Assignment 4 Documentation

-Restaurant Management System-

1. **Homework objectives:**

Main objectives:

The task is to implement a restaurant management system, having 3 users: administrator, waiter and chef. The operation for each user are: the administrator can create, delete or edit a certain item from menu, the waiter can create a new order for a table, adding items from the menu, and also, generate the bill for an order, and the chef is notified each time the waiter creates a new order, and he must cook the food for that order.

Secondary objectives:

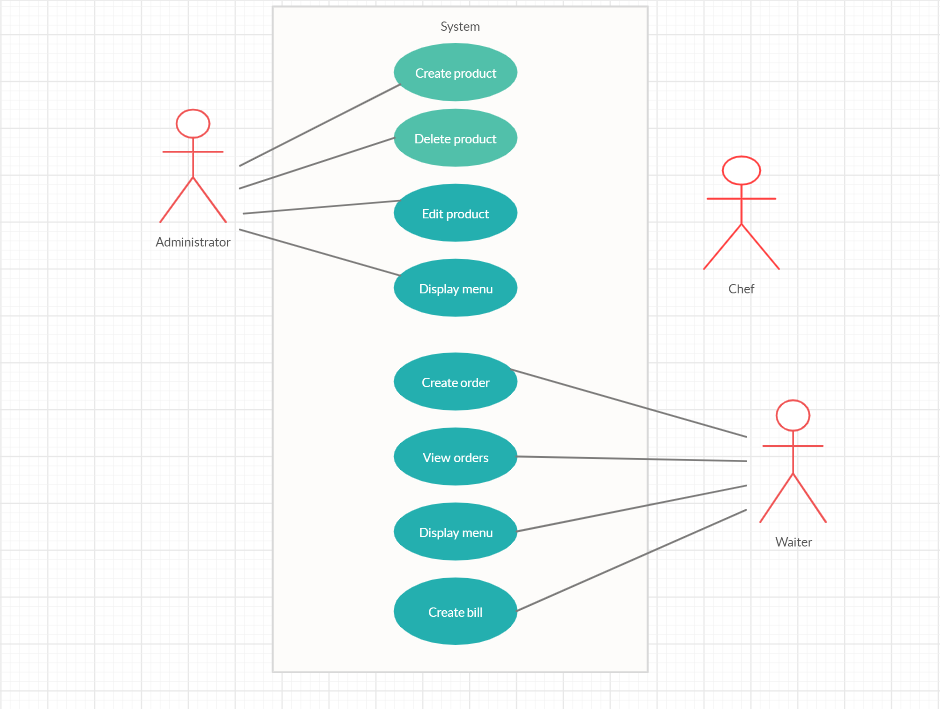
* We need to use Composite Design Pattern for defining the classes MenuItem, BaseProduct and CompositeProduct.
* We need to use Observer Design Pattern to notify the chef each time an order is created.
* The code should be implemented in the Object Oriented Programming design.
* We need to create a GUI for each user which implements their specific operations.
* The class Restaurant should be implemented using a predefined JCF collection which uses a hashtable datata structure.

1. **Problem analysis, modeling, scenarios, use cases:**

Problem analysis:

We need to implement a correct and efficient algorithm that should simulate the managing system for a restaurant having three main components: administrator, waiter and chef. The administrator is able to create a new menu product, to edit the price of a certain product, to delete any product from the menu and also to display all items from the menu. The waiter is able to display all items from the menu and select some products from it, with which he will create an order. Also, he is capable to view all orders created in the restaurant and to generate a bill in txt format for a certain order. The chef which will get a notification containing the products he has to deliver, every time the waiter creates a new order.

Use case:



Modeling:

The restaurant consists of orders and menu items. So, we have two main classes BaseProduct and CompositeProduct from which the items from the menu are created.

Scenarios:

There are two main scenarios: for the administrator and for the waiter.

