Restaurant Management

Rad Lidia

Group 30424

UTCN

1. Objective

The principal objective is to design an application for processing menu items and orders for a restaurant. The restaurant management system that 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.

In order to obtain this functionality, the main problem can be decomposed into several steps needed to reach the goal:

• Creating a good-looking, fully-functional user interface

• Implementing the provided UML class diagram

• Implementing algorithms to run according to the main objective.

• Linking the algorithms to the graphical user interface, in order to display the right output to the user, and to receive the right input from the user.

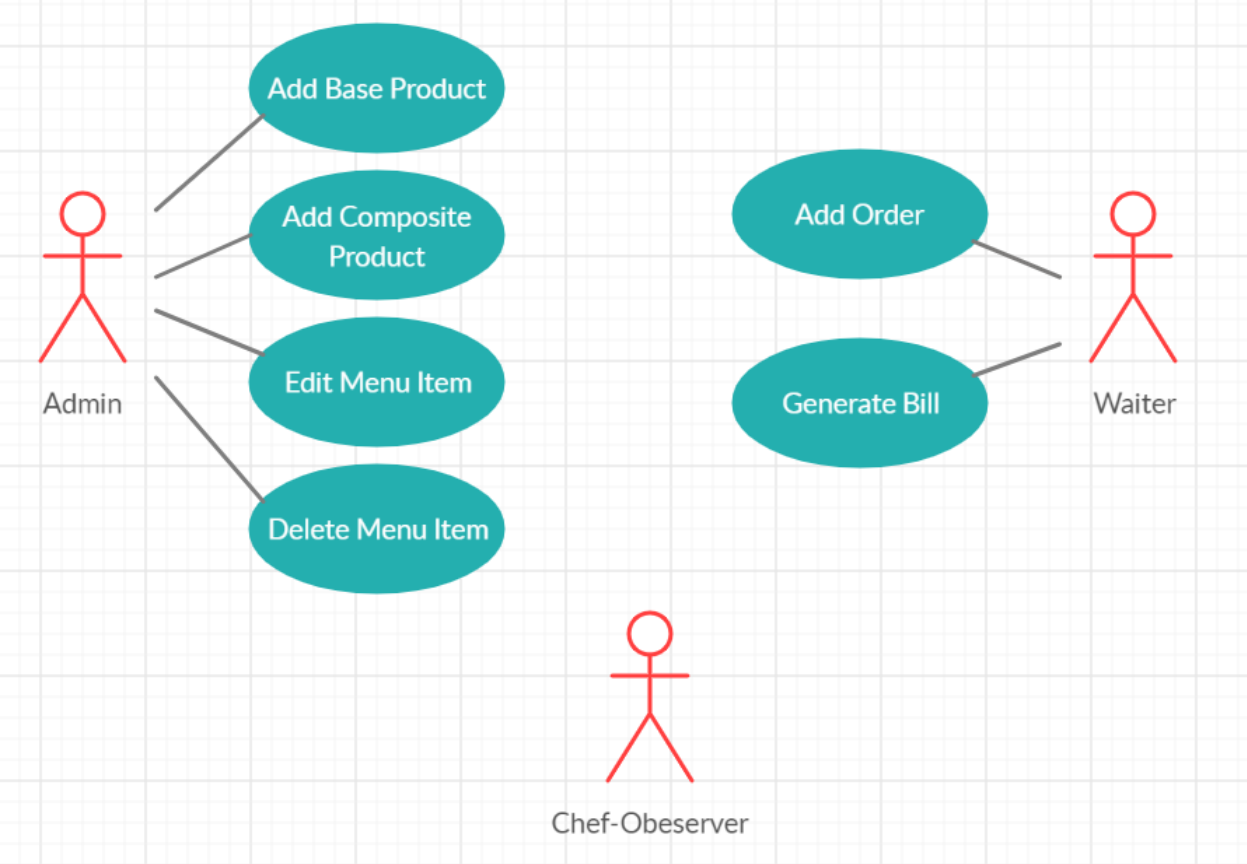
2. Analysis of the problem

This application is built to perform operations usually used by a restaurant administrator, waiter and chef. It shows to the user the menu of the restaurant, and allows him (in the "Administrator" window) to modify its contents: add, remove or edit a certain item.

