

**TECHNICAL  
UNIVERSITY**  
OF CLUJ-NAPOCA  
ROMANIA

# Restaurant management

Faculty of Automation and Computer Science  
Specialization – Computer Science

Laboratory teacher: Marcel Antal  
Student: Coroian Tudor Florentin

# 1. Objective of the project

The main objective of the current assignment is to implement 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.

The application should follow, as close as possible, the diagram below:

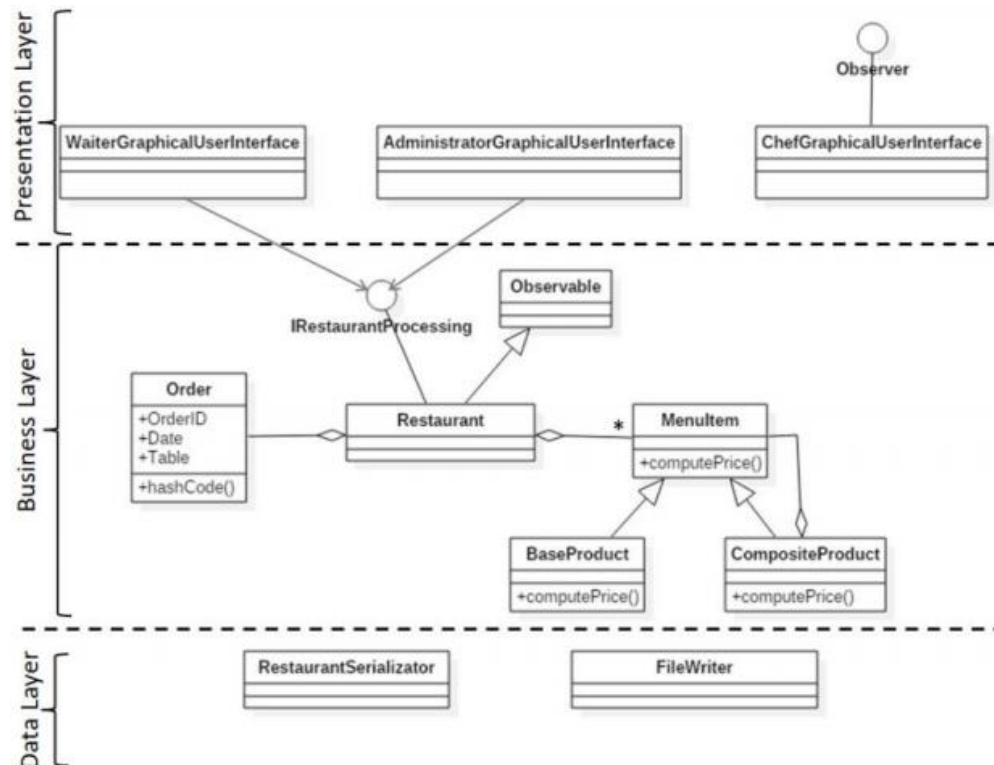


Figure 1. UML diagram of the application (given)

In order to simplify the application, we may consider that the system is used by only one administrator, one waiter and one chef at a time. Also, there is no need of a login process.

The application should allow each respective user to execute the following commands:

- An administrator can – create a new menu item, delete a menu item or edit a menu item
- A waiter can – create a new order, compute the price for an order or generate a bill in a .txt format
- A chef – waits for the waiter to create and place a new order and then cooks it

Furthermore, certain design patterns must be used when designing the classes of the application, namely:

- Composite Design Pattern (CDP) for defining the classes MenuItem, BaseProduct and CompositeProduct, as suggested by the UML Diagram in **Figure 1**
- Observer Design Pattern (ODP) for notifying the chef every time the waiter places a new order

Additional requests for this assignment are going to be described in the next paragraphs.

Implement the class Restaurant using a predefined JCF collection that is based on 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 an appropriate collection consisting of MenuItem objects to store the menu of the restaurant.

Define a method of type “well formed” for the class Restaurant.

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

The menu items for populating the Restaurant object will be loaded and saved from and to a file using serialization. The file from and to which the data will be loaded and stored will be called *Restaurant.ser*.

In order to build the application, second objectives must be considered, such as:

1. designing a class to write to the bill text files
2. deciding on a proper layout of the user interfaces files
3. deciding on a structure for the tables in which the data will be showed
4. deciding on a policy for deleting an entry of the table
5. deciding on means to let the user now which commands failed (either due to syntax errors or logical errors)
6. deciding whether the delete or update operations should be performed and then cascaded

## 2. Problem analysis, problem modeling, scenarios, use cases

### Problem analysis

..

Before moving any further, due to the loose specifications that were given in the objective of the project, some decisions were left to the latitude of the designer. Therefore, those decisions are going to be presented in **Chapter 3, Design decisions**.

First, it is clear that proper structure for the tables should be designed, before moving forward with the rest of the implementation. These will be presented when discussing the user interfaces that are generated for each user, as well as some extra interfaces that are meant to aid the person using the application.

The *Restaurant*, *MenuItem*, *BaseProduct*, *CompositeProduct* and *Order* are, as it can be seen in **Figure 1**, imposed by the UML diagram. However, the content of these models is left to the latitude of the designer. As such, these models will be presented and described in **Chapter 2, Problem modeling** and more thoroughly analyzed in **Chapter 3, Models**.

Secondly, some other classes need to be implemented to further aid the design of the application. As such the *DateT* model is implemented to serve as a building block for the *Order* class.

Keeping in mind that the assignment asks for a restaurant management system, the design of the *MenuItem* has to take into account the fact that an item on the menu can be a simple one (i.e. *BaseProduct* such as tomato, onion, wine, cheese, egg), a compound-simple one, which contains multiple simpler menu items or a compound-compound one, which contains simple and compound menu items (i.e. *CompositeProduct* such as taco, salad, burger). This relationship can be defined recursively, such as at each level an item can be broken down into multiple compound or simple items, with the remark that the simple items cannot be broken down. Taking into account these facts, an easy approach to this limitation of the model is to use the Composite Design Pattern or CDP.

Another aspect that needs to be mentioned at this step in the documentation is the way of storing the items on the menu after the application is closed. For this, the Serialization technique was implied and used. Not only the items are stored, but the orders made by the waiter as well.

On the other hand, the fact that the chef needs to be notified each time the waiter writes and places a new order leads to one simple conclusion. As such, the Observer Design Pattern or ODP was implied and used.

Lastly, Design by Contract techniques were used to implement the *RestaurantProcessing* interface and *Restaurant* class.

## Problem modeling

In what follows, the *BaseProduct*, *CompositeProduct* and *MenuItem* models will be presented, but only briefly. Further details on each class will be presented in Chapter 3.

