**FUNDAMENTAL PROGRAMMING TECHNIQUES**

**ASSIGNMENT 4**

**RESTAURANT MANAGEMENT SYSTEM**

Anghel Dan-Marian

Group 30422

Professor: Diana Balc

**Table of contents:**

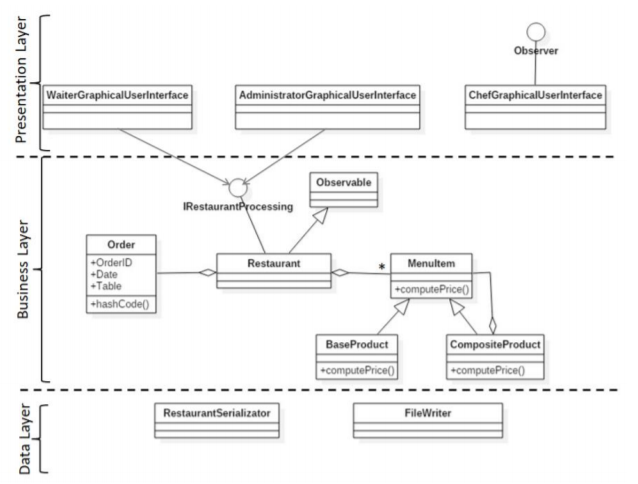
1. **Objectives**
2. **Analysis**
3. **Development**
4. **Implementation**
5. **Results**
6. **Conclusion**
7. **Bibliography**

**1. Objectives**

**Primary objectives:**

Consider implementing 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 that is ordered through a waiter.

Consider the system of classes in the diagram below.



To simplify the application, you may assume that the system is used by only one administrator, one waiter and one chef, and there is no need of a login process.

Solve the following:

1) **Define the interface** IRestaurantProcessing containing the main operations that can be executed by the waiter/administrator, as follows:

• Administrator: create new menu item, delete menu item, edit menu item

• Waiter: create new order; compute price for an order; generate bill in .txt format.

2) **Define and implement the classes** from the class diagram shown above:

• Use the Composite Design Pattern for defining the classes MenuItem, BaseProduct and CompositeProduct .

• Use the Observer Design Pattern to notify the chef each time a new order

containing a composite product is added.

3) **Implement the class Restaurant** using a predefined JCF collection that is based on a hashtable data structure. The hashtable key will be generated based on the class Order, which can have associated several MenuItems. Use JTable to display Restaurant related information.

• Define a structure of type Map<Order, Collection<MenuItem>> for storing the order related information in the Restaurant class. The key of the Map will be formed of objects of type Order, for which the hashCode() method will be overwritten to compute the hash value within the Map from the attributes of the Order (OrderID, date, etc.).

• Define an appropriate collection consisting of MenuItem objects to store the menu of the restaurant.

• Define a method of type “well formed” for the class Restaurant.

• Implement the class Restaurant using Design by Contract method (involving pre, post conditions, invariants, and assertions).

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

**Secondary objectives:**

* **Use-cases** (Chapter 2):  a **use case** is a list of actions or event steps typically defining the interactions between a role and a system to achieve a goal, usually represented through a flow-chart or an UML diagram.
* **Diagrams** (Chapter 3): they provide a mock-up for the project so that the developer can more easily grasp the concept and the small details of it.
* **Data structures** (Chapter 3): the types of data structures used in making the application.
* **The classes used** (Chapter 3): the project has been split in smaller classes that can accomplish certain tasks and provide relevant results through reflection.
* **Packages** (Chapter 3): the project is split in multiple packages in order to accomplish a layered architecture based on the model: data access layer, business layer, model and presentation.
* **Graphical User Interface** (Chapter 3): the interface that the user will interact with, in order to properly use the application.
* **Implementation** (Chapter 4): each class will be explained for a better understanding of the functioning of the program.
* **Results** (Chapter 5): the results are given in the form of PDF files for bills and reports and the SQL dump file, but will be talked over again here nevertheless.

