|  |  |
| --- | --- |
| **Project title** | **RESTAURANT** |
| **Author(s)** | **Pop Vlad-Iulian-Ioan** |
| **Group** | **30421** |

1. **Task Description**

Consider implementing a restaurant management system. The system should have three types of users: administrator, waiter and chef. The administrator can add, delete and modify existing 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

**Secondary requirements**.

• Object-oriented programming design

• Classes with maximum 300 lines

• Methods with maximum 30 lines

• Java naming conventions

• Basic documentation

• Implement the class diagram from the homework specification. Choose wisely the appropriate data structures for saving the Orders and the MenuItems

• Graphical interface: o Window for Administrator operations: add new MenuItem, edit MenuItems, delete MenuItems, view all MenuItems in a table (JTable) o Window for Waiter operations: add new Order, view all Orders in a table (JTable), compute bill for an Order

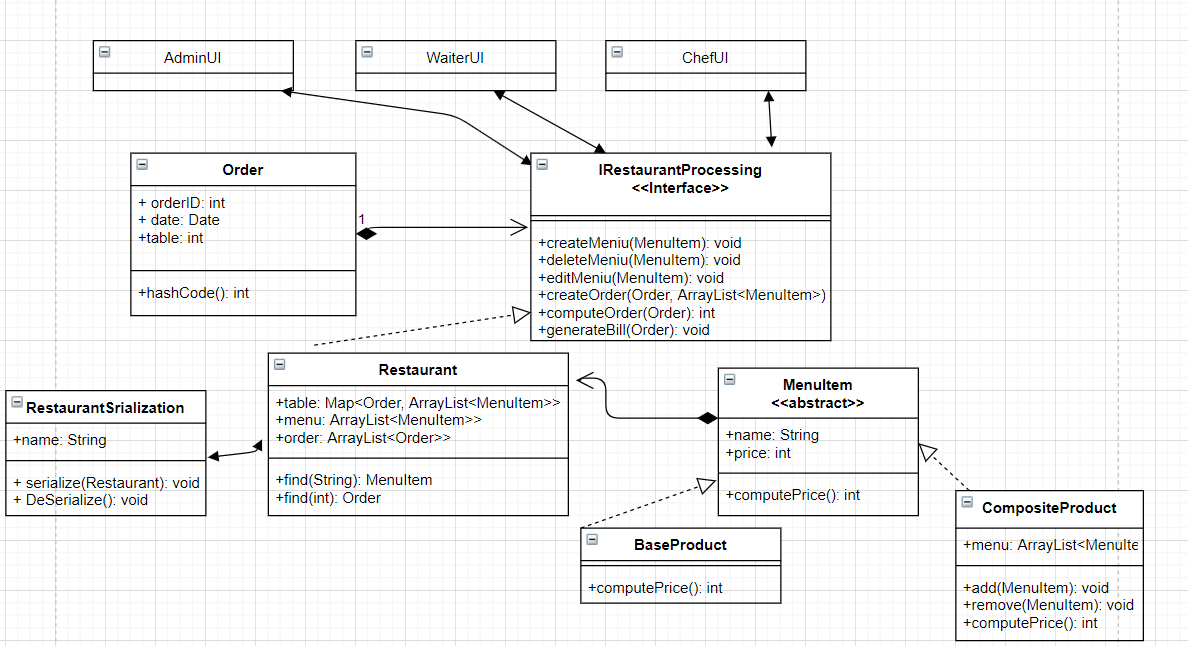
• jar file - the application should permit to be run with the following command: java -jar PT2020\_Group\_LastName\_FirstName\_Assignment\_4.jar

1. **Problem Analysis**

There can be 3 types of users for this application: waiter, admin, chef. Each of those users has its own meniu provided by the GUI. Firstly a preliminary menu will appear when starting the application from which the user can select the type of user that will work with it. The name of the file to which the Restauranta class is serialized and deserialized must be given as the first argument of the program. If the number of arguments is diffrent then one the program will print to the terminal „Wrong number of arguments” and nothig else will be done. The file given as argument is not mandatory to exist because if it does not the program will create it for you and will store there a new instance of the Restaurant class. The close button of all the 3 secondary menus will close the current menu leaving the main one or other menus opend.

* Use case: Chef will use the app
* Primary actor: Chef
* Main Success Scenario: The chef will get notified if there are any new orders
* Use case: Administrator will use the app
* Primary actor: Administrator
* Main Success Scenario: The administrator can create a new menu, modify the price of an existing menu, delete a menu or see all the menus that are available to use in a table format. The menu is edited or deleted based on the menu name. The menu will not be lost if the app is closed.
* Use case: Waiter will use the app
* Primary actor: Waiter
* Main Success Scenario: The waiter can create orders, generate a txt file representing the bill of an order or see all the orders that are still in the memory in a table format. The waiter must input an id that is unique for the current run of the app, if the app is closed the order set will be lost, the table number, press on the show menu button and select from the drop down list one or multiple menu items and the press on the create order button. The bill will be generate based on the id of the order so the field for the order id must contain the id of the order that the waiter wants to generate the bill and then press on the generate bill button. The bill will contain the id of the order, the table, date in which it was placed, the list of all the menus that the order has and in the final the total to be payed.

1. **Designing**



1. **Implementation**

**Order:**

This class represents holds the order information that the waiter will take from clients. It has 3 fields: orderID, date and table. The orderID is the number by which any order can be identified uniquely. The date is an object of the class Date and is stored automatically when the order object is created. The table field represents the table from which the waiter took the order. It has to constructor: one in which you must give as input, besides the orderID and the table number, the date as a Date object and one it which the date is taken as the current date. The class also overrides the methods equals and hashCode for the purpose of being the key in a hash table in which the actual menu the order is associated with will be stored.

