Assignment 4

Group 30421

Buzan Vlad-Sebastian

1. Scope of the project

The main purpose of this assignment was to build a Java application that would serve as a working interface for a restaurant. The application offers three separate user interfaces, one for the waiter, one for the administrator and one for the chef. The administrator has the ability to add, remove or modify the products offered by the restaurant's menu. The waiter interface offers one the ability to create orders and to generate bills based on already existing orders. The chef interface simply displays the products ordered for each table.

The secondary objectives (which have been derived from the main objective) are:

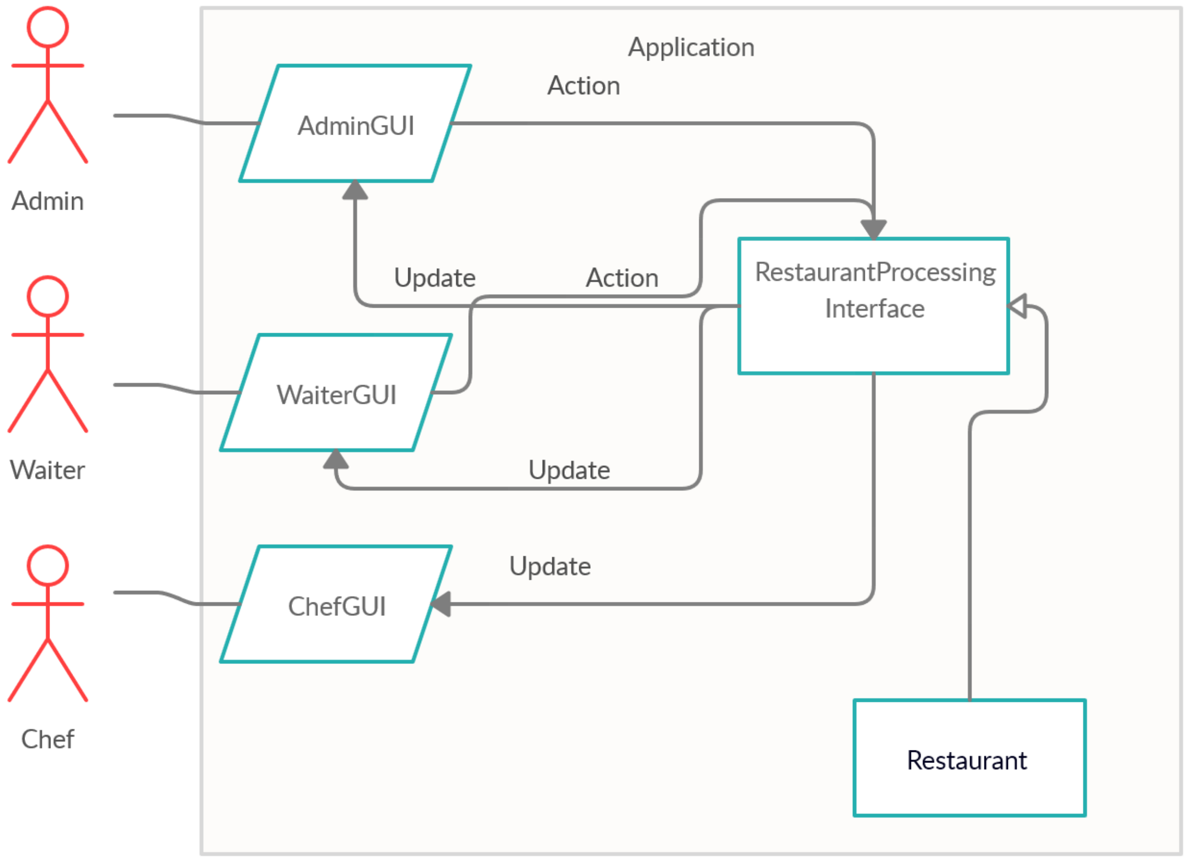
- Providing three separate user interfaces for the different types of users
- Providing each user with access to the restaurant's resources
- Designing data structures representing the restaurant, the menu items and the orders
- Providing an interface for the restaurant object, allowing it to communicate with the graphical user interfaces
- Creating an order generator whose task is to create a txt file containing billing information
- Using serialization in order to store the state of the application

2. Problem analysis, modelling, uses-cases

Firstly, one can deduce some of the classes that need to be designed in order to achieve the main goal. The input is acquired using graphical user interfaces. Since all the GUIs need to communicate with the resources contained in the restaurant, an interface called IRestaurantProcessing was provided. This way, for different structures of the restaurant, the GUIs can communicate with the given restaurant as long as the Restaurant class implements the methods contained in the interface. The methods in the IRestaurantProcessing provide information about the resources of the restaurant, as well as the option to insert products or orders.

As for the data structures used to represent an order and a product provided by the restaurant menu, the MenuItem and the Order class were designed. Since a restaurant can provided simple products (such as soda, French fries, garnishes) or more complex products (menus that include sodas and/or garnishes), the MenuItem class is extended by two classes, namely the BaseProduct class and the CompositeProduct class.

