TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

Fundamental Programming Techniques

Restaurant Management System

Assignment 4

Hunor Debreczeni

Group: 30423

Table of Contents

Objective

The objective of the application is to implement a restaurant management system. The system should be able to provide a graphical user interface for three types of users: administrator, waiter and chef. Each one may do different actions.

- The administrator can add, delete and modify the existing products from the menu list.
- The waiter can create new orders by using the existing menu items and also specify a table for each order. He/ she can also add elements later on if needed and compute the total for the order and generate a bill for it.
- The chef can only view the current orders and their contents/ items and should cook/ prepare the order in real life.

The system should also allow easy migration of data from the restaurant, through an exported file, whose name can be modified by using command line arguments.

Problem Analysis, Modeling, Scenarios, Use cases

The application may be used in real life restaurants who are working on a smaller scale. Products can be easily added by a manager and waiters/ waitresses can see each change in real time. Every change affects each interface, so there will be no misunderstanding. There are base products, that contain a single item and composite products that may contain an any number of base products. This helps to make offers/ menus that appear on the menu in a much shorter way and it's also easier to understand. An order may contain multiple instances of the same item, so any quantities may be ordered.

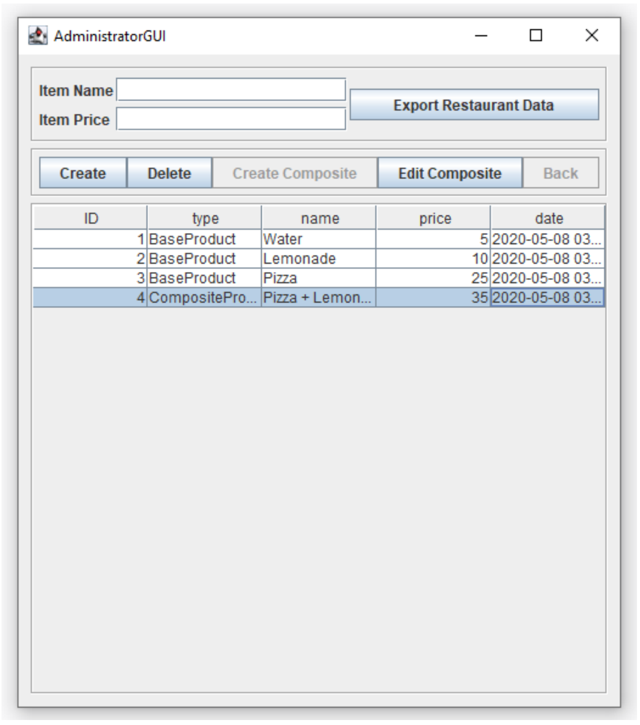
In order to use the application, the end-user should simply run the *.jar* file the following way:

Java -jar <File-Name.jar> <?File_to_Import_from>

In case you have an exported file from previous runs of the program, you may use the **restaurant.ser** name which is used by default to import data. If you want to change the name of the file, replace the *<?File_to_Import_from>* with your file name. This argument is **not necessary** and neither the input file is.

On first-run, without import file. There must be a manager, who accesses the **AdministratorGUI**, where he may, but should do the following to ensure the chance for the waiter and chef to use their interface too:

