Technical University of Cluj-Napoca

Fundamental Programming Techniques

**RESTAURANT MANAGEMENT SYSTEM**

ASSIGNMENT 4

Tilea Anda Corina

Group 30421

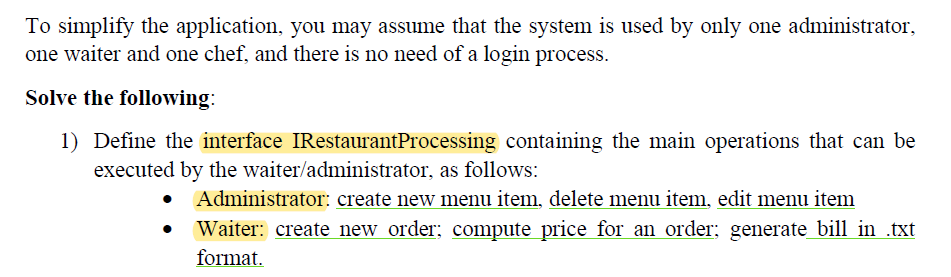
**Contents:**

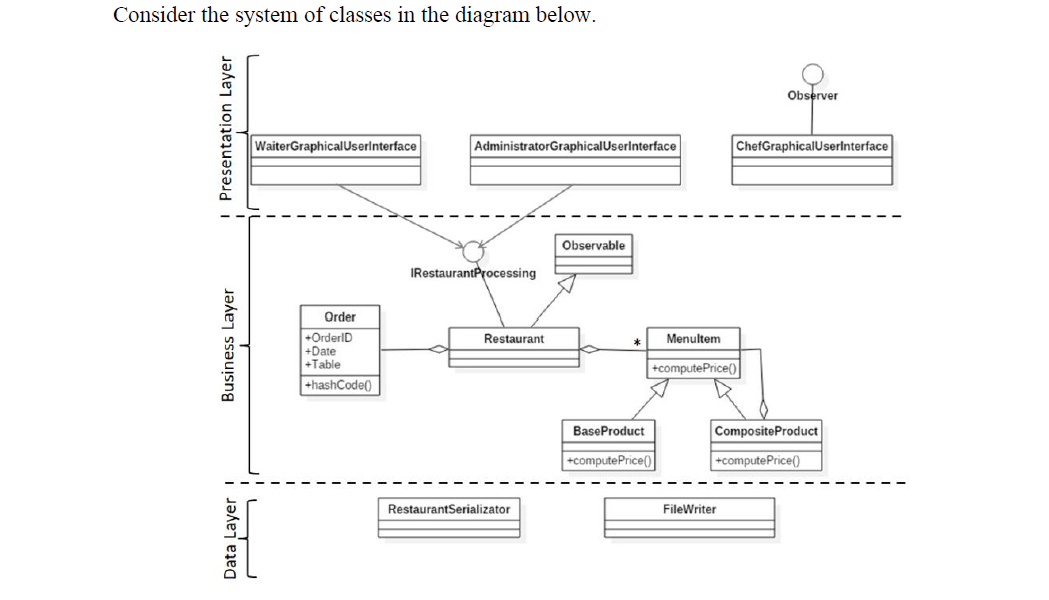
|  |  |
| --- | --- |
| 1.Problem’s Objective | pg3 |
| 2.Problem’s Analysis& Approach | pg3-7 |
| 2.1 Use Case | pg7 |
| 2.2 Scenarios | pg8-10 |
| 3. Design | pg10-11 |
| 3.1 UML Diagram | pg11 |
| 4. Classes and Packages | pg12 |
| 4.1 Implementation | pg12-19 |
| 5. Testing | pg19-20 |
| 6. Further Development | pg21 |
| 7. Conclusions | pg21 |
| 8. Bibliography | pg21 |

1. **Problem’s Objective:**

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 that is ordered through a waiter.

The entire Application will start from the following premise:





1. **Problem’s Analysis and Approach:**

The purpose of this Application is to resemble a real-life Restaurant with three main components, each one receiving a proper user interface. In our case, these components are represented by the Administrator (which is in charge of the entire Menu), the Waiter (in charge of the customer’s orders, and computing the bill, where needed), the Chef, which is notified each time an order is created in order to prepare it.As an approach:

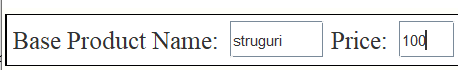
* **Administrator:**



* Describing in detail, the Administrator’s tasks are the essential component of a Restaurant. This kind of employee is able to create Menu Items, which are, in our case, of two types. The Menu may contain - **Base Products**: a simple product, served just as it is, being composed of nothing more than itself. These **Base Products** represent the raw items for the Menu of the Restaurant. It is assumed that all the product’s names in this Restaurant contain only one word.
* In order for the Administrator to create such a product, a specific name must be chosen:



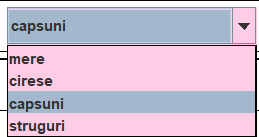
* Then, the Administrator must choose a price, for the current product which is about to be created;

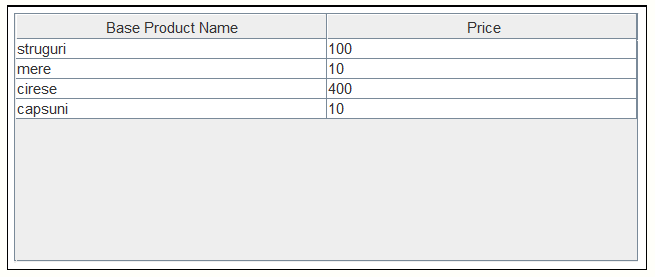


* The final step, in creating a Base Product is adding it to the List, using the implemented “Add” button.



* All the Base Products will be shown in a table for a better visualisation.

****

****

The Menu may also contain**Composite Products**: more complex items, being composed of more than one ingredient.It is assumed that all the product’s names in this Restaurant contain only one word.

* Each **Composite Product** will be added by their name, so the Administrator must enter a specific name:



* Then, the Administrator must choose from the List of products already created, which ingredients to use. In this List, all the created items (both Base and Composite) are displayed.
* The products can be selected from the List using the “Select” button. If an ingredient is selected by mistake, it can be removed using the “Remove” button.



* A **Composite Product** may contain:

-2 or more Base Products / one Base Product and a previously createdComposite Product / 2 or more Composite Products already created;

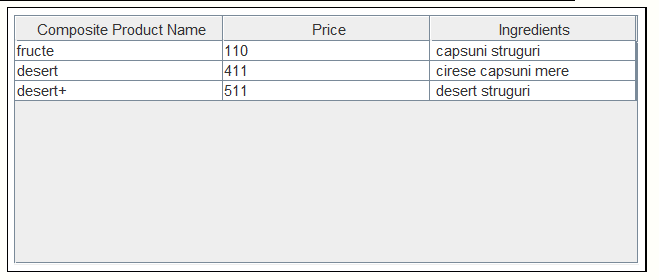
* The final step, in creating a Composite Product is adding it to the List, using the implemented “Add” button.



* The price for any **Composite Product** is computed by the Application using the price for each ingredient used. All the data will be shown in a table for a better visualisation.

Only Base Products

Base Product + another Composite Product



Furthermore, the Administrator will be able to do some other Operations on Menu Items, such as: Deletion and Edit an already created product. Each one of this Operations is performed only after pressing the corresponding buttons.

* For **Base Products**: the Administrator is allowed to edit and delete **Base Products**based on a criteria: a **Base Product** cannot be edited/deleted if it already is a part of a Composite Product.More examples for these particular cases will be described in the “Scenarios” Section. This kind of products are edited based on their price.



* For **Composite Products:** the Administrator is allowed to edit anddelete**Composite Products**, following a criteria: a Composite Product cannot be edited/deleted if it already is a part of another Composite Product. An example for this case will also be described in the “Scenarios”. This kind of products are edited based on their ingredients list.

