**Tema 4**

**Restaurant**

Catelina Ioan

Universitatea Tehnica din Cluj-Napoca

Facultatea de automatica si calculatoare, an 2

Grupa 30227

Contents

[1. Obiectivul temei 2](#_Toc8912309)

[1.1 Obiectivul principal 2](#_Toc8912310)

[1.2 Obiective secundare 2](#_Toc8912311)

[2. Analiza problemei 2](#_Toc8912312)

[3. Proiectare 3](#_Toc8912313)

[4. Implementare 4](#_Toc8912314)

[4.1 Restaurant 4](#_Toc8912315)

[4.2 MenuItem 8](#_Toc8912316)

[4.3 Order 8](#_Toc8912317)

[4.4 FileWriter 8](#_Toc8912318)

[4.5 RestaurantSerializer 9](#_Toc8912319)

[4.6 AdministratorGUI 9](#_Toc8912320)

[4.7 WaiterGUI 9](#_Toc8912321)

[4.8 ChefGUI 10](#_Toc8912322)

[4.9 BigGUI 10](#_Toc8912323)

[4.10 Controller 10](#_Toc8912324)

[5. Concluzii 10](#_Toc8912325)

[6. Bibliografie 10](#_Toc8912326)

# Obiectivul temei

## Obiectivul principal

Obiectivul fundamental al acestei lucrări de laborator este proiectarea și implementarea unui sistem de gestionare a unui restaurant. Acest lucru se revendică a fi realizat prin dezvoltarea și aplicarea mai multor concepte esențiale. În acest fel, proiectarea se face prin tehnici de programare Design by Contract, polimorfism, Design Patterns, implementări JCF HashMap și HashSet, precum și prin serializare.

## Obiective secundare

Un prim obiectiv secundar este constituit de respectarea paradigmelor programării orientate pe obiecte, structurarea codului in clase, ceea ce face ca acesta să fie mai lizibil și mai facil de urmărit. Realizarea diagramelor de clase și a use case-urilor reprezintă un alt obiectiv secundar al acestui proiect, precum și dezvoltarea ulterioară, propunându-se o serie de îmbunătățiri ale aplicației pentru o versiune mai complexă.

# Analiza problemei

Tema acestui proiect propune implementarea unui sistem de simulare a gestionării unui restaurant. Aplicația trebuie să aibă trei tipuri de utilizatori (administrator, chelner și bucătar). Administratorul poate adăuga, șterge sau modifica produsele existente în meniu. Chelnerul are abilitatea de a crea o nouă comandă pentru o masă, de a adăuga elemente din meniu, de a calcula factura pentru o comandă. Bucătarul este notificat de fiecare dată când trebuie să gătească mâncarea comandată prin intermediul chelnerului.

Implementarea claselor se realizează în funcție de o diagramă dată. Pentru simplificarea aplicației, sistemul se consideră a fi folosit de un singur administrator, un chelner și un bucătar. De asemenea, nu este nevoie de un proces de logare.

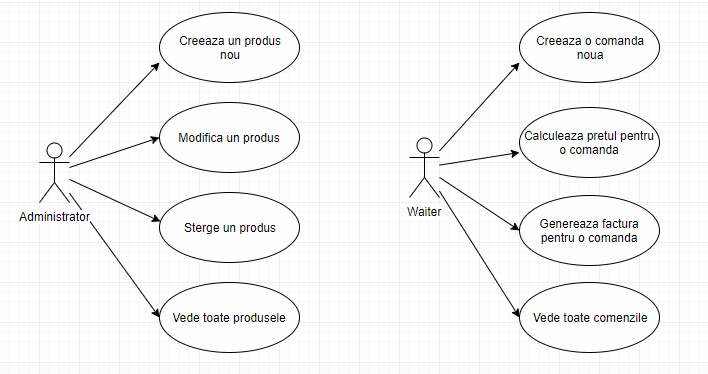
În cele ce urmează, voi prezenta mai detaliat cerințele care se cer a fi satisfăcute. În primul rând, se definește interfața RestaurantProcessing, ce conține principalele operații care pot fi executate de chelner sau de administrator, după cum urmează. Adminul are capacitatea de a crea noi elemente de meniu, de a le șterge sau de a le edita. Chelnerul poate crea o nouă comandă, calcula prețul unei comenzi și genera o factură în formatul .txt.