The main goal of the application is to allow the management of a restaurant system: keep in mind the restaurant menu, edit it, allow the waiter to place an order, compute a bill, and allow the chef to be notified when a new order has been placed.

a) Use Cases

To better illustrate how the application can be used, I have built a use-case diagram:



- Use case Administrator view:

  -firstly, from the Home menu choose Administrator - an interface it will appear

- for inserting a base product, choose Add Base button

-insert the name and the price of the base product that you want to insert, then click Add button - the product will appear in the Administrator frame, in a table

- for inserting a composite product, choose Add Composite button

-first, choose from the menu table, the components that you want to exist in the composite product

-if no product were selected, an message box will appear on the screen

-insert the name of the composite product that you want to form, then click Add button - the composite product will appear in the Administrator frame, in a table

- for editing a product, choose Edit button

-choose from the menu table, the product that you want to edit

-if no product were selected, an message box will appear on the screen

-put in the designed fields the new name and new price for the product selected, then click Edit – the product will appeared in the menu table with the new information

- for deleting a product

-choose from the menu table, the product that you want to delete

-if no product were selected, an message box will appear on the screen

-then click Delete – the product will be erased from the menu table

- Use case Waiter view:

-from the Home menu choose Waiter - an interface will appear with a table in which contains the already inserted orders

- for inserting an order, choose Add Order

-insert the id and the table of the order that you want to insert

-choose the menu items, then click Add button – the oreder will appear on the oreders table with the date from when it was created

- for generating a bill of an order

-select from the orders table the order that you want to be generated into a .txt file

-choose Get Bill – a .txt file containing the nr of order, the table, the date, the ingredients with their price plus the total price of the order will be generated

- Use case Chef view:

-from the Home menu choose Chef- an interface with a list of ordered composite products that need to be cooked by the chef

b) Modeling

An important aspect in finding a solution is to correctly represent the data. Here, in our problem, we manage to use 3 types of designs: Composite Design Pattern, Observer Design Pattern and Design by contract.

Composite pattern is a partitioning design pattern describing a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies. We chose to use the Composite Design Pattern for defining the classes MenuItem, BaseProduct and CompositeProduct, where a menu item is an abstract class an represents the Component, a base product is a leaf and composite product is the Composite.

The Observer Pattern defines a one to many dependency between objects so that one object changes state, all of its dependents are notified and updated automatically. Observer pattern, as the

name suggests it is used for observing some objects, watch for any change in state or property of subject. Here, the Observer Design Pattern is used to notify the chef each time a new order containing a composite product is added, such that the cook can know what to cook next.

Design by Contract is an approach to design a program such that the software source code can self-checking at runtime. This is achieved through the introduction of "contracts"- executable code contained within the source that specifies obligations for classes, methods, and their callers. In Design by contract we have 3 types of expressions: preconditions(conditions that must hold before a method can execute), postconditions(conditions that must hold after a method completes) and invariants(condition that must hold anytime a client could invoke an object's method). In Class Restaurant, at each method we put pre and postconditions, also we created the invariant method isWellFormed() to verify if all orders from map have a collection of menu items associated and if the collection is not null.

