

*Faculty of Automation and Computer Science*

**Restaurant Management System**

Project made by  
Ștefan Ciuprina

Contents

[1. Objective 3](#_Toc39802226)

[2. Problem analysis, scenarios, use cases 3](#_Toc39802227)

[3. Project Design 4](#_Toc39802228)

[3.1 Approach 4](#_Toc39802229)

[3.2 UML Diagram 4](#_Toc39802230)

[3.3 Packages 5](#_Toc39802231)

[3.4 Classes’ Design 5](#_Toc39802232)

[3.4 Data Structures 6](#_Toc39802233)

[3.5 User Interface 6](#_Toc39802234)

[4. Implementation 8](#_Toc39802235)

[5. Results 15](#_Toc39802236)

[6. Conclusions 18](#_Toc39802237)

[7. Bibliography 18](#_Toc39802238)

# Objective

The objective of this project is to create a restaurant management system program. The system has 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 secondary objectives are:

* Using the layered architecture method, splitting the project into 3 main parts: Data Layer, Business Layer and Presentation Layer.
* Using the Composite Design Pattern for defining the Menu Items;
* Using the Observer Design Pattern to notify the chef each time a new order is being made;
* Storing the restaurant related information using Map data structure;
* Implementing the Restaurant class using the Design by Contract method, having preconditions, postconditions and invariants;
* Saving/Loading the items from the Restaurant class to a file using Serialization;

# Problem analysis, scenarios, use cases

The functioning of this program will be as it follows: there will be three windows which could be open in parallel, namely one for the administrator, one for the waiter and one for the chef. The administrator will be able to add/delete/modify products, having complete control over the menu; then we have the waiter, which will be able to create orders, based on the products from the menu, and generate bills, and finally we have the chef, which will be notified each time a new composite product is ordered (something he has to cook, therefore not the base products, which are already ready to be served).

We have the following use case diagram:

Chef

Waiter

Administrator

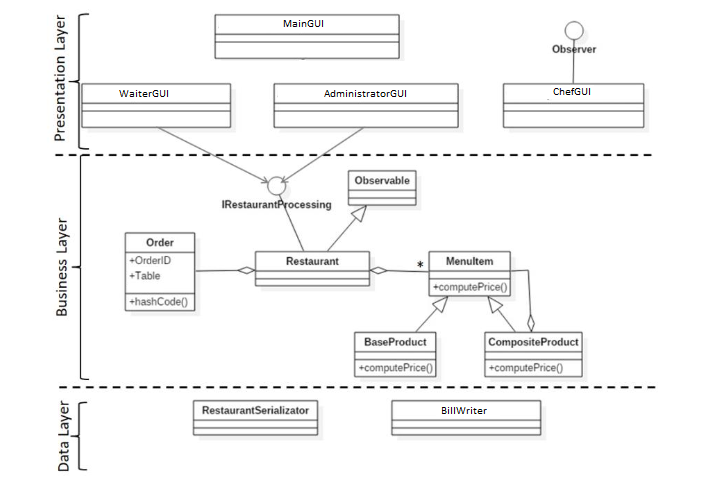
# Project Design

## Approach

As mentioned in **chapter 1**, our project follows the *Layer Architecture* technique. Therefore, there are three main packages in our project: *data*, *business*, *presentation*. There’s also a fourth package, called *start,* having only the main class which tells the presentation class to open the main window of the program. The presentation package gets the required data from the data package and together with the user input sends the required information to perform the tasks of the app to the business layer. The program loads and saves data from/to a file using serialization, presented in more detail in **section 4. *More*** information regarding the packages is presented in **section 3**.

## UML Diagram

The program was creating following the requirements specified, having only small changes in the naming of the classes and the creation of the new class *MainGUI*, the only through which the user can decide which window to open at the required time. Therefore, we have the following UML diagram:



The fields, the methods and the inner classes of the program are explained in detail in **section 4**, not being added to the diagram to make it easier to see.

