MINISTERUL EDUCAȚIEI NAȚIONALE
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Restaurant Management System

Fundamental Programming Techniques

Horvath Andrea Anett

Faculty of Automation and Computer Science
Computer Science Department
2'nd year, Group 30424

Contents

I. Objectives of the proposed laboratory work

The main objective of the proposed laboratory work was to develop a Java application that allows the management of a restaurant by an administrator, together with the management of the orders by a waiter and their processing by a chef.

As secondary objectives, the following can be considered:

- developing the application in accordance with the object oriented programming principles
- designing the system according to real life needs
- developing appropriate graphic user interfaces (GUI) for an accessible use of the program
- getting familiar with the Composite Design Pattern
- using the Observer Design Pattern
- learning how to use hash tables
- using Design by Contract
- working with Serialization
- drawing the UML diagrams and writing the code based on them (forward engineering)
- using appropriate data structures for the desired approach
- creating .txt files
- determining all the use-cases and assuring that the program takes into account all the possibilities
- getting familiar with JavaDoc files and custom tags
- running the project as a .jar file

II. Analysis of the laboratory work, modeling, scenarios and use-cases

Analysis

The Restaurant Management System will be implemented to handle the operations that managing a restaurant involves. The operations which are to be executed are distributed to three categories of users: administrator, waiter and chef.

The administrator can :

- add a base or a composite product;
- delete a menu item;
- modify a menu item;
- view menu items.

The waiter can:

- create orders for a table;
- create a bill for an order;
- visualize the previous orders.

The chef notified each time it must cook food that was ordered through a waiter.

The application will be built in accordance with the given diagram. Thus, the architecture will consist in three layers: the presentation layer which will handle the interfaces, the business layer which will be in charge for the logic of the application and the data layer which will contain the serialization and the file writing parts.

Modeling

The application will run as a .jar file getting as argument a .ser file containing the previous state of the application. The program will open an interface which allows choosing the user. Since the application will be used by only one administrator, one waiter and one chef, there is no need of a login process, but the user will still be chosen separately from usage simplicity reasons. For each type of user, a different interface will open presenting the operations that are allowed for that user. Any of the operations will also open another interface in which the operation can be performed. The input of the GUI will be validated and processed. The information provided by the interface are sent to the business layer which implements the operations. There is a permanent exchange of information between the business and the presentation layer. The presentations needs data from the business for the "view" operations and for validating the input and displaying messages in order to make the application more user friendly.

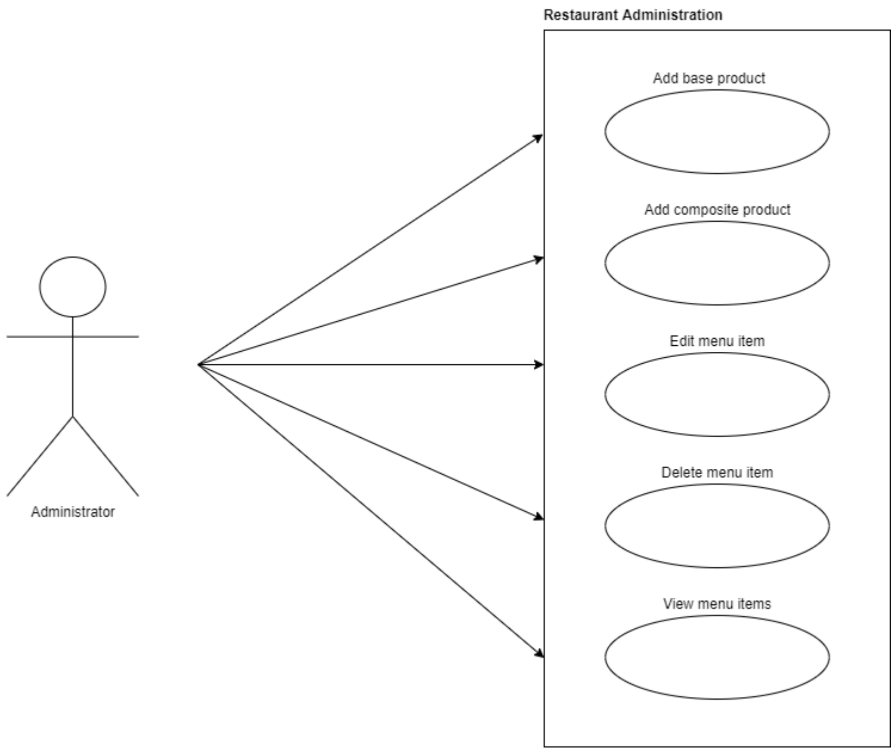
The model of the menu items is constructed using the Composite Design Pattern. Thus, the chosen approach was to have an abstract class for the menu item and two other classes extending the *MenuItem* class: *BaseProduct* and *CompositeProduct*. A base product consists in a singular menu item (i.e. fries), while a composite product will have in its structure a set of menu items which can be either base products or other composite products (i.e. fries + chicken, water). For representing this, the chosen data structure was the Set because it does not allows duplicates and it does not imposes an order. Even though the architecture of the code allows having an infinite number of levels of composite (composite containing a composite which contains a composite which contains a composite etc.), because the content of the menu items set has to be displayed, the interface allows operation on composite products with only two levels of composite (a composite can contain composites formed only by base products).

For the class *Restaurant* where the operations are implemented, the collection chosen to store the menu items is the Set for the same reasons it was chosen for the representation of the composite product. For mapping a collection of menu items to an order, through a hash table, a hashCode() method is implemented for the model of the Order.

Scenarios and use-cases

For this project, three actors can be considered: the administrator, the waiter and the chef.

The possible use-cases of the administrator are summarized in the next diagram and detailed below.



1'st use-case: *Adding a base product*

Main success scenario:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user enters a name of a base product.
4. The user enters a price.
5. The user presses the button "Add" and the program processes the input.

Alternative sequence:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.

3. The user enters an invalid name / an already existing product / an invalid price / leaves an empty field.
4. The user presses the button "Add" and the program displays an error message.
5. The flow resumes from step 3.

2'nd use-case: ***Adding a composite product***

Main success scenario:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user enters a name of a menu item.
4. The user presses "Add" after each menu item.
5. When finalizing the list, the user presses "Finalize menu item" and the program processes the input.

Alternative sequence A:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user enters an invalid name of a menu item / an inexistent base product.
4. The user presses "Add" and the program displays an error message.

Alternative sequence B:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user enters an inexistent composite product formed by existing base products.
4. The program creates the composite and displays a message.
6. The flow resumes from step 3.

3'rd use-case: ***Editing a menu item***

Main success scenario:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user enters a name of a base product.
4. The user enters a price.
5. The user presses "Edit" and the prices of the menu items containing that base product are updated.

