1. Description of the assignment

We are desired to implement 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.

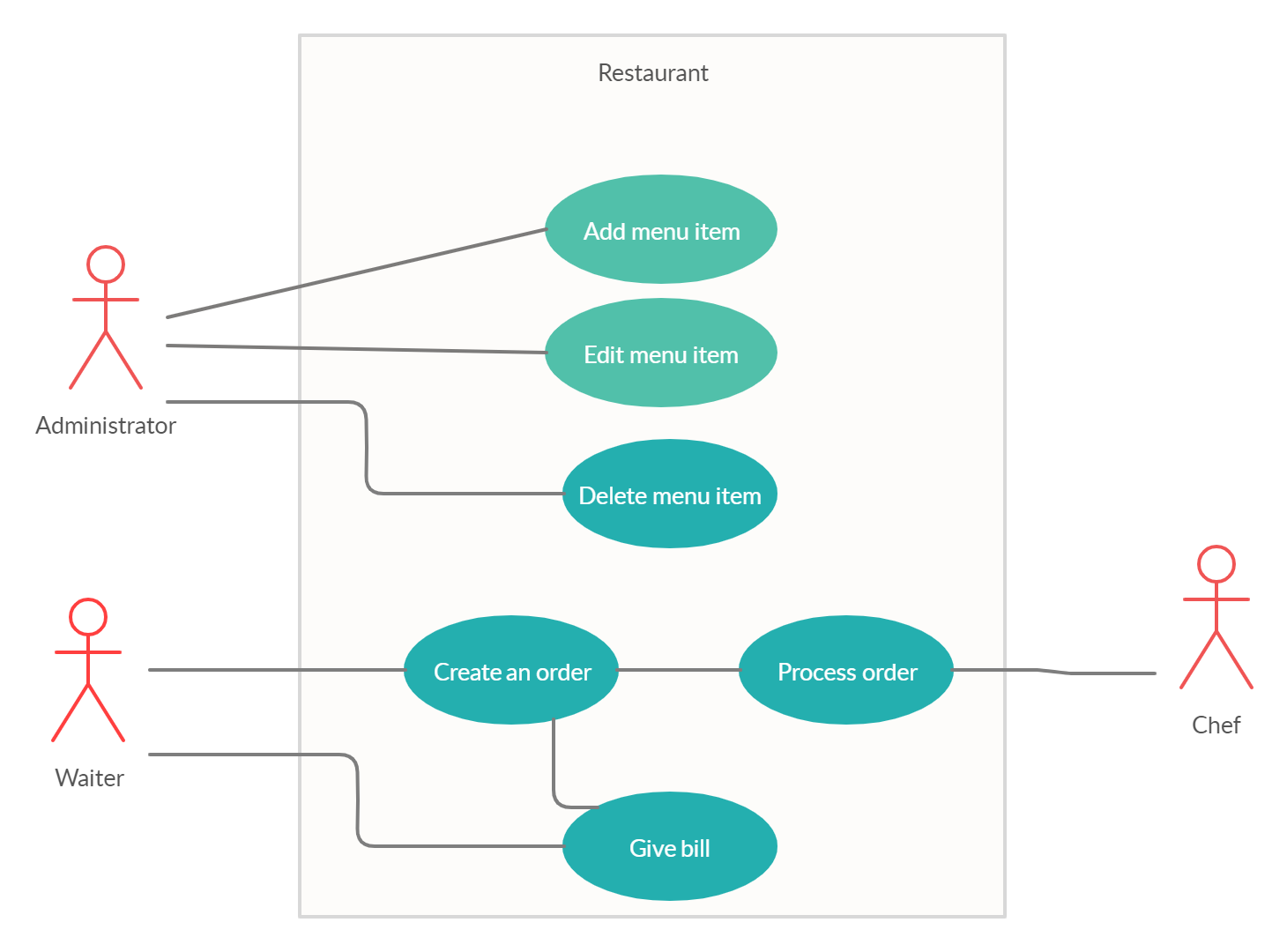
In order to do that, I followed the steps presented in the assignment support file handled to us by our professor and so I did the following (most of them will be presented in detail later into the presentation):

* Defined the interface IRestaurantProcessing containing the main operations that can be executed by the waiter/administrator;
* The administrator, using the specific GUI, can create a new menu item(either Base Product or Composite Product), delete a menu item(deletion works only for Base Products, this is something I should improve about the project) and edit a menu item (for my application I chose to edit only the compose products, because I found it a little bit difficult to edit the base products, because that meant modifying each composite product that the modified base product was part of; to improve my application I will think of ways in which to implement the edit for a base product also);
* The waiter, using its’ specific GUI, can create a new order (which implies computing its final price) and generate a bill in .txt format for that specific order;
* Defined the classes shown in the diagram that accompanied the project presentation
* Used the Composite Design Patter for defining the classes MenuItem, BaseProduct and CompositeProduct, as the rest of the classes will use the component class interface (MenuItem) to interact with objects in the composition structure (BaseProduct & CompositeProduct);
* Used the Observer Design Pattern to notify the chef each time a new order is added;
* Implemented 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 MenuItem objects
* Use JTable to display Restaurant related information (the items in the menu, their price and their type);
* Defined a structure of type Map<Order, Collection<MenuItem>> for storing the order related information in the Restaurant class.
* The items in the menu are stored in a Collection<MenuItem> object;
* Defined a “well defined” method to ensure at each step that the restaurant is in a good form (ex: the order placed is non empty);
* Implemented the Restaurant class using Design by Contract (I used assertions to test);
* Saved to a file using Serialization, to ensure that the state of the restaurant is saved even though the application may be closed.
* Designed 3 GUI for administrator, waiter and chef (they will be started at the same time when the application runs)
* Provided JavaDoc files for the class Restaurant and the interface that it implements
* Provided the .jar file

1. Problem analysis, problem modeling, scenarios, utilization cases

As the diagram already shows, the application will be mainly divided into 3 packages(DataLayer, BusinessLayer and PresentationLayer); As shown in the first assignment, the presentation layer will cover everything that has to do with input/output (in this case just the output as we don’t have anything in the first place when the application is run), the data layer would contain the model of the application and the business layer would link the `exterior` and the `interior` of the program.

To show better how the program is supposed to work, I will present some use-case diagrams next (they will be modeled as lists, showing the steps involved in the execution of each case);



Use Case: managing the stock of the restaurant

Primary Actor: Administrator

Main Success Scenario:

-The administrator chooses to add several Base Products

-The administrator chooses to create a Composite Product using the Base Products that are in stock

-The administrator edits the name of one of the Composite Products

-The administrator deletes a Base Products from the stock

Use Case: process of taking a client’s order

Primary Actor: Waiter

Main Success Scenario:

-The waiter takes the order from the specific table

-The order contains only the products in the restaurant’s menu

-The order contains at least one menu item

-The waiter instructs the chef about the order and then proceeds to give the bill for the items ordered

Use Case: process of receiving an order

Primary Actor: Chef

Main Success Scenario:

-The chef successfully receives the order details from the waiter and proceeds the order

Use Case: managing the stock of the restaurant

Primary Actor: Administrator

Alternative Sequences:

-The editing of the product resulting in changing the number of menu items (which will signal that the editing was not successful)

-Deleting a Base Product but all the Composite Products containing that product still being in the menu (this is something that my application supports, though I should work on improving this part of the program)

Use Case: process of taking a client’s order

Primary Actor: Waiter

Alternative Sequences:

-The client has an order that includes items that are not on the current menu

-The total price of the order is equal to 0, which means no items have been added to the order

Use Case: process of receiving an order

Primary Actor: Chef

Alternative Sequences:

-The waiter can’t transfer the order information to the chef, thought he doesn’t receive any orders to process

1. Project design (design decisions, UML diagrams, data structures, classes design, relationships, interfaces, packages, algorithms, graphical user interface)

In the following chapter I will discuss how I have split the problem into an object oriented one and the data structures I used for implementing the project, alongside the UML diagrams specific to this application.