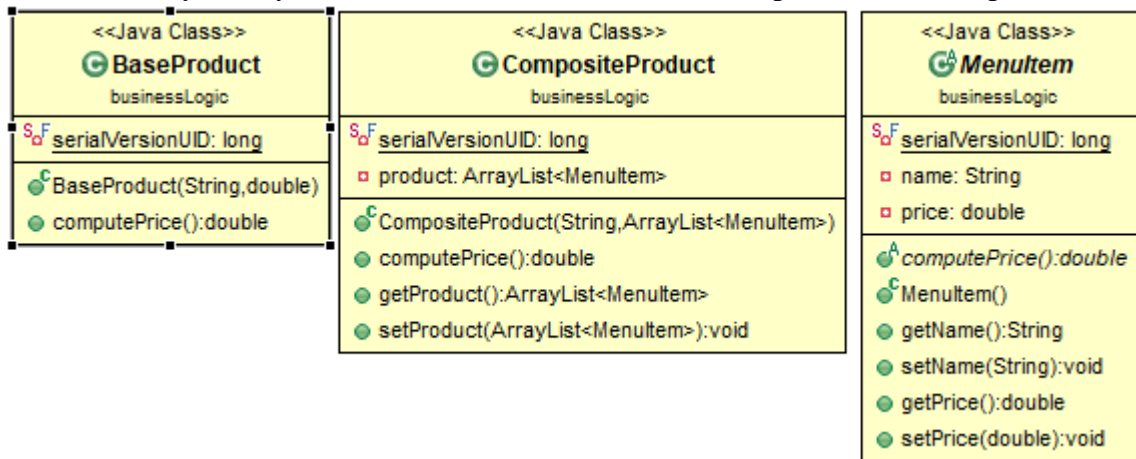


Figure 2. Models of the application - Products

These three models have been already described in the previous section. As a reminder, the design of the *MenuItem* has to take into account the fact that an item on the menu can be a simple one (i.e. *BaseProduct* such as tomato, onion, wine, cheese, egg), a compound-simple one, which contains multiple simpler menu items or a compound-compound one, which contains simple and compound menu items (i.e. *CompositeProduct* such as taco, salad, burger). This relationship can be defined recursively, such as at each level an item can be broken down into multiple compound or simple items, with the remark that the simple items cannot be broken down. Taking into account these facts, an easy approach to this limitation of the model is to use the Composite Design Pattern or CDP.

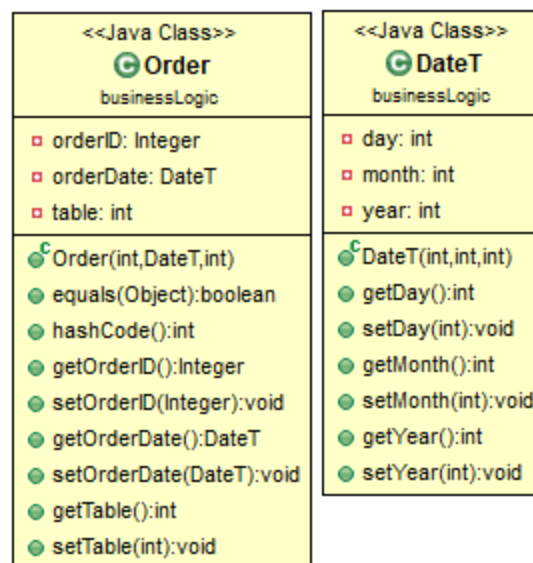


Figure 3. Models of the application - Order

In the picture above, the model for the *Order* class and *DateT* class. The *Order* class will be presented in the **Chapter 3**. The *DateT* class was designed solely for the purpose of this

application. It contains a field for the day, month and year and has the following restriction implemented:

- the day has to be an integer number between 1 and 31
- the month has to be an integer number between 1 and 12
- the year has to be an integer number between 1990 and 2020

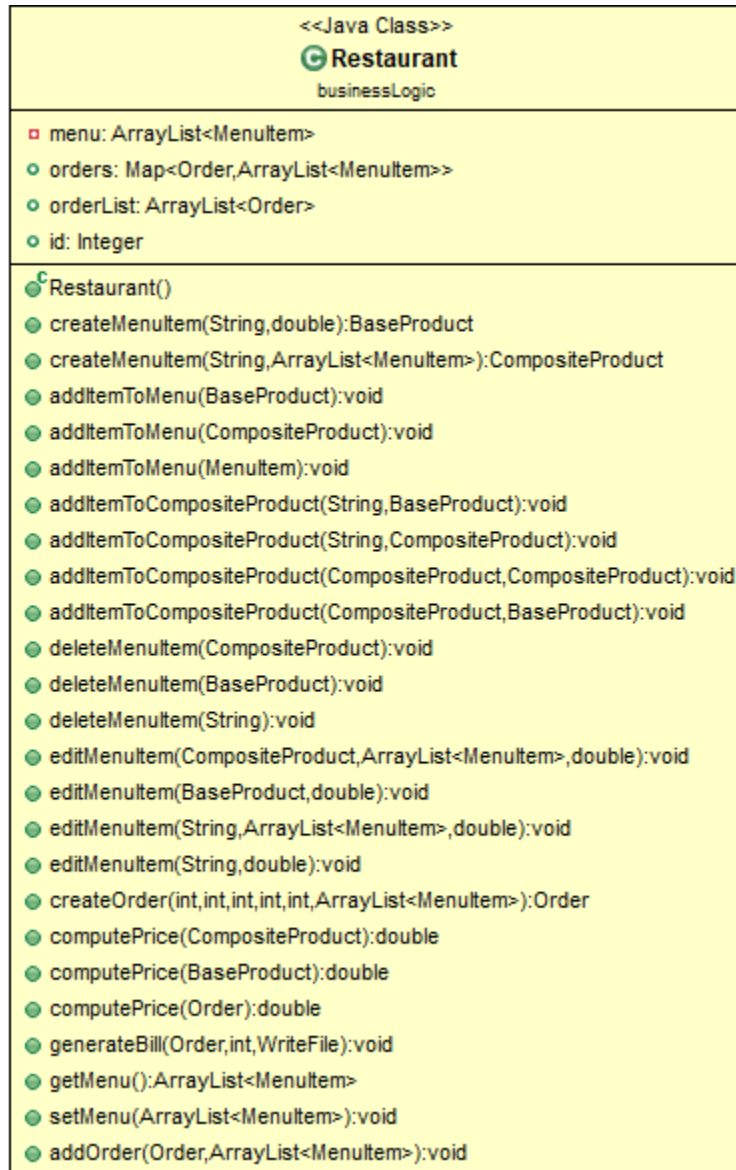


Figure 4. Models of the application - Restaurant

In the figure above, the model of the *Restaurant* class is presented. As it can be seen from the picture, there are a handful of methods that are overloaded. This is to ease the design process in the sense that we can add items to the menu that are *MenuItems*, *BaseProducts* or *CompositeProducts* (items which were constructed previously), but we can also add items to the menu based on the name and price or ingredients directly (without constructing them before). However, the second method is used only when introducing item in the menu from the user

interface directly. On the same note, this class implements the *RestaurantProcessing* interface, which is presented below.

