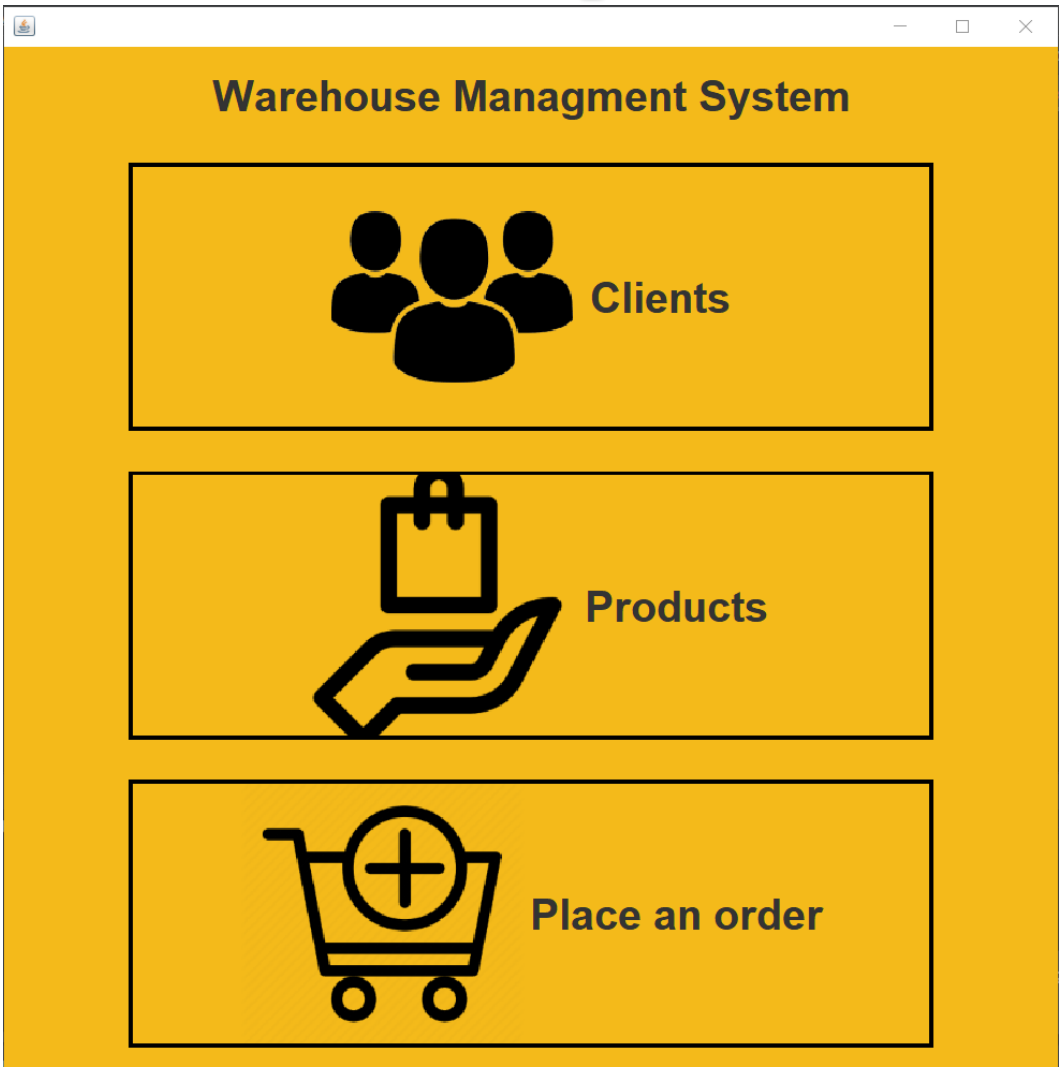
MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
SPECIALIZAREA CALCULATOARE SI TEHNOLOGIA INFORMATIEI

# Sistem de management al unui depozit




## -Documentație-

Ana-Maria Cusco

**An academic: 2020 – 2021**

# Cuprins

# Obiectivul temei

Obiectivul temei este dezvoltarea unei aplicații care sa permită gestiunea facila a clienților, a produselor si a comenzilor unui depozit.

Un sistem de tip WMS reprezinta o solutie software pentru imbunatatirea managementului depozitelor, fiind o componenta esentiala in cazul intreprinderilor deoarece automatizează activitățile de zi cu zi care au loc intr-un depozit prin utilizarea unui sistem centralizat cu ajutorul caruia diferite operatiuni cum ar fi managementul clientilor, al produselor sau al comenzilor sunt optimizate si eficientizate.

Operatii precum adaugare, modificare date, stergere sau vizualizare date sunt realizate in timp real si extrem de rapide.

Sistemul de stocare WMS (Warehouse Management System) – de ce merită?

Sistemul de depozitare oferă multe beneficii semnificative. Datorită funcționării sale, activitatea depozitului este optimizată, iar echipa de angajați este utilizată la fel de eficient. Gestiunea depozitelor nu este ușoară și generează multe erori potențiale pe care sistemul de depozitare le poate elimina. Lucrătorii din depozite și serviciile logistice au nevoie disperată de planificarea muncii îmbunătățită, iar un sistem de genul acesta le poate oferi acest lucru.

Un avantaj considerabil este scurtarea timpului de plasare a comenzilor deoarece controlul și verificarea stocurilor si a clientilor sunt mult imbunatatite prin vizualizarea datelor in timp real, si cautarea eficienta a produselor. Timpul nu poate fi supraestimat in cazul operațiunilor logistice, este întotdeauna o reducere și o utilizare maximă, iar un sistem eficient de stocare este capabil să asigure acest lucru.

# Analiza problemei, modelare scenarii, cazuri de utilizare

Cerința aplicației este de a dezvolta o aplicație de gestiune a unui depozit pentru a procesa comenzile clienților si care utilizează o baza de date relaționala pentru stocarea produselor, clienților si a comenzilor.