As I already previously mentioned, I divided my project in 3 packages, as the support diagram suggested: DataLayer, PresentationLayer and BusinessLayer.

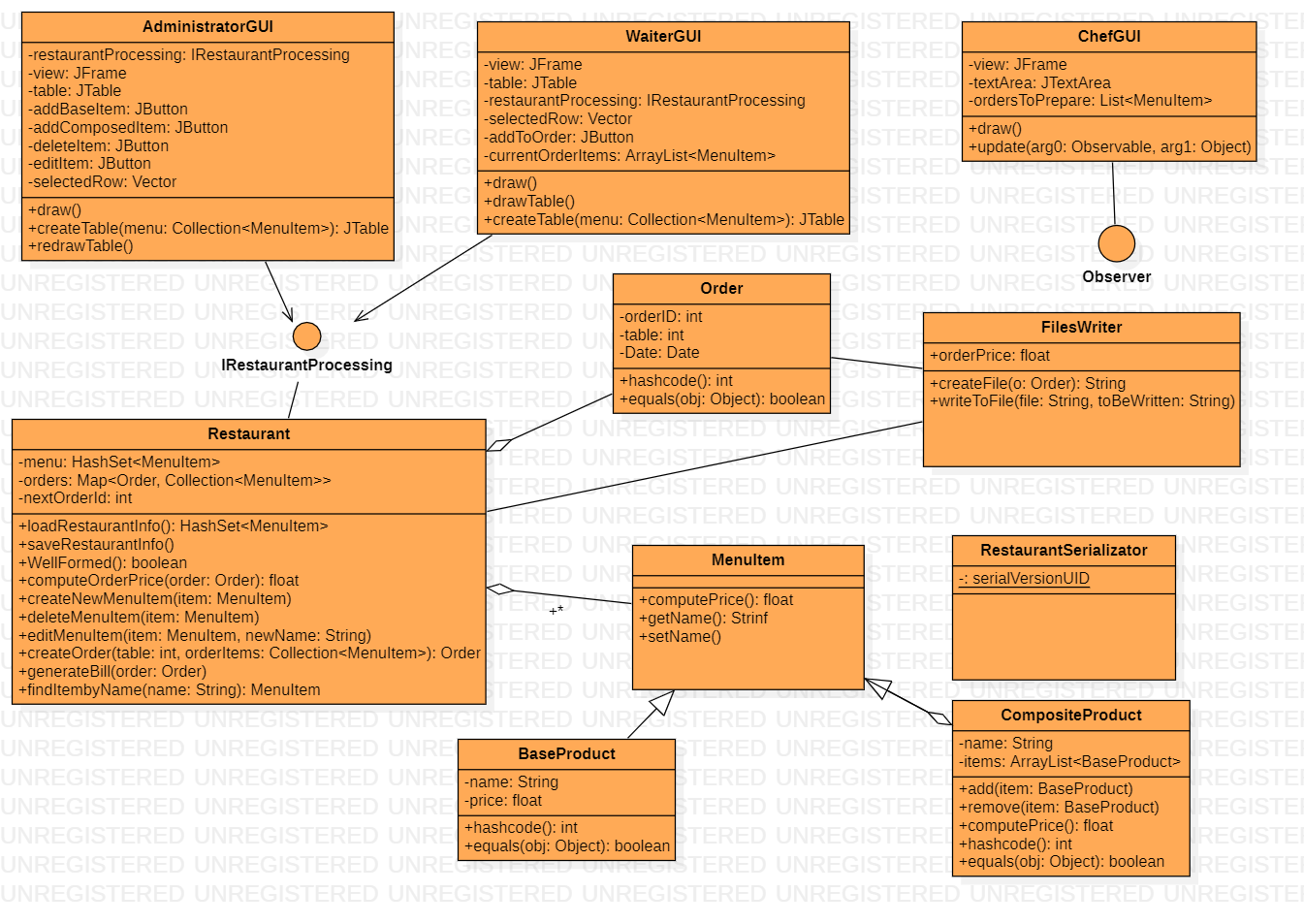
Besides these packages, the Main function is in the co.com package provided when I started the new application.

-Presentation package contains the classes that would present the output, meaning the GUI classes for administrator, waiter and chef. All these three GUIs will be started at the same time when the application is run, meaning that 3 different windows will start. Closing one of them will end the application.

-Business package contains the main classes that we will work with, connecting the orders with the menu items through the Restaurant class and linking everything with the output classes from the Presentation package.

-Data package contains the main function for writing into an external file and the main Serialization mechanism.

Next, I will attach the UML diagrams specific to my project, along with the relationships between them (the following UML diagram is made using the StarUML app); it should look like the diagram in the support presentation:



1. Implementation

In this chapter I will elaborate on each class I created, explaining the most important methods that each implement and why I chose them the way I did.

* BaseProduct class

This class represents all the object that will be taken as simple product. Using the objects of this class we will be able to create a CompositeProduct. Because these items are part of the menu, they should be taken into consideration when the serialization process is happening, that’s why the line

**private static final long serialVersionUID = 1L** should be included.

The objects of this class will be defined only by a name and a price, and the methods are the getters and setters accompanied by the hashcode() overridden method

**{return name.hashCode();}**

and the equals() overridden method which will compare the object regarding their name

**{**

**if (!(obj instanceof BaseProduct))  
 return false;  
  
BaseProduct other = (BaseProduct) obj;  
return this.getName().equals(other.getName());**

**}**

* CompositeProduct class

These objects will represent the items composed from other BaseProduct objects. Just like the items presented before, these too contain the variable for serialization, because the state of the menu has to be preserved. Unlike the BaseProduct objects, the objects of this class are characterized by a name and a list containing the Base Products used to compose them. The methods of this class are composed of:

add(): adding a base product to the list; remove(): removing a base product from the list; computePrice(): calculate a total price taking each base product from the list and summing their price. Just like the class presented before, this one too has the hashcode() overridden method, as well as the equals() one.

* Order class

The objects of this class will represent the orders. An order is defined by a unique ID, the date when it was released and the table number to which it is linked. The only method present in this class are the getters and setter, as well as the hashcode() and equals(). Because the variables present in this class are not one of them a string, the hash code is computed using a string composed of all of the variables in the following way:

**String str = orderID + "\_" + Date.toString() + "\_" + Table;**

Also, the equals() method no longer uses a string to compare, but the ID of each order.

* Restaurant class

The Restaurant class will contain the menu as a HashSet<MenuItem>, the current orders as a Map<Order, Collection<MenuItem>> and also the nextID for the orders. The constructor of this class, besides creating a new HashMap for the current orders and attributing the first order ID=1, it should load the menu, using serialization methods ( that’s why the constructor contains **menu = loadRestaurantInfo();** )

This method, loadRestaurantInfo() uses FileInputStream, ObjectInputStream objects to get the information saved in a file name “restaurant.ser” and load it into the current menu.

Similarly, the method saveRestaurantInfo() uses a FileOutputStream, ObjectOutputStream objects to save the current menu items into the same file. This can be associated to the serialization process, just like the previous method can be associated with the deserialization process.