**2. Problem Analysis**

* **General Overview**

This application should resemble a restaurant with 3 main components, namely the Administrator, who is able to create new menu items, to edit the price of the existing ones and to delete menu items, the Waiter who is able to create an order taken from the clients at a certain table, he is able to create the order and deliver it to the

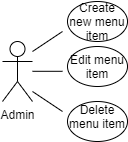
chef in order to be cooked/ prepared, to compute the total price of the order and to generate a bill in txt format and lastly, we have the chef who’s only job is to get notified by the waiter and to cook the food.

* **Input and output**

The input is given by the user from the graphical interface in the form of buttons that have certain commands attached to them. Depending on who the user is he/she has access to certain actions that other users may not be able to perform. If the admin is using the program, then the operations available are to create a new menu item or to update/delete an already existing one. If the waiter is using the program, then he/she may create a new order, generate a bill or find the total cost of an order. Lastly, if the chef is the user, then whenever a new order is created, a pop-up window will appear to notify the chef about what he/she has to prepare.

The output can be a bill in the form of a text file, containing information about the order (i.e. id, date, table, items ordered, amount to be payed). It can also resemble information in the form of pop-up windows that are meant to inform the user about the details of the operations that he performed or is about to perform.

* **Use cases**



(\* Use-case diagram for the administrator user)

* **Create New Menu Item**

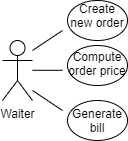
|  |  |
| --- | --- |
| Use Case | Create new menu item |
| Actor | Administrator |
| Basic Flow | This can be achieved by selecting the tab corresponding to the admin, filling in the fields with the information about the new menu item and pressing the button “new” from the admin GUI tab. |
| Alternate Flow | In case that the operation fails, the user will be prompted with an error message. |

* **Edit Menu Item**

|  |  |
| --- | --- |
| Use Case | Edit menu item |
| Actor | Administrator |
| Basic Flow | This can be achieved by selecting the tab corresponding to the admin, filling in the fields with the information about the updated menu item and pressing the button “update” from the admin GUI tab. |
| Alternate Flow | In case that the operation fails, the user will be prompted with an error message. |

* **Delete Menu Item**

|  |  |
| --- | --- |
| Use Case | Delete menu item |
| Actor | Administrator |
| Basic Flow | This can be achieved by selecting the tab corresponding to the admin, finding the menu item desired and then pressing the button “delete” from the admin GUI tab. |
| Alternate Flow | In case that the operation fails, the user will be prompted with an error message. |



(\* Use-case diagram for the waiter user)

* **Create New Order**

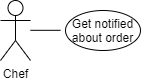
|  |  |
| --- | --- |
| Use Case | Create new order |
| Actor | Waiter |
| Basic Flow | This can be achieved by selecting the tab corresponding to the waiter, filling in the fields with the information about the new order and pressing the button “new” from the admin GUI tab. |
| Alternate Flow | In case that the operation fails, the user will be prompted with an error message. |

* **Compute Order Price**

|  |  |
| --- | --- |
| Use Case | Compute order price |
| Actor | Waiter |
| Basic Flow | This operation is performed automatically whenever the waiter creates a new order. After hitting the “new” button, a pop-up window will inform the waiter about the total cost of the order. |
| Alternate Flow | In case that the operation fails, the user will be prompted with an error message. |

* **Generate Bill**

|  |  |
| --- | --- |
| Use Case | Generate bill |
| Actor | Waiter |
| Basic Flow | This can be achieved by selecting the tab corresponding to the waiter and pressing the button “generate bill” from the admin GUI tab. A text file will be created containing the information about the order. |
| Alternate Flow | In case that the operation fails, the user will be prompted with an error message. |



(\* Use-case diagram for the chef user)

* **Get Notified**

|  |  |
| --- | --- |
| Use Case | Get notified |
| Actor | Chef |
| Basic Flow | Whenever a new order is placed by the waiter user, the chef will be notified with a pop-up window that will inform him/her that a new order was created and what the ordered items are. |
| Alternate Flow | In case that the operation fails, the user will be prompted with an error message. |

**3. Development**

The project follows the model-view-controller design pattern, as it can be seen down below. MVC is a popular way of organizing your code. The big idea behind MVC is that each section of your code has a purpose, and those purposes are different. Some of your code holds the data of your app, some of your code makes your app look nice, and some of your code controls how your app functions.