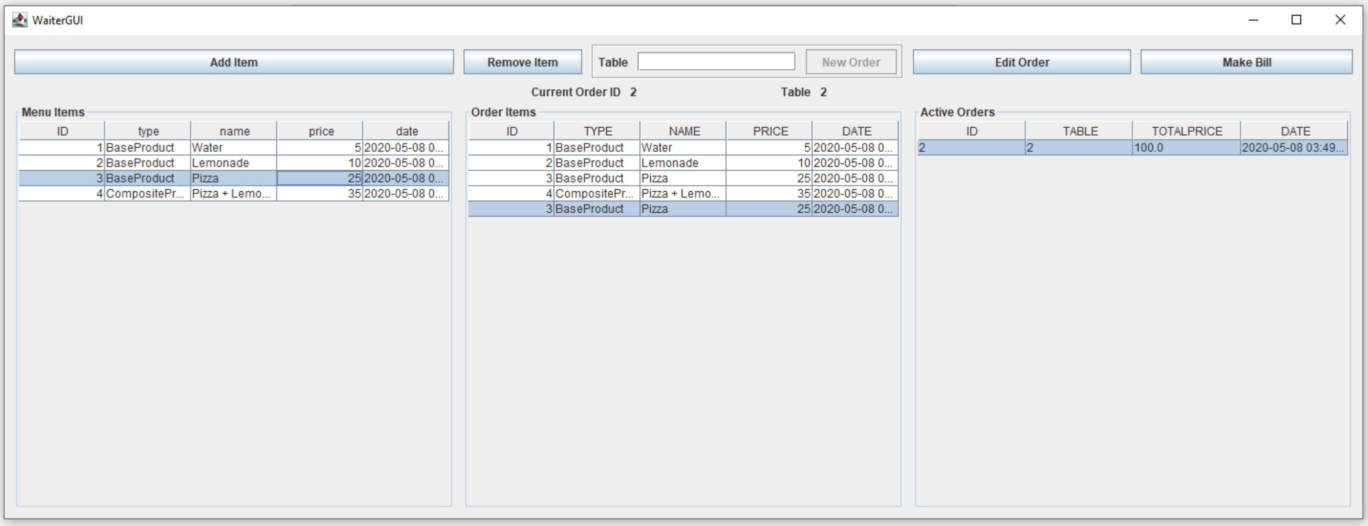
- **Add Base Products** with specific name and price (of type double)
- **Add Composite Products** with specific name
  - The price is calculated by adding up the price of the base products
  - You may not include another composite product into a new composite product
- **Delete** any kind of products
  - This deletes the product from the waiters menu items list also, but not from the orders. An ordered product can not be taken back from the client.
- **Edit** any kind of products
  - In case of the **Base Products** the administrator can edit a field by double-clicking on a editable field from the table and simply change the value. If the cell becomes red, it means that the input is wrong and he/ she should fix it, otherwise the old value will remain.
  - In case of the **Composite Products** the *Edit Composite* button should be pressed and the table will switch to the contents of the Composite Product, where the editing procedure is the same. Only if an item is deleted while in Composite Product editing mode, that item will only get removed from the composite product. Other



ID	type	name	price	date
1	BaseProduct	Water	5	2020-05-08 03...
2	BaseProduct	Lemonade	10	2020-05-08 03...
3	BaseProduct	Pizza	25	2020-05-08 03...
4	CompositePro...	Pizza + Lemon...	35	2020-05-08 03...

changes also affect the original product. The end-user may return back to every Menu Item by using the *Back* button.

After doing some operations in the Administrator GUI, the waiter can do it's job and process clients and their orders by using the already added Menu Items. He/ she may do the following tasks:



Current Order ID 2 Table 2

Menu Items

ID	type	name	price	date
1	BaseProduct	Water	5	2020-05-08 0...
2	BaseProduct	Lemonade	10	2020-05-08 0...
3	BaseProduct	Pizza	25	2020-05-08 0...
4	CompositePr...	Pizza + Lemo...	35	2020-05-08 0...

Order Items

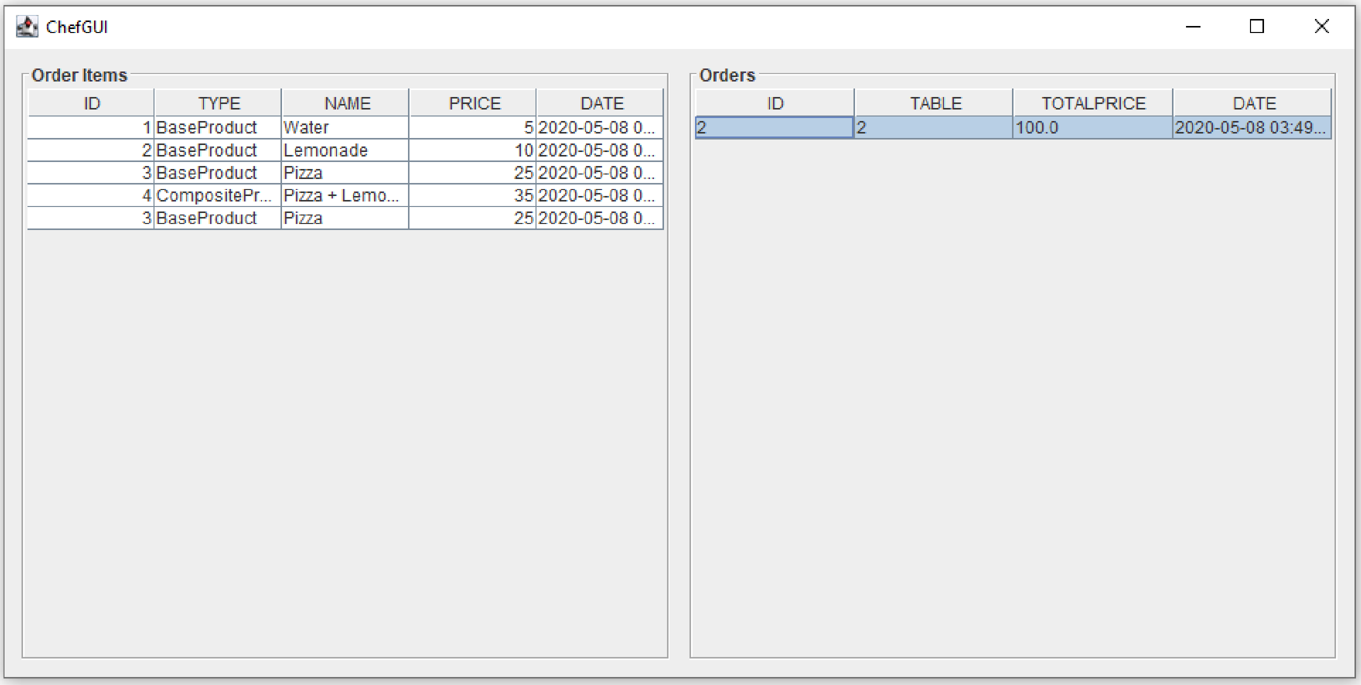
ID	TYPE	NAME	PRICE	DATE
1	BaseProduct	Water	5	2020-05-08 0...
2	BaseProduct	Lemonade	10	2020-05-08 0...
3	BaseProduct	Pizza	25	2020-05-08 0...
4	CompositePr...	Pizza + Lemo...	35	2020-05-08 0...
3	BaseProduct	Pizza	25	2020-05-08 0...