The method WellFormed() is a verification process, to make sure the changes made, especially when talking about a new order, don’t affect the state of the restaurant. For that, I chose 2 verifications:

**for (MenuItem item : orderItems)  
 if (!menu.contains(item)) {  
 OK = false;  
 break;  
 }**

**-**this one assures us that the items in the order are in fact in the menu- **if (orderItems.size() < 1) {  
 OK = false;  
 break;  
}**

**-**this one assures us that the order contains at least one item-

The method computeOrderPrice(Order order) is similar to the one that computes the price for a composite product, but this time for the given order, the sum of all products’ prices is computed.

The method createNewMenuItem(MenuItem item) adds a new item to the menu list. After adding it, we should call the “well formed” method to make sure the state of the restaurant is not affect and also save the item on the menu using serialization.

The method deleteMenuItem(MenuItem item) deletes a menu item from the current list. After the deletion, the state of the restaurant is checked again and the serialization is done.

The editMenuItem(MenuItem item, String newname) method is used to change the name of a product (I chose to change the name only for a composite product, as I earlier stated). Firstly, we should verify if the product is on the list and then remove it if it is; after removal, we change its’ name and add it again. The state of the restaurant should be checked and the serialization done.

The createOrder(int table, Collection<MenuItem> orderItems) method creates a new order, first making sure the items in the order are on the menu items list. If they are, a new order is created, which has assigned the ID from the class’ variable, which is then increase, the table number given as a parameter and the current date

**Order newOrder = new Order(nextOrderId, new Date(), table);  
nextOrderId++;  
orders.put(newOrder, orderItems);**

Besides the checking of the restaurant’s state, we should now use Observable methods to inform the chef about the new order(more so about the new order items)

**setChanged();  
notifyObservers(orderItems);  
clearChanged();**

The method generateBill(Order order) creates a .txt file containing the details about the given order, like the table number, order number, date of the order, ordered items and the total price. It uses an object of the class FilesWriter which will be presented a little bit late, and with its’ help it writes the bill.

The method findItemByName(String name) returns the item that has the specific name given as a parameter.

* IRestaurantProcessing interface

This interface presents all the methods explained before in the Restaurant class. It will be used in the GUIs to access all these methods.

* MenuItem interface

It is used to generalized a menu item, making it easier to refer to both base products and composite products at the same time. It contains the methods computePrice(), getName() and setName() implemented by both classes that implement the interface.

* RestaurantSerializator

This class has only one variable, which will be used for the serialization. It extends the exception class, being used in connection with the “well defined” method from the Restaurant class.

* FilesWriter

This class gets a variable orderPrice and it uses two methods ( createFile(Order o) & writeToFile(String file, String tobeWritten) ) in order to open a new output file and write in it. The methods are similar with the ones I used in the second assignment: for the creation of the new .txt file I used an object of type File, while for the writing process I used

**BufferedWriter output = new BufferedWriter(new FileWriter(file, true));  
output.append(toBeWritten);  
output.append('\n');**

to make sure each of the information given is written on a different line.

* ChefGUI

The class presents the graphical user interface characteristics for the chef. Because no explicit actions were suggested in the presentation regarding the chef, I chose to show just the current items from the orders that need to be prepared by the chef. For that, I used a JTextArea variable and a StringBuilder variable, to which I’m going to append each name of the items from the List<MenuItem>, each menu item on a different line

**for (MenuItem menuItem : ordersToPrepare) {  
 sp.append(menuItem.getName());  
 sp.append("\n");  
}  
textArea.setText(sp.toString());**

Because the class implements the interface Observer, the method update(Observable object1,Object object2) should be overridden, because each time a new order is placed, it should reach the chef’s `to do list`. So, for the object we will choose the orders, adding them to the list of orders already taken by the chef; then, doing the same thing as before, using a StringBuilder we will go through all the orders now correctly placed and display each of them in the text area we defined. By doing this with the function update(), we will now be sure the chef always is noted about the new orders coming in.

* WaiterGUI

For the waiter’s graphical interface, we should have a table with all the menu items available, so that we can choose from these items the ones we want to use for the order. For doing that, two buttons are needed, one to add to the order an item and one for the action of saving the final order and giving the clients their bill. The tool that will help us link the GUI with the internal construction of the application is the interface IRestaurantProcessing mentioned earlier, so we should have one variable of this type so we can easily access its methods.

Also, for an order to be complete we also need a table number, which I thought could be introduced manually by the waiter through a text box. To make sure the items from an order are correctly positioned and not one is missing, I displayed a text box where the items will appear gradually, one by one, when the button addToOrder is pressed.

**MenuItem menuItem = restaurantProcessing.findItembyName((String) selectedRow.get(0));  
textArea.setText(textArea.getText() + "\n" + menuItem.getName());  
currentOrderItems.add(menuItem);**

We first select a row in the table, and then call the Vector’s method .get(0) to get the first part (the string containing the name of the product) in order to use it in the findItembyName function mentioned earlier. Using that, we will search in the menu for the item with that specific name and return it into an MenuItem object that will be inserted in the text area destined for the items of the order and also added into the list of current orders.

For the other button, “Save & Give Bill” the order will be placed and the chef will be notified as shown in the description of the createOrder() method in the Restaurant class. The text inserted in the “Table” text box will be taken and used for the order, as well as the items displayed in the order items structure. The method generateBill() will also be called and the text area for the order items will be cleared, meaning that the order was placed and a new order can come next. Also, all the currentOrderItems will be deleted, because the order was placed.

**Integer tableNumber = Integer.*parseInt*(tabletxt.getText());  
Order o = restaurantProcessing.createOrder(tableNumber, currentOrderItems);  
restaurantProcessing.generateBill(new Order(o.getOrderID(), o.getDate(), o.getTable()));  
textArea.setText("");  
currentOrderItems.removeAll(currentOrderItems);**

One more method used is the one that creates the table for the current menu items in the restaurant’s stock. For that, we will use an object of type DefaultTableModel to which we will give as parameters two arrays containing data. The header array will contain string that will be the “head” of the columns

**Object[] header = { "Name", "Price", "Type of product" };**

The data array will contain the data of the array, giving it the values as follows

**for (int i = 0; i < array.length; i++) {  
 Object[] menuRow = new Object[3];  
 MenuItem item = array[i];  
 menuRow[0] = item.getName();  
 menuRow[1] = item.computePrice();  
 if (item instanceof BaseProduct)  
 menuRow[2] = "base";  
 else   
 menuRow[2] = "composite";  
   
 data[i + 1] = menuRow;  
}**

The first value should be the name of the product, the second should be its price and the last one should display the message “base” if the product is a base product or “composite” otherwise. The array is filled from the index 1 onward because index 0 is reserved for the header filling. To add the property of selecting a row we add

**table.getSelectionModel().addListSelectionListener(new ListSelectionListener() {**

**public void valueChanged(ListSelectionEvent e) {  
 selectedRow = (Vector) model.getDataVector().get(table.getSelectedRow());  
 }  
});**

This ensures that the parameter of the class ChefGUI named selectedRow gets the value from the selected row. That parameter was used earlier, when we added the orders by manually selecting them and then pressing the button for adding the order.

