.com/javafx/2/get_started/jfxpub-get_started.htm>
* <https://www.baeldung.com/javafx>
* <https://www.vogella.com/tutorials/JUnit/article.html>
* <https://stackoverflow.com/>
* <https://www.tutorialspoint.com/java/java_serialization.htm>
* <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>
* https://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html