**Documentație**

la disciplina

**Tehnici de programare**

**Tema 3 – Aplicație de management pentru comenzi**

**Popa Cristian Mihai, grupa 30224**

**An academic: 2021-2022**

Cuprins

[1. Obiectivul temei: 3](#_Toc99664753)

[2. Analiza problemei: 3](#_Toc99664754)

[3. Proiectare: 4](#_Toc99664755)

[4. Implementare: 5](#_Toc99664756)

[5. Concluzie: 9](#_Toc99664757)

[6. Bibliografie: 10](#_Toc99664758)

# Obiectivul temei:

* **Obiectiv principal:**

Obiectivul acestei teme consta in creare unui program care să simuleze o aplicație des întâlnita in ziua de astăzi, sub diferite forme. Aplicația constă în simularea unui sistem ce gestionează comenzi, constituite din anumite produse, ce sunt comandate de anumiți clienți. Ap

Aplicația trebuie să poată crea un client nou, un produs nou și să înregistreze o nouă comandă. De asemenea, se pot modifica datele despre clienți, produse sau comenzi. Se pot crea mai multe comenzi. Aplicația va comunica cu un server prin limbajul relațional MySQL.

* **Obiectiv secundare**:
  + Revizualizarea unor paradigme de programare orientata pe obiect: Interfețe, tehnici de reflecție, generarea unui javadoc, MySQL etc.;
  + Dezvoltarea unor clase de obiect adecvate: crearea unor clase potrivite pentru o persoana(in cazul nostru, client), produs și comandă, ce va conține informații generale despre client și despre produsele alese.
  + Dezvoltarea unor algoritmi eficienți si ușor de înțeles pentru prelucrarea in paralel a mai multor cereri;
  + Implementarea operațiilor si structuri de date specifice pentru a ne folosi de tehnicile de refelcție astfel încât programul sa funcționeze conform cerinței;
  + Implementarea unei interfețe grafice ușor de utilizat si ușor de înțeles;
  + Implementarea soluției pentru a avea un o aplicație ce simulează o aplicație din viața realăș

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

# Analiza problemei:

Trăim într-o lume tehnologizată, putem sa obținem tot ceea ce ne dorim doar cu un singur click, iar după o perioadă de timp va ajunge la noi în fața ușii. Aplicația are ca scop simularea unei aplicații folosite pentru a plasa comenzi ce conțin anumite produse alimentare. Aplicația va putea înregistra noi comenzi, noi produse, iar clienții vor putea să plaseze diferite comenzi. Pentru a vedea cum funcționează acest sistem, utilizatorul are la dispoziție o interfață prietenoasă și ușor de utilizat ce este formată din 4 panouri, cu nume sugestive: Client, Produs, Comanda și Istoric Comanda:

* Client: aici se realizează toate operațiile legate de client(creare, ștergere și update). Aici se pot vizualiza și restul clienților. Acest panou conține 6 câmpuri ce trebuie completate în vederea creări unui nou client. În partea de jos a panoului avem un tabel, unde putem vizualiza toți clienți noștri care există deja în baza de date. Tot acolo în tabel putem sa facem update la clientului care dorim să îi actualizăm datele
* Produs: aceeași interfață ca la client, dar avem mai puține câmpuri în vederea creări unui produs nou. În rest, este configurat la fel ca mai sus.
* Comanda: asemănător cu interfețele anterioare. De aici se pot crea noi comenzi. Mai întâi se alege clientul pe care va fi comanda, după se aleg produsele ce vor fi introduse în coș. La fiecare produs adăugat în coș, se actualizează și prețul total al coșului. Produsele din coș pot fi vizualizate în tabelul de jos. După, clientul poate opta pentru finalizarea comenzii.
* Istoric Comanda: este o interfață simplă, ce conține doar 2 tabele și un buton de ștergere a comenzii. În acest panou se pot vizualiza toate comenzile, precum și produsele conținute.

Pentru a funcționa corect, programul presupune introducerea datelor sub o anumita forma, ușor de înțeles. Câmpurile sunt sugestive, cu ce date trebuie introduse.

