# Restaurant management System

~ Homework 4 ~

Papita Anda

Group 30421

*Programming techniques*

## Table of Contents

## 1. Objective

The objective of this laboratory homework is the design and implementation of restaurant management system meant for processing customer orders for a restaurant. The system should have three types of users: administrator, waiter and chef. In other words, 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.

Hence, the purpose of this app is minimizing the complexity with which waiters and chefs can keep track of orders and administrators to organize their restaurant. From an educational perspective, this assignment was meant to help the student get familiar with the concepts of design by contract programming techniques, polymorphism, design patterns: observer, composite, serialization.

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

Several programming techniques were used in order to sustain the functionality of the application.

The Design by Contract (DBC) software development technique ensures high-quality software by guaranteeing that every component of a system lives up to its expectations. Central to DBC is the notion of an *assertion* -- a Boolean expression about the state of a software system. At runtime we evaluate the assertions at specific checkpoints during the system's execution. In a valid software system, all assertions evaluate to true. In other words, if any assertion evaluates to false, we consider the software system invalid or broken. We identify three different types of expressions: preconditions, postconditions and invariants.

Another concept used in the development of this assignment is polymorphism. Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways. This concept was needed in the menu item – product classes.

One of the two design patterns used is the composite pattern. Composite pattern is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Moreover, The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. One needs to add objects of type subject and observer. The sole responsibility of a subject is to maintain a list of observers and to notify them of state changes by calling their `update()` operation. The responsibility of observers is to register (and unregister) themselves on a subject (to get notified of state changes) and to update their state (synchronize their state with subject's state) when they are notified. This makes subject and observers loosely coupled.

Last but not least, this application uses serialization. Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object. The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. Only the objects of those classes can be serialized which are implementing **java.io.Serializable** interface.
Serializable is a marker interface (has no data member and method).

## *Example of use case scenarios*

**Use case title: add a menu item to the menu**

- **Summary: this use case allows the administrator to add an item to the menu table**
- **Actor: the user of the application, i.e. the administrator of the restaurant**
- **Preconditions: menu item to be introduced is not null**
- **Main success scenario:**
    - The user introduces the name of the dish
    - The user introduces the price of the dish
    - the new menu item is introduced in the menu list

