Fundamental Programming Techniques

Assignment 4- Documentation

<u>FOOD DELIVERY MANAGEMENT SYSTEM</u>

Student: Moisa Oana Miruna

Group: 30422

1. <u>Assignment objective</u>

The objective of this assignment is to develop a Java application that simulates a food delivery management system, which supports 3 kinds of users: administrators, clients, employees. The secondary objectives of this project are: understanding the requirements and how the application is supposed to work, designing the application, implementing the application (both the actual functionality and the graphical user interface). Each of these sub-objectives will be described in detail in the following paragraphs.

2. <u>Problem analysis, modeling, scenarios, use cases</u>

The functional requirements for this assignment are the following:

- The application should support 3 kinds of users: administrators, clients and employees.
- The user should be able to register (create an account), based on an username and a password.
- The user should be able to login into the application with the personal credentials.
- The client should be able to view all menu items form the menu.
- The client should be able to search menu items in the menu based on title/price/calories/protein/sodium/fats.
- The client should be able to select menu items from the menu and place an order.
- The application should generate a bill for the client, after each order.
- The employee should be notified every time a new order is placed.
- The admin should be able to import the initial set of menu items.
- The admin should be able to create new base products and new composite products.
- The admin should be able to remove a menu item.
- The admin should be able to modify an already existing menu item.
- The admin should be able to generate 4 kinds of reports.

The non-functional requirements of this problem are:

- The application should be easy to use, with a friendly interface.
- Except for the UI classes, the rest of the classes should contain at most 300 lines.
- The methods must be at most 30 lines long.
- The application should be composed of the classes presented in the diagram.
- The application should use the Composite Design Pattern, Observer Design Pattern and Design By Contract Method.
- The information from the application should be saved and loaded using serialization.
- Certain methods from the application should be implemented using streams and lambda expressions.

<u>Use cases:</u>

<u>•Use Case 1</u>: register user

Primary Actor: user

Main Success Scenario:

-The user inserts all the required information in the graphical user interface (username, password, type of account).

-The user clicks on the “Register” button.

- A new account is created.

Alternative Sequence 1: Missing fields

-The user doesn’t fill in all the text fields

-The user is warned about the mistake

-The process starts again

Alternative Sequence 2: Duplicate username

-The user inserts a username that already exists.

-The user is warned about the mistake.

-The process starts again.

•Use Case 2: login user

Primary Actor: user

Main Success Scenario:

-The user inserts all the required information in the graphical user interface (username, password).

-The user clicks on the “Login” button.

-The user can visualize his/her account.

Alternative Sequence 1: Missing fields

-The user doesn’t fill in all the text fields

-The user is warned about the mistake

-The process starts again

Alternative Sequence 2: Incorrect username/password

-The user inserts a username that doesn’t exist or the password is incorrect.

-The user is warned about the mistake.

-The process starts again.

•Use Case 3: place an order

Primary Actor: client

Main Success Scenario:

-The user selects all products he/she wants to order.

-The user clicks on the “Order” button.

-The order is saved and a bill is generated.

Alternative Sequence 1: Missing products

-The user doesn’t select any product and places the order.

-The user is warned about the mistake.

-The process starts again.

•Use Case 4: search a product

Primary Actor: client

Main Success Scenario:

-The client fills any of the search fields with the desired data.

-Only the corresponding products will be visible to the user.

•Use Case 5: import initial set of menu items

Primary Actor: admin

Main Success Scenario:

-The user clicks on the “Import” button.

-The menu item is reset to the initial set of products.

•Use Case 6: add/modify a base product

Primary Actor: admin

Main Success Scenario:

-The user inserts all the required information in the graphical user interface.

-The user clicks on the “Add/Modify” button.

- The new product is added to the menu.

Alternative Sequence 1: Missing fields

-The user doesn’t fill in all the text fields

-The user is warned about the mistake

-The process starts again

Alternative Sequence 2: duplicate product

- A product with the same name already exists in the menu.

