Restaurant management system

Documentation

1. Homework objective

Main objectives:

The implementation of a restaurant’s managements system, supporting three types of users: administrators, waiters and chefs. Each type of user takes as input & performs different variables & operations, the results being displayed in the user interface & in .txt files.

Secondary objectives:

-creating a graphical user interface for the user to interact with the internal data

-implementation following different patterns: Model-View-Control, Composite Design Pattern, Design by contract, Observer Design Pattern

-creating and efficient .txt generator, which will generate .txt files containing information about the required object (bill)

-creation of a .jar file

1. Problem analysis, modeling, scenarios, use cases

Problem analysis:

It is required to correctly and efficiently implement an application for a restaurant’s management system, taking as input different types of data and performing certain operations. The system has three types of users: administrator – add, delete & modify existing menu products, waiter – create an order, add elements from the menu & compute the bill, and chef – observer user, is notified whenever a waiter creates an order. All the resulting information & objects are displayed in a panel in the user interface. Also, a .txt file containing information about the order is created when a waiter computes the bill.

Modeling:

The project is modeled following a MVC pattern, consisting of 4 different packages:

Model package – contains all the ‘data’ classes: BaseProduct, CompositeProduct, MenuItem & Order

View package – contains all the graphical interface related classes: MainGUI, AdministratorGUI, WaiterGUI, ChefGUI, CompositeGUI

Control package – contains the control class Restaurant & an interface containing the definitions of the main operations IRestaurantProcessing

Start package – containing the main class

Scenarios:

A scenario represents a sequence of steps taken by the user to correctly interact with the application. The following are the possible scenarios:

Scenario 1: The administrator user logs into the application.

S1.1: The administrator either adds a new menu item or encounters an error (input, existing item etc);

S1.2: The administrator either deletes a menu item (and all the composite products containing it) or encounters an error (input error, item not existing etc);

S1.3: The administrator either modifies a menu item (and all the composite products containing it) or encounters an error(input error, item not existing etc);

S1.4: The administrator successfully displays the menu (either empty or not);

S1.5: The administrator presses ‘back’ and returns to the main interface;

Scenario 2: The waiter user logs into the application.

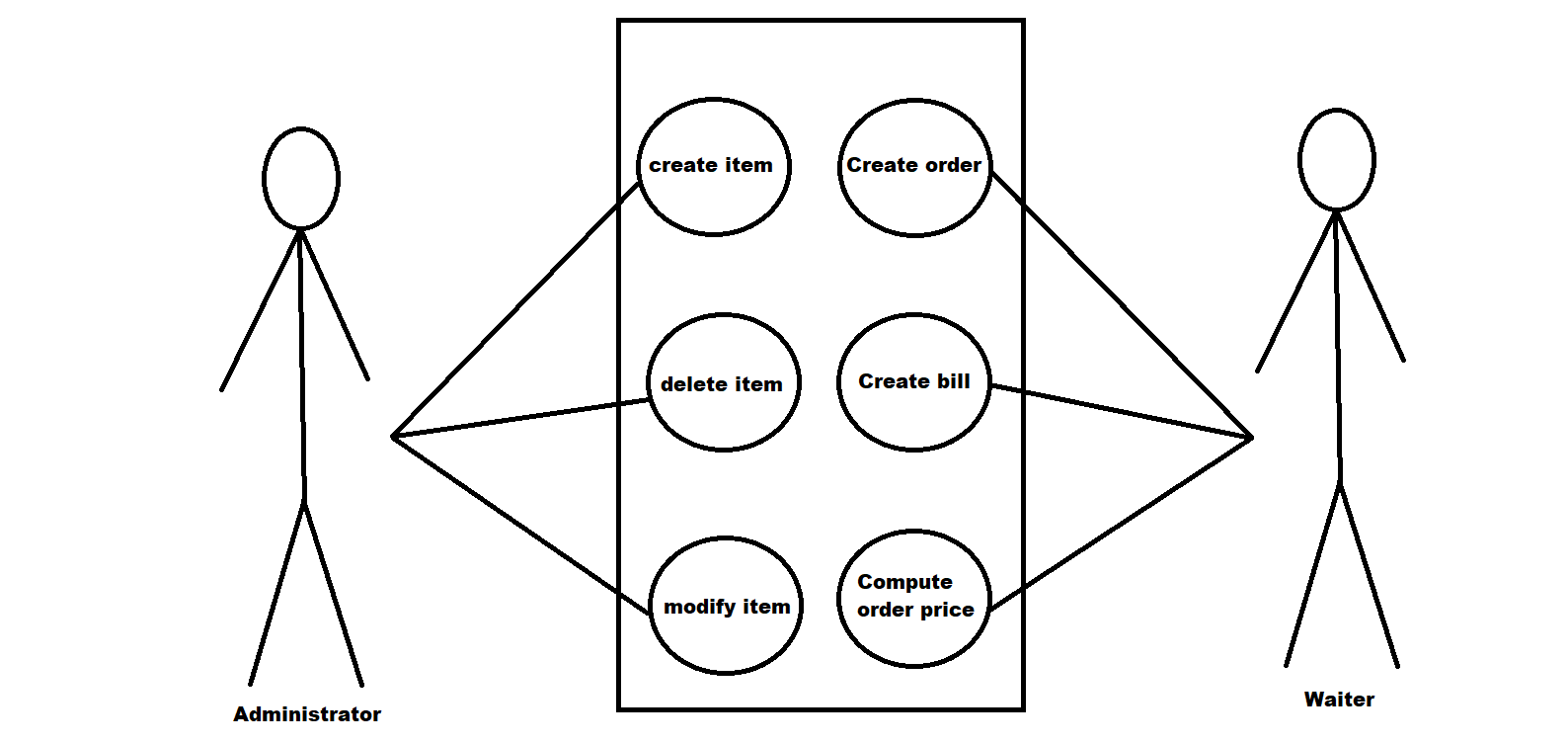
S2.1: The waiter either creates an order or encounters an error(input error); the order is created by introducing the order information and sequentially selecting menu items; whenever an order is created, it sends a notification to the ‘chef’ interface, opening the chef’s panel & displaying some information (New order, Chef is cooking, Chef is busy etc);