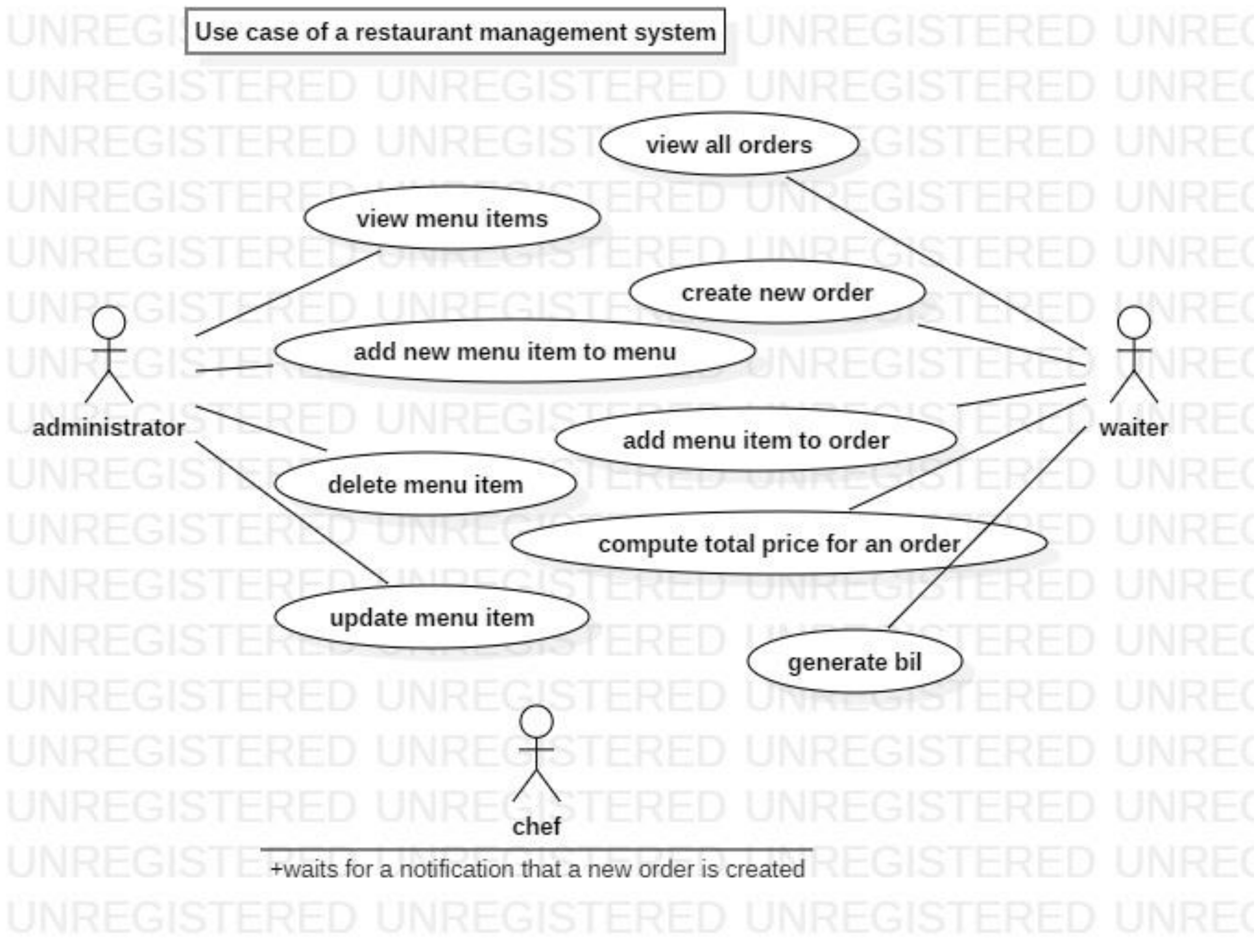
**Use case title: create a new order**

- **Summary: this use case allows the waiter to create a new order by adding at least a menu item on the order**

- **Actor: the user of the application, i.e. the waiter of the restaurant**
- **Preconditions: order details must not be null**
- **Main success scenario:**
  - The user introduces the table for which the order is created
  - The user introduces one menu item (which is the minimum for an order to be valid)
  - the new order is created and the chef is notified
  - the waiter can add other dishes to the order
- **Alternative sequences:**
  - **a)** Incorrect or inexistent dish name
    1. A pop-up message will show and tell the user the menu item is not on the menu list
    2. The app does not create the order and the waiter can reintroduce other valid data

All the other use cases are similar to these ones.

The use case diagram is the following one:

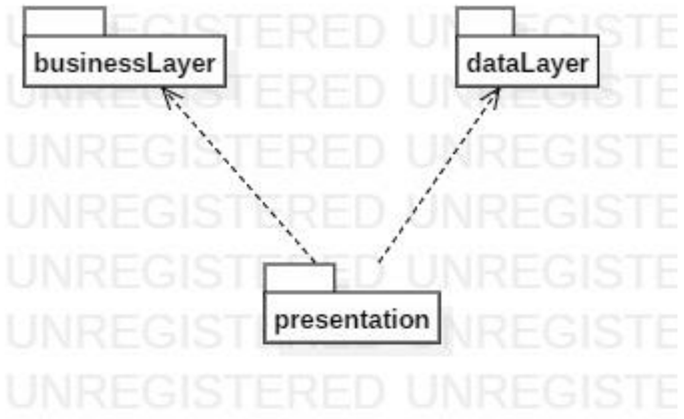

Use case of a restaurant management system
view all orders
view menu items
create new order
add new menu item to menu
administrator
waiter
add menu item to order
delete menu item
compute total price for an order
update menu item
generate bil
chef
+waits for a notification that a new order is created