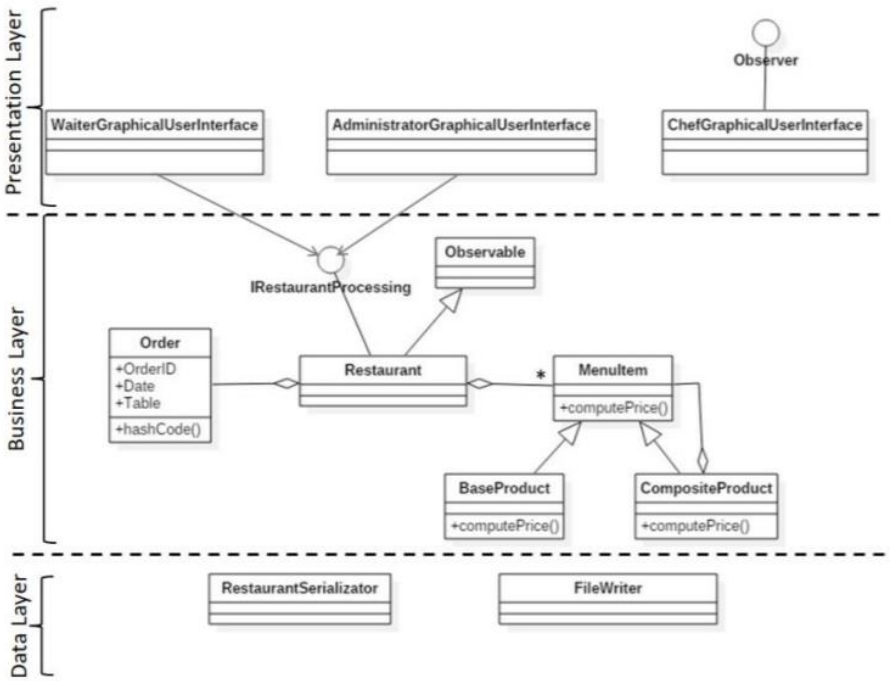
3. Design

The application was designed for managing the items from a menu of a restaurant and for performing orders for a restaurant. Products and Order features must be inserted from the user interface and serialized into file. They can be also edited or deleted after being inserted.

Data Structures:

As data structures, there was used an ArrayList of MenuItems for storing the menu items of the restautant and HashMap for linking an order with its associated items. We define a structure of type <Map> for storing the order related information in the Restaurant class. The key of the Map was formed of objects of type Order, for which the hashCode() method was overwritten to compute the hash value within the Map from the attributes of the Order.

The following UML diagram shows the entire structure of the program along with all the classes, interfaces and relationships between them.



The design of this app follows a layered design architecture. We respect the 3 packages: business, data, presentation shown in the diagram, and we add the main package for a better structure.

The business package is represented by the classes BaseProduct, CompositeProduct, MenuItem, Order, Restaurant and IRestaurantProcessing interface. These classes describe the logic of the application

The data package contains tha WriteFile class that generate a .txt file and RestaurantSerializator class that save the information from the Restaurant class in a .ser file using serialization

The main_access is the package that contains the main class, where the application starts.

The presentation package, has the 9 classes: AdministratorView, WaiterView, ChefView, AddBase, AddComposite, Edit, AddOrder, MainFrame and the Controller class that controls every interface.

4. Implementation

There are eighteen classes in this project. So, we'll explain briefly each class, especially the important methods contained.

- Data package:

RestaurantSerializator class : implements the serializable interface; it contains static methods: serialize si deserialize. The serialize method has as parameters a list with the menu items from the menu and the name of the file where the information will be written.

```
public static  void serialize(ArrayList<MenuItem> menu,String filename){
try {
    FileOutputStream fileOut=new FileOutputStream(filename);
    ObjectOutputStream out=new ObjectOutputStream(fileOut);
    out.writeObject(menu);
    out.close();
    fileOut.close();
    System.out.println("Serialized successfully");
}catch (IOException e){
    e.printStackTrace();
}
```

The deserialize method reads the information from the restaurant.ser and returns the list of the menu items from the menu, in order to be available at the start of the program.

```
public static ArrayList<MenuItem> deserialize(String filename){
    ArrayList<MenuItem> menu=new ArrayList<MenuItem>();
    try{
        FileInputStream fileIn=new FileInputStream(filename);
        ObjectInputStream in=new ObjectInputStream(fileIn);
        menu= (ArrayList<MenuItem>) in.readObject();
        in.close();
        fileIn.close();
        System.out.println("Deserialized successfully");

    }catch(IOException e){
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    return menu;
}
```

The FileWriter class has the writeBill method with the parameters: the name of the file where the bill will be printed and the billInfo - the information of the bill that will be printed.

```
public class FileWriter {
    public void writeBill(String filename, String billInfo) {
        try {
            PrintWriter file = new PrintWriter(filename, "UTF-8");
            file.write(billInfo);
            file.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

```
public class FileWriter {
    public void writeBill(String filename, String billInfo) {
        try {
            PrintWriter file = new PrintWriter(filename, "UTF-8");
            file.write(billInfo);
            file.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

- Business package MenuItem abstract class that have the main methods: computePrice(), getComposed() and setComponents(ArrayList<MenuItem> list). This class implements Serializable.

