**Documentatie-Tema 3**

**Restaurant management system**

***Calacean Ionut Eugen***

***Facultatea de automatica si calculatoare***

***Grupa 30229***

***Indrumator laborator:Pop Cristina***

**Cuprins**

1. **Obiectivul temei.**
2. **Analiza problemei, modelare, scenarii, cazuri de utilizare.**
3. **Proiectare(decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator).**
4. **Implementare.**
5. **Rezultate.**
6. **Concluzii.**
7. **Bibliografie.**

**1.Obiectivul temei**

Un sistem de management electronic pentru un restaurant ar trebui sa fie capabil sa stocheze informatii referitoate la comenzile restaurantului, modificari aduse in cadrul meniului pentru restaurant si sa aiba o implementare cat mai eficienta astfel incat timpul de asteptare pentru client si calitatea mancarilor si bauturilor sa fie maxima.

In acest scop, aplicatia ar trebui sa fie structurata in 3 mari parti, o parte destinata uzului de catre administrator, o parte destinata chelnerilor care sa nu aiba acces la ceea ce au administratorii si o parte simpla, fara interactiune, care sa fie folosita de bucatari pentru a observa comenzile in curs de desfasurare. Asadar un administrator ar trebui sa poate introduce produse de baza in restaurant, cat si sa creeze un produs compus(cum ar fi de exemplu un meniul al zilei), sa modifice pretul sau denumirea unor produse existente sau sa stearga unele produse. Iar unui chelner ar trebui sa i se permita sa introduca o noua comanda, sa calculeze pretul total al comenzii, sa genereze o factura sau sa vizualizeze comanda unui client. In timp real, in momentul efectuarii unei comenzi, bucatarul ar trebui sa fie notificat cu privire la produsele comandate, pentru a incepe deja pregatirea acestora.

**Cerinta principala** a temei este:” Considerati implementarea unui sistem de management pentru un restaurant. Sistemul ar trebui sa aiba 3 tipuri de utilizatori: administrator, chelner si bucatar. Administratorul poate sa adauge, sa stearga sau sa modifice produsele existente din meniu, chelnerul poate sa creeze o noua comanda pentru o masa, sa adauge elemente din meniu, si sa creeze o factura pentru o comanda. Bucatarul este notificat de fiecare data cand trebuie sa gateasca de catre chelner.

Pentru siplificarea aplicatiei, puteti sa considerati ca sistemul este folosit de un singur administrator, un chelner si un bucatar, si nu este nevoie de un proces de logare.

Rezolvati urmatoarele:

1. Definiti interfata **RestaurantProcessing** continand principalele operatii care pot fi executate de catre chelner sau administrator dupa cum urmeaza:
   * Administrator: creeaza un produs din meniu nou, sterge un produs sau il modifica;
   * Chelner: creeaza o comanda noua, genereaza pretul unei comenzi sau genereaza o factura;
2. Definiti si implementati clasele din diagrama de clase propusa :

* Folositi Composite Design Pattern pentru a defini clasa **MenuItem**, **BaseProduct** si **CompositeProduct**;
* Folositi Observer Design Pattern pentru a notifica bucatarul de fiecare data cand o noua comanda care contine un produs compus este adaugata;

1. Implementati clasa **Restaurant** folosind o colectie predefinita JCF care foloseste o structura de date de tip hashtable. Cheie hastable va fi generata bazandu-ne pe clasa **Order**, care poate sa aiba asociate cateva produse din meniu. Folositi JTable pentru a afisa informatiile relevante ale restaurantului.

* Definiti o structura de tipul **Map<Order, Collection<MenuItem>>** pentru a stoca informatiile relevante ale unei comenzi, pentru care metoca hashCode() va fi suprascrisa pentru a genera valoarea de hash necesara in Map pentru atributele din clasa Order(cum ar fi OrderID, date,etc);
* Definiti o structura de tipul **Collection<MenuItem>** care va salva meniul din restaurant. Alegeti tipul potrivit de colectie pentru implementare voastra;
* Definiti o metoda de tipul “wellFormed” pentru clasa Restaurant;
* Implementati clasa utilizand metoda Design by Contract(care implica conditii pre, post, invarianti si asserturi).

