FUNDAMENTAL PROGRAMMING TECHNIQUES

Assignment 4

Restaurant Management System

Student: Halmai Erik

Group: 30422

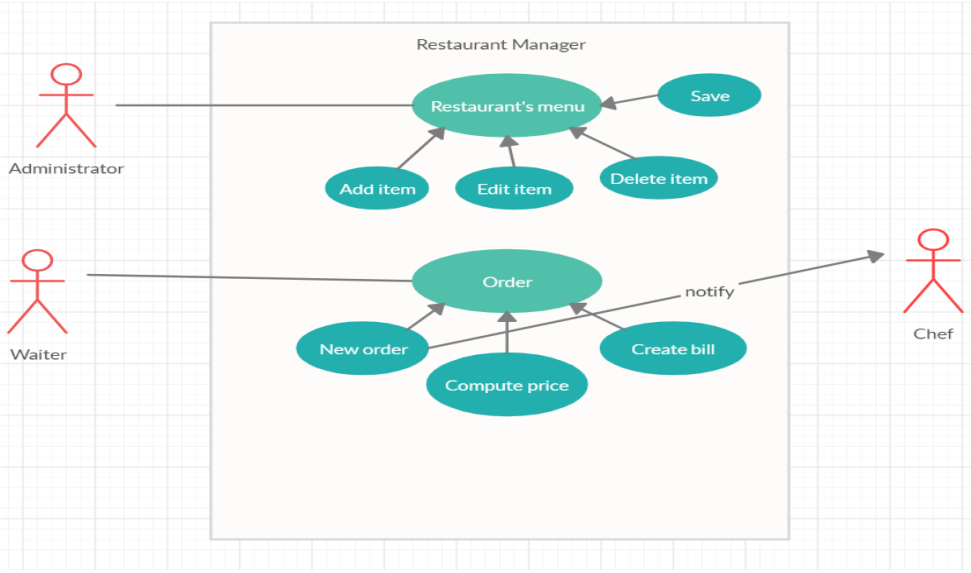
1. Objective

The main objective of the assignment is the implementation of a restaurant management system having three types of users: administrator, waiter and chef. The administrator can add, delete and modify exisiting products from the menu. The waiter can create a new order for a table, add elements from the menu and compute the bill for an order. The chef is notified each time it must cook food that is ordered through a waiter.

The system's main objective of the assignment can be divided into several secondary objectives, each achieved separately. These secondary objectives represent the steps that are taken in order to achieve the main objective of the assignment. These steps are the following:

1) **Creation of the BaseProduct, CompositeProduct and Order objects (Chapter 4)**: The base and composite product objects should be created using the Composite Design Pattern, with the help of the MenuItem interface. In this way, both complex and simple elements can be treated the same way. The Order class should overwrite the equals() and hashCode() methods, so that orders of the same table will have equal hash codes.
2) **Implementation of the Restaurant class implementing the IRestaurantProcessing interface (Chapter 4):** The Restaurant class should implement the operations that can be performed by the three users, from the IRestaurantProcessing interface. It should also contain a HashMap Collection, holding the ordered items for each order.
3) **Design of the GUI frames (Chapter 4)**: The graphical interface is designed according to the Model View Controller architectural pattern.
4) **Use of the Observer Design Pattern for the ChefGUI (Chapter 4)**: The ChefGUI should contain a table that is updated each time a waiter adds a new order of a CompositeProduct.
5) **Handling of exceptions and bug fixing (Chapter 4)**: The created application should avoid crashing when unwanted situations occur and rather, it should display an error message.
6) **Serialization (Chapter 5):** The administrator has the possibility to save the restaurant's menu, once he/she edited it. The menu will be saved, using serialization, to a file with the name given as argument when running the jar file.

2. Problem analysis, modeling, scenarios, use cases



1) **Defining the operations:** This application is created to be used by three kinds of restaurant employees, all having different operations they can do. An administrator has the permission to change the restaurant's menu and save a default menu that will be loaded every time the application is ran. The waiter can create new orders and generate the bill for a chosen table. In case the table has multiple orders, the list will be updated and the bill will contain all the orders. After generating the bill, the table will be considered to be free. The chef will be notified each time an ordered of a CompositeProduct was created by a waiter. I chose to not notify the chef in case of BaseProduct orders, because these can be grabbed by the waiters too.

