MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

RESTAURANT MANAGEMENT

ASSIGNMENT 4

Documentatie

MOLDOVAN ADELINA-STEFANIA

GRUPA 30226

CUPRINS

1. Obiectivul temei

Obiective principale

Cerința problemei: Luați în considerare implementarea unui sistem de management al restaurantelor. Sistemul ar trebui să aibă trei tipuri de utilizatori: administrator, chelner și bucătar-șef. Administratorul poate adăuga, șterge și modifica produsele existente din
meniu. Chelnerul poate crea o comandă nouă pentru o masă, poate adăuga elemente din meniu și poate calcula factura pentru o comandă. Bucătarul este informat de fiecare dată când trebuie să gătească alimente comandate prin intermediul unui chelner.

Structura aplicatiei:

- Business Logic classes – logica aplicatiei;
- Presentation classes – clasele care contin interfata cu utilizatorul;
- Data access classes – clasele in care se face accesul la date (accesul la datele salvate in fisiere);

Obiective secundare

1 . - dezvoltarea de use case-uri : folosim use case-uri (caz de utilizare) pentru a reprezenta cerinţe ale utilizatorilor. Sunt descrise acţiuni pe care un program le execută atunci când interacţionează cu actori şi care conduc la obţinerea unui rezultat .

2 . - dezvoltarea de diagrame UML : folosim diagrame UML pentru a înregistra relaţiile dintre clase .

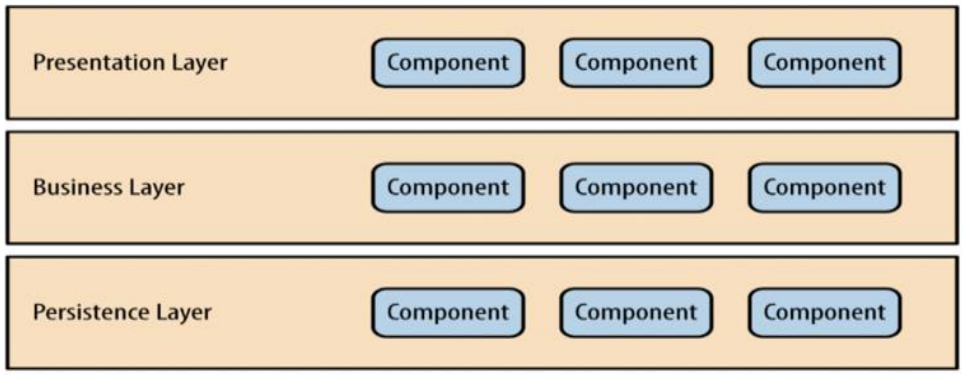
3 . - implementarea unor clase in java : sunt descrise clasele folosite si rolul acestora .

4 . – dezvoltarea algoritmilor : procesul de dezvoltare al algoritmilor utilizati pentru acest proiect .

5. – utilizarea javadoc-urilor pentru documentarea claselor

6. - Layered architecture – este un model de arhitectura cu n nivele, unde componentele sunt organizate in straturi orizontale. Aceasta este o metoda traditionala pentru proiectarea mojoritati aplicatiilor software si este menita sa fie independenta de sine. Aceasta inseamna ca toate componentele sunt interconectate, dar nu depind unele de altele.

Layered architecture – tipul de arhitectura folosit pentru implementarea aplicatiei



PresentationLayer – contine clasele care creeaza GUI.

BussinesLayer – contine clasele care descriu logica aplicatiei.

DataBaseLayer – contine clasele care definesc logica pentru accesul la datele aplicatiei.

2.Analiza problemei, modelare, scenarii, cazuri de utilizare

Use case-uri

O diagramă use case este una din diagramele folosite în UML pentru a modela aspectele dinamice ale unui program alături de diagrama de activităţi, diagrama de stări, diagrama de secvenţă şi diagrama de colaborare .

