# DOCUMENTAȚIE

TEMA 3

Nume : Rîpaș Bianca Ioana

Grupa: 30228

Cuprins

[1. Obiectivul temei 3](#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 4](#_Toc95297886)

[3. Proiectare 7](#_Toc95297887)

[4. Implementare 14](#_Toc95297888)

[5. Rezultate 29](#_Toc95297889)

[6. Concluzii 31](#_Toc95297890)

[7. Bibliografie 32](#_Toc95297891)

1.Obiectivul temei

Obiectiv principal

• Proiectarea si implementarea unei aplicatii pentru gestionarea comenzilor clientilor pentru un depozit

Obiective secundare

|  |  |
| --- | --- |
| Analizați problema și identificați cerințele | 4 |
| Proiectați aplicația de gestionare a comenzilor | 7 |
| Implementarea aplicației de gestionare a comenzilor | 14 |
| Testarea aplicației de gestionare a comenzilor | 29 |

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



Modificare produs/client/comanda

Afisare client->

produs->comanda

Adaugare client->

produs->comanda

Stergere produs/client/comanda

**Cerințe funcționale**

* Aplicația de administrare ar trebui să permită angajatului să introducă un nou client/produs/comanda.
* Aplicația de administrare ar trebui să permită angajatului să modifice un client/produs.
* Aplicația de administrare ar trebui să permită angajatului să șteargă un client/ produs.
* Aplicația de administrare ar trebui să permită angajatului să afiseze clientii/ produsele/comezile.

**Cerințe non-funcționale**

* Aplicația de simulare ar trebui să fie interactivă.
* Aplicația de simulare ar trebui să fie ușor de folosit de către utilizator.

**Cazuri de utilizare**

1.Caz de utilizare:adăugare client/produs

Actor principal: angajatul

Scenariu principal

1. Angajatul completează câmpurile necesare cu informațiile clientului/produsului
2. Angajatul apasă pe butonul “Adauga”
3. Aplicația introduce clientul în baza de date și afișează un mesaj corespunzător de succes

Scenariu alternativ

1. Angajatul nu introduce date în toate câmpurile necesare
2. Aplicația un mesaj de eroare corespunzător erorii
3. Scenariul se întoarce la punctul 1

2.Caz de utilizare:adăugare comandă

Actor principal: angajatul

Scenariu principal

1. Angajatul completează câmpul pentru Id, cantitate și selectează un client și un produs din tabelele corespunzătoare clientilor si produselor

2. Angajatul apasă pe butonul “Adauga”

3. Aplicația introduce comanda în baza de date și afișează un mesaj corespunzător de scucces

Scenariu alternativ

1. Angajatul nu introduce date valide în toate câmpurile necesare efectuarii unei comenzi

2. Aplicația un mesaj de eroare corespunzător

3. Scenariul se întoarce la punctul 1

3.Caz de utilizare:ștergere client/produs

Actor principal: angajatul

Scenariu principal

1. Angajatul completează câmpurile necesare cu informațiile clientului (se poate completa doar Id-ul clientului/produsului)

2. Angajatul apasă pe butonul “Șterge”

3. Aplicația șterge clientul din baza de date și afișează un mesaj corespunzător de succes

Scenariu alternativ

1. Angajatul nu introduce date în toate câmpurile necesare

2. Aplicația un mesaj de eroare corespunzător

3. Scenariul se întoarce la punctul 1

4. Caz de utilizare:modificare client/produs

Actor principal: angajatul

Scenariu principal

1. Angajatul completează câmpurile necesare cu informațiile clientului/produsului

2. Angajatul apasă pe butonul “Modifica”

3. Aplicația modifică clientul din baza de date și afișează un mesaj corespunzător de succes

Scenariu alternativ

1. Angajatul nu introduce date în toate câmpurile necesare efectuarii unei modificari

2. Aplicația un mesaj de eroare corespunzător

3. Scenariul se întoarce la punctul 1

5. Caz de utilizare:afișare clienti/produse/comenzi

Actor principal: angajatul

Scenariu principal

1. Angajatul apasă pe butonul “Afisare”

2. Aplicația afișează toți clienții/toate produsele /comenzile care au fost adăugați/adaugate în baza de date

Scenariu alternativ

Nu exista scenariu alternativ

3.Proiectare

Proiectul contine 8 pachete.