Active Orders

ID	TABLE	TOTALPRICE	DATE
2	2	100.0	2020-05-08 03:49...

- **Create New Order** with table ID, that is given by the waiter. The button *New Order* will be enabled only if the input field next to it is not empty. A waiter may create any number of orders with the same table ID, because the customers may ask for a separate bill at the end.
- **Add Item to an Order**. It is only possible if an order has been selected. In case of a creation of a new order, it gets automatically selected. The selected item from the *Menu Items* table will be added to the currently selected order.
- **Edit Order**. If the waiter wants to edit an older order, he/ she must select it first from the *Active Orders* table and select *Edit Order*. Afterwards, every item that belongs to that order will show up in the *Order Items* table.
- **Remove Item**. On item selection from the *Order Items* table, the button gets enabled, and upon clicking it, this will remove the selected item from the current order.
- **Make Bill**. Generates a bill for the current order with it's details in the same folder where the application is ran. At the same time, the order will be removed from the application, because there is no need for us at the moment to store any old data.

The last interface that is used in the application is the **ChefGUI**. It is made for the chef to get notified of new orders and their contents. He may select an order, and it's contents will show up on the left..



Order Items

ID	TYPE	NAME	PRICE	DATE
1	BaseProduct	Water	5	2020-05-08 0...
2	BaseProduct	Lemonade	10	2020-05-08 0...
3	BaseProduct	Pizza	25	2020-05-08 0...
4	CompositePr...	Pizza + Lemo...	35	2020-05-08 0...
3	BaseProduct	Pizza	25	2020-05-08 0...

Orders

ID	TABLE	TOTALPRICE	DATE
2	2	100.0	2020-05-08 03:49...