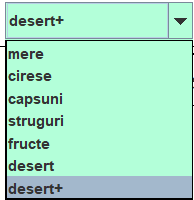


* **Waiter:**



* Describing in detail, the Waiter’s tasks are also essential in order for the Restaurant to work. This kind of employee is able to **create Orders**, by selecting the Menu Items from the menu List.
* The Menu Items can be selected from the List using the “Select” button. If an item is selected by mistake, it can be removed using the “Remove” button. All the selected Menu Items will be displayed in the form of a List.

C:\Users\user\AppData\Local\Microsoft\Windows\INetCache\Content.Word\20.png



* In order **to create the Order**, the Waiter must receive the wanted menu items,
  + - the Table number at which the clients are staying:



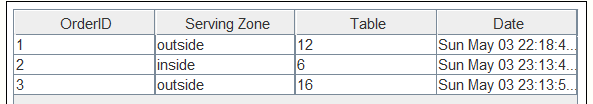
* + - the Location in which the command should be served



* The final step, in creating an Order is pressing the“Create Order” button.



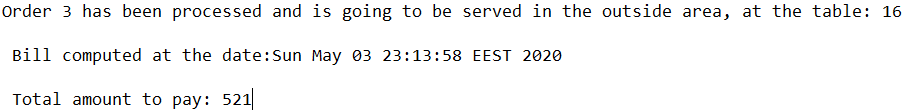
* All the Orders will receive a unique ID, which represents an identifier. The Restaurant’s information regarding the Orders will be represented more easily using a Table:



The Waiter is also responsible for computing the bill when it is requested. The employee must press the “Compute Bill” implemented button, and a bill containing the order’s id, the location, the table’s number, the date and the total amount to pay, will be generated in a “.txt” format.



The name of the output-text file will contain “bill” + order’s ID:



* **Chef:**



* The Chef represents the employee which will have to cook the food. The Chef will receive a notification, created by the Waiter, every time a new Order is given.



2.1 Use Case:

The Application will contain three different type of employees. From these types of users, only two of them can interact directly with the Application: Administrator and Waiter. They are capable of entering data in the specific text fields and execute tasks.

The Administrator can add a new menu item, edit, delete an existing item, select/remove the ingredients wanted and visualise all the products (base or composite) in the corresponding table.

The Waiter can select/remove any items from the menu list, create an order, compute the bill and visualise all existing orders, in the table. The Chef is not able to interact directly with the Application. This employee can only be notified when an Order is created.

2.2 Scenarios:

1. Use Case: <Simulation>

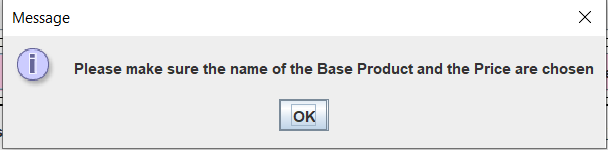
Primary Actor: User

Main success scenario:

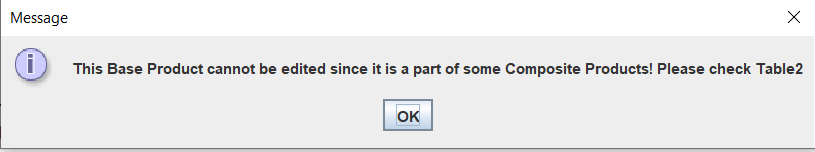
* The application is launched successfully by running the .jar file created;
* The user choses the Administrator’s interface, in order to create the Restaurant’s Menu;
* The wanted number of Base Products is added successfully, by entering their name and the price. They can be visualised due to the Base Product’s table;
* If the entered price is incorrect, the “Edit” button is used and the user can see in the s table, the updated price;
* When a Base Product is no longer needed, it is deleted;
* The list contains all the created products. They can be selected for creating a Composite Product. Also a name for this item is needed.
* If a Composite Product is no longer needed, it is deleted;
* The menu List is presented in the Waiter’s interface. The employee will select the required items, by selecting them.
* The Clients must choose a serving location and a Table, and the Waiter creates the Order;
* When required the Bill will also be computed, and generated in a output-text file;
* All the orders with their specific characteristics will be visualised in the Order’s table;
* When the user changes to the Chef’s interface, the notifications for each created order, along with the serving’s specifications will be available.

1. Alternative Sequences:

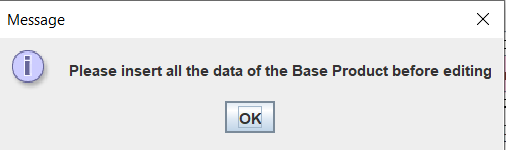
* If the user tries to **add a Base Product**, without entering a correspondingname and/or a price, a notification will be displayed and no product will be added.



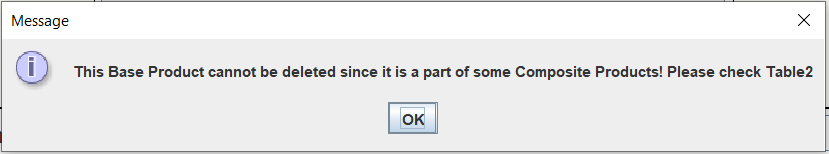
* If the user tries to **edit a Base Product**, which is already an ingredient for a previously created Composite Product, a notification will be displayed and the product will not be edited.



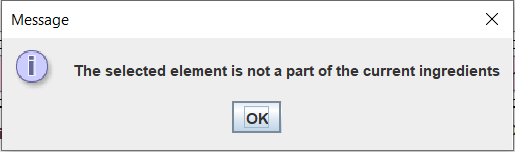
* If the user tries to **edit a Base Product**, and the price field / name field remains empty (no value is entered) a notification will be displayed along with an errorand the product will not be edited;



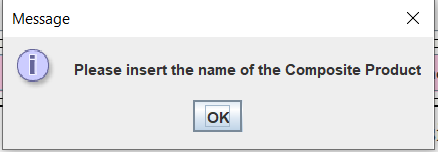
* If the user tries to **delete a Base Product** which is already an ingredient for a previously created Composite Product, a notification will be displayed and the product will not be deleted.



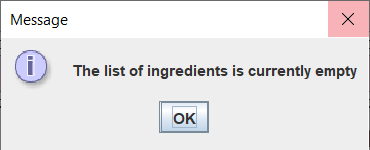
* If the user tries to **remove an ingredient** from the list, but that ingredient isn’t a part of the selected products, a notification will be displayed.



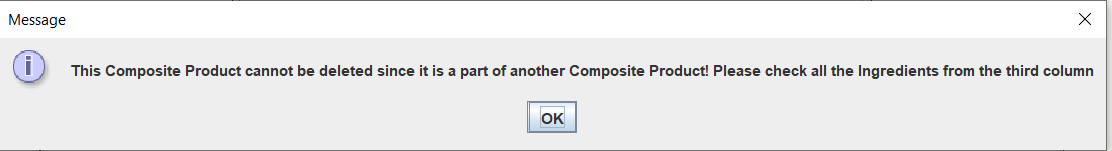
* If the user tries **to add a Composite Product**without entering a name,a notification will be displayed and the product will not be added.



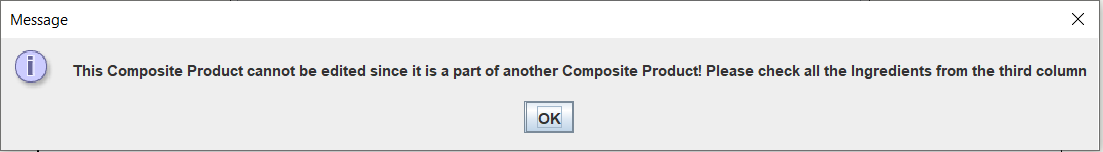
* If the user tries to **add a Composite Product**without selecting any ingredients, a notification will be displayed and the product will not be added.