## 3. Design

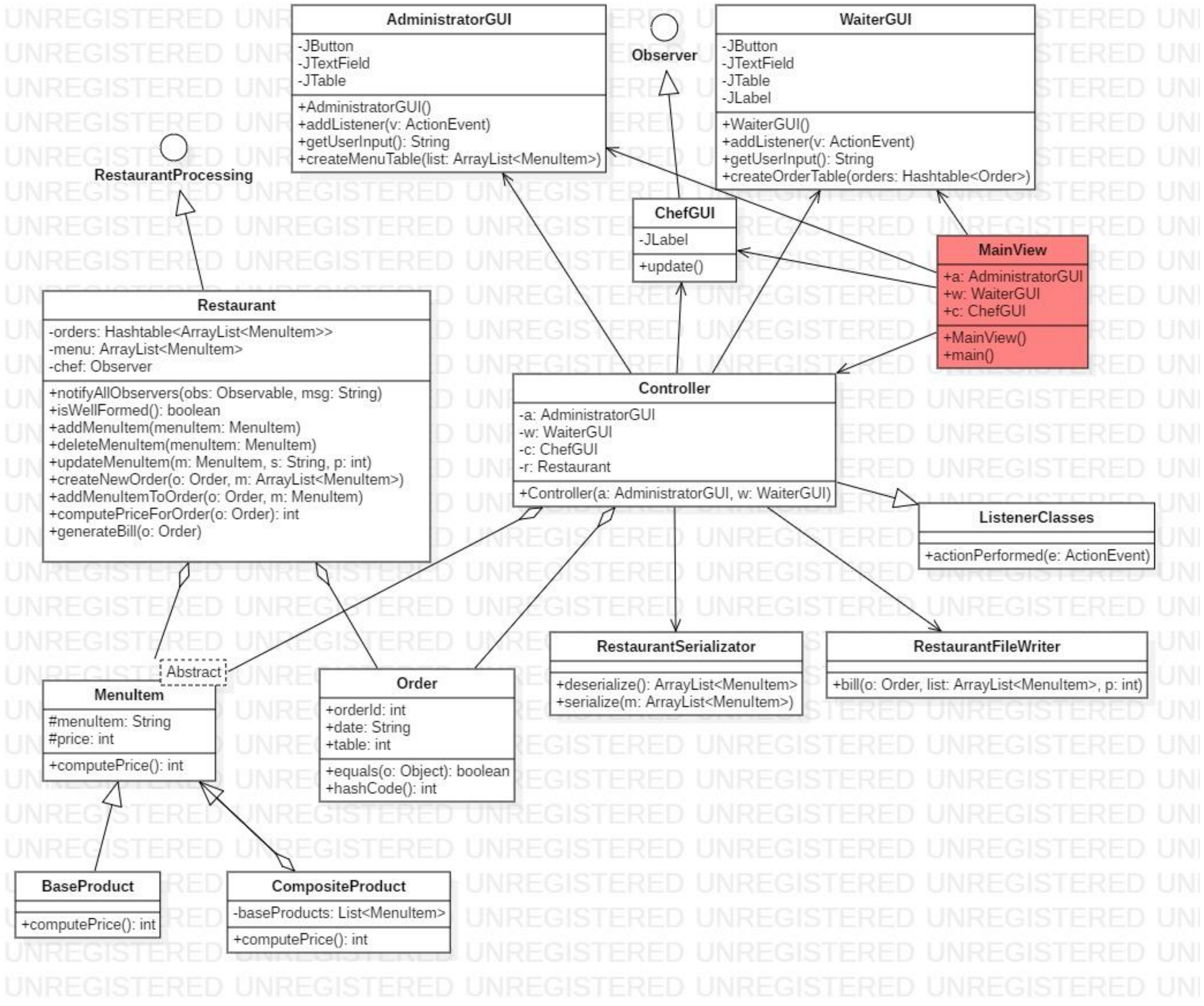
The design of this app follows a layered design architecture. The 3 packages are: presentation, businessLayer, dataLayer. Therefore, the business layer contains the MenuItem, BaseProduct, CompositeProduct, Order and Restaurant classes and the RestaurantProcessing interface. These classes are the ones which contain as attributes all the fields that can be found in the corresponding tables from the GUI. The dataAccessLayer is the package that contains the classes that handle the connectivity between the GUI and the business layer classes. It contains the RestaurantSerializator class and the RestaurantFileWriter class. The presentation package, represented by the classes AdministratorGUI, WaiterGUI, ChefGUI, MainView, Controller, follows a bit the typical MVC pattern. Thus, the GUI classes are responsible for the graphic user interface design for the each of the three users. These classes extend JTabbedPane and JPanel classes, while the class that brings them all together and starts the app, MainView, extends the JFrame class. The Controller acts on both model and view, and in addition it also uses the businessLayer classes. It also controls the data flow between what the database already has and what the user updates or introduces. More precisely, it coordinates each button the user can press to perform a specific action and helps display the resulting consequences.