Provocarea acestei aplicații  o reprezintă conectarea la o baza de date MySQL care stochează datele necesare gestionarii depozitului si oferă utilizatorului posibilitatea sa vadă si sa acceseze aceste date prin intermediul interfeței  grafice ale aplicației Java.

Accesarea unei baze de date dintr-o aplicatie Java se realizeaza prin intermediul unui program de comanda (driver) specific unui anumit sistem de gestiune a bazelor de date. Un driver intermediază legătura dintre aplicatii si baze de date.

Java DataBase Connectivity JDBC reprezinta un API care permite lucrul cu baze de date relationale. Prin intermediul JDBC sunt transmise comenzi SQL la un server de baze de date. Folosind JDBC, nu este necesara dezvoltarea mai multor aplicatii pentru a accesa servere de baze de date care utilizeaza sisteme diferite de gestiune a bazelor de date (Oracle, MySQL, Sybase). Este suficienta o singura aplicatie, care sa utilizeze API-ul JDBC, pentru a transmite comenzi SQL la serverul de baze de date dorit. In felul acesta este asigurata portabilitatea aplicatiei.

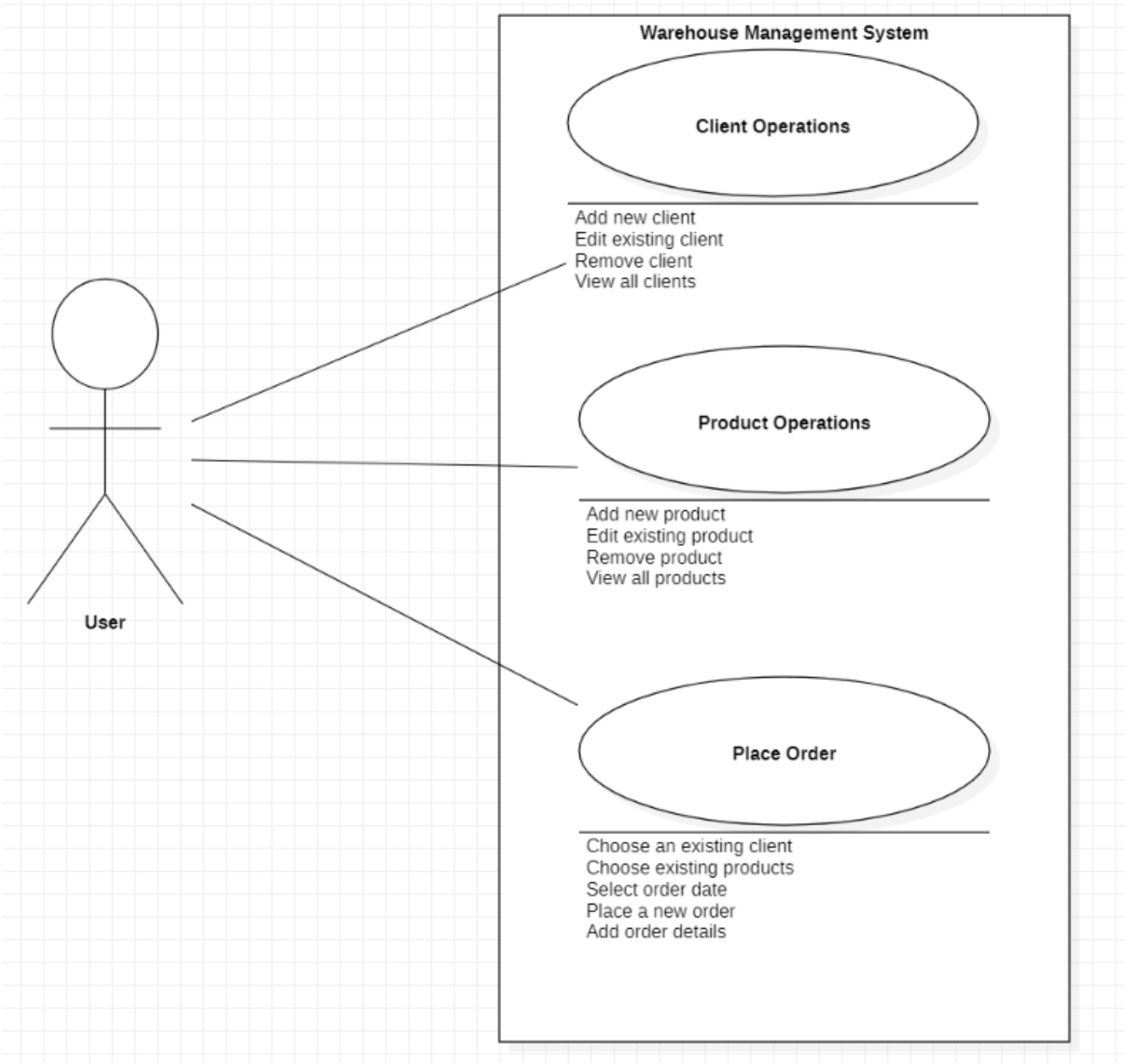
Driver-ul JDBC ofera acces uniform la bazele de date de tip relational. JDBC include clase si interfete, scrise in Java, care furnizeaza o interfata SQL standard pentru proiectantii de aplicatii cu baze de date. Clasele si interfetele necesare in API-ul JDBC sunt disponibile prin intermediul pachetului java.sql.

Accesarea unei baze de date din Java, via JDBC, presupune realizarea urmatorilor pasi:

• stabilirea unei conexiuni la serverul de baze de date si selectarea unei baze de date;

• transmiterea si rularea unor secvente SQL;

• preluarea si prelucrarea rezultatelor obținute.




