Tema 3

Program de gestiune a unui depozit

Proiect realizat de

Bursuc Andrei

Seria A, Grupa 30224

Contents

[**1.** **Obiectivul temei** 3](#_Toc481064211)

[**2.** **Analiza problemei, modelare, scenarii, cazuri de utilizare** 3](#_Toc481064212)

[**3.** **Proiectare** 4](#_Toc481064213)

[I. Decizii de proiectare 4](#_Toc481064214)

[II. Diagrama UML 5](#_Toc481064215)

[III.Proiectarea claselor și relațiile dintre ele 6](#_Toc481064216)

[IV. Interfață utilizator 16](#_Toc481064217)

# **Obiectivul temei**

Acest proiect urmareste realizarea unui program ce ajuta la gestionarea produselor dintr-un depozit, el fiind destinat atat clientilor cat si detinatorilor depozitului. Programul trebuie sa ofere atat clientilor cat si propietarului o interfata prin care ei pot sa cumpere respectiv sa adauge produse, sa creeze conturi noi de clienti, sa-si modifice datele, sa vada istoricul cosurilor pentru fiecare client in parte sau sa modifice produse. Toate aceste operatii, exceptand cea de creeare de cont nou de client, pot fi efectuate dintr-o interfata la care ai acces doar dupa ce te loghezi cu un username si o parola.

# **Analiza problemei, modelare, scenarii, cazuri de utilizare**

Problema pe care o ridica aceasta aplicatie este data de modul in care legam programul nostru de o baza de date care o sa stocheze toate datele necesare gestionarii depozitului si permitem utilizatorului sa vada si sa acceseze prin intermediul interfetei toate datele din baza de date.

Pentru inceput, in fereastra de logare, un utilizator nou trebuie mai intai sa-si creeze un cont nou prin intermediul butonului „Cont nou”, dupa in fereastra nou aparuta sa-si introduca datele si sa-si creeze contul. Dupa ce contul a fost creat sau daca utilizatorul are deja un cont, el se poate loga si accesa celelalte functii ale programului. O prima functie este cea de editare a datelor personale singurele date ce nu pot fi editate fiind id-ul si username-ul (ce trebuie sa ramana unice), o alta functie este cea de vizualizare a cosurilor deja comandate („Istoric cosuri”) utila atunci cand vrei sa vezi ce comenzi ai mai avut in aplicatie. Una din functiile principale este cea de vizualizare a produselor din depozit ce deschide o fereastra noua in care un client poate sa adauge produse noi in cos, iar proprietarul depozitului sa adauge sau sa editeze produse. In fine un client poate sa vada prin intermediul butonului „Cos curent” comenzile curente din cos, dar si pretul total al cosului, iar prin butonul „Finalizare comanda ” incheie comanda si genereaza factura pentru cos. Toate aceste operatii sunt facute pentru clientul logat, fiind imposibil sa existe mai multi clienti logati la un moment dat. Singurul mod prin care putem parasi fereastra de operatii este prin intermediul butonului „Delogare”.

Baza de date este alcatuita din 4 tabele: Clienti, Produs, Comanda\_Articol si Cos. Tabelul clienti definit de campurile: id, username, nume, adresa, email, varsta, parola si logat (camp de tip boolean care confirma daca este sau nu logat clientul), acest tabel ne ajuta sa stocam datele despre client si ne spune daca un client este logat sau nu. Tabelul Produs cu campurile: id, descriere, cantitate, pret stocheaza datele despre produse; Tabelul cos definit de: id, pret\_total, client\_id, data\_creare, comanda\_finalizata ne ajuta sa facem legatura intre comenzi si fiecare client in parte, un client putand sa aiba mai multe comenzi ce au acelasi cos\_id, tabelul comanda\_articol este definit de: id, cos\_id, produs\_id, cantitate, pret\_bucata.

# **Proiectare**

## Decizii de proiectare

Pentru realizarea aplicatiei am optat pentru utilizarea arhitecturii stratificate care imparte aplicatia in mai multe straturi:

„Presentation Layer” contine clasele folosite la definirea interfetei utilizatorului. Rolul său este de a evidenția informația obținută prin intermediul „Data Acces Layer”. Interfața a fost realizată cu ajutorul bibliotecii Javax.Swing utilizând un plugin special pentru IntelIJ denumit JFormDesigner, ce permite realizarea mai ușoară a unei interfețe atractive pentru utilizator și ușor de folosit.

„Business Layer” contine clasele care incapsuleaza logica aplicatiei

„Data Acces Layer” contine clasele ce contin interogarile pentru baza de date si practic in cazul de fata definesc functiile prin care programul interactioneaza cu baza de date;

In plus in cadrul acestei architecturi mai avem „Modelul” ce contine clase care modeleaza fiecare tabel in parte din baza de date cu care programul interactioneaza.

O alta tehnologie utilizata in cadrul proiectului este o baza de date, aceasta reprezentand o modalitate de stocare a unor [informatii](http://ro.wikipedia.org/wiki/Informa%C8%9Bie) si [date](http://ro.wikipedia.org/wiki/Dat%C4%83) pe un suport extern (un dispozitiv de stocare), cu posibilitatea extinderii usoare si a regasirii rapide a acestora.