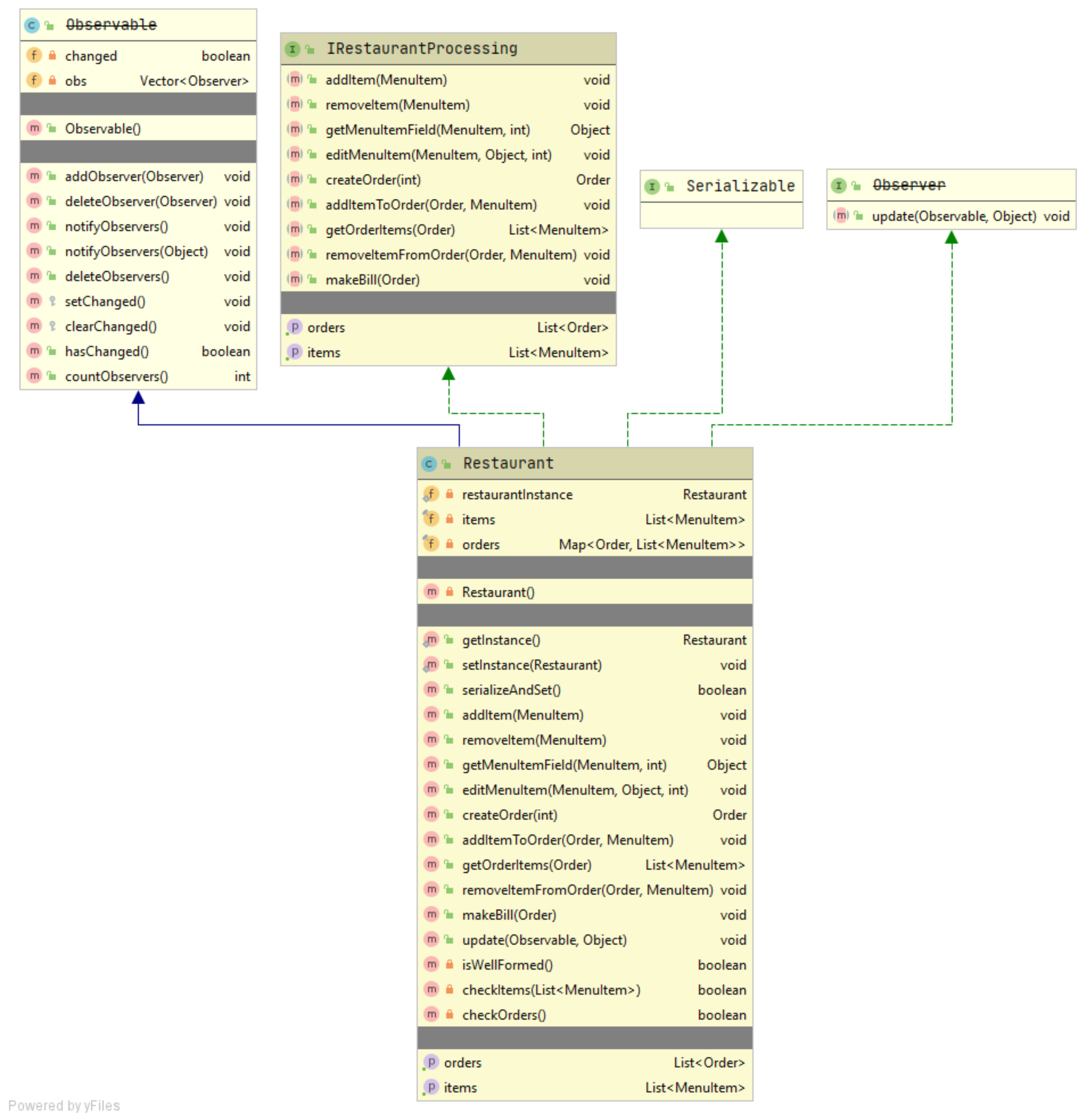
For **selecting an item**, a single click should be used on any of it's fields in a row.

For **selecting multiple items**, *CTRL* should be held down and afterwards repeating the process for the single item selection**.**

Every table is synchronised with each other.

Design

For implementing the system and to make it easy to manage, the best way is to make three different graphical user interfaces for the three types of users. To make them all synchronized, there should be a Restaurant object that will be used everywhere. On an update of Items/ Orders, each table should update.



~~Observable~~
changed boolean
obs Vector<Observer>
Observable()
addObserver(Observer) void
deleteObserver(Observer) void
notifyObservers() void
notifyObservers(Object) void
deleteObservers() void
setChanged() void
clearChanged() void
hasChanged() boolean
countObservers() int
IRestaurantProcessing
addItem(MenuItem) void
removeItem(MenuItem) void
getMenuItemField(MenuItem, int) Object
editMenuItem(MenuItem, Object, int) void
createOrder(int) Order
addItemToOrder(Order, MenuItem) void
getOrderItems(Order) List<MenuItem>
removeItemFromOrder(Order, MenuItem) void
makeBill(Order) void
orders List<Order>
items List<MenuItem>
Serializable
~~Observer~~
update(Observable, Object) void
Restaurant
restaurantInstance Restaurant
items List<MenuItem>
orders Map<Order, List<MenuItem>>
Restaurant()
getInstance() Restaurant
setInstance(Restaurant) void
serializeAndSet() boolean
addItem(MenuItem) void
removeItem(MenuItem) void
getMenuItemField(MenuItem, int) Object
editMenuItem(MenuItem, Object, int) void
createOrder(int) Order
addItemToOrder(Order, MenuItem) void
getOrderItems(Order) List<MenuItem>
removeItemFromOrder(Order, MenuItem) void
makeBill(Order) void
update(Observable, Object) void
isWellFormed() boolean
checkItems(List<MenuItem>) boolean
checkOrders() boolean
orders List<Order>
items List<MenuItem>
Powered by yFiles