The connections between packages can be seen in the following diagram:

Package relationships



The UML diagram of the project is the following one:



AdministratorGUI
-JButton
-JTextField
-JTable
+AdministratorGUI()
+addListener(v: ActionEvent)
+getUserInput(): String
+createMenuTable(list: ArrayList<MenuItem>)
Observer
WaiterGUI
-JButton
-JTextField
-JTable
-JLabel
+WaiterGUI()
+addListener(v: ActionEvent)
+getUserInput(): String
+createOrderTable(orders: Hashtable<Order>)
RestaurantProcessing
ChefGUI
-JLabel
+update()
MainView
+a: AdministratorGUI
+w: WaiterGUI
+c: ChefGUI
+MainView()
+main()
Restaurant
-orders: Hashtable<ArrayList<MenuItem>>
-menu: ArrayList<MenuItem>
-chef: Observer
+notifyAllObservers(obs: Observable, msg: String)
+isWellFormed(): boolean
+addMenuItem(menuItem: MenuItem)
+deleteMenuItem(menuItem: MenuItem)
+updateMenuItem(m: MenuItem, s: String, p: int)
+createNewOrder(o: Order, m: ArrayList<MenuItem>)
+addMenuItemToOrder(o: Order, m: MenuItem)
+computePriceForOrder(o: Order): int
+generateBill(o: Order)
Controller
-a: AdministratorGUI
-w: WaiterGUI
-c: ChefGUI
-r: Restaurant
+Controller(a: AdministratorGUI, w: WaiterGUI)
ListenerClasses
+actionPerformed(e: ActionEvent)
RestaurantSerializator
+deserialize(): ArrayList<MenuItem>
+serialize(m: ArrayList<MenuItem>)
RestaurantFileWriter
+bill(o: Order, list: ArrayList<MenuItem>, p: int)
Abstract
MenuItem
#menuItem: String
#price: int
+computePrice(): int
Order
+orderId: int
+date: String
+table: int
+equals(o: Object): boolean
+hashCode(): int
BaseProduct
+computePrice(): int
CompositeProduct
-baseProducts: List<MenuItem>
+computePrice(): int

# 4. Implementation

*MenuItem class:* this abstract class represents the format of a menu item that can be on the menu list. Its attributes are the name and the price of the menu item. It contains the abstract method for computing the price.

*BaseProduct class:* this class represents the format of a product which can be introduced in the menu table. Each object of type Product will have as attributes the ones of the MenuItem class. Its methods are only getters and setters, a constructor and a toString function and the computePrice method, inherited by extending the MenuItem class.

*CompositeProduct class:* this class is identical with the previous one, but in addition to the inherited attributes, it contains a list of base products from which the composite product is composed. It also consists of two constructors and the inherited method for computing the price.

*Order class:* this class represents what an order should look like. Its attributes are an id, the date and the table number. It has two overridden methods: one computes the hash code and the other one, called equals, checks if there already is another identical order in the table.

*Restaurant class:* this class implements the methods from the processing interface. Basically, these methods are the operations that the users of the restaurant system can perform: add menu item to the menu, create a new order etc. its attributes are the menu list, a hashtable structure that contains the order and each menu item associated to it and an object of type Observer for the chef. Almost all methods in this class contain preconditions and postconditions, implemented by the help of assertions. The method which creates a new order also contains the notify method, notifying the chef when it successfully creates a new order.