*Use Case Diagram*

Cu ajutorul acestei aplicatii utilizatorul poate interactiona cu aplicatia intr-un mod simplu si eficient, fara a avea in prealabil cunostinte de programare sau de lucru cu baze de date relationale. Modul de stocare al datelor si organizarea acestora in tabele prin intermediul bazei de date este transparent pentru utilizator, acesta putand vizualiza si modifica aceste date prin intermediul interfetei grafice.

Utilizatorul poate sa efectueze urmatoarele operatii:

- asupra clientilor: adaugarea unui nou client, editarea datelor unui client existent, stergerea unui client si vizualizarea unei liste a tuturor clientilor.

- asupra produselor: adaugarea unui nou produs, editarea datelor unui produs, stergerea unui produs, vizualizarea unei liste a tuturor produselor.
- sa plaseze o comanda noua cu detaliile aferente comenzii (datele clientului, data si ora comenzii, produsele comandate)

Dupa plasarea comenzii, trebuie sa se genereze factura clientului care contine detaliile comenzii sub forma unui fisier text sau pdf.

## Organizare structurata(tabelar) a cerintelor intr-o baza de date:

Baza de date trebuie sa stocheze următoarele informatii:

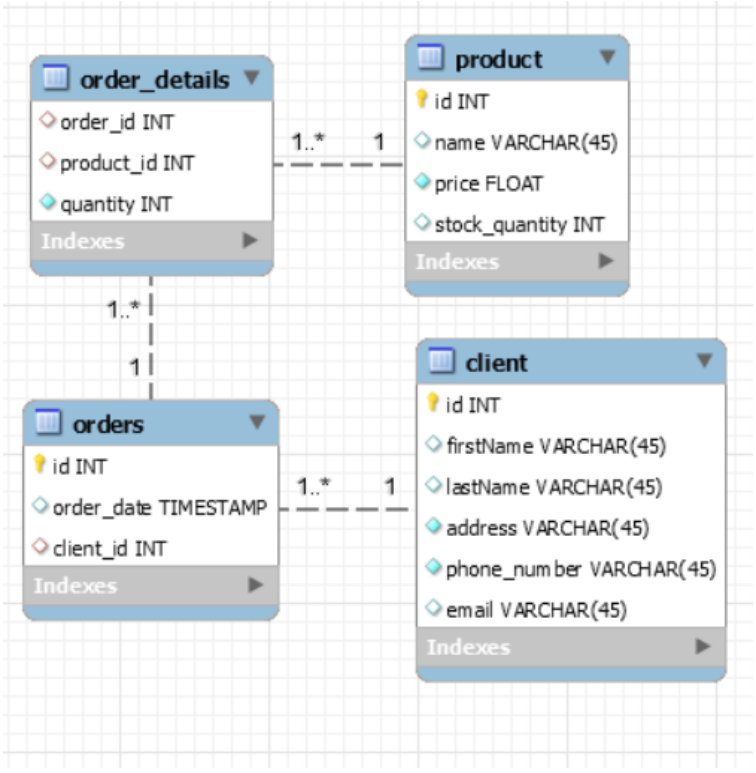
-informatii despre un client: id, nume, prenume, adresa, nr. de telefon, email

-informații despre un produs: id, nume, pret, cantitate in stoc

-informații despre o comanda: id, data plasării comenzi, id-ul clientului care plasează comanda

-detalii despre o comanda: id-urile produselor din comanda respectiva si cantitățile comandate din fiecare produs

Modelul relational ce sta la baza acestei aplicații este prezentat in figura următoare si este stocat intr-o baza de date MySQL:



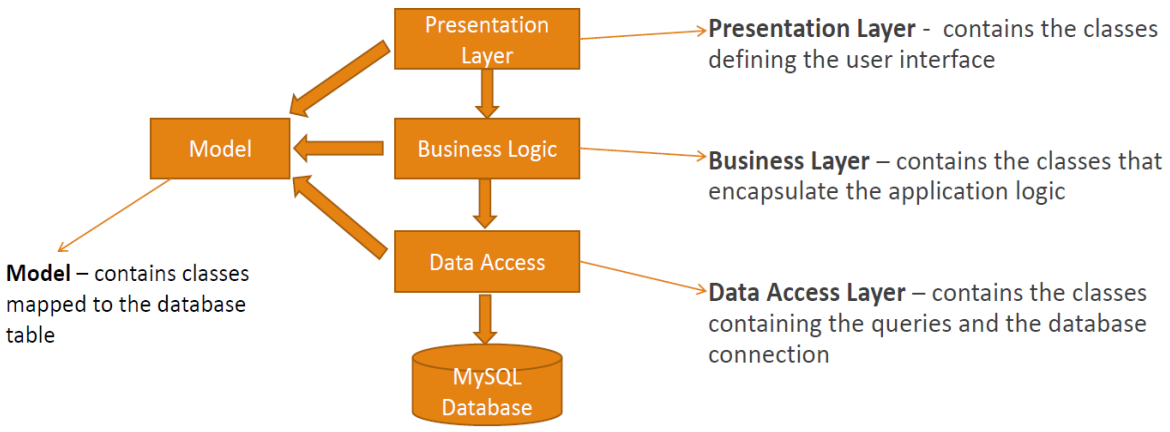
# Proiectare

## Decizii de proiectare

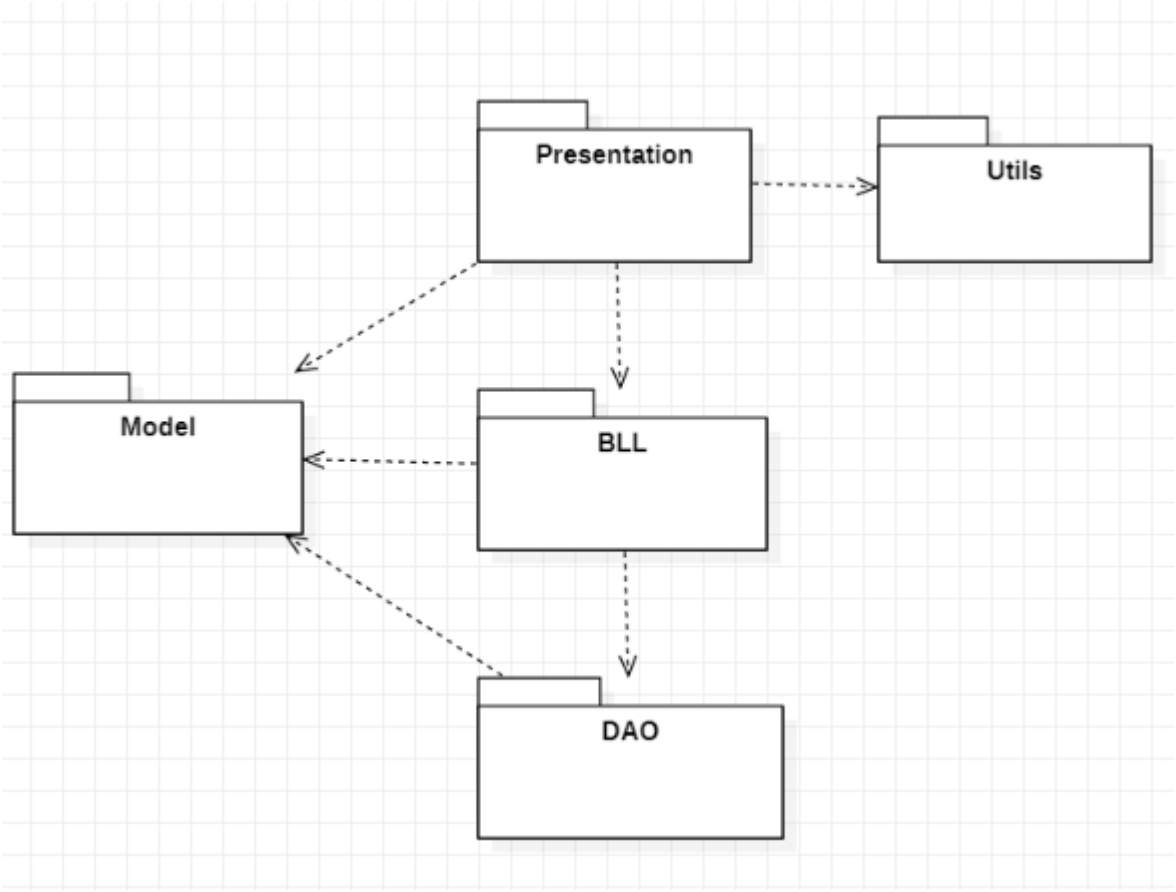
In construirea acestei aplicatii, si dupa cum specificau cerintele am ales sa folosesc o arhitectura pe mai multe niveluri (eng. multitier architecture).

Arhitectura de aplicații N -tier oferă un model prin care dezvoltatorii pot crea aplicații flexibile și reutilizabile. Prin separarea unei aplicații în niveluri, dezvoltatorii dobândesc opțiunea de a modifica sau adăuga un anumit strat, în loc să refacă întreaga aplicație. O arhitectură logică multistratificată pentru un sistem informațional cu un design orientat obiect este compusă de obicei din mai multe straturi dupa cum urmeaza:

- Stratul de prezentare (cunoscut și ca stratul UI, stratul de vizualizare, nivelul de prezentare în arhitectura pe mai multe niveluri)
- Stratul de aplicație (cunoscut și ca strat de serviciu sau stratul de control GRASP )
- Stratul de afaceri (alias strat de logică de afaceri (BLL), strat de domeniu)
- Stratul de acces la date (cunoscut și ca strat de persistență , jurnalizare, rețea și alte servicii care sunt necesare pentru a sprijini un anumit strat de afaceri)



**Diagrame UML**

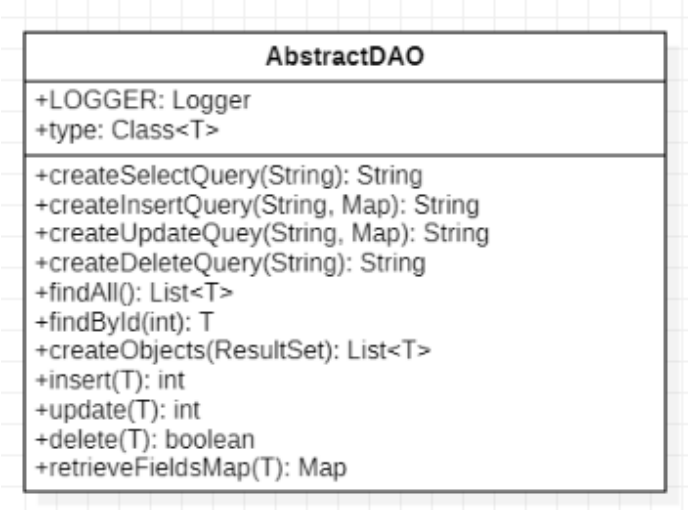



- Pachetul **Model** contine urmatoarele clase: Client, Product, Orders si OrderDetails si mapeaza tabelele din baza de date in clase Java.

Fiecare dintre aceste clase are in baza de date un tabel corespondent, iar variabilele sale de instanta se portrivesc cu numele coloanelor din acel tabel.

- Pachetul **DAO** contine urmatoarele clase: AbstractDAO, ProductDAO, OrdersDAO si OrderDetailsDAO si reprezinta nivelul in care se realizeaza operatiile cu baza de date (interogarile).

  Clasa **AbstractDAO**




Contine implementari generice pentru operatiile CRUD (insert, find, update si delete) folosind tehnici de inspectare a claselor- **Java Reflection**.
Prin intermediul acestei clase am furnizat principalele implementari de care vom avea nevoie in aceasta aplicatie pentru operatiile efectuate asupupra clientilor, produselor si comenzilor.

