Assignment 4
Restaurant Management System

Faculty: Automation and Computer Science

Department: Computer Science

Student: Dragan Iulia-Andreea, **Group** 30423/1

1. Objective

The objective of this assignment is to design and implement a **restaurant management system**. The system has must have three different types of users: administrator, waiter chef, who can perform the following tasks:

- **Administrator** – add, delete, modify existing products from the menu
- **Waiter** – create a new order for a table, add elements from the menu, compute the bill for an order
- **Chef** – is notified whenever they must cook food that is ordered through a waiter (i.e. whenever a new order is added)

The program should follow the class diagram presented in the laboratory work. The developer has to ensure that the following Java concepts / design patterns are used:

- **Composite Design Pattern** for the MenuItem, BaseProduct & CompositeProduct classes – a CompositeProduct contains a list which, in its turn, contains either BaseProducts or other CompositeProducts.
- **Observer Design Pattern** – used to notify the chef whenever they are asked to cook a new order
- Design by Contract method – by stating the pre and post conditions and the invariants for the Restaurant class.
- **Serialization** – in order to correctly and efficiently store the menu items which have been added to / modified / deleted from the menu, so that they can be accessed in multiple runs of the program.
- A **Map<Order, Collection<MenuItem>>** object to store the orders and the information about each order, i.e. the items which have been ordered by the client.

In order to successfully achieve the requested goals, an **object-oriented programming design** was used, along with some of its key features: **packages, encapsulation, inheritance**, as well as the features presented above. Thus, the following things were ensured:

- Creating the necessary classes and methods to successfully **simulate the interfaces of all the three users,** all of them linked together in a main user interface which allows to switch between them.
- Implementing the **business classes** which deal with the logic behind the application and perform all the necessary tasks on the newly introduced / already available data. **The Restaurant class** holds the highest importance, as it acts as a **bridge between the GUI layer and the application backend.**
- Creating the classes and methods necessary to **write to files**, whether it is the **serialized file** (the one storing information about the menu items) or the **bill generator** (option available within the Waiter interface)

2. Problem Analysis, Modelling, Scenarios, Use Cases

The problem requires that the developer successfully designs and implements a **restaurant management system** using the given **UML class diagram**, to simulate performing operations on data collected from a **.ser file** and to illustrate the usage of the **Composite and Observer design patterns and the Design by Contract method.**

The **HashMap<>** data structure was used as well to create an efficient and valid connection between the data (between orders and their associated items).

From the perspective of the user, the program can be used by running the **.jar file** generated by the developer in the command line. The program will act accordingly and retrieve the data from the given .ser file and modify it to fit the requirements and boot up the user interface.

In terms of scenarios and use-cases, the program behaves as follows:

- When running the **.jar file**, the user can also write one argument, which will represent **the name** of the .ser file. This was done such that the user can test the functionality of the program with a file with a random name, not necessarily a fixed one; in case a file is not specified, **the default path is "restaurant.ser"**; the command is:

```
java -jar program_name.jar [file_name.ser]
```

- The user is free to choose which interface they will operate on and how to manipulate the data. Please note that **in order to save the changes that have been brought to the menu to the .ser file, the user is required to press the "Save Menu Data" button**. The program was designed this way so that the person executing the program can choose whether they want to alter the contents of the given file or not.
- Please note that the output file is the same as the input one, and whatever data was written inside the input file initially will be overwritten if the **"Save Menu Data"** button is pressed