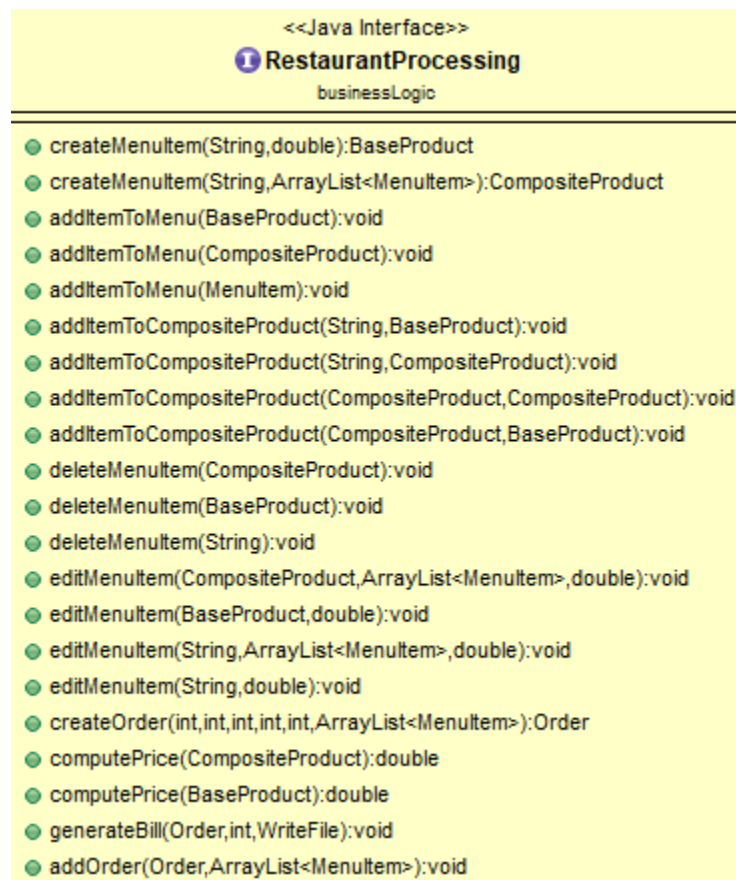


Figure 5. Models of the application - Interface

## Use-case scenario

...

Step 1: Run the *PT2020\_30423\_Tudor\_Coroian\_Assignment\_4.jar* in the command prompt as follows:

```
java -jar PT2020_30423_Tudor_Coroian_Assignment_3.jar [Restaurant.ser]
```

To be noted that the argument is optional. If it is not specified, the application will run with the default (built in) file in which the data will be stored (or from which the data will be loaded respectively).

Step 2: Press enter.

Step 3: Choose the appropriate interface depending on which user is using the application.

Step 4: Follow the tabs and the instruction on the user interface.

Step 5: When done, close the main interface (i.e. the one with the login options)

### 3. Application design

The application “Restaurant management” is designed using the MVC architectural design, that splits the execution of the program into three layers: Model, View and Controller. However, an argument can be made to the fact that the application might be seen from the Layered architectural design, which also splits the execution of the program into three layers: I/O operations with which the user interacts, business logic classes which implement the main logic of the application and data access classes which communicate with the data files. The last ones are designed using reflection techniques that allow for the same code to be performed by multiple classes. Also worth mentioning is that the project is designed using an object-oriented language and the paradigms that come with this language.

For this application the two architectural models were used together, but the main focus was on the Layered architectural design. As a consequence, the models of this application are placed in the same package as the ones

As it can be seen from the explanation above, there are four packages:

- *presentation* which has the user interfaces for the three types of users and another two extra ones: one for the menu and one for the main
- *businessLogic* which contains the classes that perform the actual functionality of the application as well as the models of the application
- *data* which contains the classes that access the data stored in the serialized file and the class that generates the bills
- *MainPackage* which contains only the class that starts the application

It is easy to see from the UML diagram the layered structure of the application. The user access the application via the user interfaces. These communicate with the *Restaurant* class to add items to the menu, edit items from the menu, delete items from the menu, add orders, generate bill or compute the total of an order. After each successful execution of one of these operations, the serialized file is updated so that no data is lost.

#### **Design decisions**

---

Before moving any further with the discussion, some designing decisions have to be stated:

1. The deletion of the entries performed as an actual deletion. The name of the item from the menu is set to null, the price is set to zero and all the ingredients from the product are disconnected. After that, the actual product is deleted.
2. The delete and update operations don't have a cascading effect. If an item is deleted, whereas that item was an ingredient in another item, that another item will not be updated. This decision was made because instead of each item holding a reference to its ingredients, they hold an actual copy of the ingredient. As such, if the restaurant decides to remove an item from the menu, it does not mean that the item that hold said item as an ingredient have to be removed.



3. There is no primary key among the items of the menu. In this circumstance, the application can hold multiple items with the same name and price. The downside is that in this situation, the user has no control over which item will be deleted or updated. When trying to delete or update an item by name, the first item that was inserted in the application with that name will be updated or deleted.
4. The bills that are generated by the waiter will have the title *Bill\_[orderId]\_table\_[tableID].txt*. As such, no two identical bills will be generated.
5. The *Restaurant* class has a field called *id* which holds an integer number that represents the first *orderId* available for the next order.

## Model

---

The diagrams for each model class was previously presented. In what follows, some details of these classes will be explained.

When talking about the *CompositeProduct* class, it is worth mentioning the way in which the price of such a product is computed. The price for a compound item is the sum of all the ingredients that make up that item. This applies recursively, if the ingredients of said product are *CompositeProducts* on their own.

```
@Override
public double computePrice() {
    double price = 0;
    for(MenuItem it : this.product) {
        if(it.getClass().toString().contentEquals("class businessLogic.CompositeProduct")) {
            price += it.computePrice();
        } else if (it.getClass().toString().contentEquals("class businessLogic.BaseProduct")) {
            price += it.getPrice();
        }
    }
    return price;
}
```

Figure 6. computePrice() method for CompositeProduct class

Also worth mentioning is the fact that both the *BaseProduct* and *CompositeProduct* extend the *MenuItem*. The model of the *MenuItem* class was presented previously. It holds the name of the item as well as the prices of the item. It has only one method, namely the computePrice() method. Given the fact that this class is an abstract one, this method is not implemented here. Rather, each extension of the *MenuItem* implements this method.

## Data layer

..