-The user is warned about the mistake.

-The process starts again.

•Use Case 7: Remove a product

Primary Actor: admin

Main Success Scenario:

-The user clicks the “-“ button for the product he/she wants to remove from the menu.

- The product is removed from the menu.

•Use Case 8: add a new composite product

Primary Actor: admin

Main Success Scenario:

-The user inserts the name of the new product and selects the menu items that it will be composed of.

-The user clicks on the “Create” button.

- The new product is added to the menu.

Alternative Sequence 1: Missing name/ less than 2 products selected

-The user doesn’t choose a name for the new product or selects less than 2 products for it to be composed of.

-The user is warned about the mistake

-The process starts again

Alternative Sequence 2: duplicate product

- A product with the same name already exists in the menu.

-The user is warned about the mistake.

-The process starts again.

•Use Case 9: generate a report

Primary Actor: admin

Main Success Scenario:

-The user inserts all the required information for generating the report.

-The user clicks on the "Generate" button.

- The report is created in a text file.

Alternative Sequence 1: Missing fields

-The user doesn't fill in all the text fields

-The user is warned about the mistake

-The process starts again

3. Design (design decisions, UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interfaces)

In this application I have used several design patterns. First of all, I have used the Composite Design Pattern for modelling the classes MenuItem, BaseProduct and CompositeProduct. MenuItem is an abstract class that contains only one abstract method called computePrice(). The method computePrice() in the BaseProduct class only returns the field price of the object, as it is known. On the other hand, in the class CompositeProduct, this method iterates through the productsList of the composite product, returning the sum of the prices of all the items from the list, this being the final price of the composite product. So, this design pattern lets us have base products and composite products, which contain a list of menuItems (so base products or composite products) as one of their fields, while also they themselves being of type MenuItem. So, a CompositeProduct (which is a list of MenuItem) is treated the same as a BaseProduct.

I have also used the Observer Design Pattern and I implemented it using PropertyChangeListener. In this case the observable class is the DeliveryService, hence why here we have the method for placing an order, and the observers are instances of the class EmployeeController. Every time an order is placed, every registered employee receives a notification of the order that needs to be processed. Every EmployeeController has a list of panding orders. When an employee clicks on the "Done" button, to mark an order as already prepared, this order is deleted from the pending list of orders of all employees. In the class DeliveryService we have an instance of the PropertyChangeSupport class, that will send alarms to the employees when certain properties change (when we place an order). We need to add to this instance all EmployeeControllers as listeners. The class EmployeeControllers must implement the PropertyChangeListener interface.

IDeliveryServiceProcessing is an interface that defines methods for all operations that can be performed by the users. DeliveryService is the class that implements this interface. This class is implemented using the Design By Contract method, meaning that for each method in the interface we define preconditions and postconditions that need to be fulfilled when the methods are implemented. We make sure the preconditions and postconditions are fulfilled by using the assert instruction. At the beginning of a method we check with assert that the preconditions are true and at the end of the method we check if the postconditions are true. Also, we may define for the DeliveryService class an invariant, a property that doesn't change its value and is always true. We also check this with assert at the beginning and at the end of each method.