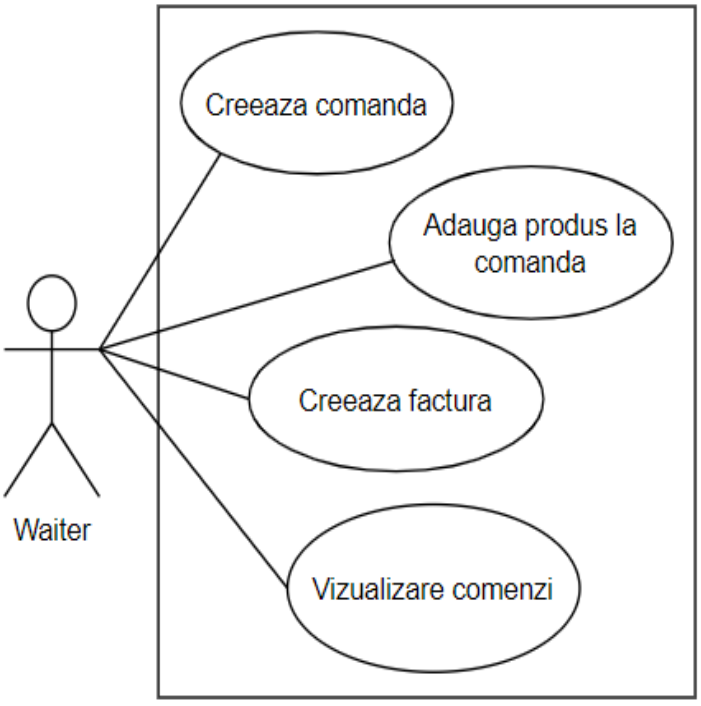
Elementele componente ale unei diagrame use case sunt:

- use case-uri ;

- actori ;

- relaţiile care se stabilesc între use case-uri, între actori şi între use case-uri şi actori .

Un use case (caz de utilizare) reprezintă cerinţe ale utilizatorilor. Este o descriere a unei mulţimi de secvenţe de acţiuni (incluzând variante) pe care un program le execută atunci când interacţionează cu entităţile din afara lui (actori) şi care conduc la obţinerea unui rezultat observabil şi de folos actorului .



Exemplu descriere use case :

Use case : Creeaza comanda

Actor: chelnerul

Scenariu:

1. Chelnerul introduce numarul mesei
2. Chelnerul introduce data
3. Chelnerul selecteza item-ul comandat din meniu
4. Chelnerul creeaza comanda

Alte variante:

- Chelnerul mai poate si sa: dauge produse la o comanda, sa creeze nota sis a vizualizeze comenzile

Deasemenea pentru aplicatia de restaurant management mai exista alti doi posibili actori : administratorul care poate adauga elemente la meni, edita elemente din meniu, stereg elemente din meniu si vizualiza intreg meniul.

Design patterns.

Observer Pattern

Design Pattern-ul *Observer* definește o relație de dependență 1 la n între obiecte astfel încât când un obiect își schimbă starea, toți dependenții lui sunt notificați și actualizați automat. Folosirea acestui pattern implică existența unui obiect cu rolul de *subiect*, care are asociată o listă de obiecte dependente, cu rolul de *observatori*, pe care le apelează automat de fiecare dată când se întâmplă o acțiune.

Acest pattern este de tip *Behavioral* (comportamental), deorece facilitează o organizare mai bună a comunicației dintre clase în funcție de rolurile/comportamentul acestora.

Observer se folosește în cazul în care mai multe clase(*observatori*) depind de comportamentul unei alte clase(*subiect*), în situații de tipul:

- o clasă implementează/reprezintă logica, componenta de bază, iar alte clase doar folosesc rezultate ale acesteia (monitorizare).
- o clasă efectuează acțiuni care apoi pot fi reprezentate în mai multe feluri de către alte clase

Composite Design Pattern

Composite Design Pattern descrie un grup de obiecte care sunt tratate la fel ca o singură instanță a aceluiași tip de obiect. Intenția unui compozit este de a "compune" obiecte în structuri de arbori pentru a reprezenta ierarhiile parțiale. Implementarea modelului compozit permite clienților să trateze uniform obiecte și compoziții individuale.

