TECHNICAL UNIVERSITY OF CLUJ-NAPOCA ROMANIA

ASSIGNMENT 4
Restaurant management

Student: Mitrica Livia-Maria

Group: 30424

Techer Assistant: Chifu Viorica

Contents

Objective

The main purpose of this application is to implement a restaurant management system. The system should have three types of users: administrator, waiter and chef.

The administrator can do the following operations on the products from the menu:

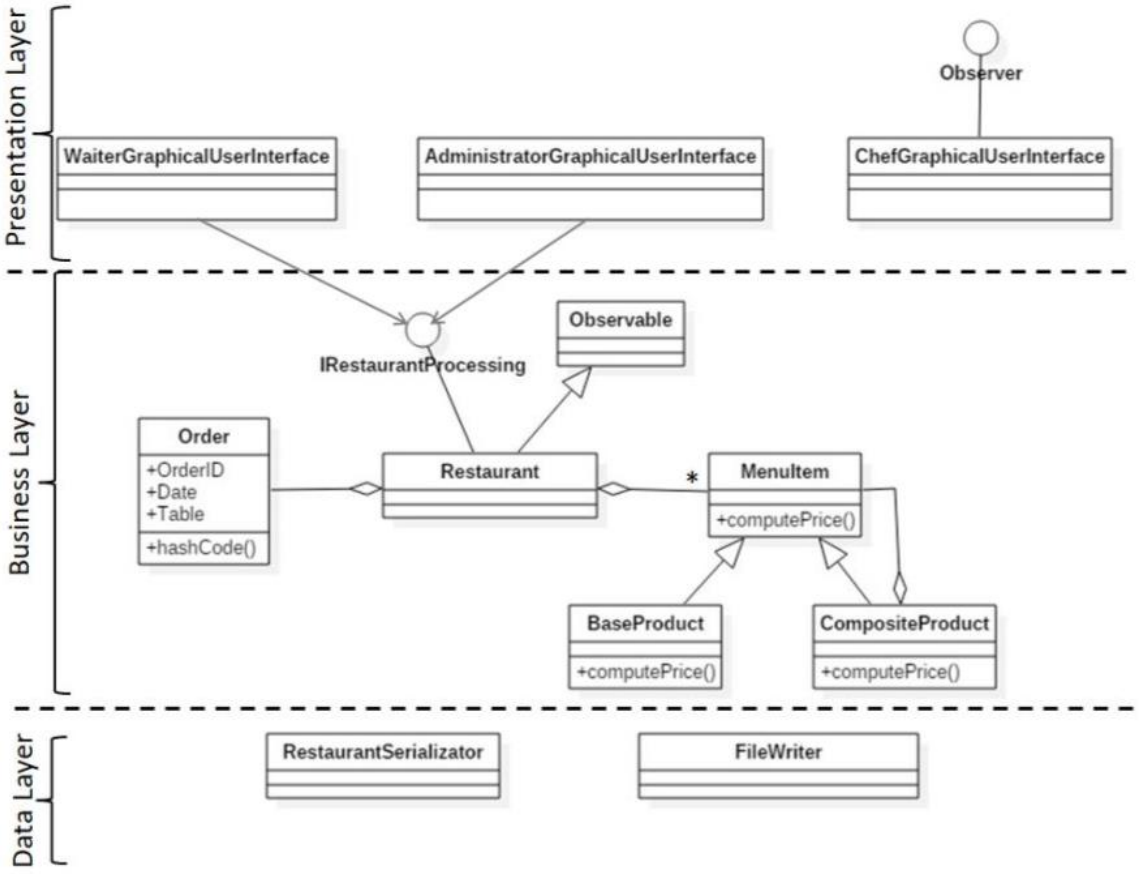
- add
- delete
- modify

The waiter can do the following operations:

- create a new order for a table
- add elements from the menu
- compute the bill for an order

The chef is notified each time it must cook food that is ordered through a waiter.

The application will be implemented the system of classes in the diagram below.

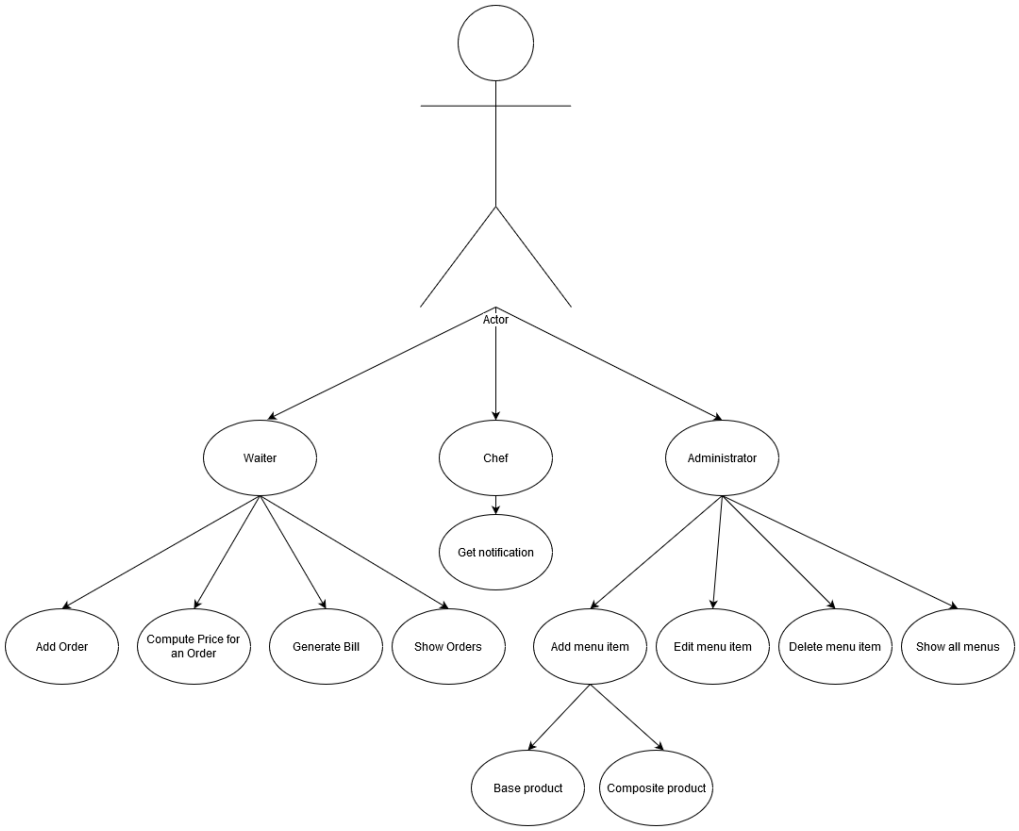


Problem analysis

The application should behave like a restaurant with 3 main users, namely an Administrator who can create a new menu item, delete a menu item, edit a menu item, a Waiter who can create a new order for a table, add elements from the menu, compute the total price for an order and generate the bill for an order, a Chef who will be notified each time a composite product is added to an order.