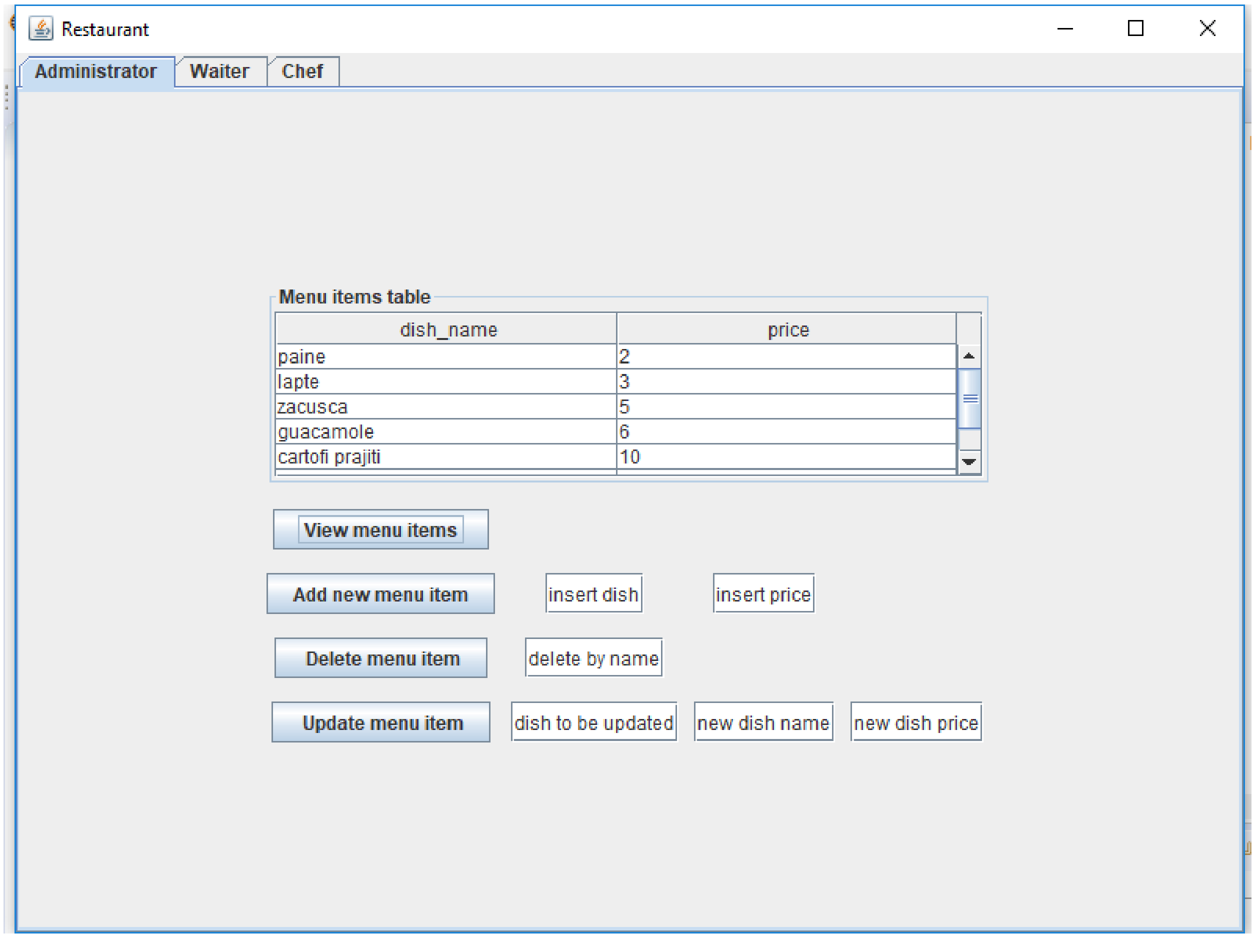
*RestaurantProcessing interface:* this interface contains all the methods that the restaurant class implements, they represent the actions that the administrator and the waiter can perform within the app.

*RestaurantFileWriter class:* this class contains a single method, one that generates the bill for an order. This method uses the classes File and FileWriter in order to create a text file with the bill, which contains all the information of an order.

*RestaurantSerializator class:* this class is the helper class which contains the methods for the serialization process: serialize and deserialize. These two methods work with an ArrayList which represents the menu.

*MainView class:* this class creates the interface the user works in regard to introducing the necessary and desired values in the menu and order tables, and also showing the tables' records. It extends the JFrame class. Its components include the three other GUI classes. It also contains a main function which launches the app.

The interface has the following structure:

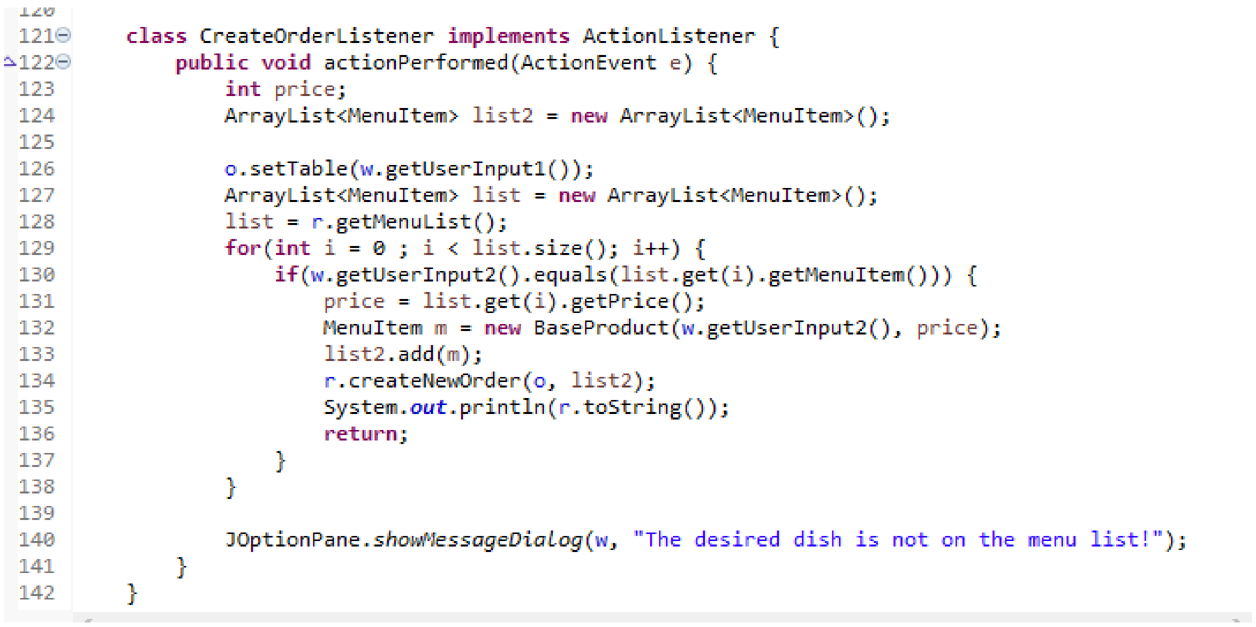


Menu items table

| dish_name | price |
|---|---|
| paine | 2 |
| lapte | 3 |
| zacusca | 5 |
| guacamole | 6 |
| cartofi prajiti | 10 |

*Controller class:* this class acts on dataLayer, businessLayer and GUI user classes. It controls the data flow that goes into the database and updates the MainView whenever data changes and the other way around. More precisely, it coordinates each button the user chooses to press. It consists of one constructor that implements view functions. The class also contains multiple listener classes for all the buttons.

For example:



```
    class CreateOrderListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            int price;
            ArrayList<MenuItem> list2 = new ArrayList<MenuItem>();

            o.setTable(w.getUserInput1());
            ArrayList<MenuItem> list = new ArrayList<MenuItem>();
            list = r.getMenuList();
            for(int i = 0 ; i < list.size(); i++) {
                if(w.getUserInput2().equals(list.get(i).getMenuItem())) {
                    price = list.get(i).getPrice();
                    MenuItem m = new BaseProduct(w.getUserInput2(), price);
                    list2.add(m);
                    r.createNewOrder(o, list2);
                    System.out.println(r.toString());
                    return;
                }
            }

            JOptionPane.showMessageDialog(w, "The desired dish is not on the menu list!");
        }
    }
```

