Programming techniques

*Homework 4*

Moldovan Horia Andrei

Group 30424

Problem Specification :

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. To simplify the application you may assume that the system is used by only one administrator, one waiter and one chef, and there is no need of a login process. Solve the following:

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.

Requirements :

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 • Documentation

Use Composite Design Pattern for modelling the classes MenuItem, BaseProduct, CompositeProduct. Create bill in .TXT format. Design by contract: preconditions and postconditions in the RestaurantProcessing interface. Implement them in the Restaurant class using the assert instruction. Define an invariant for the class Restaurant. Window for Chef user: use Observer Design Pattern to notify each time a new Order is added. Save the information from the Restaurant class in a file using serialization. Load the information when the application starts.

Problem Analysis :

In order to design an application which fulfills the above mentioned description we have to understand the main role of each one of the three users. Let’s first talk about the administrator of the server. This user will have the main role to create two types of items and add them to the menu. He could create either a BaseProduct which is a product composed of nothing more than himself, or he could also create a CompositeProduct which is an item containing two or more BaseProducts or he can also create a CompositeProduct containing another previously created CompositeProduct. Furthermore the administrator will be able to delete or edit an already created product.

Next let’s discuss about the function of the waiter in our application. He will be the one who creates orders for the clients, more precisely for a table. When creating the order he has to choose the number of the table and the previously by the administrator user created items, be them either composite or base.

Lastly, the chef will have the role to receive the, by the waiter, created orders and to ultimately cook the food.

Problem Approach :

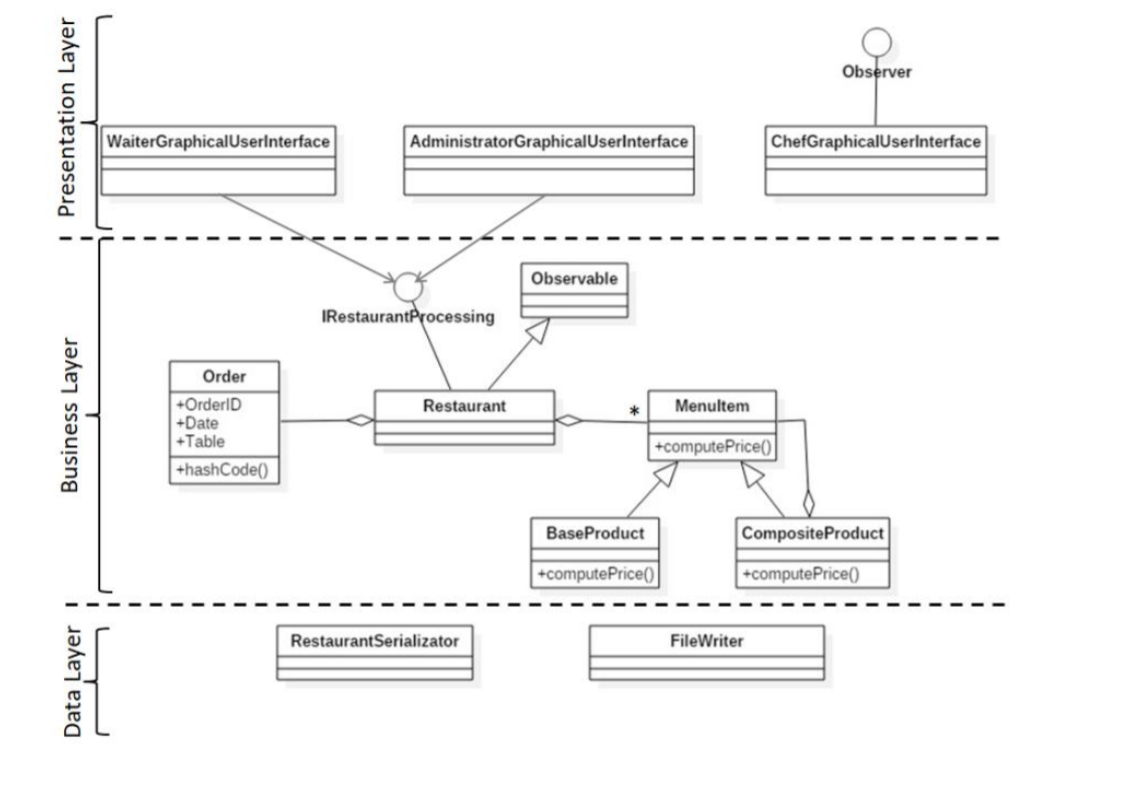
In order to put all the pieces together we will need a class which will help us to manage the duty of the three type of users. This class will contain all the main methods to be implemented by the waiter and the administrator. Each of the three users will have it’s own user interface where they will be able to complete their tasks. The restaurant will contain a list of all the created menu items and a map. The role of this HashMap is to map each list of ordered items to their specific order, so this field will contain a list of orders, each one with its own ordered items. The implementation of the HashMap will take place in the Order class, where we will create two specific methods, namely the one generating the hash code in order to know where in the map to place the mapping and another method which will override the equals method in order for us to be able to compare two orders and to tell if they are equal or not. We talked about a menu item being able to take two different forms. In order to do that we will make use of the concept of polymorphism, making use of the composite design pattern. This works as it follows: we create an abstract class which will be the general type of the further created subclasses, in our case MenuItem is it abstract class and CompositeProduct and BaseProdcut are the subclasses which extend the abstract class. By doing this we can refer generally to a composite product or to a base product as to an item from a menu, not as two different things.