Alternative sequence:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user enters an invalid name / an inexistent base product / an invalid price.
4. The user presses "Edit" and the program displays an error message.

4'th use-case: ***Deleting a menu item***

Main success scenario:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user enters a name of a menu item.
4. The user presses "Delete" and the menu items containing that item are deleted.

Alternative sequence:

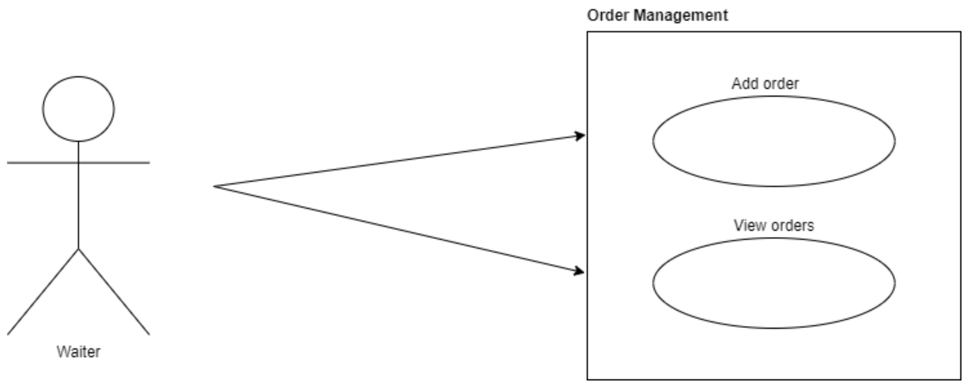
1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user enters an invalid name / an inexistent base product.
4. The user presses "Delete" and the program displays an error message.

5'th use-case: *Visualizing the menu items*

Main success scenario:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user can visualize the menu items.

The possible use-cases of the waiter are summarized in the next diagram and detailed below.



1'st use-case: *Adding an order*

Main success scenario:

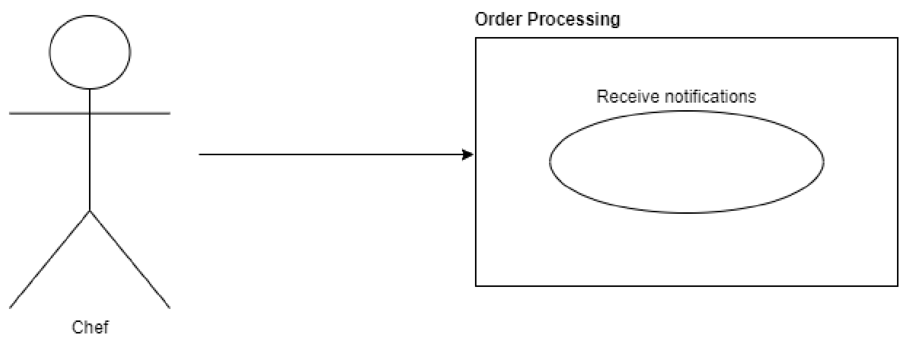
1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The users selects the item to be ordered and presses "Add".
4. The user chooses the table.
5. The user presses "Finalize order" and the program processes the input creating an order.

2'nd use-case: *Visualizing the orders*

Main success scenario:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user can visualize the previous orders.

The only use case of the chef is summarized in the next diagram and detailed below.



Use-case: ***<u>Receiving notifications</u>***

Main success scenario:

Main success scenario:

1. The user runs correctly the application.
2. The user chooses the right type of user and the right operation.
3. The user can visualize the order that it has to process.

III. Design

1. Development approach

The Restaurant Management System application is developed as before mentioned according to the given architecture, the structure obeys the object oriented paradigm. Thus, the levels or the layers of the app can be distinguished in 3 packages: *DataLayer, BusinessLayer, PresentationLayer*.

The package *DataLayer* includes the classes:

- *Serial*: contains the necessary methods for Serialization;
- *BillGenerator*: contains the method for generating a .txt bill.

The package *BusinessLayer* includes the classes:

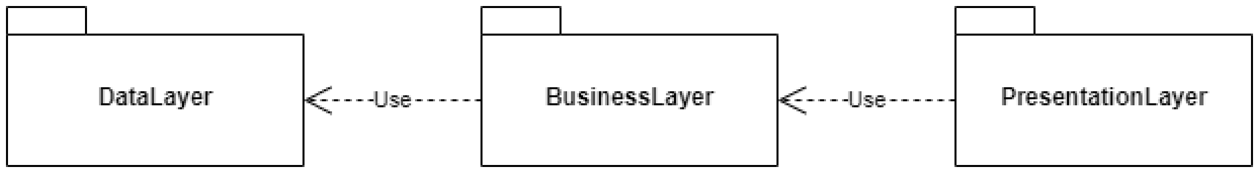
- *BaseProduct*: contains the model and the specific methods of a base product;
- *CompositeProduct*: contains the model and the specific methods of a composite product;
- *MenuItem*: abstract class, generalizing the previous two classes;
- *Order*: contains the model and the specific methods of an order;
- *Validator*: contains validation methods;
- *RestaurantProcessing*: interface containing the operations that are to be implemented within the restaurant;
- *Restaurant*: contains the methods implementing the needed operations;
- *Chef*: contains the model and the specific methods for the chef;
- *Main*: contains only the main method.

The package *PresentationLayer* includes the classes:

- *RestaurantGUI;*
- *AdministratorGUI;*
- *NewMenuItemGUI;*
- *DeleteMenuItemGUI;*
- *EditMenuItemGUI;*
- *ViewMenuItemsGUI;*
- *WaiterGUI;*
- *AddOrderGUI;*
- *ViewOrdersGUI;*
- *ChefGUI.*