Use case

The user can choose between 2 situations, Administrator or Waiter.



Use case scenarios Administrator

- Insert a new base item
  If the user inserts a valid name and a valid price, a dialog message notifying that the base product has been added successfully is shown after pressing add item button, otherwise, an error message is shown.

- Insert a new composite item
  If the user inserts a valid name and selects minimum 2 base items, a dialog message notifying that the composite product has been added successfully is shown after pressing add item button, otherwise, an error message is shown.

- Edit an item
  If the user inserts a valid product name, a new valid name and/or price to replace the ones for the old product, a dialog message notifying that the product has been updated successfully is shown after pressing edit item button, otherwise, an error message is shown.

- Delete an item
  If the user inserts a valid name, a dialog message notifying that the product has been deleted successfully is shown after pressing add item button, otherwise, an error message is shown.

- Show all menu items
  If the user presses this button all menu items will be displayed in a table. After doing create?delete?edit operations, refreshing the data, repressing the button is needed.

Use case scenarios Waiter

- Create new order
  If the waiter introduces a valid table number and chooses at least one item from the menu to add to the order, a dialog message notifying that the order has been created successfully is shown after pressing add order button, otherwise, an error message is shown

- Compute price for an order
  If the waiter introduces a correct table number and presses compute total, the total price of the order will be diplayed.

- Generate bill
  If the waiter introduces a correct table number and presses generate bill, a .txt file is generated containing the order id, the date, the list of items which have been ordered and the total price.

Use case scenario Chef

- The waiter places an order containing a composite product and the chef is notified. The notifiaction also contains details about what the menu items have been orderd and when the order has been placed.

Problem design

The implementation of this application follows the pattern model-view-controller (MVC).

Therefore, I have decided to split my code into 4 packages and apart from model, view and controller packages as expected, add a new one which is called application.

I will summarize the packages and the classes they contain here and I will detail each class in the implementation section:

1) Model-represents the underlying, logical structures of data in the application, does not contain any information about the user interface
   a) BaseProduct
   b) CompositeProduct
   c) IDgenerator
   d) DateOrder
   e) MenuItem
   f) Order
   g) Restaurant

2) View-conatins a collection of classes representing the graphical elements in the user interface (panels, buttons, text fields etc.)
   a) UserInterface (for Waiter and Administrator)
   b) ChefGUI

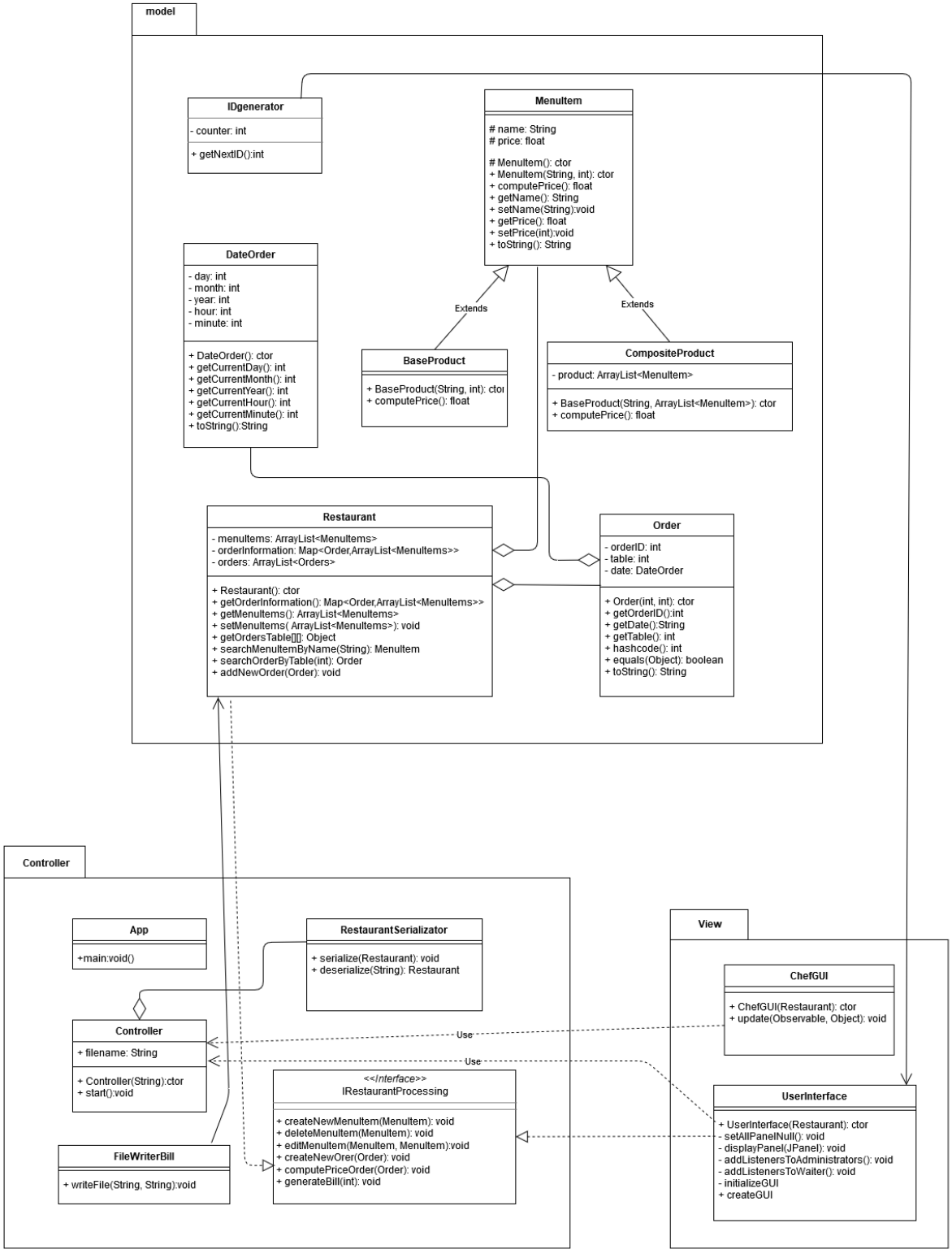
3) Controller-represent the classes connecting the model and the view, basically it is used to communicate between the classes in model and view
   a) Controller
   b) FileWriterBill
   c) Restaurant Serializator
   d) IrestaurantProcessing

4) Application-designed for starting the application
   a) App

Data structures:

Apart from primitive types like int, float or bolean, I mainly used Strings and Lists(ArrayLists) designed for storing the orders, the base products in a composite product.