S2.2: The waiter either creates the bill for an order or encounters an error(input error, order not found etc);

S2.3: The waiter successfully displays the list of orders (either empty or not);

S2.4: The waiter presses ‘back’ and returns to the main interface;

Use cases:



1. Design

The project is designed following the MVC pattern. It consists of 4 packages & their classes:

Model:

-BaseProduct: class holding information about the ‘base products’; it extends the MenuItem class;

-CompositeProduct: class holding information about the ‘composite products’; it extends the MenuItem class;

-MenuItem: class holding information about the products, whether base or composite; super class for BaseProduct

and CompositeProduct;

-Order: class holding information about each order; it also contains the overridden hashCode & equals methods

View:

-MainGUI: class defining the main user interface; holds the generation code for the main visual panel

-AdministratorGUI: class defining the administrator user interface; holds the generation code for different visual objects & some methods used for the primary Administrator operations with base & composite products;

-WaiterGUI: class defining the waiter user interface; golds the generation code for different visual objects & some methods used for the primary Waiter operations;

-ChefGUI: class defining the chef user interface; mostly an empty class, it holds the overridden update code for the Observer Design Pattern

-CompositeGUI: class defining the composite product interface; holds the code for different visual objects & some methods used for the primary Administrator operations with composite products;

Control:

-Restaurant: the application’s control class; it creates an object for each GUI class, different data structures & variables, creates the action listener for each button & implements the main operations executed by the users; it extends Observable in order to notify the Observer classes (ChefGUI) whenever a certain action takes place (an order is created), and implements the IRestaurantProcessing interface, which holds the definitions of the primary user operations;

-IRestaurantProcessing: interface, contains the definitions of the primary user operations

Start:

-View: Main class, the entry point into the application;

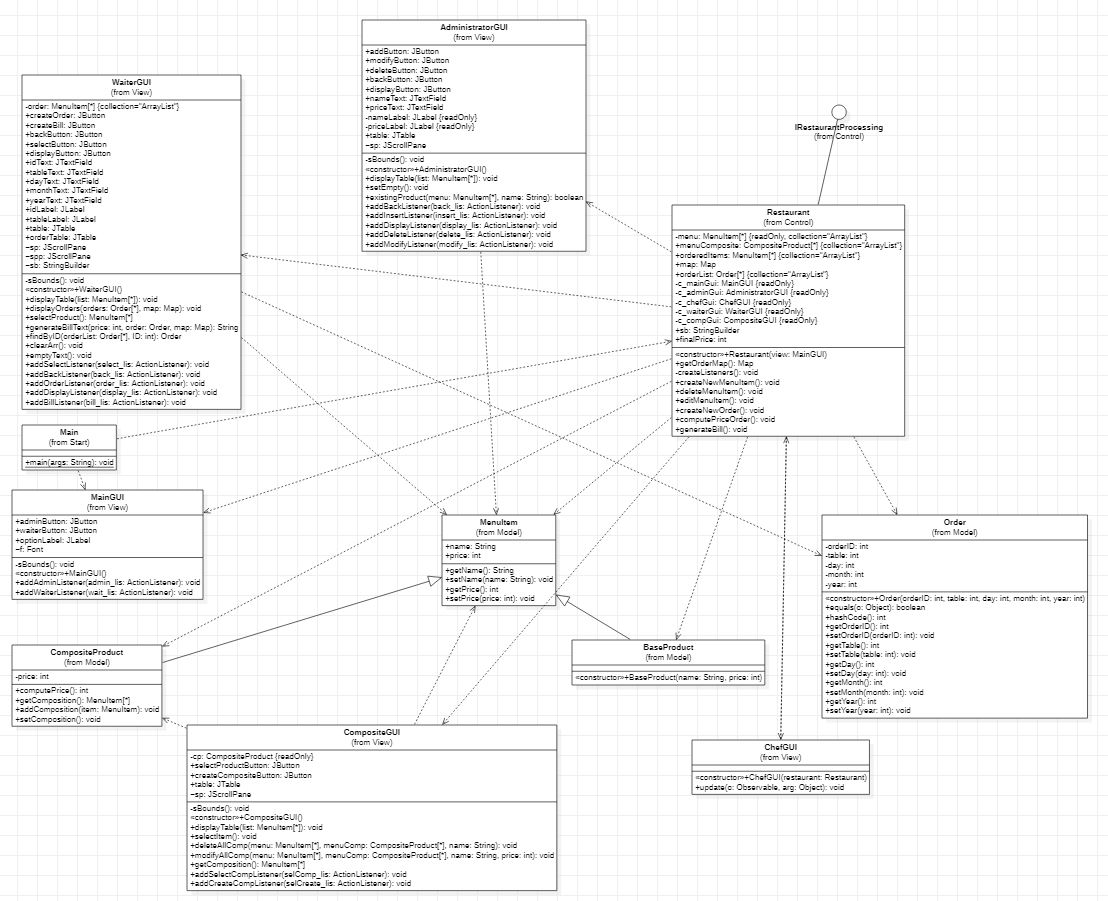
The Model classes BaseProduct, CompositeProduct & MenuItem are designed under a Composite Design pattern. The Observer Design Pattern is used to notify the Chef class whenever a waiter creates an order in the control class. Order related information is stored in a Map, whose key will be formed of objects of type order, for which the hashCode() methods is overwritten to compute the hash value within the Map from the attributes of the Oder (OrderID, TableID, Day/Month/Year). Pre/post conditions & assertions are used throughout the Restaurant class;

