Restaurant Management System

1. The assignment's objective

The main objective of this assignment is to develop an application used to manage a restaurant. The restaurant should have an administrator, a waiter and a chef, each having their own unique privileges: the administrator can add/edit/delete items from the menu, the waiter can add orders or compute bill for an order and the chef is notified every time a new order that necessitates cooking is made. In order for this to be done we have to make sure that certain secondary objectives are met, such as: having two types of menuItems: BaseProduct and CompositeProduct, that both extend the abstract class MenuItem, that contains the abstract method computePrice(), in that way using the composite Design pattern for the classes above, or the chef should be notified every time an order that contains a CompositeProduct is placed (that means that he has to cook) using the Observer design pattern. What's more the interface IRestaurantProcessing, that contains the declaration of the methods used for the operations that the waiter and administrator can do (they will be elaborated in chapter 4), should contain post and preconditions for each of those methods (design by contract). The Restaurant class should contain a Map structure that is used to store the Orders information (for each order we can have multiple MenuItems). The key will be of Order type (A class that will be also elaborated in chapter 4), so the hashCode() method will be overridden, as well as the equals() method and the value will be a collection of MenuItems. Last but not least, a UI needs to be generated, in order to ensure a more user friendly way for the administrator, waiter or chef to interact with the management system of the application, in that way we have an MVC pattern.

2. Problem analysis, modelling, scenarios, use cases

The Restaurant management system should be implemented such that it could be used through a User Interface, so there has to be a correlation between the action performed when pressing a button and the operations that need to be performed, for instance if a waiter presses the add order button, the method declared in the IRestaurantProcessing interface and implemented in the Restaurant class should be called. This correlation is done through a Controller class, that takes the Strings that are introduced, instantiates an object (an Order if the operations are done on an Order, i.e. adding, computing price or generating bill for an Order or a MenuItem if the operations are adding, editing or deleting a MenuItem) and passes that object as a parameter to a method of the Restaurant class in order for the corresponding operation to be executed. (this will be further explained when talking about the Controller class in the 4th chapter). In terms of problem analysis, for each of the operations performed, the application has to make sure that every precondition or postconditions are met. For example, when inserting a new MenuItem, we have sure that there isn't another with the same name, or that the MenuItem to be introduced respects some conditions (the price should be strictly positive). As postconditions, the menu's size (the menu consists of all the MenuItems) should have increased by one after adding the new MenuItem. Also, for each method that corresponds to a waiter or administrator operation another method called isWellFormed() is called (the isWellFormed() method makes sure that after every operation the Restaurant is well defined, i.e.: there isn't any order containing MenuItems that are not in the menu).

In order to better understand the above aspects and to summarize them, a use case will be presented below that shows how the application works if the administrator wants to add a new MenuItem:

Use Case: Restaurant Management System

Operation: Adding a new MenuItem

- The user starts the application and presses the Administrator button to go to the Administrator User Interface.
- The administrator introduces a name and a price for a MenuItem, if it is a BaseProduct or a name and a list of MenuItems for a CompositeProduct (the procedure by which the list is chosen will be presented when detailing the UI for the Administrator).
- The Controller class method that correspond to adding a new MenuItem checks if the MenuItem doesn't already exist. If it does, a message informing the administrator pops up. If not, it continues.
- The same method, checks for the name,price introduced or the list of MenuItems to be valid (the price should be strictly positive and the list should contain only MenuItems that are already in the menu. Also, the name should have length bigger than 0). If they are not valid, a message informing the administrator will pop up. If not, it continues.
- A new MenuItem is created and instantiated with its corresponding type: BaseProduct if the list of ingredients is null or CompositeProduct otherwise.
- The new MenuItem created is passed as an argument to the restaurant's method for adding a new MenuItem.
- After it is successfully added, a message informing the administrator that the MenuItem was successfully added pops up.