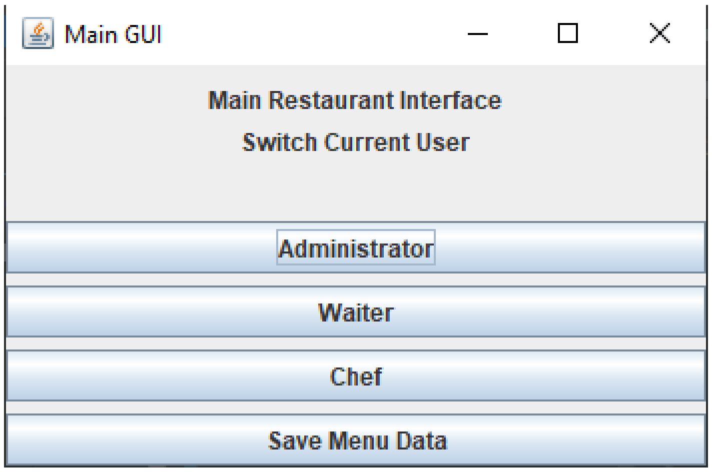
In case of editing the data, **changes do not appear instantaneously** (except for editing menu items). If, for example, the user is accessing the Administrator interface and they have opened window containing all the menu items, if an item is added / modified / deleted, then a new window of that type has to be opened (i.e. press the "View All Menu Items" button again to notice the change).

In case of the orders, **no two orders are the same**. The user (waiter) is allowed to place multiple orders for the **same table & day, and it will not be considered a duplicate**. The reason for that is that each order is identified through an **orderId** field, to ensure uniqueness. Simulating a real restaurant, multiple orders can be placed from the same table in a single day.

The bills are generated for the orders which are specified by the user in the Waiter interface, and they will be generated as text files of the form **"BillForOrderX"**, where X is the id of the order

More details on the input restrictions / limitations will be presented in the Design chapter.

In terms of use case, when running the program the user will see the following window, which will allow them to navigate easily between the interfaces:



3. Design

The Class Diagram given in the laboratory support was used as foundation for the assignment, but several additional packages and classes have been added to more efficiently distribute the workload among packages.

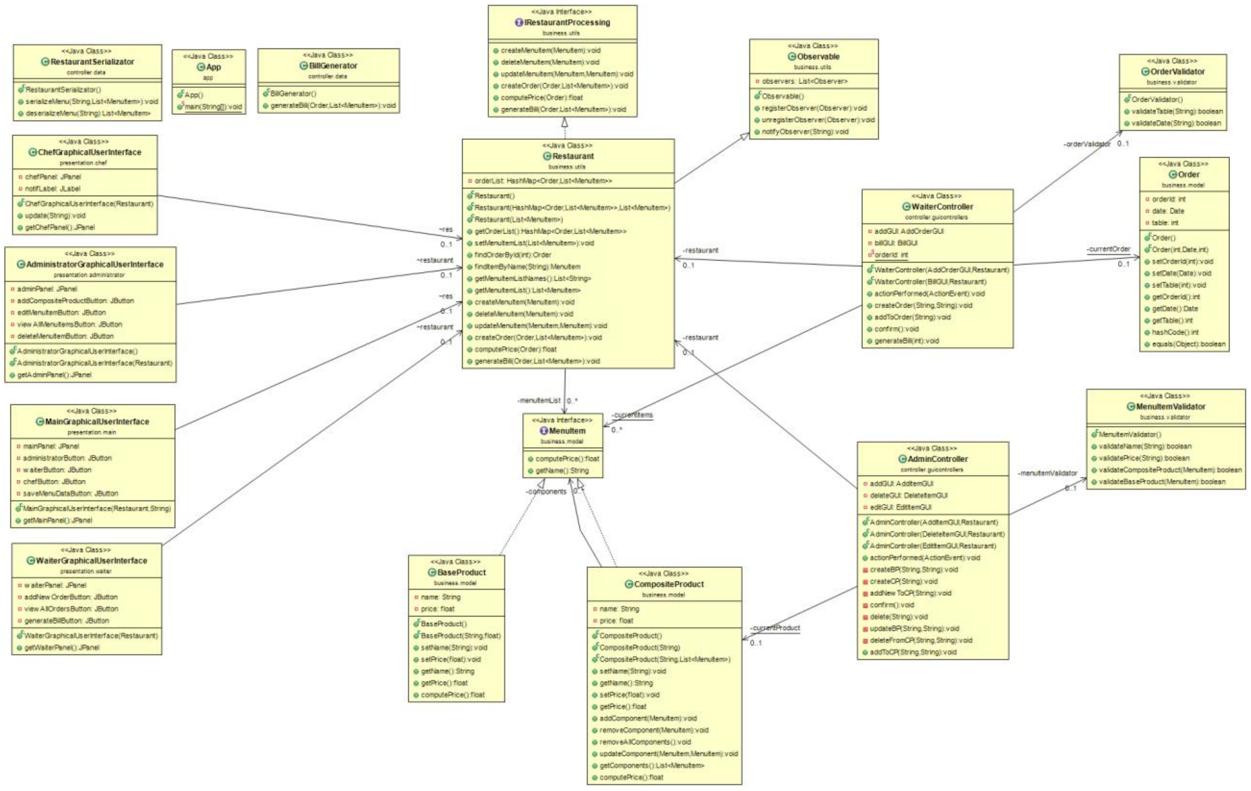
The **object-oriented programming design** was exploited through this program. Certain key object-oriented design and implementation elements were used, such as **encapsulation, inheritance** etc.