Dupa cum am specificat si inainte, accesarea unei baze de date se realizează printr-un driver.

Astfel de operatii care lucreaza cu baza de date, urmaresc pasii de mai jos:

- **Stabilire conexiune server**: o conexiune la un server de baze de date reprezinta un canal de comunicatii prin care sunt transmise cereri SQL si sunt returnate raspunsuri corespunzatoare. Stabilirea unei conexiuni dintr-o aplicatie Java presupune inregistrarea (incarcarea) unui driver si realizarea conexiunii propriu-zise prin intermediul clasei DriverManager din pachetul java.sql.
  Utilizand clasa DriverManager, stabilirea efectiva a unei conexiuni la un server de baze de date necesita specificarea unui URL prin intermediul unei cereri care are urmatoarea forma:

```
Connection conn = DriverManager.getConnection(url, nume_utilizator, parola);
```

- **Rulare comenzi SQL:** Dupa stabilirea unei conexiuni la serverul de baze de date si selectarea unei baze de date active este necesara crearea unei instante de tip PreparedStatement prin metoda prepareStatement() a clasei Connection. Instanta de tip Statement permite manipularea unor comenzi SQL.

```
Connection connection = DriverManager.getConnection(url);

PreparedStatement stmt = connection.prepareStatement();
```

Un obiect de tip Statement este un container care permite transmiterea si rularea comenzilor SQL, si obtinerea rezultatelor corespunzatoare prin intermediul conexiunii asociate. Pe langa interfata Statement, mai pot fi utilizate doua subinterfete ale acesteia: PreparedStatement si CallableStatement.

Rularea unei comenzi SQL poate fi realizata prin trei metode:

- **executeQuery()**: este utilizata pentru a rula comenzi SQL care returneaza un obiect de tip ResultSet;

```
public Product findProductsByName(String name){

.............................some code here.....................................

String query="SELECT * FROM product WHERE name LIKE '" +name+ "%'";'

PreparedStatement statement = connection.prepareStatement(query);

ResultSet resultSet = statement.executeQuery();

.............................another code here..................................

}
```

- **executeUpdate()**: este utilizata pentru a rula comenzi SQL care permit manipularea datelor (INSERT, UPDATE, DELETE – returneaza numarul de inregistrari afectate de rularea comenzii SQL) sau modificarea structurii unui tabel din baza de date (CREATE, ALTER, DROP – returneaza valoarea 0);

```
public boolean update(T t) {

.............................some code here.....................................

PreparedStatement statement = connection.prepareStatement(query);

int changedRows = statement.executeUpdate(query);

.............................another code here..................................

}
```

- **execute()**: poate fi utilizata pentru a rula orice tip de comanda SQL.

- **Manipulare si prelucrare rezultate**: De cele mai multe ori rularea unei comenzi SQL pe o baza de date are ca si rezultat un set de date care apar sub forma tabelara. Pentru a obtine date dintr-o baza de date pot fi rulate comenzi SQL prin intermediul metodei executeQuery(). Aceasta metoda returneaza informatia sub forma unor linii de date (inregistrari), care pot fi accesate prin intermediul unei instante de tip ResultSet.

```
Connection connection = DriverManager.getConnection(url);
Statement stmt = connection.createStatement();
String sql = "SELECT id, firstName, lastName, address FROM client LIMIT 5";
ResultSet rs = stmt.executeQuery(sql);
```

In cazul in care dorim sa accesam lista primilor 5 clienti din baza noastra de date este suficienta executarea unei comenzi SQL de tip SELECT, pentru tabelul client.

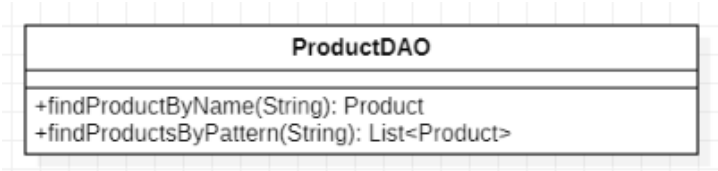
Inregistrarile dintr-un obiect de tip ResultSet pot fi parcurse cu ajutorul metodei next(). De asemenea, accesarea valorilor corespunzatoare anumitor coloane poate fi realizata prin metode de tipul get<Type>() (getString(), getInt()). Coloanele dintr-un tabel pot fi referite prin intermediul numelui sau prin intermediul pozitiei coloanei in interiorul tabelului (prima coloana din tabel are pozitia 1).

```
while(rs.next()) {
    int id = rs.getInt("id");
    String firstName = rs.getString("firstName");
    String lastName = rs.getString("lastName");
}
```

Toate tabelele unei baze de date detin meta-date care descriu denumirile si tipurile de date specifice fiecarei coloane. In acest fel poate fi utilizata clasa ResultSetMetadata pentru a obtine numarul de coloane dintr-un tabel sau denumirile coloanelor.

```
ResultSetMetaData meta = rs.getMetaData();
for(int i = 0; i < meta.getColumnCount(); i++)
    System.out.print(meta.getColumnName(i + 1) + "\t");
```

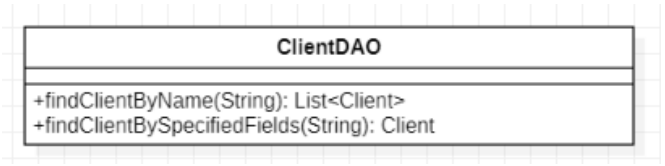
Clasa **ProductDAO**




– contine metode specifice de care am avut nevoie in realizarea aplicatiei precum:**findProductByName(String)** – metoda folosita la adaugarea unui nou produs, pentru a verifica daca nu cumva produsul pe care noi dorim sa-l inseram exista deja in baza de date.