În al doilea rând, se definește și se implementează clasele din diagrama oferită drept reper, folosindu-se două modele de design. Composite Design Pattern se folosește pentru a defini clasele MenuItem, BaseProduct și CompositeProduct. Observer Design Pattern este utilizat pentru a notifica ospătarul de fiecare dată când trebuie procesată o nouă comandă ce conține un produs te tip composite.

În al treilea rând, se implementează clasa Restaurant folosindu-se o colecție JCF predefinită, care utilizează o structură de date hashtable. Cheia din hashtable va fi generată pe baza clasei Order, care poate avea asociate mai multe MenuItems. Se folosește JTable pentru a afișa informațiile relaționate cu restaurantul. Voi prezenta acum pașii ce necesită a fi urmați în vederea realizării aspectelor menționate. Se definește o structură de tipul Map<Order>, Collection<MenuItem> pentru a stoca în clasa Restaurant datele corelate cu comanda. Cheia din Map va fi formată din obiecte de tipul Order, pentru care metoda hashCode() va fi suprascrisă pentru a calcula valoarea hash în Map din atributele comenzii (OrderID, date etc). Se definește o structură de tipul Collection<MenuItem> în care se va salva meniul restaurantului. Se cere a se alege cel mai potrivit tip de colecție pentru propria implementare. Se definește o metodă de acest tip pentru clasa Restaurant și se implementează clasa folosind metoda Design by Contract (implicând pre, post condiții, aserțiuni).

În ultimul rând, elementele din meniu pentru popularea obiectului Restaurant vor fi salvate folosind serializarea.

Mai jos este prezentata diagrama use case a proiectului.



# Proiectare

Proiectul este format din 3 pachete: BusinessLayer, DataLayer si PresentationLayer, care vor fi prezentate mai jos:

* BusinessLayer – pachet pentru functionalitatea programului; contine clasele: MenuItem, BaseProduct, CompositeProduct, Order, Restaurant si interfata IRestaurantProcessing; clasele BaseProduct si CompositeProduct mostenesc clasa abstracta MenuItem iar clasa restaurant mosteneste clasa Observable si implementeaza interfata IRestaurantProcessing
* DataLayer – pachet pentru serializare si generarea facturii; contine clasele: FileWriter (care genereaza factura) si RestaurantSerializer (care face serializarea si deserializarea)
* PresentationLayer – pachet pentru interfata grafica cu utilizatorul; contine clasele: AdministratorGUI (pentru administrator), WaiterGUI (pentru chelner), BigGUI (in care se alege administrator sau waiter), ChefGUI (care este notificat cand trebuie sa afiseze ceva) si Controller care face legatura intre interfata grafica si functionalitatea programului.

Structurile de date folosite sunt: ArrayList pentru produsele din meniu si Hashmap pentru comenzi. Hasmap-ul are ca cheie un obiect de tip Order si ca valoare un ArrayList de obiecte de tip MenuItem care vor fi selectate din meniu de catre waiter.

# Implementare

In continuare voi prezenta implementarea proiectului (clasele si metodele importante).

## Restaurant

Restaurant este clasa in care se tin produsele si comenzile. Aceasta clasa implementeaza interfata IRestaurantProcessing care este mai jos.

public interface IRestaurantProcessing {  
 void createMenuItem(MenuItem menuItem);  
 void deleteMenuItem(MenuItem menuItem);  
 void editMenuItem(int i, String newName, double newPrice);  
 void createOrder(Order order, ArrayList<MenuItem> items);  
 double computePriceForOrder(int i);  
 void generateBill(int index);  
}

Metodele cele mai importande din aceasta clasa sunt:

* createMenuItem – care are ca parametru un obiect de tip MenuItem care este adaugat un lista de produse. Daca produsul exista deja in lista sau daca produsul trimis ca parametru este null, se va arunca o eroare. Daca produsul este compus, trebuie sa aiba cel putin 2 elemente in lista de ingredinete.
* public void createMenuItem(MenuItem menuItem) {  
   assert isWellFormed() : "Something is wrong on restaurant";  
   assert menuItem != null && !menuItem.getName().isEmpty() && items != null : "Cannot create item!!";  
   assert menuItem instanceof BaseProduct ||  
   menuItem instanceof CompositeProduct &&  
   ((CompositeProduct)menuItem).getProducts().size() >= 2 : "You must to select more that one item for composite product!";  
   assert !items.contains(menuItem) : "The item already exist";  
   items.add(menuItem);  
   assert isWellFormed() : "Something is wrong on restaurant";  
  }
* deleteMenuItem – care are ca parametru tot un obiect de tip MenuItem si care il sterge din meniu. Daca parametrul este null se va arunca o eroare.

public void deleteMenuItem(MenuItem menuItem) {  
 assert isWellFormed() : "Something is wrong on restaurant";  
 assert menuItem != null && items != null : "Cannot delete item!!";  
 if (menuItem instanceof BaseProduct){  
 ArrayList<MenuItem> itemsToDelete = new ArrayList<>();  
 for (MenuItem item: items){  
 if (item.containsItem((BaseProduct) menuItem)){  
 itemsToDelete.add(item);  
 }  
 }  
 items.removeAll(itemsToDelete);  
 }  
 items.remove(menuItem);  
 assert isWellFormed() : "Something is wrong on restaurant";  
}

* editMenuItem – care are ca parametru un indice de tip int, un string is un pret de tip double si care schimba numele si pretul produsului de pe pozitia indicelui primit ca parametru. Daca indicele este negativ (cee ace din interfata nu se poate intampla), daca numele este null sau daca pretul este mai mic sau egal cu zero se arunca o eroare.
* public void editMenuItem(int i, String newName, double newPrice) {  
   assert isWellFormed() : "Something is wrong on restaurant";  
   assert i >= 0 && newName != null && newPrice > 0 : "Cannot edit item";  
   assert items != null : "Invalid data on fields!";  
   MenuItem item = items.get(i);  
   item.setName(newName);  
   item.setPrice(newPrice);  
   assert isWellFormed() : "Something is wrong on restaurant";  
  }
* createOrder – care primeste ca parametru un obiect de tip Order si un ArrayList de obiecte de tip MenuItem si adauga in HashMap un element cu cheia order (primit ca parametru) care are ca valoare lista primita ca parametru. Daca exista deja un order cu acelasi id ca se ca arunca o eroare. De fiecare daca cand se adauga un order, clasa chef este notificata, fiindui trimis ca parametru lista cu produse.