Diagrama de pachete este :

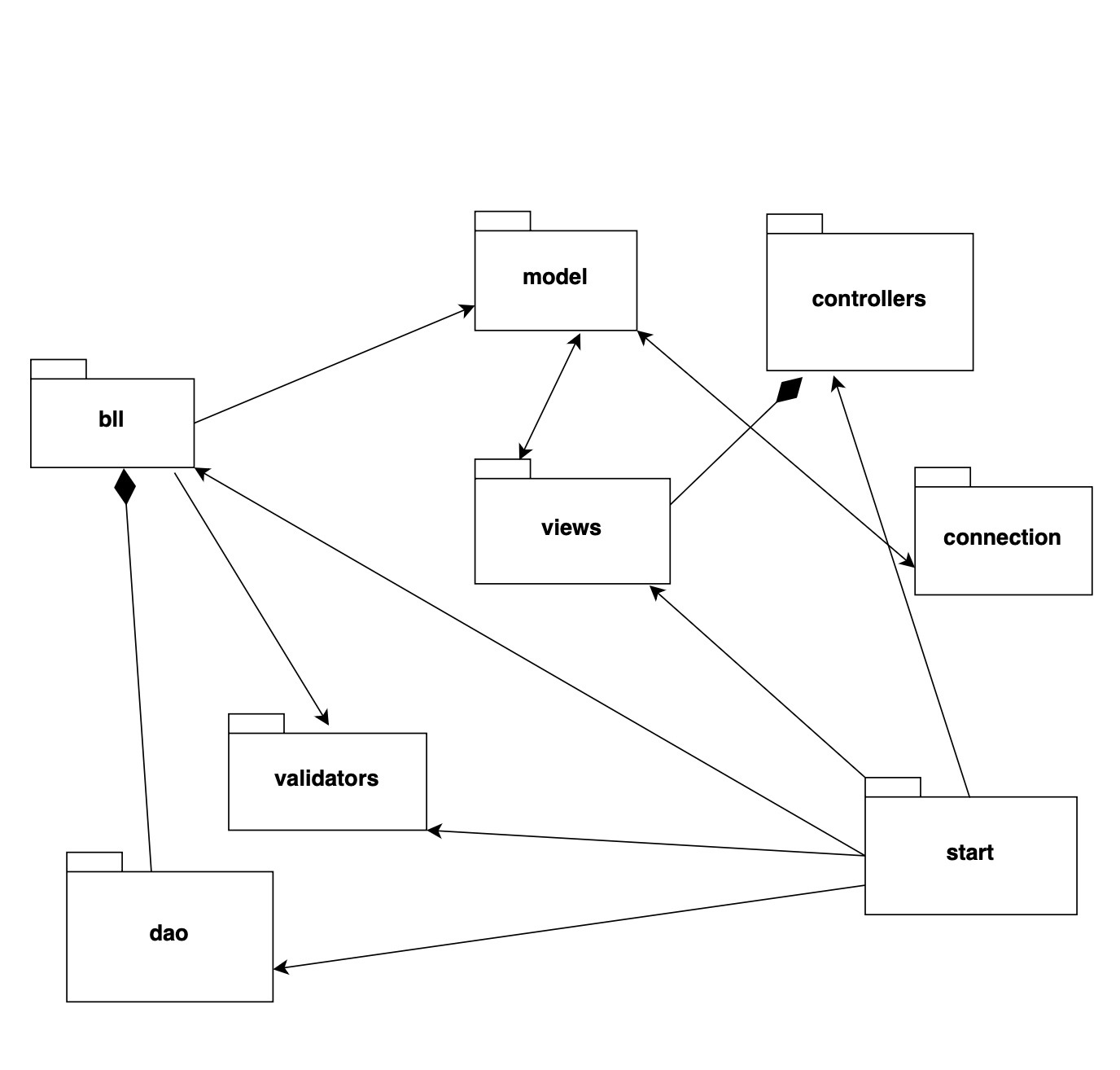
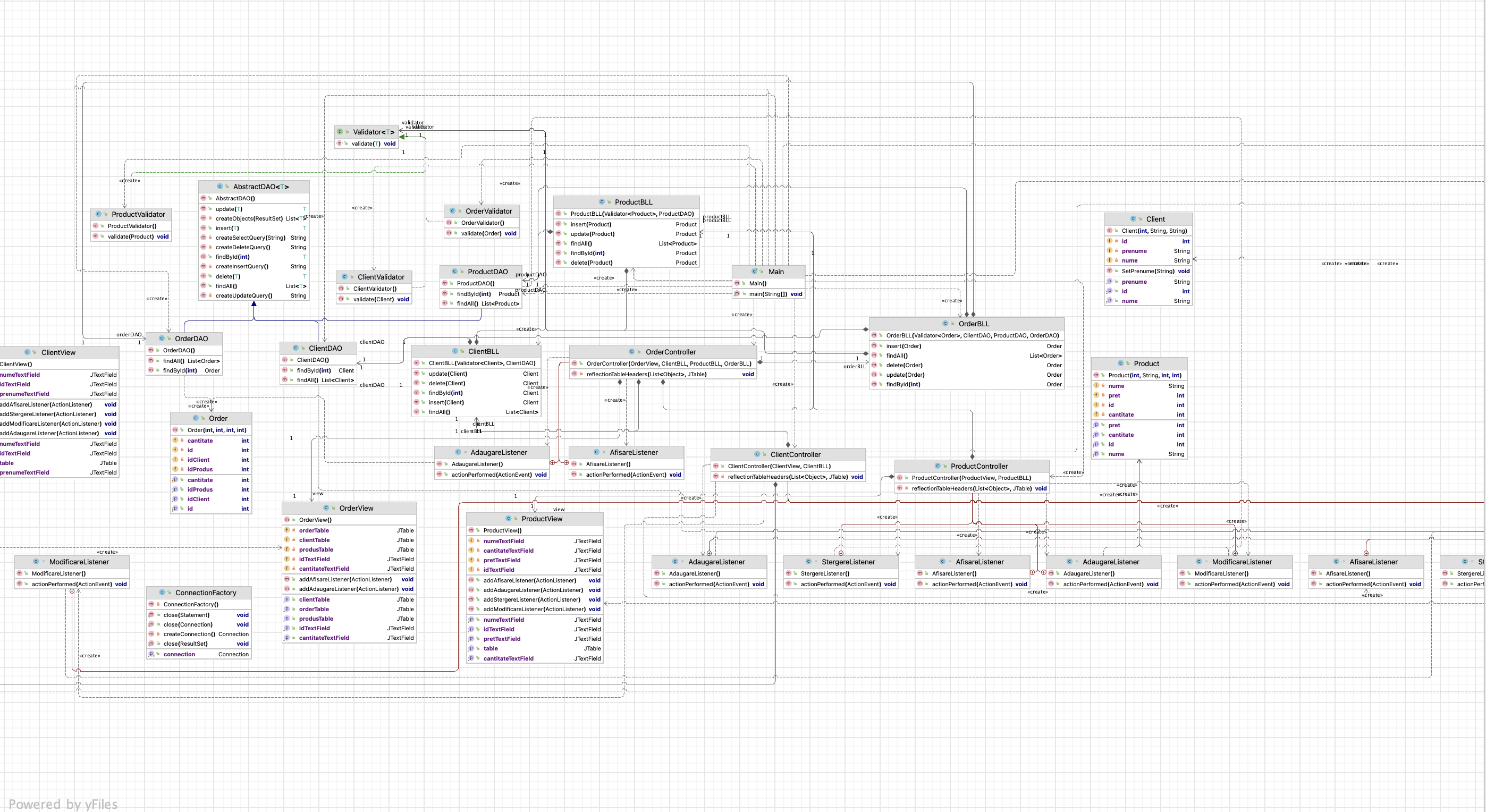


Diagrama de clase este :



Atunci cand programul ruleaza cele 3 interfete vor fii afisate angajatului:

Interfata pentru a realiza o comanda :

