

Restaurant Management System

1. Purpose of the laboratory work	2
2. Analysis, modelling and use-case scenarios	2
3. Development	3
3.1 Package Diagram	3
3.2 Class Diagram	4
4. Implementation	5
4.1 Business Layer	5
4.1.1 MenuItem	5
4.1.2 BaseProduct	5
4.1.3 CompositeProduct	5
4.1.4 Order	5
4.1.5 Restaurant	5
4.2 DataLayer	6
4.2.1 FileWriter	6
4.2.2 RestaurantSerializator	6
4.3 Start	6
4.3.1 Start	6
4.4 Presentation (functionality + UI forms)	6
4.4.1 AdminGUI	6
4.4.2 AddProduct	7
4.4.3 EditProduct	7
4.4.4 DeleteProduct	8
4.4.5 ViewMenu	8
4.4.6 WaiterGUI	9
4.4.7 AddOrder	9
4.4.8 ViewOrComputeBillForOrder	10
4.4.9 ChefGUI	11
5. Conclusions	11
6. Bibliography	11

1. Purpose of the laboratory work

The aim of this laboratory assignment is to design an application for the management of a restaurant. We can interact with the application as an Administrator, a Waiter or a Chef through a GUI.

The Administrator can modify the menu of the restaurant by adding items (base items or composite ones), by editing the price of certain items or by deleting entries from the menu.

The Waiter can create orders for a specific table by adding items from the menu to an order and submitting it, he can view all the orders made and select one to generate a bill for it.

The Chef possesses a window that shows the most recent order and its details.

We designed the Model classes that will provide this functionality. The GUI was designed through Java Swing forms and the Listeners that serve as controllers are implemented in the class bound to its specific form (there is no other way of implementing it), that is why the Controller and the View of this application will be present in the same package: Presentation.

For this application, we were not required to add a login system, so all the windows for the three users open at the same time when running the application. The development decisions for this assignment and the implementation of the required functionality will be presented in the 3rd and 4th chapter of this documentation.

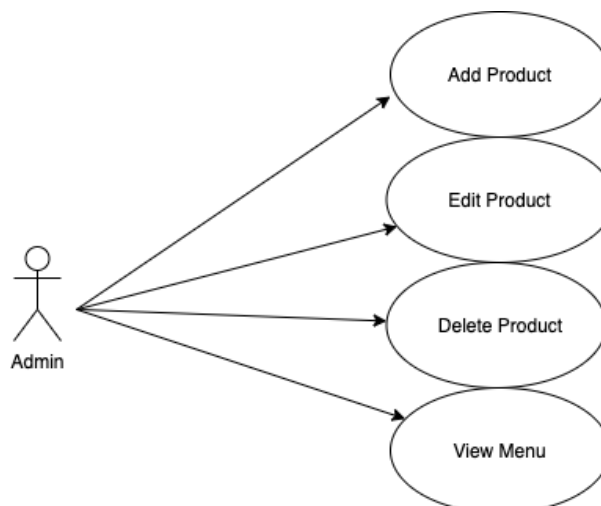
2. Analysis, modelling and use-case scenarios

We are required to have a fully functional program with three types of GUI for the three types of users.

The program can be used from the command line using the following commands from the project's directory: "java -jar PT2020_30424_Todea_Tudor_Assignment_4.jar" if we do not want to load an already existing instance of the Restaurant, or through "java -jar PT2020_30424_Todea_Tudor_Assignment_4.jar restaurant.ser" if we want to load an existing instance present in the "restaurant.ser". After every change in the program we serialize the data, meaning that the state of the Restaurant is always saved. We can choose to load it from the "restaurant.ser" or discard it by leaving the arguments empty.

We are going to present the use-case scenarios, but things will be more detailed in the 4th chapter when we discuss the interface.