  BaseProduct class has fields: name and price and contains setters and getters.

```
public class BaseProduct extends MenuItem implements Serializable{
    private String name;
    private float price;
```

ComposeProduct class has fields: name, price and components, represented in ArrayList, that contains the products of what the compose product is formed.

```
public class CompositeProduct extends MenuItem implements Serializable{
    private String name;
    private float price;
    private ArrayList<MenuItem> components = new ArrayList<MenuItem>();
```

It also have 2 important methods: getComposedBy() that forms a string out of the components stored into ArrayList, and computePrice() that calculates the total price of the order.

```
public String getComposedBy () {
    Iterator<MenuItem> i = components.iterator();
    StringBuilder sb = new StringBuilder();

    while(i.hasNext()) {
        MenuItem m = (MenuItem) i.next();
        sb.append(m.getName() + ",");
    }
    return sb.toString();
}

public float computePrice() {
    Iterator<MenuItem> i = components.iterator();

    float total = 0;
    while(i.hasNext()) {

        MenuItem m = (MenuItem) i.next();
        total += m.computePrice();
    }

  return total;
}
```

Order class has fields: orderID, date, table, components. It also contains the hashCode() method that gives the unique key for the HashMap, computed from the attributes of the Order.

```
public int hashCode() {
    int hash = 0;
    hash = this.orderID + this.date.getDay()+this.date.getMonth()
            +this.date.getYear()+ this.table;
    return hash;
}
```

The IRestaurantProcessing interface declares the methods that represent the operation that the administrator and waiter can do. These will be implemented in the Restaurant class.

```
}
```

Restaurant class extends Obervable and implements the interface IRestaurantProcessing. Also contains preconditions and postconditions from Design by contract.

```
        assert isWellFormed();
    }
```

One important method is createOrder:

```
public void createOrder(Order o, ArrayList<MenuItem> list) {
    assert o!=null && list.size()>0;
    assert isWellFormed();
    orders.put(o, list);
    for(int i = 0; i < list.size(); i++){
        if(list.get(i) instanceof CompositeProduct){
            this.setChanged();
            this.notifyObservers(list.get(i));
        }
    }
    assert isWellFormed();
    assert ordersContains(o,list);
}
```

The method checks first, the precondition – that the order exist and the list in greater than 0 and postconditions - that orders contains the order added previously. Then, if the product is composite type, notifies the chef to cook the order.

The generateBill method constructs the string tha will be printed in a .txt file.

```
public void generateBill(Order o, ArrayList<MenuItem> list) {
    assert list.size() > 0 && o != null;
    FileWriter f = new FileWriter();
    StringBuilder s = new StringBuilder();
    s.append("Order : " + o.getOrderID() + "\n");
    s.append("Table : " + o.getTable() + "\n");
    s.append("Date : " + o.getDate() + "\n");
    s.append("Ingredients : " + "\n");
    for(int i = 0; i < list.size(); i++){
        s.append((i + 1) + ". " + list.get(i).getName() + "    price : " + list.get
    }
    s.append("\n");
    s.append("Total : " + computeOrderTotal(list) + "\n");
    f.writeBill("bill.txt", s.toString());
    assert isWellFormed();
}
```

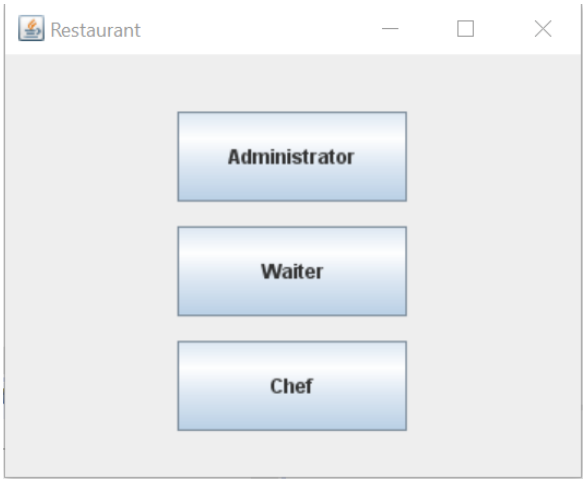
- Main package

  Contains only the main class, where the applications starts.

