Restaurant Management System

Fundamental Programming Techniques (PT)

Project #4

Motioc Alexandru

G: 30424

Contents

1. Objective
2. Problem analysis
3. Structure / Diagram
4. Code

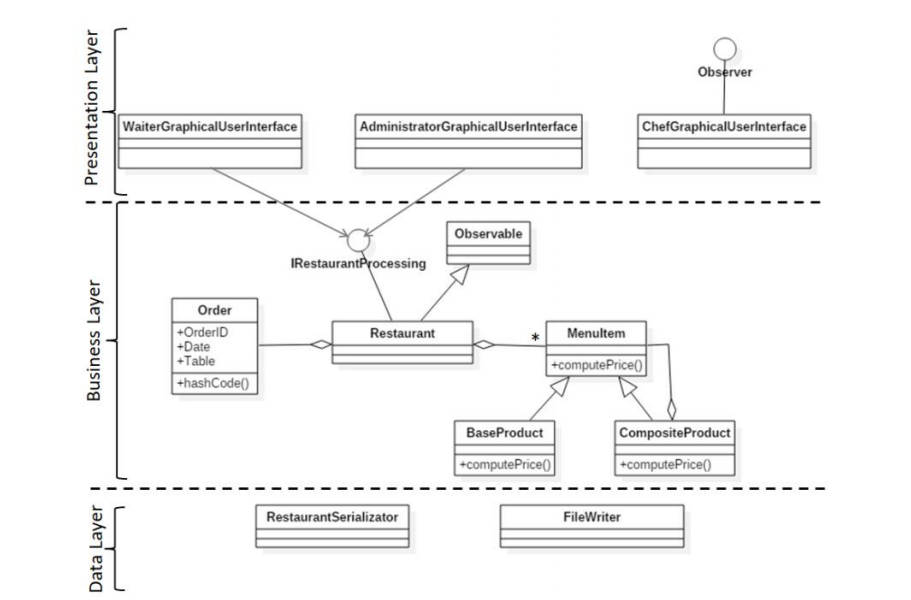
5. Conclusion

Word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by word by

1. Objective

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 ordered through a waiter.

Consider the system of classes in the diagram below.



To simplify the application you may assume that the system is used by only one administrator, one waiter and one chef, and there is no need of a login process. Solve the following:

1. Define the interface RestaurantProcessing containing the main operations that can be executed by the waiter or the administrator, as follows:

• Administrator: create new menu item, delete menu item, edit menu item

• Waiter: create new order; compute price for an order; generate bill in .txt format.

2. Define and implement the classes from the class diagram shown above:

• Use the Composite Design Pattern for defining the classes MenuItem, BaseProduct and CompositeProduct

• Use the Observer Design Pattern to notify the chef each time a new order containing a composite product is added.

3. Implement the class Restaurant using a predefined JCF collection which uses a hashtable data structure. The hashtable key will be generated based on the class Order, which can have associated several MenuItems. Use JTable to display Restaurant related information.

• Define a structure of type <Map> for storing the order related information in the Restaurant class. The key of the Map will be formed of objects of type Order , for which the hashCode() method will be overwritten to compute the hash value within the Map from the attributes of the Order (OrderID, date, etc.)

• Define a structure of type Collection which will save the menu of the restaurant. Choose the appropriate collection type for your implementation.

• Define a method of type “well formed” for the class Restaurant.

• Implement the class using Design by Contract method (involving pre , post conditions , invariants , and assertions).

4. The menu items for populating the Restaurant object will be loaded/saved from/to a file using Serialization.

Secondary Objectives:

• Creating an activity log and save information of the Restaurant objects using serialization mechanism.