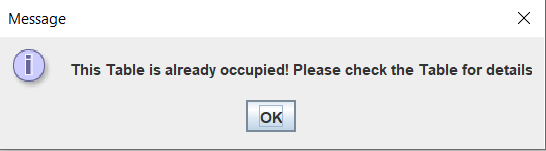
* If the user tries to **delete a Composite Product** which represents an ingredient for another Composite Product, a notification will be displayed and the product will not be deleted;



* If the user tries to **edit a Composite Product** which represents an ingredient for another Composite Product, a notification will be displayed and the product will not be edited;



* If the Clients **request a Table** which is already occupied a notification will be displayed and another Table has to be chosen.

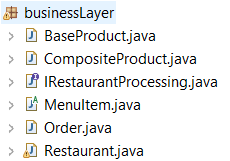


* If any other text fields are left empty/the entered data type is not the correct one, errors will occur and the task will not be performed.

1. **Design:**

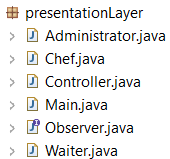
This Project will present two major parts. Firstly, the logic of the Application, and then the graphical user’s interface, the direct way of communicating between the user and the Restaurant related tasks. In order for the Application to run correctly and perform all the commands given, both parts must be well performed.

The Application is structured in packages, using a layered architecture, each layer presenting a special purpose. The “business” Layer: contains the BaseProduct, CompositeProductclasses which represent all the existing menu items. These two different forms will extend the abstract Class MenuItem, the general concept of the products, following the composite design pattern. The layer also contains the IRestaurantProcessinginterface and the Restaurant class, which represent the logic of the Application. The Restaurant will contain a list of products representing the entire menu and a Map (HashMap created for mapping each list of ordered items to their specific order). Also, the Order Class is implemented. Here, the proper implementation of the Hashmap is developed. So, two specific methods will be overridden. This methods “hashCode()” and “equals()” are responsible for generating the hash code in order to know where to place the mapping, respectively an equal method, for helping the user determine if two orders will be equal.

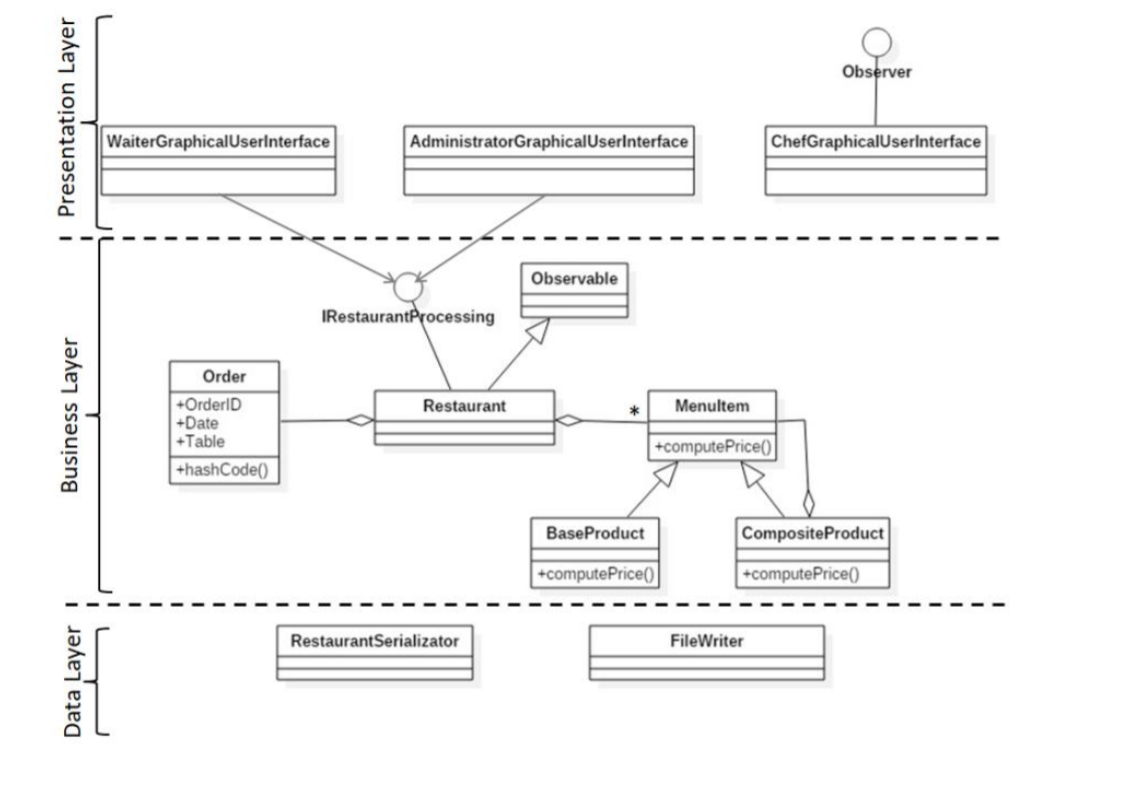


The “data” Layer contains the FileWriter Class. This Class is created and implemented for displaying the final “output” of an Order. When requested, for each Order a particular bill will be generated, containing the details needed. The bill will be generated in .txt format and can be visualised in the Project Folder, having a particular name for each Order computed. The name-format for any bill is the following: bill + “the unique identifier of the current Order”.txt. C:\Users\user\AppData\Local\Microsoft\Windows\INetCache\Content.Word\47.png

The “presentation” Layer contains all the GUI Classes, specially created for the three types of users: Administrator, Waiter and Chef. They are created particularly for the needed tasks of each employee and for a better visualisation of how the tasks are performed. Also, in this Layer the Controller Class is implemented, as an ‘Extra Class’ for implementing the specific functionality obtained by pressing the interface’s buttons. In some cases, the product list needs to be updated both for the administrator as for the waiter. In such situations, the Controller is used for assuring the communication between classes. This layer will also contain the Main Class, responsible for launching the entire application.



* 1. UML Diagram:



1. **Classes and Packages:**

* Data Structure used:
* ArrayList: this type of list is responsible for a better storage and accessing data. It uses **dynamic arrays** to store the elements. If any element is removed from the array, all the bits are shifted in memory.

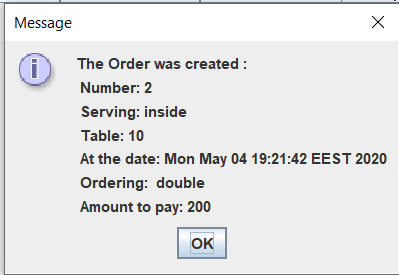
4.1 Implementation:

This chapter will provide a closer look at the role and the implementation of each class, with its attributes, constructors and methods. Some of the details can be also found in the JAVADOC files generated. The application will contain 13 classes, each one belonging to a specific package.