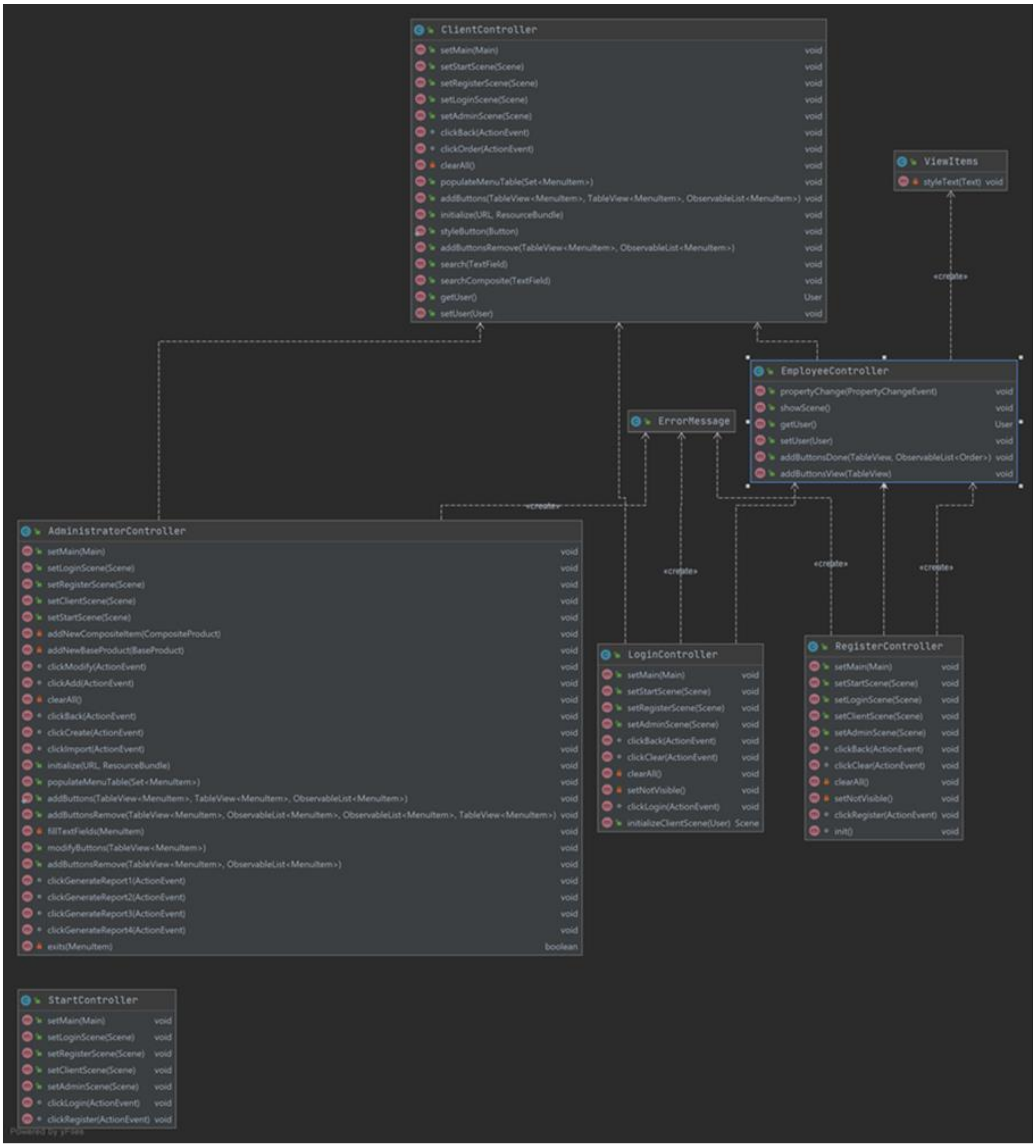
The application is divided into 3 packages: businessLayer, dataLayer, presentationLayer and start.

As for the data structures used, in the DeliveryService class we have an ArrayList of users, which contains all the users registered to use the application. The menu items are stored in a TreeSet because there can't be duplicates and we want to display them alphabetically in the menu. The orders are stored in a HashMap<Order, ArrayList<MenuItems>>. The key is the Order object, that holds all the information about the order and the value is the list of all menu items contained in that order.

The user interface is composed of the following windows: Register, Login, Client, Employee, Administrator, Start, View, Error.