**SQL** (de la numele englez Structured Query Language - limbaj de interogare structurat - care se pronunță [*es kiu el*)este un limbaj de programare specific pentru manipularea datelor în sistemele de manipulare a bazelor de date relaționale (RDBMS),  iar la origine este un limbaj bazat pe algebra relațională. Acesta are ca scop inserarea datelor, interogații, actualizare și ștergere, modificarea și crearea schemelor, precum și controlul accesului la date. A devenit un standard în domeniu (standardizat ANSI-ISO), fiind cel mai popular limbaj utilizat pentru creearea, modificarea, regăsirea și manipularea datelor de către SGBD-urile (Sistemele de Gestiune a Bazelor de Date) relaționale. Pe lângă versiunile standardizate ale limbajului, există o mulțime de dialecte și variante, unele proprietare, fiind specifice anumitor SGBD-uri și de asemenea conținând extensii pentru a suporta SBD-urile (Sistemele de Baze de Date) obiectuale (obiectual-relaționale).

SQL a fost inițial dezvoltat la compania IBM de către Donal D.Chamberlin și Raymond F.Boyce  în anul 1974. Inițial numele era **SEQUEL** (tot de la *Structured Query Language*) și avea ca scop manipularea și regăsirea datelor stocate în bazele de date relaționale IBM

Operaţiile de bază pe o bază de date sunt:

* crearea unui tabel

CREATE TABEL tbl\_name ([data\_type] [col\_name], ...)

* afişarea datelor unui table

SELECT select\_expr [, select\_expr ...]

[FROM table\_references [WHERE where\_condition]

* inserarea în baza de date

INSERT [INTO] tbl\_name [(col\_name,...)] VALUE ({expr},...),(...),...)

* modificarea datelor

UPDATE table\_reference

SET col\_name1={expr1|DEFAULT} [, col\_name2={expr2|DEFAULT}] ...

[WHERE where\_condition]

* ştergerea datelor

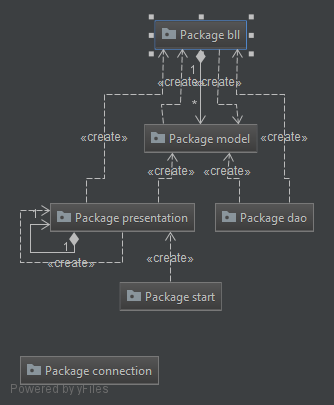
DELETE FROM tbl\_name

[WHERE where\_condition]

Codul a fost scris si compilat utilizând IDE-ul IntelIJ IDEA, mediu de dezvoltare software dezvoltat de catre compania JetBrains ( cunoscută anterior sub numele de IntelIJ ). El vine in doua variante ”Community Edition” și ”Ultimate Edition”, aceasta din urmă fiind disponibilă gratis studenților. Prima versiune a IntelIJ a fost lansată in Ianuarie 2001 și a fost unul dintre primele IDE-uri pentru java care dispunea de funcții avansate de ” Code Navigation” și ”Code Refactoring”. Ultima versiune 12.1 dispune de suport pentru Java 8, posibilitatea de dezvoltare de interfață grafică pentru Android, Play 2.0 și Scala.

## II. Diagrama UML

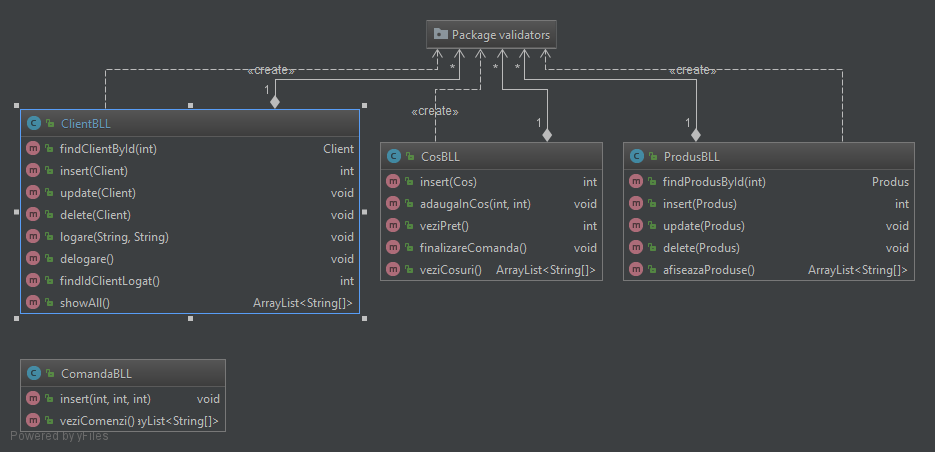
In continuare este prezentata diagrama uml a pachetelor folosite in program iar mai apoi diagrama uml a fiecarui pachet in parte:



## III.Proiectarea claselor și relațiile dintre ele

Proiectul fiind alcatuit din mai multe pachete vom prezenta pe rand fiecare pachet cu diagrama UML corespunzatoare.

**Package BLL ( Business Logic )**



Acest pachet contine clasele ce verifica functiile din pachetul „Data Acces” aplicand pe ele validatorii ce spun daca interogarea pe baza de date se poate sau nu executa. Un exemplu ar fi clasa ClientBLL.