The following picture illustrates the structure of my project on packages.


model
IDgenerator
- counter: int
+ getNextID():int
MenuItem
name: String
price: float
MenuItem(): ctor
+ MenuItem(String, int): ctor
+ computePrice(): float
+ getName(): String
+ setName(String):void
+ getPrice(): float
+ setPrice(int):void
+ toString(): String
DateOrder
- day: int
- month: int
- year: int
- hour: int
- minute: int
+ DateOrder(): ctor
+ getCurrentDay(): int
+ getCurrentMonth(): int
+ getCurrentYear(): int
+ getCurrentHour(): int
+ getCurrentMinute(): int
+ toString():String
Extends
Extends
BaseProduct
+ BaseProduct(String, int): ctor
+ computePrice(): float
CompositeProduct
- product: ArrayList<MenuItem>
+ BaseProduct(String, ArrayList<MenuItem>): ctor
+ computePrice(): float
Restaurant
- menuItems: ArrayList<MenuItems>
- orderInformation: Map<Order,ArrayList<MenuItems>>
- orders: ArrayList<Orders>
+ Restaurant(): ctor
+ getOrderInformation(): Map<Order,ArrayList<MenuItems>>
+ getMenuItems(): ArrayList<MenuItems>
+ setMenuItems(ArrayList<MenuItems>): void
+ getOrdersTable[][]: Object
+ searchMenuItemByName(String): MenuItem
+ searchOrderByTable(int): Order
+ addNewOrder(Order): void
Order
- orderID: int
- table: int
- date: DateOrder
+ Order(int, int): ctor
+ getOrderID():int
+ getDate():String
+ getTable(): int
+ hashcode(): int
+ equals(Object): boolean
+ toString(): String
Controller
App
+main:void()
RestaurantSerializator
+ serialize(Restaurant): void
+ deserialize(String): Restaurant
Controller
+ filename: String
+ Controller(String):ctor
+ start():void
Use
Use
<<Interface>>
IRestaurantProcessing
+ createNewMenuItem(MenuItem): void
+ deleteMenuItem(MenuItem): void
+ editMenuItem(MenuItem, MenuItem):void
+ createNewOrer(Order): void
+ computePriceOrder(Order): void
+ generateBill(int): void
FileWriterBill
+ writeFile(String, String):void
View
ChefGUI
+ ChefGUI(Restaurant): ctor
+ update(Observable, Object): void
UserInterface
+ UserInterface(Restaurant): ctor
- setAllPanelNull(): void
- displayPanel(JPanel): void
- addListenersToAdministrators(): void
- addListenersToWaiter(): void
- initializeGUI
+ createGUI

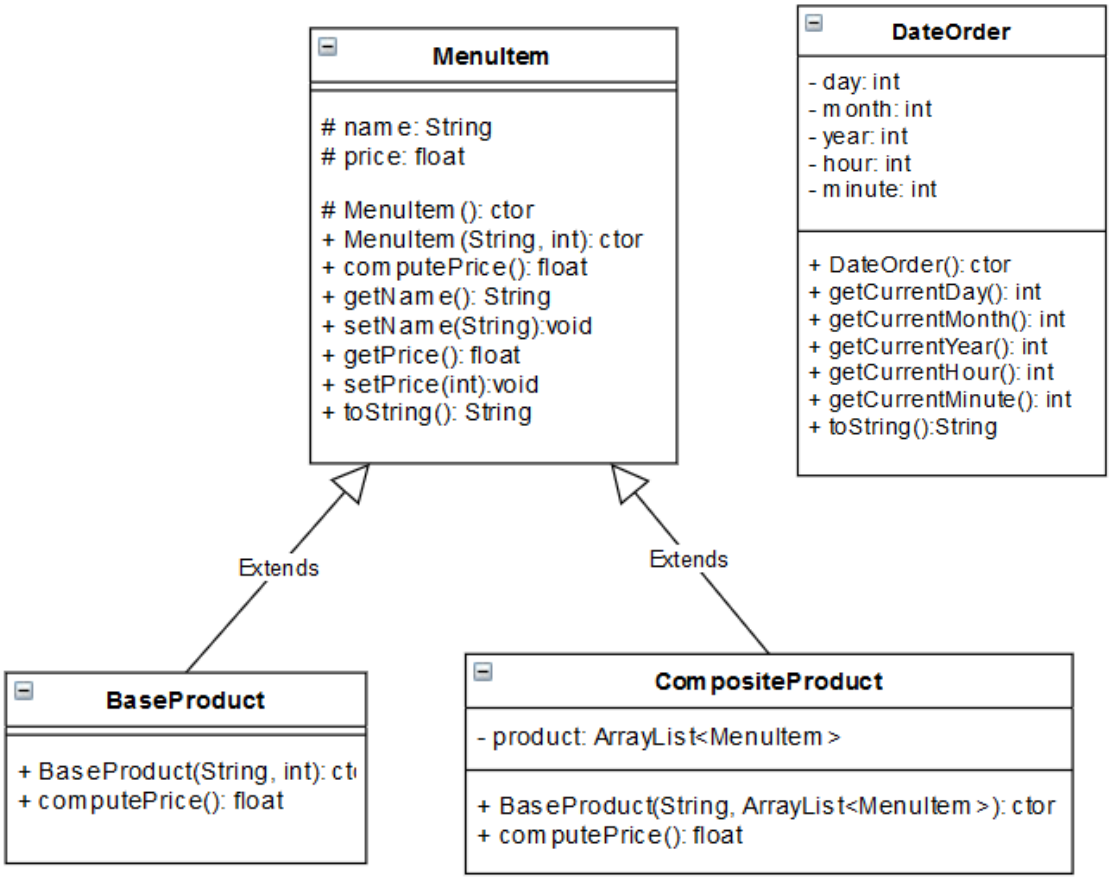
Implementation

In this section I will detail each class on packages, based on packages:

1) Model
    a. BaseProduct - class which models a basic product in the MenuItem class
    b. CompositeProduct - class which models a composite product in the MenuItem class, it is basically created froom other basic or composite products, its price being calculated as a sum of the items it is made of.

```
@Override
public float computePrice() {
   float price = 0;
   for (MenuItem p: product) {
      price+=p.getPrice();
   }
   this.setPrice(price);
   return price;
}
```

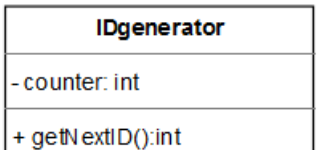
    c. MenuItem
    Abstarct class which resembles a menu item from a restaurant. As it can be seen from the picture, both classes BaseProduct and CompositeProduct extend the class MenuItem and implement the abstract method computePrice. The parent class has only basic methods like getters, setters and toString.
    d. DateOrder
    This class is designed to simplfy the class Date from Java. It gives real time (day, month, year, hour, minute) using Calendar. It is used for registering correctly the moment an order is palced.



```
public DateOrder(){
    Calendar cal= Calendar
                                .getInstance(TimeZone
                                .getTimeZone("Europe/Bucharest"));
    Date date = new Date();
    cal.setTime(date);
    this.day = cal.get(Calendar.DAY_OF_MONTH);
    this.month = cal.get(Calendar.MONTH)+1;
    this.year = cal.get(Calendar.YEAR);
    this.hour = cal.get(Calendar.HOUR_OF_DAY);
    this.minute = cal.get(Calendar.MINUTE);
}
```

e. IDgenerator

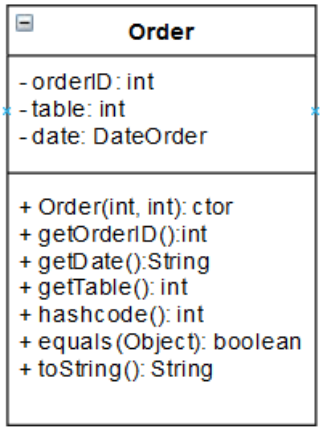
This class is a simple class for generating an id for the current order that is inserted.



f. Order

The most important class for a waiter, because it represents the waiter's job. It contains the table id, the number of the table and the date the order was placed.

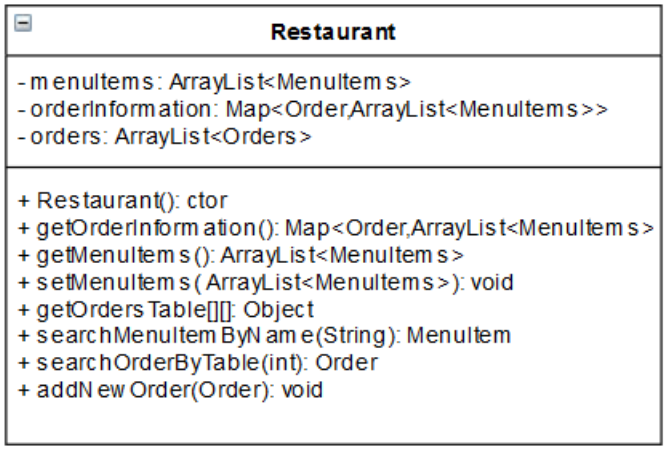
Apart from simple methods like getters, setters, it also overrides the methods equals(), hashcode() and toString().




```
@Override
public int hashCode() {
    int hashcode = 13;
    hashcode += hashcode*11 + 7*orderID+31*date.getCurrentDay()+5*table;
    return hashcode;
}
```

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Order order = (Order) o;
    return orderID == order.orderID &&
            table == order.table &&
            date.equals(order.date);
}
```

g. Restaurant
This is the „heart of the application”, because this is the place where all actions go through. It implements the interface IrestaurantProcessing and all the methods that it contains. It is build using Design By Contract, testing the pre- and postcondtions with assert in the method it implements. In order to send notifications to the chef extends Observable. (The change is set in the method addNewOrder).



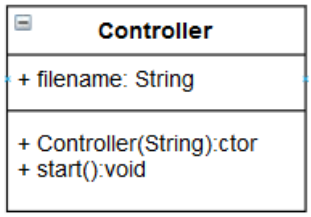

```
@Override
public void createNewMenuItem(MenuItem menuItem) {
    assert menuItem!=null;
    int preSize = menuItems.size();
    menuItems.add(menuItem);
    int postSize = menuItems.size();
    assert preSize == postSize-1;
}

@Override
public void deleteMenuItem(MenuItem menuItem) {
    assert menuItem!=null;
    int preSize = menuItems.size();
    menuItems.remove(menuItem);
    int postSize = menuItems.size();
    assert preSize == postSize+1;
}

@Override
public void editMenuItem(MenuItem oldMenuItem, MenuItem newMenuItem) {
    assert !oldMenuItem.equals(newMenuItem);
    for (MenuItem m: menuItems)
        if(m.equals(oldMenuItem)) {
            int index = menuItems.indexOf(m);
            menuItems.set(index, newMenuItem);
        }
}

@Override
public void createNewOrder(Order order, ArrayList<MenuItem> menuItems) {
    assert order!=null;
    assert menuItems!=null;

    Order currentOrder = order;

    orderInformation.put(order,menuItems);

    assert currentOrder.equals(order);
}

@Override
public float computePriceOrder(Order order) {
```

```
    assert order!=null;
    float priceOrder =0;

    if(orders.contains(order)){
        ArrayList<MenuItem> currentOrderMenuItems = orderInformation.get(order);
        for (MenuItem m: currentOrderMenuItems){
            priceOrder+=m.computePrice();
        }
    }

    return priceOrder;
}

@Override
public void generateBill(int tableNo) {
    assert tableNo>0;
    Order currentOrder = searchOrderByTable(tableNo);
    String s = new String();
    s+="Order #"+currentOrder.getOrderID()+"\n";
    s+="Date: "+currentOrder.getDate().toString()+"\n\nItems:\n";

