**Documentatie-Tema 3**

**Order management system**

***Calacean Ionut Eugen***

***Facultatea de automatica si calculatoare***

***Grupa 30229***

***Indrumator laborator:Pop Cristina***

**Cuprins**

1. **Obiectivul temei.**
2. **Analiza problemei, modelare, scenarii, cazuri de utilizare.**
3. **Proiectare(decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator).**
4. **Implementare.**
5. **Rezultate.**
6. **Concluzii.**
7. **Bibliografie.**

**1.Obiectivul temei**

Bazele de date sunt comun utilizate in gestionarea comenzilor efectuate de catre clienti, aici se stocheaza datele clientilor, datele produselor, stocurile existente, preturile, si toate comenzile efectuate de catre clienti. Gestionarea unui sistem de comezi este focusata pe introducerea in baza de date a detaliilor legate de o comanda efectuata de catre un client.

Aplicatia ar trebui sa permita introducerea, stergerea, modificarea unui client, afisarea tuturor clientilor cu detaliile aferente, acelasi lucru ar trebui permis si pentru produse( introducerea unui produs nou, modificarea unui produs existent in baza de date sau stergerea acestuia. Tot in cadrul aplicatiei utilizatorul poate sa efectueze o comanda, prin selectarea clientului care doreste sa comande ceva, prin selectarea produsului dorit, si introducerea cantitatii. Detalii care vor fi apoi mutate si procesate in tabelul de comenzi.

**Cerinta principala** a temei este:” Considerati o aplicatie de gestionat comenzi pentru a procesa comenzile unor clienti pentru un depozit. Bazele de date relationale sunt utilizate pentru a stoca pridusele, clientii si comenzile. In plus, aplicatia va folosi (minim) urmatoarele clase:

* **Model classes** –reprezinta datele modelelor din aplicatie(client , comanda, produs);
* **Business Logic classes** – acestea contin logica aplicatiei;
* **Presentation classes** – clase care contine aplicatia grafica utilizator;
* **Data access classes** – clase care contin accecsul la baza de date;

Alte clase si pachete pot fi introduse in aplicatie pentru a implementa corect functionalitatea ei.

1. Analizati domeniul aplicatiei, determinati structura si comportamentul claselor acesteia si desenati o diagrama UML de clase extinsa;
2. Implementati clasele aplicatiei. Folositit javadoc pentru a documenta clasele;
3. Folositi tehnica reflectiei pentru a crea o metoda createTable() care primeste o lista de obiecte si genereaza header-ul tabelului prin extragerea prin reflectie a proprietetilor obiectului si apoi il populeaza cu valorile elementelor din lista:

Jtable createTable(List<Object> objects);

1. Implementati un sistem de programe utilitare pentru a reporta de exemplu: under-stock, totals, filters, etc.”

**Obiective secundare:**

* Crearea unei ferestre separate pentru operatiile pe tabelul de clienti: adaugare client nou, modificare client existent, stergere client, afisare lista clienti;
* Crearea unei ferestre saparate pentru operatiile pe tabelul de produse: adaugare produs nou, modificare produs existent, stergere produs, afisare lista produse;
* Crearea unei facturi in format .txt sau .pdf pentru fiecare comanda efectuata de catre un client;
* Organizarea aplicatiei pe Layere: dataAccessLayer, businessLayer, model, presentation;
* Folosirea unei baze de date corecte, care necesita exitenta unui tabel in plus, cel de OrderItem, adica pentru un produs comandat, pentru a exista posibiliatea ca un client sa comande mai multe produse;
* Folosirea tehnicilor de reflectie pentru a crea o clasa generica care sa contina metode de accesare a bazei de date, de creare, editare, stergere sau gasire obiecte. Astfel interogarile pentru obiecte specifice care corespund unui tabel vor fi generate dinamic in aceasta clasa prin reflectie;