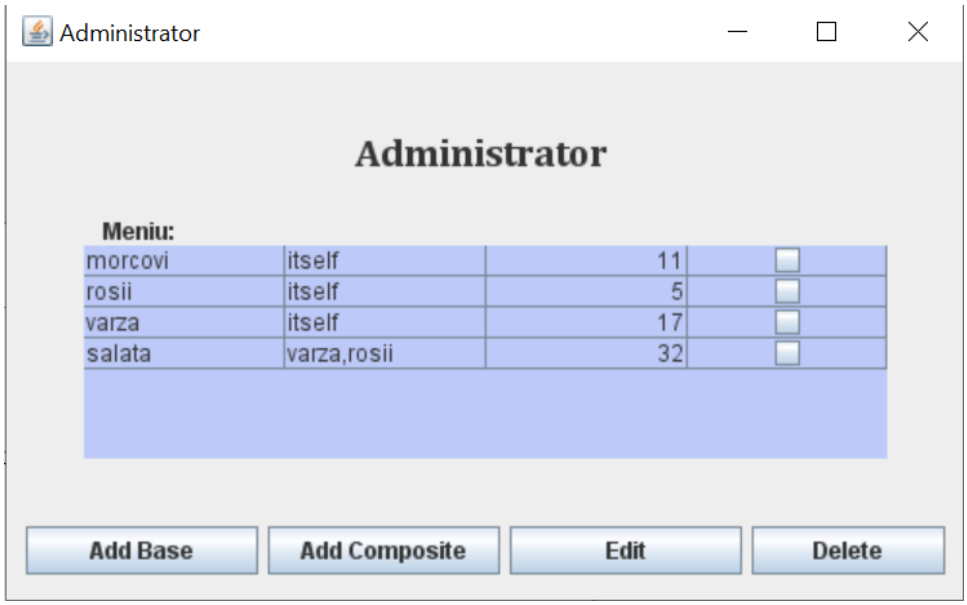
```
package Main;

import business.Restaurant;

public class Start {
    public static void main (String[] args) {

        Restaurant restaurant = new Restaurant();
        MainFrame login = new MainFrame(restaurant);
        Controller c = new Controller(login, restaurant);
        restaurant.addObserver(c.getView());
    }
}
```

- Presentation package

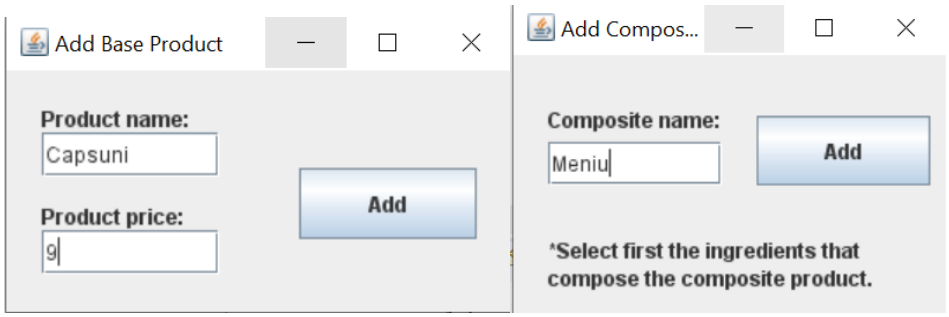
MainFrame class creates the start interface of the application, where we can choose our user.



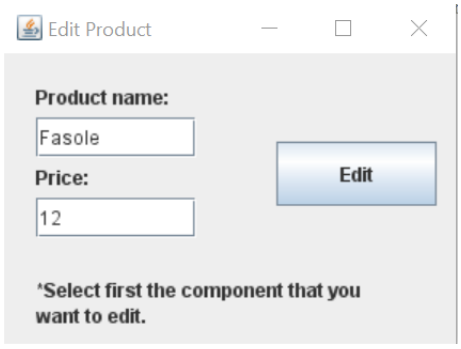
AdministratorView class creates the administrator interface that contains a JTable where all the items from the menu can be seen. As seen in figure, it has 4 button, to execute the operations that an administrator can do.



AddBase and AddComposite classes implement these two interface. A product can be added very easy.