**MenuItem:**

This class represents the products a costumer can order. It is an abstract class and it represents the component from the Composite Design Pattern. It has 2 fields name which is String and price which is an integer. It implements the toString method for obtaining the pretty print of the object.

The computePrice method returns the price of the menu.

**BaseProduct:**

This class extends the MenuItem class and it represents the leaf class of the Composite Design Pattern. It has a constructor that simply calls the super constructor and overrides the computePrice method by returning the price.

**CompositeProduct:**

This class extends the MenuItem class and it represents the composite class of the Composite Design Pattern. Its only field is an ArrayList of MenuItems which represents the composite product. The constructor of this class has as input the name of the CompositeProduct and an ArrayList of MenuItems and calls the super constructor with the name and 0 as input for the price.

The add method adds the MenuItem given as input in the ArrayList and computes the price field by calling the computePrice method overridden in this class.

The remove method just calls the remove method of the ArrayList.

The computePrice method does a foreach in which the price is computed by adding every price field of the menu list by calling the computePrice of the MenuItem class. The price total price is then returned as an integer.

The toString method is also overridden. It dies a foreach for every element in the menu list and adds to the string to be returned the retuned value from the toString method of the super class.

**IRestaurntProcessing:**

This class is an interface which will be implemented by the Restaurant class. Its methods are meant to represent all the activities the administrator can do, namely: create a menu, edit a menu and deleting a menu and all the activities the waiter can do, namely: create an order, compute the price of an order or generating a bill representing the order. The bill is a txt file with the name bill which will have written in it the order id, the table of the order, the date at which the order has been placed, the list of all the products the clients associated with this order ordered and their respective price and at the and the total sum the clients must pay for their order.

The interface also makes use of the Design by Contract Pattern. The main technic use for this pattern is the pre conditions technic by always verifying if the parameters of the methods to not be null, in the case of computing the price for an order the pre condition is that the given order actually exists in the hash table or in the case of editing a menu the condition that the price must be a positive integer and as a post condition the old price and the new price must be different. All those conditions being true will be asserted in the implementation of the function.

**Restaurant:**

This class implements the previously described class and also implements other methods that will help in solving the problem. It has 3 fields. The table field which is an instance of the structure Map<Order, ArrayList<MenuItem>> and represents the hash table for storing pairs of Order ArrayList<MenuItem> objects. The hashcode is generated based on the hashcode function overridden in the Order class so the order object is the key of the map. The menu field is an ArrayList<MenuItem> object which helps us keep track of all the menus a client can request in his order. The order field is ArrayList<Order> which helps us keep track of all the orders that the waiter created.

The addMenuItem method simply adds the MenuItem object received as input and puts it in the menu list.

The deleteMenuItem method simply removes the MenuItem object received as input from the menu list.

The editMenuItem method asserts if the MenuItem received as input is not null and if the new price is larger then 0, changes the price of the item and asserts if the old price is different than the new one.

The createOrderMethod is used for adding the order received as input to the order list and it also maps the order to the list of MenuItem objects received and stores them in the hash table. It also asserts if the input objects are not null.

The computeOrderPrice method computes the total price of the order received as input by getting the list of MenuItems from the hash table associated with it and iterating it.

The price is returned.

The generateBill method generates a file txt name bill that will represent the bill associated with the order object received as input. It makes use of the PrinterWriter class to print in the txt file.

The find method has two implementations. The firs one searches in the menu list for a menu that has the same name as the string given as input, it returns that MenuItem.

The second implementation searches for an order in the order list by the id and returns the order that has the same id as the input integer. If the object searched is not found the method returns null.

The getMenuOrder method returns a CompositeProduct object. The method creates a CompositeProduct and adds to its list all the MenuItems that match one of the names received as input through the list of strings. The list of strings is iterated and by using the find method the matches are found.

**RestaurantSerialization:**

This class is use for saving and retrieving the previous state of the restaurant. It has one field that represents the name of the file given as the single argument of the program.

The serialize method will save the restaurant state and print to the standard output the fact that the serialization succeeded.

The DeSerialize method will retrieve the previous state of the restaurant and if the file provided as the first argument does not exist print to the standard output: “class not found” and it will create the file and store in it a new Restaurant object and return it.

**MainUI:**

This class represent the GUI for the preliminary part of the GUI. It contains 3 buttons that will open one of the other 3 GUIs depending on the user. It has 3 fields matching the 3 other GUIs.

**AdminUI:**

This class represents the GUI for the administrator. It has two text boxes where the name and the price of a menu can be inserted. The editing or deleting of a menu is done by typing in the name box the name of the menu which has to be modified or deleted and pressing the corresponding button. If the name searched for is does not correspond to any existing menu then a pop up will appear saying that there is no such menu. Upon the pressing of the Show Menu button all the available menus are displayed in a JTable. The restaurant will be serialized after each pressing of a button besides the Show Menu button.

**WaiterUI:**

This class represents the GUI for the waiter. It has two text boxes for inserting the id of the order and the number of the table. The Show Menu button will make the drop down list populate with all the available menus. When pressing the add button the selected menu from the drop down list will be added to the order and it will be seen in the text area. After selecting all the menus by pressing the Create Order button the text area will be cleared. The bill will be generated based on the id that is in the id text box. To see the orders organized in a JTable the Show Orders must be pressed.

**ChefUI:**

This class represents the GUI for the chef. The chef will get notified when a new order is introduced.

1. **Testing**

The testing of the program can be done by interacting with the GUI of the program.

1. **Conclusions**

This program helped me get hands on experience with the Composite Design Pattern and the Design by Contract Pattern.

1. **Bibliography**

The pdf. Support that the laboratory teacher provided to us.

Multiple sites from the World Wide Web.