The administrator user logs into the application:

1. If the administrator introduces the data needed for creating a new item, that item will be created, else an error in encountered “invalid input”.
2. The administrator successfully displays the menu.
3. If the administrator introduces the name of the product to be deleted, that item will be removed, else an error is encountered “invalid input”.
4. If the administrator introduces the price of the item and selects from the table the item which will be edited, the price will be changed, else an error is encountered “invalid input”.
5. The administrator successfully presses the “Back” button.

The waiter logs into the application:

1. The waiter successfully displays the menu.
2. If the waiter introduces the data needed for creating a new order, that order will be created, else an error in encountered “invalid input”.
3. The waiter successfully displays the order list.
4. If the waiter introduces the data needed for generating the bill, the bill will be created, else an error in encountered “invalid input”.
5. The waiter successfully presses the “Back” button.
6. **Design and implementation:**

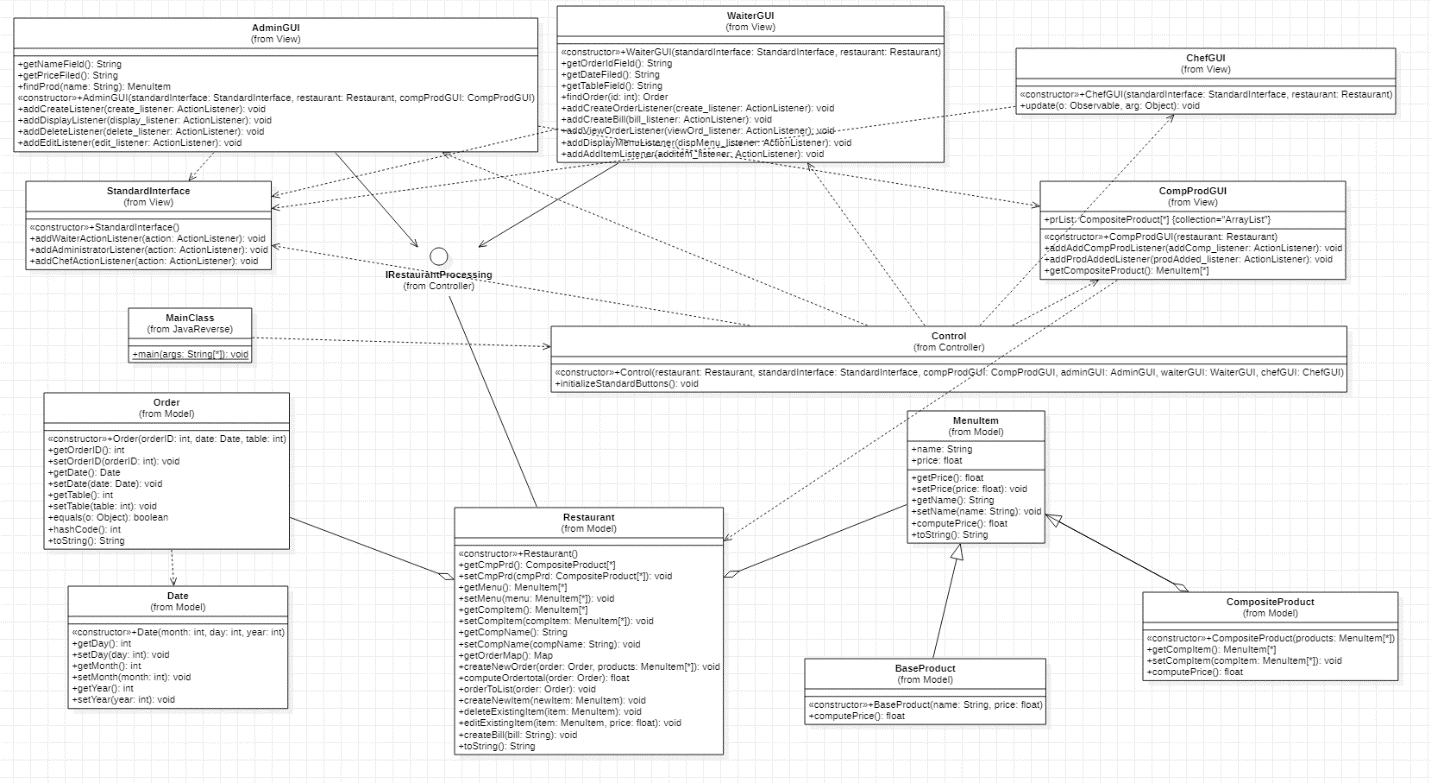
Data structures:

I chose to use ArrayList as data structure to store the products from the menu, because that kind of lists are useful for storing and accessing data. ArrayList internally uses dynamic array to store the elements. Also, for storing related information in the Restaurant class I used Map<Order, ArrayList<MenuItem>>, having the key formed of objects of type Order, for which the hashCode() will be overwritten to compute the hash value.

Packages:

In object-oriented programming, Model View Controller is a software design pattern used for developing user interfaces, which divides the related program logic into three interconnected elements. The MVC pattern is widely used in program development with programming languages such as Java, Smalltalk, C. and C++. Model represents the underlying, logical structure of data in a software application and the high-level class associated with it. The object model does not contain any information about the user interface. View is a collection of classes representing the elements in the user interface, such as buttons, display fields, and so on. Controller consists of the classes connecting the model and the view, and is used to communicate between classes in the model and the view.

UML diagram:



Class design:

I used Observer Design Pattern for Restaurant and chefGUI classes, and Composite Design Pattern for MneuItem, BaseProduct and CompositeProduct classes.

The design has the format:

* Model:
* BaseProduct – is a class that models the base product from the menu item, from which can be created other composite products.
* CompositeProduct – is a class that models more complex products, composed by other base or composed products. Basically, it is a list of others products, and the price for this sort of products is created by summing up the prices from the products it has been created.
* MenuItem – is a class that represents a single product from the restaurant’s menu. This product is characterized by its name and price.
* Order – this class represents the central part of the waiter’s job. Here are the data necessary for an order, such as order id, the date of the created order and the table’s number.
* Date – is a class that gets and sets the year, moth and day for computing a date
* Restaurant – this is the main class of the entire application. It implements the IRestaurantProcessing interface and all the methods from it, such as createNewOrder, computeOrdertotal, createBill, createNewItem, deleteExistingItem and editExistingItem.
* View:
* StandardInterface – represents the main GUI that allow the user to choose the interface for his job: AdminGUI, WaiterGUI or ChefGUI.
* AdminGUI – in this GUI, the administrator can execute one of his operation like create a new item, delete a certain item or just edit one. Also, he can display the entire menu in a JTable.
* CompProdGUI – this GUI will be visible only when the administrator wants to create a composite product. In this GUI, the administrator can display the existing items from the menu in a JTable and select those items which will compute the product he wants to create.
* WaiterGUI – in this GUI, the waiter can execute one of his operation, like displaying the whole menu in a JTable in order to select some items which will be included into an order, to create a new order by introducing the order id, the date and the table’s number, to generate a bill for a certain order and to display all orders in a JTable.
* ChefGUI – this GUI will be visible when the waiter creates a new order.
* Controller:
* Control – in this class, the buttons from the StandardInterface are initialized, and the environment of the application is all set up.
* IRestaurantProcessing – in this interface are the methods implemented in the Restaurant class and used in the AdminGUI and WaiterGUI classes.

Implementation:

* Model:

The classes BaseProduct and CompositeProduct extends the MenuItem class, and they are characterized by name and price. There are setters and getters and also a function for computing the price for each item. Since a composite product is composed of a list of menu items, its class has a function that computes its price by summing up the price of the components.