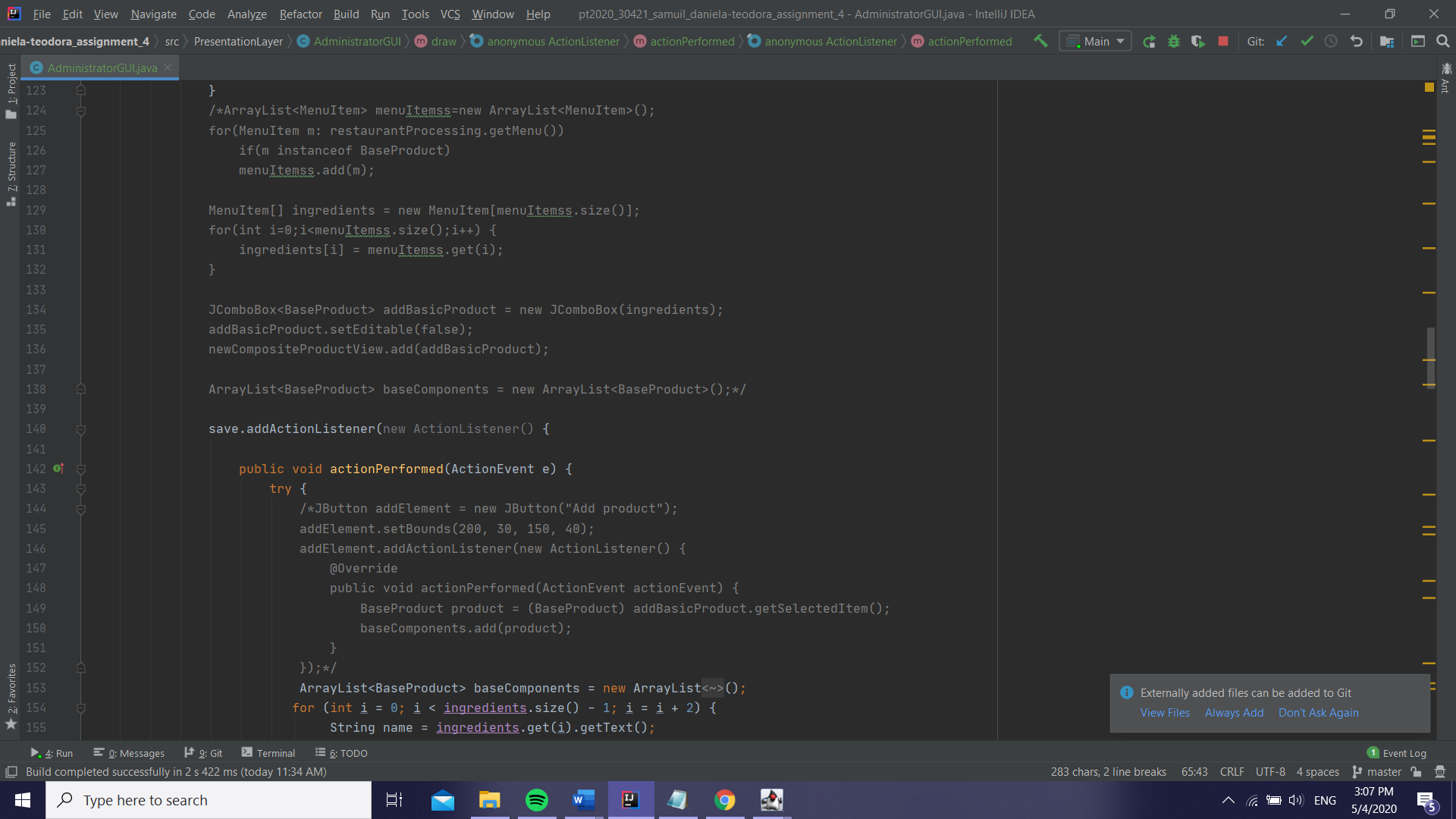
* AdministratorGUI

This class is the most complex one from all presented so far, because it should manage the menu items, which has a few possibilities: adding a base product, adding a composite product, editing a product or deleting a product. So, we will need 4 buttons, each for one of the above-mentioned actions. For the creation of new items, once the buttons are pressed, a new window will open up suggesting the fields we should complete in order to add the item. For example, once “Add Base Product” is pressed, a new window will pop up, which asks for a name input and a price input in two text boxes. Once the name and the price are inserted, the button “Save product” should be pressed in order to save the item in the menu and to close the current window. Because the new item was added, the table containing the items should be redraw

**restaurantProcessing.createNewMenuItem(new BaseProduct(nametxt.getText(), Float.*parseFloat*(priceTxt.getText())));  
newBaseProductView.dispatchEvent(new WindowEvent(newBaseProductView, WindowEvent.*WINDOW\_CLOSING*));  
redrawTable();**

As we can observe, the method from the Restaurant class is used again, so we will use an object of type IRestaurantProcessing just like in the waiter’s case.

The same kind of window will pop up when we press the “New Composed Item”, but this time we will see a text box for the name and 6 other text boxes under the tag “Ingredients”. Because as we already know, a composite product is made from base product, I chose such an item to be composed of three base components which are going to be defined in the 6 cells (3 for each name, 3 for each price). The button “Save product” will work just like in the previous case. Because of the current way the interface is looking, it is a little harder to create a composite product because we have to manually add the items that make it, and so we have to be really careful when introducing the name and price for each of them. I tried to solve this problem by introducing a Combo Box where we can choose the base products from, but unfortunately it didn’t work out well, I will definitely work on this issue a little bit more (I will attach a screenshot of the code that should implement the Combo Box alternative, though it is commented out in my application because the combo box wasn’t showing on the graphical interface when I first implemented it and I couldn’t quite figure out why). But that could be one major improvement to this project, the combo box providing a simpler method of choosing the base products that compose the composite product, ensuring at the same time that we don’t insert a wrong product (thing which can happen in my version of implementation).



For my approach, we go through the written fields and select the parts of them that we need and assert their values to an object of type BaseProduct that is then added to the list which will be given as a parameter to the creation of the new composite product. I also attached a condition verification, meaning that we should see if the menu already has the product we want to insert into the composition of the newly created product.

**for (int i = 0; i < ingredients.size() - 1; i = i + 2) {  
 String name = ingredients.get(i).getText();  
 float price = Float.*parseFloat*(ingredients.get(i + 1).getText());  
 BaseProduct b = new BaseProduct(name,price);  
 if(restaurantProcessing.getMenu().contains(b))  
 baseComponents.add(b);  
 else  
 System.*out*.println("Product is not in the menu");  
 }**

Where `ingredients` is a list of JTextField elements, taking each string we input into the text box and adding it to the list before going through it. The step is taken as i+2 because we have 2 fields (name and price) for each base product. For each of these 2 fields, we create a new BaseProduct object setting the parameters converted from the inputted strings. After we create the new product, we should test, as I previously mentioned, if it is contained in the current menu or not, else printing an error message.

After the list of base products is filled, we should create the new menu item using the method from the Restaurant class, close the current window for the creation of the element and redraw the menu items table including the new added item.

**restaurantProcessing.createNewMenuItem(new CompositeProduct(nametxt.getText(), baseComponents));  
newCompositeProductView.dispatchEvent(new WindowEvent(newCompositeProductView, WindowEvent.*WINDOW\_CLOSING*));  
redrawTable();**

In order to edit an item (in my case edit is available just for the composite products as I’ve said before, but this should be an issue that can be approached in order to improve the project), We should select first the item we want to edit and then, when the button is enabled, press it in order to open a new window with the editing details.

**editItem = new JButton("Edit Menu Item");**

**editItem.setEnabled(false);**

First, we need to make sure that the object selected is a composite item, so we test it. Then, because the window is similar to the one from the add method, it will have the same fields, one for the name and 3x2 for the ingredients (base products). The way we will edit the item is by using again a method from the Restaurant class, but in order to do this we need to remember the name of the product, because the method uses the findItembyName() method presented previously.

**String oldName=nametxt.getText();**

-nametxt being the name text field for the composite product, defined as menuItem.getName() after we select the specific row-

**restaurantProcessing.editMenuItem(new CompositeProduct(oldName, baseComponents),nametxt.getText());  
editView.dispatchEvent(new WindowEvent(editView, WindowEvent.*WINDOW\_CLOSING*));  
redrawTable();**

The composite item that needs to be replaced will have the old name and the text inserted by us will be the new string that defines the name given after the change. Once again, the table showing the items in the menu should be modified.