From this layer, the *RestaurantSerializer* class is the one worth describing. This class is the one that deals with the serialization and deserialization of the file that holds the data for the application.

```
public class RestaurantSerializer {

    public static void serialize(Restaurant restaurant) {
        try {
            FileOutputStream fileOut = new FileOutputStream("Restaurant.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(restaurant);
            out.close();
            fileOut.close();
            System.out.println("Data was serialized successfully. Check \"Restaurant.ser\".");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static Restaurant deserialize() {
        Restaurant restaurant;
        try {
            FileInputStream fileIn = new FileInputStream("Restaurant.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            restaurant = (Restaurant) in.readObject();
            in.close();
            fileIn.close();
            System.out.println("Data was deserialized successfully.");
            return restaurant;
        } catch (FileNotFoundException e) {
            System.out.println("Couldn't find file \"Restaurant.ser\".");
            restaurant = new Restaurant();
            serialize(restaurant);
            e.printStackTrace();
            return restaurant;
        } catch (IOException e) {
            System.out.println("Problem encountered with the input file.");
            restaurant = new Restaurant();
            serialize(restaurant);
            e.printStackTrace();
            return restaurant;
        } catch (ClassNotFoundException e) {
            System.out.println("Couldn't find class Restaurant.");
            e.printStackTrace();
            return null;
        }
    }
}
```

Figure 7. RestaurantSerializer class

The way this works is the file is deserialized when the application is run and after each operations of insertion, update or deletion, this file is updated (i.e. the class is serialized).

## BusinessLogic layer

...

The actual logic of the application is implemented in the business logic layer. Among these, the models of the application are found here. This decision was made in order to not complicate the design of the application. Also, each model holds the methods that can be performed on them.

Most important to specify is that this layer is where the interface *RestaurantProcessing* and the *Restaurant* class. The second implements the first. The methods of the *Restaurant* class are presented bellow.

```
addItemToCompositeProduct(businessLogic.CompositeProduct itemToAddTo,  
businessLogic.BaseProduct product)  
addItemToCompositeProduct(businessLogic.CompositeProduct itemToAddTo,  
businessLogic.CompositeProduct product)  
addItemToCompositeProduct(java.lang.String itemToAdd,  
businessLogic.BaseProduct product)  
addItemToCompositeProduct(java.lang.String itemToAdd,  
businessLogic.CompositeProduct product)  
addItemToMenu(businessLogic.BaseProduct product)  
addItemToMenu(businessLogic.CompositeProduct product)  
addItemToMenu(businessLogic.MenuItem product)  
addOrder(businessLogic.Order order,  
java.util.ArrayList<businessLogic.MenuItem> products)  
computePrice(businessLogic.BaseProduct product)  
computePrice(businessLogic.CompositeProduct product)  
createMenuItem(java.lang.String name,  
java.util.ArrayList<businessLogic.MenuItem> ingredients)  
createMenuItem(java.lang.String name, double price)  
createOrder(int id, int day, int month, int year, int table,  
java.util.ArrayList<businessLogic.MenuItem> products)  
deleteMenuItem(businessLogic.BaseProduct product)  
deleteMenuItem(businessLogic.CompositeProduct product)  
deleteMenuItem(java.lang.String name)  
editMenuItem(businessLogic.BaseProduct product, double newPrice)  
editMenuItem(businessLogic.CompositeProduct product,  
java.util.ArrayList<businessLogic.MenuItem> ingredients, double newPrice)  
editMenuItem(java.lang.String name,  
java.util.ArrayList<businessLogic.MenuItem> ingredients, double newPrice)  
editMenuItem(java.lang.String name, double newPrice)  
generateBill(businessLogic.Order order, int price, data.WriteFile writer)
```

There are many methods that do the same thing. This decision was made to ease the design process. There is no logical difference between the methods with the same name.

The methods from these classes are self-explanatory and do not need any further description.

## Presentation Layer

This is the layer that deals with all the user interfaces. Bellow, the UML diagrams of these classes will be presented.

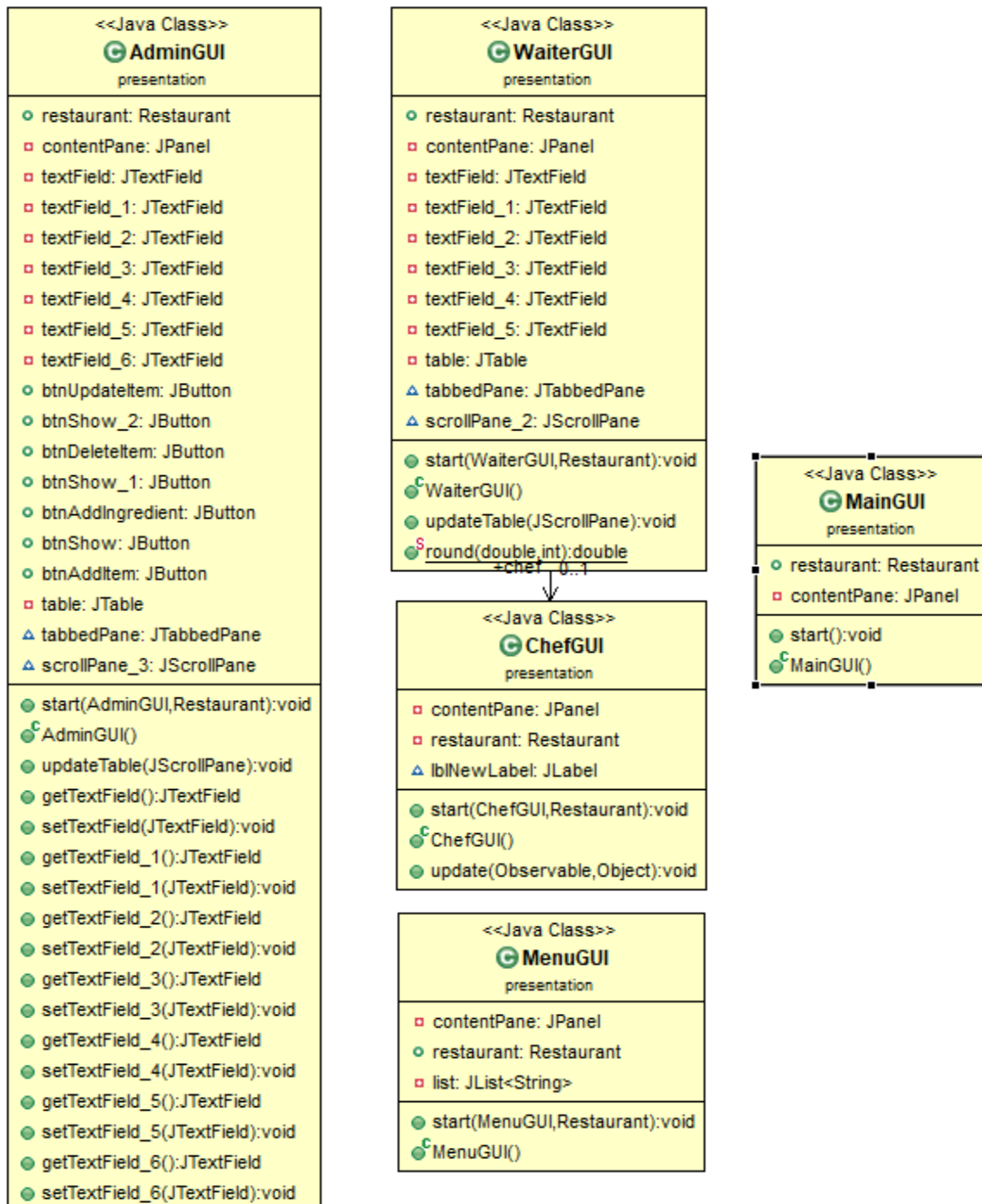


Figure 8. Interfaces of the application

Each class represents one window of the application. In what follows, each of these classes and their respective windows will be presented, due to the fact that it offers a better understanding on how to use the application.