**ClientBLL**

Ea contine o singura variabila privata:

private List<Validator<Client>> **validators**;

In ea stocam validatorii ce urmeaza a fi apelati.

public **ClientBLL**() {  
 **validators** = new ArrayList<Validator<Client>>();  
 **validators**.add(new EmailValidator());  
 **validators**.add(new AgeValidator());  
  
}

In constructor adaugam in lista validatorii ce vor trebui sa fie verificati pentru fiecare metoda din clasa.

public void logare(String username,String parola)  
{  
 **validators**.add(new DejaLogatValidator());  
 **validators**.add(new ParolaValidator());  
 **validators**.add(new ExistaUsernameValidator());  
  
 Client **client**=new Client(username,parola);  
  
 for(Validator<Client> **t**:**validators**)  
 **t**.validate(**client**);  
  
 ClientDAO.logare(username,parola);  
  
}

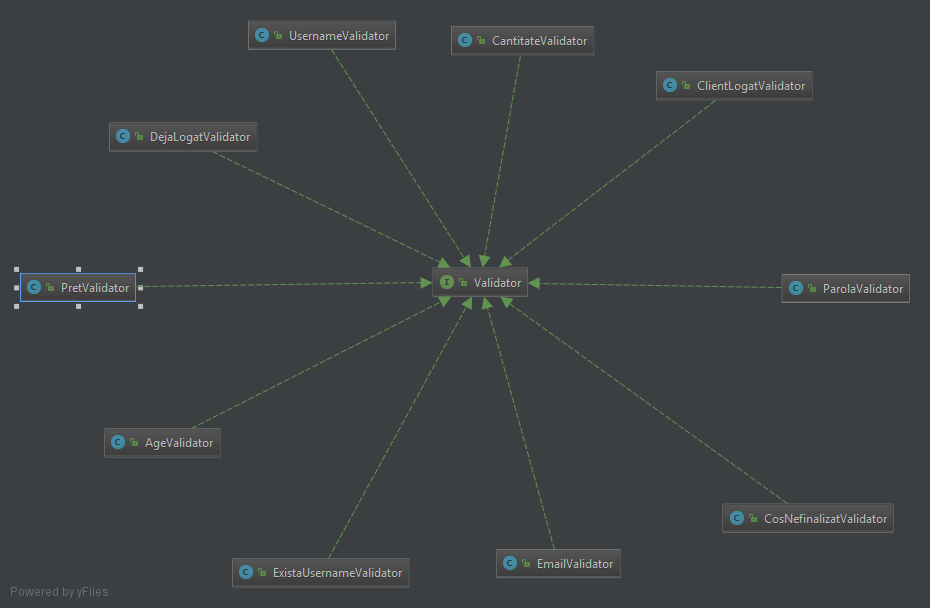
In clasa logare pentru validatorii adaugati deja in constructor care verifica daca email-ul introdus este corect si daca varsta este mai mare decat 18 ani, au mai fost adaugati 3 validatorii care verifica daca nu exista un client deja logat, parola este cea corecta si daca username-ul introdus exista. Dupa intr-un for care parcurge lista apelam metodele de validate si in caz in care datele introduse sunt corecte metoda din clasa ClientDAO se apeleaza.

Celelalte metode ce pot fi observate in diagrama UML respecta acelasi tipar cu mentiunea ca cele care nu trebuie verificare doar apeleaza metoda din ClientDAO un exemplu fiind:

public ArrayList<String[]> showAll()  
{  
 return ClientDAO.showAll();  
}

Clasele CosBLL, ComandaBLL si ProdusBLL respecta acelasi tipar ca si clasa ClientBLL avand lista de validators ca unica variabila.

**Package bll.validators**



Aceasta clasa toate clasele care implementeaza interfata Validator.

public interface Validator<T> {  
  
 public void validate(T t);  
  
}

Scopul acestor clase este sa verifice daca datele introduse de utilizator sunt corecte in cazul in care acestea sunt gresite metoda validate din clasa va arunca o exceptie. Cateva din clasele din package sunt:

public class PretValidator implements Validator<Produs> {  
  
 private static final int ***MIN\_PRET*** = 0;  
  
  
 public void validate(Produs t) {  
  
 if (t.getPret() < ***MIN\_PRET*** ){  
 throw new IllegalArgumentException("Cantitatea introdusa nu se incadreaza in limite!");  
 }  
  
 }  
}

In clasa PretValidator ce extinde interfata Validator metoda validate arunca o exceptie daca pretul produsului este negativ.

public class CosNefinalizatValidator implements Validator<Cos> {  
  
 private static final String ***cosNefinalizatString***;  
  
 static {  
 ***cosNefinalizatString*** = "select id from cos where comanda\_finalizata=false and client\_id=(select id from clienti where logat=true)";  
 }  
 public void validate(Cos t) {  
  
 Connection **dbConnection** = ConnectionFactory.getConnection();  
 PreparedStatement **cosNefinalizatStatement**=null;  
  
 try {  
 int **ok**=0;  
 **cosNefinalizatStatement**= **dbConnection**.prepareStatement(***cosNefinalizatString***);  
  