1. Produsele din meniu pentru a popula Restaurantul vor fi incarcate/salvate din/spre un fisier folosind Serializarea.

**Obiective**

* Tehnici de programare Design by Contract
* Polimorfism
* Design Patterns: Observer, Composite
* Implenmentari JCF HashMap si HashSet
* Serializare

**Obiective secundare**

* Creare factura in format text
* Notificare bucatarie in timp real
* Afisare meniu, produse comandate

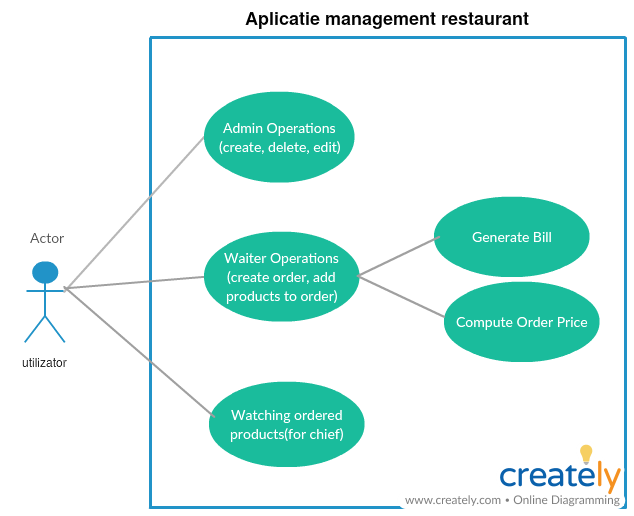
**2.Analiza problemei, modelare, scenarii, cazuri de utilizare**

Analiza preliminare a problemei ne imparte sarcina in crearea in primul rand a modelelor de baza pentru aplicatie si anume clasele care vor corespunde meniului restaurantului. Asupra acestuia si din acesta actioneaza toate cele 3 fire ale aplicatiei: administrator, bucatar si chelner. Aceste clase vor folosi Composite Design Pattern, acest fapt face ca utilizatorul sa poata trata structurile complexe( de tipul unui produs compus din mai multe produse de baza) ca si un obiect individual, si lui( utilizatorului) nu ar trebui sa ii pese astfel(sau sa stie) daca are de lucru cu un produs compus sau cu o “frunza”(produs de baza). Acest fapt face astfel mai usor sa se insereze noi componente, si sa se lucreze cu acestea.

Tot din analiza problemei ne rezulta si faptul ca avem nevoie de o mapare sub forma de hashtable, pentru a retine comenzile restaurantului, de tehnica Observer Design Pattern, la care adaugam mai multi observatori( in cazul in care am avea mai multi bucatari) la acelasi obiect observat.

Prin modelarea problemei ne rezulta faptul ca o sa avem nevoie de colectii, pentru a stoca intai elementele constituente ale unui produs compus, si mai apoi pentru a stoca produsele comandate de catre un client (acestea putand fi produse de baza sau produse compuse)- in acest caz folosim **ArrayList.** La nivelul retinerii comenzilor, pentru a stoca comenzile in locatii diferite de memorie vom folosi **HashMap** iar cheia de mapare va fi generata de clasa Order, care va contine pe langa un id unic, si o data unica, astfel se evita pe cat posibil coliziunile din tabela de mapare.

In continuare voi prezenta scenarii si cazuri de utilizare sub forma de diagrame use-case:



* **Use case:Admin operations**
* **Primary actor:utilizatorul(administrator)**
* **Main succes scenario:**
  + Utiliatorul porneste view-ul pentru oparatiile predestinate administratorui
  + Utilizatorul introduce detaliile unui produs de baza si selecteaza butonul de inserare
  + Utilizatorul selecteaza din tabel un produs si introduce detaliile pentru modificare(pret sau nume)
  + Utilizatorul selecteaza un produs si in sterge
  + Utilizatorul selecteaza mai multe produse ( de baza sau compuse din lista) si selecteaza butonul de creare a unui produs compus dupa care introduce numele produsului nou
  + Utilizatorul pria datele din tabelul curent care contine tot meniul
  + Utilizatorul continua cu lucrul pe meniu sau paraseste fereastra prin selectarea butonului inapoi
* **Use case: Operatie de insert, edit sau delete**
* **Primary actor:utilizatorul**
* **Alternative Sequences:**

