PROGRAMMING TECHNIQUES

ASSIGNMENT 4

RESTAURANT MANAGEMENT SYSTEM

Student: Chindea Miruna

Group: 30424

1. Homework objective

1.1. Main objective

The main objective of this assignment is to propose, design and implement 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 ordered through a waiter.

1.2. Secondary objectives

The secondary objectives of the homework, which will help in building the main objective, are:

1. Creating a Graphical User Interface for the system, in order to help the users interact with it. The graphical interface will contain a window for Administrator operations(add, edit, delete and see all menu items in a table), a window for Waiter operations(add and view all orders in a table, and compute the bill for an order), and a window of the Chef, where notifications with orders for the chef will be displayed.

2. Using the **Composite Design Pattern** for defining the classes MenuItem, BaseProduct and CompositeProduct.

3. Using the **Observer Design Pattern** to notify the chef each time a new order containing a composite product is added.

4. Implement the class Restaurant using a predefined JCF collection which uses a **hashtable** data structure. The hashtable key will be generated based on the class Order, which can have associated several MenuItems.

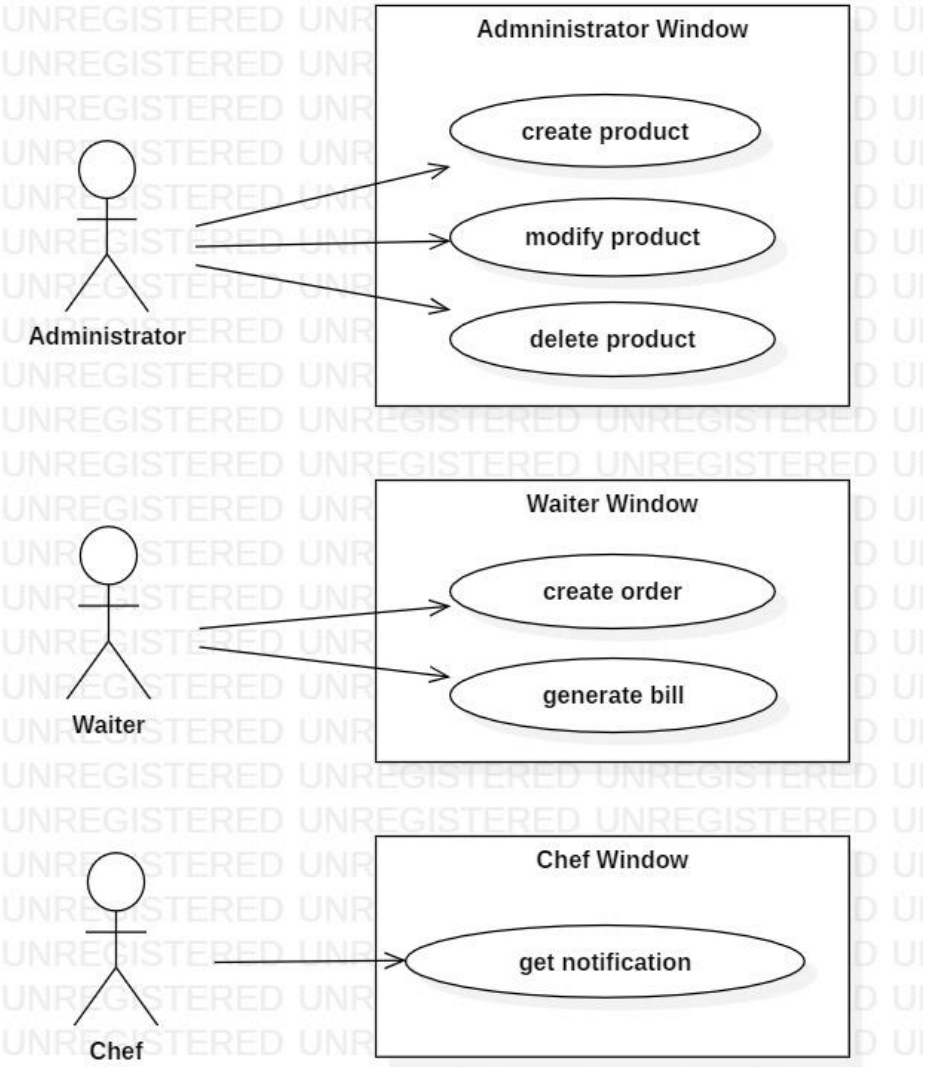
5. The menu items for populating the Restaurant object will be loaded/saved from/to a file using **Serialization**.