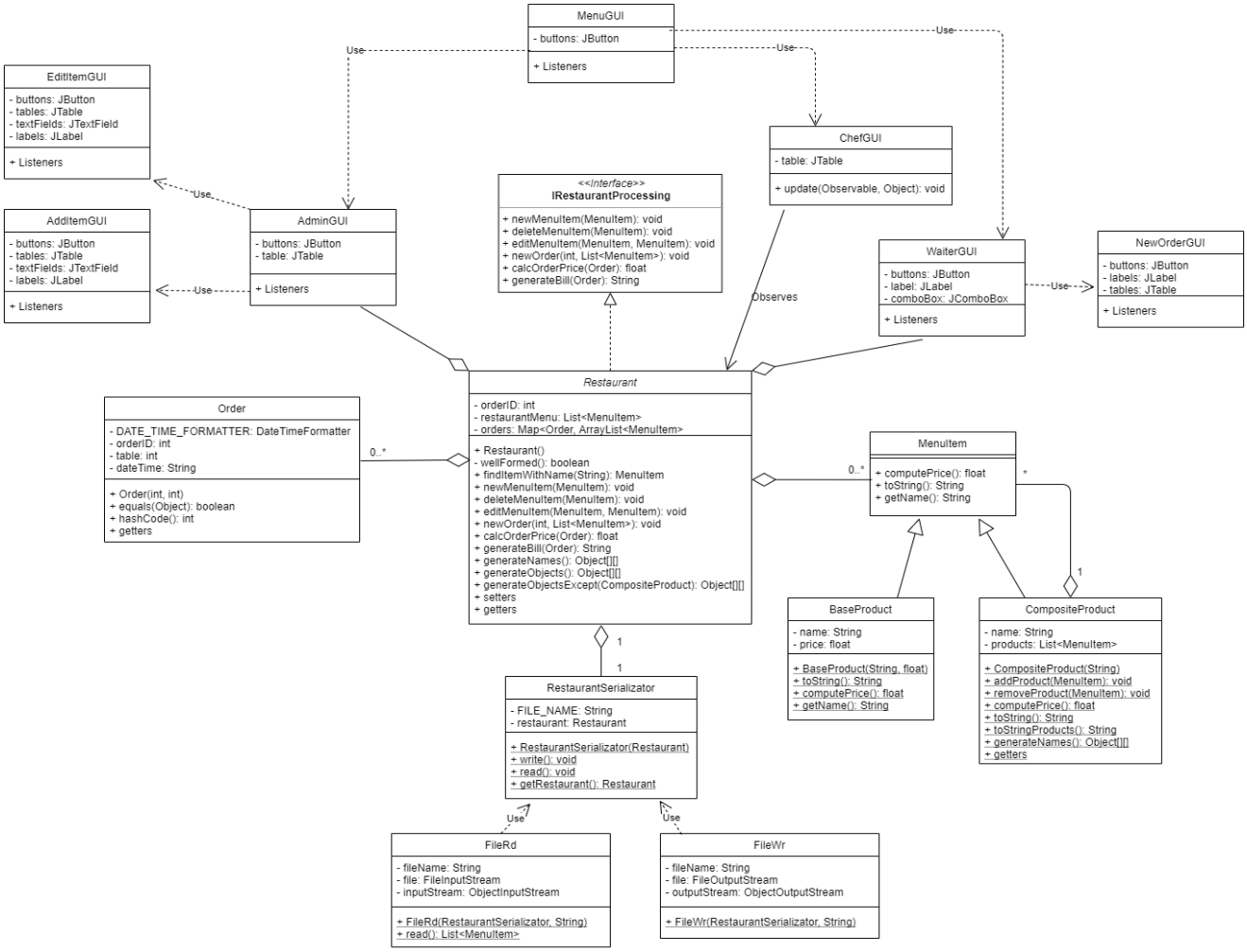
2) **Selecting the operation:** The user can select from 6 possible operations: derivative, integration, addition, subtraction, multiplication or division. Two of these six operations (derivative and integration) work on a single operand. Two buttons are located next to the two input fields, which perform these operations on the field that they are next to. The other four operations need two fields. These are marked with buttons having '+' for addition, '-' for subtraction, '*' for multiplication and '/' for division.

3) **Getting the bill:** The waiter has the permission to generate the bill for a chosen table. Each time a bill is generated, the selected table will be free and won't have any orders. In case the waiter tries to generate a bill for a free table, he/she will be prompted by an error message. The bills are of the following form:

    Ordered items:
    <ordered_item> - <price> lei
    Total price: <total_price>
    Table: <table_id>
    Date: <dd-mm-yyyy hh:mm:ss>

4) **Serializing the restaurant's menu:** If the application doesn't find the file with the name given as , it will prompt the user with a warning saying "No input file found". This file can be generated by the administrator from his/her GUI, by click on the save button. The created file will contain the restaurant's menu at that time and it will be loaded each time the program is started.