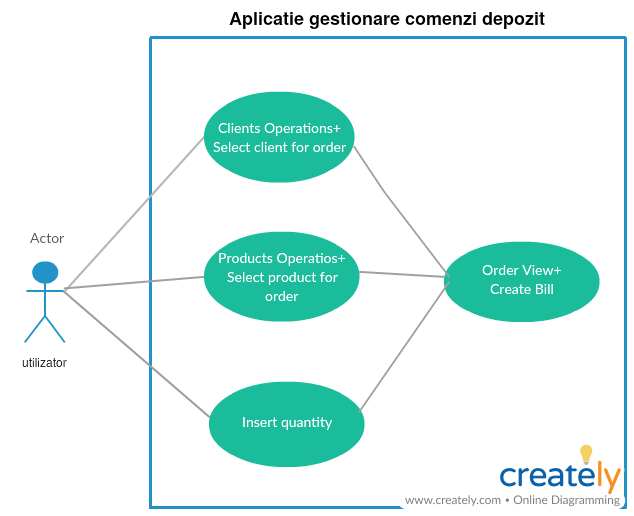
**2.Analiza problemei, modelare, scenarii, cazuri de utilizare**

Analiza preliminara a problemei ne imparte sarcina in crearea in primul rand a modelelor de baza pentru aplicatie, si anume clasele care vor corespunde cu tabelele din baza de date(Client, Product, OrderTable, OrderItem), cu ajutorul acestor clase, ale caror campuri corespund neaparat cu numele coloanelor din baza de date, se vor face interogari specifice, datele din baza noastra de date relationala vor fi stocate in aplicatia Java practic in aceste clase.

Tot din analiza problemei reiese faptul ca avem nevoie de o clasa care face conectarea la baza de date: ConnectionFactory, in aceasta clasa se vor crea conexiunile cu baza de date relationala si se vor inchide conexiunile de tip: statement, resultData, connection dupa fiecare interogare.

Prin modelarea problemei, se identifica faptul ca in clasele noastre o sa avem nevoie de colectii, de exemplu pentru stocarea validatorilor necesari la introducerea datelor in aplicatie, pentru crearea listei obiectelor rezultate in urma unor interogari, astfel avem nevoie de folosirea mai multor ArrayList-uri de diferite clase din modelul nostru, sau de validatori, pa care ii voi prezenta ulterior.

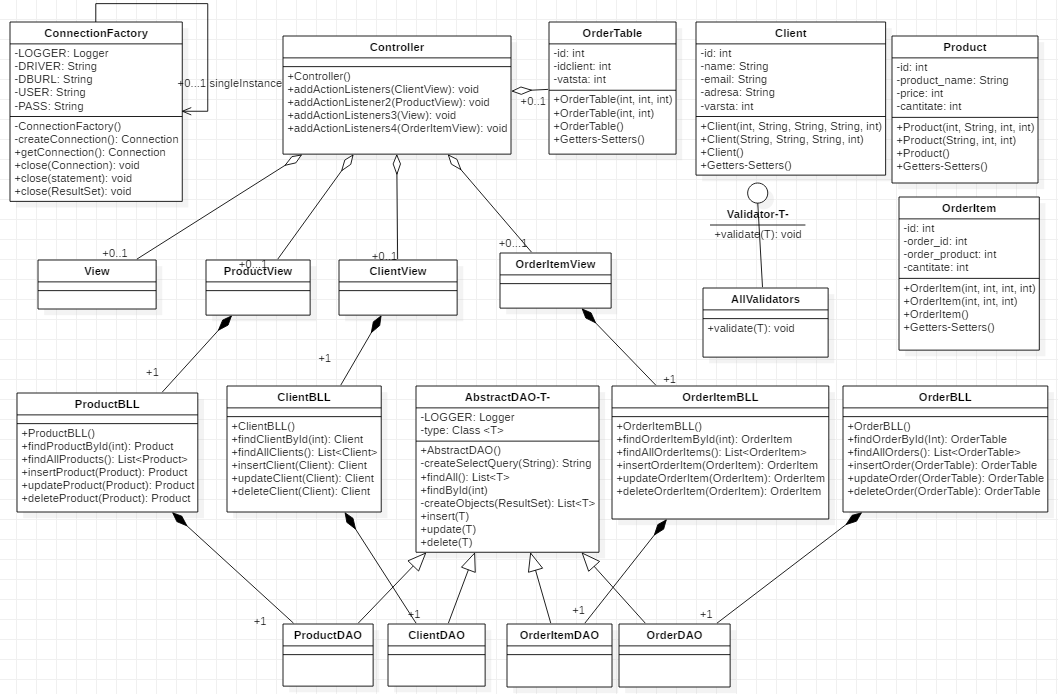
In continuare voi prezenta scenarii si cazuri de utilizare sub forma de diagrame use-case:



* **Use case: Clients operations**
* **Primary actor:utilizatorul**
* **Main succes scenario:**
  + Utiliatorul porneste view-ul pentru oparatiile cu clienti
  + Utilizatorul introduce toate datele cerute pentru a introduce sau a modifica un client existent, doar id-ul pentru a-l sterge
  + Utilizatorul apasa butonul corespunzator optiunii dorite
  + Aplicatia verifica daca inputul este corect daca se doreste inserare si toate datele sunt valide, daca se face update aplicatia verifica daca clientul exista, iar daca se face stergere la fel
  + Utilizatorul preia datele din tabelul curent afisat care contine toti clientii
  + Utilizatorul poate continua cu gestionarea tabelului curent, prin inserari, modificari sau stergeri sau reveni la meniul anterior
  + Utilzatorul poate selecta un client pentru a efectua o comanda
* **Use case: Operatie de insert, edit sau delete**
* **Primary actor:utilizatorul**
* **Alternative Sequences:**

1. Datele care se doresc a fi inserate nu sunt valide:
   1. Nume client prea lung
   2. Email invalid sau prea lung
   3. Adresa prea lunga
   4. Varsta sub 12 ani sau peste 90
   5. Nume produs prea lung
   6. Pret produs negativ
   7. Cantitate produs negativa sau peste 1000
2. Cantitatea care se doreste a fi comandata este insuficienta:
   1. Utilizatorul cere o cantitate a unui produs care nu exista in atatea exemplare

**3.Proiectare**

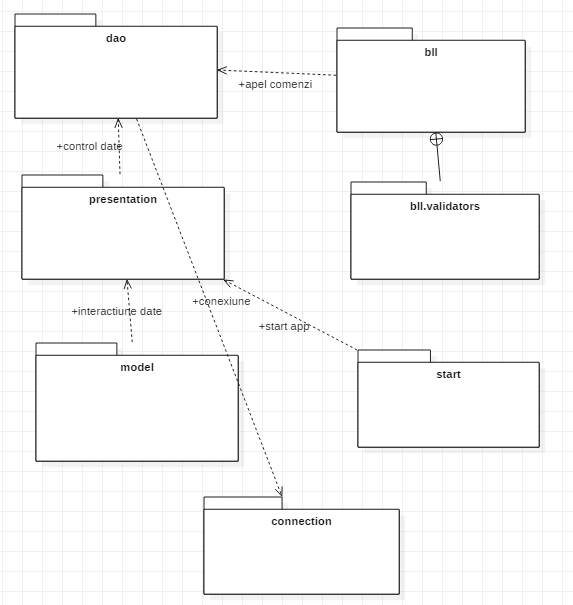


Clase Folosite:

1. Clasa **Main-**principala clasa, nu mai este pusa pe diagrama deoarece oricum este independenta, aici se construieste View-ul principal, si se leaga la Controller;
2. Clasa **ConnectionFactory**-aceasta clasa creeaza conexiunea propriu zisa cu baza de date si incide comunicarea cu aceasta
3. Clasa **Controller**-aceasta clasa controleaza functionalitatea de baza a aplicatiei, inserarea comenzilor, si dinamica graficii afisate spre utilizator
4. Clasele **OrderTable,Product,OrderItem,Client** –constituie modelul aplicatiei, atributele lor coincid cu coloanele din baza de date,acestea sunt oarecum independente fiind folosite ca si clase pentru genericitate.Clasa OrderTable este legata la Controller pentru a oferi posibilitatea crearii unei comenzi a mai multor produse de catre acelati client pe aceeasi factura de comanda
5. Interfata **Validator<T>, Clasele Validator-** folosite pentru a controla datele la inserare, pentru a pastra in bazele de date doar date corecte
6. Clasele **ClientView, ProductView,OrderItemView-** constituie cele 3 ferestere principale cu care interactioneaza utilizatorul aplicatiei
7. Clasele **ClientBLL,ProductBLL,OrderItemBLL,OrderBLL-** fac parte din partea de business layer care contine logica aplicatiei
8. Clasele **AbstractDAO,ClientDAO,ProductDAO, OrderItemDAO,OrderDAO-** fac parte din data access layer, de aici se acceseaza datele prin intermediul clasei AbstractDAO care este o clasa generica extinsa de toate celelalte clase , pentru tipul respectiv din model