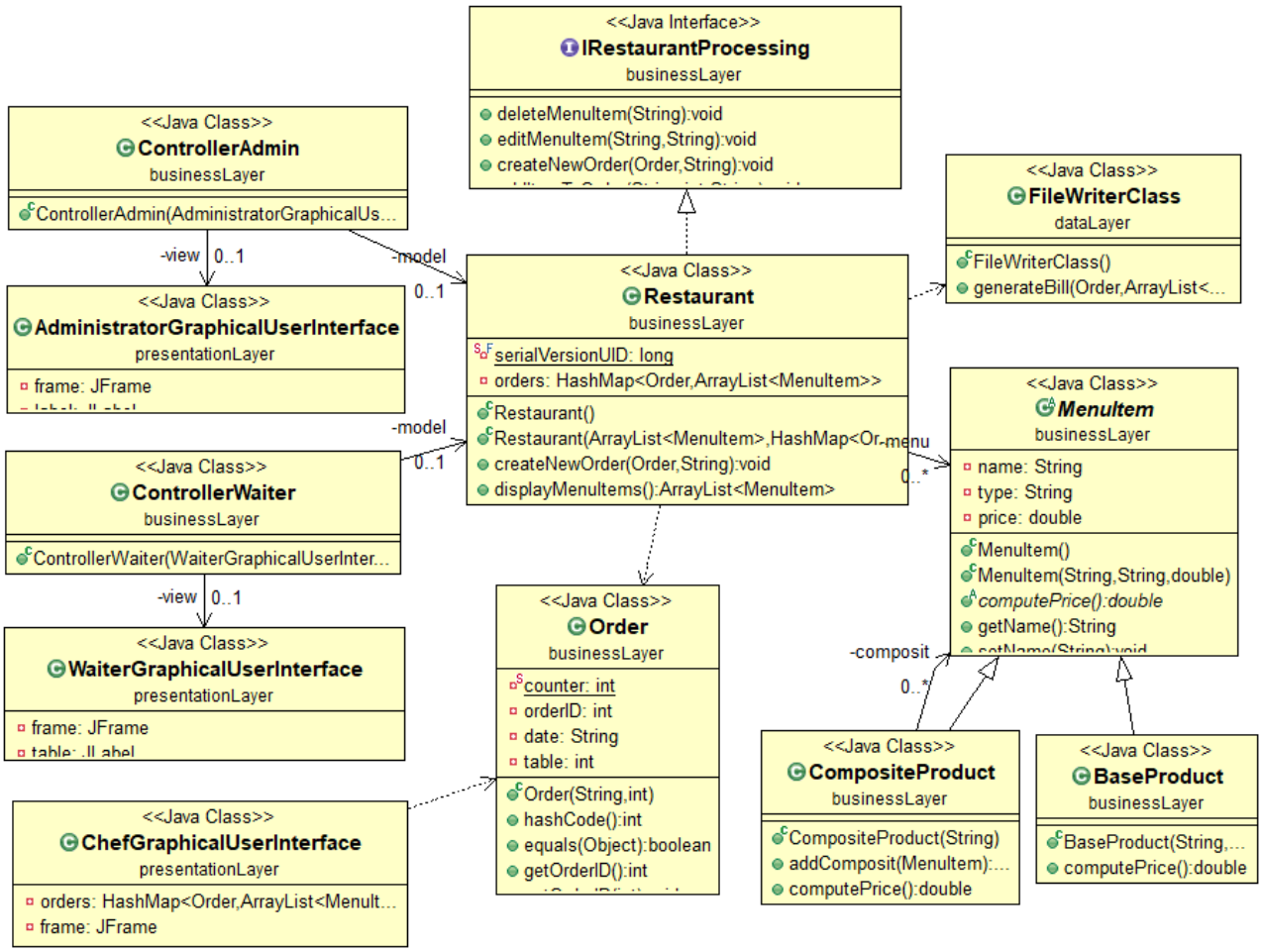
Design by Contract

Tehnica de dezvoltare a software-ului Design by Contract (DBC) asigură un software de înaltă calitate, garantând că fiecare componentă a unui sistem se ridică la nivelul așteptărilor sale. Ca dezvoltator care utilizează DBC, specificați contractele componente ca parte a interfeței componentei. Contractul specifică ce așteaptă acea componentă de la clienti și la ce se pot astepta clientii de a aceasta. Bertrand Meyer a dezvoltat DBC ca parte a limbajului său de programare Eiffel. Indiferent de originea sa, DBC este o tehnică de design valoroasă pentru toate limbajele de programare, inclusiv Java.

În timpul execuției, evaluăm afirmațiile la anumite puncte de control în timpul executării sistemului. Într-un sistem software valid, toate afirmațiile sunt evaluate ca adevărate. Cu alte cuvinte, dacă orice afirmație este evaluată la falsă, considerăm că sistemul software contine erori.