MVC is a way to organize your code’s core functions into their own, neatly organized boxes. This makes thinking about your app, revisiting your app, and sharing your app with others much easier and cleaner.

Model: Model code typically reflects real-world things. This code can hold raw data, or it will define the essential components of your app. For instance, if you were building a To-do app, the model code would define what a “task” is and what a “list” is – since those are the main components of a to-do app.

View: View code is made up of all the functions that directly interact with the user. This is the code that makes your app look nice, and otherwise defines how your user sees and interacts with it.

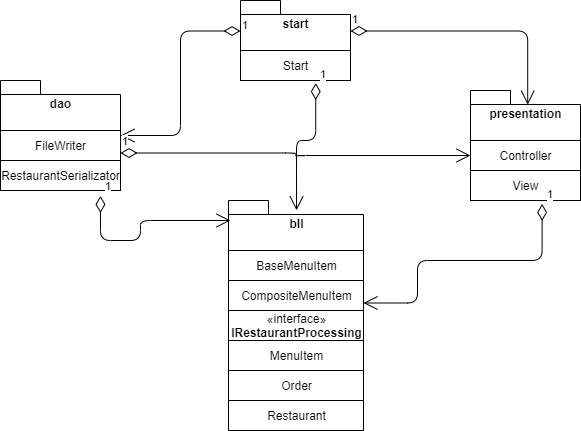
Controller: Controller code acts as a liaison between the Model and the View, receiving user input and deciding what to do with it. It’s the brains of the application, and ties together the model and the view.

Other than the usual lists, the data structure of choice for this project was the linked hash map which was used to store the orders and their respective list of menu items.

The interface IRestaurantProcessing is implemented by the restaurant class along with its methods. This interface declares the methods necessary for the implementation of the operation of the administrator and waiter applications.

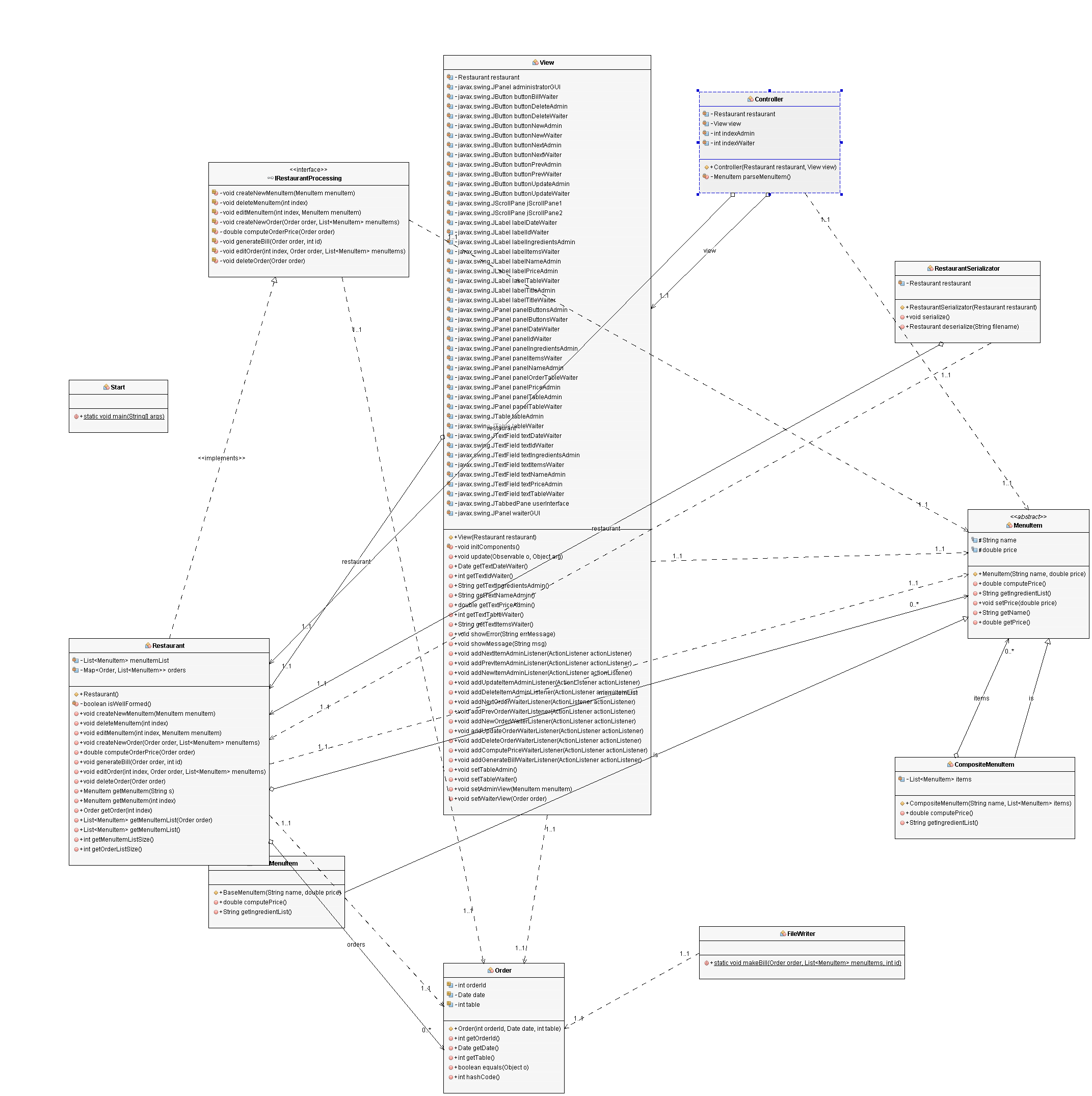
We also have the abstract class MenuItem that is inherited by the classes BaseMenuItem and CompositeMenuItem. These are made trough the composite design pattern model.