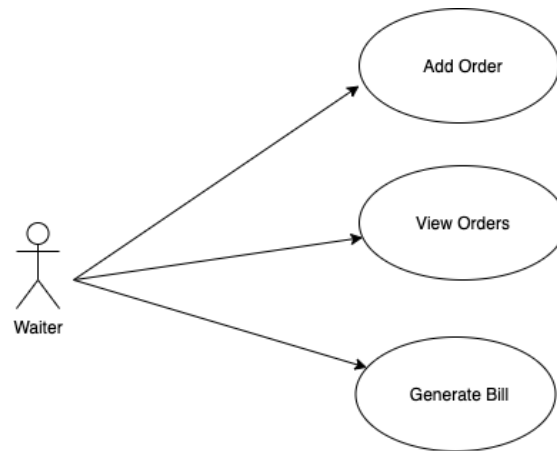
We will start with the use-case diagram of the Administrator:



We have 4 options as an administrator: we either add, edit or delete a product, or we simply want to view the Restaurant's Menu. Problems can occur when adding an item only when we introduce a BaseProduct. The user may introduce numbers instead of an alphanumeric string as the name of the product or letters for the price. If this happens, an error message is displayed telling the user to correct his mistakes and the changes will not take place. The addition of a composite item will not pose a problem since it is done by selecting other products from a table. The only problem may be when introducing an already existing item. In this case, another message will be displayed informing us that the item is already present. The edit and delete operation are done through a table and a button, so there is no place for error

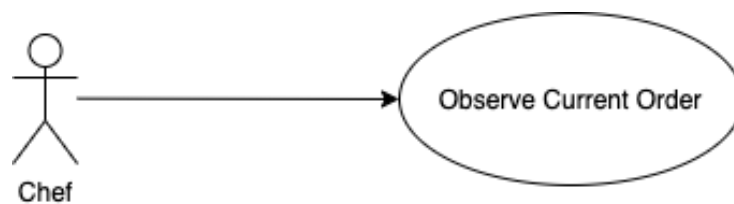
there, only if we accidentally write anything other than numbers when we edit the price of a menu entry. If that is the case, we display an error message.

The diagram for the Waiter:



The Waiter must create orders, generate bills for some selected ones and be able to view the orders. In our project the View Orders and Generate Bill are done in the same window since the Waiter can visualise the orders as a JTable and should he want to generate a bill for one, he simply needs to select it and click the button that generates the bill. Errors may arise if we leave the orders tab or the table tab empty when we add an order, or we do not enter an integer for the table number. All these erroneous cases will be met with proper error messages.

The Chef:

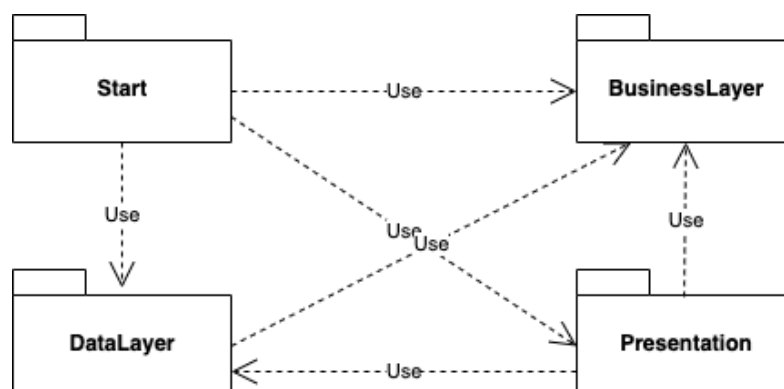


The Chef is updated whenever we place a new order and displays in a JTextField the details of that order.

