RUSU CARLA MARIA
GROUP 30421

Restaurant management system

RUSU CARLA MARIA
GROUP 30421

Objective

"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 ordered through a waiter.[…] 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."

OBJECTIVE	**ITEM**	**DETAILS**
MAIN OBJECTIVE	Restaurant Management System	The application should simulate the management system of a restaurant; The administrator should be able to create, edit or delete menu items. The waiter should be able to create a new order, compute the price and generate a bill in .txt format. The chef should use the Observer Design Pattern to be notified of each change.
SECONDARY OBJECTIVES	Composite Design Pattern	Necessary for defining and implementing the classes MenuItem, BaseProduct and CompositeProduct;
	Data	Contains the utility classes that take care of serialization and file writing.
	Business layer	The business layer is the link to the GUI. It contains the logic of the application by allowing certain CRUD events to make changes to the restaurant's menu or order list.
	Presentation classes	These contain the GUI: namely the administrator, waiter and chef GUIs. They are the direct link between the user and the business layer.
	Design by Contract	This knowledge is necessary to include pre, post conditions, invariants and assertions in the Restaurant class, such that no illegal operations take place (eg.: creating a menu item whose price is 0).
	Serialization	The menu items and the orders are loaded from/into a .ser file using serialization.

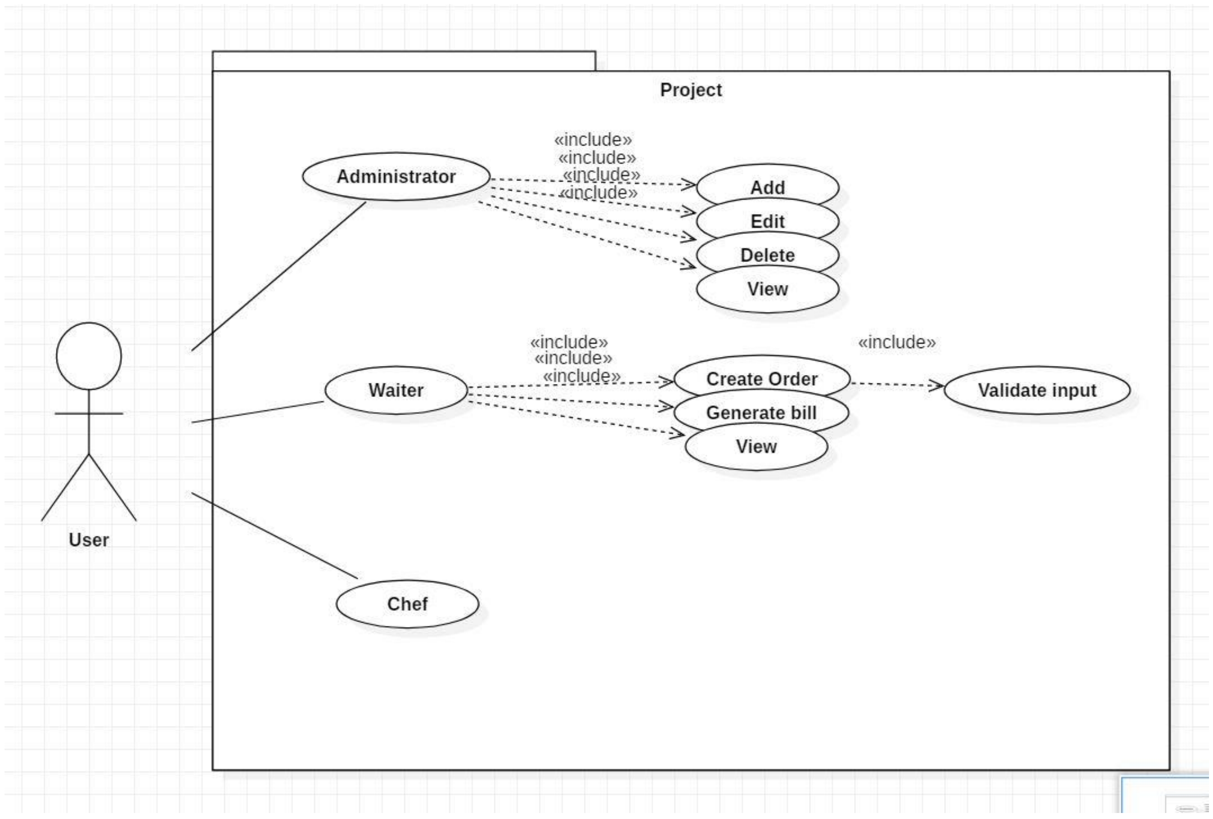
Problem analysis