Programul verifica fiecare input al utilizatorului, deoarece pot apărea greșeli(de exemplu: utilizatorul introduce litere sau alte semne nepermise) . In cazul unei erori, utilizatorul va fi notificat.

După introducerea datelor, în funcție de ceea ce dorește să introducă, poate purcede mai în a vizualiza și alte operații, cum ar fi ștergere sau update sau chiar crearea unei comenzi.. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

# Proiectare:

* **Alegerea unor clase de obiecte de baza adecvate:**
  + O comanda, descrisa de clasa Comanda va fi formata din unul sau mai multe produse și un singur client.
  + Clientul este descris de clasa Client, ce poseda următoarele atribute: „id” ce este unic pentru fiecare utilizator creat(pentru baza de date MySQL) și restul parametrilor sunt folosiți pentru a descrie lucruri uzuale despre o persoană: vârstă, adresă, număr de telefon, nume și prenume etc.
  + Client poseda câteva metode, getter si setter, un constructor si alte metode de afișare si o metoda de comparație.
  + Clasa Server este asemănătoare cu clasa client. Conține un câmp „id” ce este unic pentru fiecare produs, restul atributelor fiind uzuale în descrierea unui produs: cantitate și preț. De asemenea, această clasă conține getter si setter, un constructor si alte metode de afișare
  + O comanda, descrisa de clasa Comanda va fi formata din unul sau mai multe produse și un singur client. Clasa comanda conține un câmp „id” ce este unic pentru fiecare comandă și informații uzuale, totalul comenzii și clientul care a comandat-o.
  + A patra clasă, ProdComenzi, reprezintă produsele ce află pe o anumită comandă. Câmpul id este asemănător cu cel al comenzii. De asemenea aici sunt diferite informații despre produsele din comandă
  + Aceste 4 clase stau la baza programului, iar ceea ce conține fiecare este foarte intuitiv si ușor de înțeles. Clasele de obiecte au fost alese special sa oglindească reprezentarea lor din tabelele din baza de date MySQL.
* **Împărțirea programului pe clase:**

Programul este împărțit in 5 pachete mari:

* Pachetul GUI (Grafical User Interface) ce cuprinde 2 clase, Controller si View ce stau la baza funcționarii aplicației. Clasa View este responsabila pentru crearea interfeței grafice, iar clasa Controller ne ajuta sa interacționăm cu Text-filed-urile si butoanele de pe Form;
* Pachetul Connection conține o clasă ce este responsabilă de conectarea aplicației la baza de date.
* Pachetul DataAcces conțin clasele ce se folosesc de tehnicile de reflecție și descriu operațiile ce se pot aplica asupra clienților, comenzilor și a produselor.
* Pachetul BussinesLogic conțin implementarea metodelor din DataAcces, oferind anumite informații suplimentare.
* In plus, mai avem o clasa numita „MainClass” ce nu aparține niciunei clase care deschide interfața si inițializează controller-ul.
* Împărțirea lor pe clase:
* O imagine care conține text

  Descriere generată automat

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

* **Algoritmi folosiți:**

Cum prelucrează programul datele din câmpurile date ca si input? Nu este foarte mare filozofie, întrucât în toate câmpurile se încearcă convertirea datelor introduse. In acest caz, utilizatorul va fi notificat de aplicație, repetând introduce datele din nou, dacă greșește. După ce datele sunt validate și prelucrate, se vor trimite diferite cereri către server-ul central, în vederea stocării sau obținerii de informație.

# Implementare:

In cele ce urmează, se vor prezenta modul in care au fost implementate clasele de obiecte, din cele 5 pachete. Nu vor fi incluse setter si getter, deoarece se subînțelege ca au fost folosite.