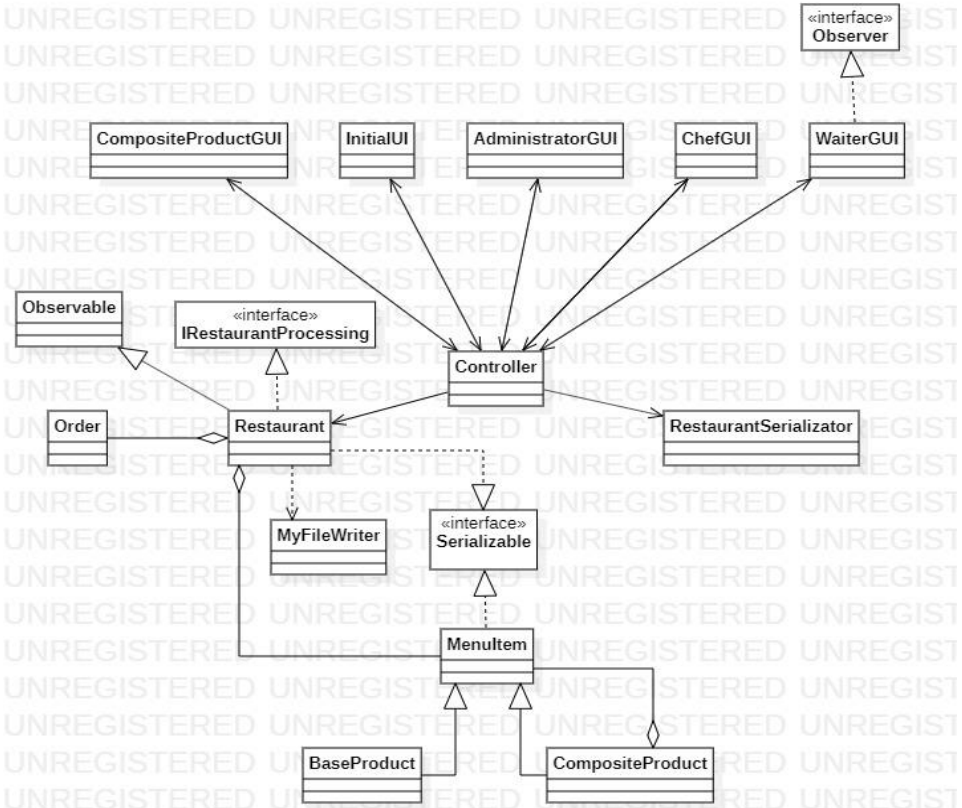
Similar use cases can be developed for any other of the operations that a waiter or an administrator can do.

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

Talking about structural decisions, the assignment has 4 packages: the Model package, that contains the Interface IRestaurantProcessing and the classes for MenuItem, BaseProduct, CompositeProduct, Order and Restaurant, the View package, that contains all the UI classes, the DataLayer that contains a Class for serializing the Restaurant's MenuItems and the Controller package that contains the Controller class. By diving the project into these Packages, we make sure that we achieve an MVC structural pattern.

The design decisions are reflected in the MenuItem abstract class and the BaseProduct and CompositeProduct classes that implement the abstract method computePrice() of the abstract class mentioned above, in that way having a Composite Design pattern. Another design pattern is used for the chef to be notified every time a new Order containing at least one CompositeProduct is placed, and that is the Observerer. This is done by the Restaurant class extending the Observable class and calling the setChanged() and notifyObservers() methods when inserting an order with a CompositeProduct and by the Chef UI implementing the Observer interface and overriding the update() method. What's more, the Restaurant class is designed using design by contract: assert is used at the beginning and end of each method that corresponds to an operation, in that way assuring that the pre and postconditions are met. Also the method isWellDefined() is called for each one of them. For the IRestaurantProcessing interface, a Javadoc is generated in order to showcase the pre and postconditions of each of the methods dedfined.