3. Design



Packages:

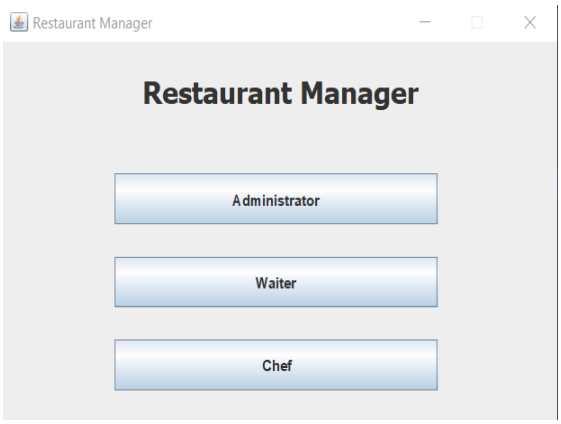
- BusinessLogic package: Contains the BaseProduct, CompositeProduct, MenuItem, Order, Restaurant classes and the IRestaurantProcessing interface.
- DataLayer package: Contains the RestaurantSerialization, FileWr and FileRd classes.
- Util package: Contains the Start class and 4 packages:
    - Admin package: Contains the AdminFrame, AdminController, AddItemFrame, AddItemController, EdiItemFrame and EditItemController classes.
    - Chef package: Contains the ChefFrame class.
    - Menu package: Contains the MenuFrame and MenuController classes.
    - Waiter package: Contains the WaiterFrame, WaiterController, NewOrderFrame and NewOrderController classes.