* **Pachetul Gui:**
  + Clasa View:

package GUI;

public class View extends JFrame{....}

Clasa View conține si inițializează toate componentele vizuale menționate mai sus: 4 butoane, 5 TextField-uri, 5 laber-uri si un area text. Este sunt plasate un panouri invizibile care la rândul lor aranjate intr-un mod specific in fereastra principala(putem considera ca sunt toate afisate sub forma de o coloana). Câteva inițializări si metode din clasa View:

public void initComboBoxes(){  
 comb1.removeAllItems();comb1.revalidate();comb1.repaint();  
 comb2.removeAllItems();comb2.revalidate();comb2.repaint();  
  
 List<Client> listaClienti = controller.getClientiFromDB();  
 List<Produs> listaProduse = controller.getProdusFromDB();  
  
 for(Client c : listaClienti){  
 comb1.addItem(new String(c.getId() + " " + c.getNume() + " " + c.getPrenume()));  
 }

* + Clasa Controller:

public void writeToFile(String msg) {  
 List<Comanda> listComanda = new ArrayList<>();  
 listComanda = comandaBLL.findAll();  
 int size = listComanda.size()+1;  
 String path = "order\_" + size + ".txt";  
 try {  
 File file = new File(path);  
 if (file.createNewFile()) {  
 System.*out*.println("Factura generata: " + file.getName());  
 } else {  
 System.*out*.println("File already exists.");  
 }

}

Poate fi considerat creierul aplicației, întrucât comunica intre interfața vizuala si logica din spatele programului, ajuta la rezolvarea operației selectate si la livrarea rezultatului.

* **Pachetul Connection:**
  + Clasa ConnectionFactory:

In aceasta clasa se poate vizualiza foarte bine care sunt pașii în vederea stabilirii, menținerii și închiderii unei conexiuni cu baza de date. Această clasă manageriză cum comunică aplicația cu serverul și are grijă ca la efectuarea operațiilor conexiunea nu pică sau serverul este afectat sau supra-încărcat de mai multe cereri.

private Connection createConnection() {  
 Connection connection = null;  
 try {  
 connection = DriverManager.*getConnection*(*DBURL*, *USER*, *PASS*);  
 } catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, "An error occured while trying to connect to the database");  
 e.printStackTrace();  
 }  
 return connection;  
}

Aceasta metoda ne închide conexiunea după efectuarea unei operații:

public static void close(Statement statement) {  
 if (statement != null) {  
 try {  
 statement.close();  
 } catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, "An error occured while trying to close the statement");  
 }  
 }  
}

* **Pachetul DataAcces:**

În acest pachet avem clasele ce descriu operațiile mai în detaliu pentru fiecare clasă din pachetul Model(Client, Produs și Comandă).

* + Clasa AbstractDao:

Clasa pentru operațiile uzuale ce se pot aplica pentru toate clasele din Model, de exemplu select în funcție de id, selecție totală sau ștergere

public T delete(int id){  
 Connection connection = null;  
 PreparedStatement statement = null;  
 String query = createDelete("id");  
  
 try {  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(query);  
 statement.setInt(1, id);  
 statement.executeUpdate();  
  
 } catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, type.getName() + "DAO:Delete " + e.getMessage());  
 } finally {  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
  
 return null;  
}

De asemenea, aici și cele 2 metode ce extrag informațiile din baza de date cu ajutorul tehnicilor de reflecție. De exemplu, metoda ce ne extrage atributele din baza de dată și le folosește ca nume pentru coloane din tabelele din aplicație.

public String[] collumnNames()  
{  
 String[] fields = new String[type.getDeclaredFields().length];  
 int cont = 0;  
 for (Field f : type.getDeclaredFields())  
 fields[cont++] = f.getName();  
 return fields;  
}

* + Clasa ClientDao:

Aici sunt implementate anumite operații specifice pentru clasa Client, gen insert și update. De exemplu, pentru inserarea unui client in baza de date:

private static final String *insertStatementString* = "INSERT INTO client (nume,prenume,varsta,email,adresa,nrTelefon)"  
 + " VALUES (?,?,?,?,?,?)";  
@Override  
public Client insert(Client cl){  
 Connection connection = ConnectionFactory.*getConnection*();  
 PreparedStatement insertStatement = null;  
 int insertedId=-1;  
 try {  
 insertStatement = connection.prepareStatement(*insertStatementString*, Statement.*RETURN\_GENERATED\_KEYS*);  
 insertStatement.setString(1, cl.getNume());  
 insertStatement.setString(2, cl.getPrenume());  
 insertStatement.setInt(3, cl.getVarsta());  
 insertStatement.setString(4, cl.getEmail());  
 insertStatement.setString(5, cl.getAdresa());  
 insertStatement.setString(6, cl.getNrTelefon());  
 insertStatement.executeUpdate();  
  