Bellow, the aspect of the MainGUI is presented.

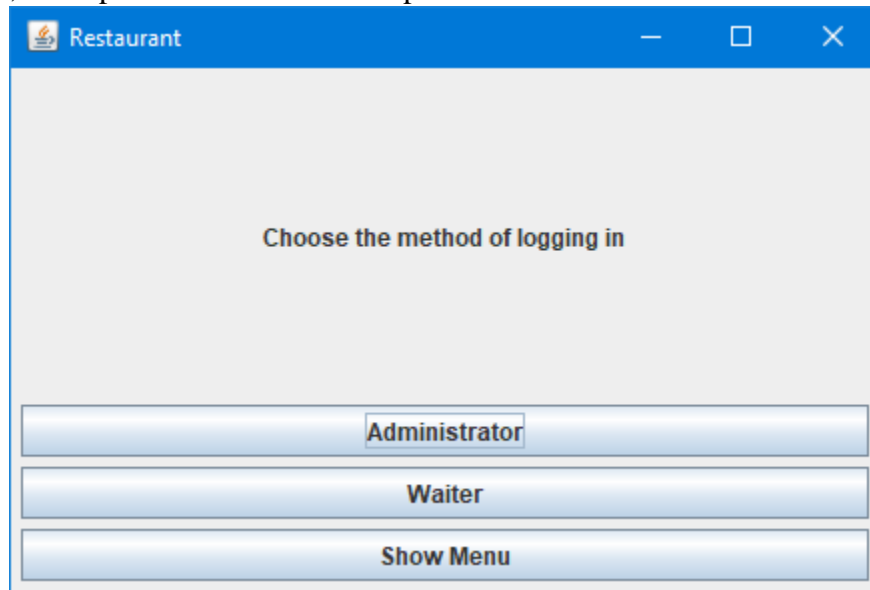


Figure 9. MainGUI - main interface of the application

As it can be seen from the picture, there are three buttons on this window. Each of them will open a new window, in turn. Each of these windows will be presented in what follows. In increasing order of complexity.

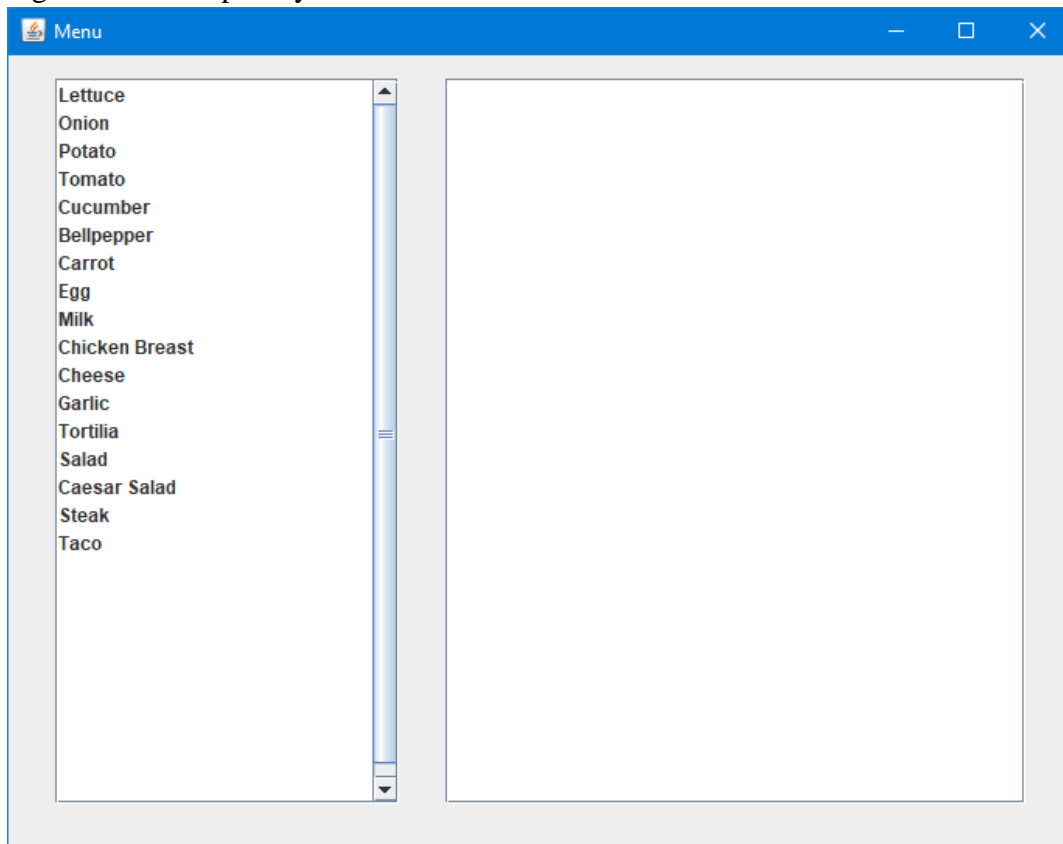


Figure 10. MenuGUI - the menu interface of the application

This is the window that pops up when clicking the *Show Menu* button. On the left, there is a list with all the item from the menu of the application. When clicking on one of the items, on the right pane a new list with all the ingredients and their price of the clicked element will appear, if the item is a compound product or a message letting the user now that the product selected is a base product.