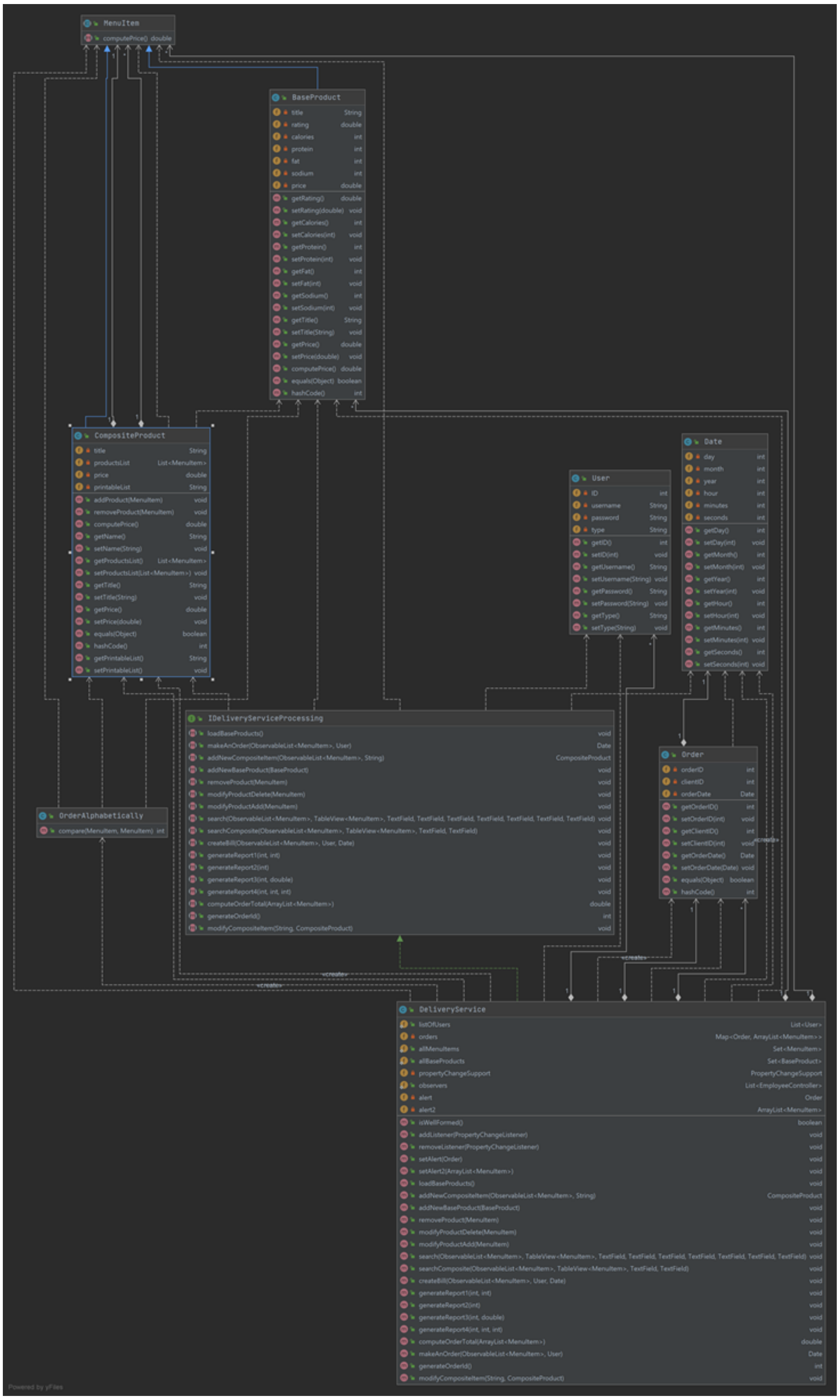
UML Diagram

presentationLayer Package

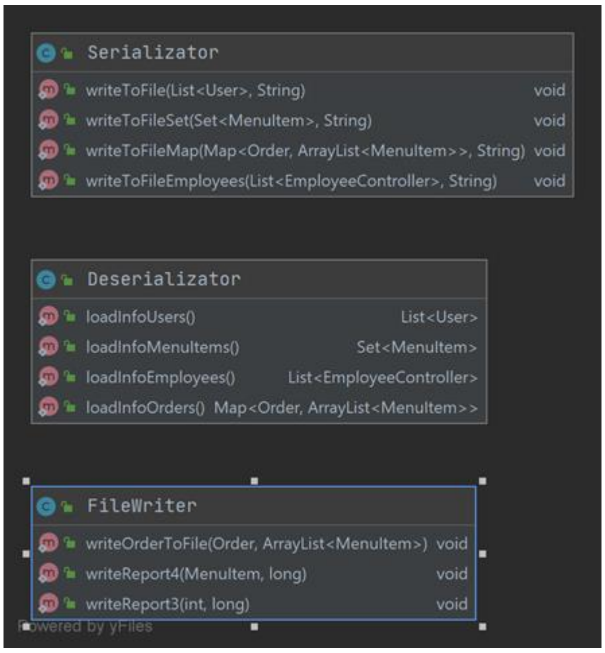


ClientController
setMain(Main) void
setStartScene(Scene) void
setRegisterScene(Scene) void
setLoginScene(Scene) void
setAdminScene(Scene) void
clickBack(ActionEvent) void
clickOrder(ActionEvent) void
clearAll() void
populateMenuTable(Set<MenuItem>) void
addButtons(TableView<MenuItem>, TableView<MenuItem>, ObservableList<MenuItem>) void
initialize(URL, ResourceBundle) void
styleButton(Button) void
addButtonsRemove(TableView<MenuItem>, ObservableList<MenuItem>) void
search(TextField) void
searchComposite(TextField) void
getUser() User
setUser(User) void
ViewItems
styleText(Text) void
«create»
EmployeeController
propertyChange(PropertyChangeEvent) void
showScene() void
getUser() User
setUser(User) void
addButtonsDone(TableView, ObservableList<Order>) void
addButtonsView(TableView) void
ErrorMessage
«create»
AdministratorController
setMain(Main) void
setLoginScene(Scene) void
setRegisterScene(Scene) void
setClientScene(Scene) void
setStartScene(Scene) void
addNewCompositeItem(CompositeProduct) void
addNewBaseProduct(BaseProduct) void
clickModify(ActionEvent) void
clickAdd(ActionEvent) void
clearAll() void
clickBack(ActionEvent) void
clickCreate(ActionEvent) void
clickImport(ActionEvent) void
initialize(URL, ResourceBundle) void
