RESTAURANT MANAGEMENT SYSTEM

Fărcaș Alexandru
Tehnici de programare 2019

Cuprins

Obiectivul temei

TP Lab–Homework 4

- Objectives
- Design by Contract Programming Techniques
- Polymorphism
- Design Patterns: Observer, Composite
- JCF HashMap and HashSet implementations
- Serialization
- Description

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 datastructure. 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.

Mandatory requirements for accepting the assignment:

- OOP paradigms
- Classes of maximum 200 lines (except the UI classes)
- Methods of maximum 30 lines
- Java naming conventions https://google.github.io/styleguide/javaguide.html

Problem analysis

As requested, the following app must give the user the possibility of processing the orders in a restaurant environment. Therefore, I implemented three types of users:

- Manager
- Chef
- Waiter

Given their attributes, each one of them can perform actions over the restaurant’s data, such as the menu, the list of products, managing the orders or just observing them in case of the chef.

The project

The application must use a Composite Design Pattern. In software engineering, the composite pattern is a partitioning design pattern. The composite pattern 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. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

The Composite design pattern is one of the twenty-three well-known GoF design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Composite design pattern solve?

A part-whole hierarchy should be represented so that clients can treat part and whole objects uniformly.

A part-whole hierarchy should be represented as tree structure.

When defining (1) Part objects and (2) Whole objects that act as containers for Part objects, clients must treat them separately, which complicates client code.

What solution does the Composite design pattern describe?

Define a unified Component interface for both part (Leaf) objects and whole (Composite) objects.

Individual Leaf objects implement the Component interface directly, and Composite objects forward requests to their child components.

This enables clients to work through the Component interface to treat Leaf and Composite objects uniformly: Leaf objects perform a request directly, and Composite objects forward the request to their child components recursively downwards the tree structure. This makes client classes easier to implement, change, test, and reuse.

The use of a Observer Design Pattern is also required. The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

It is mainly used to implement distributed event handling systems, in "event driven" software. Most modern languages such as C# have built-in "event" constructs which implement the observer pattern components.

The observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern. The observer pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits.

While the library classes java.util.Observer and java.util.Observable exist, they have been deprecated in Java 9 because the model implemented was quite limited.

Below is an example written in Java that takes keyboard input and treats each input line as an event. When a string is supplied from System.in, the method notifyObservers is then called, in order to notify all observers of the event's occurrence, in the form of an invocation of their 'update' methods.

Implementation

MENUITEM.JAVA INTERFACE

```
package businessLayer;

import java.io.Serializable;

public interface MenuItem extends Serializable {
    public int computePrice();
}
```

The MenuItem interface asks for a computePrice() method to be implemented.

BASEPRODUCT.JAVA

```
package businessLayer;

import java.io.Serializable;

public class BaseProduct implements MenuItem, Serializable {

    private int price;
    private String name;

    public BaseProduct() {
    }

    public BaseProduct(int price, String name) {
        this.price = price;
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    @Override
    public int computePrice() {
        return price;
    }
}
```

The data is stored in a BaseProduct class type or a CompositeProduct. Both of them implement the requested computePrice() method, and have a name and a price variable, that are later going to be displayed in a tableview for the user.

RESTAURANT.JAVA

```
package businessLayer;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Observable;

public class Restaurant extends Observable implements Serializable {
    private HashMap<Order, ArrayList<MenuItem>> orders = new HashMap<>();
    private ArrayList<MenuItem> products = new ArrayList<>();

    public Restaurant() {
    }

    public Restaurant(HashMap<Order, ArrayList<MenuItem>> orders,
ArrayList<MenuItem> products) {}

    public HashMap<Order, ArrayList<MenuItem>> getOrders() {
        return orders;
    }

    public void setOrders(HashMap<Order, ArrayList<MenuItem>> orders) {
        this.orders = orders;
    }

    public ArrayList<MenuItem> getProducts() {}

    public void setProducts(ArrayList<MenuItem> products) {}

    public Order getOrderById(int id){ }

    public Order getOrderByTable(int table){ }

    public CompositeProduct getProductByName(String name){ }
}
```

The restaurant class is responsible for storing the entire data used by the application. This includes a list of products and a HashMap of orders to be proccessed.