    ArrayList<MenuItem> currentOrderMenuItems = orderInformation.get(currentOrder);
    for (MenuItem m: currentOrderMenuItems) {
        s+=m.toString()+"\n";
    }
    float price =computePriceOrder(currentOrder);
    s+="\nTotal: "+price;
    FileWriterBill fileWriterBill = new FileWriterBill();
    String filename = "bill"+currentOrder.getOrderID();
    fileWriterBill.writeFile(s, filename);
}
```

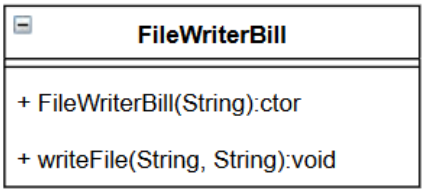
2) View
   a. UserInterface (for Waiter and Administrator) – represents the graphical interface for both waiter and administrator. It is instantiated in the controller class.

   b. ChefGUI – represents the graphical interface for the chef. Each time an order containg a composite product is made, the chef receives a message containg the corresponding details. It also implements Observer interface in order to get notifications when composite products need to be prepared.

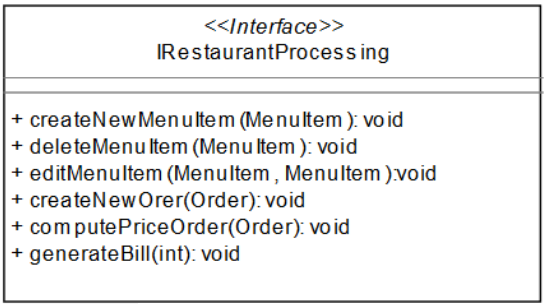
3) Controller
   a. Controller – initializes the graphical interfaces and sets up the environnment for the application
   Apart from the contructor, which takes the file containing the restaurant serialization, it also has a method in which we start the graphical user interfaces and deserialize the restaurant information.



   b. FileWriterBill – class designed for writing in a file.
   Apart from the contructor, it has a method which receive a string representing the name of the file and one containg the text to be written in the txt file.



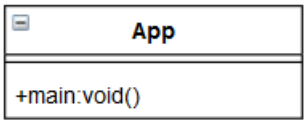
c. IRestaurantProcessing – interface that yields the main methods implemented by the Restaurant and the UserInterface.



d. Restaurant Serializator – it is responsible with serializing and deserializing the restaurant, namely the menus, when the application starts. The serialization of the restaurant is performed when operations with the menu items are perfomerd, respectively, creating, editing and deleting menu items.

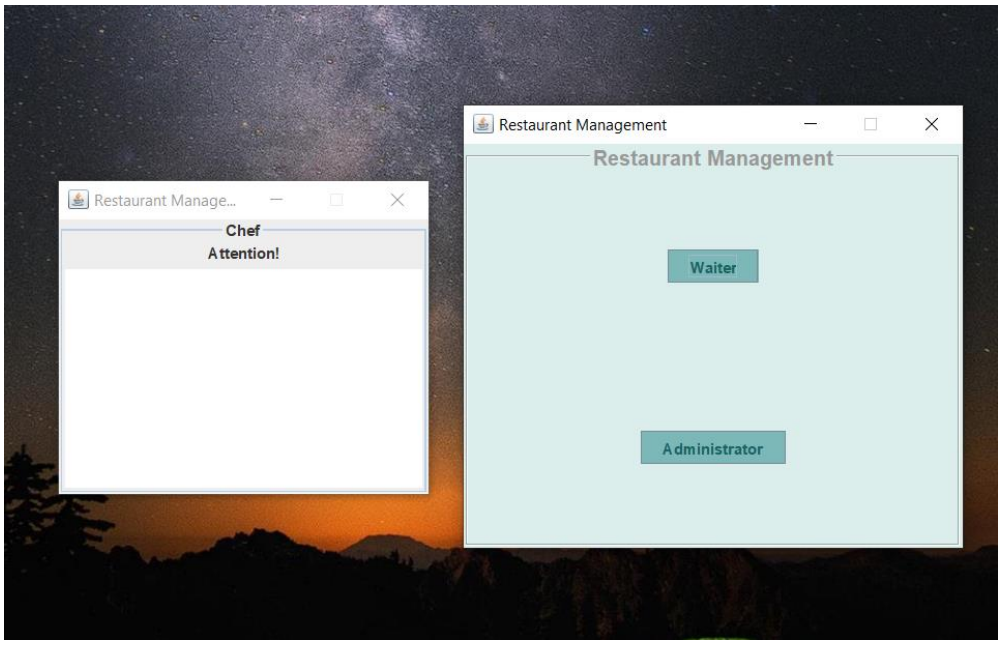

```
public static void serialize(Restaurant restaurant){
   String filename = "restaurant.ser";
   try {
      FileOutputStream file = new FileOutputStream(filename);
      ObjectOutputStream out = new ObjectOutputStream(file);
      out.writeObject(restaurant);
      out.close();
      file.close();
      System.out.println("Object serialized successfully.");
   }catch (FileNotFoundException e) {
      e.printStackTrace();
   } catch (IOException e) {
      e.printStackTrace();
   }
}