2. Problem analysis, scenarios, use cases

2.1. Functional requirements

There will be displayed a graphical interface for the users (administrator, waiter and chef) to interact with. The system should successfully implement the specific operations for each user, if the user correctly introduces the input.

2.2. Use cases



- **Use case title: create product**

  - **Summary**: The administrator creates a new product

  - **Actors**: Administrator

  - **Preconditions**: actor introduces data in all fields

  - **Main success scenario:**

    1. The user introduces product data
    2. The user presses “Add” button
    3. A new product is created
    4. The new product is displayed in the table

  - **Alternative sequences:**

    a) The id of the new product already exists

    1. The system communicates the issue to the user
    2. The scenario returns to step 1

    b) The id of the product is not an integer or is < 0

    1. The system communicates the issue to the user
    2. The scenario returns to step 1

    c) The price of the product is not a floating point number or is < 0

    1. The system communicates the issue to the user
    2. The scenario returns to step 1

- **Use case title: modify product**

  - **Summary**: The administrator modifies an existing product

  - **Actors**: Administrator

  - **Preconditions**: actor introduces data in all fields

  - **Main success scenario:**

    1. The user introduces new data for the product
    2. The user presses “Edit” button
    3. The existing product is modified
    4. The new product information is displayed in the table

- **Alternative sequences:**

a) The product user tries to modify doesn't exist

1. The system communicates the issue to the user
2. The scenario returns to step 1

b) The id of the product is not an integer or is < 0

1. The system communicates the issue to the user
2. The scenario returns to step 1

c) The price of the product is not a floating point number or is < 0

1. The system communicates the issue to the user
2. The scenario returns to step 1

- ## Use case title: delete product

- **Summary**: The administrator deletes an existing product
- **Actors**: Administrator
- **Preconditions**: actor introduces data in all fields
- **Main success scenario:**

1. The user introduces the id of the product he wants to delete
2. The user presses "Delete" button
3. The product is deleted
4. The product is not displayed in the table anymore

- **Alternative sequences:**

a) The product user wants to delete doesn't exist

1. The system communicates the issue to the user
2. The scenario returns to step 1

- ## Use case title: create order

- **Summary**: The waiter creates a new order
- **Actors**: Waiter
- **Preconditions**: actor introduces data in all fields

- **Main success scenario:**

1. The user introduces order information
2. The user presses “Add” button
3. A new order is created
4. The new order is displayed in the table

- **Alternative sequences:**

a) The id of the new order already exists

1. The system communicates the issue to the user
2. The scenario returns to step 1

b) The id of the order is not an integer or is < 0

1. The system communicates the issue to the user
2. The scenario returns to step 1

c) The table is not an integer or is < 0

1. The system communicates the issue to the user
2. The scenario returns to step 1

d) The products waiter wants to order do not exist

1. The system communicates the issue to the user
2. The scenario returns to step 1

- ## Use case title: generate bill

- **Summary**: The waiter generates the bill for an order
- **Actors**: Waiter
- **Preconditions**: actor introduces data in all fields
- **Main success scenario:**

1. The user introduces the id of the order
2. The user presses “Get bill” button
3. The bill for the order is created as a .txt file

- **Alternative sequences:**

a) The order doesn’t exist

1. The system communicates the issue to the user
2. The scenario returns to step 1

- **Use case title: get notification**

  - **Summary**: The chef is notified when a new order is placed

  - **Actors**: Chef

  - **Main success scenario:**

    1. The chef is notified when a new order is placed
    2. The products in the order are displayed

3. Design

3.1. Patterns

3.1.1. Composite Design Pattern

The **Composite Design Pattern** is used for modelling the classes MenuItem, BaseProduct and CompositeProduct. Composite pattern is a partitioning design pattern and describes 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.