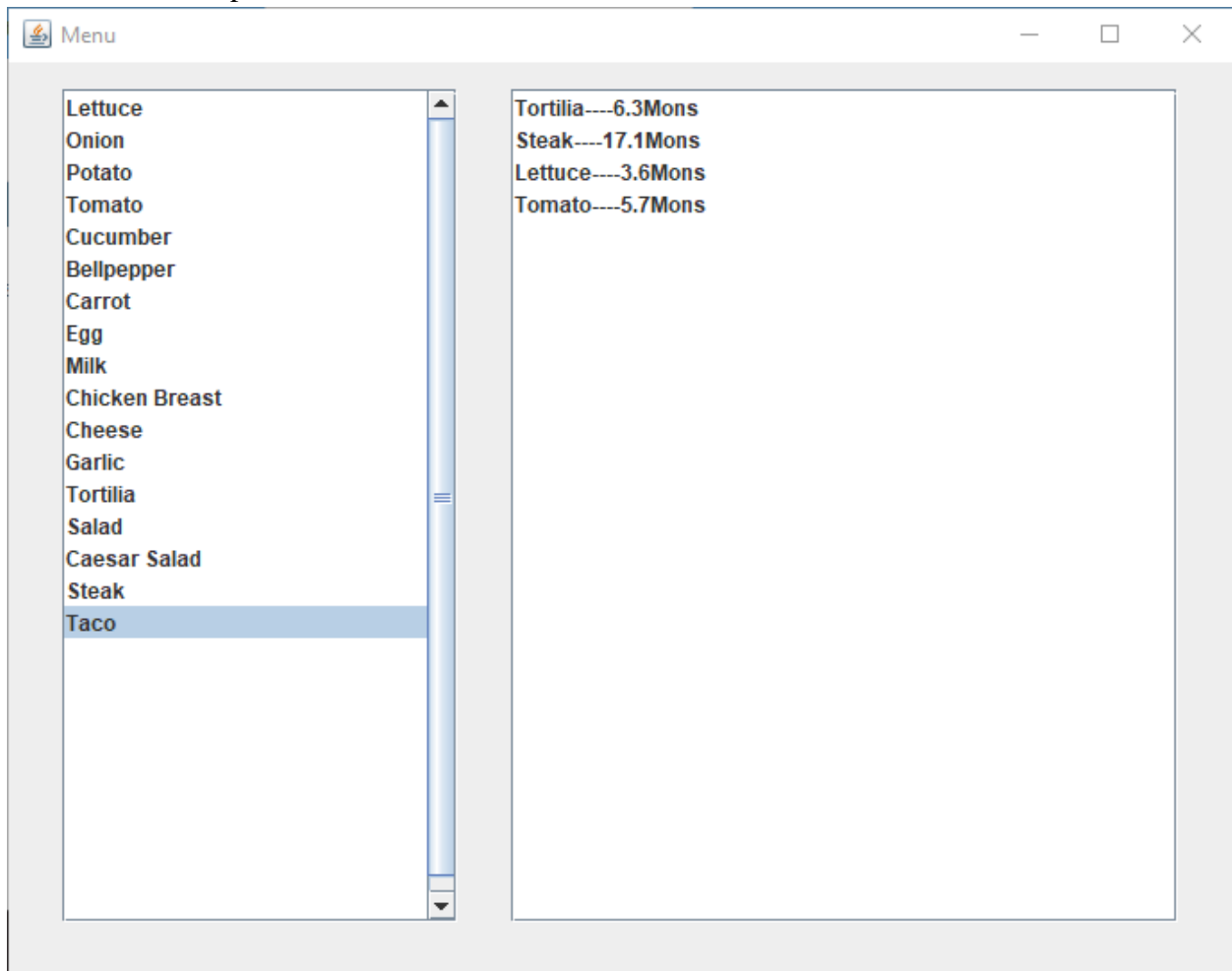


Figure 11. MenuGUI - after clicking on a compound product

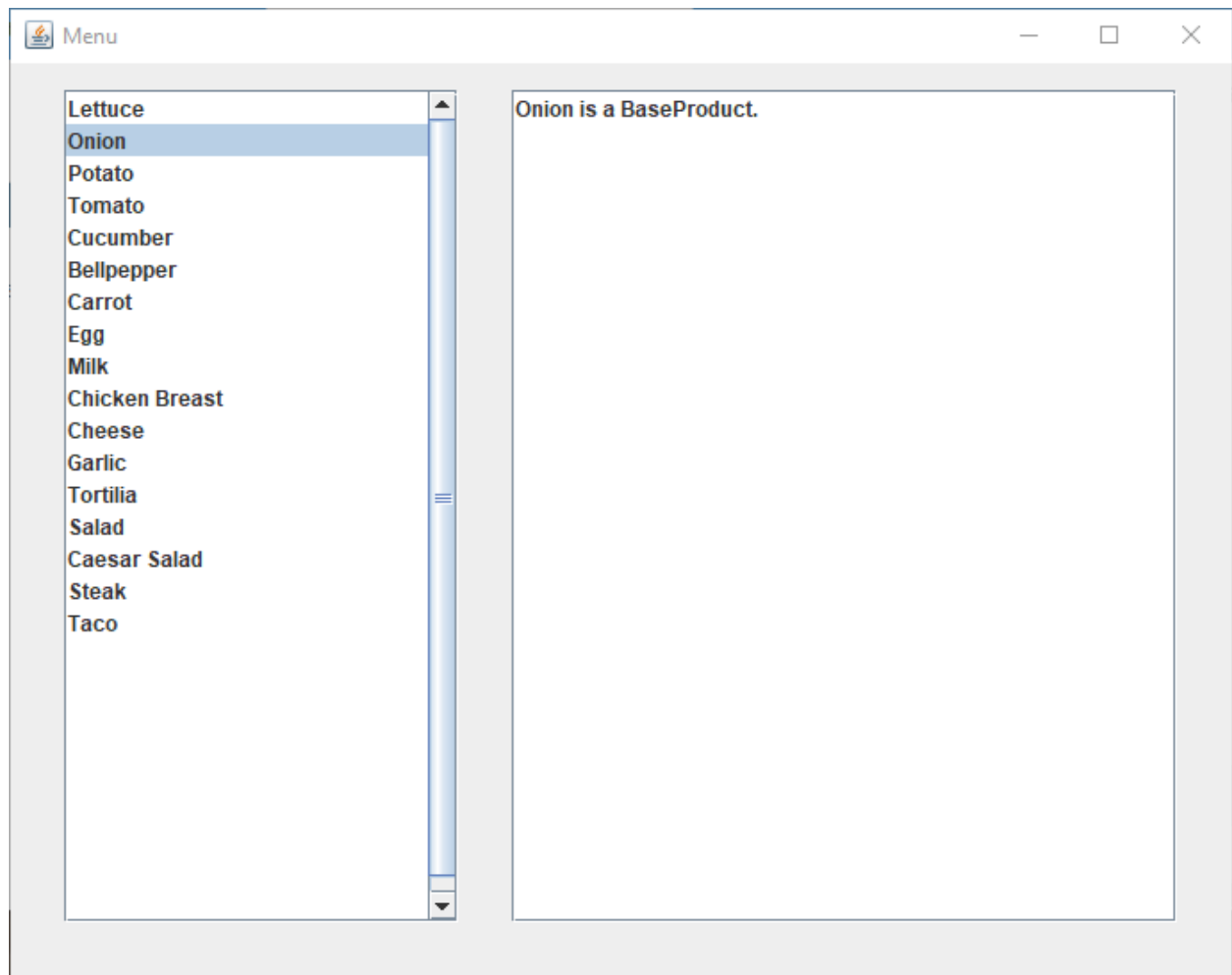


Figure 12. MenuGUI - after clicking on a base product

Waiter

Update order table

Create order

Orders

Create a new order

Table:

1

Items:

Onion, Potato

Date:

12/4/2020

Create order

Show all orders

Order id:

Price:

Bill

Show menu

Lettuce----3.6Mons

Onion----4.9Mons

Potato----4.3Mons

Tomato----5.7Mons

Cucumber----7.3Mons

Bellpepper----5.1Mons

Carrot----2.7Mons

Egg----1.7Mons

Milk----2.9Mons

Chicken Breast----10.9Mons

Figure 13. WaiterGUI



At a first glance, the interface for the waiter might seem complicated. From the beginning it can be seen that it has two tabs. One is for making the order and the other is a table with all the orders of the application.