* **BaseProduct Class (businessLayer Package):**this Class will extend the MenuItem abstract Class. It models a simple, basic, product in the restaurant’s menu. (ex. tea, coffee).
* This class will contain one Field: theprice field;
* This class will contain one Constructor: it creates a new object, using the parameters given. The “super” keyword is used for referring to the parent’s class object(in this case, the name);
* This class will contain one Method: it overrides the abstract “Compute Price” method and implements this method in order for the Application to correctly calculate the price in the base of a single (base) product.
* **CompositeProduct Class (businessLayer Package**): this Class will also extend the MenuItem abstract Class (following the design pattern). It models a more complex product built from several ingredients. These ingredients can be either two or more Base Products, some Base Products and another Composite ones, or only Composite Products. So this type of menu item, will contain a list of products after which the correct price will be generated.
* This class will contain the same one Field:a list of menu items, representing the used ingredients, is generated.
* This class will contain one Constructor: it creates a new object, using the parameters given.
* This class will contain one Method: it overrides the abstract “Compute Price” method. In this case, a different implementation is required, since every ingredient must be taken into consideration. So, this method will iterate through all the products and add the component’s prices.
* **MenuItem Class (businessLayer Package**): this Class will incorporate the entire concept of the restaurant’s menu items, being declared as abstract. The two types: base and composite will be implemented by extending this general class.
* This class will contain one Field: the name field (this will be declared as a common fields in both of the child classes). The “protected” type assures the visualisation done only by its children.
* This class will contain one Constructor: it creates a new object, using the parameters given.
* This class will contain the abstract method “Compute Price”. It needs to be implements by both children, having different methods for computing the correct price. Also, the abstract Class contains a getter method for returning the current item’s name.
* **IRestaurantProcessing Interface (businessLayer Package**): this Interface declares all the methods needed for computing the employee’s tasks. Firstly, the administrator will receive as a task: to create, edit and delete a menu item. Then, the waiter will receive the “create order” and “compute Price” and “generate bill” tasks. In the defined interface, a getter is declared and will be implemented, in order to track permanently the menu list of the restaurant, after any kind of changes take place.
* **Restaurant Class (businessLayer Package):** this Class is responsible for implementing all the interface’s methods. It truly represents the center of this Application. Every action will have to pass through the restaurant or it is executed by this class.
* This class will contain three Fields: a list which will contain all the menu items already created, the HashMap, and an Order parameter. The HashMap is used in order to map the ordered items to the specific order they belong.
* This class will contain one Constructor: it will initialize the fields already declared;
* This class will contain overridden methods from the previously described interface. These methods will be used in the GUI’s implementations. The particular “body” of each method will test some conditions, using “assert”.
  + - * Creating a menu item: this method is responsible for adding a new item to the menu list of the restaurant. It takes as a parameter a list of ingredients, in order to decide if the added product represents a Base or a Composite one. If the list contains some elements, then the user tries to insert a Composite Product made by several ingredients.
      * Editing a menu item: this method is responsible for editing an already existing menu item, by using its name. The Application checks the entire menu, and if the wanted name is found, then the product will be updated with the new information assigned.



* + - * Deleting a menu item: if the user receives the task to delete one of the menu’s products, the wanted name will be searched in the entire menu. If it is found, then that specific product with all its attributes will be removed.
      * Creating an Order: this method will takes as parameters all the values needed by the Waiter in order to be capable of computing a client’s order (the id of the order, the serving location, the number of the table, and the current date). It finally will create an Order along with its attached list of menu items. The Application will make use of the HashMap implemented.



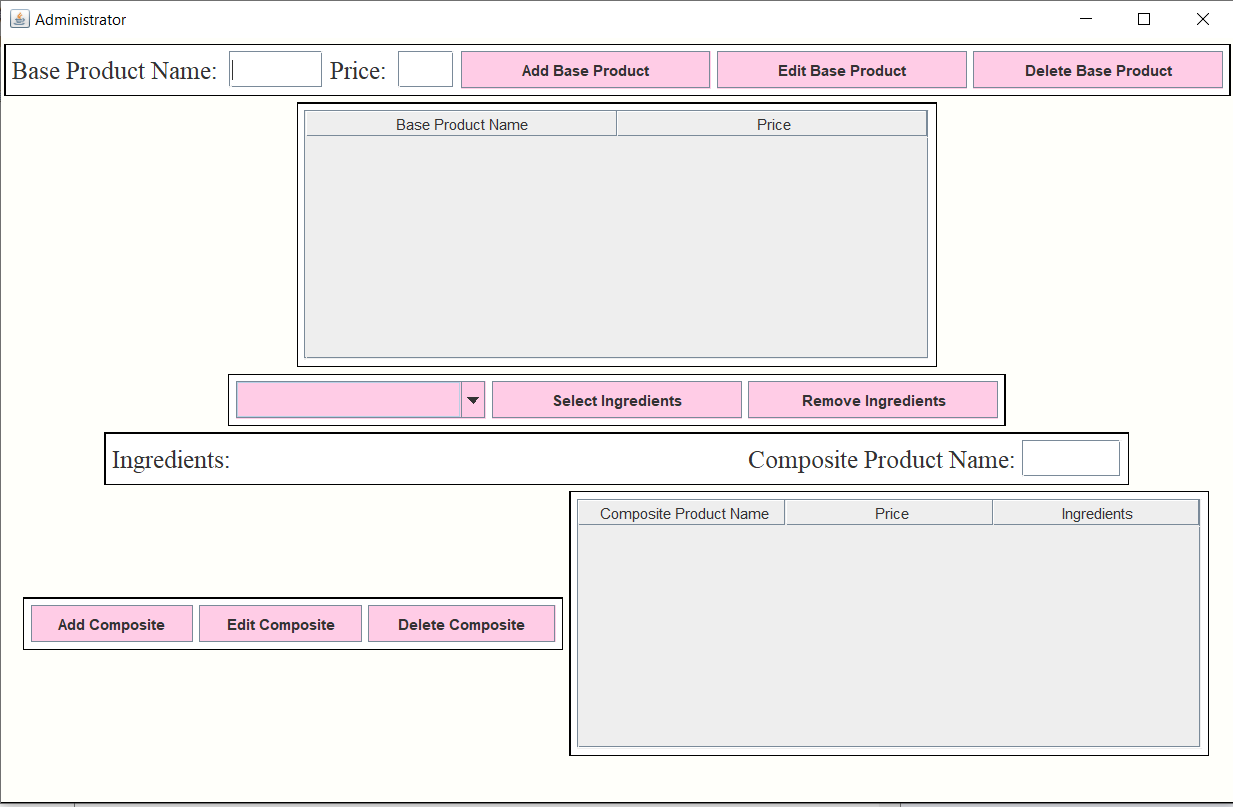
* + - * Computing the Price: the method will take each product wanted and will compute the specific sum, in order to find the total price for the current order. Each Base Product will have its own price. The Composite Products have the total price calculated from their ingredients. Their current values will be added in a sum, and the result will represent the total amount to pay.



* + - * Generating the text-format bill: this method will make use of the FileWriterClass and its implemented method, for which the order bill will be generated as an output and it will be saved in the Project Folder as a “.txt” file.



* **Order Class (businessLayer Package):**this class is in charge of the Waiter tasks. It represents the most important part of the job of this employee. It contains all the order related parameters: id, location of serving, table number and date.
* This class will contain four Fields: the order id, which represents a unique identifier, the location (inside/outside) in which the Client wants to stay, the table number at which he/she stays and the current Date at which the Order was computed.
* This class will contain one Constructor: it will create a new object with the given parameters;
* This class will contain four getter methods, in order to obtain the value for the specific order’s parameters, later when the bill must be computed in the user’s interface. Also, this class will contain two overridden methods, used by the mapping function, “hashCode()” and “equals()”.
* **FileWriter Class (dataLayer Package):**this class is responsible for generating any order’s bill. The result of this Class, will be generated as an output text file.
* This class will contain one void- method: createBill(), containing all the needed parameters for displaying an Order’s details. It creates a try-catch block in which a buffered reader will generate a message regarding the order id, the location where it should be served, the table and most importantly, the total amount a Client has to pay.
* **AdministratorGUI Class (presentationLayer Package):**it represents the specific interface in which the employee with the role of an Administrator performs the proper tasks. Here, the creation of a new product, the deletion of an existing one, the editing of an already created product, and the possibility of selecting/removing among the list of total elements, is possible. The Administrator is also capable of viewing all the existing products (base + composite) in their own JTable created. The first part of the interface, represents a place for computing the base products, meanwhile, the second part represents the composite product’s area.