Metoda **findProductsByPattern (String)** este folosita la cautarea unui nou produs pentru editarea sau stergerea acestuia. Aceasta returneaza o lista a produselor care incep cu un anumit sablon (`"SELECT * FROM product WHERE name LIKE '" + pattern + "%'");` Astfel utilizatorul poate cauta produsele atat dupa nume, cat si dupa ID, lucru ce eficientizeaza extrem de mult munca.

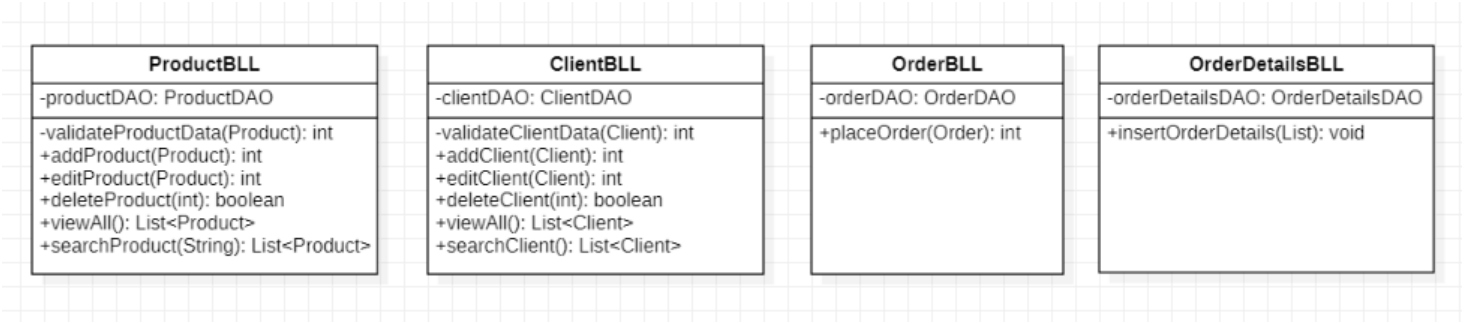
Clasa **ClientDAO**




**–** contine la randul sau metode specifice cum ar fi: **findClientsBySpecifiedFields(Client) –** o metoda ajutatoare pentru operatia de inserare pentru a verifica daca clientul care urmeaza sa fie inserat nu exista deja, si metoda **findClientByName(String) –** metoda care ne ajuta sa facem cautarea unui client dupa nume. Aceasta cautare se realizeaza in cazul in care se doreste sa se faca o operatie de actualizare a datelor clientului sau o operatie de stergere a unui client existent si se mai poate face si dupa un ID specific.

Clasele **OrderDetailsDAO** si **OrderDAO** extind clasa AbstractDAO si prin urmare mostenesc metodele CRUD pentru a lucra cu baza de date.

- Pachetul **BLL** contine urmatoarele clase: ClientBLL, ProductBLL, OrdersBLL si OrderDetailsBLL si la nivelul acestui pachet sunt implementate cerintele functionale ale aplicatiei – operatii CRUD pe clienti, produse respectiv plasarea unei noi comenzi.




Aceste clase folosesc implementarile din stratul de acces la date, si in plus contin clase de tip **<Validator>** pentru a valida informatia care este transmisa din interfata grafica inspre baza de date, in acest fel eliminandu-se orice fel de erori sau inconsistente la nivelul datelor ce sunt stocate.

- Pachetul **Presentation** contine clasele de GUI si Controller si imbraca intr-un mod frumos modelul de date care sta la baza aplicatiei, facilitand interactiunea utilizatorului cu acesta prin ferestre dedicate fiecare operatii in parte.

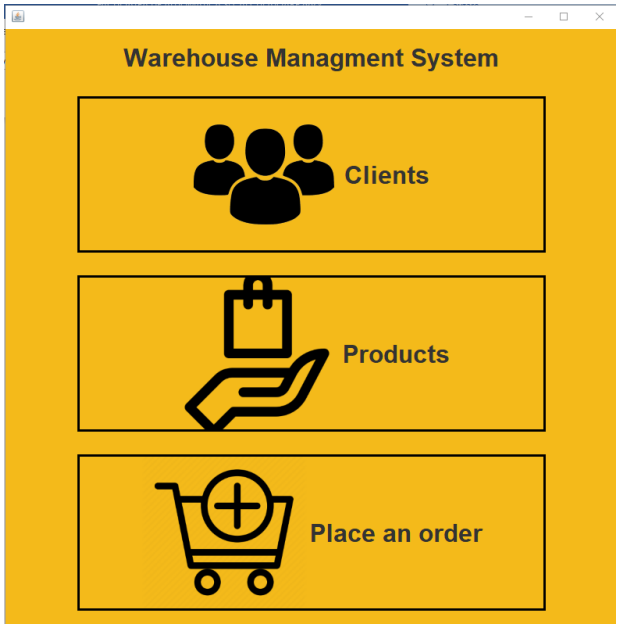
❖ Pachetul **Utils** are un rol foarte important si contine urmatoarele clase: ObjectTableModel, DisplayableObjectTableModel si FileWriteClass.

Pentru a afisa informatiile dintr-un tabel din baza de date in interfata grafica este nevoie de un model generic de tabel de obiecte, caruia ii este furnizata o lista de obiecte si care folosind tehnici de inspectare a claselor precum Java Reflection converteste atributele tabelului din baza de date (proprietatile obiectelor) intr-un header de tabel care poate fi afisat intr-o fereastra grafica, si populeaza acest tabel cu valorile specifice ale obiectelor din lista primita. Acest model este folosit apoi de un JTable in afisarea datelor in GUI.