2. <u>Structure and UML Diagrams</u>

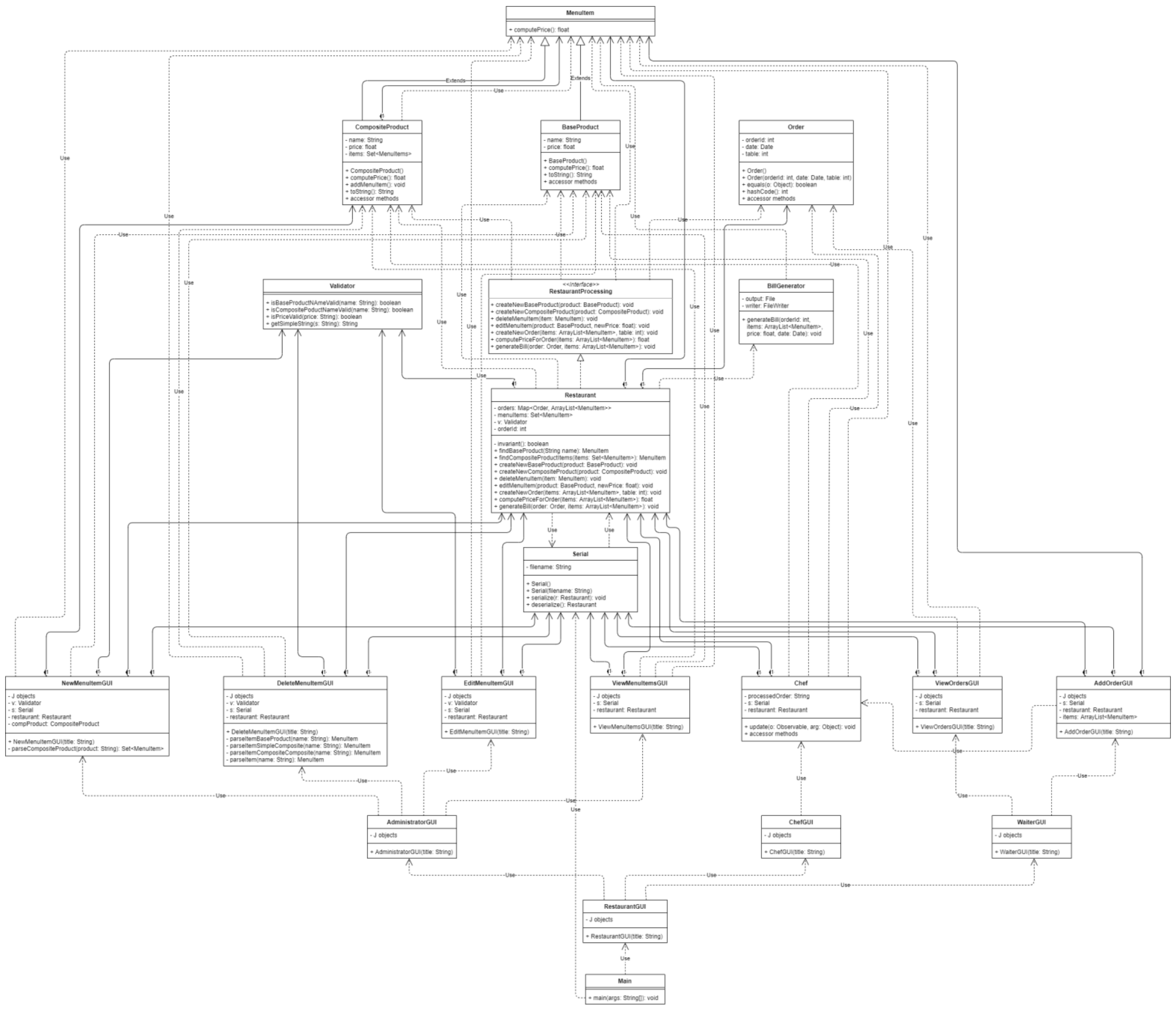
2.a. Package Diagram



The *PresentationLayer* uses the *BusinessLayer* to implement the operations and the *BusinessLayer* uses the *DataLayer* which is responsible with storing the state of the application and the .txt outputs.

The content of the packages is detailed in the following class diagram.

2.b. Class Diagram


MenuItem
+ computePrice(): float
Extends
Extends
Use
CompositeProduct
- name: String
- price: float
- items: Set<MenuItems>
+ CompositeProduct()
+ computePrice(): float
+ addMenuItem(): void
+ toString(): String
+ accessor methods
BaseProduct
- name: String
- price: float
+ BaseProduct()
+ computePrice(): float
+ toString(): String
+ accessor methods
Order
- orderId: int
- date: Date
- table: int
+ Order()
+ Order(orderId: int, date: Date, table: int)
+ equals(o: Object): boolean
+ hashCode(): int
+ accessor methods
Validator
+ isBaseProductNAmeValid(name: String): boolean
+ isCompositePoductNameValid(name: String): boolean
+ isPriceValid(price: String): boolean
+ getSimpleString(s: String): String
<<interface>>
RestaurantProcessing
+ createNewBaseProduct(product: BaseProduct): void
+ createNewCompositeProduct(product: CompositeProduct): void
+ deleteMenuItem(item: MenuItem): void
+ editMenuItem(product: BaseProduct, newPrice: float): void
+ createNewOrder(items: ArrayList<MenuItem>, table: int): void
+ computePriceForOrder(items: ArrayList<MenuItem>): float
+ generateBill(order: Order, items: ArrayList<MenuItem>): void
BillGenerator
- output: File
- writer: FileWriter
+ generateBill(orderId: int, items: ArrayList<MenuItem>, price: float, date: Date): void
Restaurant
- orders: Map<Order, ArrayList<MenuItem>>
- menuItems: Set<MenuItem>
- v: Validator
- orderId: int
- invariant(): boolean
+ findBaseProduct(String name): MenuItem
+ findCompositeProductItems(items: Set<MenuItem>): MenuItem
+ createNewBaseProduct(product: BaseProduct): void
+ createNewCompositeProduct(product: CompositeProduct): void
+ deleteMenuItem(item: MenuItem): void
+ editMenuItem(product: BaseProduct, newPrice: float): void
+ createNewOrder(items: ArrayList<MenuItem>, table: int): void
+ computePriceForOrder(items: ArrayList<MenuItem>): float
+ generateBill(order: Order, items: ArrayList<MenuItem>): void
Serial
- filename: String
+ Serial()
+ Serial(filename: String)
+ serialize(r: Restaurant): void
+ deserialize(): Restaurant
NewMenuItemGUI
- J objects
- v: Validator
- s: Serial
- restaurant: Restaurant
- compProduct: CompositeProduct
+ NewMenuItemGUI(title: String)
- parseCompositeProduct(product: String): Set<MenuItem>
DeleteMenuItemGUI
- J objects
- v: Validator
- s: Serial
- restaurant: Restaurant
+ DeleteMenuItemGUI(title: String)
- parseItemBaseProduct(name: String): MenuItem
- parseItemSimpleComposite(name: String): MenuItem
- parseItemCompositeComposite(name: String): MenuItem
- parseItem(name: String): MenuItem
EditMenuItemGUI
- J objects
- v: Validator
- s: Serial
- restaurant: Restaurant
+ EditMenuItemGUI(title: String)
ViewMenuItemsGUI
- J objects
- s: Serial
- restaurant: Restaurant
+ ViewMenuItemsGUI(title: String)
Chef
- processedOrder: String
- s: Serial
- restaurant: Restaurant
+ update(o: Observable, arg: Object): void
+ accessor methods
ViewOrdersGUI
- J objects
- s: Serial
- restaurant: Restaurant
+ ViewOrdersGUI(title: String)
AddOrderGUI
- J objects
- s: Serial
- restaurant: Restaurant
- items: ArrayList<MenuItem>
+ AddOrderGUI(title: String)
AdministratorGUI
- J objects
+ AdministratorGUI(title: String)
ChefGUI
- J objects
+ ChefGUI(title: String)
WaiterGUI
- J objects
+ WaiterGUI(title: String)
RestaurantGUI
- J objects
+ RestaurantGUI(title: String)
Main
+ main(args: String[]): void
Use
1

The original diagram will be attached to the project for a clearer visualization.