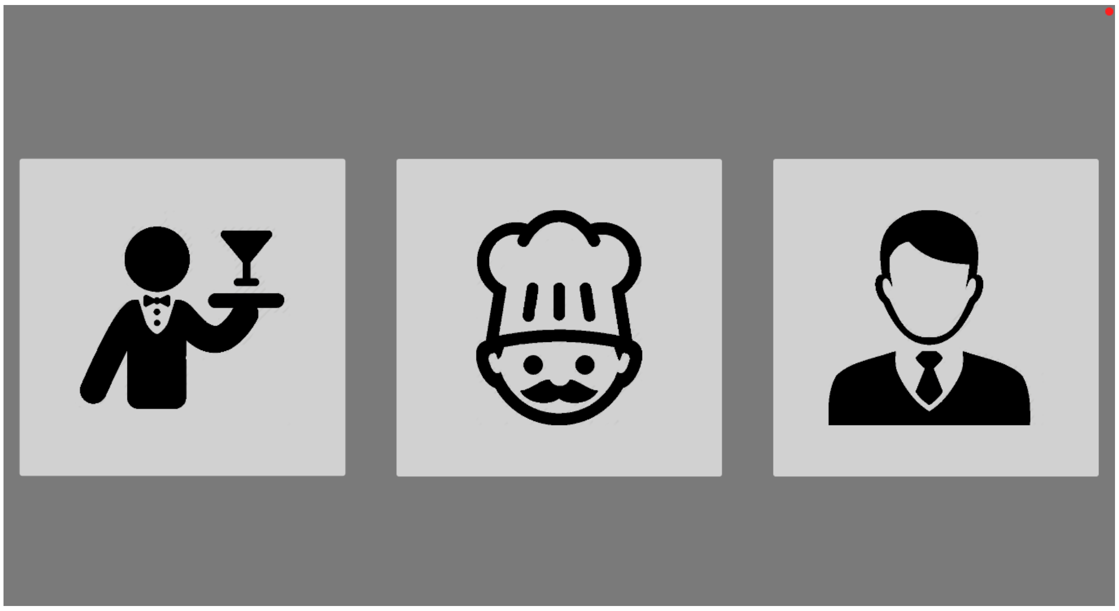
The implementation requested for a serialization and a deserialization way forstoring and restoring the data stored in this class.

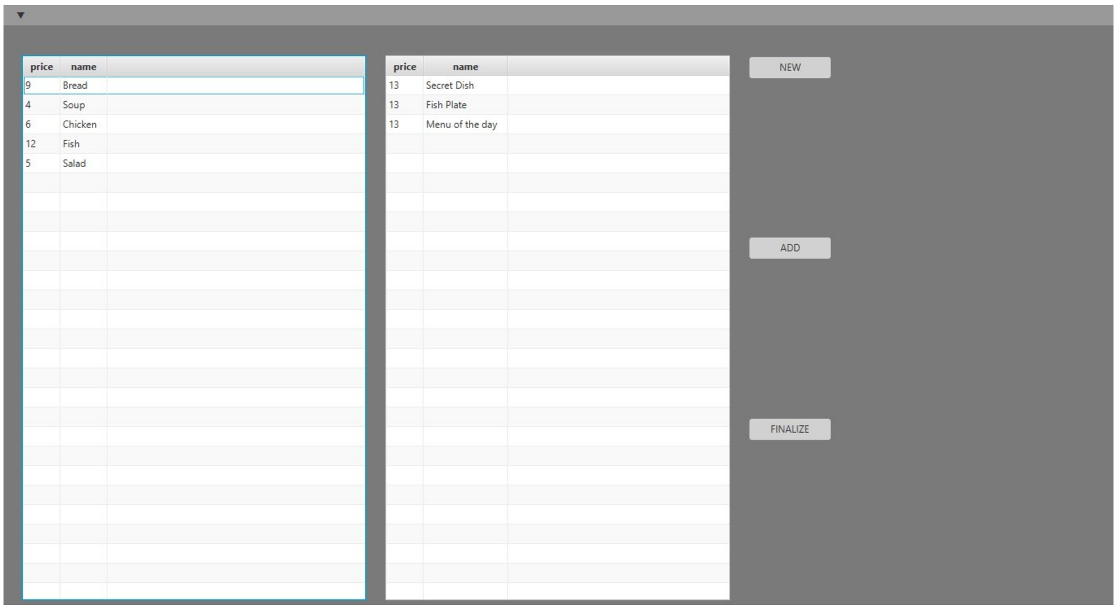
RESTAURANTSERIALIZATOR.JAVA

```
package dataLayer;

import businessLayer.*;

import java.io.*;

public class RestaurantSerializator {

    public void save(Restaurant restaurant){
        try {
            FileOutputStream fileOutputStream = new
FileOutputStream("restaurant.bit");
            ObjectOutputStream objectOutputStream = new
ObjectOutputStream(fileOutputStream);
            objectOutputStream.writeObject(restaurant);
            objectOutputStream.flush();
            objectOutputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public Restaurant load(){
        Restaurant restaurant = null;
        try {
            FileInputStream fileInputStream = new
FileInputStream("restaurant.bit");
            ObjectInputStream objectInputStream = new
ObjectInputStream(fileInputStream);
            restaurant = (Restaurant) objectInputStream.readObject();
            objectInputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        if(restaurant == null){
            return new Restaurant();
        }
        return restaurant;
    }

}
```