Ca alte tipuri de date folosite aici intra ArrayList-ul. Colectiile se folosesc pentru a crea de exemplu o lista din ResultSet, pentru interogari de tipul findAll(), astfel se construieste o lista cu tipul de date respectiv din pachetul model, cu aceasta se construieste apoi un JTable. Tot colectiile se folosesc si in cazul validatorilor, se parcurge lista de validatori pentru fiecare insert si astfel se verifica ca toate datele introduse sa aiba o corectitudine logica.

In continuare pe urmatoarea pagina se va prezenta diagrama de pachete:



Aplicatia este organizate pe layere. Astfel se identifica data access layer(pachetul dao), business layer(pachetul bll),presentation layer(pachetul presentation), model layer(pachetul model) , pe langa acestea mai avem pachetul start care porneste practic aplicatia si contine o clasa cu metode statice ReflectionTable care creeaza un DefaultTableModel, din care ulterior se va crea JTable-ul necesar pentru fiecare view in parte. Un alt pachet necesar este connection, pachet in care se creeaza si apoi se inchid conexiunile necesare intre aplicatia noastra si bazele de date relationale.

**4.Implementare**

Voi prezenta in continuare implementarea unor clase, care sunt mai importante din aplicatie:

**Clasa AbstractDAO<T>**

Una dintre cele mai importante clase, aceasta contine o variabila de tip Logger, este o clasa generica astfel in constructor se scoate tipul clasei care este tot o variabila a clasei de tipul final:

this.type = (Class<T>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[0];

Aceasta clasa contine o metoda de gasire a tuturor obiectelor de un tip cautat, printr-o interogare de tipul “SELECT \* FROM..”, iar ResultSet-ul obtinut se transmite mai departe pentru a crea o lista de obiecte specifice in functie de parametrul actual al clasei noastre:

public List<T> findAll() {

Connection connection = null;

PreparedStatement statement = null;

ResultSet resultSet = null;

String query = "SELECT \* FROM " + type.getSimpleName() + "";

try {

connection = ConnectionFactory.*getConnection*();

statement = connection.prepareStatement(query);

resultSet = statement.executeQuery();

return createObjects(resultSet);

} catch (SQLException e) {

***LOGGER***.log(Level.***WARNING***, type.getName() + "DAO:findAll " + e.getMessage());

} finally {

ConnectionFactory.*close*(resultSet);

ConnectionFactory.*close*(statement);

ConnectionFactory.*close*(connection);

}

return null;

}

private List<T> createObjects(ResultSet resultSet) {

List<T> list = new ArrayList<T>();

try {

while (resultSet.next()) {

T instance = type.newInstance();

for (Field field : type.getDeclaredFields()) {

Object value = resultSet.getObject(field.getName());

PropertyDescriptor propertyDescriptor = new PropertyDescriptor(field.getName(), type);

Method method = propertyDescriptor.getWriteMethod();

method.invoke(instance, value);

}

list.add(instance);

}

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (SecurityException e) {

e.printStackTrace();

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (InvocationTargetException e) {

e.printStackTrace();

} catch (SQLException e) {

e.printStackTrace();

} catch (IntrospectionException e) {

e.printStackTrace();

}

return list;

