Homework4 – Fundamental Programming Techniques

*Ramona-Alexandra Nechita*

**Contents**

1. General overview on task

2. Strategy and Logic

3.Project Diagram

4.Classes used explained

5.Brief description of main methods

6. GUI

7.Possible future improvements

8. User cases

9. general conclusion

**1.Task**

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 ordered through a waiter.

1. Define the interface RestaurantProcessing containing the main operations that can be executed by the waiter or the administrator, as follows:

 Administrator: create new menu item, delete menu item, edit menu item  Waiter: create new order; compute price for an order; generate bill in .txt format.

2. Define and implement the classes from the class diagram shown above:  Use the Composite Design Pattern for defining the classes MenuItem, BaseProduct and CompositeProduct

 Use the Observer Design Pattern to notify the chef each time a new order containing a composite product is added.

3. Implement the class Restaurant using a predefined JCF collection which uses a hashtable data structure. The hashtable key will be generated based on the class Order, which can have associated several MenuItems. Use JTable to display Restaurant related information.

 Define a structure of type Map> for storing the order related information in the Restaurant class. The key of the Map will be formed of objects of type Order, for which the hashCode() method will be overwritten to compute the hash value within the Map from the attributes of the Order (OrderID, date, etc.)

 Define a structure of type Collection which will save the menu of the restaurant. Choose the appropriate collection type for your implementation.  Define a method of type “well formed” for the class Restaurant.

 Implement the class using Design by Contract method (involving pre, post conditions, invariants, and assertions).

4. The menu items for populating the Restaurant object will be loaded/saved from/to a file using Serialization.

**2.Strategy and Logic**

The strategy was pretty straightforward considering the diagram was given in the task. The restaurant has a menu, an administrator, a waiter and a chef.

For the menu : It was read from/into a file via serialization. The administrator window handles changes such as adding, editing or deleting items from the menu. The waiter has access to the menu only to select the items for a new order.

Administrator : he can create new menu items, delete old ones, edit items. The items are stored in a list and displayed on the administrator window as a JTable.

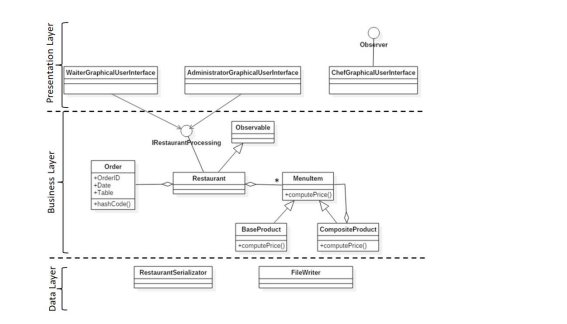
Waiter : he handles the orders and billing. He can create a new order and select the items from the menu which are supposed to be in the order, he can compute the total due of the order and it is displayed in an non-editable text field. The orders are stored in a HashMap but they are displayed in waiter’s window in a JTable as well. He can also generate the bill for a selected order, which will also be generated in a txt format in the source folder.

The menu items are created with respect to the composite design pattern. The classes BaseProduct and CompositeProduct are containing the information regarding the product and they both implement the MenuItem interface. The MenuItem interface contains methods such as getName, getPrice etc but most importantly it contains the computePrice method, which both classes need to have. If an item in the menu is a base product, there is not much to say about it and it has a fixed price, but if it is a composite product, it contains a list of other menu items being in its composition and its price is the total sum of the prices of its parts. When using the list with the menu in the Restaurant class, the elements in the list are MenuItem objects, because only a MenuItem can be either a base or a composite product. When dealing with a menu item, appropriate casts and methods are used, alongside with the getClass method.

The orders are stored in a HashMap in which the object Order is the key and at that key there is a list of menu items constituting that specific order. The method hashCode() was overloaded in the Order class in order to provide better hashing and avoid collisions(by double hashing).

**3.Project Diagram**

The project diagram was already given in the task and therefore I did not add much to it. However I did change the MenuItem class in an interface because it came in handy. It could have been an abstract class as well, without any difficulties but I preferred to use it as an interface.



The classes are divided in 3 architectural layers: data layer, business layer, presentation layer. The data layer contains the classes for accessing, getting and sending data. In the business layer there are the classes with the overall backend logic, the whole strategy.

**4.Classes used**

CompositeProduct, BaseProduct, MenuItem – the classes making the composite design pattern. They all have the computePrice method for finding out their price.

Filewriter – this class contains only a static method using the PrintWriter to create the bills. It only gets the price as a parameter and generates the bill with a standard text.

Restaurant Serialization – contains the two static methods to read the file as on Object input stream one by one and use it and another one to write an object when a new item is added to the menu.

Order – the data class, it is used as key in the hashMap. It only contains the orderID, table and Date and overrides the methods hashCode() and equals().

The GUI classes are pretty straightforward, the only special thing is at the Chef Window whose class implements the Observer interface, in order to compose the Observable design pattern. The chef is notified every time a new order is created so that he can cook it.

