**Homework 4**

1. **Objective**

• Design by Contract Programming Techniques

• Polymorphism

• Design Patterns: Observer, Composite

• JCF HashMap and HashSet implementations

• Serialization

The main objective of this homework is to propose, design and a restaurant management system. The system should have three types of users: administrator, waiter and chef. The administrator can add, delete and modify existing products from the menu. The waiter can create a new order for a table, add elements from the menu, and compute the bill for an order. The chef is notified each time it must cook food ordered through a waiter. The secondary objectives are:

1. **Designing the interface RestaurantProcessing.** This defines the basic operations that can be executed by the administrator and the waiter. This is detailed in cap 2 and the implementation can be found in cap 4;
2. **Designing the MenuItem abstract class and BaseProduct, CompositeProduct with the Composite Design Pattern.** This is used to store information about the products. This is detailed in cap 2 and the implementation can be found in cap 4.
3. **Designing the Chef with the Observer Design Pattern.** The Chef will be notified every time a new Order is placed. This is detailed in cap 2 and the implementation can be found in cap 4.
4. **Designing the Order.** This is used to store information about the Orders. This is detailed in cap 2 and the implementation can be found in cap 4.
5. **Designing the Restaurant.** The Restaurant class controls the operations and all the information about the restaurant. This is detailed in cap 2 and the implementation can be found in cap 4.
6. **Designing the RestaurantSerialization class.** Here the Menu of the restaurant is serialized and deserialized when the application starts. The implementation can be found in cap 4;
7. **Problem analysis, modeling, scenarios, use cases**

There are several use cases for this problem. This time we have several “actors”: The Chef, The Administrator and The Waiter. For the sake of simplicity, this time there is only one of each. Each one of them has his unique GUI with which they can interact, but these GUIs are part of the Restaurant Management System (RMS). Let us see the use cases:

1. Use case (Chef)

-The Chef presses the “Chef” button in the main GUI;

-The RMS pops the Chef window;

-The Chef is amazed by the lack of operations he can do;

-The Chef refuses to use this application, arguing that even the Waiter yodeling the order from across the restaurant would be more efficient;

1. Use case (Waiter-add Order)

-The Waiter presses the “Waiter” button in the main GUI;

-The RMS pops the Waiter window;

-The Waiter supplies the required input and yodelers the Order to the Chef;

-The Chef quits

1. Use case (Administrator – add Product)

-The Administrator presses the “Administrator” button;

-The RMS pops the Administrator window;

-The Administrator adds a brand-new Promotion to the Menu;

1. Use case (Administrator - update)

-The Administrator presses the “Administrator” button;

-The RMS pops the Administrator window;

-The Administrator believes that some Items on the Menu are underpriced and begins to do some changes;

1. Use case (Administrator - delete)

-The Administrator presses the “Administrator” button;

-The RMS pops the Administrator window;

-The Administrator believes that some Items on the Menu are dated and begins to do some change

1. Use case (Waiter - generateBill)

-The Waiter presses the “Waiter” button;

-The WMS pops the Waiter window;

-The Waiter is very surprised to see that he no longer has some items on the menu, but carries on,

-The Waiter generates Bills for the clients;

1. Use case (Failed Enterprise)

-The Clients storm the Administrator`s office in rage, others are exiting the Restaurant happily. Those that went to see the Administrator are angry because of price increases after they have ordered the food and those that exit happily are Clients who no longer must pay because of inexistent Items.

-The Administrator refuses to pay a dime. He blames it all on the software engineer who designed this flawed system. Little did he know; the software engineer was just trying to learn more about the Design by Contract Pattern and left intentional flaws in it to gather more data.

1. **Design (design decisions, UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interface)**

Firstly, we must design the RestaurantProcessing interface. Here the basic actions of the waiter and the Administrator are implemented. Also, here can be found a Javadoc comment for the Design by Contract Pattern. Next, the MenuItem has been created and along side it, the BaseProduct and the CompositeProduct using the Composite Design Pattern. BaseProduct represents products that are not composed of other products and CompositeProduct, products that do.

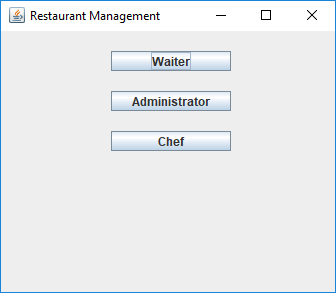
Secondly, the Order class is created. Here I overridden the hashcode() method as requested. Next, we have the Restaurant. Here I implemented all the methods from the RestaurantProcessing interface and a few extra ones to help in my implementation.