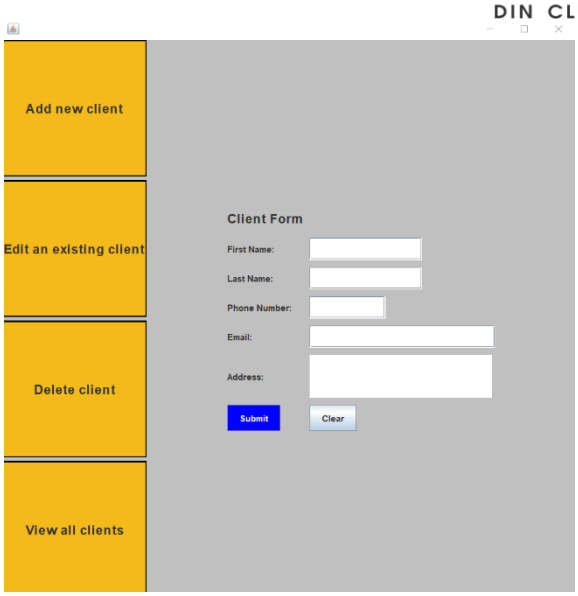
```
ObjectTableModel<Product> tableModel = new DisplayableObjectTableModel<>(Product.class);
tableModel.setObjectRows(productsList);
JTable table = new JTable(tableModel);
```

Clasa **FileWriteClass** este contine metoda **printBill(String orderDetails)** si este folosita la generarea unui fisier text care contine  pentru fiecare comanda nou creata si contine detaliile comenzii (informatiile clientului, produsele comandate, totalul de plata, data).