* This class will contain as Fields:
  + **Buttons**: this application uses eight buttons, one for each operation. The user will need to implement a method in which if a specific button is pressed, a specific command is realized.



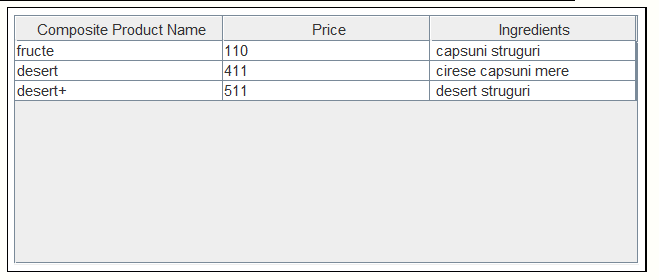
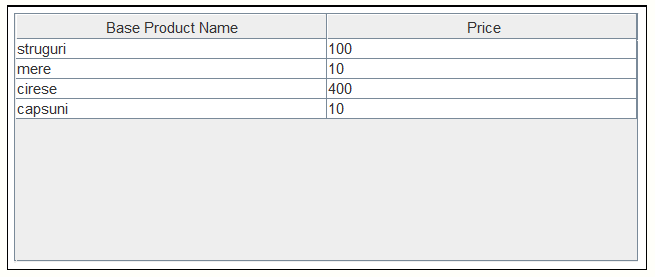
* + **Text Fields**: this application uses three text fields. The first two text fields are used in order to type the inputs regarding a base product, meanwhile the third one represents the name of a composite product. Using Restaurant class methods, all the strings are processed. After the operations are done, for example, after creating a new menu item (base/ composite), it will be shown in the specific table of the interface, and also, in the combo box implemented, representing the entire list of already created menu items.

Label

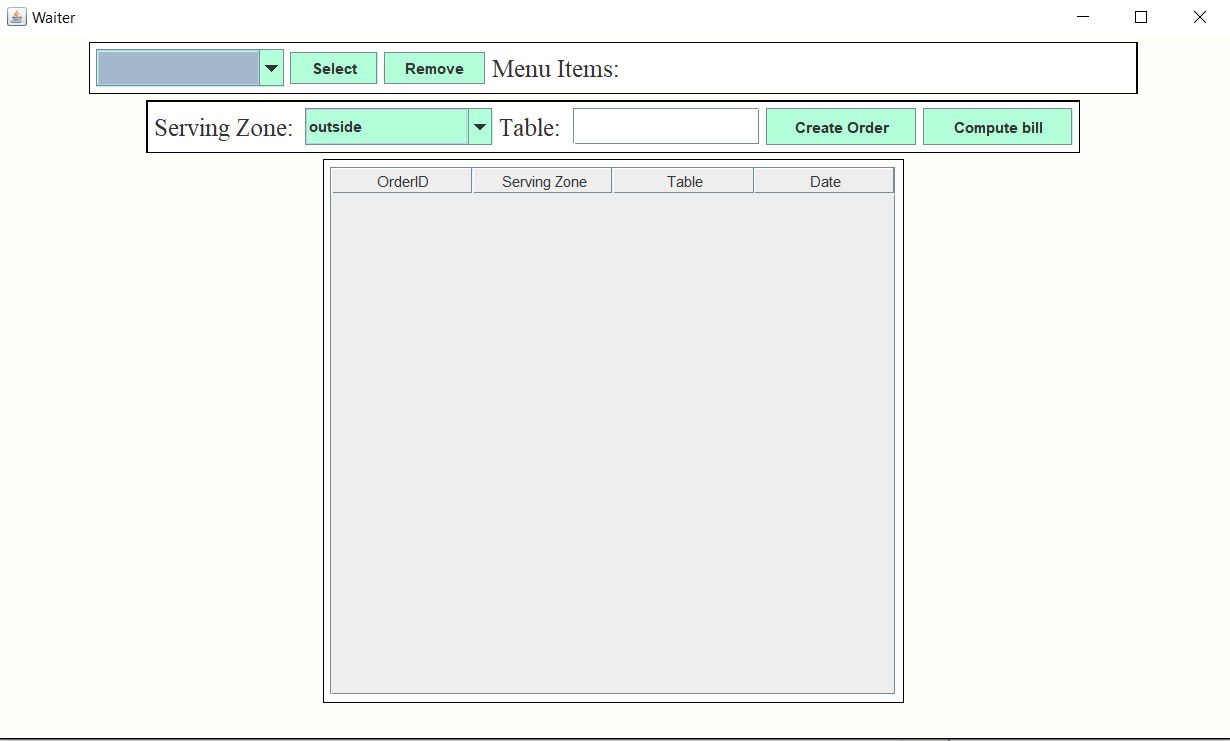
Text Field



* + **Labels**: this application useslabels. The first two labels are used in order to name the two fields for which the user must type the input data. Next on, the “Ingredients” labels, indicate that the following space is dedicated to the list of selected ingredients for a composite product. The last label represents the title for the text field in which the user must provide a name, in order to create an item or to edit/delete any existing one.
  + **Frame**: this application uses one frame, in which the panels is found. All the other components will be added in the specific panel. More panels are used in order to create a clean, organized work space for the user to perform the tasks.
* This class will contain one Constructor: here, all the elements of the graphical user’s interface, like buttons, text fields, labels will be added to the corresponding panels, and the setting functions declared will be called (**set Panels()**: it chooses the Layout, the background colour and the border for each panel involved; **set Components():** it chooses the Font and size for each label, button and text field from the interface; **set TableBase()**: it represents a setter for the base products Table, it will contain two columns using as a header the name and the price; **set TableComposite():** it represents another table setter, this time for composite products, it will contain three columns, using as a header the name, the price and the list of ingredients.



* This class will contain a method for each operation which needs to be done. The methods declared here, will be accessed using the Controller Class, creating an action listener for each button and some of results will be working together with a few tasks found in the Waiter’s interface. These methods are used for describing the functionality of each created button.
* **WaiterGUI Class (presentationLayer Package):**it represents the user’s interface in which the employee with the role of a Waiter does the job. The tasks allowed in this interface are represented by the creation of an Order (adding all the needed information: selecting a location, adding a table’s number), the computation of the corresponding bill and also selecting the wanted menu items from the menu list. The Waiter will also be responsible for the notification send to the Chef. This “message” is transmitted each time a new order is added.



* This class will contain as Fields:
  + **Buttons**: this application uses four buttons, one for each operation. The user will need to implement a method in which if a specific button is pressed, a specific command is realized. C:\Users\user\AppData\Local\Microsoft\Windows\INetCache\Content.Word\55.png



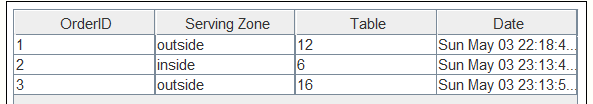
* + **Text Fields**: this application uses one text field. It is used in order for the user to directly interact with the Waiter’s interface and enter an integer number, which will represent the table at which the clients will be served.