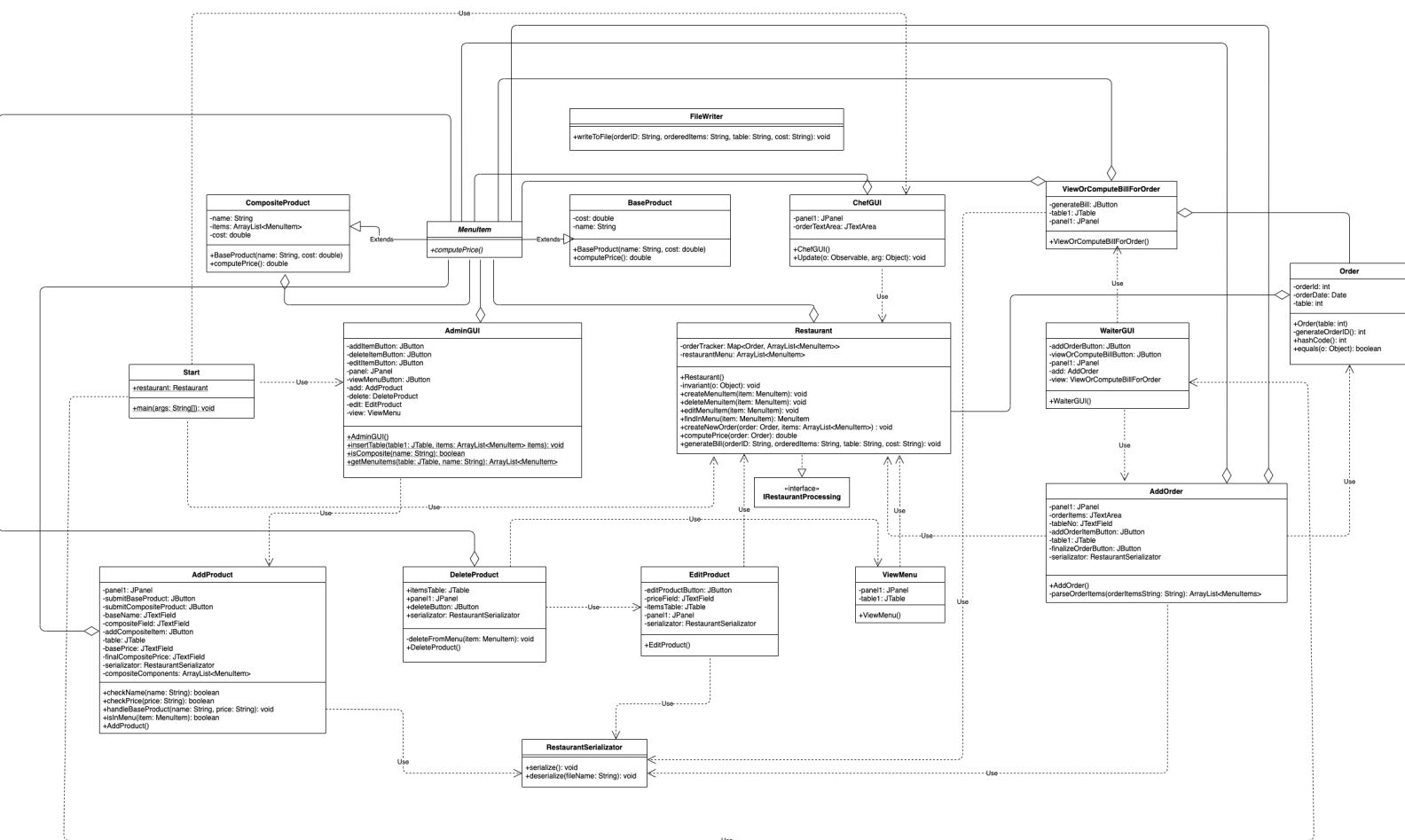
3. Development

3.1 Package Diagram

We have four packages. The DataLayer handles the reading and writing to the serialisation file and the generation of a text file for the bill. The Presentation handles the UI form and controls the Swing events with the help of the BusinessLayer package.



3.2 Class Diagram



During the development we made some key decisions which shape our project. In order to keep everything in one place we created a public and static singleton Restaurant object which we use throughout the entirety of the project. We thus ensure that this object keeps track of the whole menu and the whole history of orders, such that we have everything in the whole place. When we serialize something we create a copy of this static object. Upon deserialization (when we run the program with the serialized file as an argument) we store all the information into this singleton object.