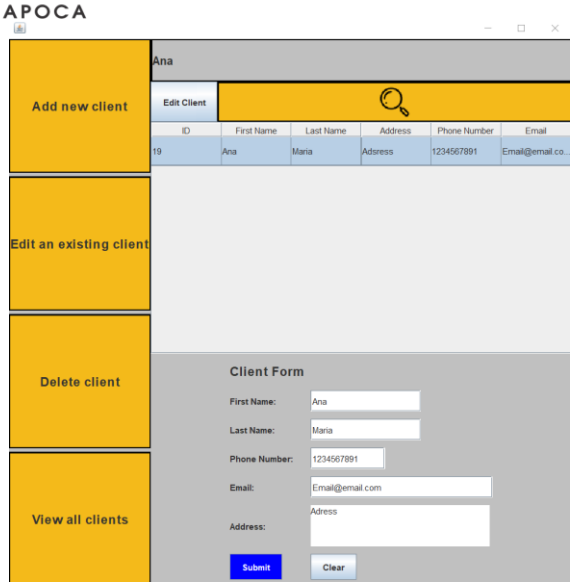
# Interfata Utilizator



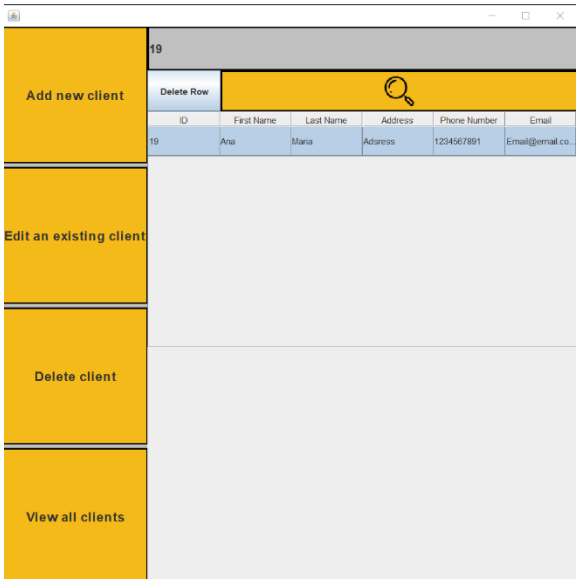

➢ Pagina de start a aplicatiei




Adaugare unui nou client




Editarea unui client existent

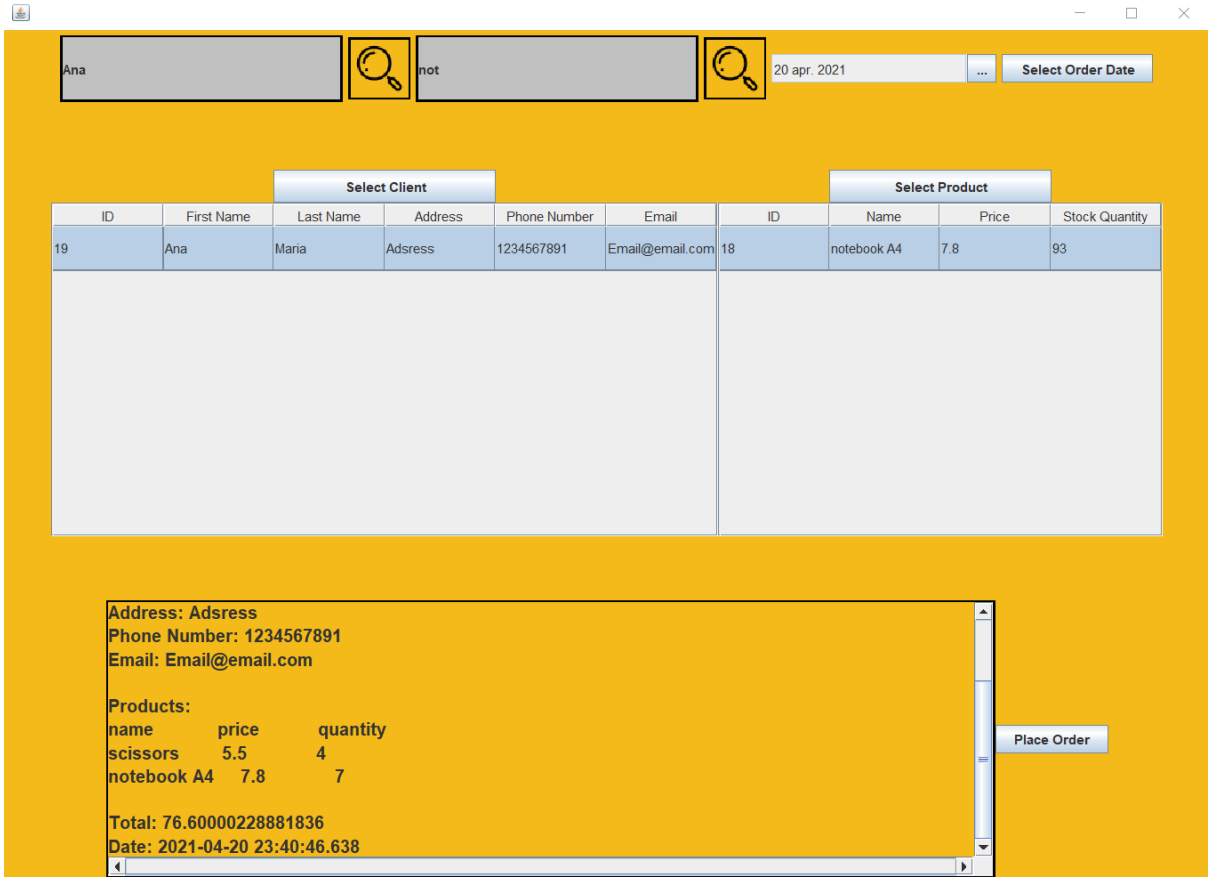



Stergerea unui client existent


Vizualizarea tuturor clientilor

Similar este realizata si fereastra pentru operatiile cu produsele.

- Plasarea unei noi comenzi



# Implementare

Proiectul “Warehouse Management System” este o aplicatie desktop realizata in limbajul de programare Java folosind mediul de dezvoltare IntelliJ IDEA. Principalul scop al acesteia este sa gestioneze clientii, produsele si comenzile unui depozit intr-o maniera cat mai usoara si mai prietenoasa pentru utilizator.

- Implementarea clasei AbsreactDAO, metodele **createDeleteQuey(String field)** respectiv **delete(int id)**

```
private String createDeleteQuery(String field) {
    StringBuilder sb = new StringBuilder();
    sb.append("DELETE FROM ");
    sb.append(type.getSimpleName());
    sb.append(" WHERE ").append(field).append(" =?");
    return sb.toString();
}
```

```
public boolean delete(int id) {
    Connection connection = null;
    PreparedStatement statement = null;
    int deletedRows;
    String query = createDeleteQuery( field: "id");
    System.out.println("Delete Query:"+query);
    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.setInt( parameterIndex: 1, id);
        deletedRows = statement.executeUpdate();

        return deletedRows>0;
    } catch (SQLException e) {
        LOGGER.log(Level.WARNING,  msg: type.getName() + "DAO:delete " + e.getMessage());
    } finally {

        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }
    return false;
}
```

In mod similar sunt implementate opetatiile de insert, find, si update.

- Verificarea corectitudinii datelor folosind clasele **<Validator>** de la nivelul stratului de afaceri (bll) , clasa ProductBLL:

```
private int validateProductData(Product product){
    ProductValidations productValidations=new ProductValidations();
    if(!productValidations.isValidName(product.getName())){
        return 1;
    }
    if(!productValidations.isValidPrice(product.getPrice())){
        return 2;
    }

    if(!productValidations.isValidStockQuantity(product.getStock_quantity())){
        return 3;
    }

    if(productDAO.findProductByName(product.getName())!=null){ //if product already exists
        return -1;
    }
    return 0;
}
```

- Metoda **deleteProduct(int id)** de la acelasi nivel:

```
public void deleteProduct(int id){
    productDAO.delete(id);
}
```

# Rezultate

Rezultatele unei implementari corecte se pot verifica prin accesarea bazei de date si verificarea corectitudinii datelor de aici. Tot in acest mod, s-a observat cum operatiile CRUD au fost implementate corect, datele ajungand in timp real la nivelul modelului de stocare si tot in timp real fiind preluate si afisate prin intermediul interfetei grafice a aplicatiei. Validarile se fac inca de din layerul de Bussiness ceea ce impiedica orice fel de erori sa ajunga la nivelul urmator, datele fiind in acest fel transmise corect in cazul operatiilor de adaugare sau modificare.

# Concluzii

Din aceasta tema am invatat rolul important pe care il au bazele de date, si cum imbinand un software cu o baza de date imbunatateste extrem de mult performanta unui serviciu, flexibilitatea, scalabilitatea si procesul de luare a deciziilor deoarece integrarea unei aplicatii care lucreaza cu baze de date simplifica gestionarea datelor, automatizeaza procesele manuale costisitoare si consumatoare de timp, eliberandu-le timp utilizatorilor si permitandu-le sa stocheze datele intr-o forma structurala si apoi sa le acceseze prin intermediul unei interfete grafice extrem de simplu si de eficient.

# Bibliografie

[1] https://en.wikipedia.org/wiki/Multitier_architecture

[2] https://uncoded.ro/lucrul-cu-baze-de-date-in-java/

[3] FUNDAMENTAL PROGRAMMING TECHNIQUES (ASSIGNMENT 3 SUPPORT PRESENTATION)

- Connect to MySql from a Java application
  https://www.baeldung.com/java-jdbc
  http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/
- Layered architectures
  https://dzone.com/articles/layers-standard-enterprise
- Reflection in Java
  http://tutorials.jenkov.com/java-reflection/index.html
- Creating PDF files in Java
  https://www.baeldung.com/java-pdf-creation
- JAVADOC
  https://www.baeldung.com/javadoc
- SQL dump file generation
  https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html

Anexa-facturarea comenzii

```
ORDER DETAILS

Client
Name: Ana Maria
Address: Adsress
Phone Number: 1234567891
Email: Email@email.com

Products:
name            price           quantity
scissors        5.5             4
notebook A4     7.8             7

Total: 76.60
Date: 2021-04-20 23:40:46.638
```

ClientBill.txt