1. Datele care se doresc a fi inserate nu sunt valide:
   1. Pretul produsului este negativ sau contine litere

**3.Proiectare**

Clase Folosite:

1. Clasa **App-** clasa principala, nu mai este pusa pe diagrama este independenta contine doar metoda main, in ea se construieste meniul initial, se serializeaza si de sesereializeaza, tot aici se construieste View-ul principal, si se leaga la Controller.
2. Clasa **BaseProduct**- aceasta clasa contine informatiile unui produs de baza(nume , pret) si o metoda de returnare a pretului unui produs, utila in calculul pretului total al unui produs compus
3. Clasa **CompositeProduct**- clasa utila in crearea meniului aceasta contine o colectie de obiecte de tipul clasei parinte si o metoda de calculare a pretului printr-o iteratie prin aceasta lista
4. Clasa **MenuItem**- parinte pentru clasele 2 si 3, aceasta ajuta la stocarea unui produs de tip atomic( indiferent ca este produs de baza sau produs compus, se trateaza la fel) se utilizeaza aici si poliporfismul.
5. Clasa **Order**- reprezinta clasa care retine informatiile de baza pentru o comanda(id, masa, data) si care este folosita pentru a mapa comanda in hastable, este clasa care genereaza cheia necesara maparii.
6. Clasa **Restaurant**- contine principalele operatii ( asociate chelnerului si administratorului), contine tehnica Design by Contract dar si 2 metode wellFormed pentru variabilele detinute(Colectia si Map-ul)
7. Interfata **RestaurantProcessing**-contine cele 6 operatii necesare dar si contracte pentru metode( pre, post, invarianti) care vor trebui respectate. Tot aici se retine si ArrayListul de MenuItems(meniul propriu zis al restaurantului) si Map-ul care retine comenzile din restaurant.
8. Clasele **AdministratorGraphicalUserInterface, WaiterGraphicalUserInterface, ChiefGraphicalUserInterface, View,**  sunt clasele care implemeneaza interfetele utilizator( coninand JTable-uri, butoane, informatii)
9. Clasa **Controller**- aceasta clasa controleaza functionalitatea de baza a aplicatiei, inserarea comenzilor,a produselor, si dinamica graficii afisate spre utilizator

Ca alte tipuri de date folosite, aici intra ArrayList-ul si HashMap-ul. ArrayList-ul este folosit in principal in 2 parti ale aplicatiei: Composite Design Pattern si ca si camp de valori in tabelul de hashmap, fiecare comanda avand o lista de produse din meniu asociata pe baza unei chei generate de comanda.

In continuare voi prezenta diagrama de clase:

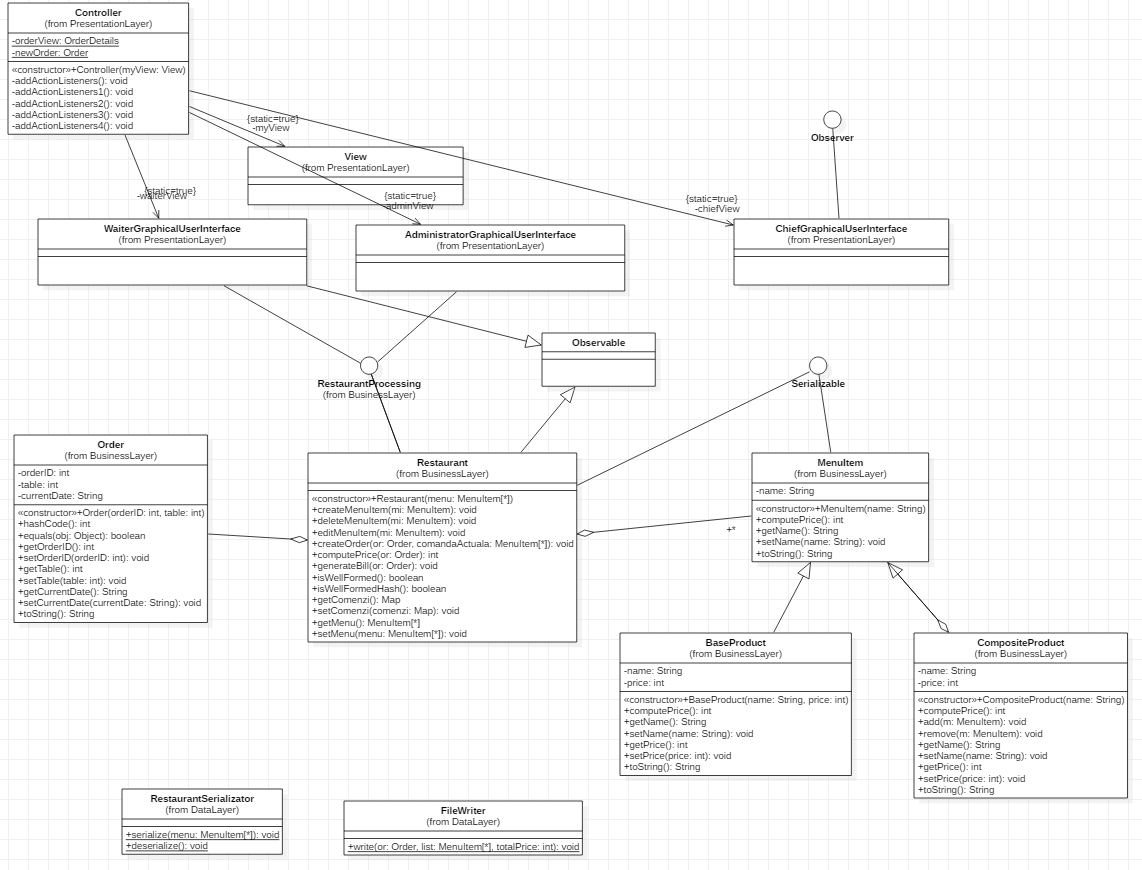
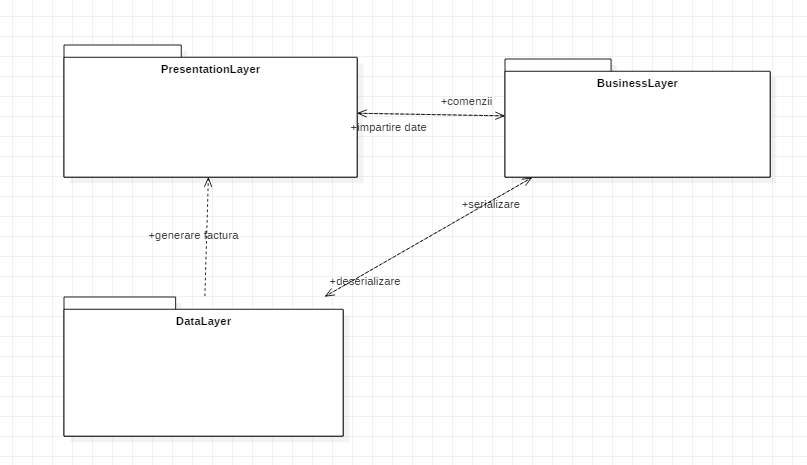


Diagrama de pachete:



Aplicatia este organizata pe layere. Astfel se identifica DataLayer, in acest pachet se face serializare si deserializare, tot aici se genereaza o factura pentru un client al restaurantului. Un alt pachet al aplicatiei BusinessLayer contine interfata RestaurantProcessing implementata de catre clasa restaurant care contine metodele asociate administratorului si chelnerului(de inserare, stergere modificare produs, inserare comanda, generare factura si calculare pret), tot aici se afla si cele 2 clase MenuItem si Order care se folosesc in Map-ul din restaurant pentru a stoca comenzile. In pachetul PresentationLayer sunt stocate clasele care implementeaza interfetele utilizator ale aplicatiei, cat si controllerul care leaga acele clase si creeaza o dinamica anume a functionalitatii interfetei.