- Choose the desired ‘window’ to interact with (Administrator, Waiter and Chef);
- Input the necessary data; depending on the tab, the fields will differ (NO INPUT NECESSARY FOR CHEF WINDOW)
  a) IDs
  b) Product names
  c) Ingredients
  d) Prices
  e) Other fields

  No input validation was implemented except for the date format when the add button is pressed. The user is presumed to be knowledgeable of the correct input formats.

- The administrator window:
  a) For adding a base product: product name and price must be filled
  b) For adding a composite product: product name, ingredients(only existing ones, separated by “,”)
  c) For editing an existing item the old name and the new name must be provided, as well as the price and (if composite) the ingredients
  d) For deleting an item the product name must be given
  e) The view button refreshes the menu table
- The waiter window:
  a) The view button refreshes the orders table
  b) For creating a new order the date must be given as well as the products separated by “,”
  c) The generate bill button generates a bill for the given order id;
- The chef window
  a) An option pane appears each time a new order is placed

RUSU CARLA MARIA
GROUP 30421



- Main success scenario:
    1. User selects the desired window, introduces the input parameters and clicks one of the buttons
    2. The application validates, parses and converts the input in some cases
    3. The result is either displayed as a .txt bill (place order) or can be viewed by refreshing the table (click 'View'); the chef window offers the result in a dialog pane
- Alternative (error) sequences:

    a) Incorrect date field in the waiter window

    1. User introduces data and selects create order on client tab

    2. The app tries to validate the input and fails

    3. An error message dialog box is issued stating that the input is not a valid date format;

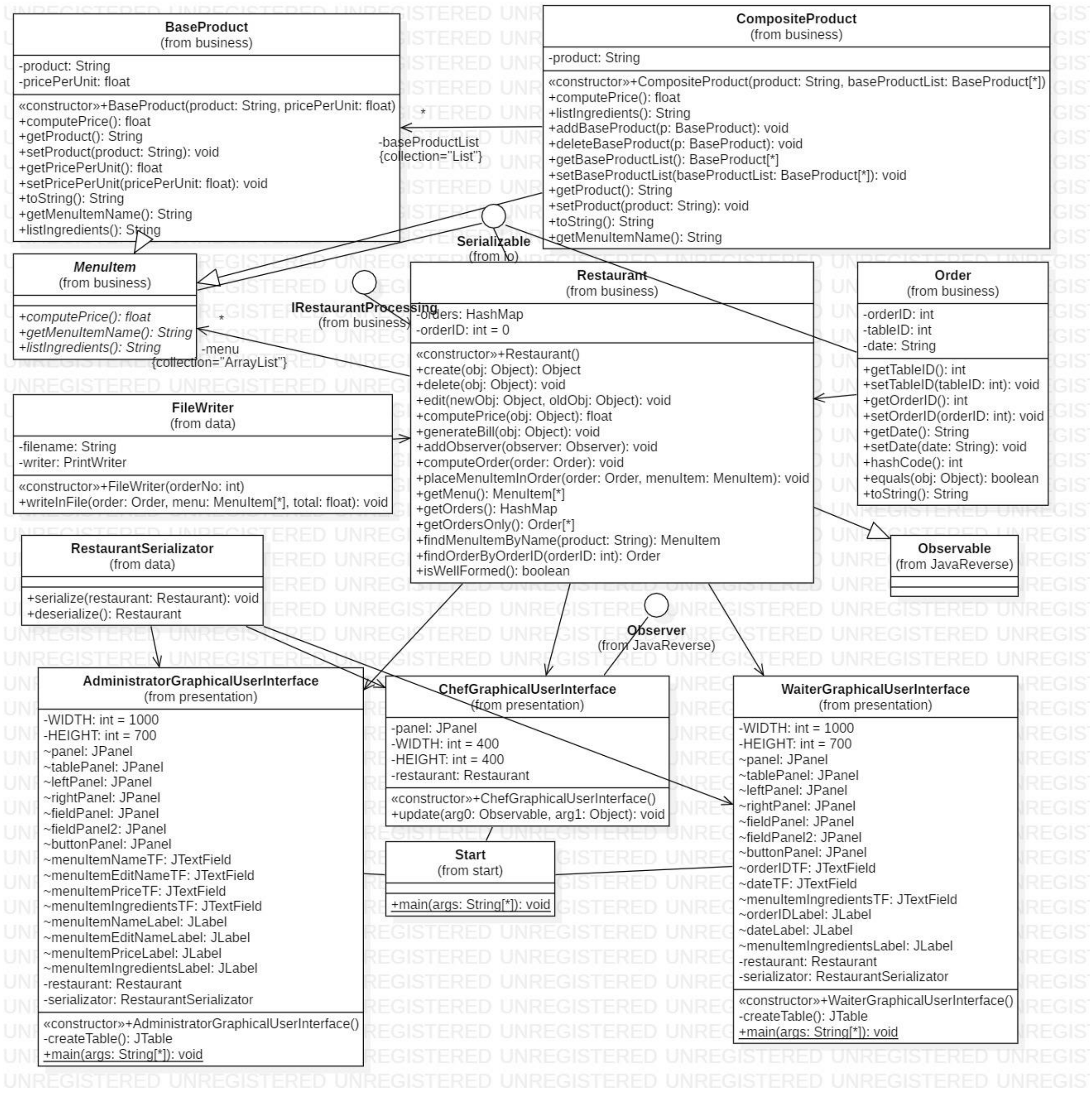
    4.The scenario returns to step 1

RUSU CARLA MARIA
GROUP 30421

Design

The application is made up of four packages and is structured on the MVC architecture: business (contains MenuItem which is composed of either BaseItem or CompositeItem, IRestaurantProcessing interface implemented by Restaurant, Order), presentation (graphical user interace classes- waiter, administrator, chef), data(contains the utility classes for serializations and file access), start(main Start class. Initialises the project). The project thus uses both an MVC architecture and a layered one (data, business and presentation);

The UML diagram displayed below offers a complete understanding of the relationship between the classes. The attributes and the methods were not included so as to provide a less cluttered, easier to understand diagram;