Items 1-4 of Bibliography were used wherever HashMap were used.

Item 5 of Bibliography was used for arraylist iterations.

Item 6 of Bibliography was used for assertions & pre/post conditions throughout the control class.

UML Diagram:



1. Implementation

Classes:

**MenuItem**:

Has the following attributes:

-name: the name of the item(product);

-price: the price of the item

Has the following methods:

-getters & setters for: name, price

**BaseProduct**:

Inherits the attributes & methods of the super class MenuItem;

The constructor of the class sets the name & the price;

**CompositeProduct**:

Inherits the attributes & methods of the super class MenuItem;

Has the following attributes:

-composition: an ArrayList containing all the MenuItems which compose the product

Has the following methods:

-getter & setter for the composition;

public int computePrice(){  
 price = 0;  
 for (MenuItem item : composition)  
 price += item.getPrice();  
 return price;  
}

This method computes the price of the composite product, by adding the prices of all the composing products.

**Order**:

Has the following attributes:

-orderID: id of the order;

-table: id of the table;

-day, month, year – the order date;

The constructor of this class sets the orderID, table, day, month & year.

Has the following methods:

-getters & setters for orderID, table, day, month, year

@Override  
public int hashCode() {  
 int hashCode = 12;  
 hashCode += 5\*hashCode + 30\*orderID + 5\*day + 9\*month + year + 4\*table;  
 return hashCode;  
}

This method computes the hash value, the number which is the input of the internal hash function of the HashMap.

@Override  
public boolean equals(Object o) {  
 if (this == o) return true;  
 if (!(o instanceof Order)) return false;  
 Order order = (Order) o;  
 return orderID == order.orderID &&  
 table == order.table &&  
 day == order.day &&  
 month == order.month &&  
 year == order.year;  
}

This method is used in order to compare key equality for get operations or to check if the object exists in the map in the case of put operation.

**MainGUI**:

Has the following attributes:

-adminButton: button for administrator;

-waiterButton: button for waiter;

-optionLabel: the upper interface text;

-f: new font;

The constructor of this class creates a JPanel, adds the items to the visible interface & calls the sBounds() method which sets the objects’ coordinates;

This method has the following methods:

-addActionListener for each button;

private void sBounds(){  
 this.setLocation(470, 100);  
 optionLabel.setBounds(65, 20, 500, 300);  
 adminButton.setBounds(20, 250, 160, 35);  
 waiterButton.setBounds(420, 250, 160, 35);optionLabel.setFont(f);  
}

This method sets the coordinates of the panel & the objects within;

**AdministratorGUI**:

Has the following attributes:

-addButton: button for adding an item;

-modifyButton: button for modifying an item;

-deleteButton: button for deleting an item;

-backButton: button for returning to the main GUI;

-nameText: textfield for the input name;

-priceText: textfield for the input price;

-nameLabel & priceLabel: labels

-table: a JTable for display

-sp: a scroll pane for display

The constructor of this class creates a JPanel, adds the items to the visible interface & calls the sBounds() method which sets the objects’ coordinates;

This class has the following methods:

-addActionListener for each button;

public void displayTable(ArrayList<MenuItem> list){  
 String[] col = {"Name", "Price"};  
 DefaultTableModel tab = new DefaultTableModel();  
 tab.setColumnIdentifiers(col);  
 Object[] obj = new Object[2];  
 Iterator<MenuItem> iterator = list.iterator();  
 while (iterator.hasNext()){  
 MenuItem item = iterator.next();  
 obj[0] = item.getName();  
 obj[1] = item.getPrice();  
 tab.addRow(obj);  
 }  
 table = new JTable(tab);  
 sp.setVisible(true);  
 sp.setBounds(10, 130, 370, 450);  
 sp.setViewportView(table);  
}

This method displays the Menu on a JTable. It creates all the necessary visual items, then iterates the menu arraylist and displays the name & price of each item on a different row.

public void setEmpty(){  
 priceText.setText("");  
 nameText.setText("");  
}

This method sets the content of the input textfields to “” (empty).

public boolean existingProduct(ArrayList<MenuItem> menu, String name){  
 Iterator<MenuItem> itr = menu.iterator();  
 while (itr.hasNext()){  
 MenuItem item = itr.next();  
 if (item.getName().compareTo(name) == 0)  
 return true;  
 }  
 return false;  
}

This method verifies if an item already exists in the menu. It iterates the menu and compares the input item with every item in the menu. It returns true if the item is already in the menu & false otherwise.

**WaiterGUI:**

Has the following attributes:

-createOrder: button for creating an order;

-createBill: button for creating a bill;

-selectButton: button for selecting products in order to create an order;

-displayButton: button for displaying the orders;

-backButton: button for returning to the main GUI;

-idText: textfield for input orderID;

-tabletext: textfield for input table;

-dayText, monthText, yearText: textfields for the input date;

-idLabel, tableLabel: labels;

-table, orderTable: Jtables for display;

-sp, spp: JScrollPanes for display

-sb: stringbuilder;

-order: an ArrayList containing the items of an order;

The constructor of this class creates a JPanel, adds the items to the visible interface, calls the sBounds() method which sets the objects’ coordinates & creates a new instance of the order object;

This class has the following methods:

-addActionListener for each button;

public void displayTable(ArrayList<Model.MenuItem> list){  
 String[] col = {"Name", "Price"};  
 DefaultTableModel tab = new DefaultTableModel();  
 tab.setColumnIdentifiers(col);  
 Object[] obj = new Object[2];  
 Iterator<Model.MenuItem> iterator = list.iterator();  
 while (iterator.hasNext()){  
 MenuItem item = iterator.next();  
 obj[0] = item.getName();  
 obj[1] = item.getPrice();  
 tab.addRow(obj);  
 }  
 table = new JTable(tab);  
 sp.setVisible(true);  
 sp.setBounds(10, 130, 370, 450);  
 sp.setViewportView(table);  
}

This method displays the Menu on a JTable. It creates all the necessary visual items, then iterates the menu arraylist and displays the name & price of each item on a different row.

public void displayOrders(ArrayList<Order> orders, Map<Order, ArrayList<MenuItem>> map){  
 String[] col = {"OrderID", "Table", "Date", "Items"};  
 DefaultTableModel tab = new DefaultTableModel();  
 tab.setColumnIdentifiers(col);  
 Object[] obj = new Object[4];  
 String s;  
 Iterator<Order> iterator = orders.iterator();  
 while (iterator.hasNext()){  
 sb = new StringBuilder();  
 Order order = iterator.next();  
 obj[0] = order.getOrderID();  
 obj[1] = order.getTable();  
 s = order.getDay() + "/" + order.getMonth() + "/" + order.getYear();  
 obj[2] = s;  
 ArrayList<MenuItem> arr = map.get(order);  
 Iterator<MenuItem> itt = arr.iterator();  
 while (itt.hasNext()) {  
 MenuItem mt = itt.next();  
 sb.append(mt.getName());  
 if (itt.hasNext())  
 sb.append(",");  
 }  
 obj[3] = sb.toString();  
 tab.addRow(obj);  
 }  
 orderTable = new JTable(tab);  
 spp.setVisible(true);  
 spp.setBounds(390, 130, 370, 450);  
 spp.setViewportView(orderTable);  
}

This method displays all the orders in a JTable. It creates all the necessary visual items & iterates the order arrayList in order to retrieve information about the order. It stores all the basic order data, then gets the arrayList of ordered items saved on the order key from the HashMap. It then iterates the arrayList, retrieves the ordered items & displays the information about the order on a new row.

public ArrayList<MenuItem> selectProduct(){  
 int row = table.getSelectedRow();  
 String nameAux = table.getModel().getValueAt(row, 0).toString();  
 String priceAux = table.getModel().getValueAt(row, 1).toString();  
 Model.MenuItem item = new MenuItem();  
 item.setName(nameAux);  
 item.setPrice(Integer.*parseInt*(priceAux));  
 order.add(item);  
 return order;  
}

This method adds the selected row item into the order ArrayList. Whenever the ‘Select’ button from the Waiter GUI is pressed, the selected JTable row is saved into the order ArrayList. After multiple ‘Select’ operations, the ArrayList will hold information about all the selected items.