## Packages

As mentioned before, our project is divided in the 4 packages:

* *start*: contains the *Main* class, used for starting the application. The only thing it does is creating an object of type *MainGUI*, which will open the program and let its execution follow from there;
* *presentation*: contains the *MainGUI, AdministratorGUI, WaiterGUI, ChefGUI*, classes representing the graphic user interface and for the connection between the user input, the data and the execution of each required task; this package also contains the *Observer* interface, which is implemented by the *ChefGUI* class, but more on that in **sections 3.4** and **4**;
* *business*: contains the *Restaurant, Order and Observable* classes, along with the *IRestaurantProcessing* interface, implemented by the *Restaurant* class, and the subpackage *menu*, itself containing another 3 classes: *MenuItem, BaseProduct* and *CompositeProduct*. This package handles how the data is stored and executes the operations needed by each user;
* *data*: has two classes, namely *BillWriter*, used for generating bills by the waiter, and the *RestaurantSerializator* class, used for serialization, saving and loading the instance of the class *Restaurant* to/from a file, making possible the save of the data in between the executions of the program.

## 3.4 Classes’ Design

There are a total of 13 classes and 2 interfaces in our project, each connected as needed with others to assure the correct execution of the program. The classes in the *presentation* package are the ones responsible for the user interface. They have been designed using *swing* method of creating user interfaces. The *MainGUI* class creates a window with 3 buttons, each one of them opening windows for the required user, namely the administrator (*AdministratorGUI*), the waiter (*WaiterGUI*) and the chef *(ChefGUI*). From there, each one of them deserializes the *Restaurant* class using the *RestaurantSerializator* class, or creates a new object of type *Restaurant* if the file doesn’t exist. Also, they collect data from the user by providing text fields and buttons for letting the user easily enter the required data. The *ChefGUI* class implements the *Observer* class since the information in this interface must be updated each time the waiter adds a new order containing *CompositeProduct* objects. This feature is realized using the Observer Design Pattern: the *Restaurant* class implements the *Observable*  interface, which lets it notify the *Observer* (namely, the chef) each time an order with a composite product is being added. Order related information is stored in the *Order* class, along with the *Restaurant* class, which, through a *Map*, makes the connection between each order and the menu items it has.

The menu items are defined in the *menu* subpackage of the *business* package, through the Composite Design Pattern technique. An abstract class is created, *MenuItem*, from which the *BaseProduct* and the *CompositeProduct* extend. A *base product* is composed only of a name and a price, while a *composite product* is composed of a name and a collection of other *MenuItem* objects (could be other base products or composite ones), and its price is computed as the sum of the prices of those objects.

## Data Structures

The data structures used for this project are:

-the *HashMap*, used for establishing a connection (a mapping) between each order and the menu items it has, this thing being implemented in the *Restaurant* class. Also, a *Map* is used for assuring no two menu items share the same name, having a map with the names as keys, and as the definition of the map states, there cannot be two map objects with the same key.

-the *ArrayList*, used by the *Observable* class to hold all the menu item names of composite products to report to the *Observer*, along with a *dates* ArrayList to hold the date for when each of the menu items is added to an order. The index of each element of the *dates* list corresponds to the element from the *menuItemsNames* list. For example, the element at index 2 in the *menuItemsNames* list has been created at the date specified in the element at index 2 in the *dates* list.

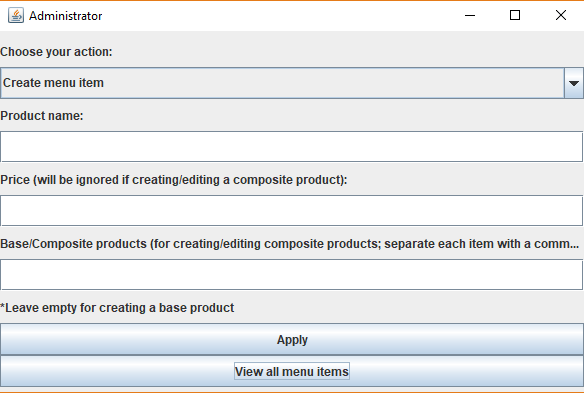
## User Interface

When the program is runned, a main window will be opened, having three butons, each one for opening the specific window for the user that is using the app. This main window won’t close after another window is opened, making possible the use of the app by multiple users at the same time. Still, closing this window will result in the termination of the program.