To delete an item, we once again need to select the desired item and then press the button. We will call one function defined in the IRestaurantProcessing interface once more, testing first what kind of item we have (base or composite). The problem with my function is that, in case of a base product that is deleted, the application will not delete also the composite product that contained that specific item. Though, the function still needs some improvements.

**if(menuItem instanceof BaseProduct)  
restaurantProcessing.deleteMenuItem(new BaseProduct((String) selectedRow.get(0), (Float) selectedRow.get(1)));  
else  
 if(menuItem instanceof CompositeProduct)  
 restaurantProcessing.deleteMenuItem(new CompositeProduct((String) selectedRow.get(0),(ArrayList<BaseProduct>) selectedRow.get(1)));  
redrawTable();**

We once again have the createTable() function that I explained earlier in the WaiterGUI section. One new method is the one that redraws the table. First, it needs to remove the current table

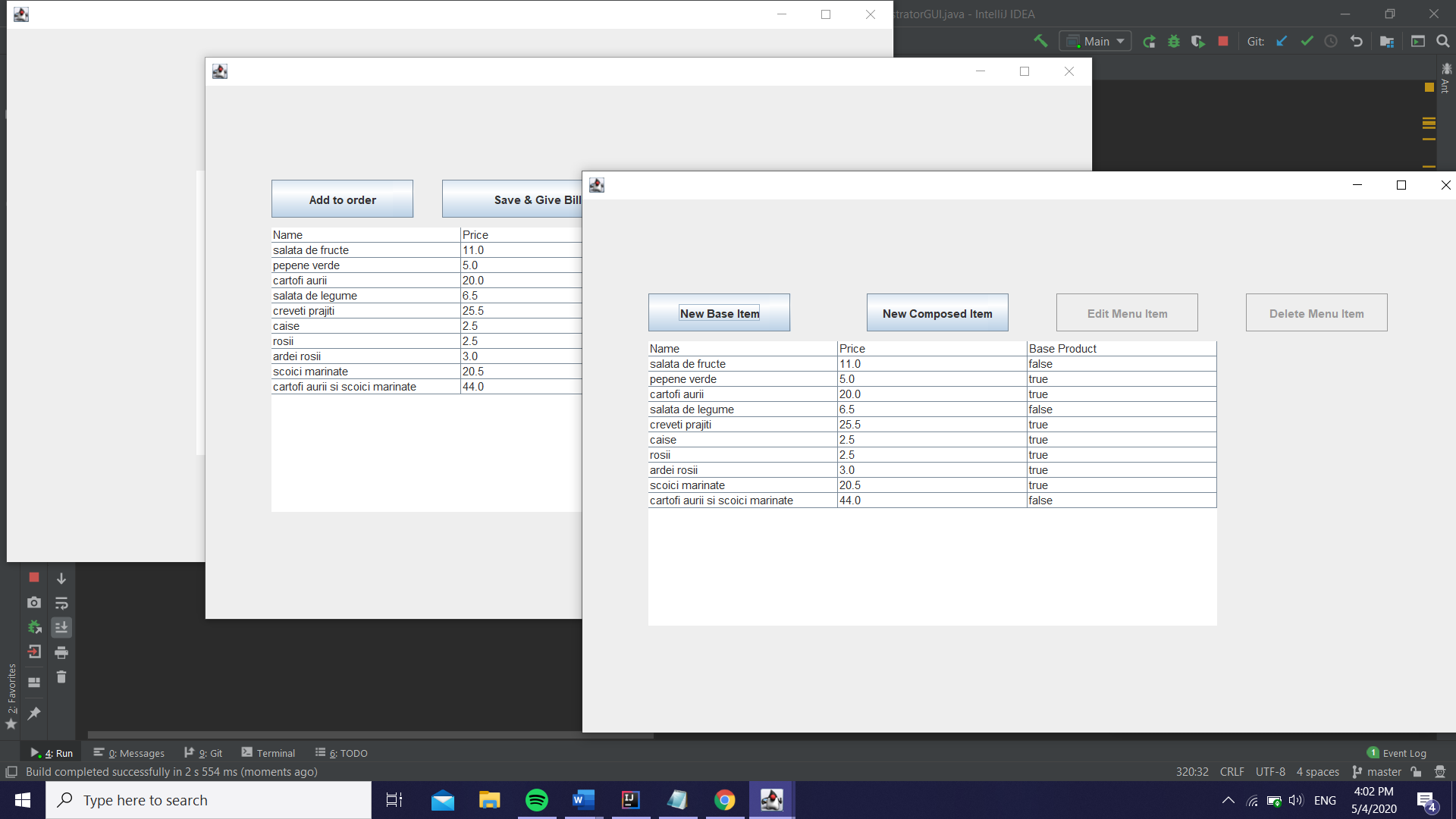
**if (view.isAncestorOf(table))  
 view.remove(table);**

Then we should add the new table from the updated menu and get the view to the final form

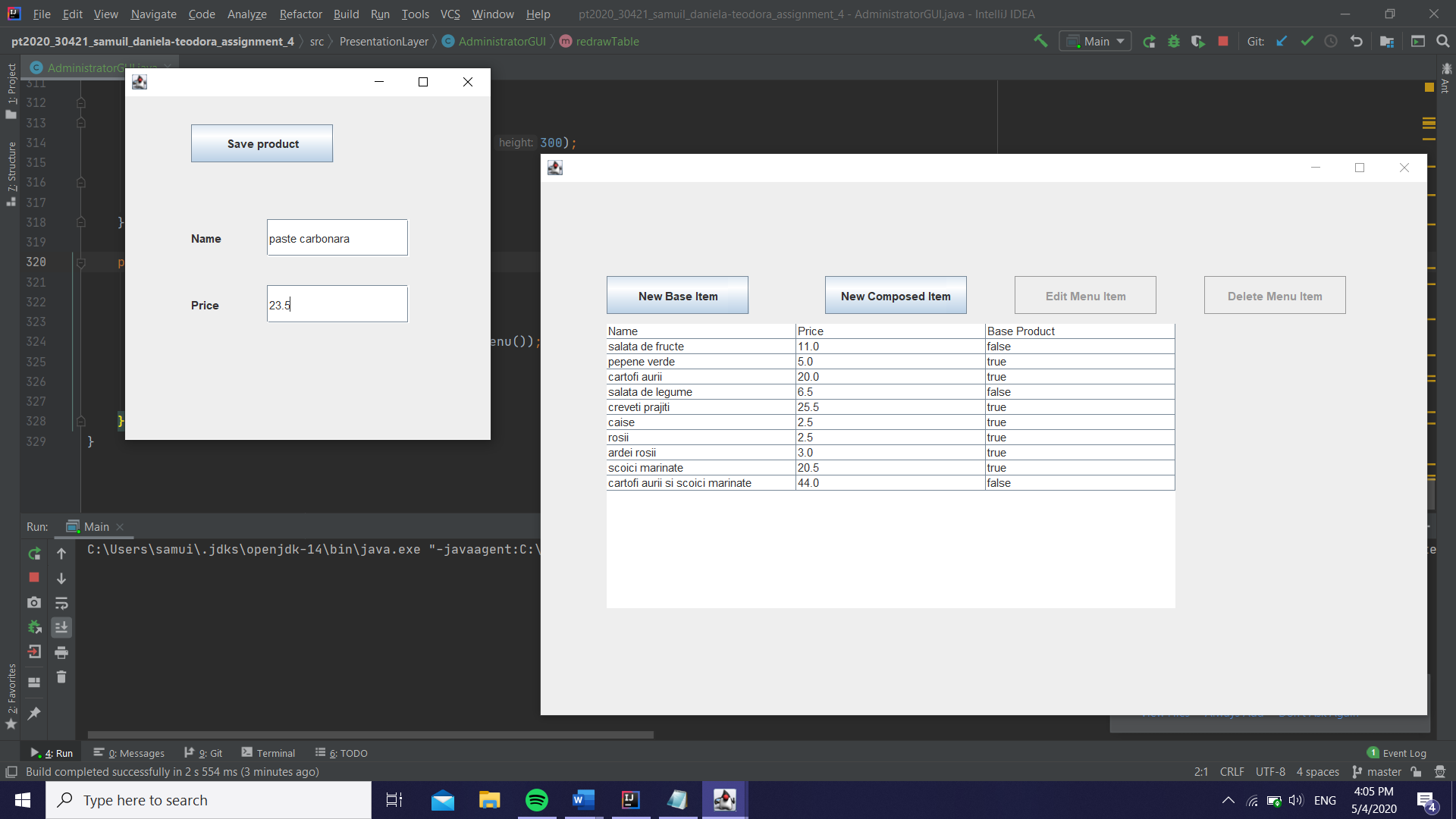
**table = createTable(restaurantProcessing.getMenu());  
if(table!=null)  
view.add(table);  
SwingUtilities.*updateComponentTreeUI*(view);**