The BaseProduct and CompositeProduct classes extend the MenuItem abstract class. They both have their independent name and cost variable since we can make a clearer separation between them when handling the items in the interface. This leads to more typecasting when writing the code, but it makes it easier to separate them.

The UI, as mentioned in the preceding chapters, is designed using GUI forms. As a result, the event listeners had to be placed in the bound class of the form. Consequently, the forms play the role of the View and the bound class plays the role of a Controller.

4. Implementation

4.1 Business Layer

4.1.1 MenuItem

Is an abstract class that will be the base of our Composite approach. Contains only one abstract method called **computePrice()**.

4.1.2 BaseProduct

This class extends the MenuItem. It is characterised by its name and its cost.

4.1.3 CompositeProduct

Also extends MenuItem. Consists of its name, its cost - which is calculated in a top down manner for each component of the composite - and an ArrayList of MenuItem. Thus, a composite product can be composed of multiple BaseProduct objects, or a combination of CompositeProduct and BaseProduct objects. There is no limit for the composition levels that can be achieved in our application.

4.1.4 Order

Each order has a characteristic *table*, *orderDate* and *orderId*. The ID of the order is the hour and the minute at which the order was placed. For instance, an order placed at 22:03 will have the ID 2203. This makes it very easy to keep track of when the order was made and to identify a specific order using the time and the table. This would be a very good approach in a real life adaptation of a restaurant. This class overrides the **hashCode()** method since we need to compute our own hash for the HashMap that will be present in the Restaurant. The **hashCode()** is computed by the following formula: $3 * orderId + 2 * table + orderDate.getMonth()$.

4.1.5 Restaurant

The restaurant has two class variables: the *orderTracker* which is implemented by a HashMap, having the Order as a key, and the ordered items as the values, and the *restaurantMenu* which is an ArrayList of MenuItems. We could have used a HashSet instead, to make sure we don't have duplicates, but since we already do this when we receive the input in order to warn the user that there was a problem, we don't need to worry about duplicates since this would be impossible.

We also implemented some notions using the Design by Contract pattern. We were required to pick an invariant for the class. There were multiple options we had to consider. What could be an invariant of such a class? Should all the items of the menu have only letters in their name and a double number as their price? Well, yes, but we already make sure this happens when we add new products. So we chose to consider that all the entries in the menu and all the orders will be non-null objects. Since we check all other constraints, it is reasonable to assume that no Restaurant would ever want or need a null entry in their menu or order tracker, so picking this as an invariant was the only option we had.

Inside the Restaurant we have multiple methods that model our application:

createMenuItem(item: MenuItem):

Adds a new item to the menu. We use design by contract to assert that the element is not null as a precondition and that the actual size of the menu increases by 1 as a postcondition. The non-nullity of the objects will also be checked by the invariant of the class.

deleteMenuItem(item: MenuItem):

We delete an item from the menu. We then make sure the size of the menu has decreased using a postcondition.

editMenuItem(item: MenuItem):

We edit the price of an item in the menu.

findInMenu(item: MenuItem):

Returns the entry provided as an argument if the item is found. Returns null otherwise.

createNewOrder(order: Order, items: ArrayList<MenuItem>):

Adds a new order to our *orderTracker* and notifies the Chef that a new order has been placed.

computePrice(order: Order):
Computes the price of an order.

generateBill():
Calls the writeToFile method to generate a bill in text format.

4.2 DataLayer

4.2.1 FileWriter

writeToFile():
Prints the order information to a text file with the name “OrderBill” followed by the orderID and the table from which the order was made.

4.2.2 RestaurantSerializator

serialize():
Creates a new file called “restaurant.ser” if it does not exist and saves all the information of the Restaurant object into that file.

deserialize(fileName: string):
Gets a file name as the input and deserialises the information into the singleton restaurant variable.