}

Tot in aceasta clasa avem implementate metodele de insert, update, si delete, toate generice, deci vor fi apelate cu un tip anume din clasa de data access layer respectiva. Dintre acestea voi prezenta metoda de insert, in aceasta se construieste prin metoda reflectiei query-ul specific, iar apoi se insereaza in baza de date. De retinut ca verificarile ca datele sa fie corecte nu se fac aici, ci in clasele care contin logica aplicatiei:

public T insert(T t) {

// **TODO**:

Connection connection = null;

PreparedStatement statement = null;

String query = "INSERT INTO " + type.getSimpleName() + "(";

int nbFields = 0;

for (Field field : t.getClass().getDeclaredFields()) {

field.setAccessible(true);

nbFields++;

}

int currIt = 0;

for (Field field : t.getClass().getDeclaredFields())

{

currIt++;

field.setAccessible(true);

if (currIt != nbFields) {

query += field.getName()+",";

} else {

query += field.getName();

}

}

query+=")";

query+=" VALUES(";

currIt = 0;

for (Field field : t.getClass().getDeclaredFields()) {

currIt++;

field.setAccessible(true);

if (currIt != nbFields) {

query += "?,";

} else {

query += "?";

}

}

query += ")";

System.***out***.println(query);

try {

connection = ConnectionFactory.*getConnection*();

statement = connection.prepareStatement(query);

int index = 1;

for (Field field : t.getClass().getDeclaredFields()) {

field.setAccessible(true);

Object value;

try {

value = field.get(t);

if (value instanceof String)

statement.setString(index++, (String) value);

else if (value instanceof Integer)

statement.setInt(index++, (Integer) value);

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

}

}

statement.executeUpdate();

} catch (SQLException e) {

***LOGGER***.log(Level.***WARNING***, type.getName() + "DAO:insert " + e.getMessage());

} finally {

ConnectionFactory.*close*(statement);

ConnectionFactory.*close*(connection);

}

return t;

}

**Interfata Validator<T>**

Interfata este folosita la validarea datelor, mai specific, pentru inserari in tabelul de clienti si in tabelul de produse. Aceasta interfata este implementata de fiecare validator in parte si apoi folosita ca tip de baza pentru un ArrayList de validatori pentru fiecare Tabela in parte, in clasa ce urmeaza se va descrie mai in detaliu rolul validatorilor.

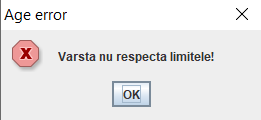
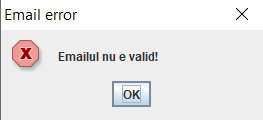
**Clasa ClientBLL**

Dintre clasele din Business Layer am ales sa prezint clasa ClientBLL care controleaza logica aplicatiei pentru operarea pe tabela de clienti.Aceasta clasa contine o lista de validatori cu tipul de baza din model: Client. Astfel in aceasta colectie sunt adaugati validatorii pentru fiecare atribut al modelului Client.

Tot in aceasta clasa mai avem o variabila de tipul ClientDAO, prin care se fac operatiile pe tabela de clienti, si modificari asupra bazelor de date.

Voi prezenta in continuare doar metoda de inserare, celelalte metode fiind mai putin complexe. Aceasta metoda incepe prin verificarea datelor introduse de utilizator in campurile respective. In cazul in care campurile respecta logica ceruta de program, clientul respectiv va fi inserat in tabela. Dupa acest lucru se mai face o verificare prin cautarea clientului in baza de date, iar daca nu se gaseste se semnaleaza o eroare in inserare.

In cazul in care datele nu sunt introduse corect, aplicatia va detecta acest lucru tocmai prin validatoarele introduse, si va afisa un mesaj sub forma celor care urmeaza(am ales 2 exemple):



**Clasa Client**

Din layer-ul model am ales sa prezint clasa Client.Cea mai importanta particularilate a claselor din model este ca atributele coloanelor din baza de date trebule sa coincida cu variabilele asociate clasei, pe langa acestea ar trebui sa aiba 2 constructori, unul pentru cazul in care se introduce si id-ul, altul fara id, incrementarea fiind facuta automat de baza de date (daca se seteaza auto-increment), un constructor gol, necesar la crearea listei din AbstractDAO, si gettere si settere.