1. Results

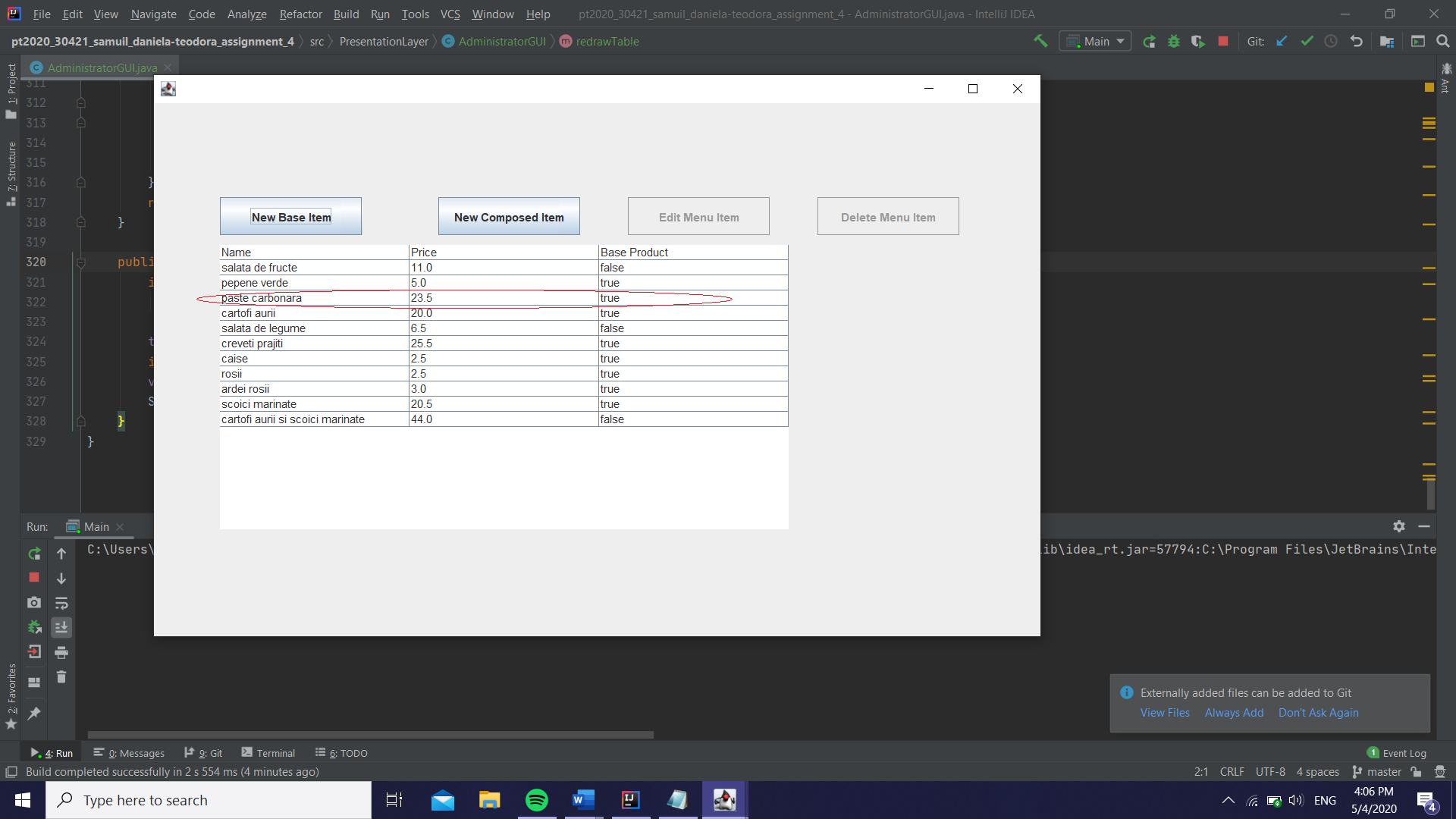
I will further attach some screenshot taken during the running time of the application.

The three GUIs open up at the same time when the program is run.

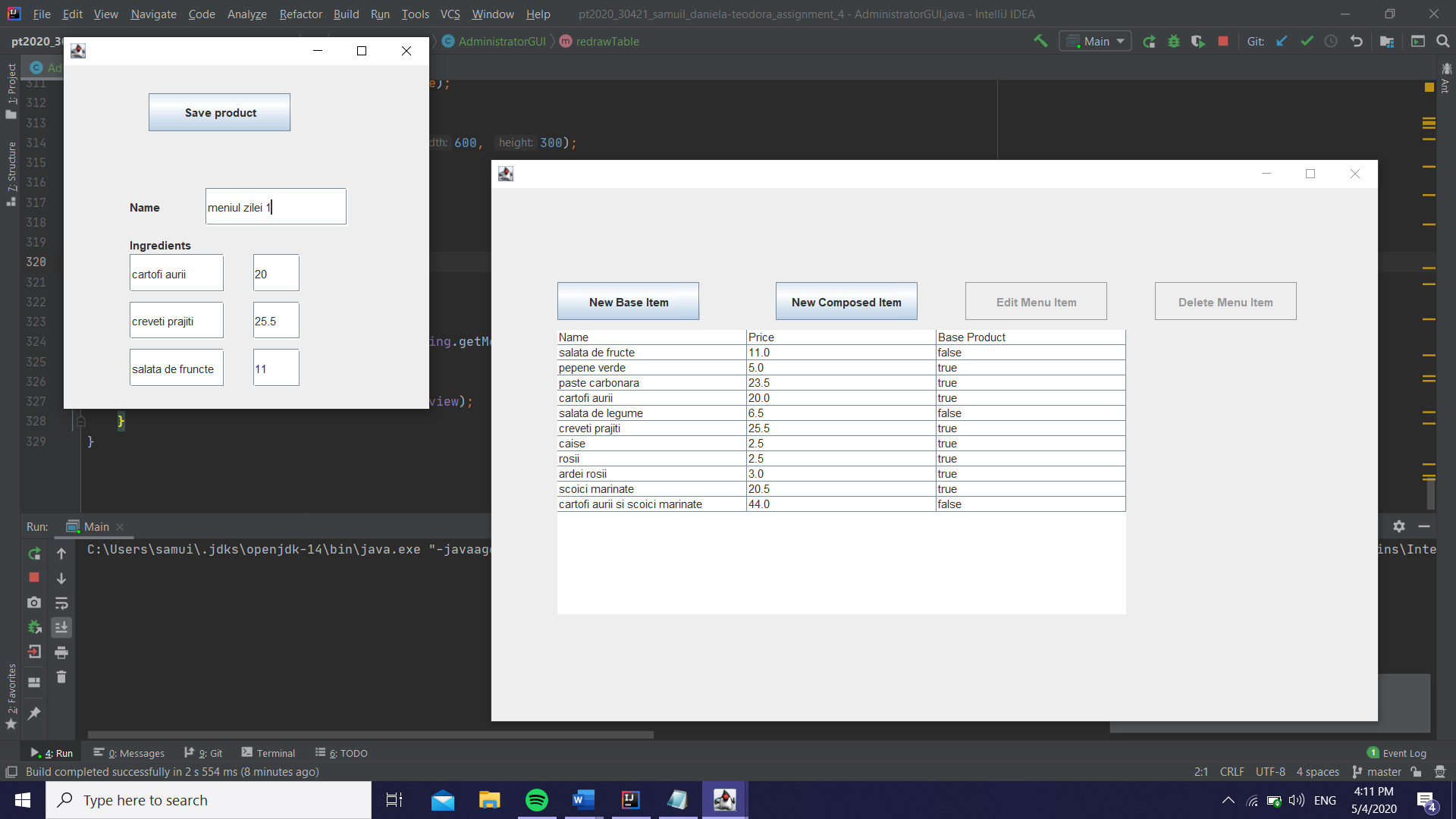
Also, the menu is automatically loaded when the application starts, due to the serialization.