Another important thing is serializing the information. What this means is the fact that we want our created data, so for example a created menu item, to be stored somewhere so next time we will open the application the data will still remain instead of getting lost.

User case :

As we already said we will have three different users: Administrator, Waiter and Chef. The chef will be able only to see changes made by the other two users, he won’t be able to make any modifications. The administrator and the waiter on the other side will have the possibility to directly interact with the applications by entering data in the text fields and pressing buttons in order to execute a task. They should be careful though when entering data because if the application expects an integer they should not introduce a string of characters, for example if the administrator wants to introduce or edit the price of an item he should not introduce a string, but an integer.

UML :



Implementation :

We will now take a closer look at the role and the implementation of each class with its attributes, constructors and methods.

***MenuItem.java***

This class is the one which will technically incorporate the two menu types: composite and base, so that’s why it is declared as abstract. Its fields are id, name and weight, representing the common fields in its child classes(BaseProduct and CompositeProduct). The fields are declared as protected so they can be seen in the child classes as well. The only method declared as abstract is the computePrice method because this one must be implemented by both subclasses.

***BaseProduct.java***

This is one of the two classed which extends MenuItem. It represents a simple menu item having the same fields as its superclass and another one for holding the price.

***CompositeProduct.java***

This is the second class extending the MenuItem class and it’s a bit more complex than the one described above because besides the fields existing in the BaseProduct class this one has an additional one which holds a list of MenuItems. We have said before that a composite product can contain an already created composite product and now we see why. It is because this composite product is also a menu item and if it consists of an array of items from the menu, that means that it could also contain a composite product. The functionality of the methods of this class is the same with the one from the base product just with a slightly different implementation. For example the method created for computing the price will have to iterate through the array of menu items and add each of their prices.

***Order.java***

This class is where we keep data concerning the information about the order. We have three private fields, the id of the order, orderID, the date when the order was processed, and the number of the table at which the order was made. The constructor of this class takes as parameter two integers and a date and creates a new order object using them. Here is also the place where we create the methods used by the mapping function, namely hashCode and equals. Their functionality was already described above.

***RestaurantProcessing.java***

This is an interface made to be implemented by the Restaurant class. Here were declared all the methods which must be implemented by the Restaurant describing all the capabilities of the users: creating a new menu item, deleting a menu item, editing a menu item, creating a new order, compute the bill of the order and generating a bill for the order.

***RestaurantSerialization.java***

This is another class in a strong relation with the Restaurant class because it contains two methods which are needed if we want to serialize our data. The concept of serialization was described above. The method serializeObject takes as parameter an object, a restaurant object in our case, and it writes the data kept in the object in a text file. The method deserializeObject will be used to create a new object, a restaurant object here, and this new object will contain the previously saved data in the file mentioned in the serialization method.

***Restaurant.java***

This is the class where the created menu items are kept, in a list having as generic parameter a MenuItem, and this is also the place where the HashMap is implemented, and used in order to map each array of ordered items to its specific order. The class Restaurant implements the two interfaces described, with their functionalities, above. That is why the methods in this class are the one declared in the RestaurantProcessing interface, but overridden with their specific function. For example, the chreateMenuItem method will add a new item to the list of menu items declared in the field of this class. This method has as parameter a list of ingredients in order for it to know whether the menu item to be added is a composite one or a base product. If this list is null then the product is a simple one(base), but if it is not, then the product item is a composite one. Another important method is the createNewOrder. This method will take as parameter the needed fields to create an order and a list of menu items and what it does is the fact that it will use the in the field of this class declared HashMap and its method put in order to connect the just created order with the given parameters and the arraylist of items.

***AdministratorGUI.java, WaiterGUI.java, ChefGUI.java***

This classes have the purpose to create the graphical user interface. Here will be declared the labels, the text fields, the tables, the buttons and all other needed components which are required for the frame of each user. As before, all this components will be declared private, for protection purposes, and they will be accessed only by using getter and setter methods. The other methods declared in this classes are the one used in the controller class. We have two type of methods: the first is the one which creates an action listener for each button declared as field in the class, and the remaining methods are used In order to describe the functionality of the buttons.

***Controller.java***

In this class we implement all the actions performed by each button created as field in the view classes. We make use also of the methods declared in that classes in order to properly create the functionality of each button.

***Conclusions***

I consider that this homework was the perfect opportunity for me to learn two new design patters and how the hashMap work in java. I also used a new concept called serialization which seem to me a very interesting and useful concept. This was the most challenging homework so far but also the most satisfying to complete.