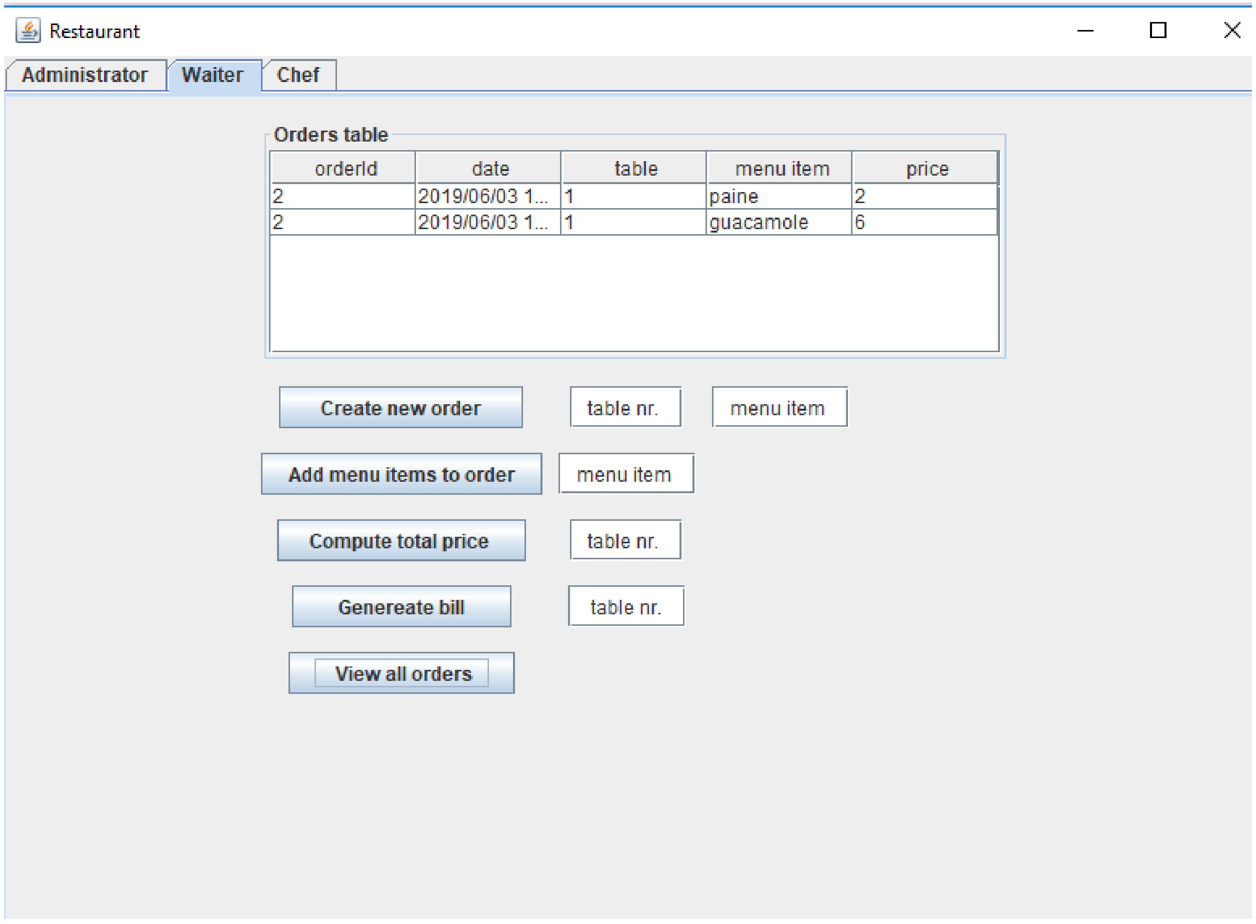
*AdministratorGUI class:* this class creates the interface the administrator uses for introducing and modifying the necessary and desired values in the menu and also showing the tables' records. It extends the JTabbedPane class, as it represents a tab in the main window. Its attributes are JButton, JTable, JTextField elements.

*WaiterGUI interface:* this class creates the interface the waiter uses for introducing and modifying the necessary and desired values in the orders table and also showing the tables' records. It extends the JPanel class, and it represents another tab in the main window. Its attributes are also JButton, JTable, JTextField elements.

*ChefGUI class:* this class represents the interface for the chef, which contains nothing but a label and a pop up message will appear on the screen when a new order is created.

## 5. Results

As we can see in the following picture, this is how the interface displays data with various information from the existent tables. This interface allows the users to modify the menu list or the hashtable structure with orders and menu items. It also shows a notification for the chef when a new order is created.



| orderId | date | table | menu item | price |
|---|---|---|---|---|
| 2 | 2019/06/03 1... | 1 | paine | 2 |
| 2 | 2019/06/03 1... | 1 | guacamole | 6 |

## 6. Conclusions

In conclusion, this type of application is in high demand and very popular with many restaurants and definitely helps the jobs of the restaurant's employees. It helps to keep track of tables' orders and overall, to manage the whole restaurant. It can further be improved by creating a log-in system, which lets each and every employee to work separately and not use the app together with the other waiters/chefs.

## 7. Bibliography

1. https://www.tutorialspoint.com/java/java_serialization.htm
2. https://www.vogella.com/tutorials/DesignPatternObserver/article.html
3. https://www.geeksforgeeks.org/composite-design-pattern/