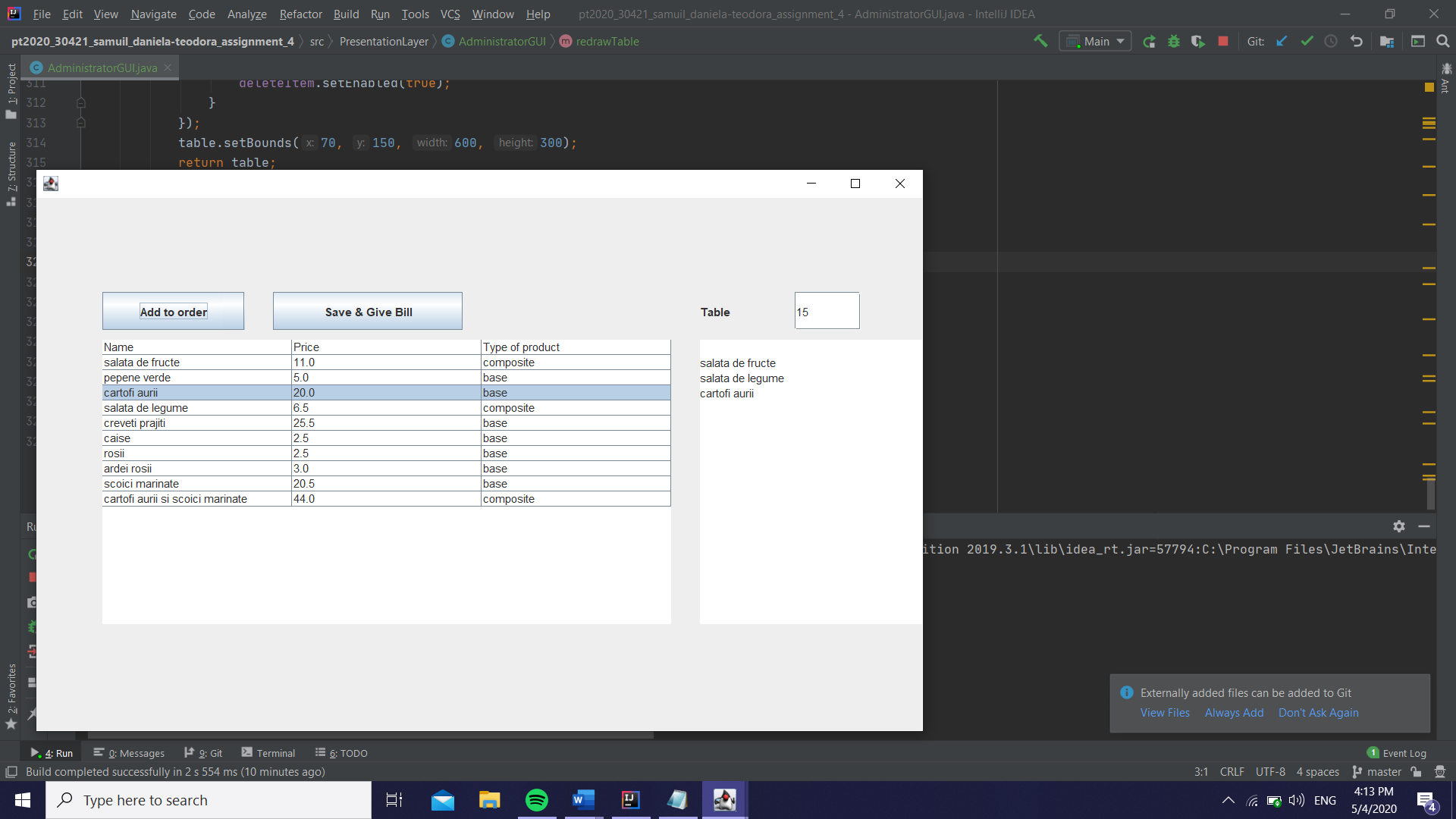
Creating a new base product



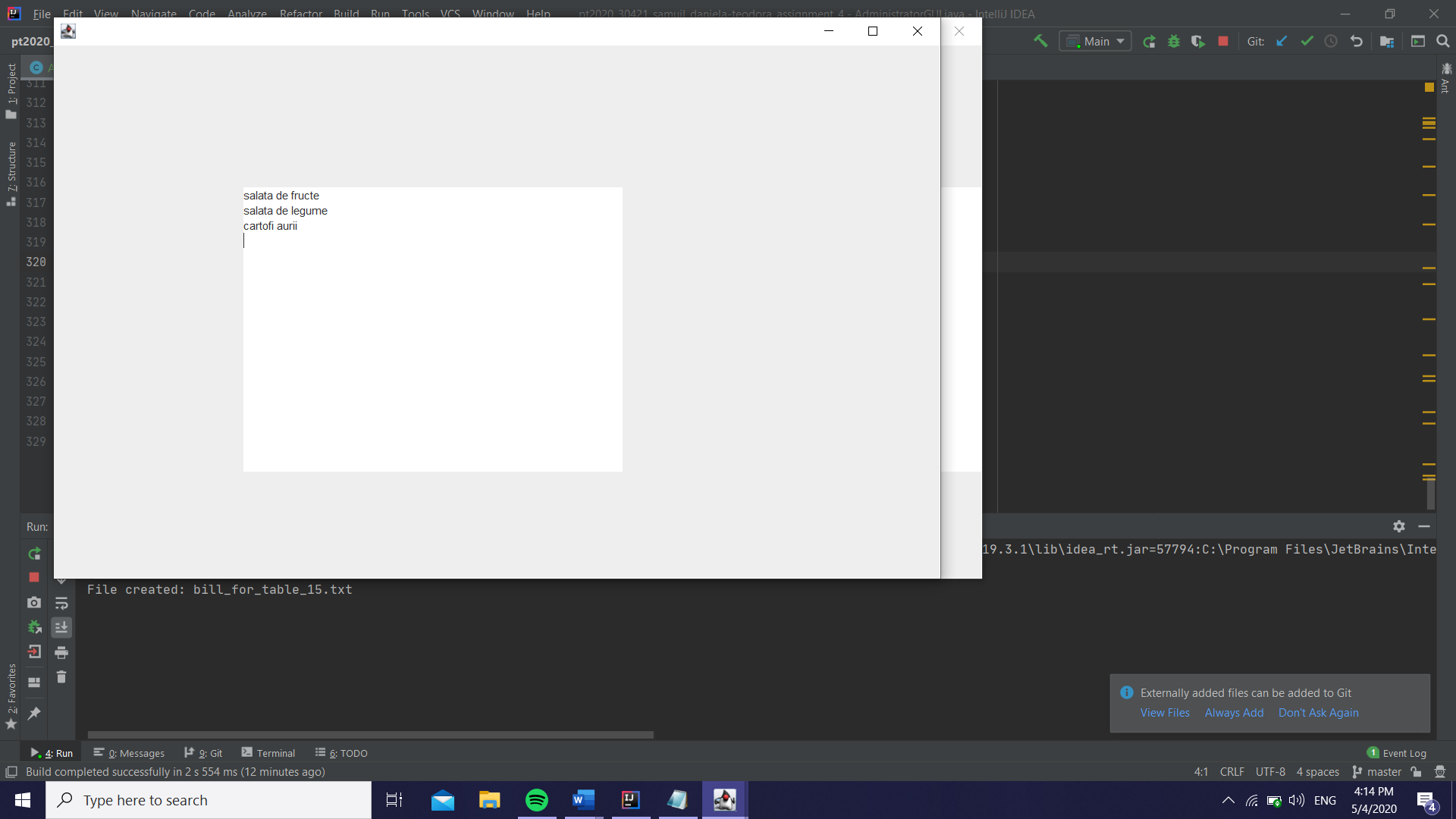
We can see that the product was added and the table updated.