Restaurant- main class. It implements the RestaurantProcessing interface which contains the methods for the administrator operations and waiter operations.

**5.Brief description of main methods**

One of the most important methods in my project is the method for creating a new menu item. I pay extensive attention to the classes and casts because in my list I can have either a base product or a composite product and their all listed as menu items. After the item is added in restaurant’s menu it also needs to be serialized and saved to the menu file. The methods with edit item and delete item have quite similar approaches and use almost the same things. The deleteItem() method is presented below as a relevant example :

public void deleteMenuItem(MenuItem m) {

for(Iterator<MenuItem> iter = menu.iterator(); iter.hasNext();){

MenuItem item = iter.next();

if(item.getClass().toString().equals("class bussinessLayer.BaseProduct") && m.getClass().toString().equals("class bussinessLayer.BaseProduct")){

if(((BaseProduct)item).price == ((BaseProduct)m).price && ((BaseProduct)item).name.equals(((BaseProduct)m).name) && ((BaseProduct)item).description.equals(((BaseProduct)m).description)){

iter.remove();

}

}

else if(item.getClass().toString().equals("class bussinessLayer.CompositeProduct") && m.getClass().toString().equals("class bussinessLayer.CompositeProduct")){

if(((CompositeProduct)item).price == ((CompositeProduct)m).price && ((CompositeProduct)item).name.equals(((CompositeProduct)m).name) && ((CompositeProduct)item).description.equals(((CompositeProduct)m).description)){

iter.remove();

}

}

}

}

Other useful and important methods are those for serialization which were briefly discussed in the class section. The getMenu() method is the more complex one because it involves reading object by object until the end of the file, as presented below :

public static List<MenuItem> getMenu() throws IOException,ClassNotFoundException {

FileInputStream fi = new FileInputStream(new File("menu.txt"));

ObjectInputStream oi = new ObjectInputStream(fi);

List<MenuItem> theMenu = new ArrayList<MenuItem>();

while(fi.available() > 0) {

theMenu.add((MenuItem) oi.readObject());

}

oi.close();

fi.close();

return theMenu;

}

**A a a a a a a**

**6.GUI**

The graphic user interfaces are pretty simple and clean. Both the waiter and the administrator have a jtable with the orders and menu respectively. There are 3 buttons in each of those windows and the waiter also has a non-editable text field with the price of a selected order.

The “observable” window of the chef is notified whenever a new order is created. It only contains a jlist which receives a new item whenever an order is added.

**7.Possible future improvements**

The program made by me meets all of the requirements and is functional. However, there are a fair number of improvements that could be done to it. First of all, the gui could be beautified because it is plain and not that well formed.

Moreover, in the GUI, when creating a new menu item it is assumed that it will be a composite product and that all the base product are in the menu from the very beginning. This is not a fair assumption and it limits the user to creating only composite products and not being able to add composites to a newly created item.

Another useful improvement would be to estimate cooking time and make orders disappear from the jlist as the chef finishes cooking them.

A great and extremely useful improvement which would be rather time consuming would be to connect a database to the project and add a login system so that the restaurant can have more than one waiter, more than one chef and more than one administrator in the same time and all of them could be able to use the application separately without interfering with each other.

**8.User cases**

In case the user is an administrator :

* Creating a new menu item, with desired name, price and description. ~ mention : the item is considered to be composite ~
* Editing an existing item : the name of the item is considered canon to the menu, because changing the name would mean bringing a new product in the menu. Therefore, it makes sense only to change the price or the description. Otherwise, the item needs to be deleted and re-created.
* Deleting an item; there are many cases when a menu item is no longer needed or is redundant to the whole menu and no longer fulfills its purpose in the app. Therefore, only the administrator can delete an item. In order to delete the item it needs to be fully specified, which means he needs to type the name, price and description of the soon-to-be deleted item

In case the user is a waiter:

* Creating a new order and adding it to the jtable. Th order is created by selecting the desired items from a jlist and then clicking the button “create” when he/she is done
* Computing the price of a selected order. In case the waiter, administrator or even a client wishes to know the total due of an order all the waiter has to do is to select the specific order from the jtable and just click the button “ compute price” and the price is displayed in the text field below
* Generating the bill in a .txt format when the client desires to leave

In case the user is a chef :

* See and be notified at each order

**9. Conclusions**

This homework brought me a better understanding of the design patterns and was a full scale application with many useful features. From the serialization to the slightly more complex gui, this homework takes us through all the steps of the development of a small scale application. Of course, there is a series of improvements that could be done but overall it has all the elements to be considered fully functional.

In conclusion, it is a complete application with a clean and user-friendly gui, which can be used to ease the process of managing a restaurant.

Lo dc djdbhf cool imi c r e s t e w o r d c o u n t u l f a r a s a f a c n I mxuc still n u c red ca o sa d I e d e stu l sa u mp l u c e le 4 0 0 d de c u v I n t e r a m a s e b u t oh w e l l w ha t c a n I do