populateMenuTable(Set<MenuItem>) void
addButtons(TableView<MenuItem>, TableView<MenuItem>, ObservableList<MenuItem>) void
addButtonsRemove(TableView<MenuItem>, ObservableList<MenuItem>, ObservableList<MenuItem>, TableView<MenuItem>) void
fillTextFields(MenuItem) void
modifyButtons(TableView<MenuItem>) void
addButtonsRemove(TableView<MenuItem>, ObservableList<MenuItem>) void
clickGenerateReport1(ActionEvent) void
clickGenerateReport2(ActionEvent) void
clickGenerateReport3(ActionEvent) void
clickGenerateReport4(ActionEvent) void
exits(MenuItem) boolean
«create»
LoginController
setMain(Main) void
setStartScene(Scene) void
setRegisterScene(Scene) void
setAdminScene(Scene) void
clickBack(ActionEvent) void
clickClear(ActionEvent) void
clearAll() void
setNotVisible() void
clickLogin(ActionEvent) void
initializeClientScene(User) Scene
«create»
«create»
RegisterController
setMain(Main) void
setStartScene(Scene) void
setLoginScene(Scene) void
setClientScene(Scene) void
setAdminScene(Scene) void
clickBack(ActionEvent) void
clickClear(ActionEvent) void
clearAll() void
setNotVisible() void
clickRegister(ActionEvent) void
init() void
StartController
setMain(Main) void
setLoginScene(Scene) void
setRegisterScene(Scene) void
setClientScene(Scene) void
setAdminScene(Scene) void
clickLogin(ActionEvent) void
clickRegister(ActionEvent) void
Powered by yFiles