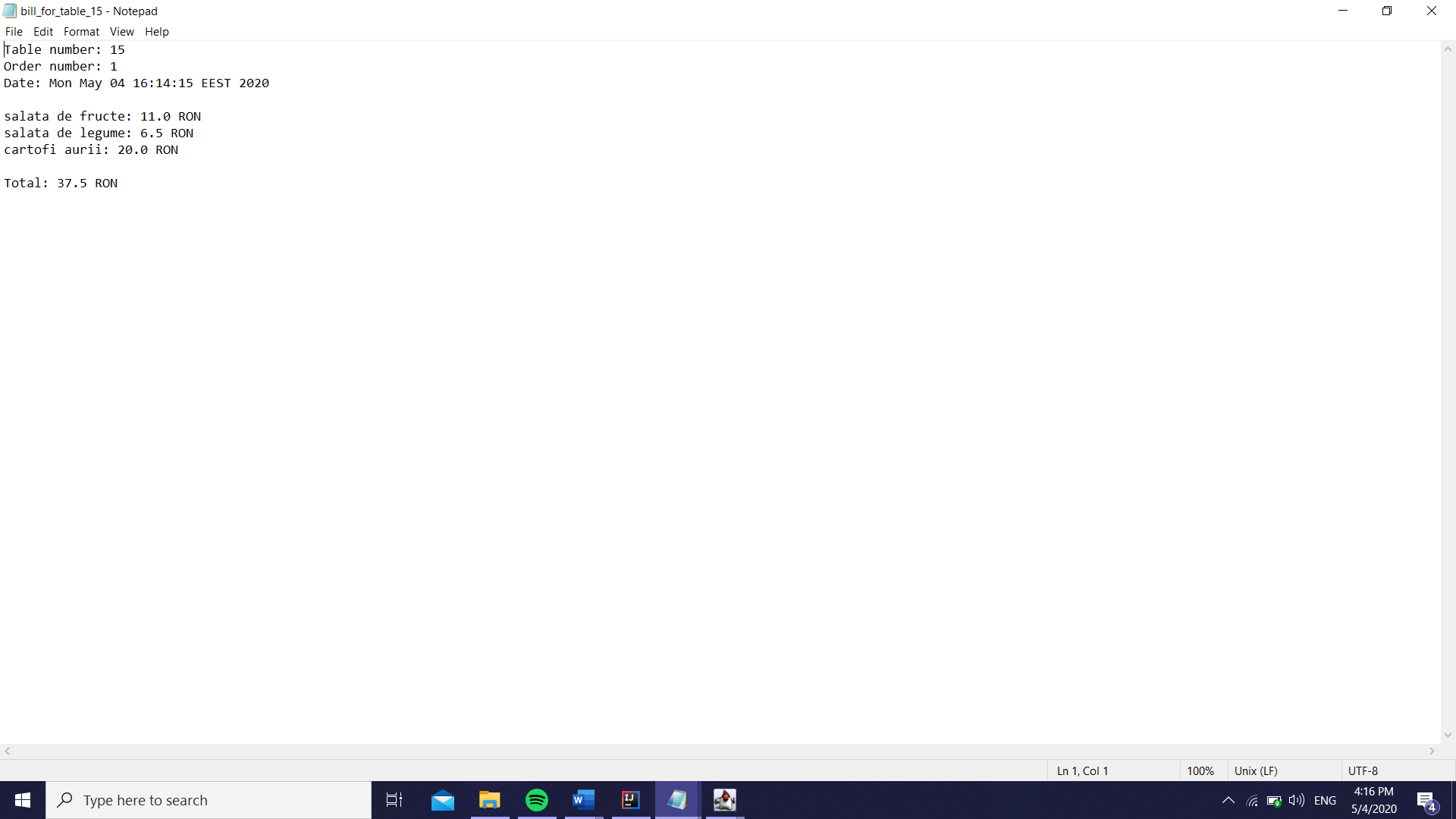
Creating a new composite product



Creating a new order for table 15 selecting the items by clicking on them and adding them to the list by pressing the button. We can see the added products in the text box on the right.



After saving the order, if we check the ChefGUI we can see that the text box was filled with the items we previously selected in the order.



When an order is saved, the bill is automatically generated, as in the example above.

1. Conclusions

In conclusion, I found this assignment the hardest so far because the complexity of the GUI, since I didn’t work with graphical interfaces that much before, but the rest was really helpful, especially the serialization, since it saves and loads data, making sure nothing gets lost. The assertion part was something new for me, but nonetheless something useful for pretesting and post testing the methods. Also, the connection between the waiter and the chef, the notifying part made with the help of the Observable interface, was something I found interesting and easy to work with, and I consider it a real help for my future work. Even though the application after all needs some improvement here and there, this project was the most `information filled` one so far, from my point of view.

1. Bibliography

I will attach some links I felt were useful for my work within this project, as well as some mentions I thought I should make:

* For understanding better hashing and the serialization concept I found useful the support presentation

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

* Also, to read more about serialization and to better understand the concepts behind it I used the following links

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

<https://www.tutorialspoint.com/java/java_serialization.html>

* I’ve read more about assertion on the following site

<https://javarevisited.blogspot.com/2012/01/what-is-assertion-in-java-java.html>

* To find more about the Observable interface and the Observer objects, I read the links below

<https://docs.oracle.com/javase/7/docs/api/java/util/Observable.html>

<https://www.javaworld.com/article/2077258/observer-and-observable.html>

* To improve my knowledge on the Composite Design Patter I followed the site below

<https://www.geeksforgeeks.org/composite-design-pattern/>

* Creating tables with the Swing interface was something new to me, so I informed myself using the following links

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

<https://www.enterprisedb.com/edb-docs/d/edb-postgres-connectors/user-guides/jdbc-guide/10.0.1/EDB_Postgres_Advanced_Server_JDBC_Connector_Guide.1.23.html>

<https://www.thoughtco.com/defaulttablemodel-overview-2033890>