**4.Implementare**

Voi prezenta in continuare o parte a implementarii care contine anumite principii:

**Composite Design Pattern**

**Clasa MenuItem:-** contine o metoda de calculare a pretului fara implementare suprascrisa in cei doi copii, asfel **clasa BaseProduct** va contine 2 variabile(nume , pret) si o metoda prin care se returneaza pretul, iar in **clasa** **CompositeProduct** se stocheaza informatiile care fac posibil acest concept **Composite Design Pattern**:

public class CompositeProduct extends MenuItem {

private String name;

private int price;

private ArrayList<MenuItem> dish;

public CompositeProduct(String name)

{

super(name);

this.name=name;

dish=new ArrayList<MenuItem>();

}

public int computePrice()

{

int price=0;

for(MenuItem c:dish)

{

price+=c.computePrice();

}

return price;

}

public void add(MenuItem m)

{

dish.add(m);

this.price=computePrice();

}

public void remove(MenuItem m)

{

dish.remove(m);

}

Astfel aceasta clasa contine o lista de clase MenuItem, clasa parinte astfel avem construita o ierarhie din obiecte primitive(produse de baza) si produse compuse(care pot fi alcatuite din mai multe produse de baza sau din alte produse comouse). Orice obiect primitiv poate fi compus in obiecte mai complexe care se pot schimba la randul lor in obiecte mai complexe si tot asa, in mod recursiv. De fiecare data cand codul nostru asteapta un obiect MenuItem, acesta poate fi atat un produs de baza cat si unul compus. Iar metoda cu importanta maxima(computePrice) parcurge lista noastra si apeleaza metoda de calcuare a pretului pentru fiecare tip de produs.

**Observer Design Pattern**

Acest concept defineste o dependenta de tip 1:n intre obiectele de tipul **Restaurant** care vor extinde clasa deja existenta in Java Observable care pune la dispozitie metodele **setChanged()** pentru a modifica starea actuala a obiectului si **notifyObservers()** care transmite un mesaj observatorilor asociati prin metoda **addObserver().** Astfel de fiecare data cand starea obiectului se schimba observatorii asociati vor fi notificati si updatati. La noi exista un observator unic, acesta fiind **ChiefGraphicalUserInterface**.

Voi prezenta codul sectionat de prin clasele amintite pentru a sublinia conceptul:

-clasa care adauga observatorii:

if(*chiefView*==null)

*chiefView*=new ChiefGraphicalUserInterface();

*waiterView*.addObserver(*chiefView*);

-clasa care extinde Observable:

***comenzi***.put(or, comandaActuala);

updateJt();

setChanged();

notifyObservers(comandaActuala);

-clasa care implementeaza Observer:

public void update(Observable arg0, Object arg1) {

ArrayList<MenuItem> list=(ArrayList<MenuItem>)arg1;

boolean containsComposed=false;

for(MenuItem mi:list)

{

ItemsChiefs newItem=new ItemsChiefs(mi.getName());

if(mi instanceof CompositeProduct)

containsComposed=true;

produse.add(newItem);

}

if(containsComposed) {

myModel = ReflectionTable.*createTable*(produse);

updateJt();

}

else

produse.clear();

}

Metoda update din interfata bucatarului are rolul de a afisa pe ecranul acestuia de fiecare data cand o comanda care contine un produs compus(pentru care ar trebui sa munceasca el, pregatindu-l) este efectuata de catre un client. Notificare este facuta cand chelnerul ia practic comanda.