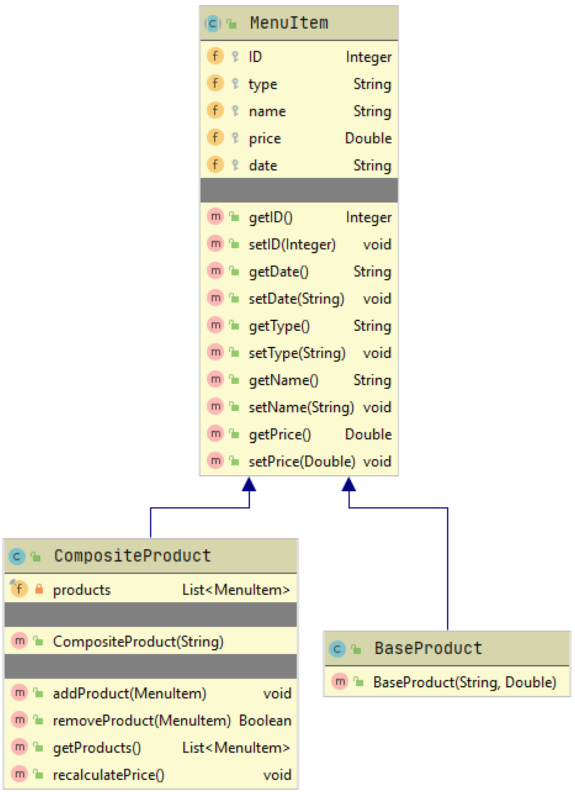
In order to make everything synchronized, the application uses a singleton pattern for the restaurant, so there will exist at most one instance of the Restaurant. By using this pattern, it also makes easier to get the instance of the restaurant from any method in the program.

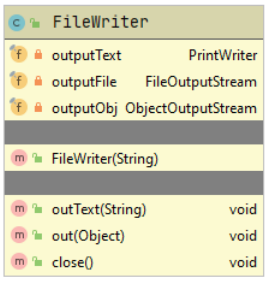
To make the tables synchronized with each other and to update dynamically, the Observer Design pattern is used. It implements an update method that is called when an object that extends the Observable class, calls the notify() method. In our case, the

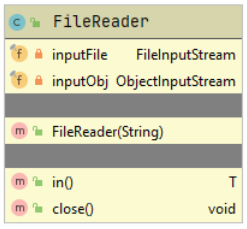
MenuItem interface and the restaurant inherits the Observable class. The MenuItem calls the notify method when the elements of a Composite Product are changed. The restaurant observes that update and it calls yet again a notify method. This call will be observed by the three GUI classes, that either update their table's content or the whole table.

The ArrayList is mostly used in the project as a structure date, due to the fact that each item has a unique ID and it makes easier to find and get the elements if needed. Although, for the orders, there's associated a list of MenuItems. To make this relation easier to handle, a HashMap is used, that gives us faster access of the elements due to the hashing technology. There's no chance that two orders have the same hash code, so it's safe to use this structure.

Another important pattern used is the Composite Design Pattern, that lets us groups objects in such a way, that they are handled in the same way as if it were a single object. Each object inherits the MenuItem abstract class, with the only difference, that the Composite Product contains a list of objects with type MenuItem and the additional methods that handle the insertion/ and remove operations on the list.







The Restaurant, Order and MenuItem classes implement the Serializable class, that makes it possible to make an "export" of the restaurant instance and maybe move every data to another PC. Upon using a FileWriter class or FileReader class, the application can load from/ save to a file every content of an object that implements the Serializable class.

Packages

Business Level Logic

Contains classes that are specific to the business, more exactly 4 classes, each specific for the models that appear in the project (Order, Restaurant, Composite Product and Base Product). Each of these classes contain methods that are only available and used in the specific business, in our case, in a restaurant. As an example making an order, making a bill, adding new Menu Items, etc.

Data Abstraction Layer

This is the part where the data is either parsed from a file or imported/ written into a file. By using the classes in this package, it's easy to read/write any kind of object/ data to/from any file.

Util

Contains helper classes, that contain global functions, like getting the values of given object of a given type by using reflection. Also implements the method that does the opposite, sets a specific value. Exceptions being caught and checked.

The **Constants** class uses static variables, as an example the IDs are taken from here on each Order/ MenuItem creation and incremented for later use. Also the name of the input file is saved and taken from here that is used on serialization of the restaurant object.