public String generateBillText(int price, Order order, Map<Order, ArrayList<MenuItem>> map){  
 StringBuilder string = new StringBuilder();  
 string.append("Bill for order: ").append(order.getOrderID()).append(System.*lineSeparator*());  
 string.append("Table served: ").append(order.getTable()).append(System.*lineSeparator*());  
 string.append("Date: ").append(order.getDay()).append("/").append(order.getMonth()).append("/").append(order.getYear()).append(System.*lineSeparator*());  
 string.append("Order: ").append(System.*lineSeparator*());  
  
 ArrayList<MenuItem> list = map.get(order);  
 Iterator<MenuItem> iterator = list.iterator();  
 while (iterator.hasNext()){  
 MenuItem item = iterator.next();  
 string.append(item.getName()).append(" ").append(item.getPrice()).append(System.*lineSeparator*());  
 }  
 string.append("Total price: ").append(price).append(System.*lineSeparator*());  
 return string.toString();  
}

This method creates the text displayed in the bill created by the waiter. It appends to a stringbuilder certain pieces of text & certain pieces of information about the order. Information about the order is retrieved from the Order object given as parameter & from the Map given as parameter. The order object holds info about the id, table & date, while the Map holds info about the ordered items. The method returns the desired string.

public Order findByID(ArrayList<Order> orderList, int ID){  
 for (Order order: orderList){  
 if (order.getOrderID() == ID)  
 return order;  
 }  
 return null;  
}

This method searches for a certain order in the arraylist of orders. It iterates the list and compares the Id given as input with the ID of each order. It returns the order if the ID’s matched (the order was found by the given id) and null otherwise (an order with the given ID does not exist).

public void clearArr(){  
 order = new ArrayList<>();  
}

This method creates a new instance of the order arraylist. It is called whenever the user needs to create a new list of ordered items, which will eventually be placed into the hashmap.

public void emptyText(){  
 idText.setText("");  
 tableText.setText("");  
 dayText.setText("");  
 monthText.setText("");  
 yearText.setText("");  
}

This method sets the content of the input textfields to “” (empty).

**ChefGUI**:

This class implements Observer.

The constructor of this class sets it as an observer of the restaurant class & creates the empty panel.

This class has the following methods:

@Override  
public void update(Observable o, Object arg) {  
 this.setVisible(true);  
 int p = JOptionPane.*showConfirmDialog*(null, arg, "Kitchen notifier", 2);  
 if (p == 0){  
 JOptionPane.*showMessageDialog*(null,"Chef cooks.");  
 this.setVisible(false);  
 }  
 else {  
 JOptionPane.*showMessageDialog*(null, "Chef is already cooking.");  
 this.setVisible(false);  
 }  
}

This method displays the chef interface on the screen, displays pieces of text & hides the interface back.

**CompositeGUI**:

Has the following attributes:

-selectProductButton: button for selecting a product

-createCompositionButton: button for creating the composition of a composite product

-table: JTable;

-sp: JScrollPane;

-cp: a composite product;

The constructor of this class creates a JPanel, adds the items to the visible interface, calls the sBounds() method which sets the objects’ coordinates & creates a new instance of the order object;

Has the following methods:

-addActionListener for each button;

private void sBounds(){  
 this.setLocation(570, 100);  
 selectProductButton.setBounds(10, 10, 80, 30);  
 createCompositeButton.setBounds(10, 50, 80, 30);  
}

This method sets the coordinates of the panel & the objects within;

public void displayTable(ArrayList<Model.MenuItem> list){  
 cp.setComposition();  
 String[] col = {"Name", "Price"};  
 DefaultTableModel tab = new DefaultTableModel();  
 tab.setColumnIdentifiers(col);  
 Object[] obj = new Object[2];  
 Iterator<Model.MenuItem> iterator = list.iterator();  
 while (iterator.hasNext()){  
 MenuItem item = iterator.next();  
 obj[0] = item.getName();  
 obj[1] = item.getPrice();  
 tab.addRow(obj);  
 }  
 table = new JTable(tab);  
 sp.setVisible(true);  
 sp.setBounds(10, 130, 370, 450);  
 sp.setViewportView(table);  
}

This method displays the Menu on a JTable. It creates all the necessary visual items, then iterates the menu arraylist and displays the name & price of each item on a different row.

public void selectItem(){  
 int row = table.getSelectedRow();  
 String nameAux = table.getModel().getValueAt(row, 0).toString();  
 String priceAux = table.getModel().getValueAt(row, 1).toString();  
 MenuItem item = new MenuItem();  
 item.setName(nameAux);  
 item.setPrice(Integer.*parseInt*(priceAux));  
 cp.addComposition(item);  
}

This method adds the selected row item into the composition of the composite product. Whenever the ‘Select’ button from the Composite GUI is pressed, the selected JTable row is saved into the order ArrayList. After multiple ‘Select’ operations, the ArrayList will hold information about all the selected items.