In terms of the data structures used, in the Restaurant class a HashMap is used for storing the orders with the Key of each order being an object of type Order and the value being a LinkedList of MenuItems. Also, a HashSet is used for storing the MenuItems, in that way assuring that we can't have two equal MenuItems in the menu.
Talking about user interfaces, we have five of them. At the launch of the application, an initial UI is visible, containing three buttons that, when pressed, lead to the corresponding UI (if we press the Administrator button, we make the Administrator UI visible and the same with the others)
In term of the Relationships, there is an association between the Controller and the GUI classes and the other way around. The Controller also contains an object of type Restaurant and one of type RestaurantSerializator. Between Restaurant and Order there is an Aggregation, as well as between Restaurant and MenuItem. BaseProduct and CompositeProduct both extend MenuItem, between CompositeProduct and MenuItem existing an Aggregation relationship as well. In terms of implementing interfaces, the IRestaurantProcessing interface is implemented by the Restaurant class, which also extend the Observable class and the Observer interface's method update() is implemented by the WaiterGUI class.
These relationships are better illustrated by the UML class diagram below:



4. Implementation

- The DataLayer package- contains two classes, RestaurantSerializator, used for serializing and deserializing a Restaurant object into and from the file named "restaurant.ser". Actually, not everything from the Restaurant's object is serialized, only the MenuItems from the menu, the other fields (such as the orders) are declared transient. (They are ignored in the serialization process). The second class is MyFileWriter that contains a single method used in the Restaurant's method generateBill(). It generates a .txt file with cinformation about an order.
- The Model package contains several classes: The Order class contains fields such as the OrderId, the date and the table, the first and last being integers. As methods, it overrides the hashCode() and the equals() methods because it is used as a key in a map of orders in the Restaurant Object. The hashCode() is overridden in such a way that it computes the hash code by multiplying the values stored in its fields with prime numbers.

  The abstract class MenuItem implements Serializable and it contains fields such as name and price, and getters and setters for these. It also contains an abstract method computePrice() that is implemented by the classes that extend this class: BaseProduct and CompositeProduct. BaseProduct's computePrice() acts like a getter for the price of the object of type BaseProduct. CompositeProduct also contains a list of MenuItems and the computePrice() method adds all of the MenuItems' prices from the list, returning their sum.

  The interface IRestaurantProcessing defined six methods: createMenuItem(), deleteMenuItem(), editMenuItem(), used by the administrator to add, delete or edit menuItems to/from the menu and createNewOrder(), computePriceOrder(), generateBil() used by the waiter to add a new Order, to compute the price or to generate a bill for an existing order. For this Interface Javadoc was generated, highlighting the pre and postconditions for each method.

  The Restaurant class extends Observable and implements the Serializable and IRestaurantProcessing interfaces. It has as fields a HashMap containing information about orders with the key of type Object and the Value a LinkedList of MenuItems and a LinkedHashSet for storing the MenuItems that are present in the restaurant's menu.

  This class contains important methods such as getOrder() that checks if an order with an OrderID already exists, or getMenuItem() that has as parameter a String and is used for finding out if a MenuItem with the name equal to the parameter exists or not. The method isWellDefined() checks if for every order, all the MenuItems ordered exist in the menu.

  Other methods are the methods defined by the IRestaurant processing. They assure that the pre and postconditions are met by using assert. The createMenuItem() method checks if the menuItem passed as parameter is not null doesn't already exist in the menu (two MenuItems with the same name should not exist), checks if all of the ingredients are in the menu and only then it adds the MenuItem to the menu list. The deleteMenuItem() checks if the passed parameter is not null and if it exists in the menu. If so, it is deleted from the Menu and all other menuItems containing it are deleted (using the checkItemContains() method that checks if an item contains other items). The editMenuItem() is given two parameters: oldMenuItem, newMenuItem and it checks that both of them are not null and that menu contains oldMenuItem and does not contain newMenuItem. After that, the oldMenuItem is removed and the newMenuItem is added and for each other MenuItem that contains the oldMenuItem the same