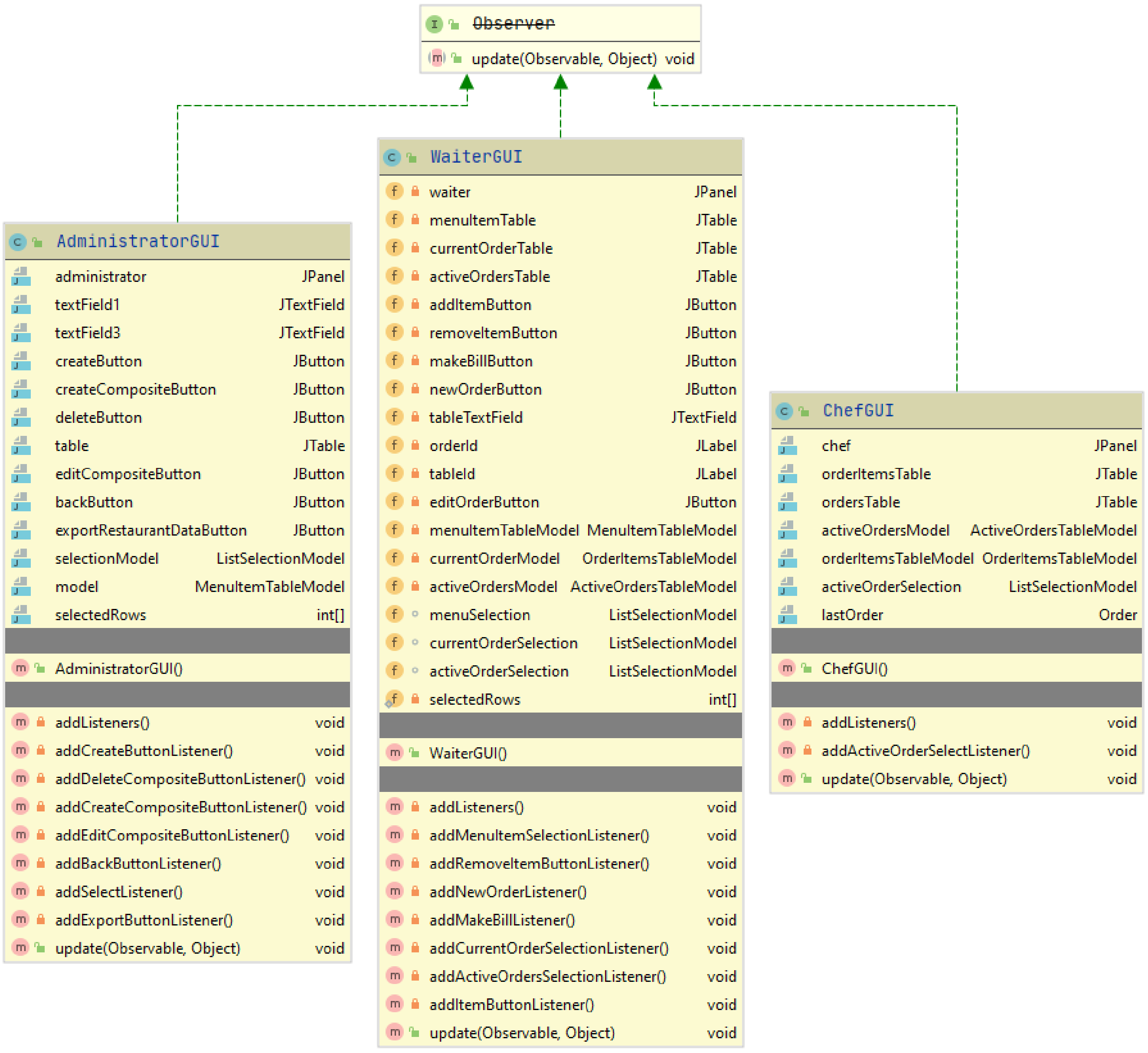
Presentation

The classes that handle or define the graphical user interface can be found in this package. The AdministratorGUI, WaiterGUI and ChefGUI classes define the front-end part of the program. At the same time, they also implement the action listeners that make the program dynamic. Each one of these 3 classes extend the Observer class, so that they know when to update the tables. By doing this, the program runs smoothly and as intended. Three more classes can also be found, that define the JTables behaviours on the GUIs and make the connection between the Front-end and the Restaurant.

Implementation

The core of the whole application is made up from two parts. The custom Table Models, that handle the interaction between the end-user end the tables. These Table Models make a call afterwards to the Restaurant’s methods that handle the data of the restaurant and makes the necessary changes. In case of a change, notifies it’s observers, that are the GUI classes.