3.Proiectare

Diagrama UML

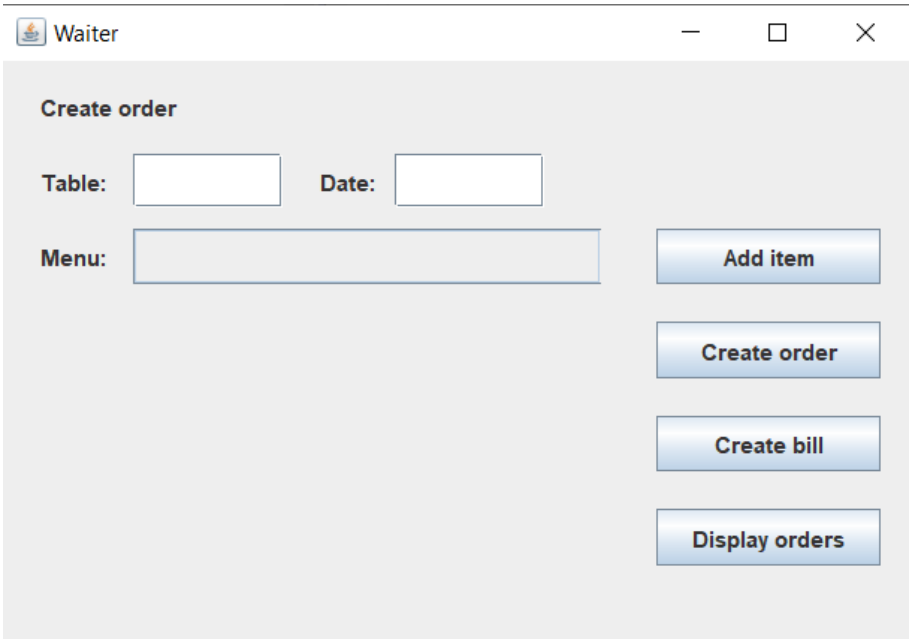


Structuri de date

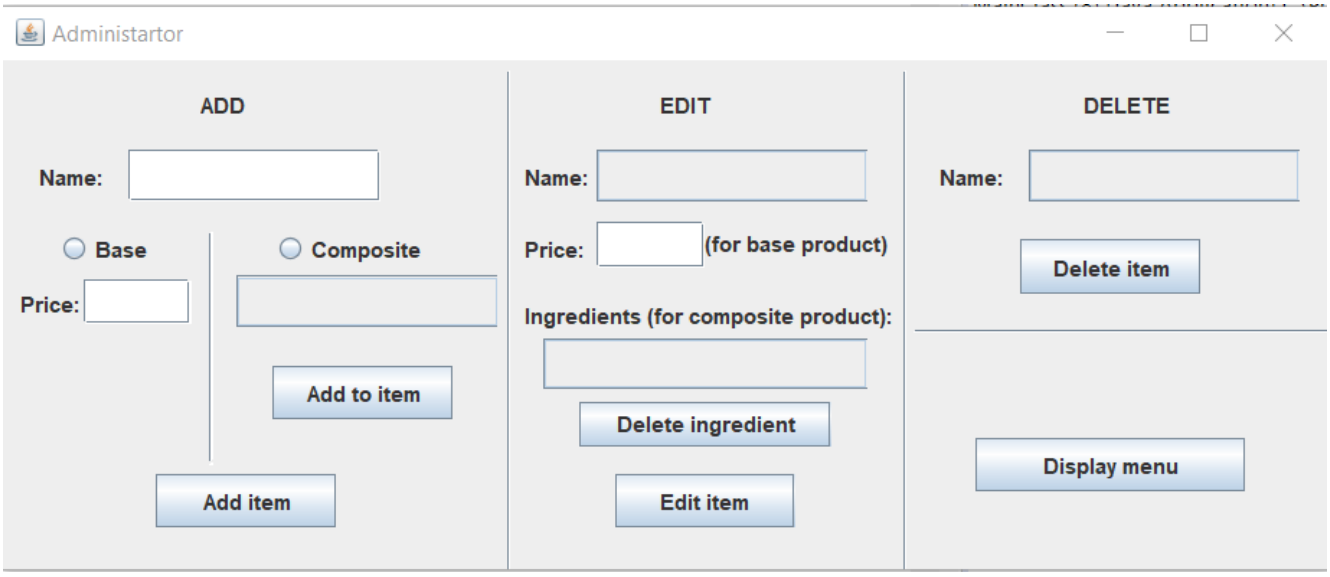
Structurile de date folosite in acest proiect sunt ArrayList -urile, care contin elemente de tipul MenuItem pentru a memora intreg continutul meniului. Deci, ArrayList-urile vor fi folosite pentru toate operatiile din cadrul programului, acestea memorand elementele care stau la baza aplicatie . Deasemenea, am folosit si HashMap, care va memora comenzile. Cheile vor fi de tipul Order(cu campurile: table, date si orderID), iar valoarea va fi de tipul ArrayList<Menu Item> pentru a putea retine toate produsele comandate la o singura masa.