Since, the goal was to create an application which can be used by three different users at the same time, when it comes to use-cases, we have three actors (the users).



The general case for when an user interacts with the application is :

- The user inserts or alters data from the Restaurant class
- The action is first validated
    - Error pop-up warns the user about the error
- Action is valid and the change is applied
- User GUI is updated with the new values

3. Design choices, data structures, class design, packages and algorithms

For this assignment, I went for the “layered architecture” design pattern, as it provided to be a way to provide abstraction between the different “layers” of the application.

The four main layers / packages designed are:

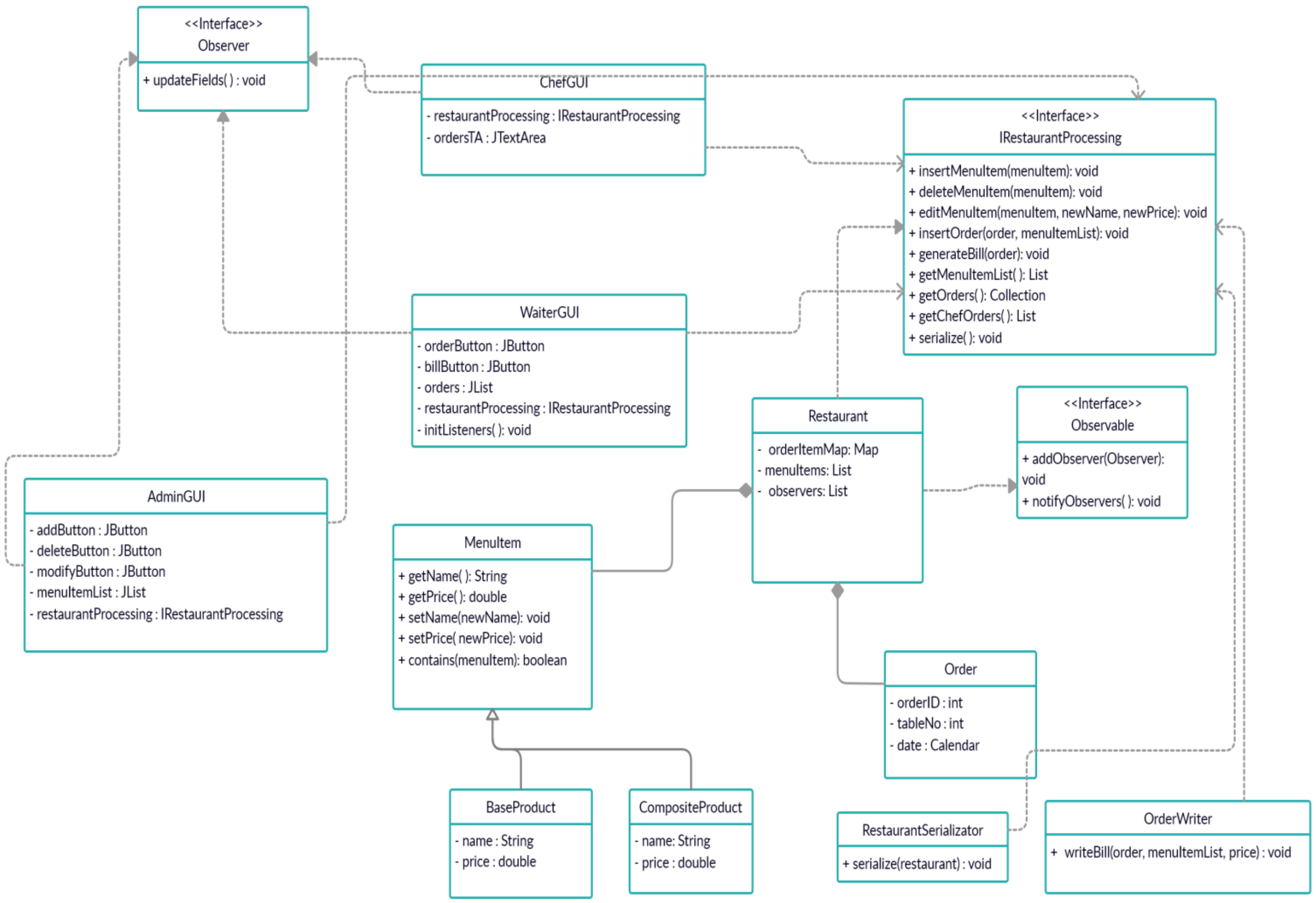
The business layer which handles the data structures and the algorithms used to manage the products and the orders. The more important classes in this package are the MenuItem class, which represent a product present in the restaurant menu, the order class, used to represent an order having an id and a table number corresponding to the table the order is supposed to be delivered to and the Restaurant class, used to manage and process all the before mentioned data.

The data layer package is mainly used to create files, including the restaurant.ser file and the bill .txt files. The two classes designed within this package are OrderWriter which is used to generate a bill text file containing all the prices of the ordered products as well as the total price, and the RestaurantSerializator which is used to serialize the Restaurant object. Whenever the application is closed, the serialize() method is called in order to preserve the state of the application for the next time it will be open.