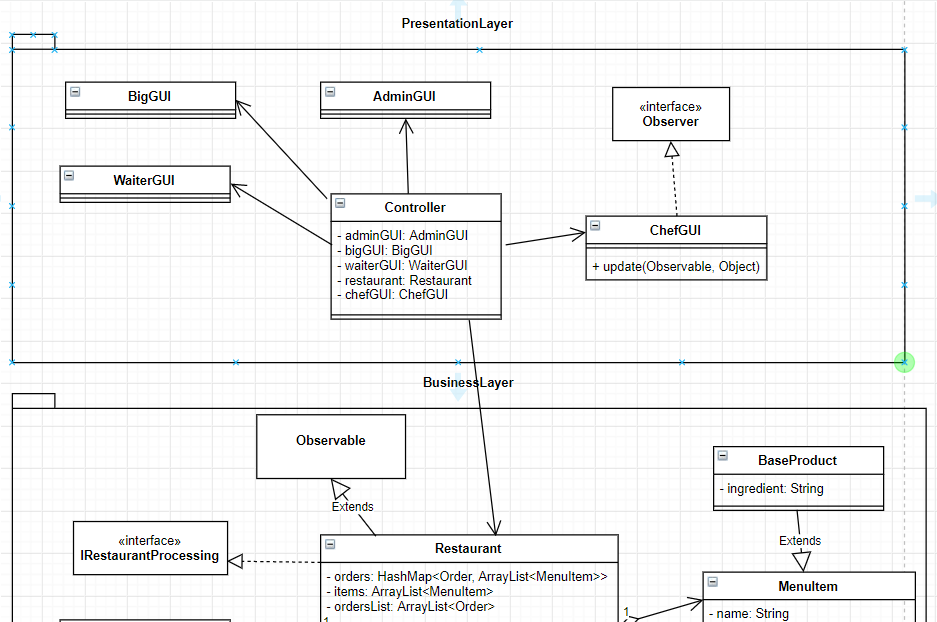
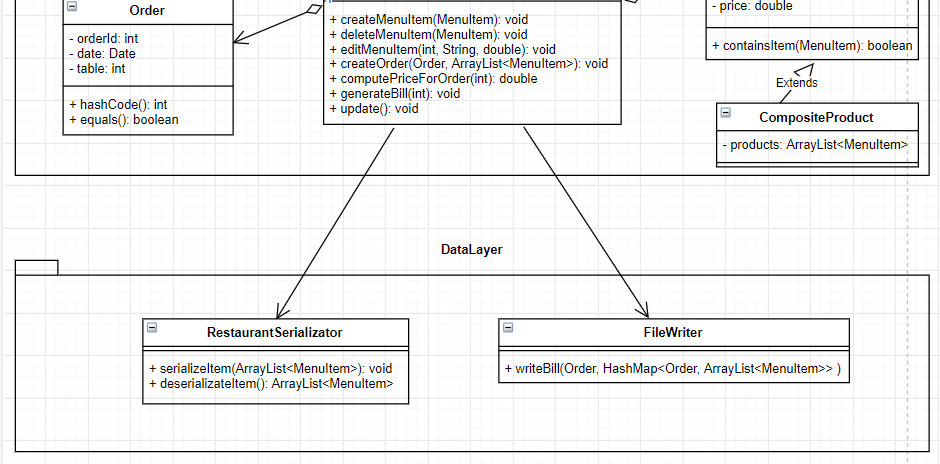
public void createOrder(Order order, ArrayList<MenuItem> items) {  
 assert isWellFormed() : "Something is wrong on restaurant";  
 assert !containsId(order.getOrderId()) && items != null : "Cannot add order!";  
 orders.put(order, items);  
 ordersList.add(order);  
 setChanged();  
 notifyObservers(items);  
 assert isWellFormed() : "Something is wrong on restaurant";  
}

* computePriceForAnOrder – care primeste ca parametru un indice si care returneaza pretul total al order-ului de pe pozitia indicelui. Daca indicele este mai mic decat 0 se va arunca o eroare.
* public double computePriceForOrder(int i) {  
   assert i > 0 : "Cannot make the bill";  
   double fullPrice = 0;  
   Order order = ordersList.get(i);  
   for (MenuItem item : orders.get(order)){  
   fullPrice += item.getPrice();  
   }  
   return fullPrice;  
  }
* generateBill – care primeste ca parametru un indice si care creeaza o facutra (fisier de tip text) cu datele comenzii de pe pozitia primita ca parametru

public void generateBill(int index) {  
 Order order = ordersList.get(index);  
 FileWriter.*writeBill*(order, orders);  
}

* update – care updateaza fisierul de serializare (rescrie lista de produse in fisier)

Mai jos este prezentata diagrama UML de clase si de pachete.



## MenuItem

Clasa abstracta care reprezinta un produs din meniu. O metoda importanta din aceasta clasa este containtItem. Clasele BaseProduct si CompositeProduct mostenesc aceasta clasa. Metoda containtItem din BaseProduct returneaza false tot timpul, iar cea din clasa Composite product returneaza true daca contine produsul primit ca parametru si false in caz contrar.

public boolean containsItem(BaseProduct product) {  
 if (products.contains(product)){  
 return true;  
 }  
 for (MenuItem menuItem : products){  
 if (menuItem.containsItem(product)){  
 return true;  
 }  
 }  
 return false;  
}

## Order

Aceasta clasa reprezinta o comanda si are ca variabile de instanta un id de tip int, o data de tip Date si masa la care se face comada care este de tip int. In aceasta clasa este suprascrisa metoda equals pentru a putea fi folosita ca cheie la un HashMap.