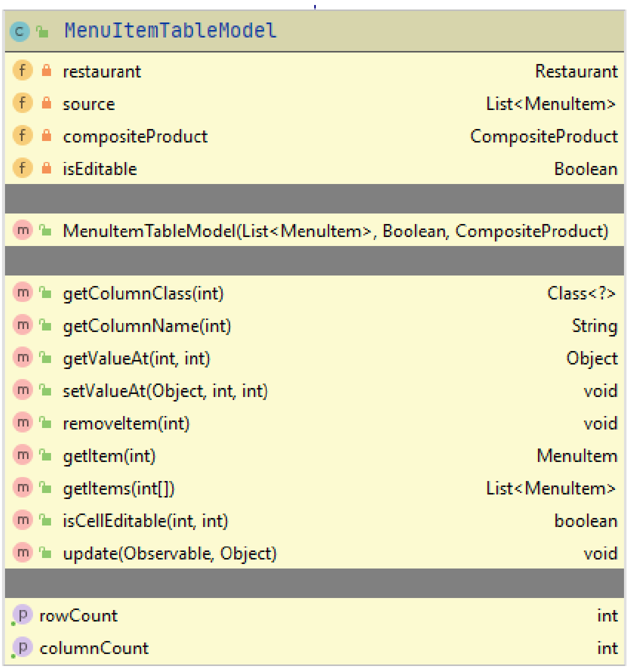
The GUI classes only contain the necessary objects, that are needed to make the front-end easy yet powerfull for every type of

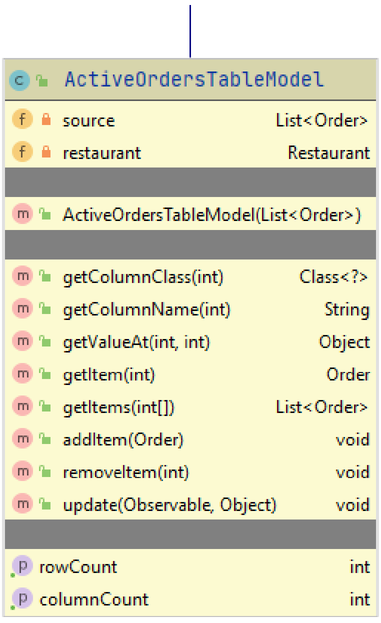


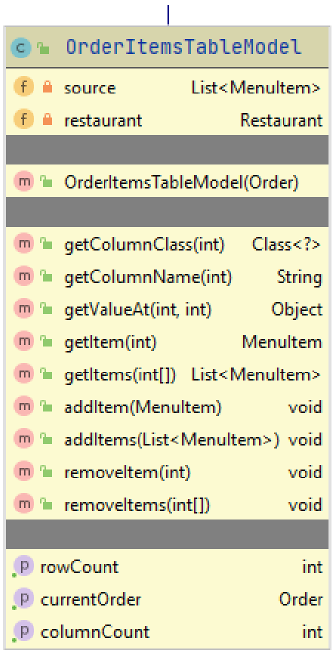
~~Observer~~
update(Observable, Object) void
WaiterGUI
waiter JPanel
menuItemTable JTable
currentOrderTable JTable
activeOrdersTable JTable
addItemButton JButton
removeItemButton JButton
makeBillButton JButton
newOrderButton JButton
tableTextField JTextField
orderId JLabel
tableId JLabel
editOrderButton JButton
menuItemTableModel MenuItemTableModel
currentOrderModel OrderItemsTableModel
activeOrdersModel ActiveOrdersTableModel
menuSelection ListSelectionModel
currentOrderSelection ListSelectionModel
activeOrderSelection ListSelectionModel
selectedRows int[]
WaiterGUI()
addListeners() void
addMenuItemSelectionListener() void
addRemoveItemButtonListener() void
addNewOrderListener() void
addMakeBillListener() void
addCurrentOrderSelectionListener() void
addActiveOrdersSelectionListener() void
addItemButtonListener() void
update(Observable, Object) void
AdministratorGUI
administrator JPanel
textField1 JTextField
textField3 JTextField
createButton JButton
createCompositeButton JButton
deleteButton JButton
table JTable
editCompositeButton JButton
backButton JButton
exportRestaurantDataButton JButton
selectionModel ListSelectionModel
model MenuItemTableModel
selectedRows int[]
AdministratorGUI()
addListeners() void
addCreateButtonListener() void
addDeleteCompositeButtonListener() void
addCreateCompositeButtonListener() void
addEditCompositeButtonListener() void
addBackButtonListener() void
addSelectListener() void
addExportButtonListener() void
update(Observable, Object) void
ChefGUI
chef JPanel
orderItemsTable JTable
ordersTable JTable
activeOrdersModel ActiveOrdersTableModel
orderItemsTableModel OrderItemsTableModel
activeOrderSelection ListSelectionModel
lastOrder Order
ChefGUI()
addListeners() void
addActiveOrderSelectListener() void
update(Observable, Object) void

user. They all implement the update method which are set up specifically for each kind of GUI. As an example, in the AdministratorGUI, the table is only refreshed. The WaiterGUI and the ChefGUI make new Custom Table Models on each update call, due to the fact that the list of the Menu Items for each order can only be taken as a new Object. So they are in no relationship with the original one. That’s why it must reload the whole model for the JTables.