In terms of the structure of the project, as required in the objective, the following **packages and subpackages** were created:

- **App** – contains a single class which, in its turn, contains the **main method**. Boots up the application and the user interface.
- **Business** – contains three additional packages. Acts as a **connection layer** between the classes in the Controller package and the user interface (the classes from the Presentation package)
    - **Model** – contains the **Model classes** for the objects – the Products (MenuItems) and the Order. The **Composite Design Pattern was** used in order to create the MenuItems, as well as the links between them.
    - **Utils** – contains several utility classes which act as link between the Model classes, such that each order has an associated List of MenuItems. The **HashMap<> data structure** was needed in order to implement this, the Order objects acting as its keys, and the List of MenuItems as its values.
    - **Validator** – contains validator classes for the input, such that no invalid input is (e.g. name containing numbers) is registered within the Restaurant object.
- **Controller** – contains classes representing the logic of the application, that is, **the link between the Business classes and GUI**. It connects the data received as input from the GUI to the backend part of the program and with the **FileWriting / output part**.
    - **Data** – classes whose purpose is to create and populate the output files (**.ser** for the restaurant menu and the **.txt** files for the bills)
    - **GUIControllers** – the **controller classes** themselves, who call methods from the Business layer in order to manipulate the data received as input from the user.

- **Presentation** – the **GUI packages and classes**. Each type of user contains several classes, a different one for each available frame. This was done such that the GUI classes are not cluttered with several frames or pop-up instructions. The **Swing UI Designer for Intellij** was used in order to create the interfaces more easily.
    - **Administrator** – interface for the administrator user – used to manipulate the **menu data.**
    - **Chef** – interface for the chef user – a single window containing a message which changes whenever a new order is added. The **Observer Design Pattern** was used in order to implement this and to notify the Chef whenever this happens.
    - **Main** – **main interface**, which appears when the program is run. Window to all the other interfaces.
    - **Waiter** – interface for the waiter user – used to **create orders and to generate bills** for them.

The **UML diagram** below illustrates the relationships between the classes. It was generated automatically using the **ObjectAid UML Explorer for Eclipse**. Please note that some of the GUI classes were not included in order to avoid cluttering the image even more.



