FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 4

**RESTAURANT MANAGEMENT SYSTEM**

Teacher: prof. Ioan Salomie

Teacher Assistant: Viorica Chifu

Student: Radu Cosmina

Group: 30424

1. Objective

**Requirement:** Consider implementing 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.

**Main objective**: to develop an application that simulates the actions that need to be done in a restaurant in the perspective of 3 users: the administrator, the waiter and the chef

**Secondary objectives**: use Observable and Serializable, create a user interface, preconditions and postconditions in the IRestaurantProcessing interface. Implement them in the Restaurant class using the assert instruction. Define an invariant for the class Restaurant, se Observer Design Pattern to notify each time a new Order is added, Save the information from the Restaurant class in a file(i.e. restaurant.ser)using serialization.

1. Problem analysis

**General overview:** This application should resemble a restaurant with 3 possible users: the Administrator, who is able to create new menu items, to edit the price of the

existing ones and to delete menu items and also create composite products, the Waiter who is able to create an order taken from the clients at a certain table, he is able to create the order and deliver it to the chef in order to be prepared, to compute the total price of the order and to generate a bill in txt format and the Chef who’s only job is to get

notified by the waiter and to cook the food .

Use case:

A close up of a map

Description automatically generated

viewOrders

showItems

When starting the application the user has 3 option:

->**Administrator** ->*to create a new menu item*: the user sets a name and a price for the product and press the button “Create Item”

-if there is no price, a message with “Invalid price” will come up

-if there is no name, a message with “Invalid name” will show up

-if the fields are correctly written, a message with “Menu item has been created” will show up

->*to delete an existing item*: the user must insert only the name of the product that needs to be deleted and press the button “Delete”

-the inserted price will be ignored

-if there is no item with the given name, a message with “This item doesn’t exist” will show up

-if the item exists, it will be deleted from the menu and also the composite product that contains it

->*to edit an existing item*: the user introduces the name of an item and a new price and then press the button “Edit Item”

-if the given name doesn’t correspond with an existing item, an error message with “Item doesn’t exist” will show up

-if the item exists, it will change its price with the mentioned value and a message with “Item edited successfully “ will show up;

->*to create a new composite product*: the user must type the name of the product in the Product NAME field and press the “New composite product” button

-if the user also types in the price, a message with “Can’t set the price for composite products” will show up;

-a new interface will show up were the user will have to choose from the already existing menu items, the items that will be contained in the composite product and then press the “Add” button

-the contains of the composite product will be shown in a table

->*to view all the products in the menu*: the user will press the “Show items” button

-a table will appear with the names and the prices of all the products

->**Waiter ->** *to create a new order*: the user must type the table number for the orders, and then choose from a list of menu items , the products for the order and then press “Add” (the order id is generated automatically, the user doesn’t need to type it”

-after all the items have been added, the user must press the “Create order” button

-if any of these fields are not completed, a message with “Error creating order” will show up

-if all the fields are correct, a message with “Order created successfully” will show up

->*to generate the bill*: the user must type in the “ID” field the id of the order for which the bill must be generated and then press the “Generate bill” button

-if the typed in id doesn’t correspond to an existing order, a message with “This order doesn’t exist” will show up

-if the order exists, a txt file will be generated with the order details and a message with “Bill generated successfully” will show up

->*to view all the orders*: the user must press the “Show orders” button and a table with all the orders (with their details) will show up

->**Chef** ->after an order is created successfully, a notification will be sent to the chef

-the chef interface will appear with a pop up message : “Notification : New order: items”

-this notification is saved in the chef interface for being “processed”

1. Design

Packages: organized in a Model- View-Controller pattern

A circuit board

Description automatically generated

Model: which represents the underlying, logical structure of data in a software application and the high-level class associated with it. This object model does not contain any information about the user interface. This is the package where the models for the restaurant are created: order, base product, composite product, menu items and the id generator

View: which is a collection of classes representing the elements in the user interface (all of the things the user can see and respond to on the screen, such as buttons, display boxes, and so forth). This package contains all the user interface classes: administrator interface, waiter interface, chef interface, composite product interface and the starting user interface;

Controller: which represents the classes connecting the model and the view, and is used to communicate between classes in the model and view. This package contains the classes: main controller, restaurant and restaurant serializator.

Process: this package contains the IRestaurantProcessing class;

Main: this package contains the class Mani that starts the applications.

Classes:

A screenshot of a computer

Description automatically generated

Main: this class contains the main method which calls the start function in order to start the application;

RestaurantSerializator: this class implements the serialize and deserialize functions for the application;

Order: this is the model class for the orders; it has as attributes the order id, the date, the table and the items; this class also contains the hashcode function;

MenuItems: implements java.io.Serializable; this is an abstract class that will be used for the other classes: Product and Composite Product; it has the attricbutes name and price and has declared the method computePrice;

Product: this class represents the base product from the menu; it extends the MenuItems class and overrides the computePrice methods and has a constructor for initializing the attributes name and price;

CompositeProduct: this is the composite products class which uses the classes MenuItems and Product; it overrides the method computePrice and is actually a list of base products;

GenerateID: this class computes the needed id for the orders using a static variable count which increments each time an order is being made

IRestaurantProcessing: this interface declares the main methods that need to be used in the graphical interfaces

Restaurant: extends Observable and implements IRestaurantProcessing and java.io.Serializable; The class that is the center of this application, every action must at least pass through the restaurant if not executed by it. It implements the IRestaurantProcessing interface and all the methods present there. It is also built using Design by Contract, testing the preconditions and postconditions with assert instructions in each implemented method.