The restaurant is composed of two array lists, one for menu items and one for orders and a HashMap which is a data structure that implements an associative array data type, a structure that can map keys to values. A HashMap uses a hash function to compute an index, also called hash code. In our case, that function is implemented in the Order class. In class Restaurant, I implemented the functions needed for each user to execute his operations. The waiter has to create new order, add order to list and generate bill. The function for creating an order takes as parameters the order to be created and the corresponding items from the menu. Here we put that order into the map using orderMap.put().

public void createNewOrder(Order order, ArrayList<MenuItem> products){  
 assert (order != null && products != null);  
 Order oldOrder = order;  
 orderMap.put(order, products);  
 assert oldOrder.equals(order);  
}

The function for adding an order to the list of orders simply adds the new order to the list. Also, here is computed the string which will notify the chef that a new order is created, showing him the products that he needs to prepare.

public void orderToList(Order order){  
 ordersList.add(order);  
 Map<Order, ArrayList<MenuItem>> map = getOrderMap();  
 ArrayList<MenuItem> itemsList = map.get(order);  
 Iterator<MenuItem> iterator = itemsList.iterator();  
 StringBuilder st = new StringBuilder();  
 st.append("New order is created\nOrder id: " + order.getOrderID() + "\n");  
 st.append("Items ordered: ");  
 while (iterator.hasNext()){  
 MenuItem item = iterator.next();  
 st.append(item.toString() + ", ");  
 }  
  
 setChanged();  
 notifyObservers(st.toString());  
}

The function for creating the bill of a certain order, takes as parameter a string, which will be the text we want to print in the bill. Here is created de buffer writer which will write the string into the text file.

public void createBill(String bill){  
 assert bill != null;  
 try {  
 BufferedWriter buff = new BufferedWriter(new OutputStreamWriter(new FileOutputStream("bill.txt")));  
 buff.write(bill);  
 buff.close();  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

When the bill is created, we need to find the total price of the order, so the method used for computing the price iterates through the order’s items to get the price of every product and to compute the total price after that.

public float computeOrdertotal (Order order){  
 assert order != null;  
 float price = 0;  
 if (this.ordersList.contains(order) == true){  
 ArrayList<MenuItem> menuItems = orderMap.get(order);  
 Iterator<MenuItem> iterator = menuItems.iterator();  
 while (iterator.hasNext()){  
 MenuItem item = iterator.next();  
 price = price + item.getPrice();  
 }  
 }  
 return price;  
}

The administrator has to create a new item menu, to delete a certain item or to edit one. The function used for creating a new item, simply takes that item as parameter and introduces it to the list of items of the menu. The function for deleting an item works on the same principle, but instead of adding an item to the list, that item is removed.

public void createNewItem (MenuItem newItem){  
 assert newItem != null;  
 int preSize = menu.size();  
 menu.add(newItem);  
 int postSize = menu.size();  
 assert preSize + 1 == postSize;  
}

For editing an item, we need to iterate through the list of menu items and find the name of the item to be edited. We set its price and after that, we put it again into the list, in its initial position, using set.

public void editExistingItem (MenuItem item, float price){  
 assert (item != null && price != 0);  
 float prePrice = 0;  
 float postPrice = item.getPrice();  
 String itemName = item.getName();  
 Iterator<MenuItem> myIterator = menu.iterator();  
 while (myIterator.hasNext()) {  
 MenuItem curentItem = myIterator.next();  
 if (curentItem.getName().equals(itemName)) {  
 int index = menu.indexOf(curentItem);  
 curentItem.setPrice(price);  
 prePrice = curentItem.getPrice();  
 menu.set(index, curentItem);  
 }  
 }  
  
 assert prePrice == postPrice;  
}

* View:

In this package the GUI is created. There are 4 panels, the first one is the main panel from which the user can choose where to go next. Here are three button, representing the three users. If the button “Administrator” is pushed, a new panel will be visible. This is the AdminGUI panel. Here are four buttons representing the operations that can be implemented in this GUI. Also, here are two text fields, where the data is introduced by the user. If the user wants to create a product, he needs to write the data first. If the user wants to add a base product, he needs to introduce both the name and the price of the product and then press “Create product” and in the restaurant will be created a new object of type base product which will be added to the menu list.

newPrice = Float.*parseFloat*(price);  
BaseProduct newProd = new BaseProduct(name, newPrice);  
restaurant.createNewItem(newProd);

If the user wants to add a composite product, he needs to introduce only the name of that product and then press “Create button”. A new panel, CompProdGUI will be visible. Here, the user computes the new product. Press the button “Display menu items” in order to display the menu existing till that moment in the restaurant. The menu will be shown into a JTable. The user needs to select the items from the table, press “Add item” to add that item to the list of the products needs to create the new one. When the selection is done, press “Create product” to create the new composite product. For deleting an existing element, the user must write the name of the product to be deleted and then press “Delete product”. If it was a base product that was deleted, the composite products containing that base product will be deleted too. This is done by iterating through the list of composite products, and for each product we need to iterate through its list to check the base products. If we find a product with the same name as the deleted one, we iterate through the menu items’ list for deleting that composite product.