public static Restaurant deserialize(String filename){
   Restaurant restaurant = new Restaurant();
   try {
      FileInputStream file = new FileInputStream(filename);
      ObjectInputStream in = new ObjectInputStream(file);
      restaurant = (Restaurant) in.readObject();
      in.close();
      file.close();
      System.out.println("Object deserialized successfully.");
   } catch (FileNotFoundException e) {
      restaurant = new Restaurant();
      serialize(restaurant);
      return restaurant;
   } catch (IOException e) {
      restaurant = new Restaurant();
      serialize(restaurant);
      return restaurant;
   } catch (ClassNotFoundException e) {
   }
   return restaurant;
}
```

4) Application
   a. App – class designed for starting the application, contains only the main method

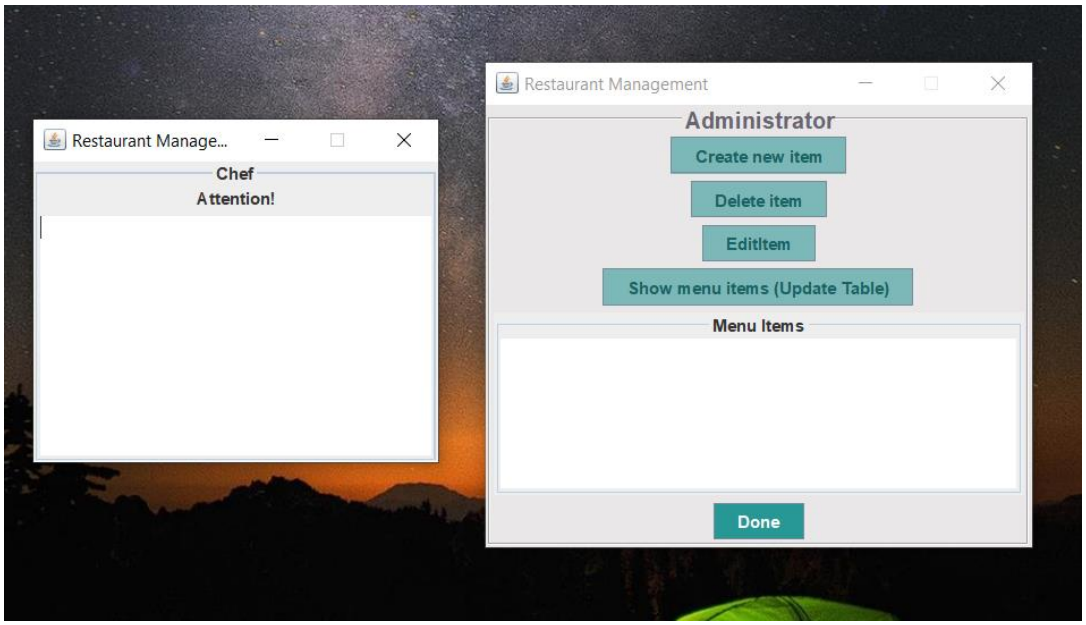


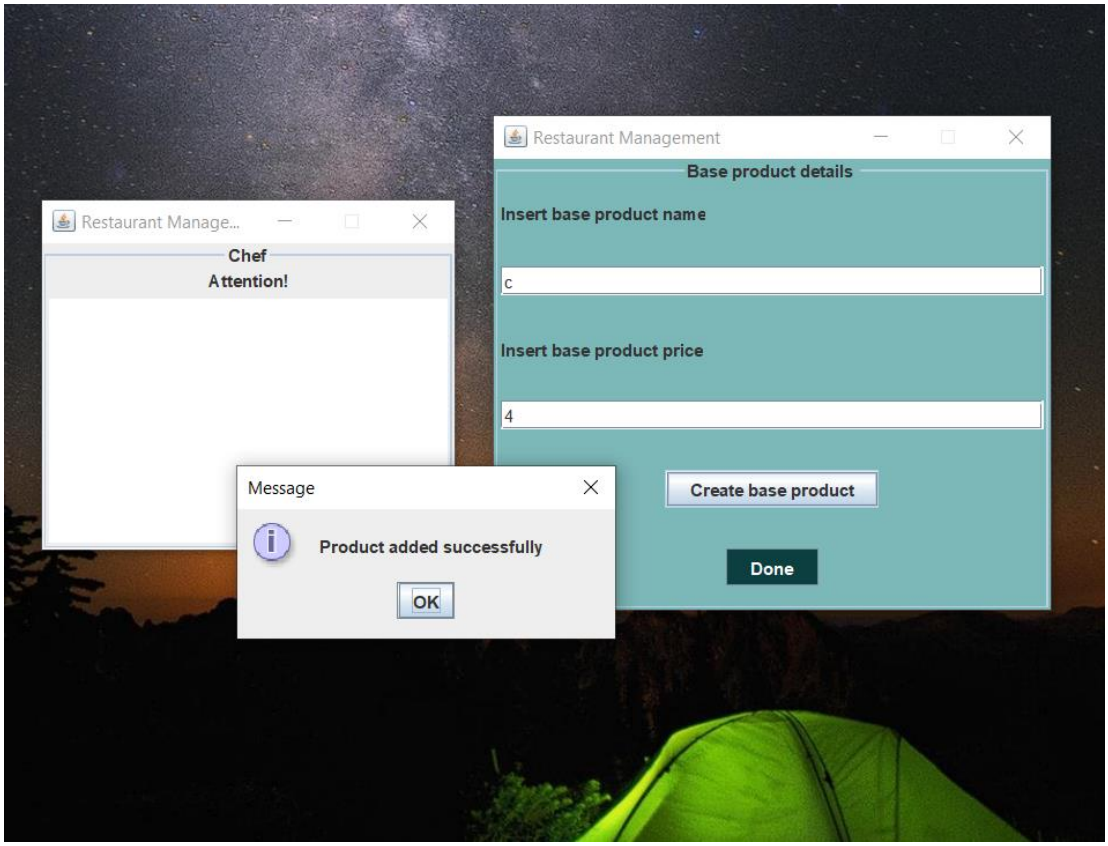
Results

Select „Adminitrator” button:



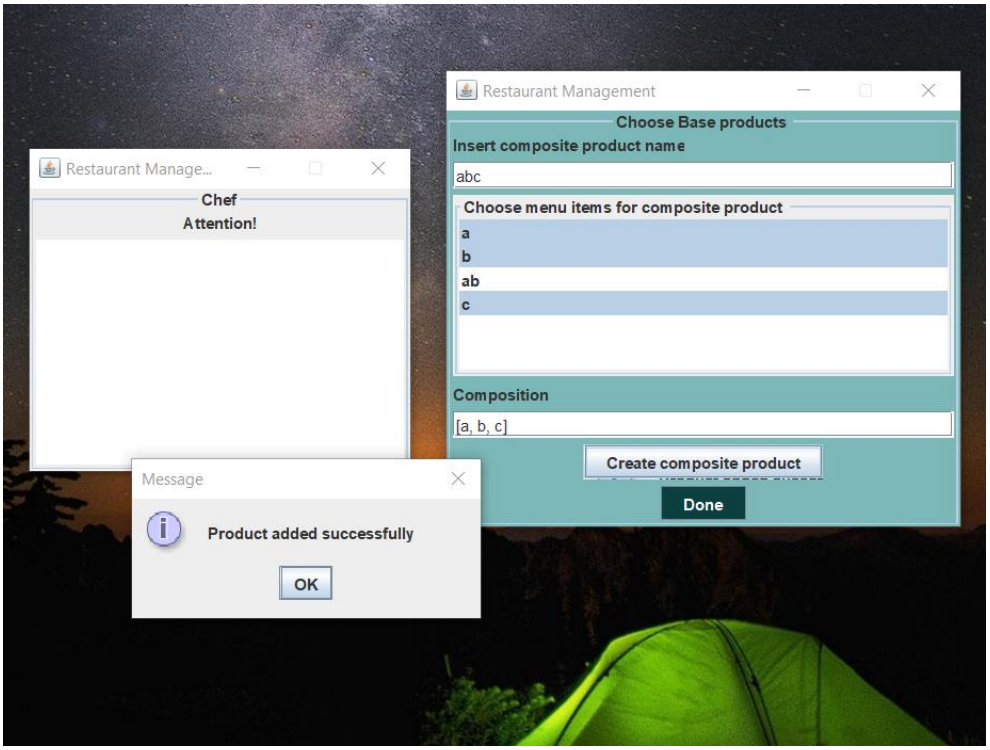
Select afterwords „Create new item button”. Insert the details and then press „Create base product” button. Click „OK”, the „Done” to go back to the previos panel.