**Clasa ReflectionTable**

Aceasta clasa se foloseste de metoda reflectiei pentru a crea un DeflaulTableModel dintr-o lista de obiecte data ca parametru:

public static DefaultTableModel createTable(List<? extends Object> objects)

{

//JTable toReturn;

DefaultTableModel tableModel;

int nbFields=0;

for(Field field:objects.get(0).getClass().getDeclaredFields())

{

field.setAccessible(true);

nbFields++;

}

String[] column =new String[nbFields];

String[][] data = new String[objects.size()][nbFields];

int fieldsIndex=0;

for(Field field:objects.get(0).getClass().getDeclaredFields())

{

field.setAccessible(true);

column[fieldsIndex++]=field.getName();

}

Iterator<? extends Object> it=objects.iterator();

int rowIndex=0;

while(it.hasNext())

{

Object myOb=it.next();

int columnIndex=0;

for(Field field:myOb.getClass().getDeclaredFields())

{

field.setAccessible(true);

Object value;

try {

value=field.get(myOb);

data[rowIndex][columnIndex++]=""+value+"";

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

}

}

rowIndex++;

}

//toReturn = new JTable(data,column);

//toReturn.setBounds(100,200,400,500);

//return toReturn;

tableModel=new DefaultTableModel(data,column);

//tableModel.fireTableDataChanged();

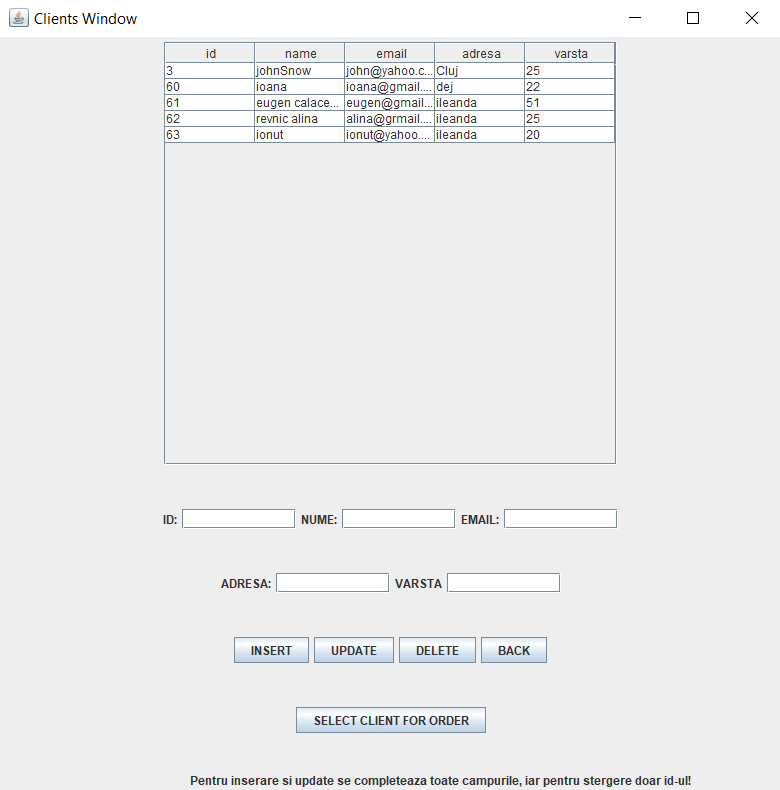
return tableModel;

}

Clasa ar putea sa returneze si direct un JTable(acest fapt este comentat mai sus), motivul pentru care am folosit DefaultTableModel este metoda acestuia : **fireTableDataChanged()** care ne reactualizeaza tabelul care utilizeaza modelul curent(adica JTable-ul nostru).