IV. Implementation

1. DataLayer package

1.a. Class Serial

Instance variables: *filename*: String.

In this class, the methods for serialization technique are implemented.

The first method, `public void serialize(Restaurant r) {...}` is used implement the serialization part. It receives as argument an object of type *Restaurant* of which state's it writes to a .ser file.

The second important method is `public Restaurant deserialize() {...}`, which returns the last stored state of a *Restaurant* type object from the .ser file.

1.b. Class BillGenerator

Instance variables: *output*: File, *writer*: FileWriter.

The only method of this class is `public void generateBillFile(int orderId, ArrayList<MenuItem> items, float price, Date date) {...}` and it has the purpose of creating a .txt file containing data from a specific order. Hence, it receives as parameters all the values that it has to write to the file.

2. BusinessLayer package

2.a. Class MenuItem

No instance variables.

It is an abstract class which will be extended by specific classes: *BaseProduct* and *CompositeProduct*, everything being implemented according to the Composite Design Pattern. The only method is `public abstract float computePrice();`, which is to be implemented in the extending classes. The class implements *Serializable* because objects of this type are to be serialized.

2.b. Class BaseProduct

Instance variables: *name*: String, *price*: float.

Besides constructors, getters and setters, the class implements the method form its abstract class, *MenuItem*, `public float computePrice() {...}`, which only returns the price of the object since it is a singular object.

Another method that is present in this class is the `public String toString() {...}` method which returns a string containing the attributes of a base product, useful when displaying the content of the menu.

2.c. Class CompositeProduct

Instance variables: *name*: String, *price*: float, *items*: Set<MenuItem>.

This class is similar with the previous one, it implements mostly the same methods since it also extends the *MenuItem* abstract class. The difference consists in computing the price in the `public float computePrice() {...}` method where the prices of each menu item from the set *items* is added to the final price of the composite product. One extra method comparing to the *BaseProduct* class is the `public void addMenuItem(MenuItem item) {...}` method which adds a menu item to the set of menu items of the composite product.

2.d. <u>Class Order</u>

Instance variables: *orderId*: int, *date*: Date, *table*: int.

Firstly, this class contains constructors, getters and setters in order to be able to access objects of this type.

The important method in this class is the `public int hashCode() {...}` method which computes a hash code using the attributes of the order.

The class implements *Serializable* interface because objects of this type are to be serialized.

2.e. <u>Class Validator</u>

No instance variables.

The class contains the methods `public boolean isBaseProductNameValid(String name) {...}`, `public boolean isCompositeProductNameValid(String name) {...}`, `public boolean isPriceValid(String price) {...}` are used to validate an input string. They return a boolean value according to a regular expression.

The other method in this class is `public String getSimpleString(String s) {...}`, which returns a simpler version of the string. It eliminates the extra spaces and the spaces at the beginning and at the end of the string received as parameter.

The class implements *Serializable* interface because objects of this type are to be serialized.

2.f. <u>RestaurantProcessing Interface</u>

This interface contains the methods corresponding to the operations to be implemented for the three types of users: administrator, waiter and chef. The methods will be implemented in the *Restaurant* class.

2.g. <u>Class Restaurant</u>

Instance variables: *orders*: Map<Order, ArrayList<MenuItem>>, *menuItems*: Set<MenuItem>, *v*: Validator, *orderId*: int, *menuItemsS*: Set<MenuItem>.

The class extends the class *Observable* because an object of this type is to be observed by a chef. The chef will be notified after a "create order" command will be executed. It implements the *RestaurantProcessing* and the *Serializable* interfaces; an object of this type is to be serialized.

The variable menuItems is static and it stores the same content as menuItemsS. The variable is static because all the packages need access to the same content. The reason for having two variables which store the same content is because the static one cannot be serialized.

The class contains an invariant method as a result of using Design by Contract: `private boolean invariant() {...}`. The purpose of this method is to check whether the set of menu items is formed only by non-null objects. It is a method of type "well formed" because it checks a characteristic that should always be true.

The method `public MenuItem findBaseProduct(String name) {...}` searches and returns a menu item having the name given in the argument. Firstly, the given string is simplified and then the method iterates all over the menu items in the set checking every base product's name. When it finds a product with the desired name, it returns it. In case no product is found, the method returns null.

The method `public MenuItem findCompositeProductItems(Set<MenuItem> items) {...}` has a similar usage as the previous one but it is used to search a composite product. Since it is difficult to identify exactly a composite product by its name, the method receives as argument the set of its menu items. In the same manner as the previous method, this one also iterates all over the menu items, but checking the composite products to determine whether the set of menu items in their composition matches to the one given as parameter. If a match is found then the method returns that menu item, otherwise it returns null.

The method `public void createNewBaseProduct(BaseProduct product) {...}` is the one implementing a part of the "add menu item" operation corresponding to the administrator. The application is built to create separately the base products from the composite product, so this method only implements the creation of a base product. The method checks firstly as pre-condition that the product to be inserted is not-null. Then, the invariant condition is also checked and the product is inserted in the set of menu items. The state of the restaurant is serialized and as a post-condition, the size of the current set should be incremented with one. The existence of the product and the validity of the name and price are checked in the GUI.

The method `public void createNewCompositeProduct(CompositeProduct product) {...}` behaves in the exact same manner as the previous one, with the only exception that the argument is a composite product, thus it inserts a composite product in the menu.

The method `public void deleteMenuItem(MenuItem item) {...}` is the one responsible with the "delete menu item" operation, also corresponding to the administrator. First, the method checks that the item to delete is non-null as a pre-condition and then it creates a set of menu items that have to be deleted. The method will delete all the items that have in their composition the item received as an argument. It iterates through the set of menu items and if the item is a base product, then it simply inserts it into the set of items to be deleted, otherwise, if it is a composite product, then it checks whether in its list is present the item form the argument. After all the products to be deleted are marked, they are removed from the set of menu items, the state of the restaurant is serialized and the invariant condition is checked.