 ResultSet **rs** = **cosNefinalizatStatement**.executeQuery();  
 if(**rs**.next())  
 **ok**=1;  
  
 if(**ok**==1)  
 throw new IllegalArgumentException("Exista deja un cos creat!");  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }finally{  
 ConnectionFactory.close(**dbConnection**);  
 ConnectionFactory.close(**cosNefinalizatStatement**);  
 }  
  
 }  
}

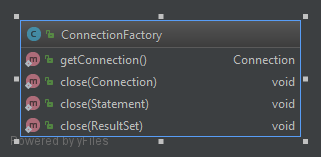
In clasa CosNefinalizatValidator ce extinde interfata Validator metoda validate verifica prin intermediul unei interogari pe baza de date daca exista un cos nefinalizat deja creat pentru client, si in cazul in care acesta exista arunca o exceptie. Exceptia arunca id-ul unui cos cu campul comanda\_finalizata setat pe valoarea false si campul client\_id egal cu id-ul clientulului logat.

public class ExistaUsernameValidator implements Validator<Client> {  
  
 private static final String ***usernameString***;  
  
 static {  
 ***usernameString*** = "select username from clienti";  
 }  
  
 public void validate(Client t) {  
  
 Connection **dbConnection** = ConnectionFactory.getConnection();  
 PreparedStatement **usernameStatement**;  
  
 try {  
 **usernameStatement** = **dbConnection**.prepareStatement(***usernameString***);  
 int **ok**=0;  
 ResultSet **rs** = **usernameStatement**.executeQuery();  
  
 while (**rs**.next()) {  
 String **username** = **rs**.getString("username");  
 if(**username**.equals(t.getUsername()))  
 {**ok**=1;  
 break;}  
 }  
  
 if(**ok**==0)  
 throw new IllegalArgumentException("Nu exista acest username!");  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

Un alt exemplu este clasa ExistaUsernameValidator care prin intermediul interogarii „Select username from clienti” verifica daca exista username-ul introdus in campul de logare si in cazul in care el nu exista se arunca o exceptie.

Restul claselor ce pot fi observate in diagrama UML respecta acelasi model ca si una din clasele descrise mai sus.

**Package Connection**



**Clasa ConnectionFactory**

Aceasta clasa este definita de variabilele:

private static final Logger ***LOGGER***=Logger.getLogger(ConnectionFactory.class.getName());  
private static final String ***DRIVER***="com.mysql.cj.jdbc.Driver";  
private static final String ***DBURL***="jdbc:mysql://localhost:3306/warehouse";  
private static final String ***USER***="root";  
private static final String ***PASS***="and5544";

Ele contin datele necesare conectarii programului la baza de date precum username-ul si parola, dar si driverul si url-ul.

De mentionat este faptul ca in aceasta clasa folosim design pattern-ul singleton ce permite crearea unei singure instante a clasei ConnectionFactory, acest lucru usurandu-ne modul in care realizam conexiunea cu baza de date inainte de apelarea unei interogari.

private static ConnectionFactory **singleInstance**=new ConnectionFactory();  
  
private **ConnectionFactory**(){  
 try{  
 Class.forName(***DRIVER***);  
 }  
 catch (ClassNotFoundException e)  
 {  
 e.printStackTrace();  
 }  
}

Alte metode sunt createConnection:

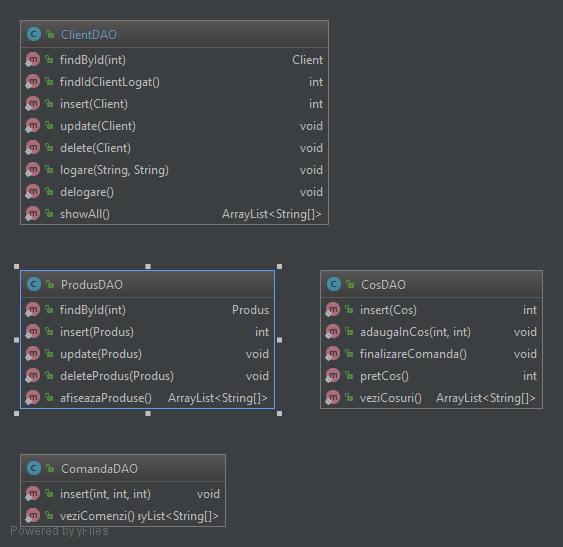
private Connection createConnection()  
{  
 Connection **connection**=null;  
 try {  
 **connection**=DriverManager.getConnection(***DBURL***,***USER***,***PASS***);  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return **connection**;  
  
}

Care se foloseste de URL, username si parola pentru a instantia obiectul de tip connection.

De asemenea mai sunt definite 3 metode care inchid conexiunile la baza de date. Una dintre ele fiind:

public static void close(Connection connection) {  
  
 try {  
 connection.close();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
}

**Package dao**



Aici sunt definite clasele care contin metodele prin care interactionam cu baza de date si putem vedea sau modifica continutul ei.