This is the package diagram of the project where it can be observed the MVC design pattern (the bll package is equivalent to the model).



(\* Package diagram of the project)

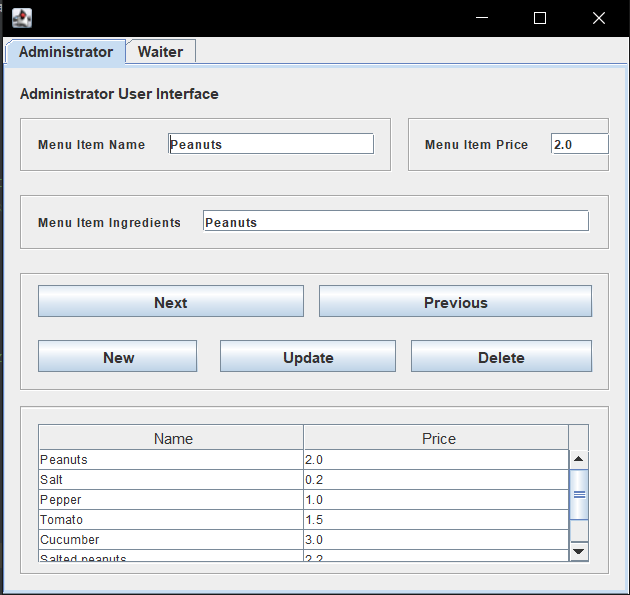
This is the class diagram of the project, where it can be observed the layered structure.



(\* Class diagram of the project)

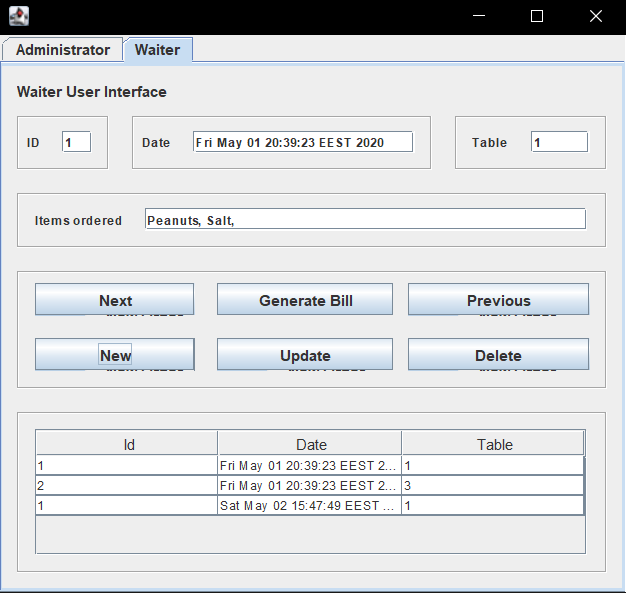
This image has been generated using the easyUML plugin of the NetBeans IDE.