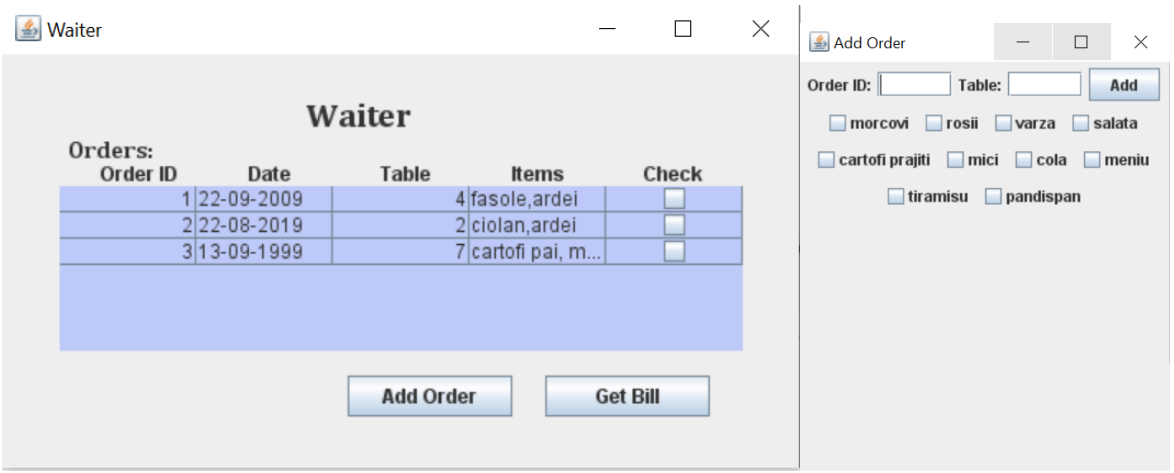


Edit class implement the interface where a product can be edit.

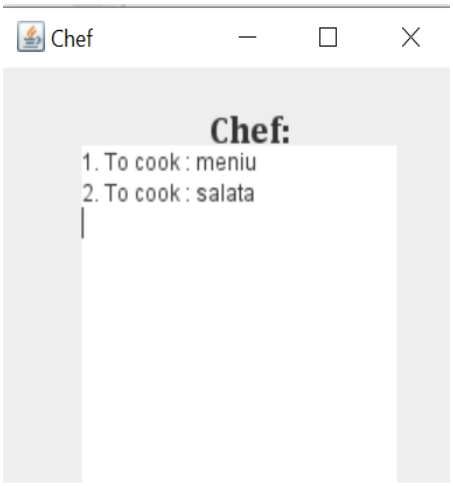


WaiterView class implements the interface of the waiter. It contains a table where old orders can be seen. Has 2 button that implements the operations that a waiter can do.

AddOrder class implements the interface where you can add an order.



ChefView class implements the chef view. When an order contains a composite product from menu, the chef is announced.



Controller class implements all the action listeners, coordinating all these interfaces. This is a part of code, where the composite product is added to the table of the administrator interface and also in the menu. There products that were checked before, are added toghether and put in din components variable of composite product by the method addComponent(). Then, the values from the AddComposite interface are collected, after that the item is printed on the Jtable by addCompositeToTable() method. Similary, all listeners of the buttons are implemented here, in controller class.

```
if (e.getSource() == composite.getAddButton()){
    int size = admin.getTableSize();
    admin.checkSelected();
    String name = composite.getName();

    CompositeProduct comp = new CompositeProduct(name);

    for(int i = 0; i < size; i++){
        MenuItem aux = admin.menuItemChecked(i, restaurant);
        if(aux != null){
            ((CompositeProduct) comp).addComponent(aux);
        }
    }

    float price = comp.computePrice();
    comp.setPrice(price);
    restaurant.createMenuItem(comp);
    admin.addCompositeToTable(comp);
    order.addCheckBox(restaurant, comp);
    RestaurantSerializator.serialize(restaurant.getMenu(),"restaurant.ser");

}
```

5. Results

After all the necessary steps, design, implementation, we managed to obtain a functional restaurant interface. The system have three types of users: administrator, waiter and chef. Each user have a set of operations that can be realized easier now with the help of the interface.

6. Conclusions

During the making of this assignment I improve my Java programming skills, by learning how to implement a quite complex interface and learning about Composite and Observer design patterns.

I have learned how to use serialization works and also about preconditions and postconditions.To sum up, my Java programming skills have been improved.

Future improvements of the project can be:

-improve the interface even more, by inserting more details or some imagines

-creating more functionality regarding the Restaurant class

7. Biobliography

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

(for better understanding of the requirements)

https://www.geeksforgeeks.org/java-util-observable-class-java/ (for Observer design pattern)

https://www.java67.com/2014/04/how-to-make-executable-jar-file-in-Java-Eclipse.html

(for generating .jar file)

https://app.creately.com/diagram/OI01BYOepkA/edit (for use case diagram)

https://stackoverflow.com (for various questions)