MainController: this class initializes all the user interfaces and sets up the environment for the application;

UserInterface: this is the first graphical interface that appears when strating the application; it is connected to the other interfaces through the 3 button options: Administrator, Waiter, Chef;

AdminInterface: extends JFrame and implements IRestaurantProcess; this is the class that allows modifications to the menu items : creating, deleting and editing them; It acts as an administrator (thus the name) and is also connected to the Composite product interface; The administrator can also view the Menu and decide if he / she should add or not more products.

CompProductInterface: this class creates a new composite product; it is used only form the administrator interface;

WaiterInterface: extends JFrame and implements IRestaurantProcessing; this class generates the orders made with the menu items created by the administrator; each time an order is being made, a the chef is notified in the form of a message with the new order; the waiter also generates the bill in a txt format, each order has its own bill in separate txt files;

ChefInterface: extends JFrame and implements Observable; this is where the notification of a new order is being sent;

Graphical User Interface:

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

Used data structures:

ArrayList – I chose to use this structure to keep my queues and to temporary hold my clients, because this kind of lists is useful for storing and accessing data ArrayList internally uses dynamic array to store the elements. Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. This class can act as a list only because it implements List only.

Iterator: is an interface which belongs to collection framework. It allows us to traverse the collection, access the data element and remove the data elements of the collection.

Collection: is a framework that provides an architecture to store and manipulate the group of objects. Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion. Java Collection means a single unit of objects

Serializable : Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object. To make a Java object serializable we implement the **java.io.Serializable** interface.  
The ObjectOutputStream class contains **writeObject()** method for serializing an Object. The ObjectInputStream class contains **readObject()** method for deserializing an object.

Observable: Observer is a behavioral design pattern. It specifies communication between objects: observable and observers. **An** observable **is an object which notifies** observers **about the changes in its state.**

1. Implementation

Main methods:

RestaurantSerializator-> serialize: this methods saves the data written in the interface in a file Restaurant.ser;

->deSerialize: this method receives the data previously saved in the Restaurant.der file for further use;

Restaurant -> createMneuItem: this method is done by the administrator, it creates a new base product; it has tested pre and post conditions using assert

->deleteMenuItem: this method is done by the administrator, it deletes an existing item from the menu and also the composite product that contains it;

->editMenuItem: this method is done by the administrator, it edits the price of an existing base product;

->createOrder: this method is done by the waiter, it creates an order for a specific table with the products from the menu, generated by the administrator;

A close up of a logo

Description automatically generated ->addOrder-> each time an order is created, the chef is notified with the order informations (chef being an observer)

->priceOrder: this method computes the final price for an order based on the selected menu items;

->generateBill: this method is done by the waiter, based on the selected id , it generates a txt file with the order details and the final price; each bill is saved in a different txt file using the id of the order: 

Order: ->this class overrides the hashcode function A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generatedGUI: All the interfaces have a “Back” button which will bring the user back to the main interface (UserInterface) :

Buttons -> “Create item” ->generates a new base product for the menu ; it then calls the serialize method from the Restaurant class:

A screenshot of a cell phone

Description automatically generated

->”Delete item”->deletes an item from the menu using the “findByName” function

->”Edit item”->changes the price of an item from the menu using the “ findByName” function

->”New composite product” ->sets visible the Composite Product Interface and displays an error if the price is also typed in: A screenshot of a cell phone

Description automatically generated

->”Show items/orders” ->when these buttons are pressed, the data saved in the Restaurant.ser file is deserialized and used as object to populate the tables in order to show the needed items: 

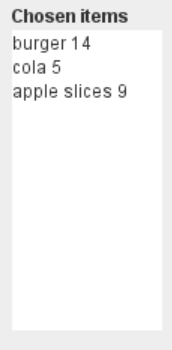
A picture containing food

Description automatically generated

A screenshot of a cell phone

Description automatically generated

Text fields: “Chosen items”: these text fields are present in the WaiterInterface and in the CompProductINterface; here the user can see the selected items from the menu in order to create a new composite product or a new order; after the user is satisfied with the chosen items, by pressing the button “Finish composite product” or “Create order” , the objects are created based on the data and items the user has inserted



1. Results

The application works as it should: no exception is being called during the simulation, each time an action is being made, an error/ success message is being shown to the user, the interface is easy to use and understand. The implementation follows the instruction and the given diagram as for the project structure.

1. Conclusions

This project has been a good opportunity to exercise my OOP skills such as data manipulation , MVC patterns, Interface implementing and so on. I managed to improve my knowledge about the Composite and Observer design patters, as well as the Serialization and Deserialization mechanisms. These are very useful mechanisms which I will also use for the future projects.

1. Bibliography

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

<https://www.baeldung.com/java-observer-pattern>

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

<https://www.javatpoint.com/collections-in-java>

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

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

<https://www.javatpoint.com/serialization-in-java>

<https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html>

<https://www.edureka.co/blog/serialization-in-java/>

<https://www.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html>

<https://www.guru99.com/java-swing-gui.html>