DOCUMENTATION

ASSIGNMENT NUMBER 4:

**— FOOD DELIVERY MANAGEMENT SYSTEM—**

NAME: Chețe Doru-Gabriel

GROUP NUMBER: 30423

1. Objective of the assignment

The main object of the assignment is to build an application that manages the products, clients and orders of a food ordering system. In order for the users to have separate views, a log in mechanism will be provided and every user will have its own role with its permission to do certain things. The idea is to use a system where an administrator can edit, add or create new menus and where a client can order products, while an employee is notified of new orders.

The secondary objectives are:

1. Analyze the problem and identifying the requirements
2. Design the application that can manage the orders
3. Implement the orders management application
4. Test the orders management application

With these objectives clearly set, the start of the analysis of the problem will be presented:

2. Analysis of the problem, modeling, scenarios and use cases

Analysis:

The problem can be viewed as a matter of separating the views of the users based on their respective roles and providing an underlying mechanism to store, load and use the client/order/menu data. The main success scenario for an administrator would be if he or she selects the option to add a new product and if all corresponding data is inserted than an item will be added, once he or she hits the add button. The application would in this case save the product’s data and display an acknowledge message.

An alternative unsuccessful scenario is if the administrator, the primary actor of the application, doesn’t insert any of the necessary fields of an item. Then an error message is displayed.

Modeling:

In order to model this problem, we can consider three packages: Business Logic, GUI and Data Access.

The User class is the model for all user data and has a specific field for the role to be able to recognize a user’s permissions. The Order class models an order, with its respective ID, the user who ordered it and the time it happened at. MenuItem class and the two classes that extend it, BaseProduct and CompositeProduct use the Composite Design Pattern.

Finally, within the GUI package, the controller class will handle all of the UI elements.

Scenarios:

The use scenarios for this application are a bit more varied. Once the application is launched, one has to log in with its account. If someone doesn’t have an account, he or she can register as a new user (with basic privileges, so a client) and return to the login page to use the credentials. If the log in isn’t successful, an error message will be displayed.

Once logged in, a regular (client) user can see its corresponding interface, where he or she can find a product based on some search criteria or can add products to a basket and then finally order something. If the user is an employee, he can go to its corresponding view, and see if any new orders were created.

If a user is an admin, then he or she can edit the menu items list, or create a new menu much in the same way a client would create an order.

Use cases:

Diagram

Description automatically generated

The use cases for this program are in conclusion quite simple, and refer to actions performed on the user, product and orders data. As we’ll see in the following sections, this data is actually persisted using serialization, so there’s a lot of loading and storing the data going on, and the UI updates while the data is being serialized or deserialized.

3. Design of the order management app

For the design of the management application, I started with the ideas presented in the modeling part of the documentation and followed them faithfully.

The main idea in the beginning was to separate the UI of the different type of users. From the log window admin has to be separated from the other users, and while an employee can also order, only he can access the new orders that are generated. For this I tried to design an app that while doesn’t use encryption or any real security system, tries to create a form of access rights, so that parts of the UI can only be accessed by certain users and thus provides some different user experiences for the different roles.

The idea of serialization was something that had to be kept in type at all times, so making sure the user/product/orders data was being saved and loaded correctly was another priority.

Despite there being multiple .fxml files, only one Controller class is used and a controller object is created everytime a new window is created. (a new window is not actually created in the case of the employee, it is just hidden from the user and can be viewed only by a user with employee rights).

Data structures that are used:

With respect to the data structures that are used, we can talk mostly about lists, but obviously in the case of the orders data, a map data structure is used for storing the order and set of ordered items pair. Lists are used with the ArrayList implementation mostly, within the controller for storing and loading serialized data, as well as for building an order or building a new menu. The ObservableList is also used to satisfy the Observer Design Pattern. While the app is running (and new orders may be created) an observable list of orders is created within the controller, and any new orders, while being also serialized, are added to the observable list, which is bound to a JavaFX TableView that corresponds to the Employee. While it is not practically possible to access both client and employee at once, the update is done at once, by the observer TableView object, as visible if trying to order something as an employee and then switching to the Employee page.