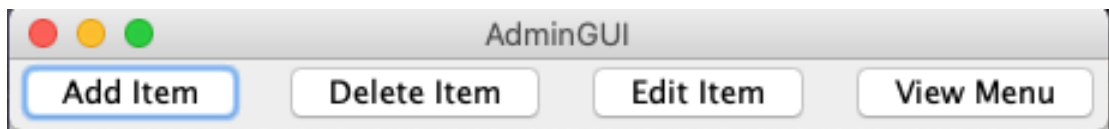
4.3 Start

4.3.1 Start

Contains the main function of the program. Initialises the 3 GUI forms and deserialises the Restaurant object if needed.

4.4 Presentation (functionality + UI forms)

4.4.1 AdminGUI



Initialises the AdminGUI form which contains the buttons for adding, editing and deleting a product as well as viewing the menu of the restaurant. This class also contains three static methods:

insertTable(table1: JTable, items: ArrayList<MenuItem>):
Creates a table with two columns: “Product name” and “Price” containing all the entries from a list of items.

isComposite(name: String):
Takes a string which represents the name of a restaurant menu entry and determines if it is a composite entry.

getMenuItems(table: JTable, name: String):
Takes an entry from the table and returns the items contained in that entry.

4.4.2 AddProduct

Product Name	Price
Pizza	4.0
Pizza + Pasta	11.0
Coke	2.0
Ciorba de burta	4.0
Fanta	2.0
Snitel	3.0
Cartofi prajiti	2.0

The methods **checkName**, **checkPrice** and **handleBaseProduct** checks if the addition of a base product is valid (the name is formed only of letters and spaces and the price is an integer or a floating point number) and if it is not already present in the table, it will be inserted and the Restaurant object serialized.

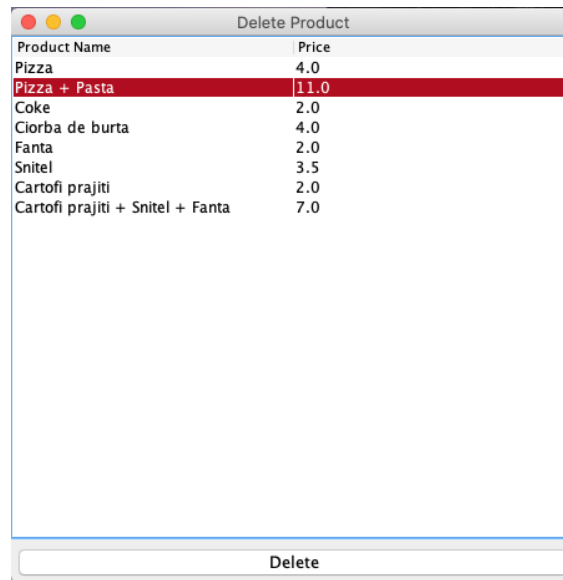
To add a composite product, we select an entry from the table and click on the “Add Composite Item”. The entry will be displayed on the JTextField “Final Composite Product”. We can do this with as many entries as we want. When we are done creating this composite product, we press “Submit Composite Product” and if it doesn’t already exist, the composite product will be added to the menu.

4.4.3 EditProduct

Product Name	Price
Pizza	4.0
Pizza + Pasta	11.0
Coke	2.0
Ciorba de burta	4.0
Fanta	2.0
Snitel	3.0
Cartofi prajiti	2.0
Cartofi prajiti + Snitel + Fanta	7.0

We click on an entry in the JTable to highlight it. We enter the new price and click “Edit Product”. If the new price entered is valid, the price of the item will be modified and the state of the Restaurant object will be serialized.

4.4.4 DeleteProduct

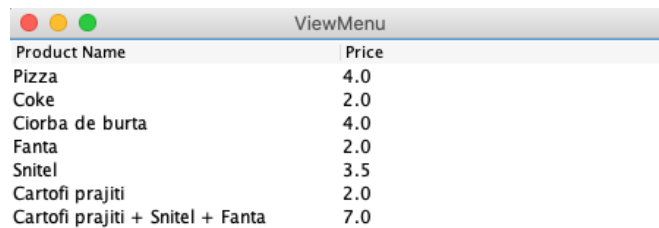
A screenshot of a Java Swing window titled "Delete Product". It contains a table with two columns: "Product Name" and "Price". The table lists several items, with "Pizza + Pasta" highlighted in red. Below the table is a "Delete" button.