businessLayer Package


MenuItem
computePrice() double
BaseProduct
title String
rating double
calories int
protein int
fat int
sodium int
price double
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
getTitle() String
setTitle(String) void
getPrice() double
setPrice(double) void
computePrice() double
equals(Object) boolean
hashCode() int
CompositeProduct
title String
productsList List<MenuItem>
price double
printableList String
addProduct(MenuItem) void
removeProduct(MenuItem) void
computePrice() double
getName() String
setName(String) void
getProductsList() List<MenuItem>
setProductsList(List<MenuItem>) void
getTitle() String
setTitle(String) void
getPrice() double
setPrice(double) void
equals(Object) boolean
hashCode() int
getPrintableList() String
setPrintableList() void
User
ID int
username String
password String
type String
getID() int
setID(int) void
getUsername() String
setUsername(String) void
getPassword() String
setPassword(String) void
getType() String
setType(String) void
Date
day int
month int
year int
hour int
minutes int
seconds int
getDay() int
setDay(int) void
getMonth() int
setMonth(int) void
getYear() int
setYear(int) void
getHour() int
setHour(int) void
getMinutes() int
setMinutes(int) void
getSeconds() int
setSeconds(int) void
IDeliveryServiceProcessing
loadBaseProducts() void
makeAnOrder(ObservableList<MenuItem>, User) Date
addNewCompositeItem(ObservableList<MenuItem>, String) CompositeProduct
addNewBaseProduct(BaseProduct) void
removeProduct(MenuItem) void
modifyProductDelete(MenuItem) void
modifyProductAdd(MenuItem) void
search(ObservableList<MenuItem>, TableView<MenuItem>, TextField, TextField, TextField, TextField, TextField, TextField, TextField) void
searchComposite(ObservableList<MenuItem>, TableView<MenuItem>, TextField, TextField) void
createBill(ObservableList<MenuItem>, User, Date) void
generateReport1(int, int) void
generateReport2(int) void
generateReport3(int, double) void
generateReport4(int, int, int) void
computeOrderTotal(ArrayList<MenuItem>) double
generateOrderId() int
modifyCompositeItem(String, CompositeProduct) void
OrderAlphabetically
compare(MenuItem, MenuItem) int
Order
orderID int
clientID int
orderDate Date
getOrderID() int
setOrderID(int) void
getClientID() int
setClientID(int) void
getOrderDate() Date
setOrderDate(Date) void
equals(Object) boolean
hashCode() int
«create»
DeliveryService
listOfUsers List<User>
orders Map<Order, ArrayList<MenuItem>>
allMenuItems Set<MenuItem>
allBaseProducts Set<BaseProduct>
propertyChangeSupport PropertyChangeSupport
observers List<EmployeeController>
alert Order
alert2 ArrayList<MenuItem>
isWellFormed() boolean
addListener(PropertyChangeListener) void
removeListener(PropertyChangeListener) void
setAlert(Order) void
setAlert2(ArrayList<MenuItem>) void
loadBaseProducts() void
addNewCompositeItem(ObservableList<MenuItem>, String) CompositeProduct
addNewBaseProduct(BaseProduct) void
removeProduct(MenuItem) void
modifyProductDelete(MenuItem) void
modifyProductAdd(MenuItem) void
search(ObservableList<MenuItem>, TableView<MenuItem>, TextField, TextField, TextField, TextField, TextField, TextField, TextField) void
searchComposite(ObservableList<MenuItem>, TableView<MenuItem>, TextField, TextField) void
createBill(ObservableList<MenuItem>, User, Date) void
generateReport1(int, int) void
generateReport2(int) void
generateReport3(int, double) void
generateReport4(int, int, int) void
computeOrderTotal(ArrayList<MenuItem>) double
makeAnOrder(ObservableList<MenuItem>, User) Date
generateOrderId() int
modifyCompositeItem(String, CompositeProduct) void
«create»
«create»
«create»
Powered by yFiles