UML diagram

Timeline

Description automatically generated

Algorithms:

For algorithms, since this is an application that is heavily based on the interaction with the user/product/order, there isn’t really that much in the way of algorithms. However, some simple algorithms in order to ensure proper functioning of the way serialization is being done, as well as handling the ensuing problems from different controller objects that may be instantiated. For example, a way of making sure the logged in user’s data is being saved for future access by a new controller object (for an admin view or a user/employee view) also makes use of serialization. Once logged in, a user’s username and role gets saved for a future controller object to access, namely the controller object that was initialized after the new .fxml was loaded.

4. Implementation

The implementation, considering the constraints of the design is quite straightforward. JavaFX may complicate things, however the libraries and functionalities it provides are very useful. There are multiple .fxml files that can be loaded to perform different steps of the program. The first one to be loaded is the log in one, which represents the log in page, where once user data is valided, its corresponding .fxml file will also be loaded. If a user doesn’t have an account, he may open the registration page where another .fxml is loaded. Back to the log in, once successful, I implemented the two different interfaces separately. Each of the fields of the admin had to be validated, when an action is performed. For adding a new product, all the fields must be filled, while for editing, at least some have to be filled. Then for creating an order I thought of a system where an admin can select certain IDs of products and them one by one to a list, from where they can be added later to the new menu that is being built. While the admin is selecting IDs of products and adding them to the new menu by pressing a button, their names show up on the screen so he or she may see what is being added. At the end, when the create menu button is pressed, they are all created into a menu item object that gets added to the products list, serialized and shows up in the table view. Similarly, when a client is building an order, he may view what he’s ordering while he’s doing it. The “basket” is being shown with all its contents in another box. When ready, the order button is pressed and the order is processed. As I previously mentioned, the regular user and the employee pages are not truly separate. In terms of UI, they are two JavaFX Anchor Panes stacked within a Stack Pane and the way the UI is being shown is managed by the permissions one has. An employee may proceed further and the Employee Anchor Pane which contains the UI, namely the Table View with the new orders is only then being shown, actually the Anchor Pane above it is sent to the back and thus displaying what is underneath it. The controller object is thus the same, so the observable list that contains the new orders updates the observer (the Table View). So the Table View will update automatically and while this cannot be seen only if two windows were open, one can see that as an employee, if one orders something and immediately after goes to the employee page, the table is updated.

5. Results

For testing the implementation of the program, I tried to consider all scenarios, including some scenarios that generate errors. I tried logging in with invalid data, registering. I tested a scenario where the user didn’t have an account already and a lot of scenarios where the employee both orders something and views its order, this scenario in particular was useful to seeing that the update is done well. A lot of the data in the .csv was duplicated so I had to test to make sure there were no more duplicates, as once the imported data is serialized it keeps getting used and modified and duplicates can lead to some issues with ID indexing and such. I did some testing to ensure that while serializing there aren’t any issues and data doesn’t get corrupted in any way.

I also considered some scenarios where users with different roles try to access other parts of the UI than what is permitted to them.

6. Conclusions

During the analysis and development of this project I learned a lot about development of applications using JavaFX. Some aspects of using these vast libraries weren’t clear to me up to this point. I learned more about how JavaFX UI elements such as Table View can be bound to observable lists or observable objects and can update themselves accordingly. I learned more about using elements such as the stack pane and managing more interfaces while only showing one and leaving the rest of them hidden from the user. This may rise some security concerns, but for the purpose of this assignment I chose to ignore them, in order to be able to implement the required Observer design pattern. In conclusion, I can say there’s a lot of lessons I learned, but mainly that problems that arise while developing a project, especially while designing UI are hard to handle and it’s probably best to just consider them ahead of time and try to avoid them as much as possible.

7. Bibliography

Helped me with finding RGB colour codes for the UI: [https: //www.color -hex.com /color /5d1049](https://www.color-hex.com/color/5d1049)

Helped me learn more about the File Writer, used for bill generation: https: //www.geeksforgeeks.org/ filewriter-class-in-java/

Helped me understand how to use Table View in JavaFX: https:// jenkov .com /tutorials

Helped me understand Java Doc: https:// www. baeldung.com/ javadoc

Helped me understand how a Java FX element can be bound to an observable list using a Property object : https: //stackoverflow.com/ questions/ 56970878/bind-tableview-to-observablelist

Helped me understand Java serialization: https: //www.baeldung.com/ java-serialization

Helped me understand more about the Stack Pane in JavaFX: https://www.javatpoint.com/javafx-stackpane