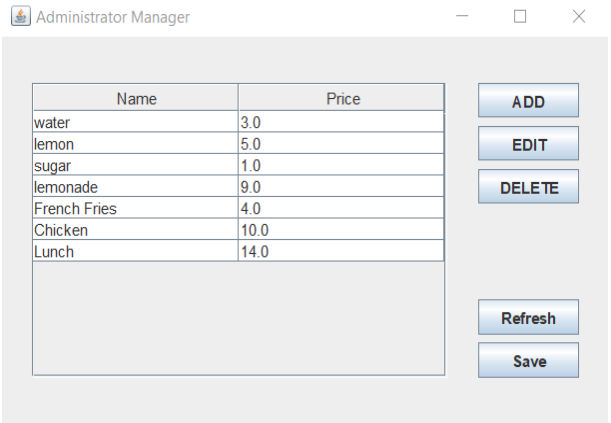
4. Implementation

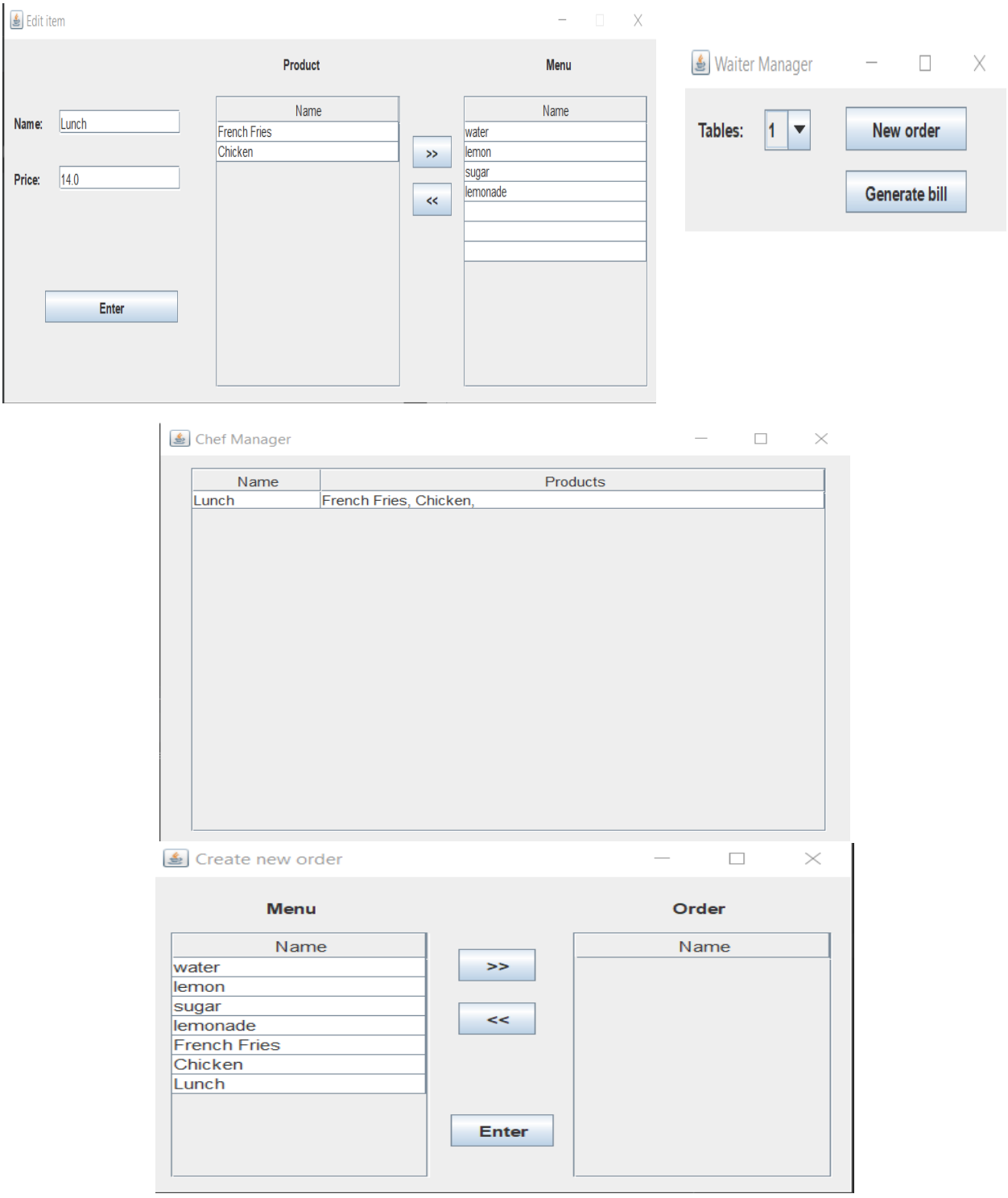
1) **BaseProduct**: Represents the base products from the restaurant's menu. Extends the MenuItem abstract class.
   a) BaseProduct(String, float): Constructor method. Sets the BaseProduct object's name and price to the values given as parameters.
   b) toString(): Overwritten toString method for the bill creation.
   c) computePrice(): Implemented computePrice method of MenuItem abstract class. In this case, of the BaseProducts, it returns the price of the product.
2) **CompositeProduct**: Represents the composite products from the restaurant's menu. Products that contain more items. Extends the MenuItem abstract class.
   a) CompositeProduct(String): Constructor method. Sets the CompositeProduct object's name to the String given as parameter and initializes the list of products in the object.
   b) addProduct(MenuItem): Adds a MenuItem object to the list of products of the CompositeProduct object.
   c) removeProduct(MenuItem): Removes a MenuItem object from the list of products of the CompositeProduct object.
   d) computePrice(): Implemented computePrice method of MenuItem abstract class. In this case, of the CompositeProduct, it calculates the sum of the prices in the list of products.
   e) toString(): Overwritten toString method for the bill creation.
   f) toStringProducts(): Returns a string containing the products inside the composite object.
   g) generateNames(): Generates a matrix of objects containing the names of the products in the composite object. Used to populate the JTables.
3) **Order:** Represents the details of an order. It does not include the ordered items.
   a) Order(int, int): Constructor method. Sets the orderID and table number to the arguments given as parameters and sets the dateTime String by generating the current LocalDateTime and formatting it to the pattern "dd-MM-yyyy HH:mm:ss".
   b) equals(Object): Overwritten equals method for the hashCode method.
   c) hashCode(): Overwritten hashCode method. Generates the hash code for the Order object.
4) **Restaurant:** Represents the whole restaurant system. It implements the IRestaurantProcessing interface and all of its methods.
   a) Restauran(): Constructor method. Initializes the orderID variable, restaurantMenu list and orders HashMap.
   b) wellFormed(): Well Formed type method. Verifies the variables of the Restaurant object.
   c) findItemWithName(String): Finds the MenuItem in the restaurant's menu that has the name equal to the String given as parameter.
   d) newMenuItem(MenuItem): Overwritten newMenuItem method of IRestaurantProcess interface. Adds the new menuItem object to the list containing the restaurant menu. If the menuItem is already in the list, it doesn't do nothing. Checks if wellFormed.