Product Name	Price
Pizza	4.0
Pizza + Pasta	11.0
Coke	2.0
Ciorba de burta	4.0
Fanta	2.0
Snitel	3.5
Cartofi prajiti	2.0
Cartofi prajiti + Snitel + Fanta	7.0

Delete

We select an entry from the menu and highlight it, then we click delete to remove that entry from the table. The state of the Restaurant is then serialized.

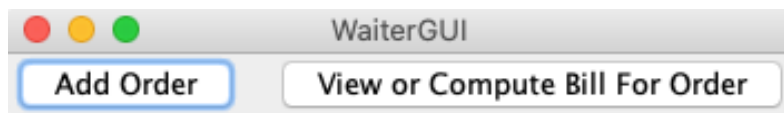
4.4.5 ViewMenu

A screenshot of a Java Swing window titled "ViewMenu". It contains a table with two columns: "Product Name" and "Price". The table lists the same items as the "Delete Product" window.

Product Name	Price
Pizza	4.0
Coke	2.0
Ciorba de burta	4.0
Fanta	2.0
Snitel	3.5
Cartofi prajiti	2.0
Cartofi prajiti + Snitel + Fanta	7.0

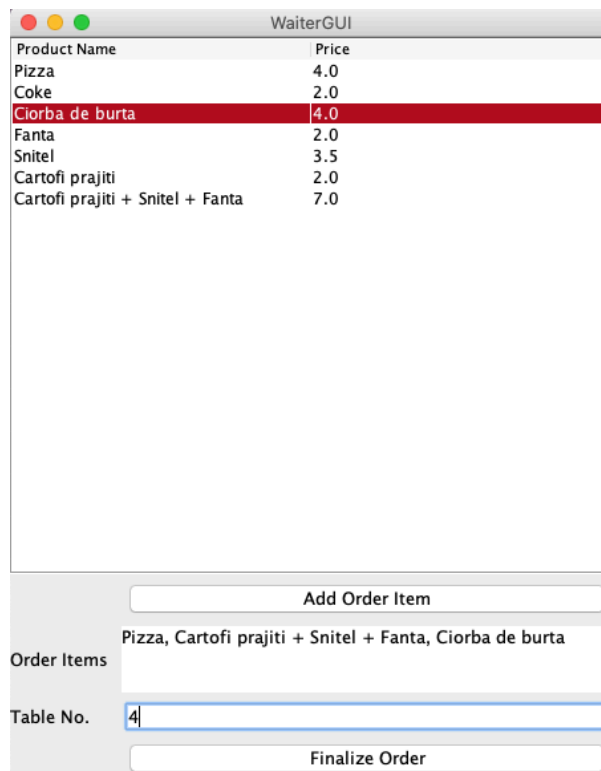
This option simply prints the menu of the restaurant in a JTable. Keep in mind that all the JTables in this project are build on a JScrollPane, so we can add as many items as we wish.

4.4.6 WaiterGUI



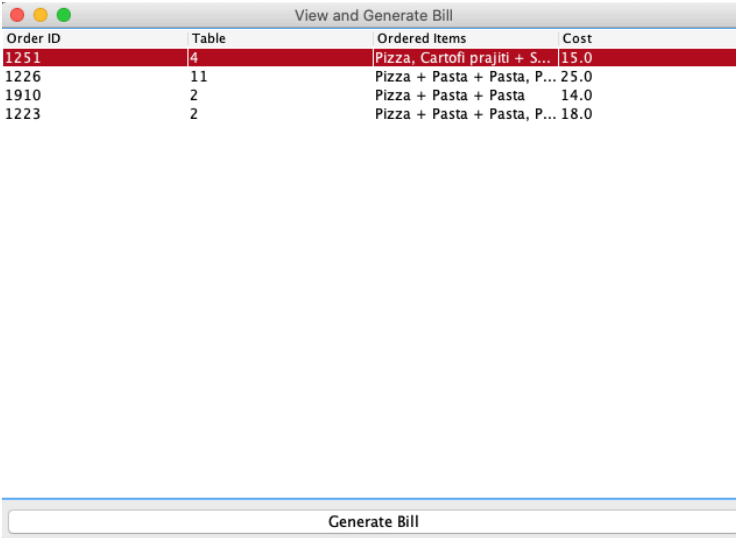
We can select one of the two options: “Add Order” or “View or Compute Bill For Order”. The first one opens a window which allows us to add an order and the latter prints the list of orders and allows us to compute the bill for any of them.