public void deleteAllComp(ArrayList<MenuItem> menu, ArrayList<CompositeProduct> menuComp, String name){  
 Iterator<CompositeProduct> itr = menuComp.iterator();  
 while (itr.hasNext()){  
 CompositeProduct newCp = itr.next();  
 Iterator<MenuItem> itrr = newCp.getComposition().iterator();  
 while (itrr.hasNext()){  
 MenuItem item = itrr.next();  
 if (item.getName().compareTo(name) == 0){  
 Iterator<MenuItem> itrrr = menu.iterator();  
 while (itrrr.hasNext()){  
 MenuItem menuItem = itrrr.next();  
 if (menuItem.getName().compareTo(newCp.getName()) == 0){  
 itrrr.remove();  
 itr.remove();  
 }  
 }  
 }  
 }  
 }  
}

This method deletes all the composite products containing a menu item deleted by the administrator. It iterates the arraylist containing all the composite products, then it iterates the arraylist containing the composition of each composite product. It verifies if the name of the deleted item is found within the composition of a composite product. If yes, it then iterates the menu and deletes the said composite product from the menu arraylist & the composite products arraylist (whenever an item is deleted, we delete all the composite products containing it from the menu & from the list of existing composite products).

public void modifyAllComp(ArrayList<MenuItem> menu, ArrayList<CompositeProduct> menuComp, String name, int price){  
 Iterator<CompositeProduct> itr = menuComp.iterator();  
 while (itr.hasNext()){  
 CompositeProduct newCp = itr.next();  
 ListIterator<MenuItem> itrr = newCp.getComposition().listIterator();  
 while (itrr.hasNext()){  
 MenuItem item = itrr.next();  
 if (item.getName().compareTo(name) == 0){  
 item.setName(name);  
 item.setPrice(price);  
 itrr.set(item);  
 newCp.setPrice(newCp.computePrice());  
 ListIterator<MenuItem> itrrr = menu.listIterator();  
 while (itrrr.hasNext()){  
 MenuItem item1 = itrrr.next();  
 if (item1.getName().compareTo(newCp.getName()) == 0){  
 itrrr.set(newCp);  
 }  
 }  
 }  
 }  
 }  
}

This method modifies the price of all the composite products containing a menu item modified by the administrator. It iterates the arraylist containing all the composite products, then it iterates the arraylist containing the composition of each composite product. It verifies if the name of the modified item is found within the composition of a composite product. If yes, then the item from the composition is replaced with an item of the name name & new price. Then, the new price of the composite product is computed by calling the computePrice method. It then iterates the menu arraylist and sets the new composite product (the product with the new price computed) on the place of the older composite product (same product, but with unmodified price). Whenever an item’s price is modified, we modify the price of all the composite products containing the said item.

**Restaurant**:

Extends observable, implements IRestaurantProcessing.

Has the following attributes:

-menu: arraylist containing all the available menu items;

-menuComposites: arraylist containing all the composite products from the menu

-orderedItems: arraylist containing all the items from an order

-orderList: arraylist containing all the orders;

-map: map, saves orderedItems values on order keys;

-c\_mainGui: an instance of the main GUI;

-c\_adminGui: an instance of the administrator GUI;

-c\_chefGui: an instance of the chef GUI;

-c\_waiterGui: an instance of the waiter GUI;

-c\_compGui: an instance of the composite GUI;

-sb: StringBuilder;

-finalPrice: the final price of an order

The constructor of this method creates all the GUI objects, initializes all the arraylists and calls the createListeners method in order to create a listener for each button;

Has the following methods:

-getter for the map;

private void createListeners(){  
 c\_mainGui.addAdminListener(new operationListener());  
 c\_mainGui.addWaiterListener(new operationListener());  
 c\_adminGui.addBackListener(new operationListener());  
 c\_adminGui.addInsertListener(new operationListener());  
 c\_adminGui.addDisplayListener(new operationListener());  
 c\_adminGui.addDeleteListener(new operationListener());  
 c\_adminGui.addModifyListener(new operationListener());  
 c\_waiterGui.addBackListener(new operationListener());  
 c\_waiterGui.addSelectListener(new operationListener());  
 c\_waiterGui.addOrderListener(new operationListener());  
 c\_waiterGui.addBillListener(new operationListener());  
 c\_waiterGui.addDisplayListener(new operationListener());  
 c\_compGui.addSelectCompListener(new operationListener());  
 c\_compGui.addCreateCompListener(new operationListener());  
}

This method creates a new operatioListener for each button within the application.