The Composite Pattern has four participants:

- **Component**: declares the interface for objects in the composition, implements the default behavior as appropriate, declares an interface for accessing and managing child components (in this project, MenuItem)
- **Leaf**: represents leaf objects in the composition; defines the primitive behavior (in this project, BaseProduct)
- **Composite**: stores children / composite behavior (in this project, CompositeProduct)
- **Client**: accesses objects in the composition via Component interface (in this project, the Administrator)

3.1.2. Observer Design Pattern

The **Observer Design Pattern** is used to notify the chef each time a new order containing a composite product is added. As the name suggests it is used for observing some objects. Observers watch for any change in state or property of subject. Suppose you are interested in a particular object and want to get notified when its state changes then you observe that object and when any state or property change happens to that object, it gets notified to you.

The Observer Design Pattern has four components:

- Subject: knows its observers, has any number of observers, provides an interface to attach and detaching observer object at run time (in this project, the Restaurant)
- Observer: provides an update interface to receive signal from subject (in this project, the Chef)
- ConcreteSubject: stores state of interest to ConcreteObserverobjectsSend notification to its observer
- ConcreteObserver: maintains reference to a ConcreteSubjectobject, maintains observer state consistent with subjects

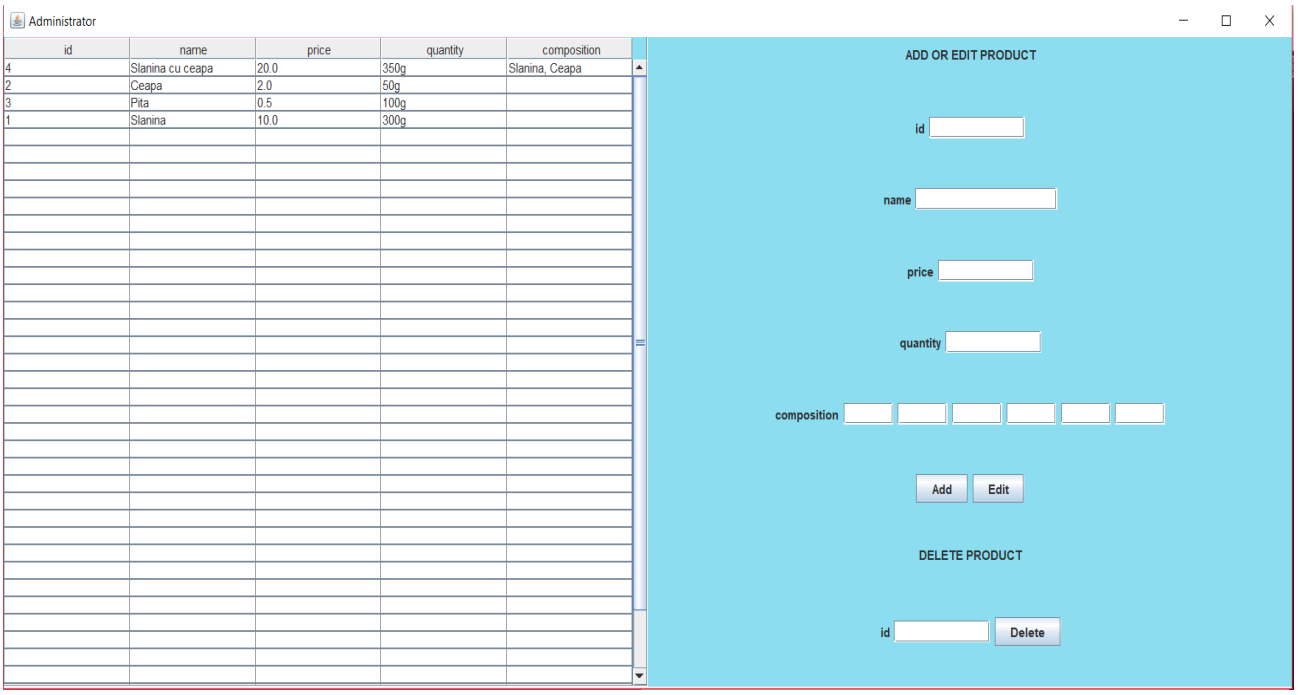
3.2. GUI Design

In the Graphical User Interface, there will be displayed three windows:

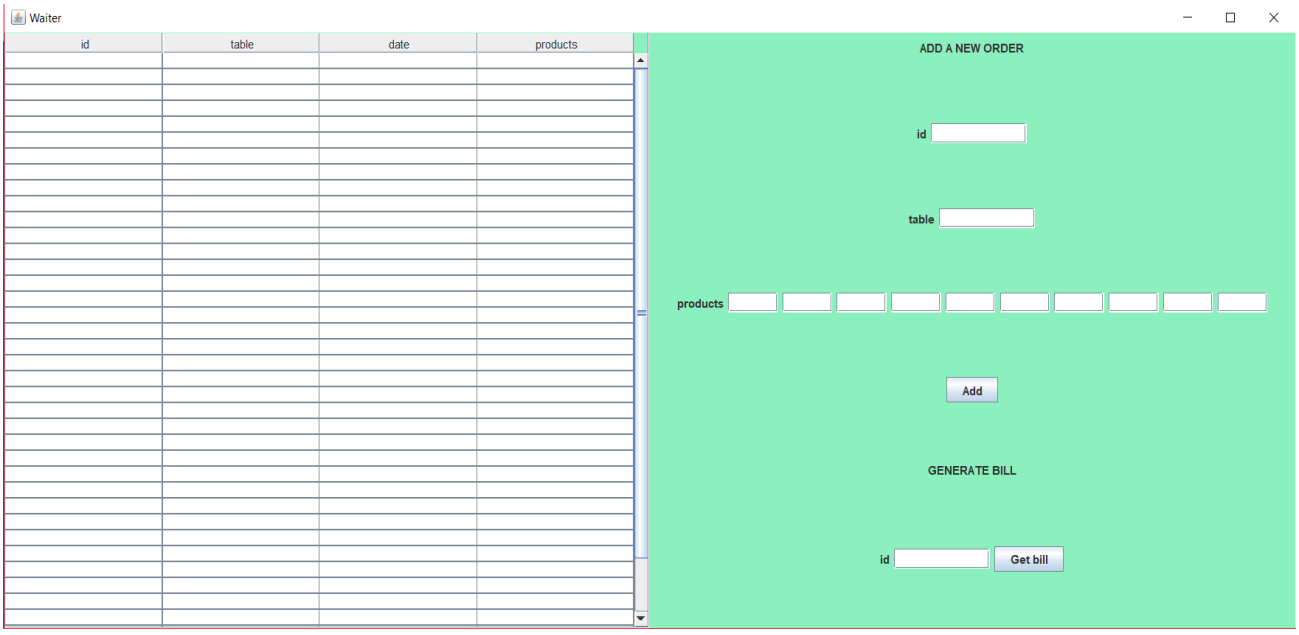
- a window for **Administrator** operations: add new MenuItem, edit MenuItems, delete MenuItems, view all MenuItems in a table (JTable)

- a window for **Waiter** operations: add new Order, view all Orders in a table (JTable), compute bill for an Order