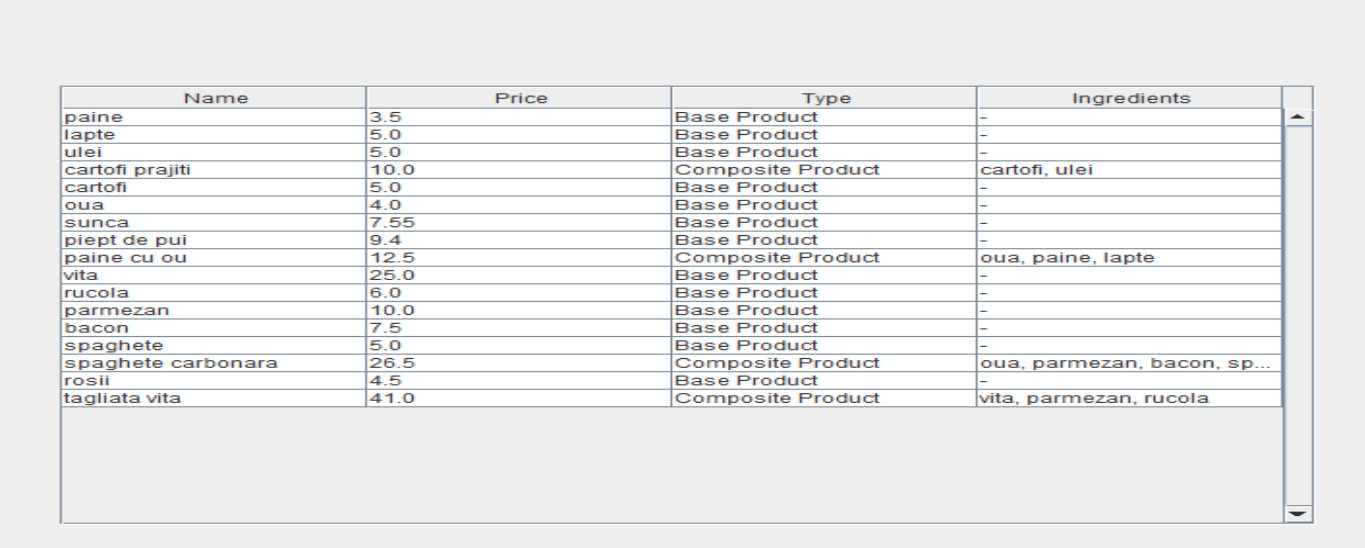
procedure is done(by using the method updateCompositeProduct() that checks if a composite product contains a MenuItem and if so, it removes it and adds a new MenuItem). The createNewOrder() method checks if the orders map doesn't contain a key the same as the one given as parameter, checks if all the MenuItems ordered exist in the menu. If so, a new order is inserted in the HashMap. The computePriceOrder() and generateBill() methods both check that there exist in the HashMap an order with the key equal with the one given as parameter. The generateBill() method calls the MyFileWriter's generateBill() method that generates a bill in the "bill.txt" file for an order using the computePriceOrder() method to compute the total price of the order.

- The Controller package contains the Controller class that has as fields every class of GUI present in the View package (they will be detailed below) an object of type Restaurant and one of type RestaurantSerializator. In the constructor, all the GUI classes are initialized having as Controller object this Controller, in that way all the GUI classes share it. Also, the Restaurant object is initialized by deserializing form a file. The methods of these class are similar to the ones from the Restaurant class but they are used for taking strings as parameters, checking if they are valid, instantiating a new object and then calling a method from the Restaurant(example: for adding a new menuItem, the parameters are a name, price, and list of strings that represent the ingredients. It is checked that the name is different than "" as well as the price, then the price is checked with the .matches(regex) method to see if it matches a positive double. After that, the list of ingredients is compared to null, if it is that means that the MenuItem we are about to create is BaseProduct. Else, it means that it is a CompositeProduct and then it is checked that all the ingredients in the list of ingredients are in the menu. If any of these checks fail, a message informing the user that there is an invalid menuItem is displayed. If not, a new BaseProduct/CompositeProduct object is created and it is passed as an argument to the restaurant's method createMenuItem(). The other methods work similar: they validate the strings and then create an object of appropriate type that is passed as an argument to a method of the restaurant object). Each of these methods, when successful, after calling the method from the Restaurant object, also call the writeObject() method of the RestaurantSerializator() object, which is equivalent to serializing the object.
- The View Package: contains five UI classes. One for the chef, one for the waiter, one for the administrator, one for the Composite Products' adding and editing and the initial one, that is visible at the launch of the application, that contains three buttons, administrator, waiter or chef that each take the user to the AdministratorGUI, WaiterGUI and ChefGUI respectively.
  The AdministratorGUI class contains multiple JButtons, JLabels, JTextFields and a JTable. It contains two JTextFields for the name and the price and buttons for adding, deleting, editing a MenuItems. Another two JTextFields are present (newName and newPrice) and they are not taken into consideration only when pressing the edit button and if they have something written in them (so if for example we want to edit the name we will write the new name in the newname JTextField and then we will press edit; if we want to edit both the price and the name, we will have to enter values in both these JTextFields).So, when editing we have to specify the name of the product that we want to edit and new values for it in the JTextFields dedicated for them. We also have a button for seeing the menu that, when pressed, shows a JTable with every menu item in the menu, their price, their type and their ingredients (if they are Composite). Another button is for adding/editing compositeItems that when clicked, takes the user to the CompositeItems user interface.

