DOCUMENTAȚIE

TEMA 3 – *Order management*

NUME STUDENT: *Bustan-Jeflea Ștefania*
GRUPA: *30223*

Cuprins

1. Obiectivul temei

Scopul proiectului a fost să dezvoltăm în Java un program care să simuleze gestionarea comenzilor clienților către un depozit. Programul trebuia să includă cel puțin trei tabele: Client, Product și Order. Programul trebuia să citească dintr-un fișier și să efectueze operațiile specificate în fișierul de

intrare (inserare, ștergere, actualizare). Programul trebuia să aibă o conexiune către o bază de date SQL cu care să interacționeze.

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

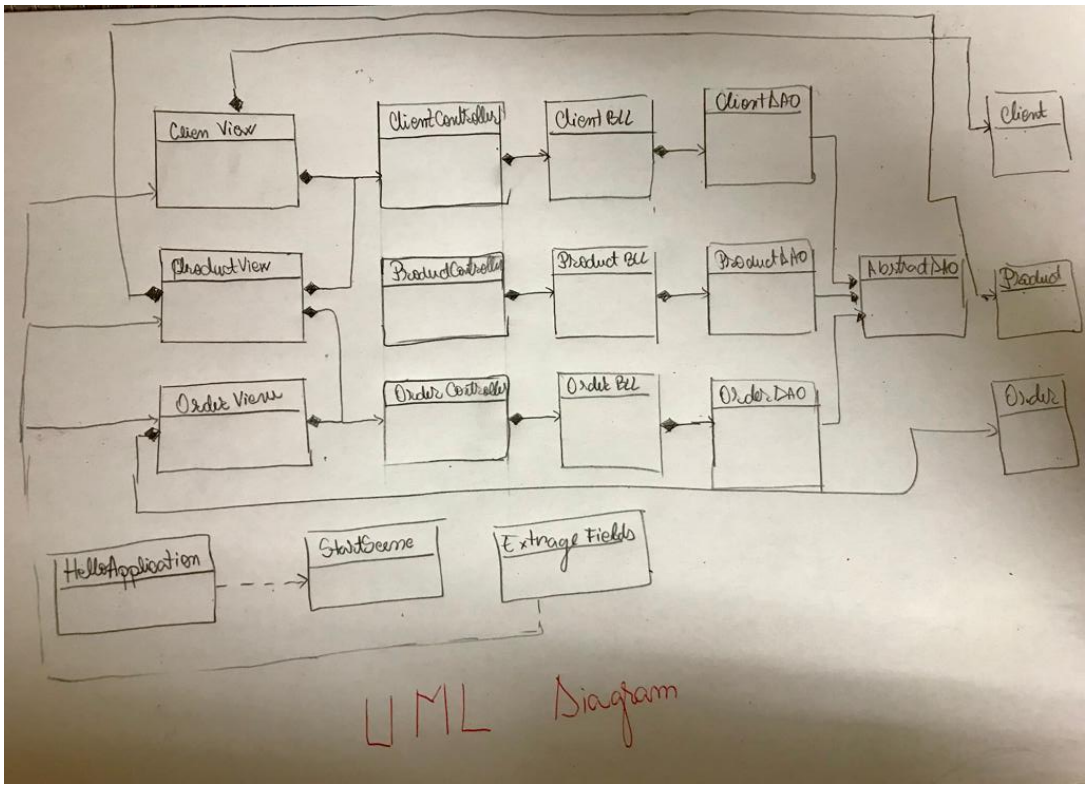
Proiectarea problemei a fost realizată uramtorul mod: fiecare tabel este asociat cu o clasă care are ca atribute coloanele respectivei tabele. Modelarea a fost realizată pe straturi, conform cerințelor, astfel încât există 4 niveluri de execuție, fiecare reprezentat printr-un pachet:

- Model (conține clasele: Clients, Products, Order) - acest nivel conține clase asociate cu tabelele din baza de date.
- Dao (conține clasele: ClientsDAO, ProductsDAO, OrderDAO) - în acest nivel sunt implementate interogările (query-urile) pentru fiecare acțiune din fiecare clasă.
- BLL (conține clasele: ClientsBLL, ProductsBLL, OrderBLL) - acest nivel este responsabil de definirea concretă a acțiunilor pentru fiecare clasă. Aici sunt apelate metodele din Dao
- Presentation - acest nivel conține clasele Controller (care se ocupă de extragerea datelor de intrare din interfata și apelarea metodelor corespunzătoare din BLL) și HelloApplication (care este punctul de intrare al proiectului).