Inspiration source: <https://www.tutorialspoint.com/java/java_serialization.htm>

• Using composite design patterns in order to represent the menu’s items as two types of objects:

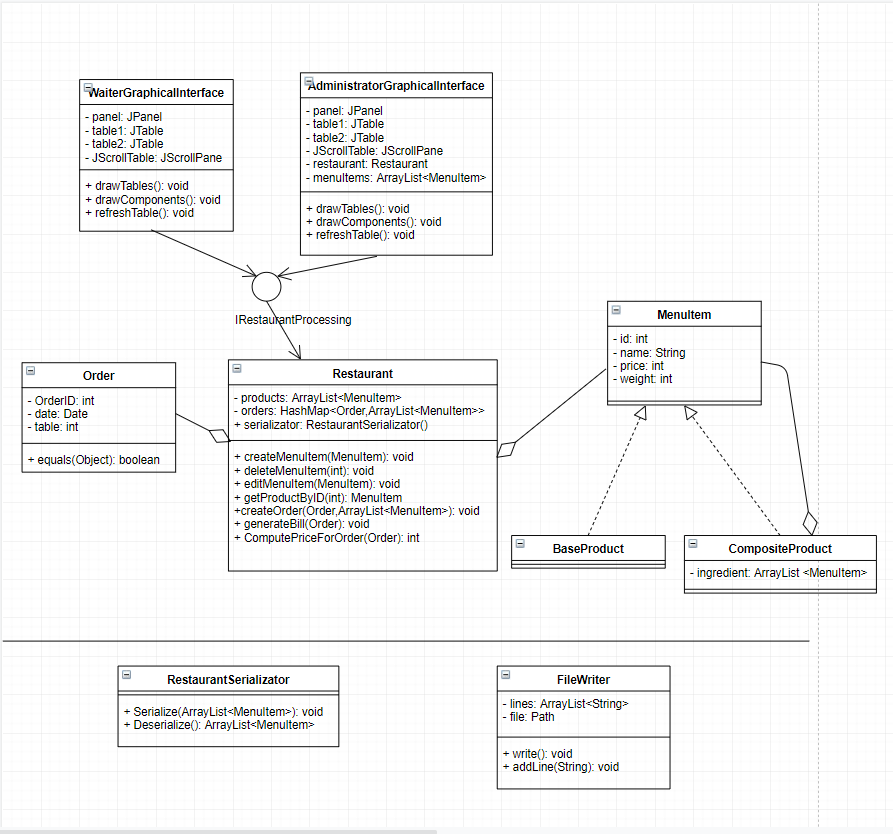
Base objects and composite objects (a composite object can contain other composite objects or base objects)

• Waiter can create and format bills in .txt format

• Window for Chef user: use Observer Design Pattern to notify each time a new Order is added

• Design by contract: preconditions and postconditions in the RestaurantProcessing interface.

UML diagram



2. Problem Analysis

For this project we are going to define each product in the restaurant’s menu as a MenuItem type object. MenuItem class is going to be divided into two child classes: BaseProduct and CompositeProduct.

Basically, a BaseProduct is an item with no ingredients. While a CompositeProduct is an item made out of BaseProduct’s and CompositeProducts. We are going to use Composite Design Pattern to describe these groups of objects into tree structures to represent part-whole hierarchies. In our program, the price and weight of a CompositeProduct is going to be computed based on the price / weight of the items they are composed of. For example:

* We have: Cheese[Price: 3 / Weight: 10] , Ham[Price: 5 / Weight: 8] , Bun[Price: 3 / Weight: 6], Salad[Price: 4 / Weight: 1] , Broccoli[Price:4 / Weight: 3]
* We are going to create a new item – Sandwich, that is going to contain Cheese,Ham and Bun
* Using the computeWeight and computePrice methods in compositeProduct class, we are going to serialize the new menu item as: Sandwich[Price: 11 / Weight: 24]

Additionally, the administrator of the restaurant can: add, remove or edit the items situated in the menu.

This program will also a generate a window for the restaurant’s waiter. Our waiter can: create an order, generate bill in a .txt file. The waiter can create a new order by entering the id of the order, the date of the order and the table requesting said order. This is where we use our Order model class.

In case of unsuccessful scenarios, we are going to use assert statements in order for the program to register in console’s log the failure of respecting the asked parameters of our graphical interface.

1. Successful scenario:

* User presses the desired command button in the administrator window( i.e.: ADD, REMOVE, EDIT)
* Price entered for the menu items are greater than 0
* Status log saves the executed command

1. Unsuccessful scenario:

* Users introduces foul type of input data in a certain column (e.g.: introduces a string in age column)
* Orders are not introduced as null values
* Error message log is displayed
* The program can’t successfully parse the introduced data

3. Structure

The application is divide into 11 classes: BaseProduct, CompositeProduct, IRestaurantProcessing, MenuItem, Order, Restaurant, FileWriter, RestaurantSerializator, Main, AdministratorGraphicalUserInterface, WaiterGraphicalUserInterface.