BaseProduct (from business)
-product: String
-pricePerUnit: float
«constructor»+BaseProduct(product: String, pricePerUnit: float)
+computePrice(): float
+getProduct(): String
+setProduct(product: String): void
+getPricePerUnit(): float
+setPricePerUnit(pricePerUnit: float): void
+toString(): String
+getMenuItemName(): String
+listIngredients(): String
CompositeProduct (from business)
-product: String
«constructor»+CompositeProduct(product: String, baseProductList: BaseProduct[*])
+computePrice(): float
+listIngredients(): String
+addBaseProduct(p: BaseProduct): void
+deleteBaseProduct(p: BaseProduct): void
+getBaseProductList(): BaseProduct[*]
+setBaseProductList(baseProductList: BaseProduct[*]): void
+getProduct(): String
+setProduct(product: String): void
+toString(): String
+getMenuItemName(): String
*
-baseProductList {collection="List"}
Serializable (from io)
MenuItem (from business)
+computePrice(): float
+getMenuItemName(): String
+listIngredients(): String
*
-menu {collection="ArrayList"}
IRestaurantProcessing (from business)
Restaurant (from business)
-orders: HashMap
-orderID: int = 0
«constructor»+Restaurant()
+create(obj: Object): Object
+delete(obj: Object): void
+edit(newObj: Object, oldObj: Object): void
+computePrice(obj: Object): float
+generateBill(obj: Object): void
+addObserver(observer: Observer): void
+computeOrder(order: Order): void
+placeMenuItemInOrder(order: Order, menuItem: MenuItem): void
+getMenu(): MenuItem[*]
+getOrders(): HashMap
+getOrdersOnly(): Order[*]
+findMenuItemByName(product: String): MenuItem
+findOrderByOrderID(orderID: int): Order
+isWellFormed(): boolean
Order (from business)
-orderID: int
-tableID: int
-date: String
+getTableID(): int
+setTableID(tableID: int): void
+getOrderID(): int
+setOrderID(orderID: int): void
+getDate(): String
+setDate(date: String): void
+hashCode(): int
+equals(obj: Object): boolean
+toString(): String
FileWriter (from data)
-filename: String
-writer: PrintWriter
«constructor»+FileWriter(orderNo: int)
+writeInFile(order: Order, menu: MenuItem[*], total: float): void
Observable (from JavaReverse)
RestaurantSerializator (from data)
+serialize(restaurant: Restaurant): void
+deserialize(): Restaurant
Observer (from JavaReverse)
AdministratorGraphicalUserInterface (from presentation)
-WIDTH: int = 1000
-HEIGHT: int = 700
~panel: JPanel
~tablePanel: JPanel
~leftPanel: JPanel
~rightPanel: JPanel
~fieldPanel: JPanel
~fieldPanel2: JPanel
~buttonPanel: JPanel
~menuItemNameTF: JTextField
~menuItemEditNameTF: JTextField
~menuItemPriceTF: JTextField
~menuItemIngredientsTF: JTextField
~menuItemNameLabel: JLabel
~menuItemEditNameLabel: JLabel
~menuItemPriceLabel: JLabel
~menuItemIngredientsLabel: JLabel
-restaurant: Restaurant
-serializator: RestaurantSerializator
«constructor»+AdministratorGraphicalUserInterface()
-createTable(): JTable
+main(args: String[*]): void
ChefGraphicalUserInterface (from presentation)
-panel: JPanel
-WIDTH: int = 400
-HEIGHT: int = 400
-restaurant: Restaurant
«constructor»+ChefGraphicalUserInterface()
+update(arg0: Observable, arg1: Object): void
Start (from start)
+main(args: String[*]): void
WaiterGraphicalUserInterface (from presentation)
-WIDTH: int = 1000
-HEIGHT: int = 700
~panel: JPanel
~tablePanel: JPanel
~leftPanel: JPanel
~rightPanel: JPanel
~fieldPanel: JPanel
~fieldPanel2: JPanel
~buttonPanel: JPanel
~orderIDTF: JTextField
~dateTF: JTextField
~menuItemIngredientsTF: JTextField
~orderIDLabel: JLabel
~dateLabel: JLabel
~menuItemIngredientsLabel: JLabel
-restaurant: Restaurant
-serializator: RestaurantSerializator
«constructor»+WaiterGraphicalUserInterface()
-createTable(): JTable
+main(args: String[*]): void

Implementation

A detailed explanation of the implemented classes and algorithms is given below;