Această arhitectură în straturi asigură o separare clară a responsabilităților și o organizare eficientă a codului în funcție de funcționalitatea acestuia.

3. Proiectare

3.1 Diagrama UML



3.2 Clase si pachete

Proiectarea claselor a fost facuta dupa modelul pe nivele prezentat. Astfel, fiecare pachet are un rol prestabilit de dinainte si in proiectare acestora, nu s-a sarit peste nivele. Fiecare nivel face apeluri doar catre nivelul de “sub” el.

- Pachetul *Connection*:

1. Clasa **ConnectionFactory** este clasa in care se realizeaza conexiunea cu baza de date. Are ca argumente: *Logger* LOGGER, *String* DRIVER, *String* DBURL, *String* USER, *String* PASS. Ca metode se gasesc constructor, createConnection(), metode de get pentru conexiune, metoda de inchidere a conexiunii, close().

- Pachetul *Model*:

1. **Clients** simuleaza un client, si are ca atribute *int* id care reprezinta id-ul lui unic in tabel, *String* nume, care este format din nume si prenume separate prin spatiu si reprezinta numele acestuia si *String* adresa, care este adresa de domiciliu a clientului. Ca metode, se gasesc constructori (cu si fara id), metode de get si set pentru atribute.
2. **Products** simuleaza un produs vandut de un magazine, care are ca atribute *int* idproducts, care este id-ul produsului in tabel si este unic, *String* nume care este numele produsului, *int* cantitate reprezentand stocul disponibil pentru acest produs, *double* pret, care reprezinta pretul pentru o bucata aferent produsului. Ca metode, se gasesc constructori (cu si fara id), metode de get si set pentru atribute.
3. **Order** simuleaza o comanda a unui client pentru un singur produs. Are ca atribute *int* idorder, care reprezinta id-ul unic al comenzii in tabel, *String* numeclient, care reprezinta numele clientului care plaseaza comanda, *String* numeproduct care reprezinta numele produsului comandat de client si *int* cantitate care reprezinta cantitatea comandata, cantitate cu care urmeaza sa se scada stocul curent pentru acel produs. Ca metode, se gasesc constructori, metode de get si set pentru atribute. Ca metode, se gasesc constructori (cu si fara id), metode de get si set pentru atribute.

- Pachetul *DAO*:

1. Clasa **ClientsDAO** este clasa in care se face legatura cu baza de date prin apelarea metodei din ConnecionFactory si se implementeaza query-urile pentru clasa/tabelul Clients.
2. Clasa **ProductsDAO** este clasa in care se face legatura cu baza de date prin apelarea metodei din ConnecionFactory si se implementeaza query-urile pentru clasa/tabelul Product.
3. Clasa **OrderDAO** este clasa in care se face legatura cu baza de date prin apelarea metodei din ConnecionFactory si se implementeaza query-urile pentru clasa/tabelul Clients.
4. Clasa **AbstractDAO** este clasa in care se definesc operațiile pe tipul generic specificat <T> (! T poate fi orice clasă de model Java, adică mapat la baza de date și are același nume ca tabelul și aceleași variabile de instanță și tipuri de date ca câmpurile tabelului.

- Pachetul *BLL*:

1. Clasa **ClientsBLL** : reprezinta clasa service pentru obiectele de tip Client. Aici se realizeaza legatura intre *Controlller* si *DAO*.

2. Clasa **ProductsBLL** : reprezinta clasa service pentru obiectele de tip Product. Aici se realizeaza legatura intre *Controlller* si *DAO*.

3. Clasa **OrderBLL** : reprezinta clasa service pentru obiectele de tip Orders. Aici se realizeaza legatura intre *Controlller* si *DAO*.

- Pachetul *presentation*:

1. Clasa **HelloApplication** reprezinta main-ul proiectului
2. Clasa **ViewClient** este reliazata interfata pentru clienti
3. Clasa **ViewProduct** este reliazata interfata pentru produse
4. Clasa **ViewOrder** este reliazata interfata pentru comenzi

5. Clasa **ClientController**: este prima clasa cu care se incepe realizarea legaturii intre interfata si baza de date. Aceasta contine un obiect de tip ClientBLL cu scopul unui Controller.
6. Clasa **ProductController**: este prima clasa cu care se incepe realizarea legaturii intre interfata si baza de date. Aceasta contine un obiect de tip ProductBLL cu scopul unui Controller.
7. Clasa **OrdersController**: este prima clasa cu care se incepe realizarea legaturii intre interfata si baza de date. Aceasta contine un obiect de tip OrderBLL cu scopul unui Controller.