e) deleteMenuItem(MenuItem): Overwritten deleteMenuItem method of IRestaurantProcess interface. Deletes the given menuItem from the restaurant's menu. It also deletes the items containing the given menuItem from the restaurant's menu.
f) editMenuItem(MenuItem, MenuItem): Overwritten editMenuItem method of IRestaurantProcess interface. Replaces the first menuItem with the second.
g) newOrder(int, List<MenuItem>): Overwritten newOrder method of IRestaurantProcess interface. Creates a new order with tableId and list of ordered items given as parameters. If the table already has an order, it adds the menuItems to that order. If the order contains composite products, it notifies the Chef. Checks if wellFormed.
h) calcOrderPrice(Order): Overwritten calcOrderPrice method of IRestaurantProcess interface. Calculates the price of the order given as parameter.
i) generateBill(Order): Overwritten generateBill method of IRestaurantProcess interface. Generates the bill for the order (table) given as parameter.
j) generateNames(): Generates a matrix of objects containing the names of the items in the restaurant's menu. Used to populate the JTables.
k) generateObjects(): Generates a matrix of objects containing the names and prices of the items in the restaurant's menu. Used to populate the JTables.
l) generateObjectsExcept(CompositeProduct): Generates a matrix of objects containing the names of the items in the restaurant's menu, except the name and the name of items in the CompositeProduct given as parameter. Used to populate the JTables.

5) **FileWr:** It is used to write the serialized objects into a file.
   a) FileWr(RestaurantSerializator, fileName): Constructor class that writes the serialized object in the file with the name given as parameter.
6) **FileRd:** It is used to read the serialized objects from a file.
   a) FileRd(RestaurantSerializator, String): Constructor method that sets the input streams and the name of the input file.
   b) read(): Returns the read list of menu items from the input file.
7) **RestaurantSerializator:** Represents the object that serializes and deserializes the restaurant's menu.
   a) RestaurantSerializator(Restaurant): Constructor method that sets the restaurant object.
   b) write(): Writes the serialized restaurant's menu in a file
   c) read(): Reads the serialized restaurant's menu from a text file given as the jar argument







5. Results

The application yields the results of the commands executed by the three users. The waiter can create the bills for the orders. These bills will be text files containing the bill's string of the format described above. The created text files will have appropriate names that include the date and time that

they were created, ex: “bill20200624125502.txt”. This makes it easier to search for bills from a given date and it also makes the coding for the saving process easier.

The administrator can create, if not already existing, or update the text file with the serialized objects.. This file contains the MenuItem objects from the restaurant’s menu, created through serialization. Each time the application is started, it will search for a file having the name given as argument when running the jar file and if found, it will deserialize the information from the file and load it to the restaurant’s menu.

6. Conclusions

By following the secondary objectives, the created application achieves its main objective. The created application is simple and it could use some feature development, but it does the operations it should. The system simulates a real-life application and it could be used by a real restaurant after some bug fixing, database and server connections.

The Composite and Observer design patterns made the writing of code much easier. Because of the Composite Design Pattern, there was no need to use the ‘instanceof’ operator for the addition, deletion and editing of the menu. Moreover, it made it possible to store the items in an ArrayList, that forms the restaurant’s menu. The Observer design pattern made the chef’s GUI straight forward to implement. The only thing I needed to do is to extend the Observable class in the Restaurant class, notify the observers once an order of a CompositeProduct was made, implement the Observer interface in the Chef’s GUI and write the appropriate update() method to add a new row to the Chef GUI’s table.

The HashMap used to store the information for each order, made the waiter’s operations simple to implement. The key of the HashMap is an Order object, that generates the hash code according to it’s tableID (orders from the same table will have the same hash code) and the value is an ArrayList of MenuItem objects that form the ordered items.

Through serialization, the administrator can create the restaurant’s menu and save it, so that it can be used by the waiters and chefs. This is a new mechanism that I didn’t use before, but I learned how useful it is through this assignment. It makes it possible to save values used by the application without needing to connect to a database for example.

For future development, I have 3 ideas:

1) Implementation of a more user friendly and well-designed GUI.
2) Possibility for the chef to mark the prepared items as complete and signal the waiter, that created the order, to deliver it.
3) A login and register system to authorize administrators, waiters and chefs each time they want to log in.

7. Bibliography

- https://www.geeksforgeeks.org/serialization-in-java/
- http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_4/Assignment_4_Indications.pdf
- https://www.baeldung.com/java-observer-pattern
- https://www.baeldung.com/java-composite-pattern
- https://www.baeldung.com/java-equals-hashcode-contracts