- ***MenuItem, BaseProduct, CompositeProduct classes***
  all fields have getters and setters; method toString() is overridden; abstract class MenuItem contains abstract methods implemented through inheritance by the other two classes mentioned; BaseProduct contains the name of the product and the price; CompositeProduct contains a list of base products and the name of the resulting product; MenuItem can be either a simple (base) product or a complex (composite) one;
- ***Order class***
  The class contains an order id, table id and a date (int, int and String); all fields have getters and setters; method toString() is overridden; hashCode() and equals are also overridden such that the objects of this class can be used as keys in HashMaps/ HashTables;
- ***IRestaurantProcessing interface***
  Defines the CRUD methods for the restaurant (create- used either for an order or a menu item, delete, edit, computePrice and generateBill);
- ***Restaurant class***
  Has a list of menu items representing the menu, a hashmap in which the orders are the keys and the lists of menu items are the order contents and an orderID used for creating new orders;
  Can create a new menu item or order, depending on the parameter (a composite or base product or a String representing the date); can delete or edit a menu item as well as compute an order's price and generate a bill; it follows the Observable-Observer Design Pattern as it notifies the chef class of any new order; it also loads its data to and from files through serialization; the ability to place items in or
- ***Administrator, Waiter and Chef GUI classes***

  The ChefGraphicalUserInterface provides an option pane each time a new order has been placed, providing as well the order information; The Observer design pattern was used to provide a "real time " notice system; in order to do that, the update method has been overridden;

  The AdministratorGraphicalUserInterface provides text fields and labels for the input and buttons for the possible actions; it uses the restaurant class, serialization to update the stored data;

  The WaiterGraphicalUserInterface provides provides text fields and labels for the input and buttons for the possible actions; it uses the restaurant class, serialization to update the stored data and initialises the chef GUI to notify it of new orders;

- ***FileWriter Class***

  Is used to generate a bill; Has a constructor which takes an int as a parameter (the order's id) in order to set the file's name and extension (.txt); initialises the PrintWriter object; writeInFile() method takes as parameters an order, the items that are included in said order and the price, then it fills the bill with necessary information (the order's information, the list of products ordered, the price of each product and the final price);

- ***RestaurantSerializator class***

  Contains two methods: serialize and deserialize; serialize is used to store the data contained by the restaurant class in a .ser file and deserialize is used to load said data into a Restaurant object;

Results

The results for generating a bill for orders with id 0 and client with id 1 are given below as examples:

```
order_information: orderID=0, tableID=1, date=03/06/2019


list_of_products:


product=Cartofi prajiti, price=7.0


product=Omleta, ingredients={Oua, Ciuperci, Mozarella}, price=15.5


product=Antricot de vita, ingredients={Carne de vita, Cartofi prajiti, Busuioc}, price=36.0


total_amount_to_pay: 58.5
```

```
order_information: orderID=1, tableID=12, date=01/06/2019

list_of_products:

product=Salata caprese, ingredients={Rosii cherry, Mozarella, Busuioc, Otet balsamic},
price=16.5

product=Omleta, ingredients={Oua, Ciuperci, Mozarella}, price=15.5

total_amount_to_pay: 32.0
```

Conclusions

To conclude, the application meets the requirements: has an OOP design, is organized in packages, its methods are under 30 lines and it respects Java naming conventions. An intuitive graphical user interface on which CRUD operations can be done is provided, as well as the ability to create bills and to "observe" the orders through the Chef's GUI.

While designing this application I have further developed my OOP knowledge by acquiring some insight on Composite Design Pattern, Observer Design Pattern, Design by Contract and serialization. For the first time, I had the chance to learn about the benefits of serialization and the Observer Design Pattern, and how they can provide a good storage means and producer-client notification system. I have understood the utility of using pre, post conditions ,invariants and assertions.

Bibliography

- http://users.utcluj.ro/~igiosan/teaching_poo.html
- http://users.utcluj.ro/~cviorica/PT2019/
- http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/HW4_Tema4/HW4_Tutorial_Hashing_In_Java.pdf
- https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html
- https://www.tutorialspoint.com/java/java_polymorphism.htm
- http://www.tutorialspoint.com/java/java_serialization.htm
- https://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html