4.4.7 AddOrder



Inside this class we have the method **parseOrderItems()** which takes the string from the “Order Items” field and splits it into items. We add items to our order by selecting (highlighting them) and pressing the “Add Order Item” button. The item will be appended in the “Order Items” JTextField. Each item in our order is separated by a “,”. When we are done adding the items to our order, we just type in the “Table No.” and press “Finalize Order”. The order will be parsed and added to the list of orders in the Restaurant singleton object.

4.4.8 ViewOrComputeBillForOrder



The screenshot shows a window titled "View and Generate Bill". Inside, there is a table with four columns: "Order ID", "Table", "Ordered Items", and "Cost". The first row is highlighted in red. Below the table is a button labeled "Generate Bill".

Order ID	Table	Ordered Items	Cost
1251	4	Pizza, Cartofi prajiti + S...	15.0
1226	11	Pizza + Pasta + Pasta, P...	25.0
1910	2	Pizza + Pasta + Pasta	14.0
1223	2	Pizza + Pasta + Pasta, P...	18.0

We just highlight the order for which we want to compute the bill (in the above picture we will compute the bill for the order we just placed in the AddOrder example) and press “Generate Bill”.

```
OrderID: 1251
Table: 4
Ordered Items:
Pizza
Cartofi prajiti + Snitel + Fanta
Ciorba de burta
Cost: 15.0
```

We will obtain the result in a text file in the folder of our project. In our case the text file will be name “OrderBill1251T4.txt” since the bill is for the order with ID 1251 and the number of the table is 4.

If we just wish to just view the orders, we can just do so without pressing the “Generate Bill” button.

4.4.9 ChefGUI

```
The chef is now preparing the following order:  
Order with ID: 1251  
Items:  
Pizza  
Cartofi prajiti + Snitel + Fanta  
Ciorba de burta  
For table: 4|
```

The Chef (the Observer) gets information from the waiter about the latest order he has to prepare. He is updated by the Observable (which is the Restaurant) whenever a new order is placed and executes the printing of the latest order in the **update()** method inside the ChefGUI class. In conclusion, as we can see above, the Chef is displaying the information for the order he is currently taking care of, which is the order we placed in the AddOrder example. If we place a new order, this information will instantly change and the ChefGUI will display the details of the new order.

5. Conclusions

During the development of this application we have learned how to use the Observable design pattern and the Design by Contract method. We used assertions to assure the integrity of our application, and the Observable pattern to update a part of the code based on changes in another part. We learned how to use the Composite design pattern and how to achieve unlimited levels of composition. The application can be a usable and functional tool used in a real restaurant. We may develop it further and use a database system to store the information, a login system to differentiate between the users, maybe use a timer to approximate the duration of preparing an order. Perhaps we may add a finite number of tables to represent a real place and extend the graphical interface to show each table that is busy and the order that is bound to that table.

6. Bibliography

http://www.coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_4/Assignment_4_Indications.pdf

<https://www.youtube.com/watch?v=2HUnoKyC9l0>

<https://objectcomputing.com/resources/publications/sett/september-2011-design-by-contract-in-java-with-google>

<https://www.geeksforgeeks.org/java-util-observable-class-java/>

<https://www.youtube.com/watch?v=22MBsRYuM4Q>