Label

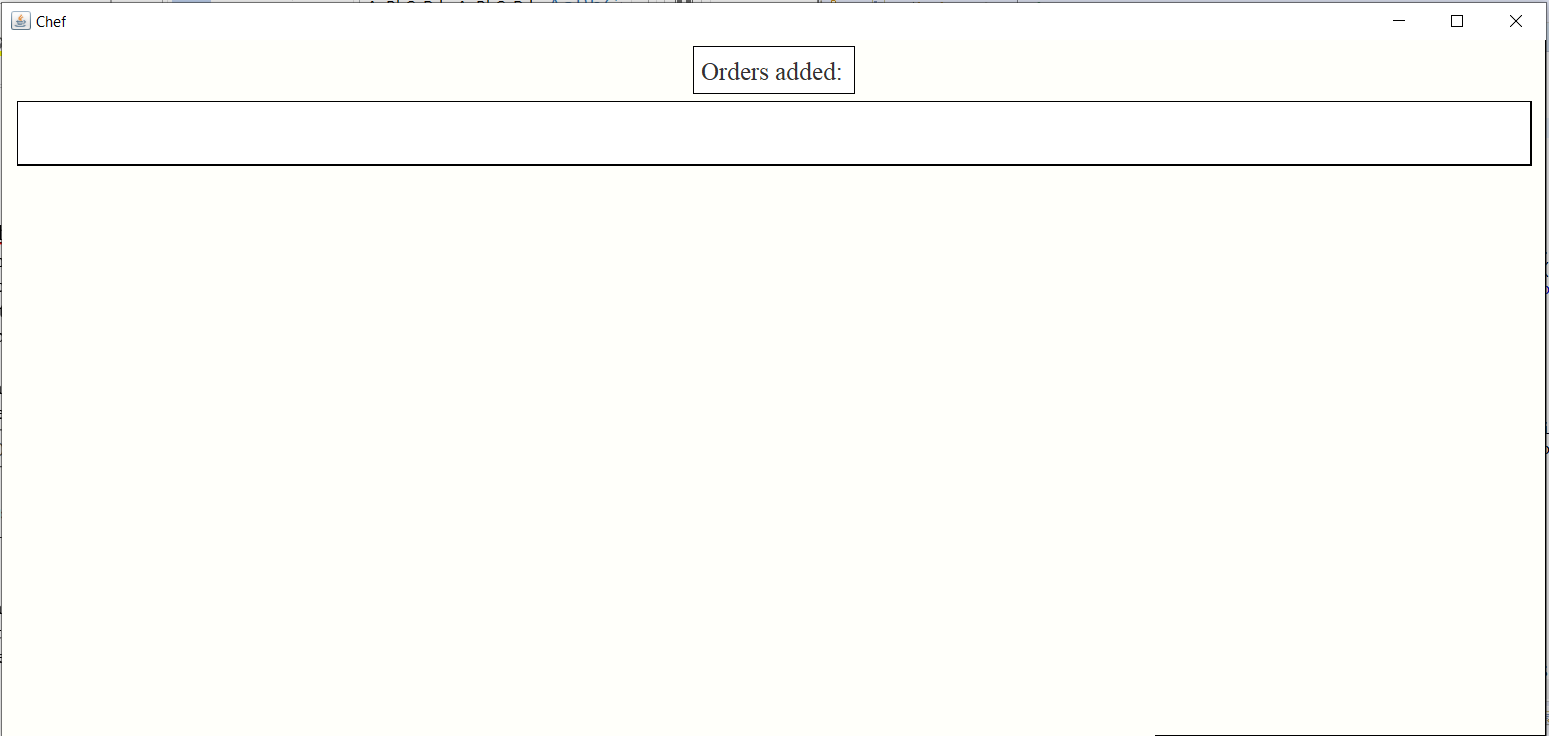
Text Field



* + **Labels**: this application uses three labels. The first one indicates that the following space is dedicated to the list of selected menu items, ready to be ordered. The next label represents the title for the combo box in which the available choices for a location are available, in order for the user to choose the desired one. The last label represents the title of a text field, presenting another detail related to the successfully computation of an order: the table at which the command should be taken.
  + **Frame**: this application uses one frame, in which the panels is found. All the other components will be added in the specific panel. More panels are used in order to create a clean, organized work space for the user to perform the tasks.
* This class will contain one Constructor: here, all the elements of the graphical user’s interface, like buttons, text fields, labels will be added to the corresponding panels, and the setting functions declared will be called (**set Panels()**: it chooses the Layout, the background colour and the border for each panel involved; **set Components():** it chooses the Font and size for each label, button and text field from the interface; **set Table()**: it represents a setter for the orders Table, it will contain four columns using as a header the order id, the serving zone, the table and the current date.



* This class will contain a method for each operation which needs to be done. The methods declared here, will be accessed using the Controller Class, creating an action listener for each button. These methods are used for describing the functionality of each created button. Also, another method declared here, is a getter for the entire menu list. This method is declared, being responsible of a better functionality between the 2 user interfaces: Administrator and Waiter. Each time a new item is added by the administrator, the waiter’s menu list will also add the specific name. The same procedure also stands for the deletion of an existing element.
* **ChefGUI Class (presentationLayer Package):**this class represents the environment in which the Restaurant’s chef prepares all the ordered items. The employee with this role will receive a “pop-up” message each time an order is created. This message will contain some details regarding the orders, such as their id, and where the food should be served (location + table).



* This class will contain as Fields two labels. The first one represents the title for the following area, represented by the second label, in which the order’s pop-ups will appear.



* This class will contain one Constructor: here, all the elements of the graphical user’s interface, like text fields, labels will be added to the corresponding panels and also set to the preferred characteristics like: layouts, background colour, border.
* This class will contain an overridden method with the name: “update()”. This method is used in order to connect the Chef with the Observer interface. The chosen design pattern helps the Chef employee in receiving the orders in real time, as soon as the Waiter computes them.
* **Controller Class (presentationLayer Package):**this class is responsible for the implementation of all the actions performed by the administrator’s buttons and also by the waiter’s buttons. For each button, a certain function belonging to the corresponding interface is called. The methods implemented in the Administrator and Waiter classes are strictly and specifically designed for their usage in the Controller.

This class will contain one Constructor: here, all the methods for adding an action listener are called.

This class will contains several methods, grouped accordingly to their interface and where the case requires, the type of products which use the buttons. The “admin - base product” category will contain three buttons: add, edit and delete.

The “admin - composite product” category will contain five buttons: select ingredients, remove ingredients, add, edit, delete.

The “waiter” category will contain 4 buttons: select menu items, remove menu items, create order and compute bill. The action listener implemented for the “compute bill” button will actually be implemented in the Waiter’s class due to the better usage of the order table.

* **Observer Interface (presentationLayer Package):** the observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioural pattern category. Observer pattern uses three actor classes. Subject, Observer and in our case Chef. Here, an interface *Observer* and a concrete class *Subject* are implemented.

This interface will contain two Fields represented by a subject and the declaration of the update() function. This function will be implemented as mentioned above, in the ChefGUI Class.

Also, the Class Subject will be presented here. Subject is an object having methods to:

Attach observers:



Get the State:



Set the State: the state will actually be changed in the Waiter’s Class in the moment in which a new order is created. The current state will be updated to the message chosen:



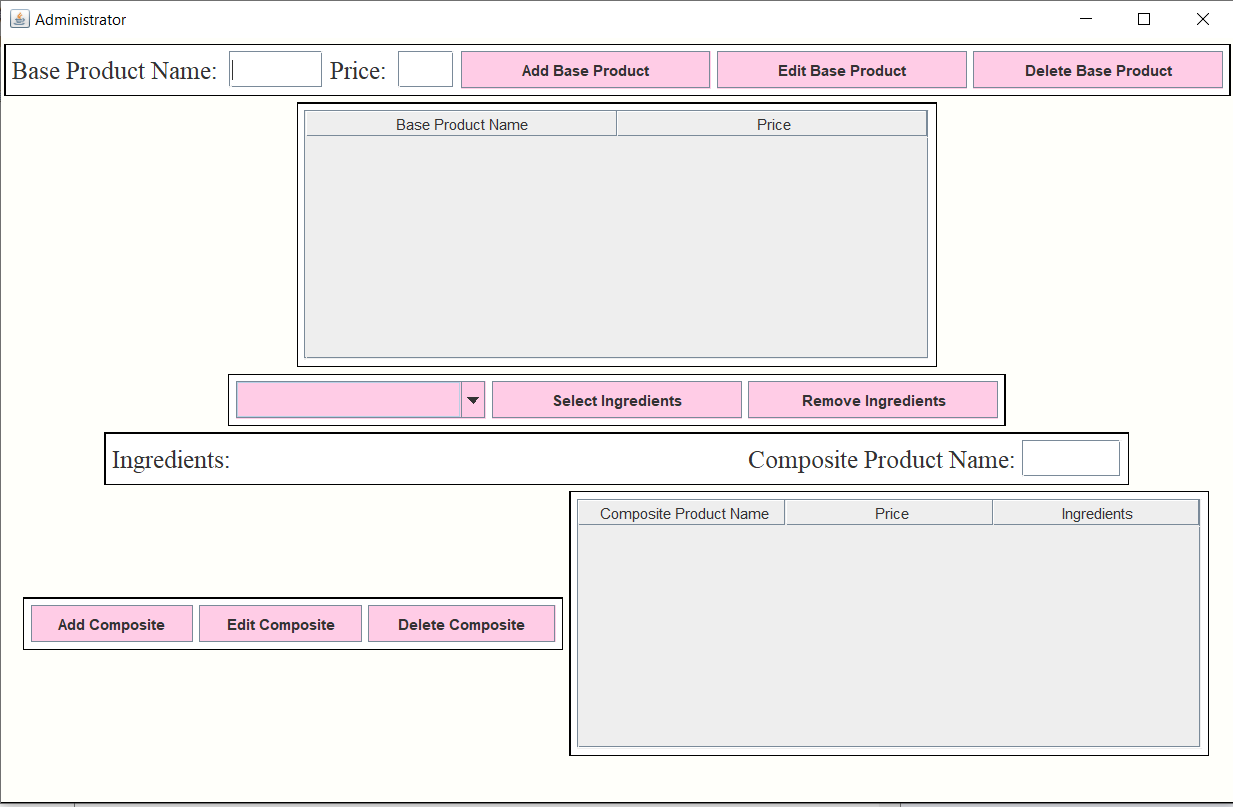
* **Main Class (presentationLayer Package):** it represents the class which will launch the entire application. The interface’s are set to be open in the order: Chef, Waiter and then Administrator. In this way the user can enter the input more easily.

1. **Testing**

Firstly, the application is launched using the already created .jar file, or by typing a command prompt command.

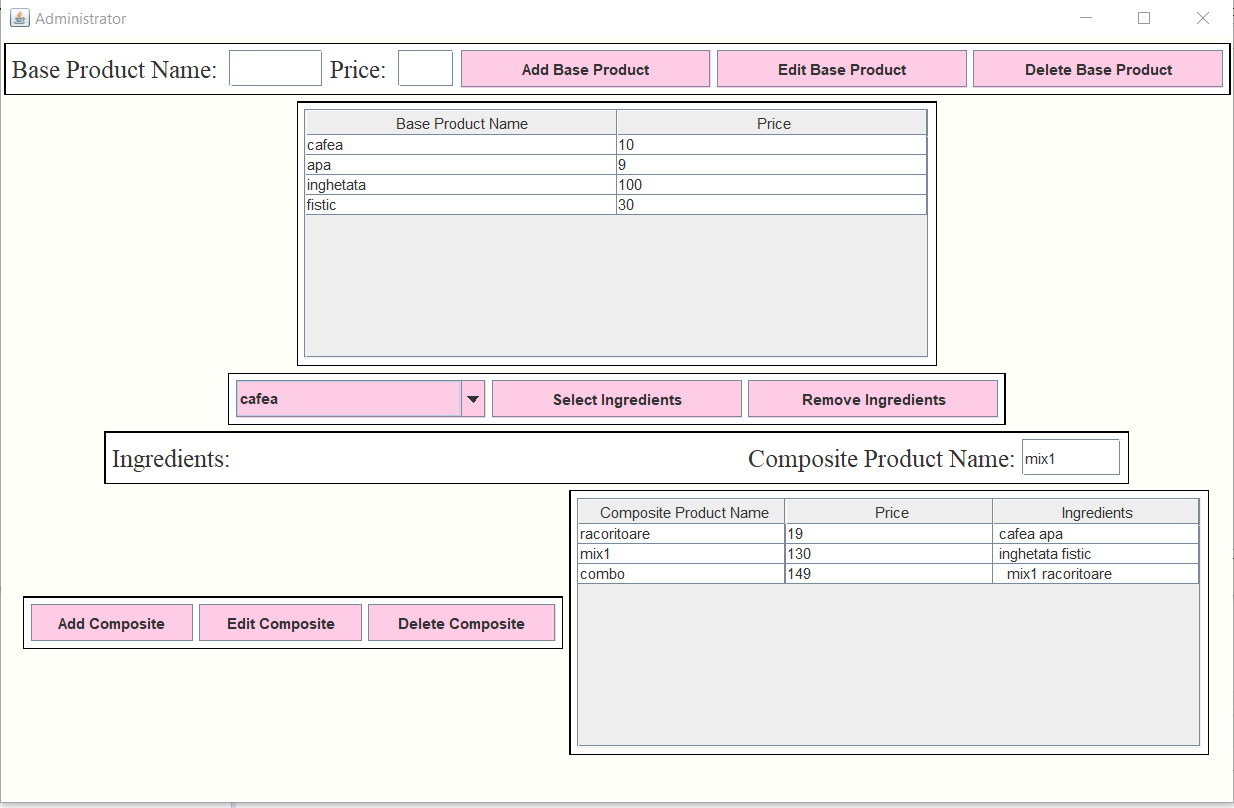


The next step is for the User to take the Administrator Role and create the Restaurant’s menu. Initially, the items list is empty. A more detailed way of completing the administrator’s tasks is described in the Analysis Section at the page 4.