A screenshot of a computer

Description automatically generated

Interfata pentru a insera un client :

A screenshot of a computer

Description automatically generated with medium confidence

Inserare produs :

A screenshot of a computer

Description automatically generated

Atunci cand se insereaza un client/produs/comanda va aparea acest lucru in interfata :

A screenshot of a message

Description automatically generated with medium confidence

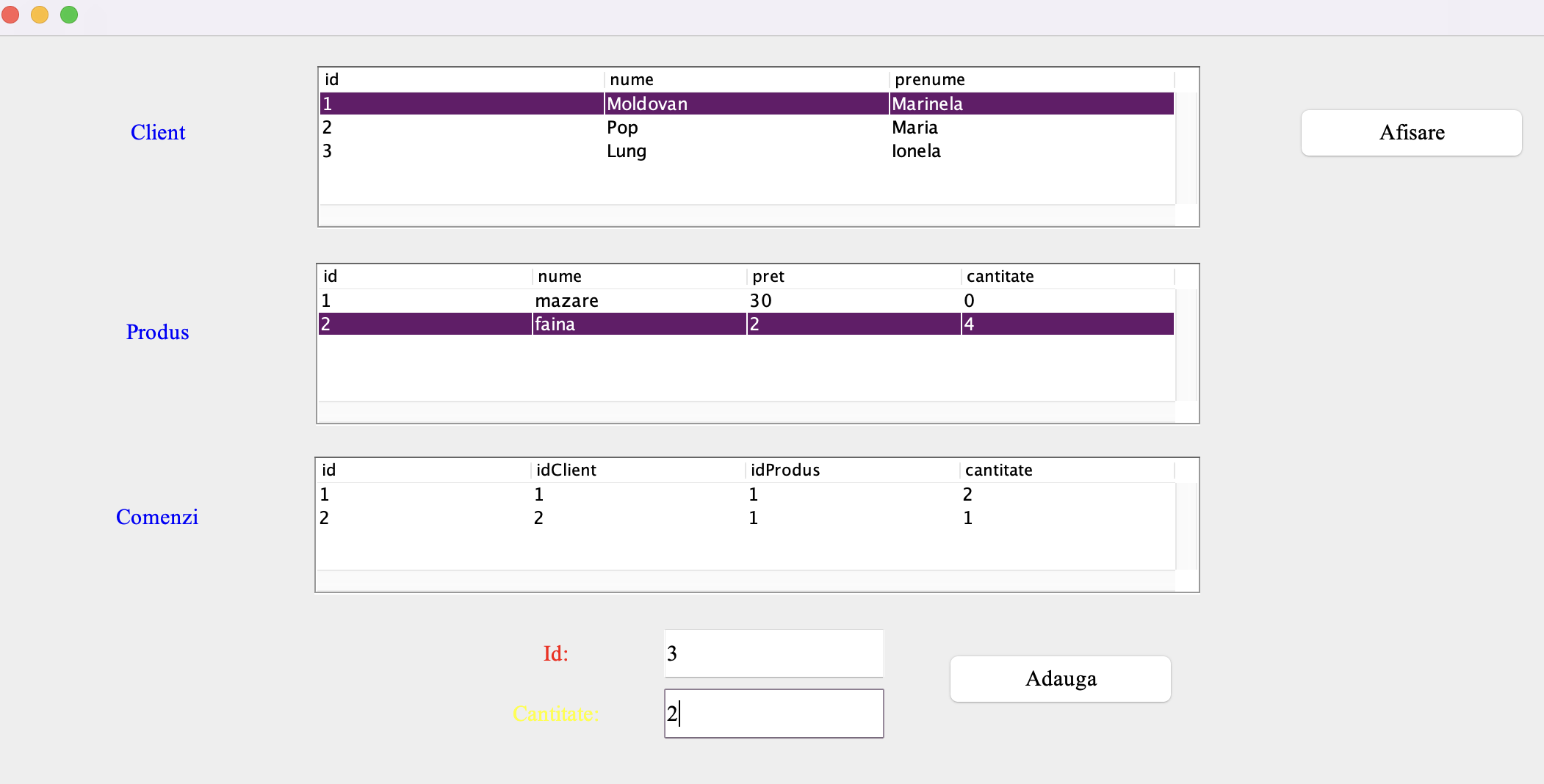
La momentul stergerii :