dataLayerPackage


Serializator
writeToFile(List<User>, String) void
writeToFileSet(Set<MenuItem>, String) void
writeToFileMap(Map<Order, ArrayList<MenuItem>>, String) void
writeToFileEmployees(List<EmployeeController>, String) void
Deserializator
loadInfoUsers() List<User>
loadInfoMenuItems() Set<MenuItem>
loadInfoEmployees() List<EmployeeController>
loadInfoOrders() Map<Order, ArrayList<MenuItem>>
FileWriter
writeOrderToFile(Order, ArrayList<MenuItem>) void
writeReport4(MenuItem, long) void
writeReport3(int, long) void
Powered by yFiles

4. <u>Implementation</u>

- <u>BaseProduct class</u>
  This is the class that defines a base product from the menu and it extends the abstract class MenuItem. It has the following fields: title, rating, calories, protein, fat, sodium, price. It has 2 constructor, one which takes 7 arguments for all the fields and one which takes an array of strings as an argument, each entry being the value of a field. This constructor is used for creating the objects after reading them from a given file. The overridden computePrice() method just returns the price because it is known.

- <u>CompositeProduct class</u>
  This is the class that defines a composite product from the menu and it also extends the abstract class MenuItem. It has the following fields: title, price, productsList, printableList. The productsList is an ArrayList of all the MenuItem objects that it is composed of. The printable list is a string used for printing the productsList in a table view in the GUI. The overridden computePrice() method returns the sum of all prices of the elements in the productsList.

- <u>MenuItem class</u>
  This is an abstract class with one abstract method called computePrice().

- <u>User class</u>
  This is a class that defines a user, having the following fields: username, password, id and type. The type can be one of the 3: administrator, client or employee. The id is generated automatically.

- <u>Date class</u>
  This class defined a structure for storing the date of the order having the following fields: day, month, year, hour, minutes, seconds.

- <u>DeliveryService class</u>
  This is the most important class of the application. It implements all the methods for the interface. It has some static fields: a list of users, a set of menu items, a set of base menu items, a list of observers (EmployeeControllers) and some private fields: a map for storing the orders (keys of type Order, values of type ArrayList<MenuItem>, a PropertyChangeSupport field, an “alert” of type Order and another “alert” of type ArrayList<MenuItem>. In the constructor, we deserialize the list of users, the set of menu items, the list of observers and the orders. Also, we make each element from the

EmoloyeeController list to be a listener of the PropertyChangeSupport object, so that they will be notified. For the Observer Design Pattern, we have two methods that modify the two alerts and fire a property change that is transmitter to the observers. The loadBaseProducts() method is implemented using streams and lambda expressions. The method addNewCompositeItem() creates a new CompositeProduct object, adds it to the list of menuItems and serializes the list. Basically, every time there is any modification on the list of users, the set of menuItems, the list of observers or the orders, we serialize the collection again so that the data saved in the text files is always updated. The methods addNewBaseProduct, removeProduct, modfiyProductDelete, modifyProductAdd are done similarly. Th two search methods and the 4 four generate report methods are also done with streams and lambda expressions. In the makeAnOrder method, we create an object of type Date based on the real-life LocalTime and LocalDate of when the order is placed.

- IDeliveryServiceProcessing interface

  Defines methods for all operations that can be performed by users, with preconditions and postconditions: loadBaseProducts, makeAnOrder , addNewCompositeItem, addNewBaseProduct, removeProduct, modifyProductDelete, modifyProductAdd, search, searchComposite, createBill, generateReport1, generateReport2, generateReport3, generateReport4, computeOrderTotal, generateOrderId, modifyCompositeItem.

- Order class

  This is the class for orders, having the following fields: orderId, clientID, date.