<<Java Class>> RestaurantSerializator controller.data
RestaurantSerializator()
serializeMenu(String,List<MenuItem>):void
deserializeMenu(String):List<MenuItem>
<<Java Class>> App app
App()
main(String[]):void
<<Java Class>> BillGenerator controller.data
BillGenerator()
generateBill(Order,List<MenuItem>):void
<<Java Interface>> IRestaurantProcessing business.utils
createMenuItem(MenuItem):void
deleteMenuItem(MenuItem):void
updateMenuItem(MenuItem,MenuItem):void
createOrder(Order,List<MenuItem>):void
computePrice(Order):float
generateBill(Order,List<MenuItem>):void
<<Java Class>> Observable business.utils
observers: List<Observer>
Observable()
registerObserver(Observer):void
unregisterObserver(Observer):void
notifyObserver(String):void
<<Java Class>> OrderValidator business.validator
OrderValidator()
validateTable(String):boolean
validateDate(String):boolean
<<Java Class>> ChefGraphicalUserInterface presentation.chef
chefPanel: JPanel
notifLabel: JLabel
ChefGraphicalUserInterface(Restaurant)
update(String):void
getChefPanel():JPanel
<<Java Class>> Restaurant business.utils
orderList: HashMap<Order,List<MenuItem>>
Restaurant()
Restaurant(HashMap<Order,List<MenuItem>>,List<MenuItem>)
Restaurant(List<MenuItem>)
getOrderList():HashMap<Order,List<MenuItem>>
setMenuItemList(List<MenuItem>):void
findOrderById(int):Order
findItemByName(String):MenuItem
getMenuItemListNames():List<String>
getMenuItemList():List<MenuItem>
createMenuItem(MenuItem):void
deleteMenuItem(MenuItem):void
updateMenuItem(MenuItem,MenuItem):void
createOrder(Order,List<MenuItem>):void
computePrice(Order):float
generateBill(Order,List<MenuItem>):void
<<Java Class>> WaiterController controller.guicontrollers
addGUI: AddOrderGUI
billGUI: BillGUI
orderId: int
WaiterController(AddOrderGUI,Restaurant)
WaiterController(BillGUI,Restaurant)
actionPerformed(ActionEvent):void
createOrder(String,String):void
addToOrder(String):void
confirm():void
generateBill(int):void
<<Java Class>> Order business.model
orderId: int
date: Date
table: int
Order()
Order(int,Date,int)
setOrderId(int):void
setDate(Date):void
setTable(int):void
getOrderId():int
getDate():Date
getTable():int
hashCode():int
equals(Object):boolean
<<Java Class>> AdministratorGraphicalUserInterface presentation.administrator
adminPanel: JPanel
addCompositeProductButton: JButton
editMenuItemButton: JButton
viewAllMenuItemsButton: JButton
deleteMenuItemButton: JButton
AdministratorGraphicalUserInterface()
AdministratorGraphicalUserInterface(Restaurant)
getAdminPanel():JPanel
<<Java Class>> MainGraphicalUserInterface presentation.main
mainPanel: JPanel
administratorButton: JButton
waiterButton: JButton
chefButton: JButton
saveMenuDataButton: JButton
MainGraphicalUserInterface(Restaurant,String)
getMainPanel():JPanel
<<Java Class>> WaiterGraphicalUserInterface presentation.waiter
waiterPanel: JPanel
addNewOrderButton: JButton
viewAllOrdersButton: JButton
generateBillButton: JButton
WaiterGraphicalUserInterface(Restaurant)
getWaiterPanel():JPanel
<<Java Interface>> MenuItem business.model
computePrice():float
getName():String
<<Java Class>> BaseProduct business.model
name: String
price: float
BaseProduct()
BaseProduct(String,float)
setName(String):void
setPrice(float):void
getName():String
getPrice():float
computePrice():float
<<Java Class>> CompositeProduct business.model
name: String
price: float
CompositeProduct()
CompositeProduct(String)
CompositeProduct(String,List<MenuItem>)
setName(String):void
getName():String
setPrice(float):void
getPrice():float
addComponent(MenuItem):void
removeComponent(MenuItem):void
removeAllComponents():void
updateComponent(MenuItem,MenuItem):void
getComponents():List<MenuItem>
computePrice():float
<<Java Class>> AdminController controller.guicontrollers
addGUI: AddItemGUI
deleteGUI: DeleteItemGUI
editGUI: EditItemGUI
AdminController(AddItemGUI,Restaurant)
AdminController(DeleteItemGUI,Restaurant)
AdminController(EditItemGUI,Restaurant)
actionPerformed(ActionEvent):void
createBP(String,String):void
createCP(String):void
addNewToCP(String):void
confirm():void
delete(String):void
updateBP(String,String):void
deleteFromCP(String,String):void
addToCP(String,String):void
<<Java Class>> MenuItemValidator business.validator
MenuItemValidator()
validateName(String):boolean
validatePrice(String):boolean
validateCompositeProduct(MenuItem):boolean
validateBaseProduct(MenuItem):boolean
~res 0..1
~restaurant 0..1
~res 0..1
~restaurant 0..1
-restaurant 0..1
-restaurant 0..1
-orderValidator 0..1
-currentOrder 0..1
-menuItemList 0..*
-currentItems 0..*
-components 0..*
-currentProduct 0..1
-menuItemValidator 0..1