The method `public void editMenuItem(BaseProduct product, float newPrice) {...}` is implementing the "edit menu item" operation. The arguments represent the base product which price's is updated. As pre-conditions, the product is checked to be non-null, the price to be greater than 0, and the invariant is also checked. The base product given as a parameter is searched and its price is modified and than for all the composite products in the set of menu items, which contain that base product, the price is recomputed. In the end, the state of the restaurant is serialized and the invariant is checked again.

The method `public void createNewOrder(ArrayList<MenuItem> items, int table) {...}` is one specific to the waiter and it inserts a new order in the hash table. As pre-conditions, the validity of the parameters is checked, the list of items to be non-null and the table greater than 0. Than an order for the given table is created having the current time and the next available id in increasing order. A new entry of the hash table is created having as key the order and as content, the list of menu items. A bill is automatically generated and the state of the restaurant is serialized. The chef is notified that a new order was placed and it has to be processed.

The method `public float computePriceForOrder(ArrayList<MenuItem> items) {...}` is implemented to get the price of an order. As pre-condition, the list given as parameter is checked to be non-null. The price of each item in the list is added to the final price which is returned right after it is checked to be greater than 0 as a post-condition.

The last method in this class is `public void generateBill(Order order, ArrayList<MenuItem> items) {...}` which only calls the *BillGenerator* to build the .txt file.

2.h. Class Chef

Instance variables: *processedOrder*: String, *s*: Serial, *restaurant*: Restaurant.

The class implements the interface Observer because each time the restaurant creates an order, it receives a signal that the chef has to prepare something. Hence, the class implements the `public void update(Observable o, Object arg) {...}` method which constructs a string which containing the last order which is handled by the chef. The string will be displayed in its interface.

2.i. Class Main

This class only contains the main method which sets gets the .ser file for serialization and sets the main frame to visible.

3. PresentationLayer package

3.a. Class RestaurantGUI

This class creates a GUI from where the user is chosen. Depending on what type of personal the user chooses, "Administrator", "Waiter", "Chef" , a new interface is open.

3.b. Class AdministratorGUI

This class creates a GUI from where the operation executed by the administrator is chosen. Each of the operation "New menu item", "Delete menu item", "Edit menu item", "View menu items" is placed on a button, each of them opening a new corresponding graphic interface.

3.c. Class NewMenuItemGUI

This class creates a GUI through which the administrator can create a menu item. The user can create base and composite products separately.

When creating a base product, the name and the price are validated. The program checks whether this product already exists and if it does, then a message is displayed. Otherwise, a base product is created with those attributes and it is inserted in the set of menu items through the corresponding method in class *Restaurant*.

When creating a composite product, the user enters either existing base products, either composites formed only by bases, existing or not. After each item, when user presses the button "Add item", the code verifies the correctness of the input and in case there is a problem, a message is displayed. If the entered simple composite product does not exist then it is created. Each of these items are added into a list for the final composite item. When the button "Finalize item" is clicked, a new composite product having that list of products is created through the corresponding method in class *Restaurant*.

3.d. Class DeleteMenuItemGUI

This class creates a GUI through which the administrator can delete a menu item. The user enters a menu item in the text field and then the code validates and parses the input. If the entered product does not exist, a message is displayed. The obtained menu item is sent as a parameter to the *deleteMenuItem* method in class *Restaurant* and in this way it is deleted. Also the items containing this product in their items list are deleted.

3.e. Class EditMenuItemGUI

This class creates a GUI through which the administrator can edit a menu item. The user enters an existing base product which price's has to be updated and the new price. The program validates the input and checks the existence of the product. If the inputs are good, than the base product is obtained and the rest of the operation is implemented by the *editMenuItem* method from the class *Restaurant*.

3.f. Class ViewMenuItemsGUI

This class creates a GUI which displays the content of the menu. It constructs a table in which the name of the product and the price are visible for each menu item.

3.g. Class WaiterGUI

This class creates a GUI from where the operation executed by the waiter is chosen. Each of the operation "Create order", "View orders" is placed on a button, each of them opening a new corresponding graphic interface.

3.h. Class AddOrderGUI

This class creates a GUI through which an order can be placed. The entire content of the menu is placed inside of a combo box and for each ordered product, the waiter selects it from the combo box and presses the button "Add item". The wanted items are added in a list which will be sent to the *createNewOrder* from the class *Restaurant*. Another parameter sent to that function will be the number of the table which is also obtained from a combo box in the interface.

3.i. Class ViewOrdersGUI

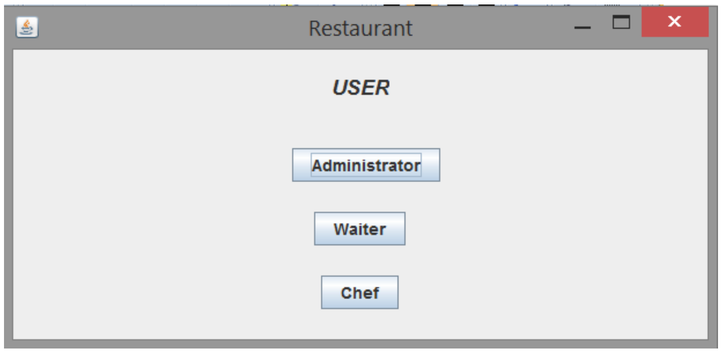
This class creates a GUI which displays the content of the hash table storing the orders. It constructs a table in the same way it is constructed in the class *ViewMenuItemsGUI*.