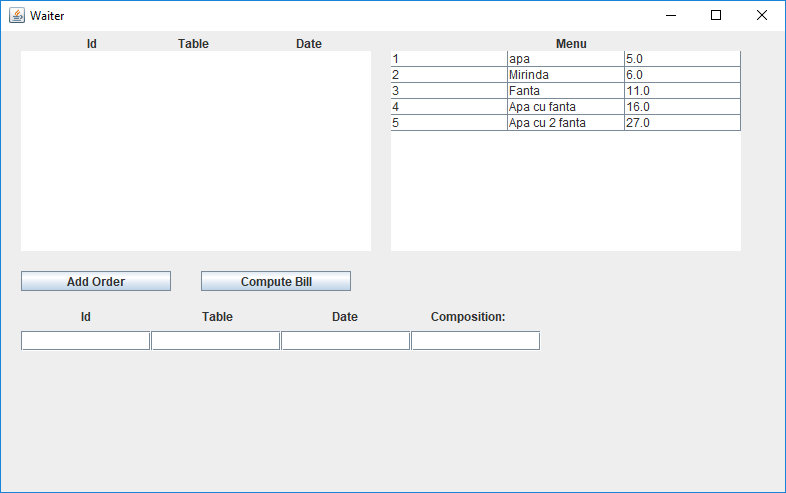
Thirdly, I created the main GUI and after it the rest of the GUIs: Chef, Waiter, Administrator; Chef uses the Observer Pattern to be notified of any new orders, and so does the Waiter, but to know when a change in the menu has occurred.

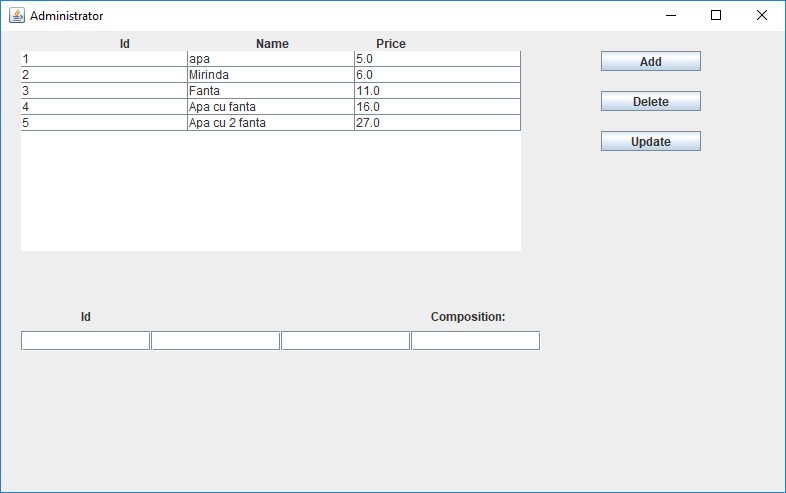
Lastly, I created the RestaurantSerialization class to serialize and deserialize the Menu of the restaurant. Another step was to create the generation of the bills.

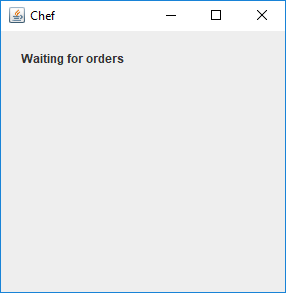
This project has 3 packages: businessL, dataL and presentationL.

Some pictures of the gui:



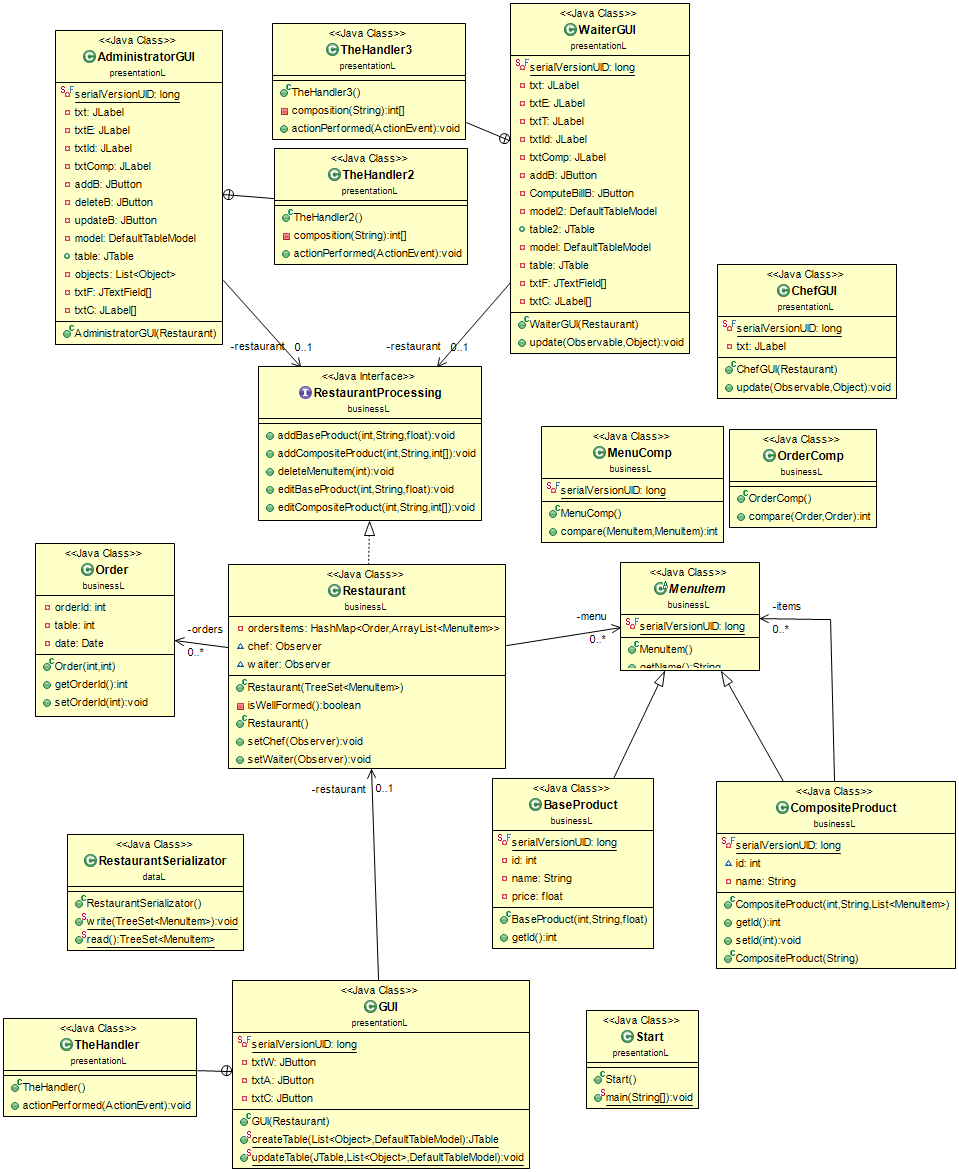






After a successful Bill generation.



****

1. **Implementation**

I believe the most important class used was the Restaurant. Here is its implementation:

**package businessL;**

**import java.awt.Desktop;**

**import java.io.File;**

**import java.io.FileWriter;**

**import java.io.IOException;**

**import java.io.PrintWriter;**

**import java.util.ArrayList;**

**import java.util.HashMap;**

**import java.util.List;**

**import java.util.Observable;**

**import java.util.Observer;**

**import java.util.TreeSet;**

**public class Restaurant extends Observable implements RestaurantProcessing {**

**private TreeSet<Order> orders;**

**private HashMap<Order, ArrayList<MenuItem>> ordersItems;**

**private TreeSet<MenuItem> menu;**

**Observer chef;**

**Observer waiter;**

**public Restaurant(TreeSet<MenuItem> menu) {**

**this.ordersItems = new HashMap<Order, ArrayList<MenuItem>>();**

**this.orders = new TreeSet<Order>(new OrderComp());**

**if (menu == null)**

**this.menu = new TreeSet<MenuItem>(new MenuComp());**

**else**

**this.menu = menu;**

**assert isWellFormed() : "The restaurant has orders which have products that do not exist anymore";**

**}**

**private boolean isWellFormed() {**

**for (Order o : orders) {**

**for (MenuItem a : ordersItems.get(o))**

**if (selectMenuItem(a.getId()) == null)**

**return false;**

**}**

**return true;**

**}**

**public Restaurant() {**

**// TODO Auto-generated constructor stub**

**}**

**public void setChef(Observer o) {**

**chef = o;**

**}**

**public void setWaiter(Observer o) {**

**waiter = o;**

**}**

**public void addBaseProduct(int id, String name, float price) throws AssertionError {**

**assert id > 0 : "Id must be positive";**

**assert !name.equals("") : "Must have a name";**

**assert price > 0 : "Price must be positive";**

**BaseProduct a = new BaseProduct(id, name, price);**

**int sizePre = menu.size();**

**assert menu.add(a) : "The id must not repeat itself";**

**int sizePost = menu.size();**

**assert sizePost == sizePre + 1 : "Size is not ok";**

**if (waiter != null)**

**this.waiter.update(this, null);**

**}**

**private boolean goodProduct(int id, int[] itemsId) {**

**for (int i = 0; i < itemsId.length; i++)**

**if (id == itemsId[i])**

**return false;**

**return true;**

**}**

**public void addCompositeProduct(int id, String name, int[] itemsId) {**

**assert id > 0 : "Id must be positive";**

**assert !name.equals("") : "Must have a name";**

**assert itemsId.length > 1 : "We need at least 2 products";**

**assert itemsId.length < 8 : "We have at most 8 products";**

**assert goodProduct(id, itemsId) : "Cannot have a loop";**

**int sizePre = menu.size();**

**ArrayList<MenuItem> items = new ArrayList<MenuItem>();**

**for (int i = 0; i < itemsId.length; i++) {**

**assert selectMenuItem(itemsId[i]) != null : "There is no product in the menu with id=" + itemsId[i];**

**}**

**for (int i = 0; i < itemsId.length; i++) {**

**items.add(selectMenuItem(itemsId[i]));**

**}**

**CompositeProduct a = new CompositeProduct(id, name, items);**

**assert menu.add(a) : "The id must not repeat itself";**

**int sizePost = menu.size();**

**assert sizePost == sizePre + 1 : "Size is not ok";**

**if (waiter != null)**

**this.waiter.update(this, null);**

**}**

**public MenuItem selectMenuItem(int id) {**

**for (MenuItem a : menu) {**

**if (a.getId() == id) {**

**return a;**

**}**

**}**

**return null;**

**}**

**private boolean noProductHasIt(MenuItem m) {**

**for (MenuItem a : menu) {**

**if (a instanceof CompositeProduct) {**

**for (MenuItem b : ((CompositeProduct) a).getItems()) {**

**if (b.getId() == m.getId())**

**return false;**

**}**

**}**

**}**

**return true;**

**}**

**public void deleteMenuItem(int id) {**

**MenuItem a = selectMenuItem(id);**

**assert a != null : "Item does not exist";**

**assert noProductHasIt(a) : "This Product is used in other Products!";**

**int sizePre = menu.size();**

**menu.remove(a);**

**int sizePost = menu.size();**

**assert sizePost == sizePre - 1 : "Size is not ok";**

**assert isWellFormed() : "The restaurant has orders which have products that do not exist anymore";**

**if (waiter != null)**

**this.waiter.update(this, null);**

**}**

**public void editBaseProduct(int id, String name, float price) {**

**assert price >= 0 : "Price must be positive";**

**assert !name.equals("") : "Must have a name";**

**BaseProduct a = (BaseProduct) selectMenuItem(id);**

**assert a != null : "Item does not exist";**

**if (!a.getName().equals(name))**

**a.setName(name);**

**if (a.computePrice() != price)**

**a.setPrice(price);**

**assert a.getName().equals(name) : "Name is different";**

**assert a.computePrice() == price : "Price is different";**

**if (waiter != null)**

**this.waiter.update(this, null);**

**}**

**public void editCompositeProduct(int id, String name, int[] itemsId) {**

**assert itemsId.length != 1 : "At least 2 products are needed";**

**assert itemsId.length < 8 : "We have at most 8 products";**

**assert goodProduct(id, itemsId) : "Cannot have a loop";**

**ArrayList<MenuItem> items = new ArrayList<MenuItem>();**

**for (int i = 0; i < itemsId.length; i++) {**

**assert selectMenuItem(itemsId[i]) != null : "There is no product in the menu with