THE GRAPHICAL INTERFACE

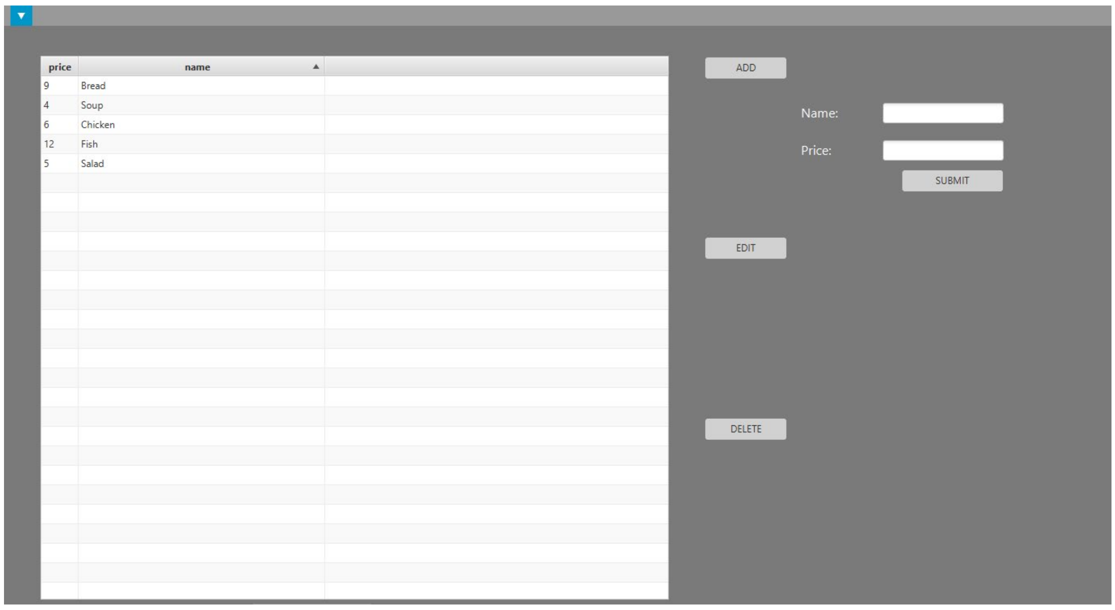
The implemented graphical interface uses JAVAFX and CSS for a user friendly environment.





price	name
9	Bread
4	Soup
6	Chicken
12	Fish
5	Salad

price	name
13	Secret Dish
13	Fish Plate
13	Menu of the day



price	name
9	Bread
4	Soup
6	Chicken
12	Fish
5	Salad

Results

This app could be easily adapted for commercial use. The design is creaded for multiple users, whit accessible commands for the wide useage.

Bibliography

→ http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/HW4_Tema4/Barem_HW4
→ https://en.wikipedia.org/wiki/Observer_pattern
→ https://en.wikipedia.org/wiki/Composite_pattern