Interfata utilizator

Prin intermediul interfetei se realizeaza comunicarea cu utilizatorul. Dintre pachetele care contin elemente de grafica am folosit javax.swing, care vine cu o gama extinsa de componenete si facilitati .

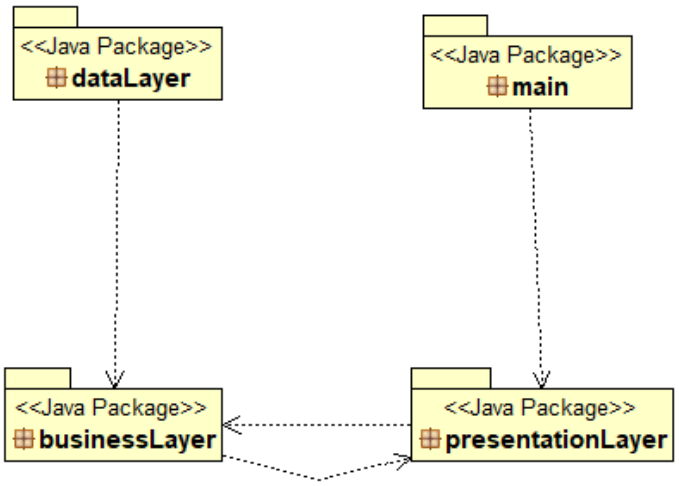


Interfata corespunzatoare chelnerului



Interfata corespunzatoare administratorului care poate adauga iteme in meniu, poate modifica iteme din meniu si poate sterge iteme din meniu.

Packages



Pachetul (package) este o grupare de elemente ale unui model şi reprezintă baza necesară controlului configuraţiei, depozitării şi accesului . Este un container logic pentru elemente între care se stabilesc legături . Toate elementele UML pot fi grupate în pachete (cel mai des pachetele sunt folosite pentru a grupa clase). Un element poate fi conţinut într-un singur pachet

4.Implementare

Programul este organizat conform șablonului “Layered architecture” in care avem urmatoarele pachete : dataLayer, businessLayer, presentationLayer si main.

- **Pachetul dataLayer** contine clasele FileWriterClass si Restaurant Serializator

Clasa FileWriterClass

- este clasa in care se genereaza nota pentru comanda in format .txt

- contine o singura metoda, si anume generateBill(Order order, ArrayList<MenuItem> menu) ;

- metoda primeste ca parametrii date despre comanda si anume numarul mesei, id-ul comenzii, data si produsele comandate pentru masa respective;

Clasa RestaurantSerializator

- implementeaza clasa Serializable

- java permite un mecanism mai avansat si anume serializarea obiectelor. In forma cea mai simpla serializarea obiectelor inseamna salvarea si restaurarea starii obiectelor. Obiectele oricarei clase care implementeaza interfata *Serializable*, pot fi salvate intr-un stream(fluxde date) si restaurate din acesta.

- pachetul java.io contine doua clase speciale *ObjectInputStream* respectiv *ObjectOutputStream* pentru serializarea tipurilor primitive; acestea sunt metodele impementate de clasa RestaurantSerializator

- **Pachetul businessLogicLayer** contine clasele: BaseProduct, CompositeProduct

ControllerWaiter, ControllerAdmin, MenuItem, Order, Restaurant

Clasa BaseProduct

- extinde clasa abstracta MenuItem si este clasa care implementeaza unul din cele doua tipuri de produse din meniu si anume produse de baza care pot fi folosite in compozitia altui produs

- clasa contine un constructor si implementeaza metoda computePrice din clasa MenuItem

```
public class BaseProduct extends MenuItem{
	public BaseProduct(String nume, double price) {
		super(nume,"base",price);
	}
	public double computePrice() {
		return this.getPrice();
	}
}
```

Clasa CompositeProduct

- extinde clasa abstracta MenuItem si este clasa care implementeaza unul din cele doua tipuri de produse din meniu si anume produse compuse, adica sunt produse care au in compozitia lor alte produse din meniu, fie produse de baza, fie produse compuse;