4. Implementare

- **Pachetul *Connection***:

Clasa ConnectionFactory implementează următoarele metode:

- createConnection() - realizează conexiunea la baza de date
- getConnection() - returnează conexiunea curentă cu baza de date
- close(Connection) - întrerupe conexiunea cu baza de date
- close(Statement) - închide un statement primit ca parametru.

Mai jos se poate observa cum se realizeaza conexiunea cu baza de date:

```
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;
}
```

- **Pachetul *Model***

Clasa Clients:

- toString() - afișează într-un format prietenos pentru utilizator informațiile despre un client. Mai jos este implementata metoda pentru realizarea afisariii unui client.:

```
@Override
public String toString() {
    return "Client   " + id +
            ". " + nume  +
            "
" + adresa;
}
```

Clasa Product:

- toString() - afișează într-un format prietenos pentru utilizator informațiile despre un produs.

Mai jos este implementata metoda pentru realizarea afisariii unui produs:

```
@Override
public String toString() {
    return "Product   " +
             idproduct +
            ".   " + nume +
```

```
                "                          " + cantitate +
                " Stock                                    " + pret +
                " RON";
    }
```

Clasa Order:

- toString() - afișează într-un format prietenos pentru utilizator informațiile despre o comandă

Mai jos este implementata metoda pentru realizarea afisariii unei comenzi:

```
➢   @Override
    public String toString() {
        return "Order " + idorders +
                ".CLIENT: " + numeclient +
                "                PRODUCT: " + numeprodus+
                "                    STOCK: " + cantitate;
    }
```

- **Pachetul *DAO*:**

Clasa ***AbstractDAO***:

Aceasta reprezinta clasa in care s-a realizat reflection. *Reflection* în Java este o caracteristică a limbajului care îi permite unui program să inspecteze, să manipuleze și să genereze dinamic obiecte Java, metode, câmpuri și alte elemente la timpul de execuție. Aceasta oferă o modalitate de a examina și modifica comportamentul unei clase sau a unui obiect în timpul execuției, fără a avea acces direct la codul sursă. Reflection se foloseste intr-o varietatea de scenarii. Cateva dintre acestea sunt:

1. Inspectarea și manipularea structurii claselor: Reflection permite obținerea informațiilor despre clase, cum ar fi numele, metodele, câmpurile și anotările asociate. Acest lucru este util în cazul în care doriți să inspectați și să interacționați cu clasele într-un mod dinamic.
2. Instantierea dinamică a claselor: Reflection permite crearea de obiecte ale unei clase la timpul de execuție, chiar dacă numele clasei este cunoscut doar în momentul rulării programului. Acest lucru este util în situații în care nu cunoașteți clasele pe care doriți să le utilizați în avans și doriți să creați obiecte dinamic în funcție de condițiile de la timpul de execuție.
3. Invocarea metodelor dinamice: Reflection permite apelarea metodelor unei clase la timpul de execuție, fără a cunoaște numele metodei în avans. Acest lucru permite crearea de aplicații flexibile, care pot apela metode în funcție de condițiile de la timpul de execuție.
4. Manipularea câmpurilor și a atributelor: Reflection permite citirea și scrierea valorilor câmpurilor unei clase la timpul de execuție, chiar și atunci când acestea sunt private sau protejate. Acest lucru este util în situații în care doriți să inspectați și să modificați starea internă a obiectelor dinamice.
    - In aceasta clasa aceste lucruri am dorit si eu sa le implementez. Mai jos se poate observa codul pentru metoda de inserare:

```
➢   public void insert(T t) {
        // TODO:
        Connection connection = null;
        PreparedStatement statement = null;
        ResultSet resultSet = null;
        StringBuilder query = new StringBuilder();
        query.append("INSERT INTO ");
        Class c;
        try {
            c = Class.forName("Model." +
    t.getClass().getSimpleName());
        } catch (ClassNotFoundException e) {
```

```
            throw new RuntimeException(e);
        }
        query.append(c.getSimpleName().toLowerCase());
```