ArrayList<MenuItem> menu = restaurant.getMenu();  
ArrayList<CompositeProduct> cmpList = restaurant.getCmpPrd();  
Iterator<CompositeProduct> itC = cmpList.iterator();  
while (itC.hasNext()) {  
 CompositeProduct newP = itC.next();  
 newP.setCompItem(compProdGUI.getCompositeProduct());  
 ArrayList<MenuItem> composition = newP.getCompItem();  
 Iterator<MenuItem> itP = composition.iterator();  
 while(itP.hasNext()){  
 MenuItem item = itP.next();  
 if(item.getName().equals(name)){  
 Iterator<MenuItem> itMenu = menu.iterator();  
 while (itMenu.hasNext()){  
 MenuItem meniu = itMenu.next();  
 if(meniu.getName().compareTo(newP.getName()) == 0)  
 itMenu.remove();  
 }  
 }  
 }  
}  
restaurant.deleteExistingItem(findProd(name));

For editing the price of a product, the user needs to select the product from the table and write the new price in the text field and then press “Edit product”. Same as deletion, if a base product changes, the composite product containing that base product, changes too. Here we also need to iterate through the list of composite products and through the list of each product. When the base product is found, we set its new price and compute again the price for the initial product. After that we set the product into the menu list. Last but not least, if the button for “Display menu” is pressed, the menu will be displayed through a table. Going further to the waiter operations by pressing “Waiter” button from the standard interface. Here are five buttons representing the waiter’s operations and three text field for introducing the data. When he presses the button “Display menu”, the menu will be displayed in a table and he can select from there the items for an order by selecting the items in the table, then clicking “Add item”. For creating a new order, the specific data of the order must be introduced and after that he needs to press “Create order”.

int id = Integer.*parseInt*(getOrderIdField());  
int month = Integer.*parseInt*(d.nextToken());  
int day = Integer.*parseInt*(d.nextToken());  
int year = Integer.*parseInt*(d.nextToken());  
 int tableNr = Integer.*parseInt*(getTableField());  
 Date newDate = new Date(month, day, year);  
 Order newOrder = new Order(id, newDate, tableNr);  
 Iterator<MenuItem> iterator = menuProducts.iterator();  
 while (iterator.hasNext()) {  
 MenuItem item = iterator.next();  
 selectedProd.add(item);  
 }  
 restaurant.createNewOrder(newOrder, selectedProd);  
 menuProducts.removeAll(menuProducts);  
 restaurant.orderToList(newOrder);  
 ordersList.add(newOrder);

For creating the bill, the waiter has to create the order first. Then, he has to write the id for the order he wants to generate a bill.txt. That order is found in the list of orders using a function that finds it by its id. Then, the text we want to print in the bill is computed. For viewing all order, he needs to press “View orders” button, and it works on the same manner as the displaying of the menu. Last user is the chef. The ChefGUI panel will be visible each time the waiter creates a new order. This contains a notification for the new order and the items of which it is composed.

1. **Conclusions:**

This project was a good opportunity for me to work more with the graphical user interface and getting familiar with more elements and methods for creating a GUI. Also, through this project I managed to learn more about OOP design style and to get familiar with new design patterns such as Observer and Composite.

1. **Bibliography:**

<https://www.baeldung.com/java-hashcode>

<https://www.w3schools.com/java/java_hashmap.asp>

<https://www.geeksforgeeks.org/composite-design-pattern/>

<https://www.tutorialspoint.com/design_pattern/observer_pattern.htm>