- clasa contine un constructor si implementeaza metoda computePrice din clasa MenuItem

- de asemenea are si un atribut , un ArrayList de MenuItem care reprezinta ingredientele produsului

Clasa MenuItem

- este clasa cu ajutorul careia vom stoca meniul sub forma unui arrayList de MenuItem

- este o clasa abstract, care contine un constructor si metoda abstracta computePrice

- este mostenita de clasele BaseProduct si CompositeProduct

Clasa Order

- clasa care va memora data la care s-a facut comada, masa si va genera un ID unic pentru fiecare comnda

- este clasa cu ajutorul careia vom memora comenzile;

- va reprezenta key-ul dintr-un hashMap in care vom stoca comenzile, de aceea clasa suprascrie metodele hashCode() si equals()

Clasa Restaurant

- este clasa principala a aplicatiei; extinde clasa Observable si implementeaza interfetele Serializable si IRestaurantProcesing

- avand in vedere ca in aplicatie avem interfete grafice, acestea au fost realizate cu ajutorul sablonului MVC(model- view-controller), asadar clasa restaurant reprezinta clasa model pentru acest sablon

- este clasa care implementeaza toate operatiile care pot fi efectuate de administrator si chelner, si totodata notifica chef-ul cand trebuie sa prepare o comanda noua;

- principalele functii implementate de clasa Restaurant

```
public void deleteMenuItem(String item);

public void editMenuItem(String nume, String pret);

public void createNewOrder(Order order, String item);

public void addItemToOrder(String data, int table, String item);

public void generateBill(String table, String date);

public void createNewBaseMenuItem(String nume, String pret);

public void createNewCompositeMenuItem(String nume,String item);
```

Clasa ControllerWaiter

- este clasa care implementeaza logica corespunzatoare controller-ului din sablonului MVC(model-view-controller) pentru utilizator chelner si interfata grafica WaiterGrphicalUserInterface

- este clasa care face legatura dintre view(WaiterGrphicalUserInterface) si model(Restaurant)

```
public class ControllerWaiter {
        private WaiterGraphicalUserInterface view;
        private Restaurant model;
        public ControllerWaiter(WaiterGraphicalUserInterface view, Restaurant model) {
                this.view = view;
                this.model = model;
                this.view.addCreateOrderButtonActionListener(new CreateOrderListener());
                this.view.addItemButtonActionListener(new AddItemListener());
                this.view.addCreateBillButtonActionListener(new CreateBillListener());
                this.view.addDisplayButtonActionListener(new DisplayListener());
                view.selectItem.removeAllItems();
                ArrayList<MenuItem> meniu = model.displayMenuItems();
                for (MenuItem i : meniu) {
                        view.selectItem.addItem(i.getName());
                }
        } . . .
```

Clasa ControllerAdmin

- este clasa care implementeaza logica corespunzatoare controller-ului din sablonului MVC(model-view-controller) pentru administrator si interfata grafica AdministratorGrphicalUserInterface

- este clasa care face legatura dintre view(AdministratorGrphicalUserInterface) si model(Restaurant)

- **Pachetul presentationLayer** contine clasele: ChefGraphicalUserInterface, AdministratotGraphicalUserInterface, WaiterGraphicalUserInterface

Clasa WaiterGraphicalUserInterface

- este clasa care creeaza interfata grafica pentru utilizatorul chelner , cu ajutorul interfetei grafice chelnerul poate creea o comanda, poate adauga produse la o comanda, poate crea o nota de plata si poate vizualiza toate comenzile;

Clasa AdministratotGraphicalUserInterface

- este clasa care creeaza interfata grafica pentru utilizatorul administrator, cu ajutorul interfetei grafice acesta poate adauga produse la comanda sau poate modifica lista de ingredient pentru pun produs; poate modifica pretul unui produs sau poate sterge ingredient din produs; administratorul paote sterge un produs si poate vizualiza intreg meniul;

Clasa ChefGraphicalUserInterface

- este clasa care creeaza interfata grafica pentru utilizatorul : chef, acesta este notificat de clasa Restaurant ori de cate ori apare o comanda noua