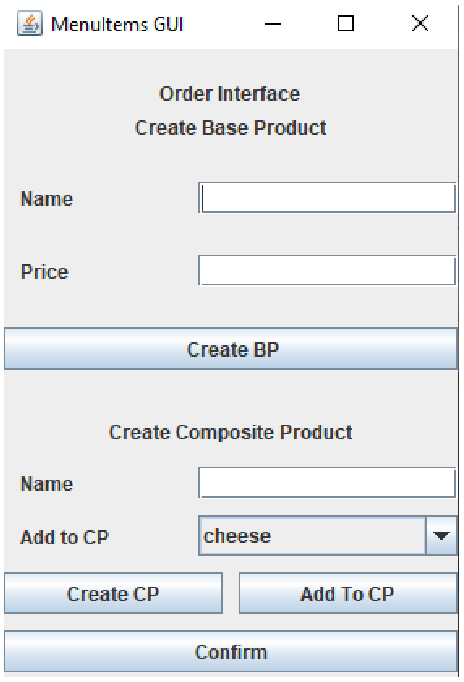
In terms of the **interfaces**, several things should be mentioned:

Each time an operation is performed, whether successfully or not, a **pop-up message** will appear to announce the success / failure. If the message does not appear, then the command was not completed (either the button was not placed properly or an unexpected error appeared).

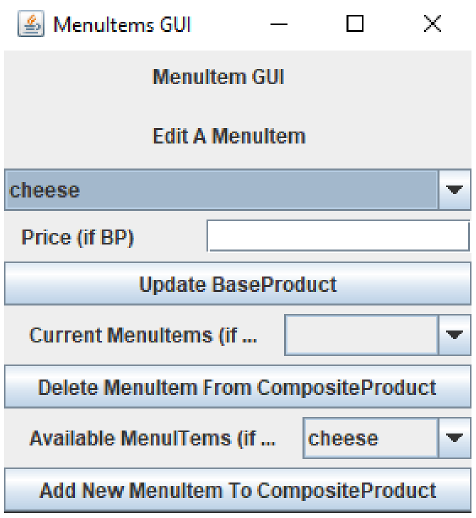
For each item / ordered added, the **validators** will check whether the input is correct or not. If it is not, no operation will be performed and a pop-up message will appear.

- **Administrator Interface**

Add Menu Item: After adding a new item, Base or Composite, in order for it to appear in the **ComboBox**, the frame must be opened again. When adding a Composite Product, the buttons must be placed in the following order: **Create CP** to create the product object, and then **AddToCP** for each extra product added. An item cannot be added to a non-existing Composite product. It will generate an exception. Likewise, a **Composite product cannot be added to the menu if it does not contain at least one product**. An error message will appear. The user has to press the **Confirm** button in order for the item to be registered. Otherwise, the data will be lost and it will not be added to the restaurant menu. **Only press Confirm when you're done adding products to the CP.**



Edit Menu Item: Each button was pressed under its respective field in order to avoid confusion and to ensure that data is not introduced incorrectly. In case the data is not correct (for example, if the user tries to change the price of a Composite product, an **error pop-up message** will appear. This window does not need to be reopened whenever a new item is picked from the main combo box. The one below will be updated automatically. **When a Composite product is left empty** (it doesn't contain any other product), **it will be deleted from the menu.**