To breakdown the window, we will present it from top to bottom.

The first button, “Update order table” does exactly as it is stated. It updates the table on the “Orders” tab.

Moving down, the rest of the window can be broken up in two parts. The upper part (everything above and including the “Create order” button) can be seen as the order input section. The waiter enters the data in the required fields and then presses the “Create order” button. This action creates the order and notifies the chef to cook said order.

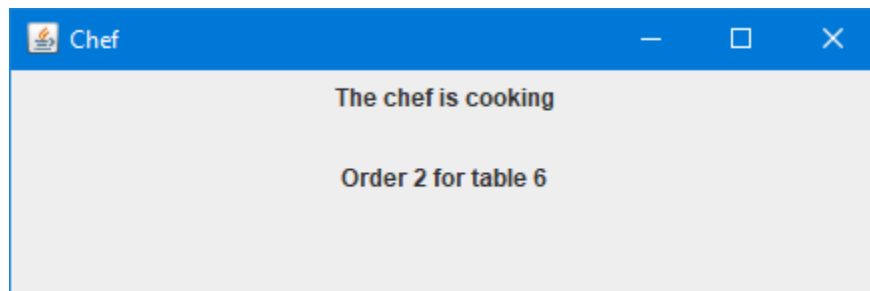


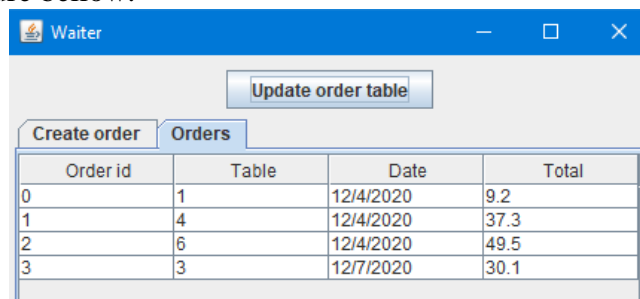
Figure 14. ChefGUI - after pressing "Create new order"

Moving down, the second part of the window contains the bill generating part (everything bellow and including the “Show orders” button). If the waiter presses “Show orders” all the ids of the orders already made will appear in the list bellow. When clicking on one of these ids, the fields will be automatically field with the data of that order. This mechanism is useful for computing the price of the order and generating a bill for a previously made order. However, one **CANNOT COMPUTE THE PRICE OR GENERATE THE BILL** for an order that was made in another run of the application, even if those orders are still kept in the application.

When the fields are completed, if the waiter presses the “Price” button, the total price of the order will appear in the text field near it. After that, the waiter can press the “Bill” button and a bill will be generated, followed by a successful message appearing in the text field near it.

Lastly, the “Show menu” button, does exactly as it is said. It shows the items on the menu in the list bellow. This button will appear on each window and will not be explained further.

Moving over to the “Orders” tab, if the waiter enters this tab and then presses the “Update order table”, a table with all the orders made (from all the runs of the application). This can be seen in the picture bellow.

A screenshot of a window titled 'Waiter'. The window has a blue title bar with standard Windows window controls. Below the title bar is a button labeled 'Update order table'. Below that are two tabs: 'Create order' and 'Orders'. The 'Orders' tab is selected, showing a table with four columns: 'Order id', 'Table', 'Date', and 'Total'. The table contains four rows of data.

Order id	Table	Date	Total
0	1	12/4/2020	9.2
1	4	12/4/2020	37.3
2	6	12/4/2020	49.5
3	3	12/7/2020	30.1

Figure 15. Orders table in WaiterGUI

Figure 16. AdminGUI - Add item tab

The window above is the one that appears after the user presses the “Administrator” button. As it can be seen from the picture, the window has four tabs which will be described next. First of all, the button at the top of the window, “Update Menu Table” has the same function as the one in the same position on the WaiterGUI. It only affects the “Menu Table” tab.

The first tab, “Add item”, is the one used by the administrator to add a new item to the menu. The first two text fields are used to input a BaseProduct item. The “Add item” button does exactly that. It is worth mentioning that any item that is added in the application via the “Add item” button will be a BaseProduct. In order to create a CompositeProduct, one must choose a BaseProduct ALREADY INSERTED in the application (i.e. one that appears on the list generated after pressing the “Show” button), type any number greater than zero in the “Price” field (it doesn’t matter which number, it will be updated anyway) and then choose an ingredient from the item list and type it in the “Ingredient” field. Finally, the admin has to press the “Add ingredient” button and after this, the BaseProduct will become a CompositeProduct.

