**Assignment 4, PT**

**Andriesă Maria**

**-Restaurant Management-**

**1.Objectives and Specifications**

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.

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<Order, Collection<MenuItem>> 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<MenuItem> 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.

**2.Analysis ,Modelling, Use cases Scenarios**

2.1 Analysis

The application that has been implemented provides a comfortable environment for managing a restaurant, providing functionalities for an administrator and the waiter. Due to the complexity of menu, which has basic foods and special menus made up of basic foods, the Composite Pattern has been used because it allows us to compose objects into tree structures to represent part-whole hierarchies and, thus, lets clients to treat individual objects and compositions of objects uniformly. Also, because there exists a chef who should always receive the orders when they are made, the Observer Pattern has been used to notify the chef of every new incoming order. This pattern defines a one-to-many dependency between objects so that when one object’s state changes, all its dependents are notified and updated automatically.

In order to maintain a fairly loose coupling, there has been used an interface RestaurantProcessing which holds the headers from all the functions in Restaurant that are needed to communicate with the Controller and the views.

2.2 Modelling

The modelling of this application has been done having as guideline the diagram given in the project specification. The structure given has not been altered, only added to. The “business” package holds the classes for the main structure of the application , namely the BaseProduct, CompositeProduct, MenuItem, Order, Restaurant and RestaurantProcessing classes. The data package holds the classes for data serialization and deserialization, that is RestaurantSerializator, and the class for the connection to gmail servers for sending and email with the order bill, also holds the bill writer, class BillGenerator. I’ll explain how serialization works in Java:

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.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

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, but one method in particular stands out −

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object –

public final Object readObject() throws IOException, ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

As such I have serialized the Restaurant class, but only the list of menu items and the map of orders from that class. I only desearilize the file once, at the start of the application.

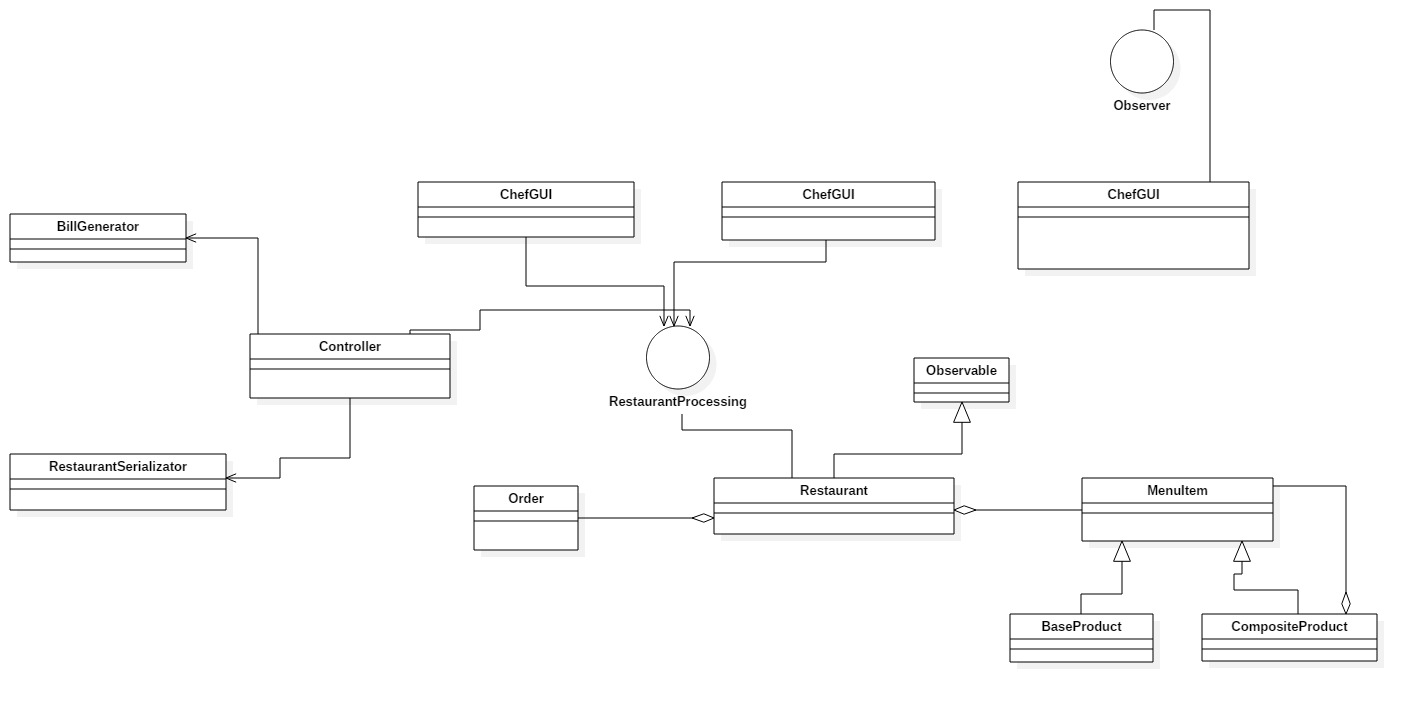
**Use cases and Scenarios**

**Use case #1**

1. The application is started and a panel with two choices comes up. The user can select if he has the role of administrator or waiter.
2. If administrator is selected, a window with the current menu items will appear.
3. Administrator can insert a new menu item. He has to just complete the fields with the name, description and price of the new item that has to be inserted.
4. The user click on new Base Product, if the product he inserts is jus an item and not a special menu composed of many individual items.
5. Else, if he wants to insert a Composite Product he has to choose from the drop down a base product from which he wants to make a new menu.
6. After he has chosen the product he has to click the plus button which will represent the fact that the item has been added to the composition.
7. After he selects all the items that he needs to be in the composition, the user must write a name for the new menu. He must not insert anything in the description and price fields, as they will be automatically filled.
8. Finally, he clicks on the Composite Product product to add it to the menu.