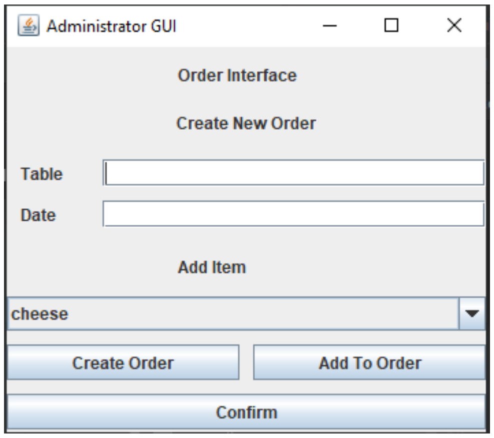

MenuItems GUI
MenuItem GUI
Edit A MenuItem
cheese
Price (if BP)
Update BaseProduct
Current MenuItems (if ...
Delete MenuItem From CompositeProduct
Available MenuITems (if ...
cheese
Add New MenuItem To CompositeProduct

Note: Though the ComboBox allows the user to add a CompositeProduct to itself, **please do not** do that as it will be viewed as an illegal operation and it will generate an exception.

Delete Menu Item: Whenever a product is deleted from the menu, **all the other products containing it will be deleted as well** and the prices will be updated.

- **Waiter Interface**

Create Order: As mentioned at point (2), orders with **duplicate tables and dates are allowed**, as they are differentiated by their ids. As for the Menu Items, the order is the following: first, press **Create Order** to create it, then **Add To Order** and finally, press **Confirm** to add it to the OrderList. **Only press confirm when you're done adding products to the order.**



Important!! Please do not forget to press the **Save Menu Data** button from the Main Interface if you wish to save the modified menu. Otherwise, none of the changes performed to it will be saved after closing the application.

4. Implementation

In order for everything to be easier to read and understand, the classes will be presented in groups, based on the package / subpackage they belong to. If the classes are too similar, they will not be discussed in detail, only the use of the package will be explained.

- **App package**
  - App class

Main class containing the main method. Its purpose is to call the methods necessary to deserialize the menu from the .ser file, if available and to boot up the main user interface.

- **Business package**
  - **Model subpackage**
    - **MenuItem interface**

Object used to store data in the restaurant. Has two methods: **computePrice() and getName(),** which must be implemented by all the classes which implement it.

    - **BaseProduct class**

Object used to store Base Products (i.e. products which **do not contain any additional items**). **Getter and setter methods.**

    - **CompositeProduct class**

Object used to store the Composite Products (i.e. products which **are made of several other items**. **Getter and setter methods.** The price is the total **sum of the prices of all the items the product contains**.

The **Composite Design pattern** was used here in the sense that a CompositeProduct can contain inside its list of products both BaseProducts and other CompositeProducts (in short, any type of MenuItem).

- **Order class**

Object class for the **orders**, containing data about the table and the date of the order. **Overrides the hashCode() and equals() method**, as it will be used as **key in the HashMap<> object**, when pairing it with a List of MenuItems.

- <u>**Utils subpackage**</u>
  - **IRestaurantProcessing interface**

Contains the **six methods** necessary to obtain, alter and generate data. Implemented by the Restaurant class. The methods involve: **creating / updating / deleting a MenuItem and creating / computing the pricve for / generating the bill for an Order.**

- **Observable class**

Class used for the **Observer Design Pattern**. As the **Java Observable** class is deprecated, I created one of my own to simulate its purpose and functionality. Contains the methods necessary to register, unregister and notify an **Observer**. Extended by the Restaurant class.

- **Restaurant class**

Class containing most of the logic behind the program. **Extends the Observable class and the IRestaurantProcessing interface** and, as a result, implements all its six methods along with several others which allow for easier access to the stored data.

Contains the **HashMap<> object** mentioned above, whose keys are **Orders** and values are **Lists of MenuItems** associated to those orders. It also contains **a List of MenuItems** used to store all the items available in the menu.

Design By Contract was used in order to specify the **pre and post conditions of each method, as well as the class invariant**. The pre and post conditions were further enhanced by using the **assert function** to verify them at the beginning and at the end of the methods, when necessary. For more details on these, as well as the functionality of the class itself, please refer to the **JavaDoc document**.

- <u>**Validator subpackage**</u>
  - **MenuItemValidator class**
  - **OrderValidator class**

Validator classes for the MenuItem and Order fields respectively. If the given parameters are not valid, an **error pop-up message** will appear, instructing the user to input valid data. Otherwise, the methods return true and the process continues as expected.