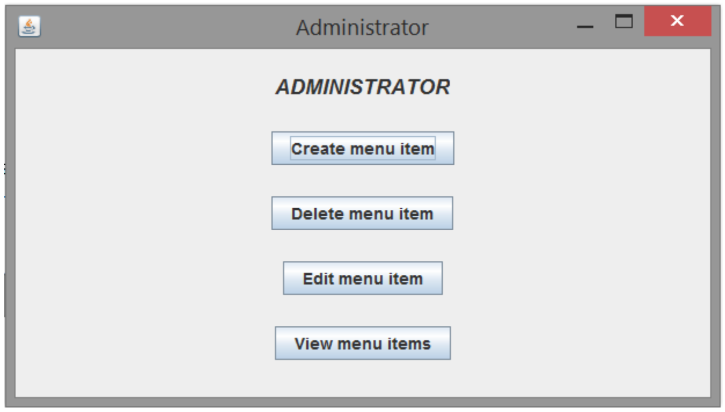
3.j. Class ChefGUI

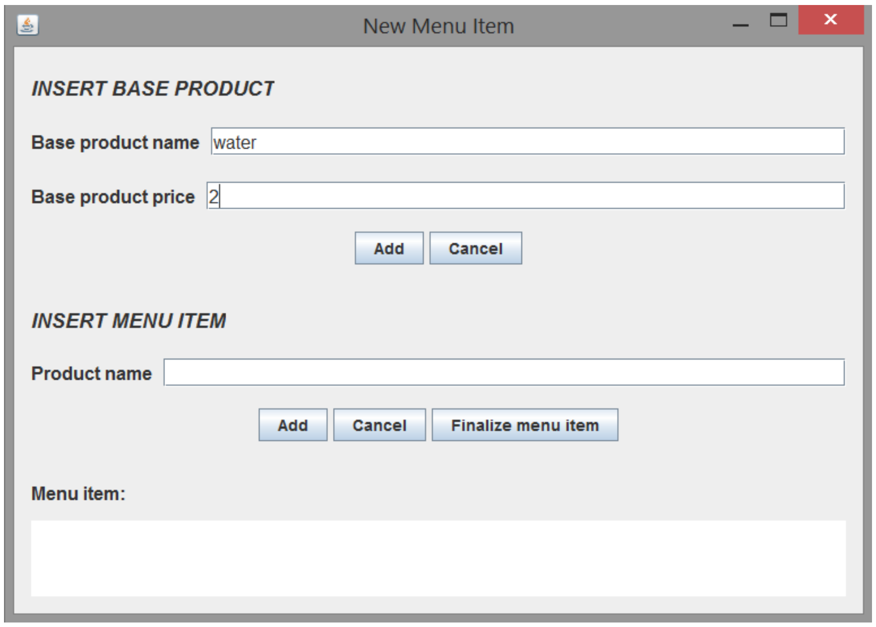
Is the last class in this package and creates a GUI which displays the order currently processed by the chef.

V. Results

The results are visible through the created interfaces. The results of each operations can be checked by viewing the menu items or the orders tables. Also the serialization process can be checked in this way. After re-running the application, the content of these tables is kept.