 }catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, "ClientDAO:insert " + e.getMessage());  
 } finally {  
 ConnectionFactory.*close*(insertStatement);  
 ConnectionFactory.*close*(connection);  
 }  
 return cl;  
}

Analog și pentru celelalte clase Dao.

* **Pachetul Model:**
  + Clasa Client: una din clasele se stau la baza aplicației
* private int id;  
  private int varsta;  
  private String nrTelefon = new String();  
  private String nume = new String();  
  private String prenume = new String();  
  private String email = new String();  
  private String adresa = new String();  
  public Client(int id, String name, String prenume, int age, String email, String adress, String tel) {  
   this.id = id;  
   this.varsta = age;  
   this.nrTelefon = tel;  
   this.nume = name;  
   this.prenume = prenume;  
   this.email = email;  
   this.adresa = adress;  
  }

. . . }

Dintre metodele importante, regăsim:

@Override  
public String toString(){  
 return "Client[id="+this.id+", Nume="+this.nume+", prenume="+this.prenume+", varsta="+this.varsta+  
 ", email="+this.email+", adresa="+this.adresa+", telefon="+this.nrTelefon+"]";  
}  
  
@Override  
public boolean equals(Object o) {  
 if (this == o) return true;  
 if (!(o instanceof Client)) return false;  
 Client client = (Client) o;  
 return getId() == client.getId() && getVarsta() == client.getVarsta() && getNrTelefon().equals(client.getNrTelefon()) && getNume().equals(client.getNume()) && getPrenume().equals(client.getPrenume()) && getEmail().equals(client.getEmail()) && getAdresa().equals(client.getAdresa());  
}

Analog și pentru restul de 3 clase. Conțin informații uzuale.

* + Clasa Produs:
* private int id;  
  private String numeProdus;  
  private int pretProdus;  
  private int cantitate;  
    
  public Produs(){}  
    
  public Produs(int id, String nume ,int price, int cant){  
   this.id=id;  
   this.numeProdus=nume;  
   this.pretProdus=price;  
   this.cantitate=cant;  
  }

. . .}

* **Pachetul BussinesLogic:**

Conține implementarea explicită a operațiilor din clasele DAO. Practic, cu ele vom lucra când efectuăm diferite operații în interfața grafica. Sunt apelate funcțiile din clasele BLL, care ele vor apela mai departe metodele din clasele Dao.

* + Clasa ClientBLL: exemplu pentru metoda de găsire a tuturor intrărilor din tabela client:

private ClientDao cldao = new ClientDao();  
  
public List<Client> findAll() {  
 List<Client> rez = cldao.findAll();return rez;  
}

# Concluzie:

Aplicația dezvoltata respecta paradigmele programării orientate pe obiect și a manipulării unei baze de date MySQL și este un bun reminder al cunoștințelor învățate sem. trecut. Sistemul creat gestionează corespunzător o aplicație ce poate suporta mai multe operații cu privire la manipularea comenzilor efectuate de la distanță. Se aseamenă cu aplicațiile ce se regăsesc în viața reală(aplicații de food delivery de pe telefon de exemplu)

Deși este simpla aplicația si își deservește scopul, totuși este loc de îmbunătățiri si upgrade-uri. Aplicația poate avea pe viitor o metodă de login.

# Bibliografie:

<https://docs.oracle.com/en/>

<https://www.baeldung.com/java-write-to-file>

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

<https://www.w3schools.com/java/java_files_create.asp>

<https://www.jetbrains.com/help/idea/working-with-code-documentation.html>

<https://www.baeldung.com/java-pdf-creation>

https://jenkov.com/tutorials/java-reflection/index.html

http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html