- <u>**Presentation package**</u>
  - <u>**Administrator subpackage**</u>
    - **AddItemGUI class**
    - **AdministratorGraphicaluserInterface class**
    - **DeleteItemGUI class**

        - **EditItemGUI class**
    - <u>**Chef subpackage**</u>
        - Observer interface
        - **ChefGraphicalUserInterface class**
    - <u>**Main subpackage**</u>
        - **MainGraphicalUserInterface**
    - <u>**Waiter subpackage**</u>
        - **AddOrderGUI class**
        - **BillGUI class**
        - **ViewOrdersGUI class**
        - **WaiterGraphicalUserInterface class**

Self-Explanatory **Graphical User Interface classes**. They were created using the **Swing UI Designer from Intellij**. Each frame represents a different class in order to avoid cluttering the code excessively.

Though the **ActionListeners** for the buttons are described in the **Controller classes**, the **ItemListeners for the ComboBoxes** were left within the GUI classes, as it was deemed more fit to produce and manipulate data changes and to modify the lists from the GUI classes themselves, rather than from a different package altogether.

JTables were used to show the menu and order data.

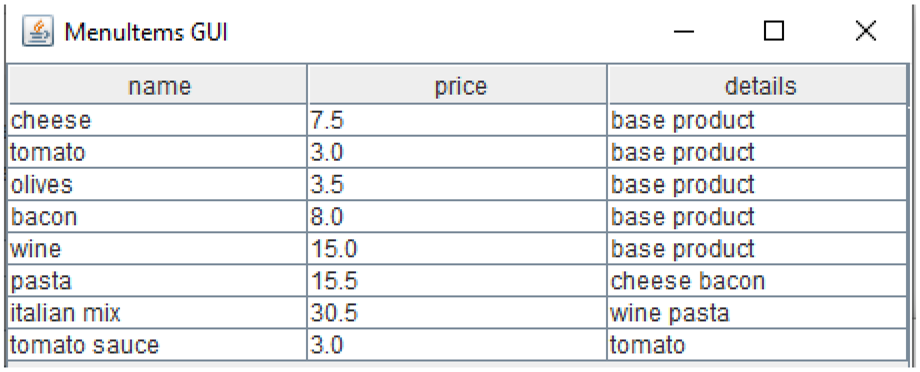
5. <u>Results</u>

The results are as expected.

Concerning the MenuItems, the following points can be made:

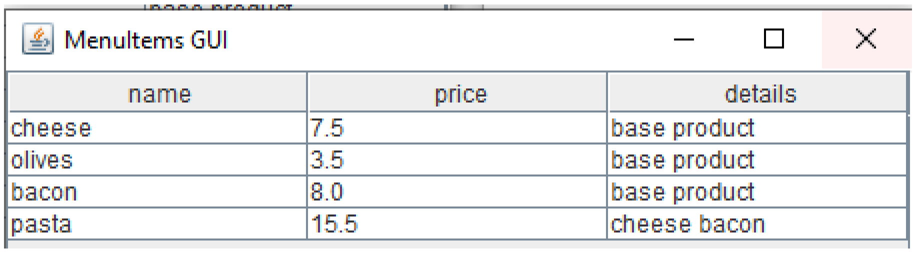
- A CompositeProduct can contain both BaseProducts and other CompositeProducts
- When a CompositeProducts does not contain anything anymore, it is deleted
- Deleting a product results in the deletion of all the products containing it, and price updates for each of them

List before deleting wine and tomato:



name	price	details
cheese	7.5	base product
tomato	3.0	base product
olives	3.5	base product
bacon	8.0	base product
wine	15.0	base product
pasta	15.5	cheese bacon
italian mix	30.5	wine pasta
tomato sauce	3.0	tomato

List after deleting wine and tomato:


MenuItems GUI

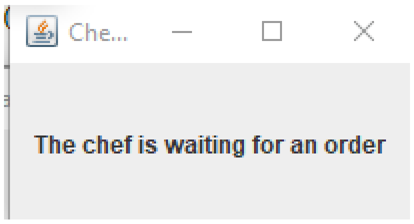
name	price	details
cheese	7.5	base product
olives	3.5	base product
bacon	8.0	base product
pasta	15.5	cheese bacon

Note: the Italian mix was deleted through cascading. tomato => pasta; pasta + wine => Italian mix.