**Clasa ClientDao**

Clasa contine variabilele statice:

private final static String ***findStatementString***;  
private static final String ***insertStatementString***;  
private static final String ***updateStatementString***;  
private static final String ***deleteStatementString***;  
private static final String ***logareString***;  
private static final String ***delogareString***;  
private static final String ***findIdString***;  
private static final String ***showAllString***;

Ele sunt definite intr-un bloc static si sunt folosite pentru a putea apela o interogare pe baza de date:

static {  
 ***findStatementString*** = "SELECT \* FROM clienti where id=?";  
 ***insertStatementString*** = "insert into clienti(username,nume,adresa,email,varsta,parola)"+" VALUES (?,?,?,?,?,?)";  
 ***updateStatementString*** = "update clienti"+" set nume=?,adresa=?,email=?,varsta=?,parola=? where id=?";  
 ***deleteStatementString*** = "delete from clienti"+" where id=?";  
 ***logareString***="update clienti set logat=true where username=?";  
 ***delogareString***="update clienti set logat=false where logat=true";  
 ***findIdString***="select id from clienti where logat=true";  
 ***showAllString***="select id,username,nume,adresa,email,varsta from clienti";  
}

Un exemplu de metoda din aceasta clase e:

public static Client findById(int clientId){  
 Client **toReturn**=null;  
 Connection **dbConnection**= ConnectionFactory.getConnection();  
 PreparedStatement **findStatement**=null;  
 ResultSet **rs**=null;  
  
  
 try {  
 **findStatement**=**dbConnection**.prepareStatement(***findStatementString***);  
 **findStatement**.setInt(1,clientId);  
 **rs**=**findStatement**.executeQuery();  
 **rs**.next();  
 String **username**=**rs**.getString("username");  
 String **nume**=**rs**.getString("nume");  
 String **adresa**=**rs**.getString("adresa");  
 String **email**=**rs**.getString("email");  
 int **varsta**=**rs**.getInt("varsta");  
 String **parola**=**rs**.getString("parola");  
 **toReturn**=new Client(clientId,**username**,**nume**,**adresa**,**email**,**varsta**,**parola**);  
  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }finally {  
 ConnectionFactory.close(**rs**);  
 ConnectionFactory.close(**findStatement**);  
 ConnectionFactory.close(**dbConnection**);  
 }  
  
 return **toReturn**;  
  
}

In metoda mai intai obtinem conexiunea folosindu-ne de metoda getConnection() din ConnectionFactory, apoi instantiem obiectul de tip PreparedStatement cu stringul „find-StatementString”, setam prima variabila din interogarea definita in string ca si parametrul clientID primit de functie si executam interogarea cu metoda executeQuery salvand rezultatele in variabila de tip ResulSet rs. Cu rs.next() obtinem rezultatele pe care le preluam in variabile String si int cu metodele getString si getInt. De aceste variabile ne folosim sa instantiem variabila ce trebuie returnata de metoda, dar inainte de a returna rezultatul inchidem conexiunile realizate cu baza de date folosind metodele din ConnectionFactory.

Alta metoda care extrage date din tabel este metoda showAll care returneaza un ArrayList de tipul String[] ce contine datele din tabel referitoare la toti clientii din baza de date. Acest ArrayList va fi folosit la initializarea unui obiect de tip JTabel ce va contine aceste date.

Un alt tip de metoda din aceasta clasa, dar prezenta si in alte clase din acest pachet este cea prin care modificam date din baza de date, un exemplu fiind metoda insert.

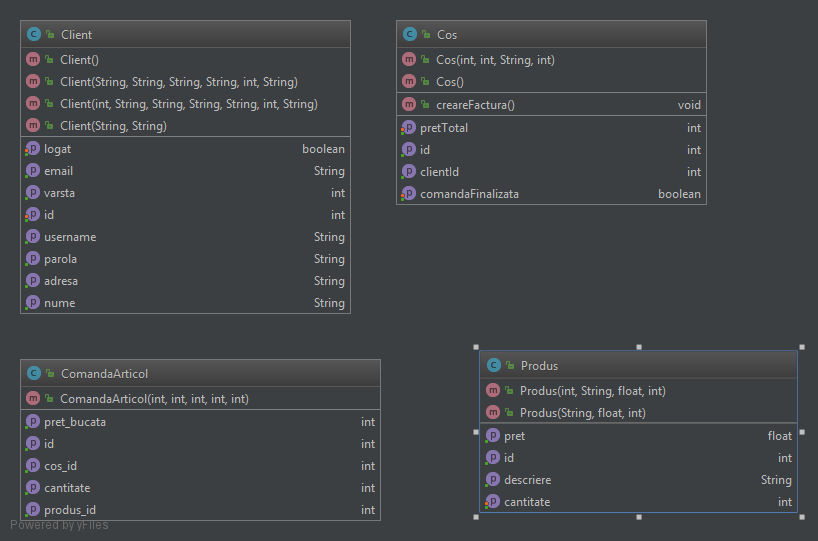
public static int insert(Client client){  
  
 Connection **dbConnection**= ConnectionFactory.getConnection();  
 PreparedStatement **insertStatement**=null;  
 int **insertedId**=-1;  
  
  
 try {  
 **insertStatement**=**dbConnection**.prepareStatement(***insertStatementString***, Statement.***RETURN\_GENERATED\_KEYS***);  
  