- a window for the **Chef**, who gets notified when a new order is placed



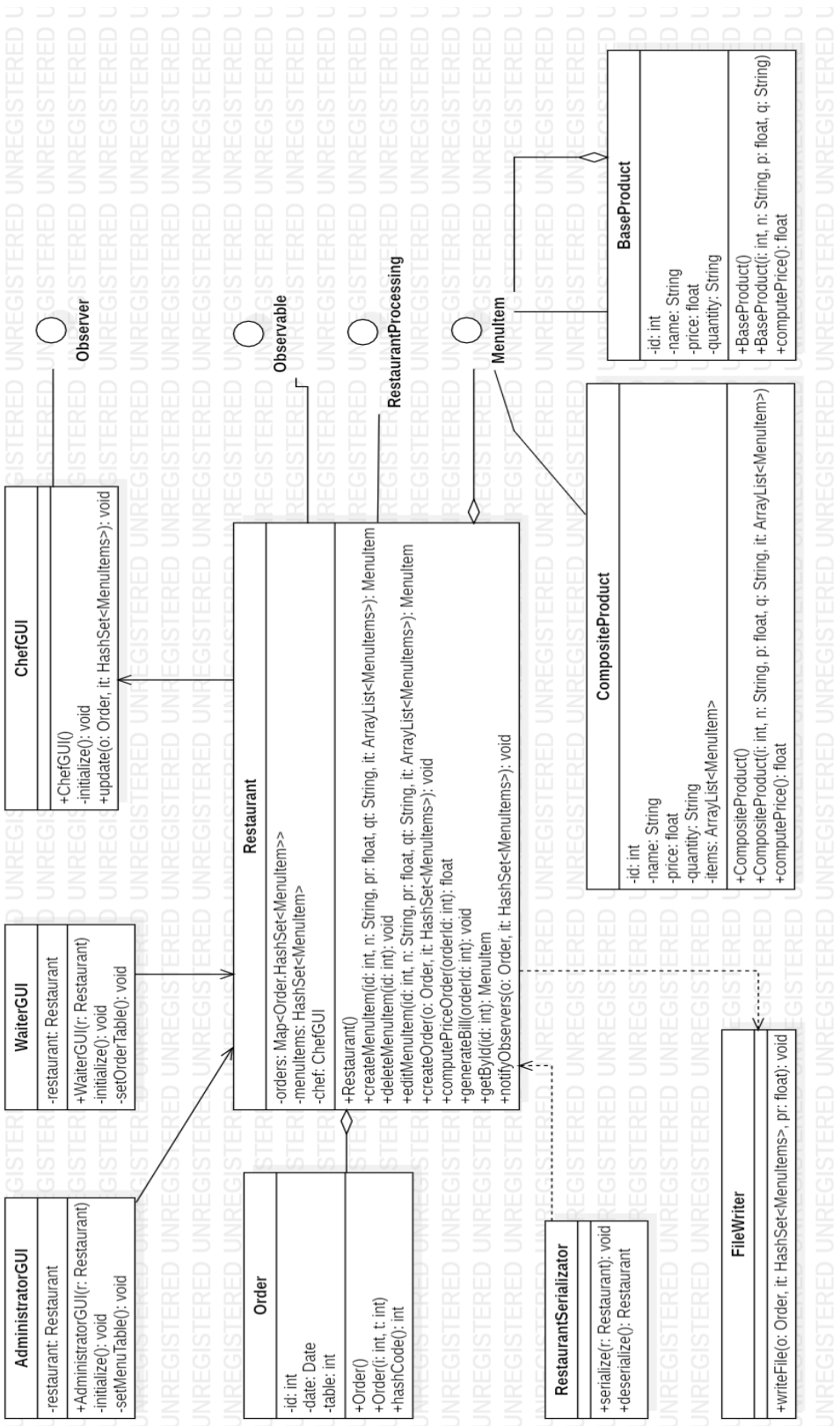
Administrator Window



Waiter Window

Chef Window

3.3. Class Diagram



3.4. Relationships

- Restaurant-Order: there is an aggregation relationship between these two classes. The restaurant contains a list of orders.