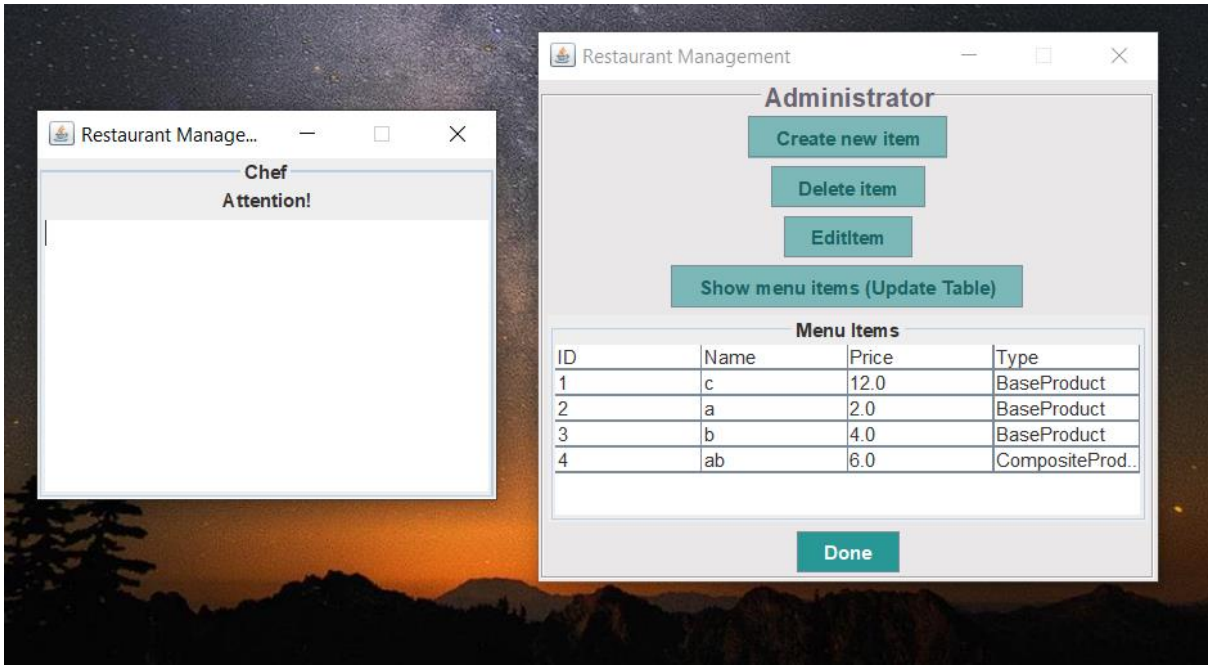




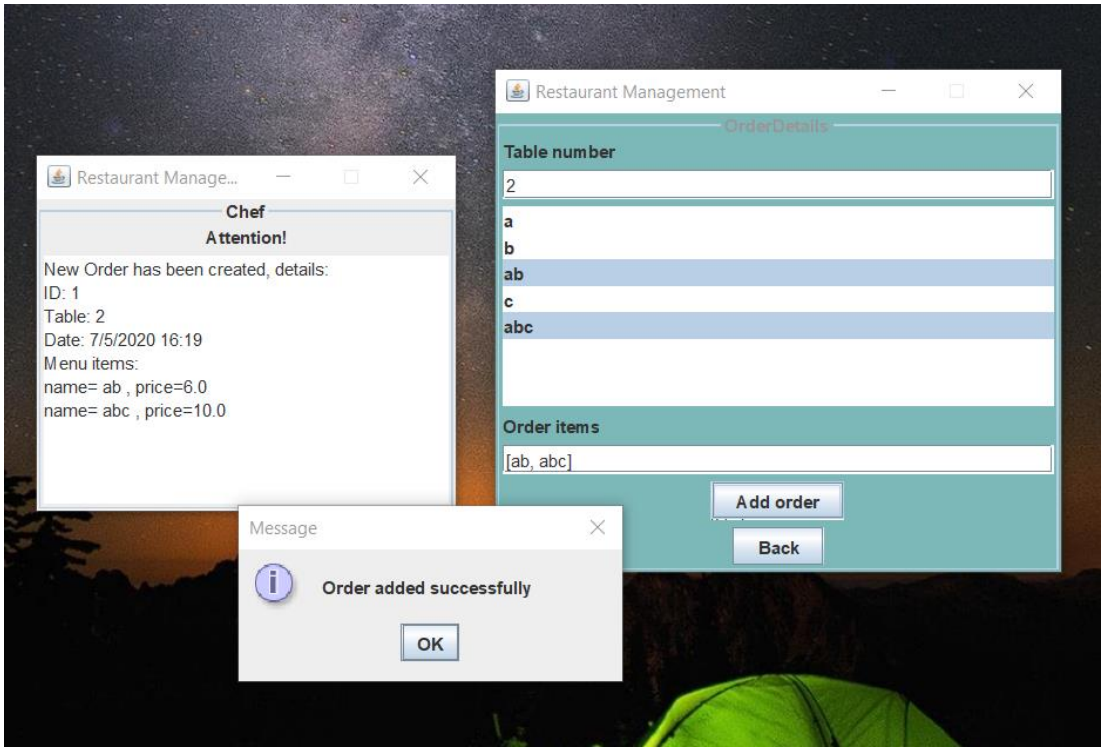
Create a composite product by selecting the items, inserting the name.

Pressing „Show menu items(Update table)” will display available menu items. After operations like edit or delete you need to press again for refreshing the data.

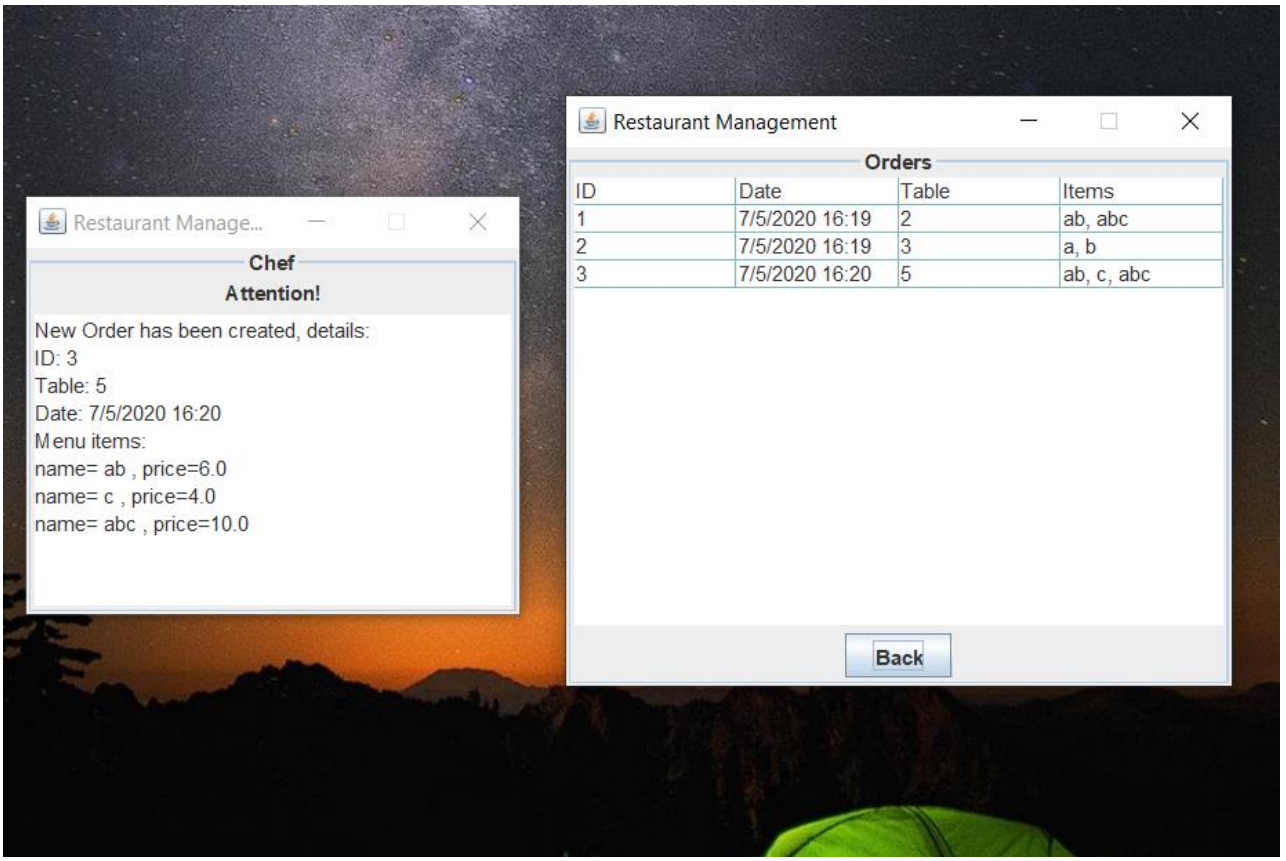


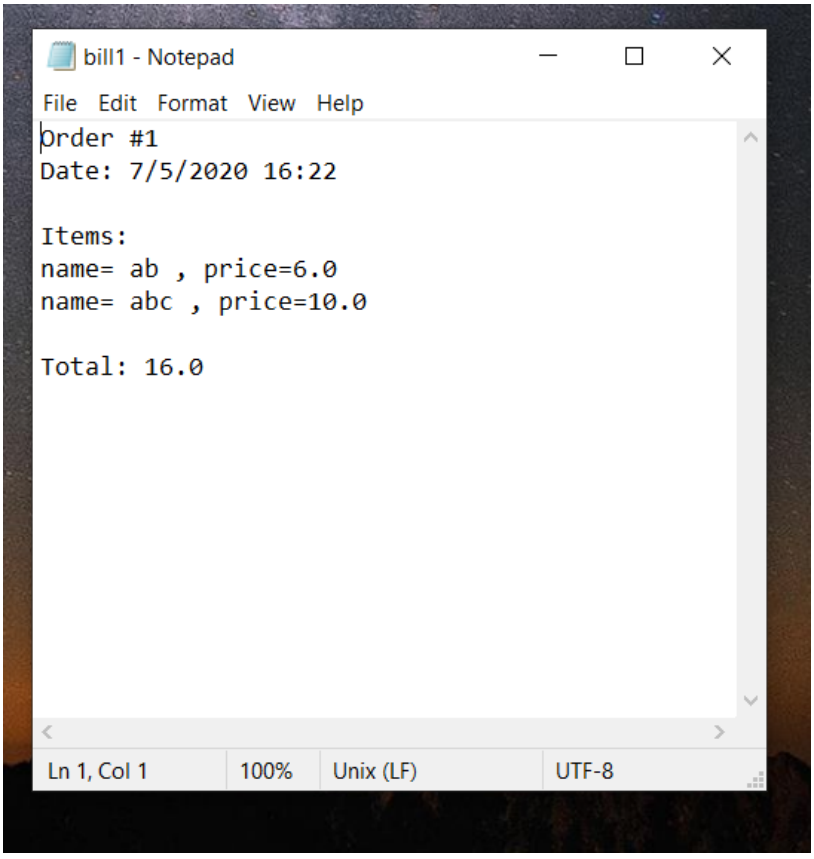


Press „Done” or „Back” to get back to main panel.



Insert an order by choosing multiple menu items. A composite product in an order will lead to a message displayed on the chef frame. When everything is done, generate the bill.


Restaurant Manage...
Chef
Attention!
New Order has been created, details:
ID: 3
Table: 5
Date: 7/5/2020 16:20
Menu items:
name= ab , price=6.0
name= c , price=4.0
name= abc , price=10.0
Restaurant Management
Orders
ID
Date
Table
Items
1
7/5/2020 16:19
2
ab, abc
2
7/5/2020 16:19
3
a, b
3
7/5/2020 16:20
5
ab, c, abc
Back


bill1 - Notepad
File Edit Format View Help
Order #1
Date: 7/5/2020 16:22
Items:
name= ab , price=6.0
name= abc , price=10.0
Total: 16.0
Ln 1, Col 1
100%
Unix (LF)
UTF-8

Conclusions

By doing this project I became more aware of the importance of good organisation. It is better to analyse carefully the problem, sketch a solution and then model it properly.

By the means of this project I also learned to use serialization, Observable interface. I also gained more experienced in designing GUI.

As far as future improvements could be made, a login method could be designed for the users, so that Waiter, Administrator and Chef are separted.

Apart from this multiple waiters, administrators and chefs could login, and we could have information about the person performing operations like inserting menu items/orders etc. for example.

Also, an evidence of the base product should be kept, so that when the restaurant is nearly out of certain base product, the administrator should be notified too.

Bibliography

Lecture and laboratory materials

https://www.youtube.com/watch?v=bKPGEqJHWaE

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

https://docs.oracle.com/javase/tutorial/uiswing/components/table.html