**Use case #2**

1. . The application is started and a panel with two choices comes up. The user can select if he has the role of administrator or waiter.
2. If the user is the Waiter than a window will appear with a drop down list with all the items in the menu, a button for inserting a new order and a table with the orders.
3. The waiter selects the product from the drop down and clicks the plus button to add the item to the order.
4. After selecting all the requested products by the client, the waiter inserts the table number of the customers that made the order and then click on New Order.
5. Once New Order is clicked, it will add the order in the map and display its id, date and table number in the table. Also, it will generate a bill in .txt format and will send that file via email to the customer.
6. Also, the new incoming order will notify the chef, and the chef , when receives the notification, will make a window to pop and display a message “Incoming order”.

**3.Class Diagram**

**4.Decisions**

In the following paragraphs, I will describe how each design pattern has been used to build the respective classes:

MenuItem : defines an interface for all the objects in the composition: both the composite and the leaf nodes. It may implement a default behavior for add(), remove(), getChild() and its operations.

BaseProduct: it represents a leaf of the structure and it has no children. A leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports. Note that the BaseProduct also inherits methods like add(), remove() and getChild() , which won’t make much sense for a leaf node. It holds the fields to describe one element of the menu like name, description of ingredients and price.

CompositeProduct: its role is to define the behavior of the components having children and to store child components. It also implements the BaseProduct-related operations. So, some of these may not make sense on a Composite, as such an exception might be generated of type “Unsupported Operation”.

Restaurant: is the client class which uses the MenuItem interface to manipulate objects in the composition. It holds a Map where the orders are stored with the key being the Order class, and the value is represented by a list of the ordered items. It also holds an Array List of Menu Items and an Array List of observers. Due to the inheritance of Observable class , it overrides the addObserver method and , every time when an order is inserted the Chef is notified and shows a message that a new order is incoming.

SMTP protocol for EmailSender: 🡪Mail processing model

Email is submitted by a mail client (mail user agent, MUA) to a mail server (mail submission agent, MSA) using SMTP on TCP port 587. Most mailbox providers still allow submission on traditional port 25. The MSA delivers the mail to its mail transfer agent (mail transfer agent, MTA). Often, these two agents are instances of the same software launched with different options on the same machine. Local processing can be done either on a single machine, or split among multiple machines; mail agent processes on one machine can share files, but if processing is on multiple machines, they transfer messages between each other using SMTP, where each machine is configured to use the next machine as a smart host. Each process is an MTA (an SMTP server) in its own right.

The boundary MTA uses the Domain name system (DNS) to look up the mail exchanger record (MX record) for the recipient's domain (the part of the email address on the right of @). The MX record contains the name of the target host. Based on the target host and other factors, the MTA selects an exchange server: see the article MX record. The MTA connects to the exchange server as an SMTP client.

Message transfer can occur in a single connection between two MTAs, or in a series of hops through intermediary systems. A receiving SMTP server may be the ultimate destination, an intermediate "relay" (that is, it stores and forwards the message) or a "gateway" (that is, it may forward the message using some protocol other than SMTP). Each hop is a formal handoff of responsibility for the message, whereby the receiving server must either deliver the message or properly report the failure to do so.[14]

Once the final hop accepts the incoming message, it hands it to a mail delivery agent (MDA) for local delivery. An MDA saves messages in the relevant mailbox format. As with sending, this reception can be done using one or multiple computers, but in the diagram above the MDA is depicted as one box near the mail exchanger box. An MDA may deliver messages directly to storage, or forward them over a network using SMTP or other protocol such as Local Mail Transfer Protocol (LMTP), a derivative of SMTP designed for this purpose.

Once delivered to the local mail server, the mail is stored for batch retrieval by authenticated mail clients (MUAs). Mail is retrieved by end-user applications, called email clients, using Internet Message Access Protocol (IMAP), a protocol that both facilitates access to mail and manages stored mail, or the Post Office Protocol (POP) which typically uses the traditional mbox mail file format or a proprietary system such as Microsoft Exchange/Outlook or Lotus Notes/Domino. Webmail clients may use either method, but the retrieval protocol is often not a formal standard.

SMTP defines message transport, not the message content. Thus, it defines the mail envelope and its parameters, such as the envelope sender, but not the header (except trace information) nor the body of the message itself. STD 10 and RFC 5321 define SMTP (the envelope), while STD 11 and RFC 5322 define the message (header and body), formally referred to as the Internet Message Format.

**Bibliography**

<https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol>

**8.Bibliography**

1. Head First Design Patterns- A Brain-Friendly Guide,By Bert Bates, Kathy Sierra, Eric Freeman, Elisabeth Robson