**Implementarea HashMap**

Clasa **Restaurant** contine un HashMap in care pe baza unei chei generate de un obiect de tipul **Order** se mapeaza ca valoare intr-un hashtable o lista de MenuItem care contine produsele comandate de catre o persoana. Pentru aceasta insa trebuie suprascrise metodele de hashMap() si equals()(pentru verificare de egalitate in cazul in care avem coliziune):

public static Map<Order, ArrayList<MenuItem>> ***comenzi***=new HashMap<Order,ArrayList<MenuItem>>();

*@Override*

public int hashCode() {

int hash=3;

hash=29\*hash+ orderID;

hash=29\*hash+ table;

return hash;

}

*@Override*

public boolean equals(Object obj) {

if(this==obj) return true;

if(obj==null) return false;

if(this.getClass()!=obj.getClass())return false;

Order order=(Order)obj;

return orderID==order.orderID && table==order.table && currentDate.equals(order.currentDate);

}

**Implementarea serializarii**

Prin acest concept, detaliile unei anumite clase, la noi clasa **Restaurant** sunt salvate intr-un fisier, iar mai apoi sunt preluate din acest fisier si incarcate in aplicatie. Voi prezenta in continuare codul pentru acest lucru:

public static void serialize(ArrayList<MenuItem> menu) {

Restaurant rest=new Restaurant(menu);

String filename="restaurantdata.txt";

try {

FileOutputStream file=new FileOutputStream(filename);

ObjectOutputStream out=new ObjectOutputStream(file);

out.writeObject(rest);

out.close();

file.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

public static void deserialize()

{

String filename="restaurantdata.txt";

Restaurant rest=null;

try {

FileInputStream file=new FileInputStream(filename);

ObjectInputStream in=new ObjectInputStream(file);

rest=(Restaurant)in.readObject();

System.***out***.println(rest.*restmenu*);

RestaurantProcessing.***menu***.clear();

RestaurantProcessing.***menu***.addAll(rest.*restmenu*);

in.close();

file.close();

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

**Conceptul Design by Contract**

Programatorul trebuie sa specifice pentru fiecare metoda(pe langa codul functional) niste pre-conditii, post-conditii sau niste invarianti. Acesti invarianti sunt de fapt niste metode create de programator care verifica starea obiectului: eu am implementat metoda wellFormed pentru clasa Restaurant. Voi prezenta in cele ce urmeaza ceea ce tine de conceptul de Design by Contract pentru un exemplu din codul meu:

/\*\*

\* Generate a bill for client

\* **@pre** or!=null

\* **@pre** list.get(or)!=null

\* **@post** **@nochange**

\* **@invariant** isWellFormedHash()

\*/

public void generateBill(Order or);

public void generateBill(Order or) {

assert or != null && ***comenzi***.get(or) != null;

Map<Order, ArrayList<MenuItem>> preMap = new HashMap<Order, ArrayList<MenuItem>>();

preMap.putAll(***comenzi***);

assert isWellFormedHash();

int totalPrice = computePrice(or);

FileWriter.*write*(or, ***comenzi***.get(or), totalPrice);

assert preMap.equals(***comenzi***);

assert isWellFormedHash();

public boolean isWellFormedHash() {

Iterator it = ***comenzi***.entrySet().iterator();

int n = 0;

while (it.hasNext()) {

Map.Entry<Order, ArrayList<MenuItem>> pair = (Map.Entry<Order, ArrayList<MenuItem>>) it.next();

if (pair.getValue() == ***comenzi***.get(pair.getKey()))

n++;