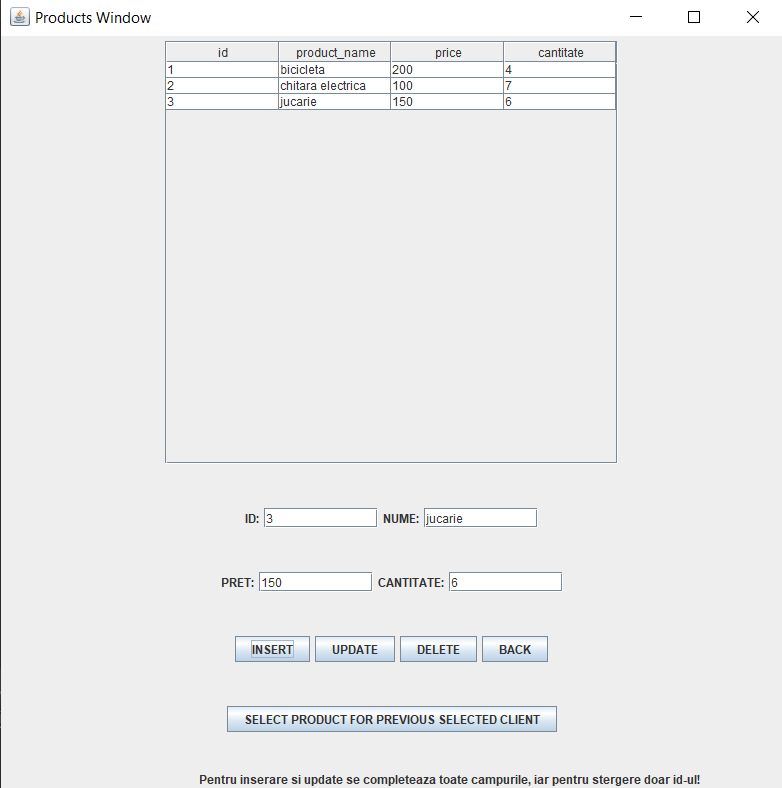
**Clasa ClientView**

Aceasta clasa creeaza interfata pentru client care dupa modificarile efectuate( in principal m.am folosit de BoxLayout pentru aranjare), un JScrollPane adaugat la tabel(fapt care face automat sa se afiseze si header-ul tabelului meu va arata in felul urmator:



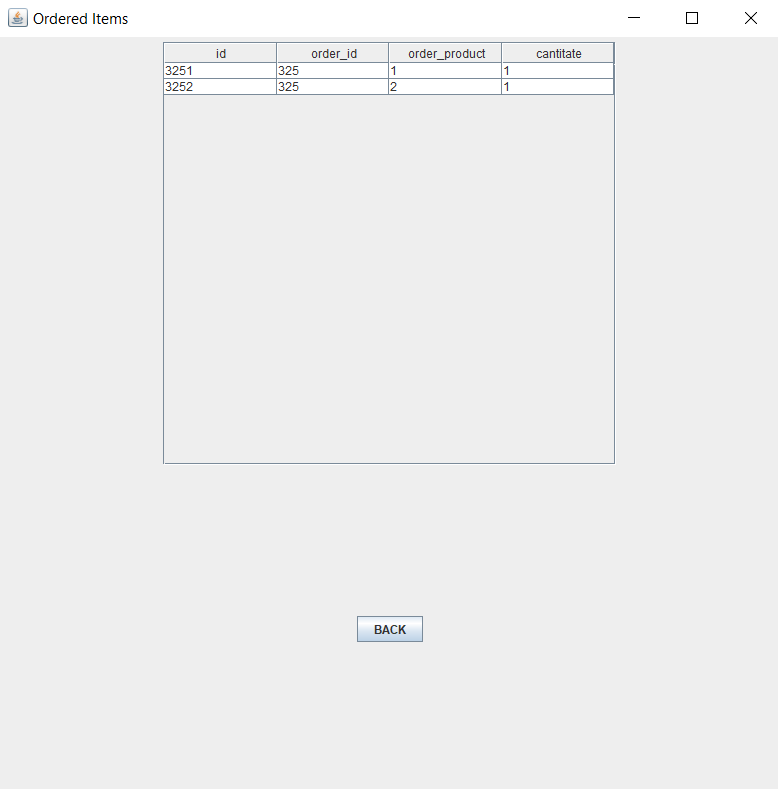
**Clasa ProductView**

Simulara cu cea descrisa anterior, voi prezenta rezultatul obtinut:



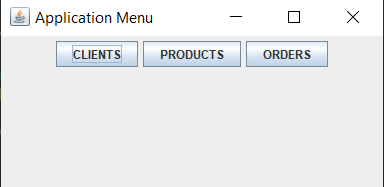
**Clasa OrderItemView**

Aceasta clasa doar ofera posibilitatea afisarii tutuor produselor comandate. Pentru a comanda efectiv un produs este necesara o parcurgere in ordine a celor 2 interfete prezentate anterior.Rezultatul obtinut pentru fereastra de vizualizare produse comandate:

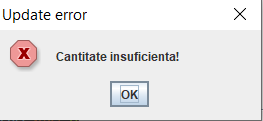


**Clasa View**

Aceasta clasa contine View-ul principal,care este initializat in pachetul start si apoi legat de Controller. Interfata principala, de selectare a tabelulul asupra caruia urmeaza sa se opereze, arata in felul urmator:



Aplicatia este facuta astfel incat sa detecteze cazurile de stoc insuficient( un anumit client comanda un produs intr-o cantitate care este peste cantitatea disponibila in depozit) acest lucru este semnalat printr-o grafica care arata in felul urmator:



**5.Rezultate**

Pe baza aplicatiei se poate manageria un depozit, prin controlul total asupra clientilor, produselor si comenzilor efectuate. Astfel administratorul de baze de date are o misiune mai usoara, Java permitand efectuarea de operatii pe baza de date intr-un mod exponential mai usor.

Astfel se pot obtine usor informatii cu privire la stocurile de marfa, la comenzile in derulare, la adresa de expeditie a comenzii, la produsele cele mai comandate, care ar necesita un stoc mai ridicat, la perioada cand se comanda mai mult un produs, sau care sunt produsele care raman pentru mult timp in stoc, nefiind in preferintele clientilor.

Aplicatia mentine un istoric al comenzilor (finalizate sau nu) astfel se poate gestiona si din punct de vedere economic depozitul(vanzari efectuate, profit,pierderi) etc.

**6.Concluzii**

Ca o concluzie la acest proiect, se observa importanta incontestabila a utilizatii bazelor de date si a conexiunii Java cu bazele de date relationale. Castigul exista de ambele parti, pe de o parte prin intermediu JAVA API si a interfetelor construite gestionam mult mai usor bazele de date, iar pe de alta parte prin construirea generica a unor metode, colectiile se obtin foarte usor in mediu de dezvoltare Java, astfel maleabilitatea gestionarii comenzilor, produselor si a clientilor fiind foarte crescuta.

Unul dintre aspectele importante care m-a ajutat la dezvoltarea cunostintelor de programare este folosirea tehnicii de reflectie: prin aceasta putem manipula mult mai usor datele , asta pe langa faptul ca ne scuteste de mult cod. Un alt aspect important este folosirea genericitatii puse la dispozitie de Java, prin aceasta iarasi suntem scutiti de a repeta scrierea unui cod pentru niste parametrii diferiti. Folosirea si conectarea la bazele de date este tot un lucru nou pentru mine, importanta acestuia fiind una incontestabila.

Din punct de vedere al dezvoltarii ulterioare, s-ar putea implementa si un tabel de produse favorite, prin intermediul acestuia clientul ar putea selecta unele produse de care este interesat, iar pe baza acestui tabel, cel care gestioneaza depozitul ar putea sa profite de intentiile clientilor(reduceri pentru produsele preferate, aducerea unor produse similare). Tot ca dezvoltare ulterioara s-ar putea implementa unele metode de afisare ale produselor celor mai vandute, celor mai ieftine, scumpe, cu cele mai multe review-uri etc), review-urile clientilor, ar putea constitui de exemplu un alt camp de tip lista in atributele produsului;

**7.Bibliografie**

[1] <https://stackoverflow.com/questions/3179136/jtable-how-to-refresh-table-model-after-insert-delete-or-update-the-data>

[2] <http://users.utcluj.ro/~crisb_pop/pt2018.html>

[3] <https://utcn_dsrl@bitbucket.org/utcn_dsrl/pt-layered-architecture.git>