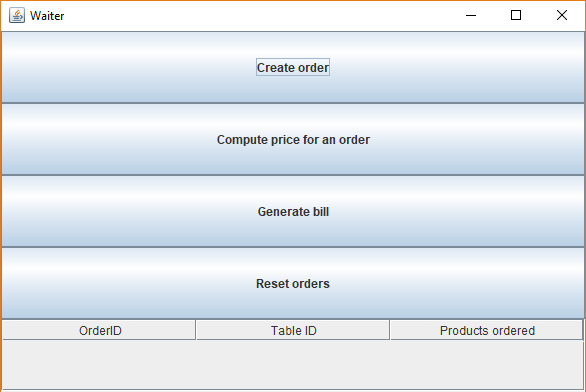


The administrator window contains a combo box for selecting the operation the administrator decides to do: create, edit or remove a menu item. Then, in the next text box, it must specify the name of the product he wants to create. In case of the creation/editing of a base product, the price will be specified and the next text box will be let empty. In case of a composite product, the price text box will be ignored, and the administrator must specify the products the composite product is made of in the third text box of the window, separating each item with a comma (no spaces). After entering all the data, the apply button must be pressed to execute the operation. The administrator also has the possibility to view all the items the restaurant holds at the moment using the *View all menu items* button.

\*To edit a product, the user will specify the existing product name in the first text box, then its new specific data in the following text boxes. If the specific product doesn’t exist, an error message will be displayed.

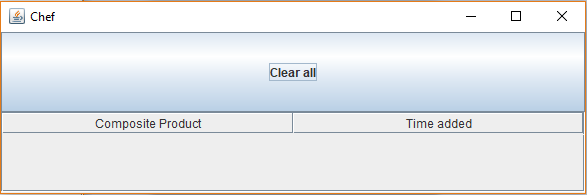


The waiter window provides a button for each waiter operation that could be done and a table containing the current orders created by the waiter. A reset option for the orders table is also available.



\*When a waiter generates a bill for a table, all the orders of that table will be deleted after the generation of the bill, “closing” this way the specific table and making letting other customers use it without keeping the table’s previous orders.

The chef window contains a table of the composite products added by the waiter in the orders he creates. The table will be updated each time the waiter creates a new order containing a composite product. The chef also has the possibility of clearing the contents of the table using the *Clear all* button.



# Implementation

* *Main* class:

Creates only an object of type *MainGUI* that opens the graphical user interface for selecting the type of user that will use the app.

public static void main(String[] args) {  
 new MainGUI();  
}

* *MainGUI* class:

Fields:

private JButton administratorButton;

private JButton waiterButton;

private JButton chefButton;

private AdministratorGUI administratorGUI;

private WaiterGUI waiterGUI;

private ChefGUI chefGUI;

Constructor:

public MainGUI() : instantiates the other GUI classes, creates three buttons, one for the administrator GUI, one for the waiter and one for the chef, along with listeners to set them visible when pressed.

Inner classes: listeners for opening the required window when a button is pressed:

private class administratorListener implements ActionListener {  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 administratorGUI.openWindow();  
 }  
}  
  
private class waiterListener implements ActionListener {  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 waiterGUI.openWindow();  
 }  
}  
  
private class chefListener implements ActionListener {  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 chefGUI.openWindow();  
 }  
}

* *AdministratorGUI* class:

Fields:

private Restaurant restaurant;

private JLabel actionLabel;

private JComboBox<String> actionComboBox;

private JLabel nameLabel;

private JTextField nameTextField;

private JLabel priceLabel;

private JTextField priceTextField;

private JLabel baseProductsLabel;

private JTextField otherProductsTextField;

private JLabel otherProductsTipLabel;

private JButton applyButton;

private JButton viewItemsButton;

Constructor:

public AdministratorGUI() : deserializes the restaurant object and sets up the view for the administrator.