- Restaurant-AdministratorGUI: there is an association relationship between these two classes. The AdministratorGUI class has a Restaurant entity as attribute.

- Restaurant-WaiterGUI: there is an association relationship between these two classes. The WaiterGUI class has a Restaurant entity as attribute.

- Restaurant-ChefGUI: there is an association relationship between these two classes. The Restraurant class has a ChefGUI entity as attribute.

- ChefGUI-Observer: the ChefGUI class implements Observer interface.

- Restaurant-Observable: the Restaurant class implements Observable interface.

- Restaurant-RestaurantProcessing: the Restaurant class implements RestaurantProcessing interface.

- Restaurant-MenuItem: there is an aggregation relationship between Restaurant and the classes that implement MenuItem interface. The restaurant contains a list of MenuItems.

- CompositeProduct-MenuItem: the CompositeProduct class implements MenuItem interface.

- CompositeProduct-MenuItem: the CompositeProduct class contains a list of entities that implement MenuItem interface.

- BaseProduct-MenuItem: the BaseProduct class implements MenuItem interface.

- Restaurant-FileWriter: there is a dependency relationship between these two classes. The Restaurant class calls the file writer in order to export an order as .txt file.

- Restaurant-RestaurantSerializator: there is a dependency relationship between these two classes. The RestaurantSerializator serializes an entity of Restaurant class.

4. Implementation

In this chapter, there will be rigorously described:

- all the classes with their attributes and important methods;
- all the created and implemented interfaces;
- the Graphical User Interface.

4.1. Classes

4.1.1. Restaurant

This class represents the Restaurant entity. It has as attributes a list of menu items, a map of orders and menu items, and a chef.

Important methods:

- public MenuItem **createMenuItem**(int id, String name, float price, String quantity, ArrayList<MenuItem> items): creates a new menu item and returns it;
- public void **deleteMenuItem**(int id): removes an item from the menu;
- public void **editMenuItem**(int id, String name, float price, String quantity, ArrayList<MenuItem> items): modifies a menu item;
- public void **createOrder**(Order order, HashSet<MenuItem> items): creates a new order;
- public float **computePriceOrder**(int orderId): computes the price of an order;
- public void **generateBill**(int orderId): generates the bill for an order as .txt file;
- public MenuItem **getById**(int id): searches for an item by its id and returns the item;
- public void **notifyObservers**(Order order, HashSet<MenuItem> items): notifies the observer(chef);
- protected boolean **isWellFormed**(): checks if the entity is well formed.

```
public class Restaurant implements Observable, RestaurantProcessing,

    private Map<Order, HashSet<MenuItem>> orders;
    private HashSet<MenuItem> menuItems;
    private ChefGUI chef;

    public Restaurant() {
        this.setOrders(new HashMap<Order, HashSet<MenuItem>>());
        this.setMenuItems(new HashSet<MenuItem>());
    }
```

4.1.2. Order

This class represents the Order entity. It has as attributes an id, a date and a table. Besides the getters and setters it has a hashCode() method, which computes the hash code of an order entity, using its attributes, in order to place the order in the restaurant.

```
public class Order implements java.io.Serializable{

    private int id;
    private Date date;
    private int table;

    public Order() {
        this.id = 0;
        this.date = new Date();
        this.table = 0;
    }

    public Order(int id, int table) {
        this.id = id;
        this.date = new Date();
        this.table = table;
    }
```

4.1.3. BaseProduct

This class represents the base product entity. It has as attributes an id, a name, a quantity and a price. This class only contains its getters and setters.

```
public class BaseProduct implements MenuItem, java.io.Serializable{

    private int id;
    private String name;
    private float price;
    private String quantity;

    public BaseProduct() {
        this.id = 0;
        this.name = new String();
        this.quantity = new String();
        this.price = 0;
    }
```