@Override  
public void createNewMenuItem() {  
 String price = c\_adminGui.priceText.getText();  
 String name = c\_adminGui.nameText.getText();  
 int preSize = menu.size();  
 try {  
 if (!(name.compareTo("") == 0) && !(price.compareTo("") == 0)) {  
 BaseProduct bp = new BaseProduct(c\_adminGui.nameText.getText(), Integer.*parseInt*(c\_adminGui.priceText.getText()));  
 if (!c\_adminGui.existingProduct(menu, name)) {  
 menu.add(bp);  
 int postSize = menu.size();  
 assert postSize == preSize + 1;  
 JOptionPane.*showMessageDialog*(null, "Item added");  
 c\_adminGui.setEmpty();  
 }  
 else  
 JOptionPane.*showMessageDialog*(null, "Item already in menu");  
 } else if (!(name.compareTo("") == 0)) {  
 if (!c\_adminGui.existingProduct(menu, name)) {  
 c\_adminGui.setVisible(false);  
 c\_compGui.setVisible(true);  
 c\_compGui.displayTable(menu);  
 }  
 else  
 JOptionPane.*showMessageDialog*(null, "Item already in menu");  
 } else  
 JOptionPane.*showMessageDialog*(null, "Input error");  
 }  
 catch (NumberFormatException ex){  
 JOptionPane.*showMessageDialog*(null, "Input error");  
 }  
}

This method creates a new menu item. It saves the price & name from the input textfield of the Administrator GUI. If a new base product is added (both name & price are given), it is simply added to the menu. If a composite product is added (only name is given), we display the composite GUI which deals with the creation of a new composite product. During the creation of the menu item, we can encounter different errors such as ‘Input error’/’Item already in menu’. These errors occur when the input data has an invalid/forbidden format. When a new item is created, the message ‘Item added’ is displayed. In order to create a base product, both name & price must be entered. If only the name is given, the composite product GUI will be displayed. We need to click on the desired menu item & select in order to add it to the composition. Repeat this step for as many products as we want, then press create.

@Override  
public void deleteMenuItem(){  
 String price = c\_adminGui.priceText.getText();  
 String name = c\_adminGui.nameText.getText();  
 boolean exist = true;  
 int preSize = menu.size();  
 try {  
 if (!(name.compareTo("") == 0) && (price.compareTo("") == 0)) {  
 Iterator<MenuItem> itr = menu.iterator();  
 while (itr.hasNext()) {  
 MenuItem it = itr.next();  
 if (it.getName().compareTo(name) == 0) {  
 itr.remove();  
 int postSize = menu.size();  
 assert postSize == preSize - 1;  
 JOptionPane.*showMessageDialog*(null, "Item & composites containing it removed");  
 exist = false;  
 }  
 }  
 c\_compGui.deleteAllComp(menu, menuComposite, name);  
 if (exist)  
 JOptionPane.*showMessageDialog*(null, "Item not existent");  
 } else  
 JOptionPane.*showMessageDialog*(null, "Input error");  
 }  
 catch (NullPointerException ex){  
 JOptionPane.*showMessageDialog*(null, "Empty menu");  
 }  
 c\_adminGui.setEmpty();  
}

This method deletes an item & all the composite products containing it from the menu. It saves the data loaded in the input textfields. It then iterates the menu arraylist, compares the name given as input with the names of the menu items, and deletes them if they match. Then, the deleteAllComp method from the Composite GUI is called, in order to delete all the composite products containing the given product. During the deletion of the item, we can encounter different errors, such as ‘Input error’/’Empty menu’/’Item not existent’. These errors occur then the input data has an invalid/forbidden format. When an item is deleted, the message ‘Item & composites containing it deleted’ is displayed. In order to delete a product, only the name of the product should be given as input.

public void editMenuItem() {  
 String price = c\_adminGui.priceText.getText();  
 String name = c\_adminGui.nameText.getText();  
 try {  
 if (!(name.compareTo("") == 0) && !(price.compareTo("") == 0)){  
 c\_compGui.modifyAllComp(menu, menuComposite, name, Integer.*parseInt*(price));  
 ListIterator<MenuItem> itr = menu.listIterator();  
 while (itr.hasNext()){  
 MenuItem item = itr.next();  
 if (item.getName().compareTo(name) == 0){  
 item.setPrice(Integer.*parseInt*(price));  
 item.setName(name);  
 itr.set(item);  
 JOptionPane.*showMessageDialog*(null, "Item & composites containing it modified");  
 }  
 }  
 }  
 else  
 JOptionPane.*showMessageDialog*(null, "Input error");  
 }  
 catch (NumberFormatException ex){  
 JOptionPane.*showMessageDialog*(null, "Input error");  
 }  
 c\_adminGui.setEmpty();  
}

This method modifies the price of an item & of all the composite products containing it. It saves the data loaded in the input textfields. It iterates the menu arraylist, compares the name given as input with the names of the menu items, and modifies the price if they match. Then, the modifyAllComp method from the Composite GUI is called, in order to modify the prices of all the composite products containing the given item. During the modification of an item, we can encounter the ‘Input error’, which occurs when the input data has an invalid/forbidden format. When an item is modified, the message ‘item & composites containing it modified’. In order to modify a product, we enter its name, and the new price value that we desire.