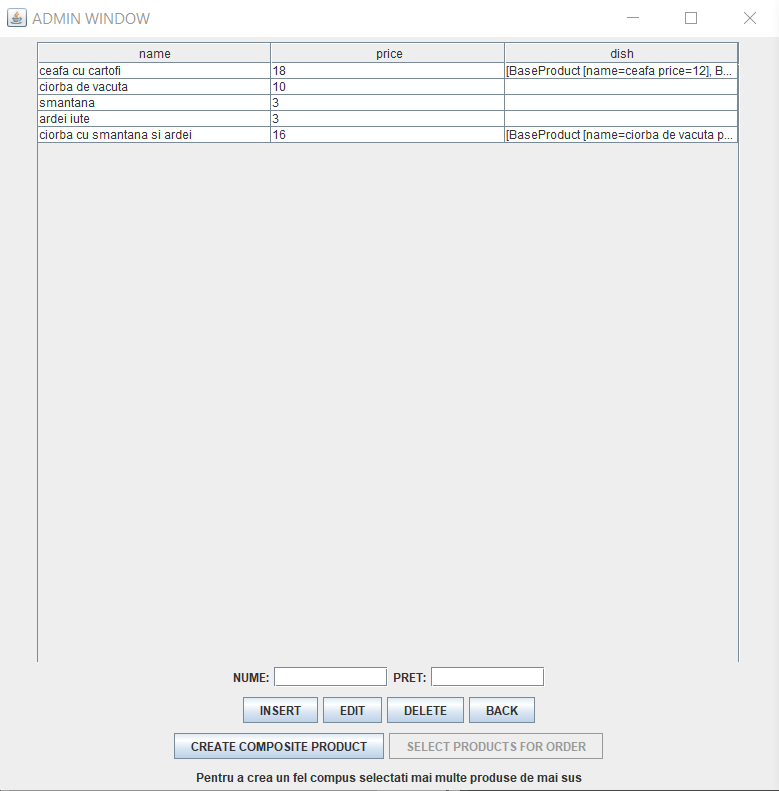
}

return n == ***comenzi***.size();