id=" + itemsId[i];**

**}**

**for (int i = 0; i < itemsId.length; i++) {**

**items.add(selectMenuItem(itemsId[i]));**

**}**

**CompositeProduct a = (CompositeProduct) selectMenuItem(id);**

**assert a != null : "Item does not exist";**

**if (!name.equals(a.getName()))**

**a.setName(name);**

**if (!items.equals(a.getItems()))**

**a.setItems(items);**

**assert a.getName().equals(name) : "Name is different";**

**assert a.getItems().equals(items) : "Items are different";**

**if (waiter != null)**

**this.waiter.update(this, null);**

**}**

**private boolean emptyTable(int table) {**

**for (Order o : orders) {**

**if (o.getTable() == table)**

**return false;**

**}**

**return true;**

**}**

**public void createNewOrder(int orderId, int table, int[] itemsId) {**

**assert orderId > 0 : "Id must be positive";**

**assert itemsId.length > 0 : "Must buy at least one product";**

**assert itemsId.length < 8 : "Can buy at most 8 Products";**

**assert table > 0 && table < 21 : "Table does not exist";**

**assert emptyTable(table) : "Table is taken";**

**ArrayList<MenuItem> items = new ArrayList<MenuItem>();**

**for (int i = 0; i < itemsId.length; i++) {**

**assert selectMenuItem(itemsId[i]) != null : "There is no product in the menu with id=" + itemsId[i];**

**}**

**for (int i = 0; i < itemsId.length; i++) {**

**items.add(selectMenuItem(itemsId[i]));**

**}**

**int preSize = orders.size();**

**Order o = new Order(orderId, table);**

**assert orders.add(o) : "The id must not repeat itself";**

**ordersItems.put(o, items);**

**if (chef != null)**

**this.chef.update(this, o);**

**int postSize = orders.size();**

**assert postSize == preSize + 1 : "The size did not change";**

**assert isWellFormed() : "The restaurant has orders which have products that do not exist anymore";**

**}**

**public Order selectOrder(int id) {**

**for (Order a : orders)**

**if (a.getOrderId() == id)**

**return a;**

**return null;**

**}**

**public void deleteOrder(int id) {**

**Order a = selectOrder(id);**

**assert a != null : "Order not found";**

**orders.remove(a);**

**ordersItems.remove(a);**

**}**

**public float computeTotalPrice(int id) {**

**float price = 0;**

**Order a = selectOrder(id);**

**assert a != null : "Order not found";**

**ArrayList<MenuItem> items = new ArrayList<MenuItem>();**

**items = ordersItems.get(a);**

**for (MenuItem b : items) {**

**price += b.computePrice();**

**}**

**return price;**

**}**

**public ArrayList<Object> getMenu() {**

**if (menu != null)**

**return new ArrayList<Object>(menu);**

**else**

**return new ArrayList<Object>();**

**}**

**public ArrayList<Object> getOrders() {**

**if (orders != null)**

**return new ArrayList<Object>(orders);**

**else**

**return new ArrayList<Object>();**

**}**

**public List<MenuItem> getOrderItems(Order o) {**

**assert o != null : "Order does not exist";**

**assert ordersItems.get(o) != null : "This order does not have a list of Products";**

**return ordersItems.get(o);**

**}**

**public TreeSet<MenuItem> getMenu2() {**

**return menu;**

**}**

**public void generateBill(int orderId) {**

**Order a = null;**

**ArrayList<MenuItem> items = null;**

**for (Order o : orders) {**

**if (o.getOrderId() == orderId) {**

**items = ordersItems.get(o);**

**a = o;**

**}**

**}**

**assert a != null : "Order not found";**

**assert items != null : "List of Products for this order was not found";**

**try {**

**FileWriter fileWriter = new FileWriter("OrderBill" + a.getOrderId() + ".txt");**

**PrintWriter printWriter = new PrintWriter(fileWriter);**

**printWriter.println("SC Simple Restaurant");**

**printWriter.println("Zorilor");**

**printWriter.println("Cluj-Napoca");**

**printWriter.println("");**

**printWriter.println("Bill Table " + a.getTable());**

**printWriter.println("");**

**for (MenuItem i : items) {**

**printWriter.println(i.getName() + "......................" + i.computePrice());**

**}**

**printWriter.println("");**

**printWriter.println("Total.........................." + computeTotalPrice(orderId));**

**printWriter.println("We hope you had a great stay!");**

**printWriter.close();**

**openBill(String.valueOf(orderId));**

**} catch (IOException e) {**

**// TODO Auto-generated catch block**

**e.printStackTrace();**

**}**

**}**

**public void openBill(String id) throws IOException {**

**File file = new File("D:\\All kind of stuff\\me java\\Udemy\\HW4\\OrderBill" + id + ".txt");**

**Desktop desktop = Desktop.getDesktop();**

**if (file.exists()) {**

**desktop.open(file);**

**deleteOrder(Integer.parseInt(id));**

**}**

**}**

**}**

1. **Results**

There are no JUnit tests required in this homework.

1. **Conclusions**

This homework took me more than usual. Not because of its complexity, but because of the amount of new information required for it. Nevertheless, it was very interesting to learn these new Design Patterns. For improvements I would suggest a better user Interface and more Products for the Restaurant, also, a security check for those that use the application.

1. **Biography**

-mostly the sites given in the grading.