Additionally, the classes contained in this application are divided into 4 packages:

1. BussinessLayer

This package contains the models the logic behind the functions that the administrator can execute as main operations, same can be applied to the waiter case. IRestaurantProcessing is defined as interface containing the definitions of the methods that implement the logic behind the program. Also, the objects that the application works with are defined in this package, mainly within menuItem which is divided into the child classes: BaseProduct, CompositeProduct. Overall, the classes contained within BussinessLayer are: BaseProduct, CompositeProduct, IRestaurantProcessing, Order, MenuItem, Restaurant.

2. DataLayer

DataLayer contains our FileWriter class, which is used by our generateBill method situated in the Restaurant class, in order to print the details of each order on a .txt file. RestaurantSerialization stores each object contained in the database into a temp file, which can be used to later deserialize and write the prior information that was saved in previous uses of the application. Each change in the HashMap and menuItem ArrayList is going to be serialized as a way to be memorized by the program.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types

3. Main

Main class executes the GUI classes and displays the application in order to be used by a user.

4. PresentationLayer

This is where the graphical interface is formatted and where the logic of the application is introduced(behind actionListeners). DrawTables is going to initialize the tables of our menu items by introducing the deserialized information into the the data matrix and afterwards implemented in the JTable. In the drawComponents method, the actionListeners (add, remove, edit, generateBill) that are going to be accessible to either administrator or waiter, are being implemented.

4. Code implementation

The core objects that are going to be used by the logic of our application is situated in the BLL package, based on the MenuItem class, which also inherits the BaseProduct and CompositeProduct. By using the Composite Design Pattern, the menu items are represented as a binary tree composed by BaseProducts and CompositeProduct. BaseProducts are always going to be leaves of the tree data structure, while CompositeProduct is going to have children’s that can be either BaseProducts or other CompositeProducts (that are also inherently composed by other items). CompositeProduct contains an overwritten toString() method, that returns the information about said product in a string enlisting every asked element(id, price, name, weight). This is essentially going to be useful when generating a bill and introducing it in a .txt file. Moreover, compositeProduct contains computeWeight and computePrice methods, which are used in order to calculate the price and weight of a composite product, by adding the values of the ingredient’s attributes.

Restaurant contains the main logic of our application, by defining the main methods that are going to be embedded into the gui, such as: createMenuItem, editMenuItem, generateBill etc.

The orders created within the application are going to be noted in a HashMap of

Order-type keys, and elements as ArrayLists of menuItems (which could contain either CompositeProducts or BaseProducts). Everytime a change is made in the structure of our database, the changes is remembered by the program by serializing the changed objects. For example, if product is being added in the products ArrayList<MenuItem>, then we are going to call: serializator.Serialize(products);

Assert techniques are also present throughout the methods implemented within the Restaurant class, in order to keep track of unsuccessful operations commanded by the user.

AdministratorGUI class is composed by two JTables and 3 JButtons which command the changes demanded towards our table. The tables are:

1. The Menu table

2. The ingredients table, where the user can change the ingredients of e certain item in the menu.

When pressing the add button, first the application searches for desired ingredients in the second table, and afterwards checks if the introduced menu item is classified as a baseProduct or compositeProduct based on the results of the search. There will be searched for the parsed values introduced in the ingredients table, and if they exist, they will be added to the certain menu item. If no ingredients are found in the menu item’s arrayList, then it is considered a baseProduct. After the changes are made, the table is going to be refreshed by using the TableRefresh method. TableRefresh() first removes all the elements contained within the panel then it calls again the drawTables() and drawComponents methods in order for the new changes to be shown in the table, without having to close and re-opening the application.

The waiter’s add order command works in a similar way as the administrator’s add item command, however, a waiter can generate a bill on the already created orders, using FileWriter’s methods.

Composite pattern is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies. It allows you to have a tree structure and ask each node in the tree structure to perform a task.

Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

5. Conclusion

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.