The CompositeItems UI contains a JTextField for the name and one for the newName, that is taken into consideration similar to how the newName JTextField from the AdministratorGUI is taken. If we want to add new Ingredients to the composite product that we want to create we follow the procedure: we press the Possible Ingredients button, then the menu will appear, we select the ingredient that we want to add, we click the Add Ingredient button and then in a JTextArea the ingredient will be shown. After we selected all the ingredients that we want to add to our Composite Product, we click on either the Add Composite Product button or on the Update one, depending on the operation that we want to perform.
The WaiterGUI also contains multiple button for creating order, computing price, seeing all orders, generating bill. What is important for this UI is the way in which MenuItems are added to an order. It is done in the following way: we click on the Show Menu button, the menu will appear, we select all of the items that we want to order, then we click the Add Items Selected button to add them and then we can click on the create order.
The ChefGUI contains a JTextArea where, when an order containing a CompositeItem is placed, a message informing the chef will be displayed. Also, the order will be added into a table of orders visible for the chef, containing the Order number (this is only for the chef, to see the order in which the orders were placed, it is different from OrderID). All the orders that a chef has to cook are visible is he clicks on the Show Orders button. The other button that can be clicked is: Finalized An Order; that is used by the chef to remove the first order from the table, meaning that it has been cooked.

5. Results

The Application, at launch, deserializes from a file named “restaurant.ser” an object of type Restaurant that contains several MenuItems in the menu. The menu is:



Name	Price	Type	Ingredients
paine	3.5	Base Product	-
lapte	5.0	Base Product	-
ulei	5.0	Base Product	-
cartofi prajiti	10.0	Composite Product	cartofi, ulei
cartofi	5.0	Base Product	-
oua	4.0	Base Product	-
sunca	7.55	Base Product	-
piept de pui	9.4	Base Product	-
paine cu ou	12.5	Composite Product	oua, paine, lapte
vita	25.0	Base Product	-
rucola	6.0	Base Product	-
parmezan	10.0	Base Product	-
bacon	7.5	Base Product	-
spaghete	5.0	Base Product	-
spaghete carbonara	26.5	Composite Product	oua, parmezan, bacon, sp...
rosii	4.5	Base Product	-
tagliata vita	41.0	Composite Product	vita, parmezan, rucola

We can add, delete, edit MenuItems, as well as add orders, generate bills and compute prices.
And if we want to compute a bill for an order with the orderID=1 the following will be an example:

```
BILL FOR THE ORDER NUMBER: 1
Table: 5
Date: 2020-05-07T11:03:34.458
Product ordered: cartofi prajiti        Price: 10.0
Product ordered: piept de pui   Price: 9.4
Product ordered: spaghete carbonara     Price: 26.5
Product ordered: tagliata vita  Price: 41.0

Total price to pay: 86.9
```

6. Conclusions

This assignment develops an application that simulates a Management System for a restaurant. Some further developments would be adding new features such as using the Observer design pattern for the chef to also notify thew waiter when an order has been cooked or the application to be extended to the ability of having more cooks, waiters and developing a register mechanism for them.

7. Bibliography

https://www.baeldung.com/java-serialization
http://javarevisited.blogspot.ro/2011/02/how-hashmap-works-in-java.html
http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html