 **insertStatement**.setString(1,client.getUsername());  
 **insertStatement**.setString(2,client.getNume());  
 **insertStatement**.setString(3,client.getAdresa());  
 **insertStatement**.setString(4,client.getEmail());  
 **insertStatement**.setInt(5,client.getVarsta());  
 **insertStatement**.setString(6,client.getParola());  
  
 **insertStatement**.executeUpdate();  
  
 ResultSet **rs** = **insertStatement**.getGeneratedKeys();  
 if (**rs**.next()) {  
 **insertedId** = **rs**.getInt(1);  
 }  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }finally {  
 ConnectionFactory.close(**insertStatement**);  
 ConnectionFactory.close(**dbConnection**);  
 }  
  
 return **insertedId**;  
}

In aceasta metoda primii pasi sunt asemanatori cu aceia din metoda „findById” diferenta fiind ca metoda prepareStatement primeste ca parametrii insertStatementString si Statement.RETURN\_GENERATED\_KEYS, primul continand textul interogarii care introduce clientul in baza de date si cel de al doilea permitandu-ne sa extragem id-ul cheii primare a noului produs inserat in tabel, aceasta cheie fiind returnata la finalul metodei. Se mai poate remarca faptul ca pentru aceasta interogare trebe setati mai multi parametrii decat pentru metoda anterioara, acest lucru depinzand de interogare si ce parametrii primeste. Dupa ce toti parametrii au fost furnizati, apelam obiectul de tip Statement cu metoda „executeUpdate” folosita pentru interogarile care modifica baza de date. In finalul blocului try extragem cheia principala a clientului introdus si o salvam in variabila ce va fi returnata. In blocul finally inchidem aceiasi parametrii ca in metoda anterioara si la finalul metodei returnam cheia primara a elementului introdus.

Restul metodelor din aceasta clasa (exceptand primele doua despre care am vorbit inainte) sunt de tipul celei insert doar ca la unele nu mai trebuie setati parametrii pentru ca interogarea nu are nevoie.

Clasele ComandaDAO, CosDAO, ProdusDAO sunt structurate la fel ca si ClientDAO cu exceptia ca in unele clase pot exista metode de un singur fel nu de ambele ca in ClientDAO.

**Package model**



Clasele din acest pachet reprezinta corespondentul tabelelor din baza de date, un exemplu este clasa Client cu parametrii:

private String **username**;  
private String **nume**;  
private String **adresa**;  
private String **email**;  
private int **varsta**;  
private String **parola**;  
private boolean **logat**;

Acestia sunt fix aceeaiasi parametrii din tabelul definit fiind si de acelasi tip:

CREATE TABLE IF NOT EXISTS Clienti

(id int unique auto\_increment primary key,

username char(20),

nume char(20),

adresa char(100),

email char(40),

varsta int,

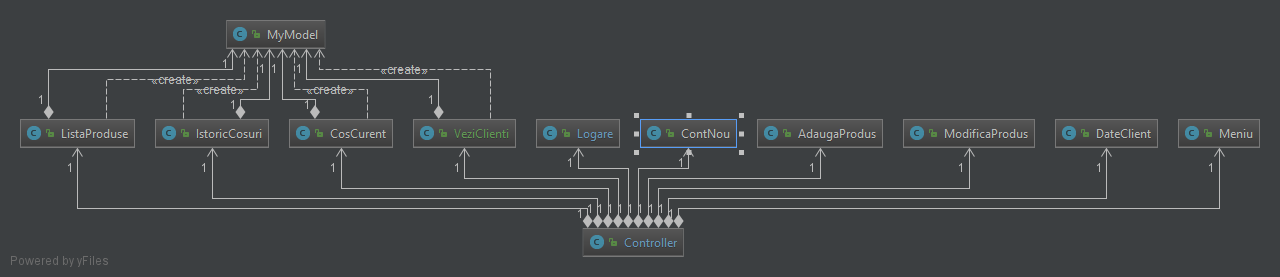
parola char(20),

logat boolean

);

Restul metodelor din clasa sunt gettere sau settere pentru fiecare variabila in parte. In general toate clasele respecta acelasi tipar ca si „Client”.

**Package Presentation**



In acest pachet avem in mare parte clase construite folosind plugin-ul grafic special din IntelIJ codul fiind generat automat de IDE in acest caz, dar au mai fost definite inca doua clase.

**Clasa Controller**

Aceasta contine variabile de tipul fiecarui frame construit cu ajutorul plugin-ului, ele fiind necesare pentru a putea aplica listnere pe butoanele din unele fereste care sa deschida alte ferestre:

private Logare **logare**;  
private Meniu **meniu**;  
private DateClient **dateClient**;  
private ListaProduse **listaProduse**;  
private CosCurent **cosCurent**;  
private ContNou **contNou**;  
private ModificaProdus **modificaProdus**;  
private AdaugaProdus **adaugaProdus**;  
private IstoricCosuri **istoricCosuri**;  
private VeziClienti **veziClienti**;

In rest pe langa constructor in care in general initializam variabilele de mai sus si apoi apelam metode din ele prin care fixam listeneri pe butoanele din diferite fereste, mai avem subclase cu care cream listeneri de pe butoane.