Methods:

public void openWindow() – sets the window visible to the user

private void showError(String message) – shows an error message based on the String entered

private void showSuccess() – shows an success message

private String getOperation() – gets the operations selected by the user in the combo box;

private String getMenuItemName() – gets the menu item name entered by the user in a text box to insert/delete/update

private String[] getOtherProductsNames() – in case the user wants to create a composite product, it will specify the product names the composite product must contain, and this method gets the ones entered by the user (which he separates using a comma) and returns them in a string array

private double getPrice() – when the user creates a base product, it must specify the price in a text box; this method gets the price entered by the user and returns it as a double;

Inner classes:

private class ApplyListener implements ActionListener – implements the behavior of the program when the *Apply* button is pressed; updates the *restaurant* object and saves it through serialization;

private class viewItemsListener implements ActionListener – opens a new window containing a table of the menu items through the static class *MenuItemsTable*, implemented just below this class.

* *WaiterGUI* class:

Fields:

private JFrame frame;

private Restaurant restaurant;

private JButton createOrderButton;

private JButton computePriceButton;

private JButton generateBillButton;

private JButton resetOrdersButton;

private JTable table;

private final String[] columnNames ;

private JScrollPane sp;

private JPanel panel;

private Observer chef;

Constructor:

public WaiterGUI(Observer chef) : deserializes the restaurant object and sets up the view for the administrator.

Methods:

public void openWindow() – sets the window visible to the user

private void showError(String message) – shows an error message based on the String entered

private void showSuccess() – shows an success message

private void showPrice(double price) – shows a message dialog containing the price sent as parameter to the method

private void updateTable() – since the waiter GUI also contains a table (with the current orders), it must be updated each time a new order is being added. This method will be called each time a modification to the orders has been made in order to let the waiter see the modification in the table

Inner classes:

private class createOrderListener implements ActionListener – creates a new *createOrderFrame* class, which will pop up a new window, letting the waiter enter the order related information for the order he wants to create; after the order has been successfully been placed, the waiter can close this window and continue proceeding with other operations;

private class computePriceListener implements ActionListener – creates a new object of the inner class *computePriceFrame*, class implemented just below this one. This class will open a new window for letting the waiter input the order ID for which he wants to compute the price for;

private class generateBillListener implements ActionListener – creates a new object of the inner class *generateBillFrame*, class implemented just below this one. This class will open a new window for letting the waiter input the table ID for which he wants to generate the bill for; the bill will be generated by a call to the *BillWriter* class

private class resetOrdersListener implements ActionListener – the waiter interface also includes a fourth button for reseting all the existing orders, namely deleting them and resetting the order number count used for creating unique IDs. This inner class handles the resetting operation when the button is pressed, using the *restaurant* object and updating the table accordingly after deleting the orders.

* *ChefGUI* class

Fields:

private ArrayList<String> compositeProducts;

private ArrayList<String> dates;

private JPanel panel;

private JButton clearAllButton;

private JTable table;

private JScrollPane sp;

private final String[] columnNames;

Constructor:

public ChefGUI() – sets up the interface for the chef, namely a table for viewing all the composite products added in orders by the chef, and a button for clearing the table.

Methods:

public void openWindow() – sets the window visible to the user

public void update(ArrayList<String> compositeProducts, ArrayList<String> dates) – since this class implements the *Observer* interface, it must implement this method; this method will be called by the restaurant each time a new order containing a composite product is being added for making possible the update of the table in the chef window in real time, following the Observer Design Pattern. This gets the required data to update the table and calls the *display* method, which will actually update the table in the window.

private void clear() – clears the table of the composite products in the chef window

Inner class:

private class clearAllListener implements ActionListener – clears the table in the chef window with the help of the *clear* method and the *restaurant* object.