4.1.4. CompositeProduct

This class represents the composite product entity. The difference between composite product and base product is that the composite one has a list of entities that implement MenuItem interface (BaseProduct or CompositeProduct).

```
public class CompositeProduct implements MenuItem,java.io.Serializable {

    private int id;
    private String name;
    private float price;
    private String quantity;
    private ArrayList<MenuItem> items;

    public CompositeProduct() {
        this.id = 0;
        this.name = new String("");
        this.price = 0;
        this.quantity = new String("");
        this.items = new ArrayList<MenuItem>();
    }
```

4.1.5. FileWriter

This class is used to export the bill for an order as a .txt file. It has only one method, which takes as attributes the details to be written in the file.

```
public class FileWriter {

    public static void writeFile(Order order, HashSet<MenuItem> items, float price) {
        String orderName = "Order " + order.getId();
        try {
            PrintWriter writer = new PrintWriter(orderName);
            writer.println("Order number: " + order.getId());
            writer.println("Table: " + order.getTable());
            writer.println("Date: " + order.getDate() );
            writer.println();
            for(MenuItem item : items) {
                writer.println(item.getName() + " " + item.computePrice());
            }
            writer.println();
            writer.println("Total: " + price);
            writer.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

}
```

4.1.6. RestaurantSerializator

This class contains only 2 methods, and it is used for serializing and deserializing a restaurant entity.

```java
public class RestaurantSerializator {

    public static void serialize(Restaurant restaurant) {
        try {
            FileOutputStream fileOut = new FileOutputStream("restaurant.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(restaurant);
            out.close();
            fileOut.close();
        }catch(IOException i) {
            i.printStackTrace();
        }
    }

    public static Restaurant deserialize() {
        Restaurant restaurant = null;
        try {
            FileInputStream fileIn = new FileInputStream("restaurant.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            restaurant = (Restaurant) in.readObject();
            in.close();
            fileIn.close();
        }catch(IOException i) {
            i.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return restaurant;
    }

}
```

4.2. Interfaces

4.2.1. MenuItem

This interface is implemented by BaseProduct and CompositeProduct classes. The restaurant class has a list of entities that implement MenuItem interface.

```java
public interface MenuItem {

    public int getId();
    public void setId(int id);
    public String getName();
    public void setName(String name);
    public float computePrice();
    public void setPrice(float price);
    public String getQuantity();
    public void setQuantity(String quantity);
    public ArrayList<MenuItem> getItems();
    public void setItems(ArrayList<MenuItem> items);

}
```

4.2.2. Observable

This interface is implemented by Restaurant class, which is the observable object which notifies the chef that an order has been placed.

```
public interface Observable {

    public void notifyObservers(Order order, HashSet<MenuItem> items);
}
```

4.2.3. Observer

This interface is implemented by ChefGUI class, which is the observer object that is notified when an order has been placed and updates the window with the order.

```
public interface Observer {

    public void update(Order order, HashSet<MenuItem> items);
}
```

4.2.4. RestaurantProcessing

This interface is implemented by Restaurant entity and contains the main operations that will be performed in the restaurant, by the administrator and waiter.

```
public interface RestaurantProcessing {

    // Administrator operations
    public MenuItem createMenuItem(int id, String name, float price, String quantity, ArrayList<MenuItem> items);
    public void deleteMenuItem(int id);
    public void editMenuItem(int id, String name, float price, String quantity, ArrayList<MenuItem> items);

    // Waiter operations
    public void createOrder(Order order, HashSet<MenuItem> items);
    public float computePriceOrder(int orderId);
    public void generateBill(int orderId);

}
```