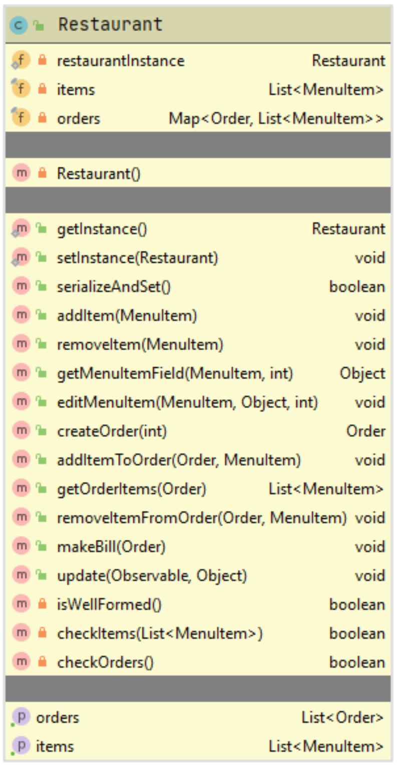
The custom Table Models were needed, so that editing the items can be made in-place (inside the table, by editing the cells of the table). The methods getValueAt and setValueAt were the real reason, these Models were implemented. They use reflection to get the values of a given object and also to set them. The only method that implements the setValueAt is the MenuItemTableModel, because editing should be only allowed for the administrator. This can be set in the isCellEditable method, which takes the isEditable variable to decide if it's editable or not. The first 2 columns are by default un-editable (ID and Type) and the price of CompositeProducts.


MenuItemTableModel
restaurant Restaurant
source List<MenuItem>
compositeProduct CompositeProduct
isEditable Boolean
MenuItemTableModel(List<MenuItem>, Boolean, CompositeProduct)
getColumnClass(int) Class<?>
getColumnName(int) String
getValueAt(int, int) Object
setValueAt(Object, int, int) void
removeItem(int) void
getItem(int) MenuItem
getItems(int[]) List<MenuItem>
isCellEditable(int, int) boolean
update(Observable, Object) void
rowCount int
columnCount int


ActiveOrdersTableModel
source List<Order>
restaurant Restaurant
ActiveOrdersTableModel(List<Order>)
getColumnClass(int) Class<?>
getColumnName(int) String
getValueAt(int, int) Object
getItem(int) Order
getItems(int[]) List<Order>
addItem(Order) void
removeItem(int) void
update(Observable, Object) void
rowCount int
columnCount int



The brain of the application is the Restaurant class, that decides, how is every data handled. First of all, it uses the Singleton Pattern, that won't let the application to make 2 instances of the restaurant.


Restaurant
restaurantInstance Restaurant
items List<MenuItem>
orders Map<Order, List<MenuItem>>
Restaurant()
getInstance() Restaurant
setInstance(Restaurant) void
serializeAndSet() boolean
addItem(MenuItem) void
removeItem(MenuItem) void
getMenuItemField(MenuItem, int) Object
editMenuItem(MenuItem, Object, int) void
createOrder(int) Order
addItemToOrder(Order, MenuItem) void
getOrderItems(Order) List<MenuItem>
removeItemFromOrder(Order, MenuItem) void
makeBill(Order) void
update(Observable, Object) void
isWellFormed() boolean
checkItems(List<MenuItem>) boolean
checkOrders() boolean
orders List<Order>
items List<MenuItem>

Contains basic functions like adding an item to an order, removing an item from an order, making a bill, adding a new item to the menu, etc. Besides these functions, it also checks before and after each method that was ran if it's well formed or not. This means, that everything is all right and nothing unexpected happened. Every function that is overriden from the IRestaurantProcessing interface implements the Design by Contract pattern. Checks the incoming variables for correctness and also the outcome of the method.

Results

As results, the end-user and his/ her team gets a fully functioning restaurant management system that should satisfy the needs of any smaller size restaurant or maybe even medium size if it's well coordinated.

Conclusions

I have learnt how to handle JTables and most importantly how to make custom Table Models that does enough for our needs.

Serialization is a very useful thing in my opinion, because it helps the developer or even the end-user if it is implemented well, to move data or even store data for different runs of a program.

Bibliography

- https://dzone.com/articles/composite-design-pattern-in-java-1
- https://www.tutorialspoint.com/java/java_serialization.htm