* *Observer* interface – is implemented by the *ChefGUI* class for realizing the Observer Design Pattern; defines the method *update*, which has been detailed above in the implementation details of the *ChefGUI* class
* *RestaurantSerializator* class – used for serializing the restaurant object for saving and loading its contents to/from a file, keeping the data in between the uses of the program
* *BillWriter* class – contains only a static method *printBill*, which creates a txt file containing the bill information given from the *Restaurant* class.
* *Restaurant* class

Fields:

private Map<Order, Collection<MenuItem>> orders;

private Map<String, MenuItem> menu;

private int numberOrders;

* This class implements the *IRestaurantProcessing* interface, containing the methods for realizing the execution of the operations for the administrator (create,delete,edit a menu item) and for the waiter (create order, compute price, generate bill). Those methods are explained below, when detailing the implementation of the *IRestaurantProcessing* interface. Along this methods, it has a few extra ones, namely the (“well formed”) class invariants checkNullPrecondition(Object o) and private static void checkIDPrecondition(int ID), used for the implementation of the program using the Design by Contract method. Other methods of this class are:

public MenuItem getMenuItem(String menuItemName) – gets the menu item object from the map based on the name given as parameter

public void resetNumberOrders() – sets the number of orders to 0

public String[][] getMenuString() – returns a string array of the names of the menu items in the menu *Map;* this method is used when creating tables in the interfaces

public String[][] getOrdersString() – same as *getMenuString()*  method, but for orders

public void deleteAllOrders() – deletes all orders from the orders *Map*.

* *Order* class

Fields:

private int id;

private int tableID;

* This class holds information regarding the order, namely the ID and the table ID assigned for this order. In order to make each order unique, it makes two order objects appear the same if they share the same id. This has been done by overriding the *equals*  and *hashCode* methods. This class also provides getters for the id and the table id.
* *Observable* class:

Fields:

private ArrayList<String> menuItemsNames;

private ArrayList<String> dates;

* This class is extended by the Restaurant class, for implementing using Observer Design Pattern technique. Its fields hold information about what needs to be displayed in the chef’s window, fields that are updated once the *notifyObservers* method is called by the *Restaurant* class. This method calls the *update* method of the *Observer*. The *Observable* class also provides getters for its fields and a method *clearAll* to clear all the strings stored in its fields.
* *IRestaurantProcessing* interface

Provides the methods needed for executing the operations for the administrator (create, delete, edit an order) and the waiter (create order, compute price, generate bill). Using the JavaDoc comments present in this interface, it is shown the implementation using the Design by Contract method. The classes present preconditions, postconditions and use of invariants. The works of those methods are explained in this manner and can be viewed in the JavaDoc documentation. The methods this interface defines are:

boolean createMenuItem(MenuItem menuItem);

boolean deleteMenuItem(String name);

boolean editMenuItem(MenuItem menuItem);

boolean createOrder(Observer chef, int tableID, Collection<String> menuItemsNames);

double computePrice(int orderID);

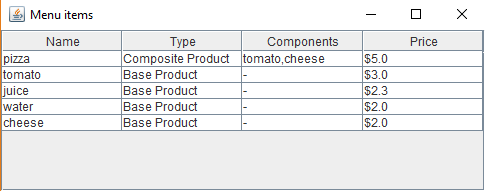
boolean generateBill(int tableID);

* *MenuItem, BaseProduct, CompositeProduct* classes
* These classes are in strong relationship, following the Composite Design Pattern implementation method. The *MenuItem* class is an abstract class, holding the fields for the name and the price for the menu item and defining the abstract methods *computePrice* for getting the price and *getMenuItems* for getting the menu items in case of a composite product. The *BaseProduct* and *CompositeProduct* classes extend from this abstract class, the base product being able only to set the name and the price for a base product, while the composite product being able to hold a collection of other products (other *MenuItem* objects, either base or composite products), setting a name and computing the price as the sum of all the products it is composed of.