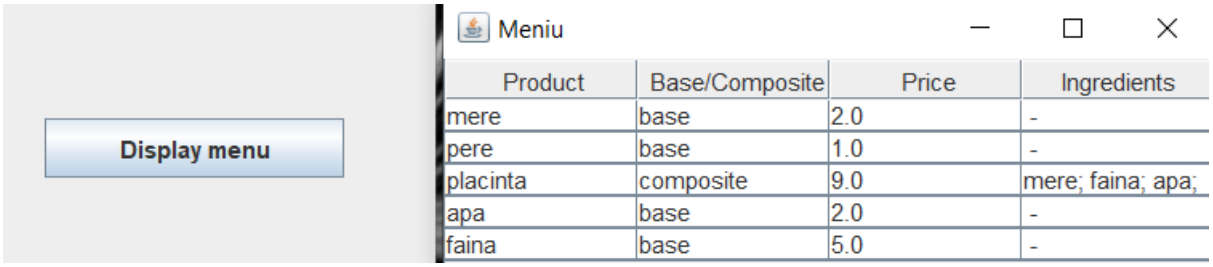
- este implementata cu ajutorul observer pattern design si implementeaza interfata Observer;

```
public class ChefGraphicalUserInterface implements Observer {
        private JFrame frame;
        private JLabel text;
        public ChefGraphicalUserInterface() {
                frame = new JFrame("Chef");
                frame.setSize(400, 250);
                frame.setVisible(true);
                frame.setLayout(null);
                text = new JLabel();
                text.setBounds(10, 10, 700, 90);
                frame.add(text);
                frame.repaint();
        }
        public void update(Observable o, Object arg) {
                System.out.println("Update");
                String message = "S-a realizat o comanda pentru preparatul: " + ((MenuItem) arg).getName();
                text.setText(message);
        }
}
```

5 Rezultate

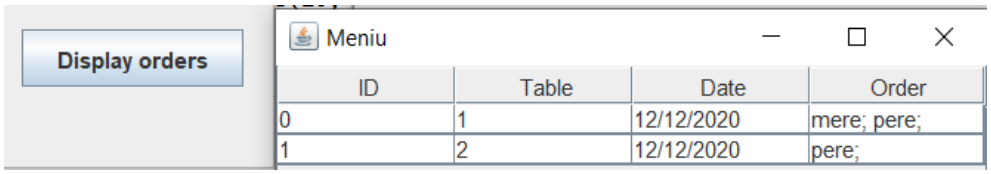
Exemple de rezultate care pot fi obtinute:

Daca administratorul doreste sa vizualizeze intreg meniul, acesta poate apasa pe butonul ‘DisplayMenu’



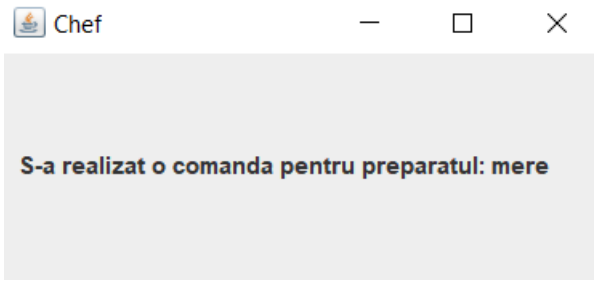
Product	Base/Composite	Price	Ingredients
mere	base	2.0	-
pere	base	1.0	-
placinta	composite	9.0	mere; faina; apa;
apa	base	2.0	-
faina	base	5.0	-

De asemenea, chelnerul poate viualiza toate comenzile din momentul curent:



ID	Table	Date	Order
0	1	12/12/2020	mere; pere;
1	2	12/12/2020	pere;

Cheful este notificat de fiecare data cand apare o comanda noua :



Chelnerul creaza facturi pentru comenzi:

```
                BILL
Data:12/12/2020
Masa:1
Produse comandate:
-> mere. . . . . 2.0
-> pere. . . . . 1.0
Suma de platit: 3.0
```

6 Concluzii

In concluzie, datorita acestei teme am reusit sa ma familiarizez cu proiectarea unei aplicatii care foloseste mai multe interfete grafice, dar si cu tipul de structurare al unei aplicatii, si anume Layered Architecture.

Deasemenea, am invatat metoda noi de design care pot fi introduse in aplicatii, si anume: Design by Contract method, Composite Design Pattern, Observer Design Pattern.

7.Bibliografie

- https://www.baeldung.com/java-serialization

- http://users.utcluj.ro/~igiosan/teaching_poo.html

- https://www.javatpoint.com/

- https://stackoverflow.com/

- https://www.w3schools.com/java/

- https://objectcomputing.com/resources/publications/sett/september-2011-design-by-contract-in-java-with-google

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