Concerning the Orders:

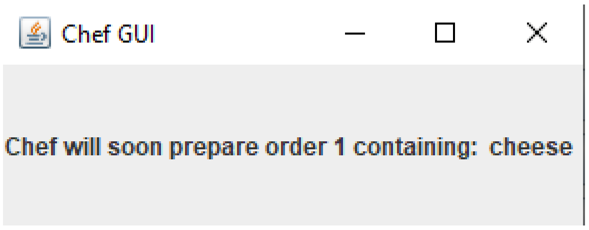
- Orders with the same table and date can be made and they will be considered as different due to their orderIds. This was done to simulate real life situations as accurately as possible, as more than one order can be ordered from the same table on the same day.
- When a new Order is added, the chef is notified and, if the Chef GUI is open, the change in text can be noticed (going from "waiting for order" to "will soon prepare order").
- The Total shown on the bill text file represents the total of all the products ordered.

Chef before adding the 1st order:


Che...

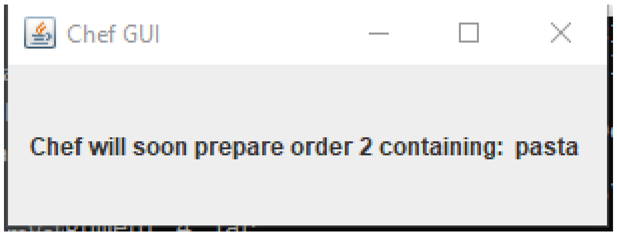
The chef is waiting for an order

Chef after adding the 1st order:

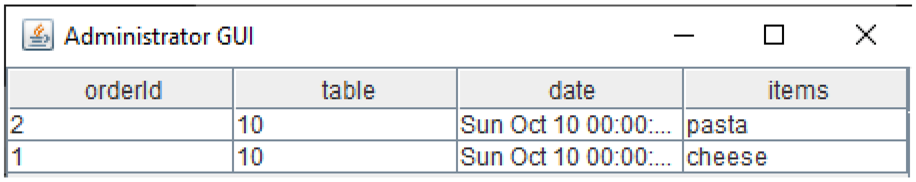

Chef GUI

Chef will soon prepare order 1 containing: cheese

List & chef after adding the 2nd order with the same data:


Chef GUI

Chef will soon prepare order 2 containing: pasta


Administrator GUI

orderId	table	date	items
2	10	Sun Oct 10 00:00:...	pasta
1	10	Sun Oct 10 00:00:...	cheese

6. Conclusions

In conclusion, this assignment can be viewed as i**ntroduction** to several objected oriented programming features and patterns, to be exact: **Serialization, Design by Contract, Observer Design Pattern, Composite Design Pattern.**

Object-oriented programming was exploited through the use of **packages, classes, encapsulation and inheritance.**

Serialization was used when generating the **restaurant.ser** file, as well as when retrieving data from it.

Design by Contract was discovered within the **Restaurant class** by adding **pre/post conditions, invariants and assertions** for each method.

The Observer Design Pattern was used to **notify the Chef** whenever a new order has been added to the list.

Finally, the **Composite Design Pattern** was used when creating the objects for **the MenuItems, BaseProduct and CompositeProduct**, the latter of which contains a list of MenuItems of any type.

As further enhancements, the user interface could be designed with more care, to make it more aesthetically pleasing (and, possibly, more user-friendly). It could also be possible to add several more actions to the chef, so that the user interface can become more interactive, by adding a queue of orders to be served, which gradually changes throughout the execution of the program.

7. Bibliography

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

https://www.baeldung.com/java-serialization

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

https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html

https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#tag

https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html

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