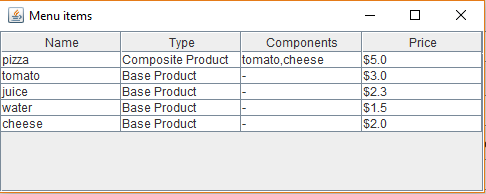
# Results

After the implementation of the project has finished, a jar file was created to make possible the running of the program in the command line. There, the workings of the program have been tested as follows:

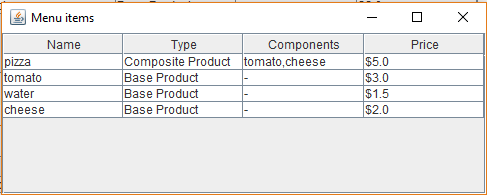
The administrator created the following products and viewed them in the table as shown below:



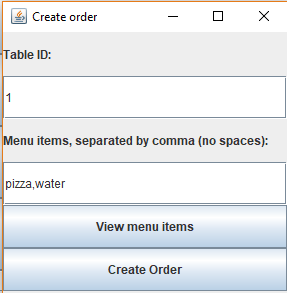
Then, the administrator edited the product “water”, making it cost $1.5.



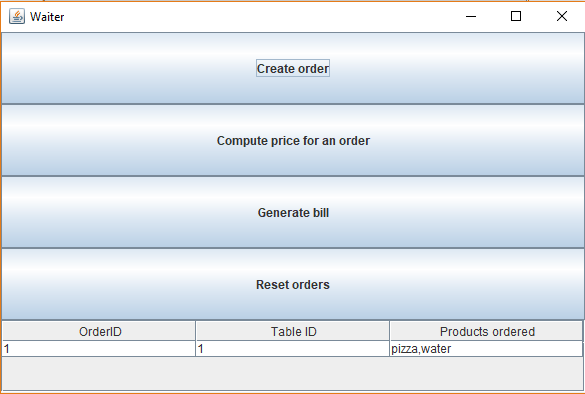
Then, the administrator deleted the product “juice”.



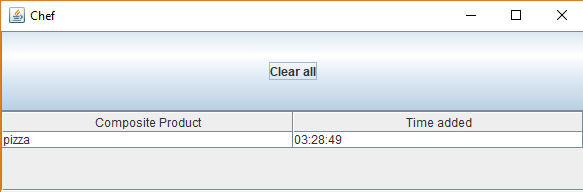
The waiter created an order for table 1, of a pizza and a water.



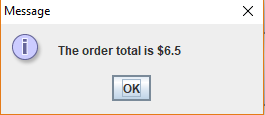
The result were automatically updated in the orders table.



Since a composite product has been added to an order (“pizza”), the chef was immediately announced in the chef window.

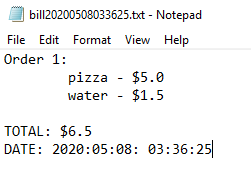


The waiter then wanted to see the price of the current order, so he clicked the “Compute price for an order” button and specified order ID 1.



The order total is correct, since the order is composed of a pizza ($5) and a water ($1.5), the total being $6.5.

The waiter then generated the bill for table 1, which was outputted in a txt file, containing all the orders of the specific table (in our case, only order ID 1).



The chef also cleared all his orders, a thing good to do after he finished preparing all the items requested by the waiter.

As it can be seen above, the program behaved as expected.

# Conclusions

All of the objectives presented in **section 1**have been achieved, the final form of the project being the wanted Restaurant Management System app, capable of handling the operations required in a restaurant, having a digital menu set by the administrator from which the waiter can create orders and generate bills, and through which the chef could know when and what to prepare for the customers. By working on this project, I learned to apply the Composite Design Pattern, Observer Design Pattern and Design by Contract methods and also have improved my skills using java data structures such as the *HashMap*. Further improvements could be done to this project in the future, such as support for multiple admins, waiters and chefs through a login process, assigning images to each product and many more.

# Bibliography

* Assignment specification:
* <http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_4/Assignment_4.pdf>
* HashMap, Design by Contract, Serialization:  
  [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 )
* Composite Design Pattern:  
  <https://www.geeksforgeeks.org/composite-design-pattern/>
* Observer Design Pattern:  
  <https://www.geeksforgeeks.org/observer-pattern-set-2-implementation/>
* Custom JavaDoc tags:  
  [https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#tag](https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html%23tag)