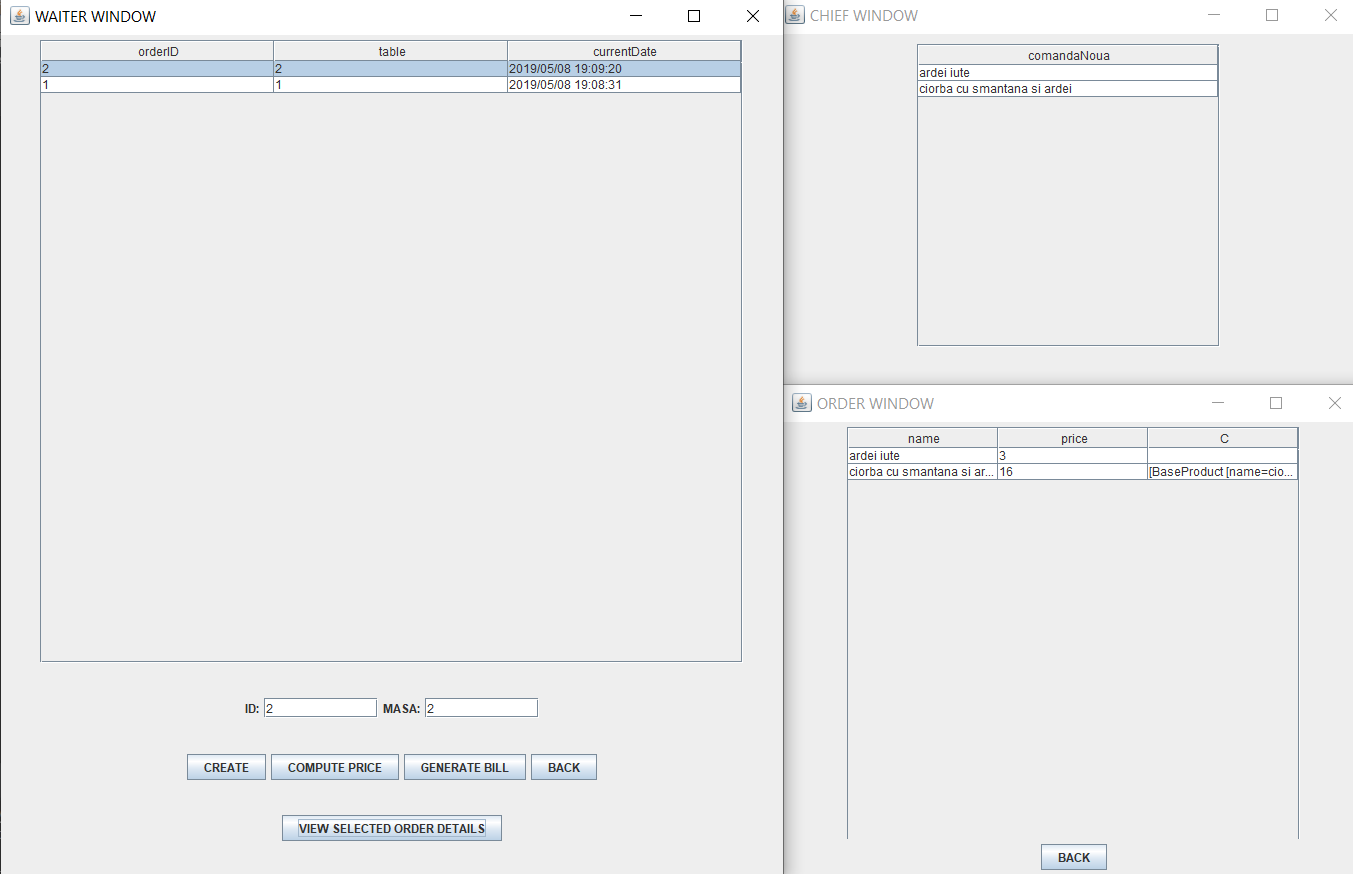
} }

**Clasa AdministratorGraphicalUserInterface**

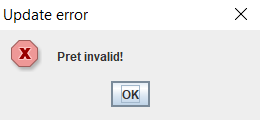
Aceasta clasa creeaza interfata pentru administrator care dupa modificarile efectuate( in principal m-am folosit de BoxLayout pentru aranjare), un JScrollPane adaugat la tabel(fapt care face automat sa se afiseze si header-ul tabelului meu) va arata in felul urmator:



Asemanator(tot avand la baza un JTable) vor arata si celelalte 2 interfete(asociate chelnerului si bucatarului) cu mentiunea ca bucatarul nu are acces la butoane:



Aplicatia detecteaza introducerea eronata de catre administrator a pretului( in cazul in care acesta este negativ sau contine altceva decat cifre) caz in care eroarea ne este semnalata prin urmatoarea grafica:



**5.Rezultate**

Pe baza acestei aplicatii se poate manageria un restaurant, prin controlul total al produselor existente in stoc in acea zi, prin controlul asupra comenzilor efectuate rezulta o eficienta crescuta in gestionarea stocului de produse sau in castigarea de timp pretios pentru clientii restaurantului, deoarece bucatarul este notificat in timp real de fiecare data cand chelnerul ia o comanda care contine un produs care trebuie preparat.

Aplicatia mentine astfel si un istoric al comenzilor, se poate calcula profitul/pierderea, identifica produsele care trec sau perioadele in care sunt necesare suplimentari.

Aplicatia poate fi conectata la facturier, deoarece produsele comandate de catre un client pot genera o factura, la cererea chelnerului, insa in aplicatie aceasta este in format .txt.

**6.Concluzii**

Ca o concluzie la acest proiect se observa importanta incontestabila a unor principii de programare cum ar fi Composite Design Pattern sau Observer Design Pattern, cat si importanta exitentei in Java a unor colectii de tipul HashMap. Toate acestea ofera o maleabilitatea remarcabila in gestionarea obiectelor necesare in programare, iar metodele deja implementate de catre background-ul Java isi afirma importanta.

Unul dintre aspectele importante care m-a ajutat la dezvoltarea cunostintelor de programare este folosirea Serializarii: astfel se poate salva starea unui obiect la un moment dat, aceasta fiind folosita ulterior la redeschiderea aplicatiei( toate acestea fiind o alternativa la folosirea bazelor de date). Un alt aspect important este principul Observer Design Pattern care poate fi folositor in multe cazuri cand avem o dependenta de tip 1:n.

Din punct de vedere al dezvoltarii ulterioare, aplicatia ar putea implementa si o metoda prin care bucatarul sa notifice chelnerul cu privire la finalizarea comenzii(daca nu ar exista mijloc de comunicare directa intre bucatarie si chelneri). Tot ca dezvoltare ulterioara s-ar putea implementa o metoda de notificare tot din partea bucatarilor , dar de aceasta data care sa vizeze administratorul cu privire la stocul redus al unor produse necesare in bucatarie.

**7.Bibliografie**

[1] <http://users.utcluj.ro/~crisb_pop/pt2018.html>

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

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