public boolean equals(Object obj) {  
 if (obj == this){  
 return false;  
 }  
  
 if (!(obj instanceof Order)){  
 return false;  
 }  
 Order order = (Order)obj;  
 return orderId == order.orderId &&  
 date.equals(order.date) &&  
 table == order.table;  
}

## FileWriter

Aceasta clasa contine o singura metoda statica care primeste ca parametru un order si un HashMap, cauta comanda cu cheia order primita ca parametru in hashmap si folosind datele acelei comenzi creeaza un fisier de tip txt si afiseaza daca curenta, fiecare produs de la acea comanda impreuna cu pretul fiecaruia si la final pretul total al comenzii.

public static void writeBill(Order order, HashMap<Order, ArrayList<MenuItem>> orders){  
 try {  
 PrintWriter writer = new PrintWriter("BillFor " + order.getOrderId() + ".txt", "UTF-8");  
  
 writer.println("Date: " + order.getDate());  
 writer.println();  
 int total = 0;  
 for (MenuItem product : orders.get(order)){  
 writer.println(product.getName()  
 + " price " + product.getPrice() +  
 " RON");  
 total += product.getPrice();  
 }  
 writer.println();  
 writer.println("Total to pay: " + total + " RON");  
  
 writer.close();  
 } catch (FileNotFoundException | UnsupportedEncodingException ex) {  
 ex.printStackTrace();  
 }  
}

## RestaurantSerializer

Aceasta clasa are doua metode pentru serializare si deserializare. Prima metoda serializeaza un ArrayList de MenuItem primit ca parametru, iar cealalta metoda citeste un ArrayList din fisier si il returneaza.

## AdministratorGUI

Reprezinta interfata pentru administrator. Aceasta interfata care 3 butoane (CREATE, EDIT si DELETE), un tabel in care sunt produsele din meniu, un text area in care for fi afisate elementele dintr-un compositeProduct atunci cand se da dublu click pe o linie din tabel si un panel in care sunt doua campuri pentru introducerea numelui si pretului la creearea sau editarea unui produs nou, plus inca 2 butoane de adaugare si unul de salvare a datelor pentru un produs.

## WaiterGUI

Aceasta interfata este asemanatoare cu administratorGUI doar ca aici avem 3 butoane pentru creearea unui order, calcularea pretului si generarea facturii. Exista 2 tabele, unul pentru comenzi si unul pentru produsele din meniu. Chelnerul va introduce in campuri id-ul si masa comenzii, va selecta produsele pentru acea comanda si va apasa butonul ADD, iar acea comanda va aparea in tabelul pentru comenzi. Daca dam dublu click pe o comanda din tabel, va aparea intr-un text area produsele din acel order.

## ChefGUI

Clasa care reprezinta “bucatarul”, care contine un text area in care vor aparea produsele compuse unui order in momentul in care este creeat. Acest lucru se face folosind interfata Observer si metoda update.

public void update(Observable o, Object arg) {  
 ArrayList<MenuItem> items = (ArrayList<MenuItem>)arg;  
 food.setText("");  
 for (MenuItem item : items){  
 if (item instanceof CompositeProduct){  
 food.append("- ");  
 food.append(item.getName());  
 food.append("\n");  
 }  
 }  
 food.revalidate();  
 this.revalidate();  
}

## BigGUI

Aceasta clasa contine doar doua butoane, unul pentru administrator si unul pentru waiter prin care se alege una din interfetele celor doua variante.

## Controller

Aceasta clasa face legatura intre interfata grafica si functionalitate. Contine ascultatoarele pentru toate butoanele. Are ca variabile de instansa cate un obiect de tip AdministratorGUI, ChefGUI, WaiterGUI si Restaurant.

# Concluzii

Din aceasta tema am invatat sa folosesc design by contract, composite design pattern, observer design pattern si sa folosesc serializarea pentru stocarea datelor.

# Bibliografie

<https://stackoverflow.com/questions/16295942/java-swing-adding-action-listener-for-exit-on-close>

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

<http://coned.utcluj.ro/~salomie/PT_Lic/5_Courses/>