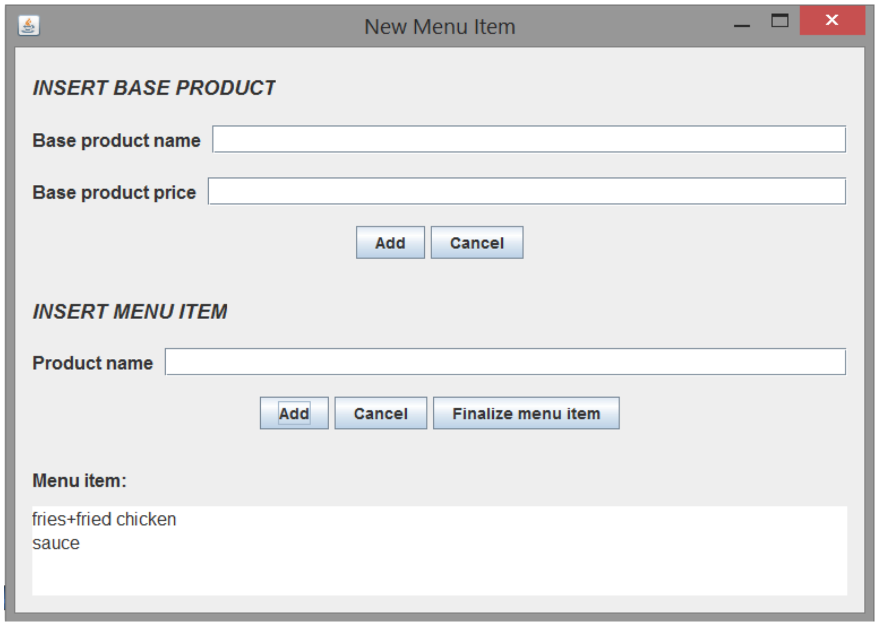




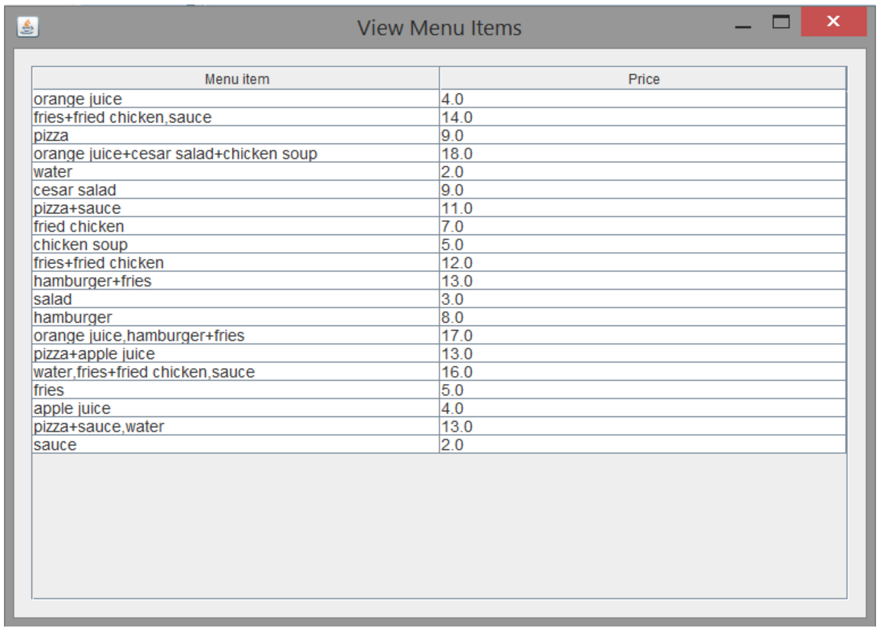


Base product name water

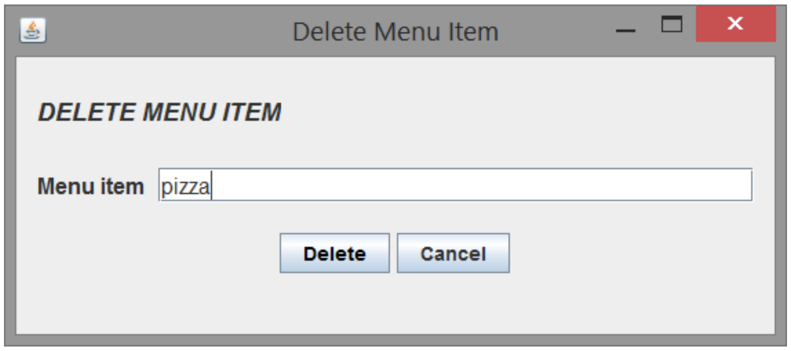
Base product price 2



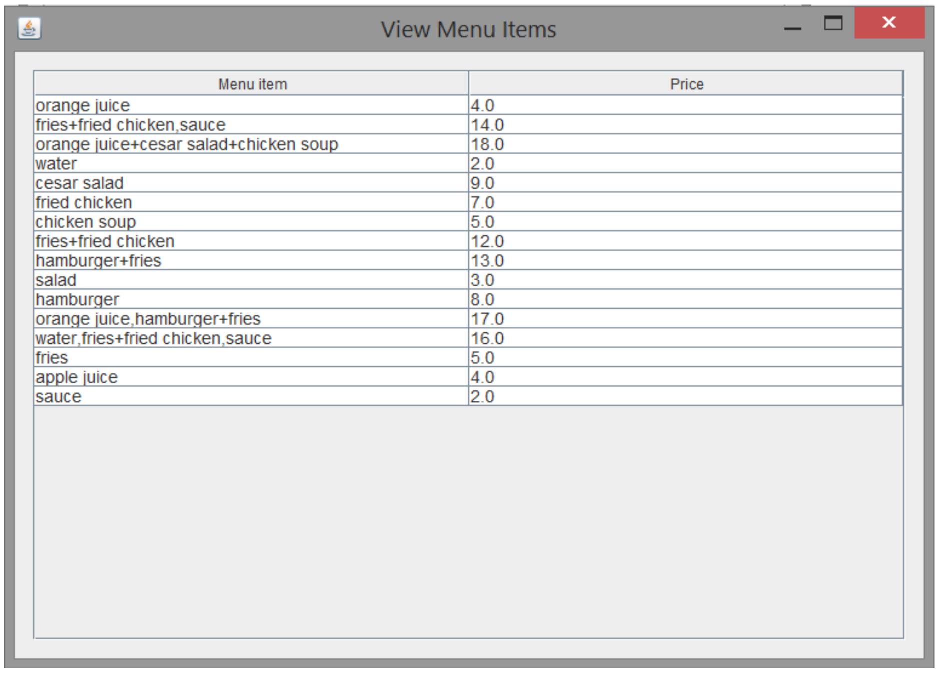
fries+fried chicken
sauce



Menu item	Price
orange juice	4.0
fries+fried chicken,sauce	14.0
pizza	9.0
orange juice+cesar salad+chicken soup	18.0
water	2.0
cesar salad	9.0
pizza+sauce	11.0
fried chicken	7.0
chicken soup	5.0
fries+fried chicken	12.0
hamburger+fries	13.0
salad	3.0
hamburger	8.0
orange juice,hamburger+fries	17.0
pizza+apple juice	13.0
water,fries+fried chicken,sauce	16.0
fries	5.0
apple juice	4.0
pizza+sauce,water	13.0
sauce	2.0

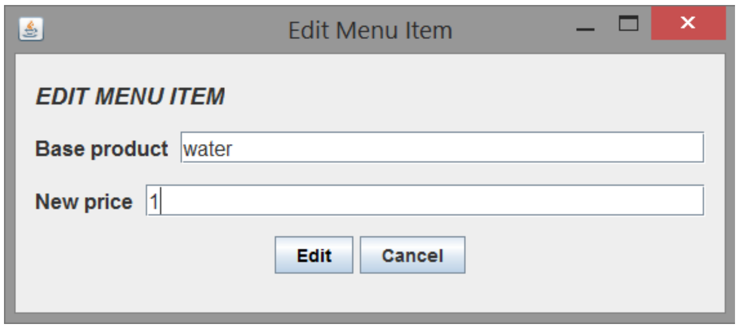


Menu item: pizza



View Menu Items

Menu item	Price
orange juice	4.0
fries+fried chicken,sauce	14.0
orange juice+cesar salad+chicken soup	18.0
water	2.0
cesar salad	9.0
fried chicken	7.0
chicken soup	5.0
fries+fried chicken	12.0
hamburger+fries	13.0
salad	3.0
hamburger	8.0
orange juice,hamburger+fries	17.0
water,fries+fried chicken,sauce	16.0
fries	5.0
apple juice	4.0
sauce	2.0