public void createNewOrder() {  
 try {  
 if (!(c\_waiterGui.idText.getText().compareTo("") == 0) && !(c\_waiterGui.tableText.getText().compareTo("") == 0)  
 && !(c\_waiterGui.dayText.getText().compareTo("") == 0) && !(c\_waiterGui.monthText.getText().compareTo("") == 0)  
 && !(c\_waiterGui.yearText.getText().compareTo("") == 0)) {  
 int orderID = Integer.*parseInt*(c\_waiterGui.idText.getText());  
 int table = Integer.*parseInt*(c\_waiterGui.tableText.getText());  
 int day = Integer.*parseInt*(c\_waiterGui.dayText.getText());  
 int month = Integer.*parseInt*(c\_waiterGui.monthText.getText());  
 int year = Integer.*parseInt*(c\_waiterGui.yearText.getText());  
 Order order = new Order(orderID, table, day, month, year);  
 Order oldOrder = order;  
 orderList.add(order);  
 map.put(order, orderedItems);  
 assert oldOrder.equals(order);  
 c\_waiterGui.clearArr();  
 String s = "New order!";  
 setChanged();  
 notifyObservers(s);  
 }  
 else  
 JOptionPane.*showMessageDialog*(null, "Input error");  
 }  
 catch (NumberFormatException ex) {  
 JOptionPane.*showMessageDialog*(null, "Input error");  
 }  
 c\_waiterGui.emptyText();  
}

This method creates a new order. It saves all the data loaded in the input textfields, converts in into integer values and sets the attrivutes of a new order object. The order is added into the arraylist of orders. Then, it adds the arraylist of ordered items (composition of the order) into the map on the key ‘order’. After adding the arraylist of ordered items into the map, the arraylist is emptied. A string ‘New Order’ is created, and the observers are notified that the order is ready (in our case, the only observer is the chef). During the creation of the order, we can encounter the ‘Input error’, which occurs when the input data is invalid/forbidden. In order to create an order, we must fill in ALL fields with the desired values (id, tableid, day/month/year), and select items from the displayed table (one by one) by pressing select. When the desired order is ready, ‘Create’ is pressed and the chef is notified.

@Override  
public void computePriceOrder() {  
 finalPrice = 0;  
 int initialPrice = finalPrice;  
 try {  
 int ID = Integer.*parseInt*(c\_waiterGui.idText.getText());  
 Order order = c\_waiterGui.findByID(orderList, ID);  
 ArrayList<MenuItem> array = map.get(order);  
 Iterator<MenuItem> iterator = array.iterator();  
 while (iterator.hasNext()) {  
 MenuItem item = iterator.next();  
 finalPrice += item.getPrice();  
 }  
 assert finalPrice != initialPrice;  
 } catch (NumberFormatException | NullPointerException e) {  
 JOptionPane.*showMessageDialog*(null, "Order not found");  
 }  
}

This method computes the price of an order. The price is initialized as 0. It then calls the findByOrder method from the waiter GUI to find the appropriate order (the order having the ID equal to the ID given as input). After finding the order, we iterate the arraylist containing its composition, and add the price of each item to the final price. During the computation, we can encounter the ‘Order not found’ error, which occurs when the input data is invalid/forbidden. The price of an order is computer automatically when the bill is created.

@Override  
public void generateBill() {  
 int ID;  
 Order order;  
 String init = "";  
 assert init.equals("");  
 try {  
 ID = Integer.*parseInt*(c\_waiterGui.idText.getText());  
 computePriceOrder();  
 order = c\_waiterGui.findByID(orderList, ID);  
 try{  
 init = c\_waiterGui.generateBillText(finalPrice, order, map);  
 assert !init.equals("");  
 BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream("bill" + ID + ".txt")));  
 writer.write(init);  
 writer.close();  
 JOptionPane.*showMessageDialog*(null, "Bill created!");  
 }  
 catch (IOException ex){  
 ex.printStackTrace();  
 }  
 }  
 catch (NumberFormatException | NullPointerException ex){  
 JOptionPane.*showMessageDialog*(null, "Error creating the bill");  
 }  
}

This method generates the bill. It calls computePriceOrder in order to compute the price of the order with the given ID. It then calls the generateBillText method from the waiter GUI in order to get the requested string containing all the info about the given order. During the creation of the bill, we can encounter the ‘error creating the bill error’, which occurs when the input is invalid/forbidden. When a bill is created, the text ‘Bill created’ is displayed**. In order to compute the bill of an order, we must introduce the ID of the bill whose order we want to compute and press Bill.**

The operationListener class implementing actionlistener contains specific code for each button. Whenever the source of the action event matches a specific button, the button’s predefined operation will be executed.

1. **Conclusions**

This assignment has given me a better understanding of working with hashmaps, different GUI objects & design patterns such as Observer, and given me the chance of revising & adapting the knowledge I had regarding different topics, such as the MVC pattern, working with bufferedwriters iterators and many more. In conclusion, this assignment has been a great opportunity to both learn new things and revise information I already had.

1. **Bibliography**

1.<https://stackoverflow.com/questions/1066589/iterate-through-a-hashmap>

2. <https://www.geeksforgeeks.org/hashmap-put-method-in-java/>

3. <https://javahungry.blogspot.com/2014/03/hashmap-vs-hashtable-difference-with-example-java-interview-questions.html>

4. <https://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html>

5. <https://www.logicbig.com/how-to/java-collections/list-replace-all.html>

6. <https://www.geeksforgeeks.org/assertions-in-java/>