Listenerii in general nu fac altceva decat sa ascunda sau sa afiseze anumite ferestrea dar si sa apeleze metodele din clasele din package-ul bll despre care am discutat anterior.

**Clasa MyModel**

Aceasta clasa extinde AbstractTableModel si este folosita pentru a instantia obiecte de tip tabel, ea fiind necesara pentru ca tabelele sa fie updatate in timp real.

Variabilele din clasa sunt:

private String[] **columnNames**;  
private ArrayList<String[]> **list**;

Si ele sunt modificate prin metodele:

public void setList(ArrayList<String[]> list)  
{  
 this.**list**=list;  
}

public void setColumnNames(String[] columnNames)  
{  
 this.**columnNames**=columnNames;  
 fireTableStructureChanged();  
}

Acestea pe langa faptul a modifica continutul tabelelor apeland metode care determina schimbarea modului in care arata tabel.

In restul claselor folosite la hui pe langa metode care seteaza listeneri pe butoane cum ar fi:

public void setModificaButton(ActionListener e) {  
 **modificaButton**.addActionListener(e);  
}

mai putem remarca metoda ModelUpdate:

public void modelUpdate() {  
 String[] **columnNames** = {"Id", "Descriere", "Cantitate", "Pret"};  
  
  
 ProdusBLL **produsBLL** = new ProdusBLL();  
 ArrayList<String[]> **date**;  
 **date** = **produsBLL**.afiseazaProduse();  
  
 **model**.setColumnNames(**columnNames**);  
 **model**.setList(**date**);  
  
}

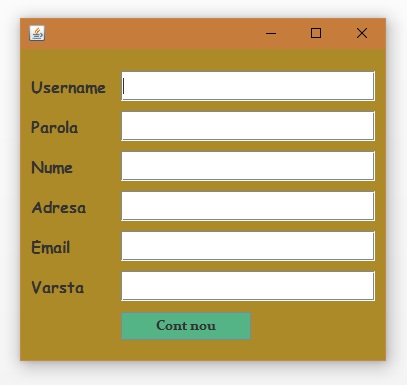
In ea mai intai initializam numele coloanelor din tabel ca mai apoi sa apelam o metoda specifica dintr-o un obiect de tipul unei clase din package-ul bll si sa sa setam continutul tabelului cu numele coloanelor si datele returnate de metoda apelata anterior. In general aceasta metoda se apeleaza de fiecare data cand vrem sa modificam date din tabel.

## IV. Interfață utilizator

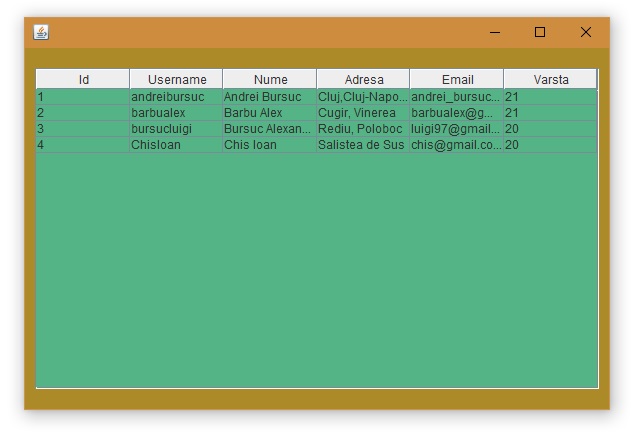
La deschiderea programului prima fereastra cu care utilizatorul va interactiona va fi cea de logare:



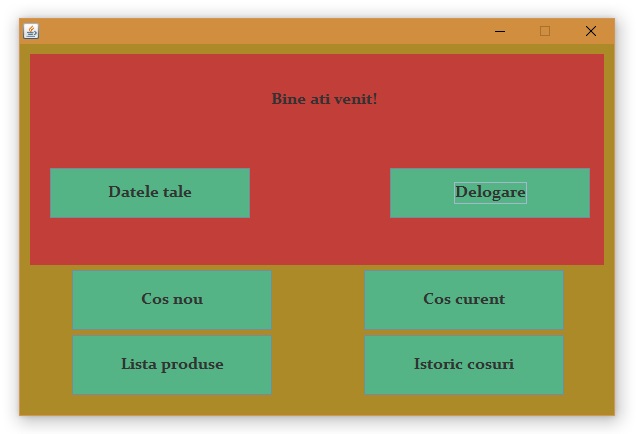
In cazul in care utilizatorul nu are un cont creat isi poate crea unul apasand butonul „Cont nou” care va afisa fereastra:



Dupa introducerea datelor clientul isi va crea un cont nou si va primi un mesaj de confirmare. Pentru a vedea daca contul a fost creat acesta poate apasa pe butonul „Clientii” care va afisa o fereastra ce contine un tabel cu clientii din baza de date.