4.3. Graphical User Interface Classes

4.3.1. AdministratorGUI

This class contains the window for administrator operations: add product, modify product and delete product. The class contains a frame, panels, labels, text fields, buttons, a scroll pane and a table. It has a method called initialize(), which, according to its name, initializes the window, and a method called setMenuTable(), which populates the table with all the menu items in the restaurant.

```
private void setMenuTable() {
    String[] columns = { "id", "name", "price", "quantity", "composition" };
    String[][] rows = new String[40][5];
    int i = 0;
    for (MenuItem item : restaurant.getMenuItems()) {
        rows[i][0] = item.getId() + "";
        rows[i][1] = item.getName();
        rows[i][2] = item.computePrice() + "";
        rows[i][3] = item.getQuantity();
        String composition = new String("");
        // composite product
        if (item.getClass().toString().equals("class businessLayer.CompositeProduct")) {
            for (MenuItem item2 : item.getItems()) {
                if (item2.equals(item.getItems().get(item.getItems().size() - 1)))
                    composition += item2.getName() + "";
                else
                    composition += item2.getName() + ", ";
            }
        }
        rows[i][4] = composition;
        i++;
    }
    @SuppressWarnings("serial")
    DefaultTableModel model = new DefaultTableModel(rows, columns) {
        public boolean isCellEditable(int row, int col) {
            return false;
        }
    };
    table_menu.setModel(model);
    table_menu.repaint();
}
```

4.3.1. WaiterGUI

This class contains the window for waiter operations: add a new order and generate bill as .txt file. The class contains a frame, panels, labels, text fields, buttons, a scroll pane and a table. It has a method called initialize(), which, according to its name, initializes the window, and a method called setOrderTable(), which populates the table with all the orders in the restaurant.

```
private void setOrderTable() {
    String[] columns = { "id", "table", "date", "products" };
    String[][] rows = new String[40][4];
    int i = 0;
    for (Map.Entry<Order, HashSet<MenuItem>> order : restaurant.getOrders().entrySet()) {
        rows[i][0] = order.getKey().getId() + "";
        rows[i][1] = order.getKey().getTable() + "";
        rows[i][2] = order.getKey().getDate() + "";

        String items = new String("");
        for (MenuItem item : order.getValue()) {
            items += item.getName() + ", ";
        }

        items = items.substring(0, items.length() - 2);
        rows[i][3] = items;
        i++;
    }
    @SuppressWarnings("serial")
    DefaultTableModel model = new DefaultTableModel(rows, columns) {
        public boolean isCellEditable(int row, int col) {
            return false;
        }
    };

    table_order.setModel(model);
    table_order.repaint();
}
```

4.3.1. Chef

This class contains the window for the chef. The chef gets a notification when a new order is placed, with details like the order number and the items from the menu which the chef has to cook. The chef plays the role of the Observer, so it is implemented a method called update(), which is called as soon as an order is placed.

```
public void update(Order order, HashSet<MenuItem> items) {
    StringBuilder notification = new StringBuilder("");
    notification.append("Order " + order.getId() + ": ");
    for(MenuItem item : items) {
        notification.append(item.getName() + ", ");
    }
    t_notify.append(notification.substring(0, notification.length() - 2));
    t_notify.append("\n");
    t_notify.repaint();
}
```

5. Conclusions

In conclusion, if the input is correctly introduced, the system will successfully perform the operation for each user (administrator, waiter and chef). If the user introduces the input wrong (e.g. the id is not an integer or is smaller than zero, or the user tries to delete a product which doesn't exist), there will be displayed an error message which communicates the issue to the user.

I learnt from this project two new design patterns, Composite and Observer, and also about design by contract and serialization.

This application can be improved in many ways, for example:
- allowing more users to use the application, by creating a log-in system;
- allowing to order more items of the same kind;
- improving the GUI into a more complex and user-pleasing one;

6. Bibliography

https://www.geeksforgeeks.org/composite-design-pattern/
https://www.tutorialspoint.com/java/java_serialization.htm
https://www.geeksforgeeks.org/object-serialization-inheritance-java/
http://www.tutorialspoint.com/java/java_serialization.htm
http://javarevisited.blogspot.ro/2011/02/how-hashmap-works-in-java.html
http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html
http://javarevisited.blogspot.ro/2012/01/what-is-assertion-in-java-java.html
http://stackoverflow.com/questions/11415160/how-to-enable-the-java-keyword-assert-in-eclipse-program-wise