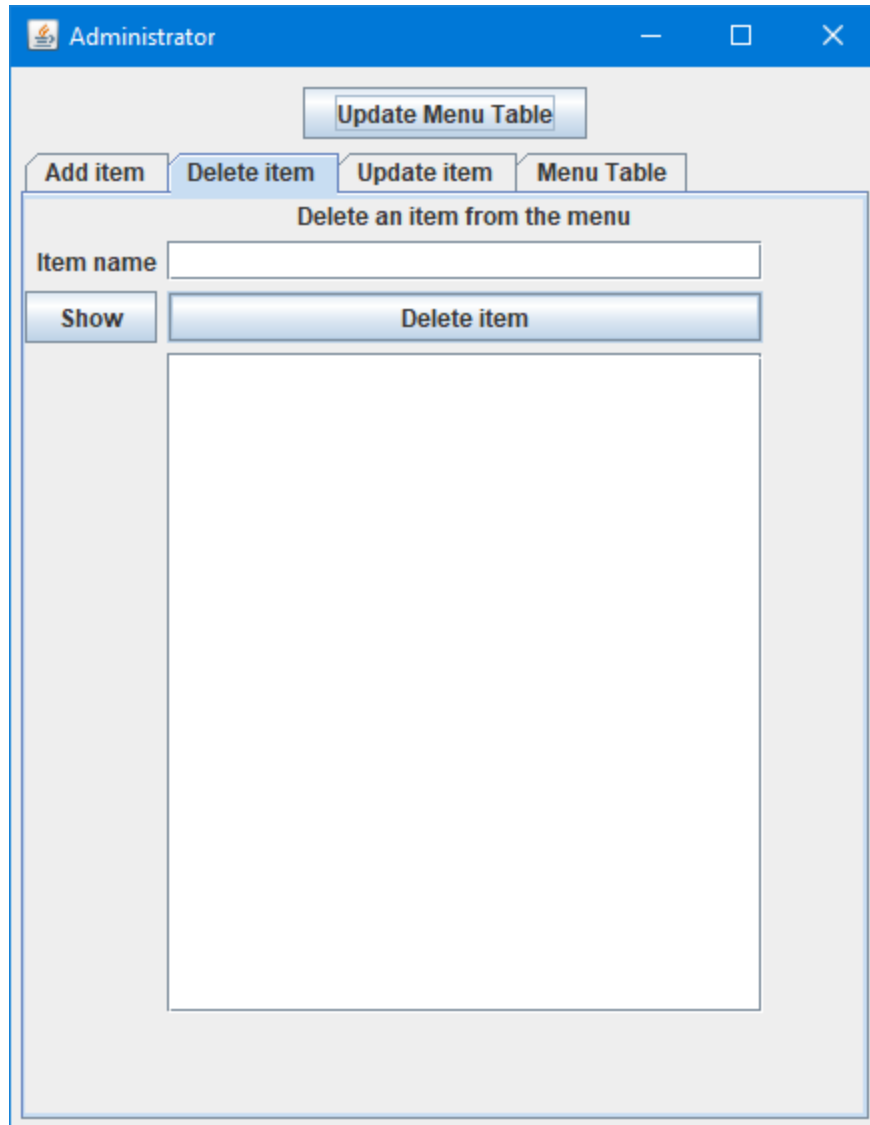


Figure 17. AdminGUI - Delete item tab

Above is presented the “Delete item” tab. The “Show” button has the same function as presented before. To delete an item, the administrator has to input the name of the item in the text field then press the “Delete item” button.

Bellow it is presented the “Update item” tab. The “Show” button has the same function as presented before. To update an item, the administrator hat to input the name of the item in the text field, then add a new price for that item (if the price needs to be changed) or type the name of a new ingredient to be added to the item. The update cannot remove ingredients from the item.

Administrator

Update Menu Table

Add item Delete item Update item Menu Table

Update an item from the menu

Item name

New price

New ingredient

Show Update item

Figure 18. AdminGUI - Update item tab

The last tab that needs to be presented is the “Menu table” tab. It contains a table with all the items and their price in the application. This tab is presented bellow.

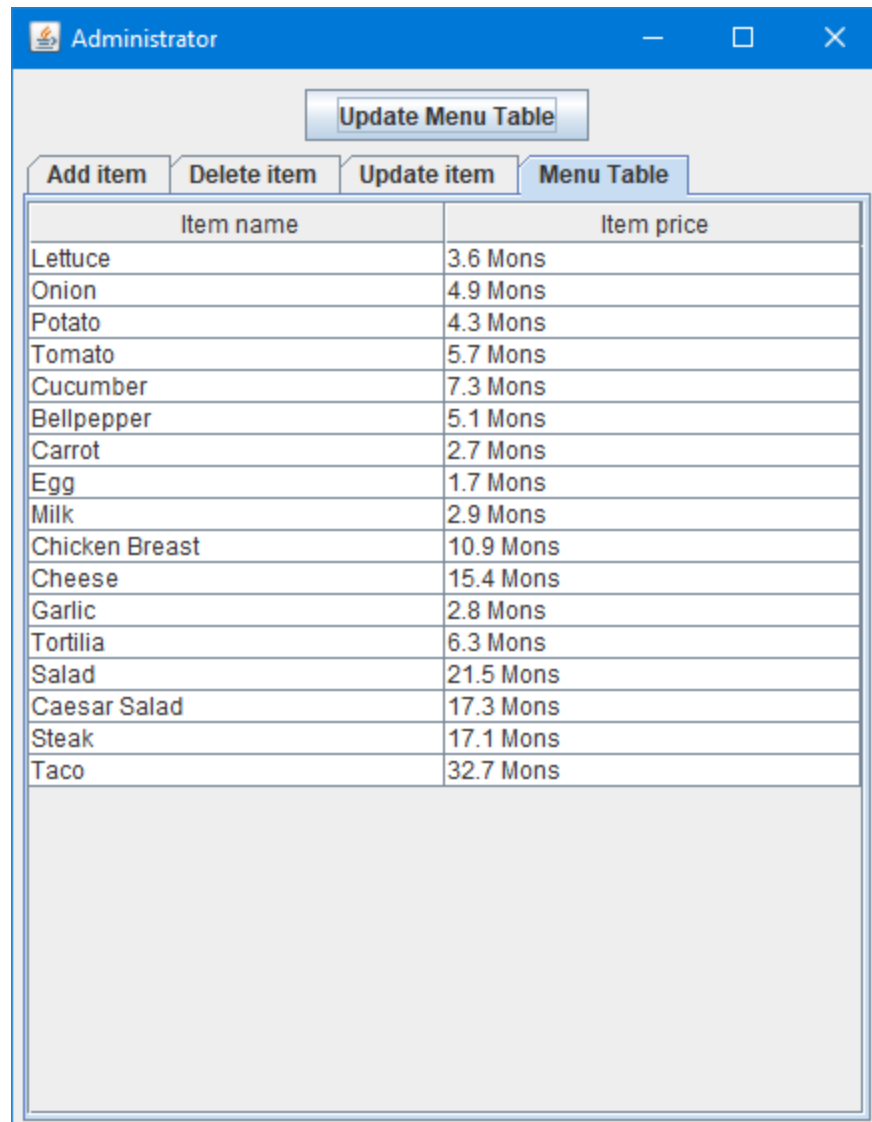


Figure 19. AdminGUI - Menu table tab

## 4. Implementation

### Overview

All the classes and method have been presented in the previous chapters.

One more note has to be made on the implementation of the graphical interfaces. In order to avoid complications of the structure of the application, there are no listener classes implemented for the buttons. All the actions are implemented as anonymous classes. Although this might make the code look heavier, it is easier to follow the flow of the execution.

This decision was made to help aid the design of the application and to not create too many classes for an application of small proportions.

## 5. Results

### **Overall view of the application**

...

The UML diagram of the application was given in the requests of the assignments. It represents the application almost perfectly, with the addition of some more classes to spread the actions done.

### **Testing**

...

The testing for this application consisted on testing every corner case that a user of this application might run into.

The data from my runs can be found in the *Restaurant.ser* file.

Three bills have been generated for the three orders that were created during the testing of the application.

### **User interface**

...

For this application, the user interface is a complex one and can seem complicated to understand at first. However, this interface was presented thoroughly in the chapters before as well as how the interface is made to be used.

## **6. Conclusions and further development**

The user interface can be upgraded even more. The same mechanism that was applied for the autocompletion of the fields when selecting an order id from the list of orders in the WaiterGUI can be implemented for all the other tabs where the user is requested to input something that was already introduced in application.

Furthermore, minor changes can be made for a more intuitive approach to using the application.

## **7. Bibliography**

[http://coned.utcluj.ro/~salomie/PT\\_Lic/4\\_Lab/Assignment\\_4/Assignment\\_4\\_Indications.pdf](http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_4/Assignment_4_Indications.pdf)

[http://www.tutorialspoint.com/java/java\\_serialization.htm](http://www.tutorialspoint.com/java/java_serialization.htm)

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

<https://www.geeksforgeeks.org/serialization-in-java/>

<https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html>

<https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#tag>