- OrderAlphabetically class

  This is a class used for ordering the elements from the TreeSet of menu items. It implements the comparator interface and overrides the compare method.

- Deserializator class

  This class contains methods for deserializing information into data structures, so that we can load the application with the latest data. We have methods for deserializing sets, lists and maps.

- Serializator class

  This contains methods for serializing information into text files, so that we can preserve the state of the application even after closing. We have methods for serializing sets, lists and maps.

- FileWriter class

  This class contains 3 methods for writing to a text file, which are helper methods used by the main methods which generate the 4 reports. They are static methods.
- AdministratorController class

  This is the controller for the admin profile window. It has 2 TableViews: one for displaying all the products and one for displaying the selected products to create a new composed menu item out of. The user can click the "+" buttons to select an item, but there can't be duplicates o the same product in a composite menu item. The user can also press the "-" button for removing an item from the menu. The "M" button can be also clicked if the admin wants to modify a product. The corresponding text fields will be filled with the old information of the menu item and the user can edit it then click "Modify". The same text fields are also used for entering the information for creating a new base product, which is added to the menu by clicking "Add". To create a composite item, the admin must select at least 2 products and give it a name in the required text field. And lastly, there are 4 sections for generating 4 types of reports. The user must fill in all the required information and then click generate. The report will be generated in a text file.
- ClientController class

  This is the controller for the client profile window. It is composed of 3 TableViews: one for displaying the base products, one for displaying the composite products and one for displaying the selected products for ordering. There are also text fields above all columns of the tables, used for searching products easily. There is also an uneditable text field that display the total price of the order. When the user clicks the "Order" button, the order is placed and a bill is generated in a text file.

- EmployeeController class

  This is the controller for the employee profile window, which implement the PropertyChangeListener interface. Each instance of this class has a list of pending orders(ArrayList<Order>) and a list of the corresponding lists of ordered menu items(ArrayList<ArrayList<MenuItem>>). The method propertyChange(PropertyChangeEvent evt) is called for every instance of EmployeeController when a new order is placed and adds to the two lists the new order information. This window is composed of a table view that diplays all the pending orders that have nor yet been processed. When an employee prepares an order he/she can click the View button to see the ordered menu items and then click the Done button, to indicate that the order is ready. When "done" is clicked, the pending order is removed from the lists of all the employees.

- ErrorMessage class

This is the controller for an alert window which pops up anytime the user makes some type of mistake. Its constructor takes a string as a parameter and displays that message.

- Login Controller class

This is the controller for the Login window in the GUI. It has 2 text fields for the user to enter the username and password. There is a Login button, a Clear button and a Back button. The user must fill in both text fields with correct information in order to open his profile.

- RegisterController class

This is the controller for the Register window in the GUI. It has 2 text fields for the user to enter a new username and password. The password must not have been used before. We also have a combo box, from which the user can choose what kind of account he/she creates, based on their position in the restaurant: client, admin or employee. Once the register button is clicked and the data is validated, the account information is serialized in the "users.txt" file. If the account is of type employee, then an EmployeeController instance is also created and added as an observer to the PropertyChangeSupport object in the DeliveryService class.

- StartController class

This is the controller for the start window, where users can register if they don't have an account or directly login into their profile.

View class

This is the controller for a window that pops up when an employee wants to see the ordered menu items from a specific pending order. It is simply a list of menu items displayed in a GridPane.

5. Conclusions

In conclusion, I believe that I have learned a lot of things from this assignment, as it was a complex one that required a lot of work, research and thinking. Most of the things I used in this project I haven't used before (streams, lambda expressions, serialization, composite design pattern, observer design pattern, design by contract, etc). Now, I can say that I am comfortable using them and I can see their applicability. As for further development, I think there could be a lot more to do to the GUI, to make it look more professional, maybe even add pictures to every menu item.

6. Bibliography

-ASSIGNMENT_4_SUPPORT_PRESENTATION.pptx
-Lecture notes