Metoda insert prezentată utilizează Reflection pentru a insera un obiect de tip T într-o bază de date. Pana in acest punct se realizeaza initializarea variabilelor connection, statement și resultSet cu valoarea null, precum și un obiect StringBuilder numit query pentru a construi interogarea SQL. Apoi, se adaugă la interogare prefixul "INSERT INTO" si se obține clasa obiectului t și se adaugă numele clasei la interogare.

```
query.append(" (");
int i=0;
for (Field f : t.getClass().getDeclaredFields())
{
    query.append(f.getName());
    if(t.getClass().getDeclaredFields().length!=i+1)
    query.append(", ");
    i++;
}
query.append(") VALUES ");
StringBuilder cName=new StringBuilder();
cName.append("DataAcces.");
cName.append(c.getSimpleName());
cName.append("DAO");
```

In aceasta parte a codului se realizeaza : adaugarea la interogare parantezele deschise pentru specificarea coloanelor în care se vor insera valorile, parcurgerea tuturor câmpurile declarate ale clasei obiectului t și se adaugă numele lor la interogare. Dacă nu este ultimul câmp, se adaugă și o virgulă. Se adaugă la interogare parantezele închise și cuvântul cheie "VALUES" si se construiește un StringBuilder numit cName pentru a forma numele unei clase DAO, bazându-se pe numele clasei obiectului t.

```
Class cc;
try {
    cc = Class.forName(cName.toString());
} catch (ClassNotFoundException e) {
    throw new RuntimeException(e);
}
try{
    Class [] attr=new Class[1];
    attr[0]=t.getClass();
    StringBuilder meth=new StringBuilder();
    meth.append("add");
    meth.append(t.getClass().getSimpleName());
    String nameMeth= String.valueOf(meth);

query.append(cc.getMethod(nameMeth,attr).invoke(cc.getDeclaredConstructor()
.newInstance(),t));
} catch (InvocationTargetException | InstantiationException |
NoSuchMethodException | IllegalAccessException e) {
    throw new RuntimeException(e);
}
```

Codul de mai sus realizeaza urmatoarele obiective: se obține clasa corespunzătoare numelui construit anterior și se construiește un array de clase attr cu un singur element de tipul clasei obiectului t, se construiește un StringBuilder numit meth pentru a forma numele metodei add urmată de numele clasei obiectului t, se obține metoda corespunzătoare numelui construit anterior prin intermediul clasei DAO și se apelează metoda folosind obiectul t ca argument.Rezultatul apelării metodei este adăugat la interogare.

```
try {
    connection = ConnectionFactory.getConnection();
    statement = connection.prepareStatement(query.toString());
```

```
    statement.executeUpdate();
} catch (SQLException e) {
    LOGGER.log(Level.WARNING, type.getName() + "DAO:insert " +
e.getMessage());
} finally {
    ConnectionFactory.close(statement);
    ConnectionFactory.close(connection);
}
```

Iar in ultima parte a metodei se obține o conexiune la baza de date prin intermediul clasei ConnectionFactory. Se pregătește declarația SQL și se execută interogarea pentru a insera obiectul în baza de date. În caz de excepție, se înregistrează un mesaj de avertizare în jurnalul LOGGER. În final, se închid declarația și conexiunea cu baza de date prin intermediul clasei ConnectionFactory.

- Insert- apelează o interogare pentru tabela Client/ Product/ Order, în care inserează un obiect de tipul Client/ Product/ Order primit ca parametru.
- Delete- apelează o interogare pentru tabela Client/ Product/ Order , în care șterge Client/ Product/ Order cu id primit ca parametru.
- Update- - apelează o interogare pentru tabela Client/ Product/ Order , în care se salveaza noile valori pentru Client/ Product/ Order cu id primit ca parametru.

Exista o relatie de *mostenire* intre clasa AbstractDAO si clasele: Clasa ClientsDAO, Clasa ProductsDAO, Clasa OrderDAO.

5. Concluzii

Acest proiect a reprezentat prima mea experiență de lucru cu baze de date în Java. Am găsit partea de interogări destul de ușoară, având în vedere că am avut anterior o materie dedicată bazelor de date. Cu toate acestea, partea de conectare la baza de date a fost puțin mai dificilă, dar cu ajutorul colegilor am reușit să înțeleg aproximativ ce se întâmplă și cum trebuie să procedez.

6. Bibliografie

- *https://docs.oracle.com/javase/tutorial/essential/io/*
- *https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html*
- *https://jenkov.com/tutorials/java-reflection/index.html*
- *https://www.baeldung.com/javadoc*