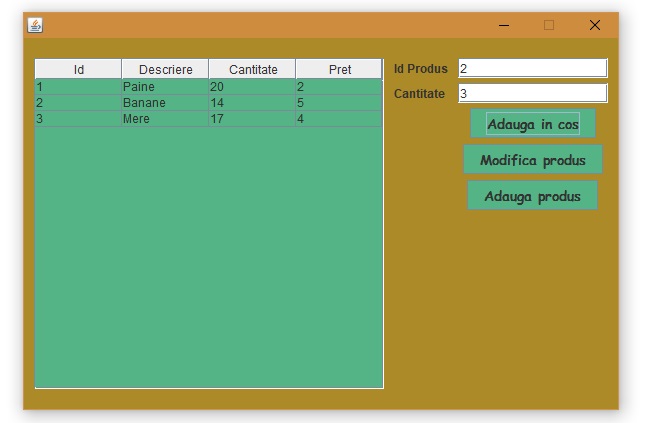
Dupa logare utilizatorul va fi intampinat de o fereastra ce contine butoane cu functiile pe care acesta le poate opera in program:



Prin apasarea butonului „Datele tale” se va deschide o fereastra asemanatoare cu cea de creare cont nou, din care va pute modifica anumite date prin apasarea butonului „Actualizare date”.

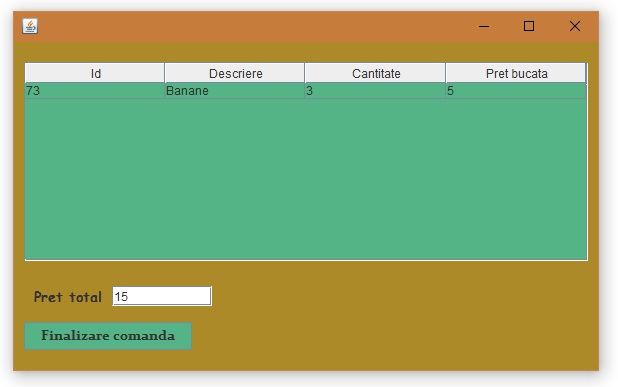
Pentru a crea un cos nou utilizatorul poate folosi butonul special creat „Cos nou” si in caz de succes va primi mesajul „A fost creat un cos nou!”, in cazul in care exista deja un cos creat el va primi mesajul de eroare „Exista deja un cos creat!”.

Dupa crearea cosului utilizatorul trebuie sa apese pe butonul „Lista produse” si o sa apara fereastra:



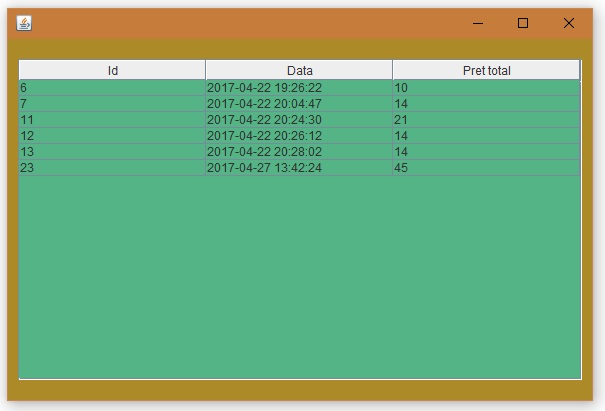
De aici clientul introduce id-ul produsului de comandat si cantitatea dorita si apasa butonul „Adauga in cos”.

La apasarea butonului se va deschide inca o fereastra cu ce contine cosul curent.



Cand clientul si-a ales toate produsele dorite poate apasa butonul „Finalizare comanda” si cosul va fi inchis iar o factura generata.

O alta functie pe care o poate folosi utilizatorul este aceea de vizualizare a cosurilor sale comandate pana acum. La apasarea butonului „Istoric cosuri” va fi afisat urmatorul tabel:



Pentru a parasi fereastra de Meniu utilizatorul va apasa pe butonul „Delogare” si se va reintoarce la fereastra de logare.

# **Concluzii si dezvoltari ulterioare**

Aceasta aplicatie poate fi folosita pentru gestionarea unui depozit de alimente in asa fel incat sa nu mai fie nevoie de salvarea stocurilor in format fizic, iar clientul sa poata comanda mai usor produsele pe care le doreste.

Dezvoltarea acestei aplicatii m-a ajutat sa inteleg mai bine modul in care conectam un program java la o baza de date, dar in acelasi timp m-a facut sa inteleg ca utilizarea unei baze de date este modul cel mai eficient de stocare a datelor dintr-un program. In dezvoltarea programului am avut ocazia sa invat mai mult despre arhitectura stratificata si despre cum putem organiza mai bine un program prin impartirea acestuia in mai multe pachete cu func-tionalitati diferite.

Posibile dezvoltari ulterioare ale programului ar fi prezenta unei descreri mai detaliate a produselor ( poze, descriere), posibilitatea de a sterge comenzi din cos, posibilitatea creerii unei liste de dorinte si aparitia unei notificari cand produsul apare pe stoc. Introducerea unei modalitati de recuperare a parolei in cazul in care utilizatorul o uita, dar si posibilitatea de a modifica datele produsului direct pe tabel fara a mai fi nevoie de casutele aditionale.

# **Bibliografie**

1. <http://docs.oracle.com/javase/>
2. <http://stackoverflow.com/>
3. Kathy Sierra, Bert Bates. SCJP Sun Java Certified Programmer for Java 6. McGraw Hill, USA 2008
4. <https://dzone.com/articles/layers-standard-enterprise>