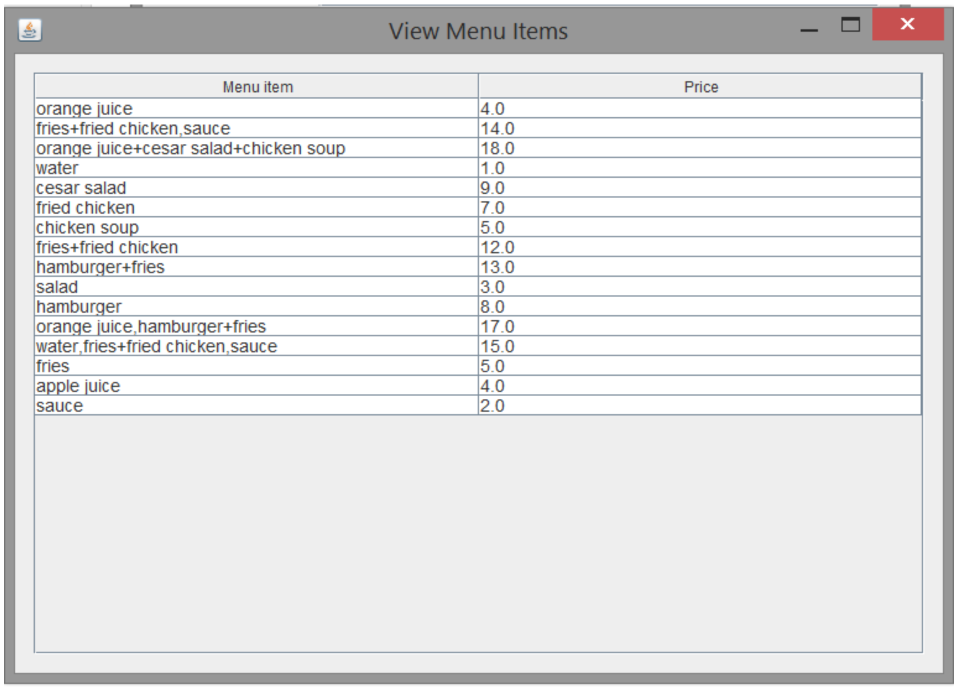


Edit Menu Item

EDIT MENU ITEM

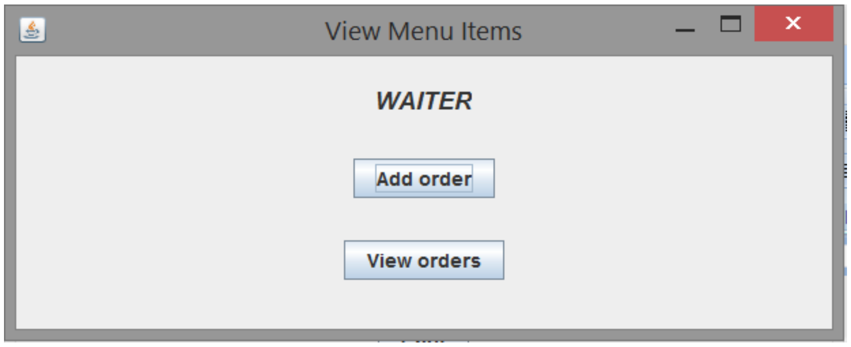
Base product: water

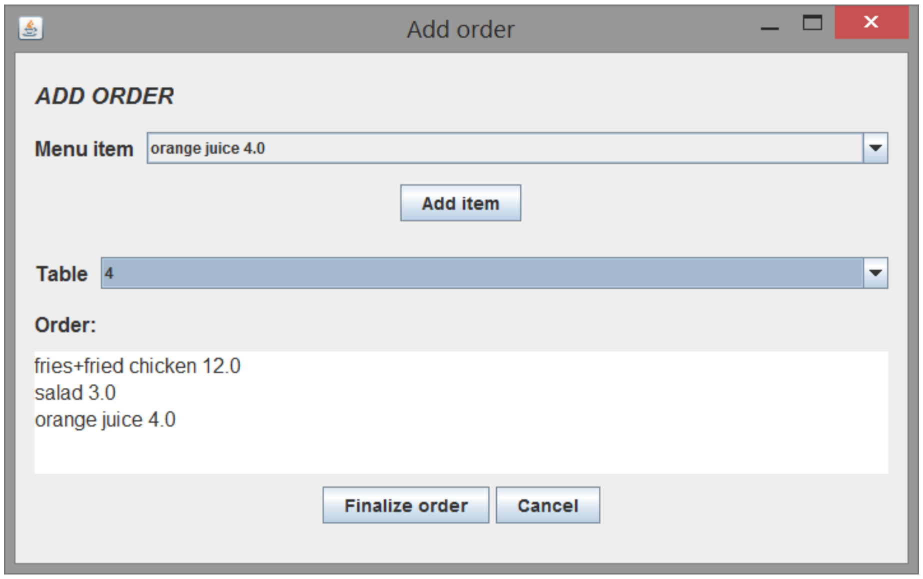
New price: 1



View Menu Items

Menu item	Price
orange juice	4.0
fries+fried chicken,sauce	14.0
orange juice+cesar salad+chicken soup	18.0
water	1.0
cesar salad	9.0
fried chicken	7.0
chicken soup	5.0
fries+fried chicken	12.0
hamburger+fries	13.0
salad	3.0
hamburger	8.0
orange juice,hamburger+fries	17.0
water,fries+fried chicken,sauce	15.0
fries	5.0
apple juice	4.0
sauce	2.0



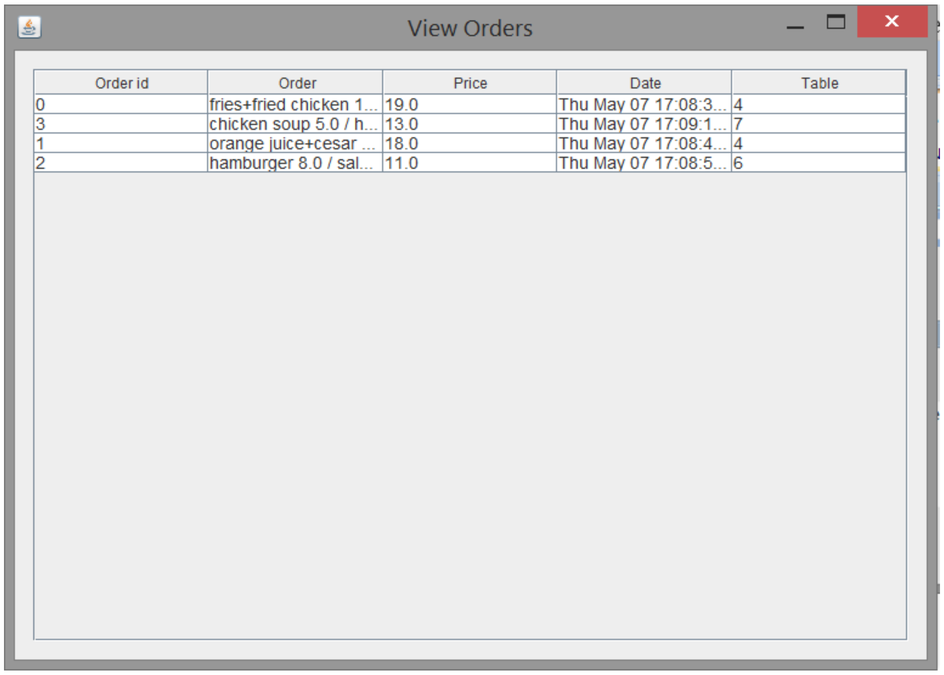


Menu item: orange juice 4.0

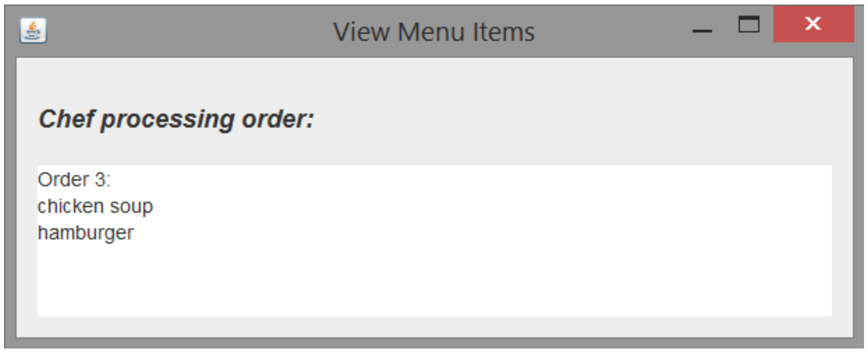
Table: 4

Order:

fries+fried chicken 12.0
salad 3.0
orange juice 4.0



Order id	Order	Price	Date	Table
0	fries+fried chicken 1...	19.0	Thu May 07 17:08:3...	4
3	chicken soup 5.0 / h...	13.0	Thu May 07 17:09:1...	7
1	orange juice+cesar ...	18.0	Thu May 07 17:08:4...	4
2	hamburger 8.0 / sal...	11.0	Thu May 07 17:08:5...	6



Chef processing order:

```
Order 3:
chicken soup
hamburger
```

VI. Conclusions

The Restaurant Management System fulfills all the requirements of the laboratory work.

For future improvements the following can be considered:

- The possibility of having more than one user of each type;
- Improving the "New menu item" interface such that it allows more levels of composite.

VII. Bibliography

1. https://www.tutorialspoint.com/java/java_serialization.htm

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

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

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