A better-quality representation can be observed at the link: <https://imgur.com/a/WZceBYn>

These are a few screenshots of the application in different moments of the operations.****

(\* User interface for the administrator)

Both the admin and the waiter have the possibility of stepping trough the list of menu items/orders using the buttons “next” and “previous”.

****

(\* User interface for the waiter)

**4. Implementation**

(\* describing each package and their classes)

**4.1 start**

* **Start** – This is the main class of the project; it initializes all components of the MVC design.

Furthermore, it is also responsible of deserializing the .ser file in order to get the data for the restaurant object.

**4.2 presentation**

* **View** – This is the graphical user interface. It is uses a tabbed panel in order to display both the admin interface and the waiter interface in the same window.

It has methods for adding listeners to the various graphic elements, getters, setters for updating the state of the window.

* **Controller** – This class creates the actual listener classes for the various operations implemented by the restaurant object.

It also has a method named parseMenuItem() that takes the user input from the text boxes of the interface and creates a menu item from those strings.

**4.3 dao**

* **FileWriter** – This class creates text files for the “generate bill” operation. It creates a bill with the corresponding information about the order (i.e. id, date, table, items ordered).
* **RestaurantSerializator** – This class is the model class for the “product” data base table. It contains the same fields as the table, a few constructors, the overridden to string method and the corresponding setters and getters.

**4.4 bll**

* **IRestaurantProcessing** – This interface declares the necessary methods for the minimal functionality of the application. It will be implemented by the restaurant class along with its methods.
* **Restaurant** – This class implements the interface IRestaurantProcessing along with some other methods needed for the functionality of the program. It provides a method “wellFormed()” as an invariant for the class. It also has certain getters for accessing different types of data.
* **MenuItem** – This is an abstract class, made by the composite design pattern style. This represents the component object. It is extended by the classes BaseMenuItem and CompositeMenuItem. It has the abstract methods “computePrice()”, that calculates the price of the item, and “getItemList()” that return the list of items.
* **BaseMenuItem** – This class extends the abstract class MenuItem, implementing its abtract methods. It is the leaf object from the design pattern, due to its simplicity. It has only one item that is itself.
* **CompositeMenuItem** – This class extends the abstract class MenuItem, implementing its abtract methods It is the composite object from the

design pattern, due to the fact that it is complex. It has a list of menu items and methods that need to be implemented are more complex.

* **Order** – This represents a real-life order having an id, date and table. It is mainly used by the waiter user. It overrides the equals() and hashCode() methods due to the fact that it is used as the key for a map.

**5. Results**

The results are given in the form of bills that are text files. The bills specify information about the order, i.e. the id of the order, the date of the order, the table that the people sat at, the list of items ordered, alongside with their respective costs and lastly the total cost of the order. Also, there are pop-up windows to notify the users about the current status of the application, i.e. when a new order is placed, or the cost of that order, etc.

**6. Conclusions**

The restaurant management system project was a great remembering exercise to strengthen the knowledge about the MVC design model and to develop new skills, such as the composite design pattern. Furthermore, this project managed to get me involved in learning new and exciting programming techniques, as well as some nice tricks.

There is great improvement potential to this project in the form of adding new operations for each user. There is even room to add some different type of functionality like, sorting the data .

**7. Bibliography**

* Class diagram of the project: <https://imgur.com/a/WZceBYn>
* EasyUML plugin for generating class diagrams inside NetBeans: <http://plugins.netbeans.org/plugin/55435/easyuml>
* Easy to use website to generate UML diagrams, flowcharts, etc.: <https://app.diagrams.net/>
* Model view controller design pattern description <https://www.codecademy.com/articles/mvc>