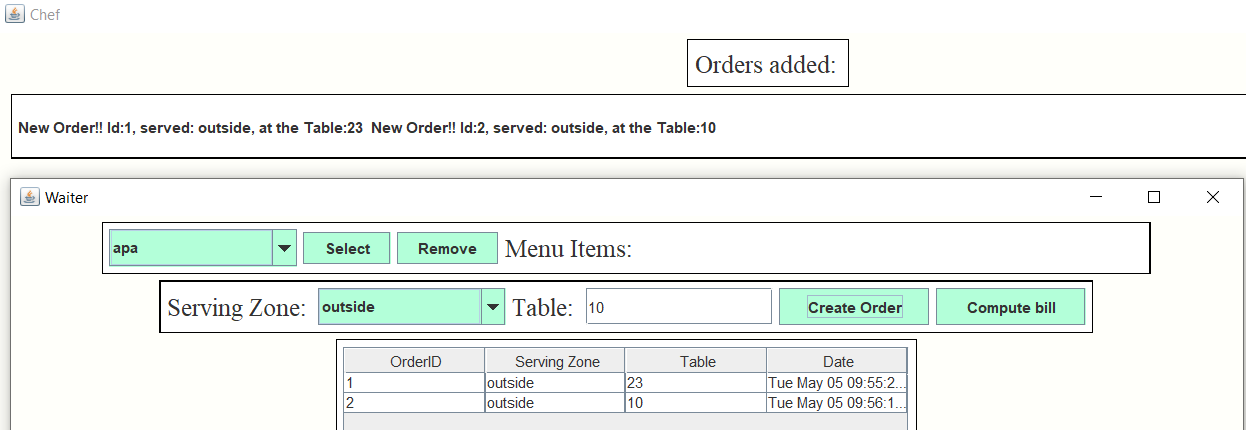
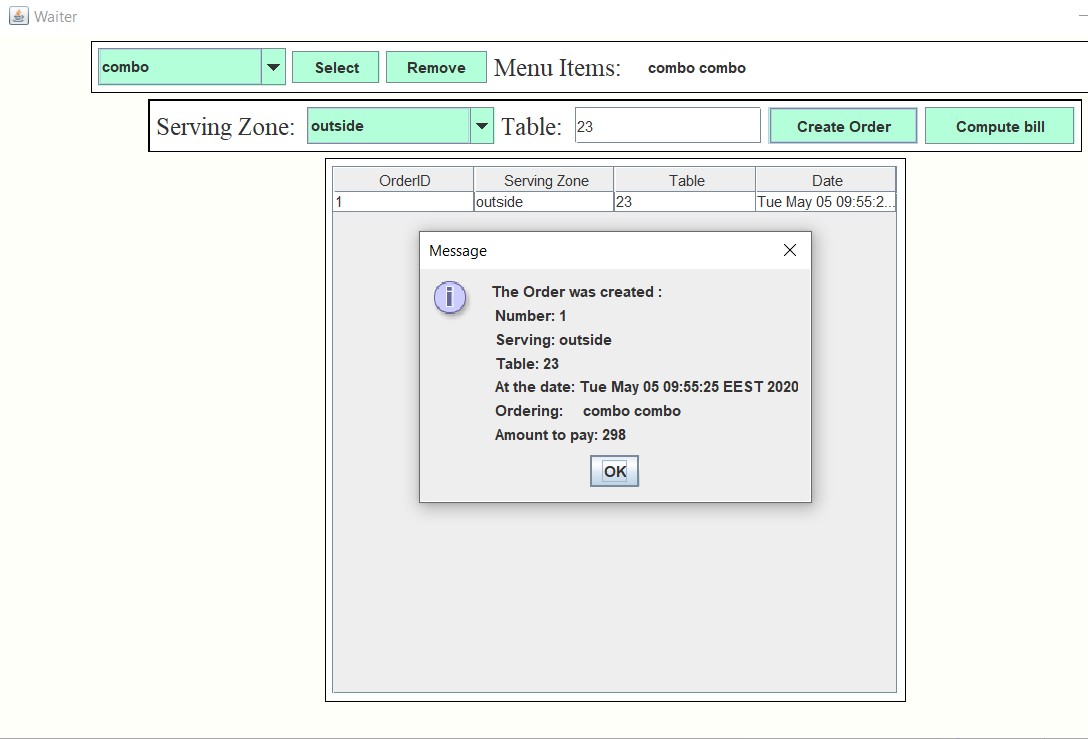
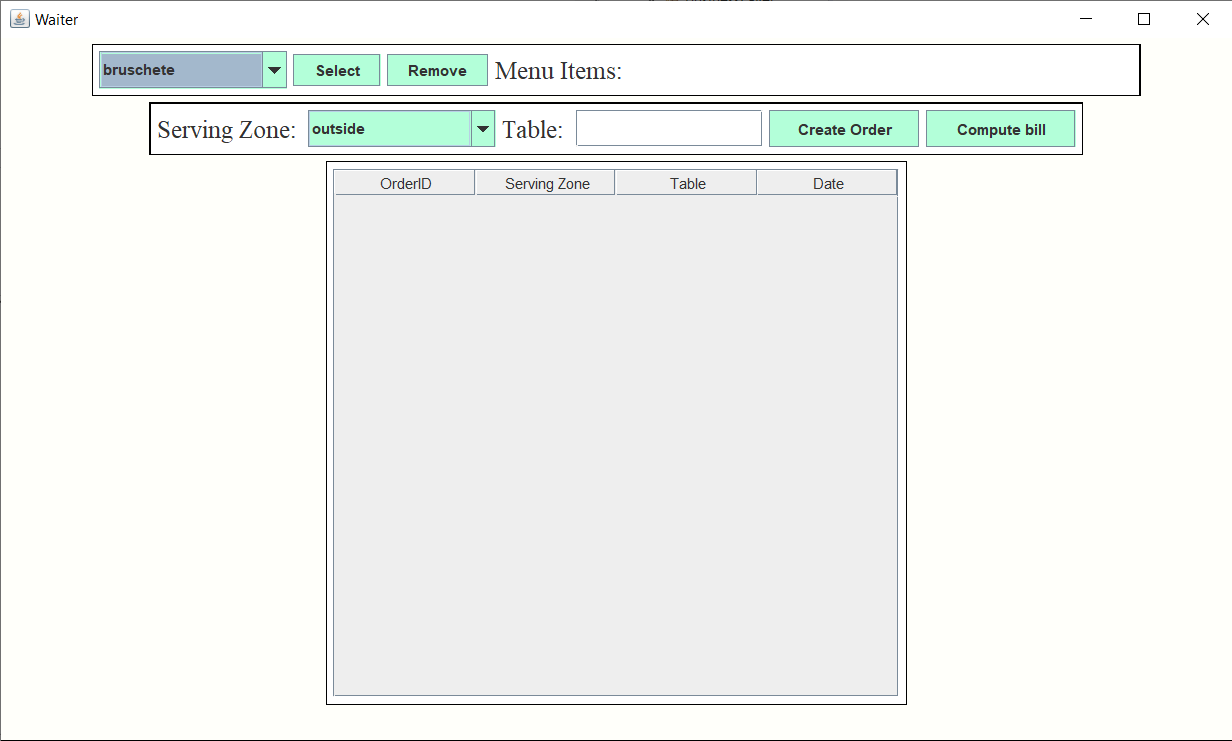


Made only from base products

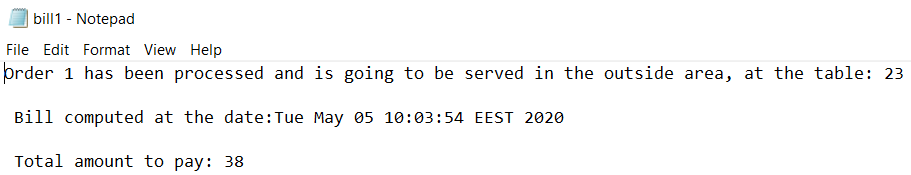
Made only from other composite Products



The menu is chosen. Following, we will check the Waiter’s interface to create the orders. Initially the table is empty. After the orders are created, the user can switch to the Chef’s view and visualise the order’s notifications.



Finally, when some Clients are ready to go, the “Compute Bill" button is pressed and the bill will be generated.



1. **Further Development**

The Restaurant could be extended by presenting more than one administrator. He maybe able to hire more waiters and chefs using a login process. The salary of the waiters can be computed based on the number of orders they complete.

Another improvement may be the active involvement of the Chef in the restaurant’s tasks. He could choose from a list of ingredients, the same items contained in the menu. When an item does not exist anymore, the Chef should be able to transmit a pop-up in order for the stock to be remade.

The restaurant could receive a time interval in which the activities take place. After that simulation time is gone, the restaurant will be closed until the next day.

1. **Conclusions**

This Application was a very good way of finding more about Design Patterns and how they work. It was the first time working with more than one type of user’s interface and the simulation of a Restaurant really brought a better understanding, having to deal with some real-life situations.

It was an opportunity to work again with a Layered Architecture, continuing to believe that this method is more useful and more attractive to a regular User. Each package will serve a different purpose and it is easier to understand the concept and the Logic of the Application.

A Restaurant requires a lot of small but relevant details, I really appreciated the freedom to implement those kind of details whenever needed.

**8.Bibliography**

<https://www.tutorialspoint.com/design_pattern/observer_pattern.htm>

<http://www.blackwasp.co.uk/Composite.aspx>

<https://sourcemaking.com/design_patterns/composite>

<https://www.guru99.com/java-swing-gui.html>

<https://netbeans.org/kb/docs/java/gui-functionality.html>

<https://www.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html>

<https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/>

<https://www.geeksforgeeks.org/arraylist-vs-hashmap-in-java/>