The presentation layer package is the layer which handles everything related to graphical user interfaces. The classes designed within this package are: AdministratorGraphicalUserInterface (the UI for the admin user), ChefGraphicalUserInterface (the UI for the chef), WaiterGraphicalUserInterface (the UI for the waiter), AddItemWindow (window used for inserting products, allows the creation of composite products by selecting from a list products those one wishes to constitute the composite product), AddOrderWindow (window used when inserting an order, allows the selection of multiple products) and ModifyWindow (used for modifying an product).

One of the main challenges was creating an abstraction for the composite products and the base products (meaning, we want to treat them equally). This was accomplished using the composite design pattern. This way, the MenuItem abstract class was created, offering an interface for an product, handling it whether it is composite or not. This way, a hierarchy was created, for example a composite product can be made out of another composite product and a base product. The classes extending the MenuItem abstract class are BaseProduct (name, price) and CompositeProduct(name, price, list of products).

Another important aspect when designing the application was getting the graphical user interfaces to update whenever a change was detected in the data held by the Restaurant class. The approach I took was using the Observer design pattern. What this implies is that there are two types of classes who can interact with one another: Observers and Observables. Each Observable has a list of Obvservers associated to it. Whenever a certain change in the Observable object occurs, all the Observers associated to that Observable object will be notified. In my case, this approach was implemented using interfaces. The Observable class is the Restaurant class, meaning that whenever an action that changes the data (be it orders or products) occurs, the Restaurant notifies it’s Observers, those being the graphical user interfaces.



4. Implementation

The main use of this application is that of manipulating data, therefore the algorithms used in this project are all quite minimalistic and simple. The main operations the three types of users can perform are: inserting products, removing products, modifying products, inserting orders, creating bills and querying orders.

One of the operations that raised a problem was that of removing a product. Since there are two categories of products: composite and base, the product that the user wishes to remove can be part of a composite product. To address this problem, what I did was a recursive search inside each composite product, so that if any of the products included in the composite product contain the item the user wishes to delete, the whole composite product is removed.

The graphical user interface classes communicate with the data in the Restaurant class using the IRestaurantProcessing interface. The interface allows for all the operations one may need to use in order to manipulate the data in the restaurant. This way, when a button is pressed and for example a method that inserts a product is called, the GUI class creates the object to be inserted and then calls the method from the interface to insert it. Inside that method declared in the interface and defined in the Restaurant class, the parameter is validated, and then used to perform the desired operation.

In order to have the GUI classes update when a change is detected in the data of the Restaurant class, the Observer design pattern was implemented. In order to achieve the desired behavior, the Observer and Observable interfaces were created. The Observer interface needs only one method: updateFields(). When implemented, usually, this methods updates the information displayed in the graphical user interface. The Observable interface is usually implemented by the object upon which a change is supposed to affect the state of different objects. In order to determine which object's state should change when the state of the Observable object is changed, the interface provides the following methods: addObserver(Observer) which signals the Observable object that the given Observer object's state should change when a certain event occurs in the Observable. The notifyObservers() method simply calls the updateFields() method for each Observer related to the Observable object.

The RestaurantSerializator, as its name suggests, is used to create a .ser file holding information about the state of the restaurant. The only method inside this class is serialize(Restaurant). In this application when serialization is done, only the Restaurant and all the Objects it is composed of, namely : Orders and MenuItems (BaseProduct, CompositeProduct) are serialized. This way, we avoid serializing the

JFrames used for the graphical user interfaces of each user, which would not make any sense. Each graphical user interace class constructor takes as argument an object implementing the IRestaurantProcessing interface, in this case, that object being the Restaurant object. When the application is launched, if a "restaurant.ser" file is found, the Restaurant object is read from that file, otherwise a new instance is being created. Whether the .ser file is found or not, the graphical user interface classes are being created using their constructors. This way, they function the same regardless of the method the Restaurant object is created.

The OrderWriter class is used for generating .txt files containing billing information. The only method implemented inside this class is the writeBill(Order, List<MenuItem>, price) method. This method creates a new txt file whose name is generated using information held in the Order object passed as argument. The method proceeds to fill the txt file with each MenuItem object that corresponds to the given order, writing the name and the price of the product. Finally a new row is written, having the total price, which is also given as an argument.

5. Results

The application runs as expected, having all the presented features working properly. When opened, it displays three different graphical user interfaces, one for each type of user. As an action is performed on one of the windows, the user can observe how the other windows update accordingly. The Bill button inside the waiter window creates a new .txt file containing billing information corresponding to the selected order. The add and modify buttons inside the admin window prompt new windows, whose use were not discussed thoroughly in the documentation, as they closely related to the main graphical user interface classes, and their only use being that of creating a more intuitive user interface.

6. Conclusions

The layered architecture, alongside the observer and the composite design pattern proved to be of great use when designing the application. The ability to easily notify changes from one class to another using the observer design pattern proved to be useful in the context of multiple graphical user interfaces. The composite design pattern deemed useful for representing the products, allowing one to use a more general class (MenuItem) using polymorphism, that allows one to treat each type of product the same.

Bibliography:

https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html

https://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html