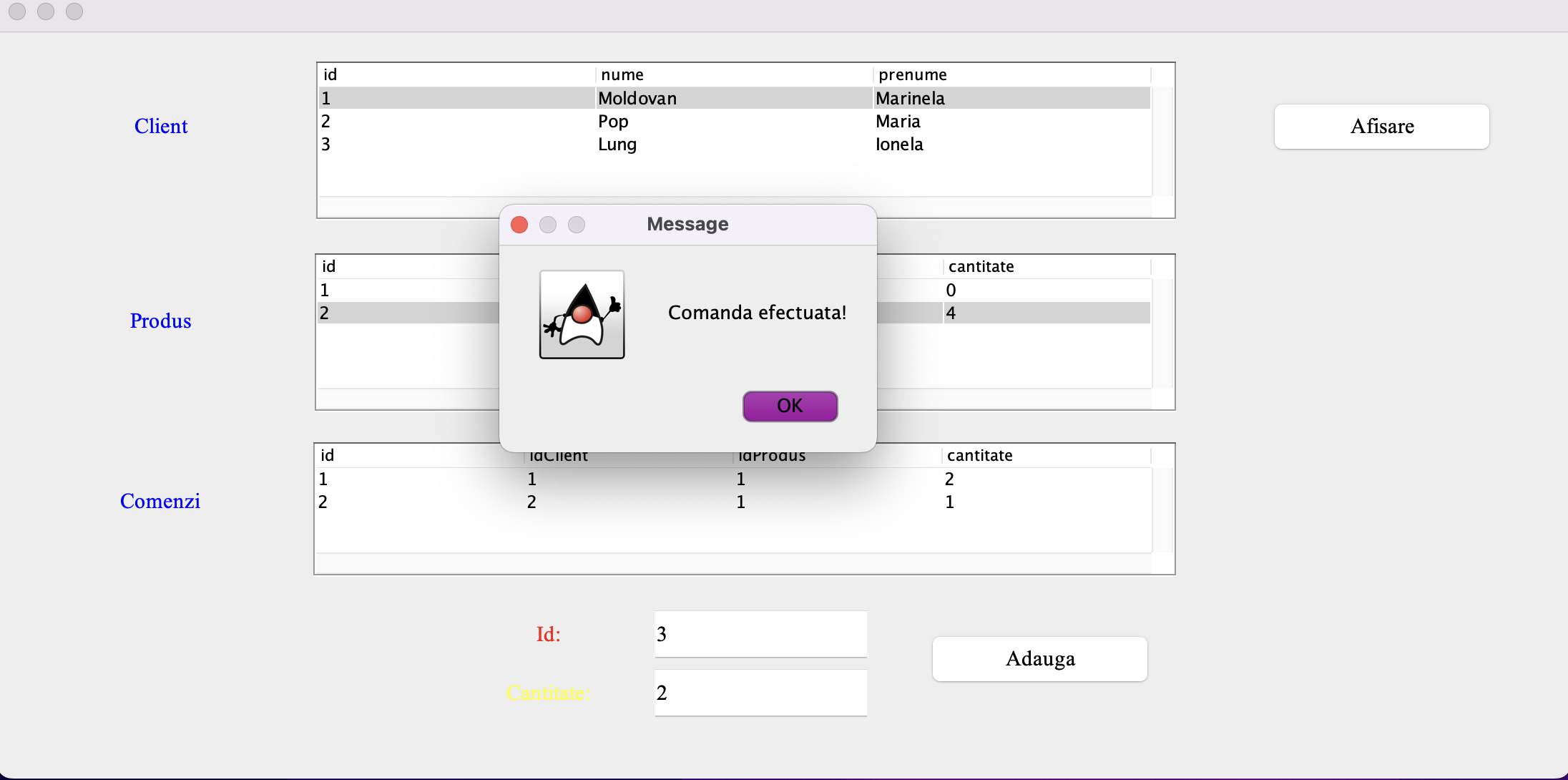
Daca un produs/client este modificat :



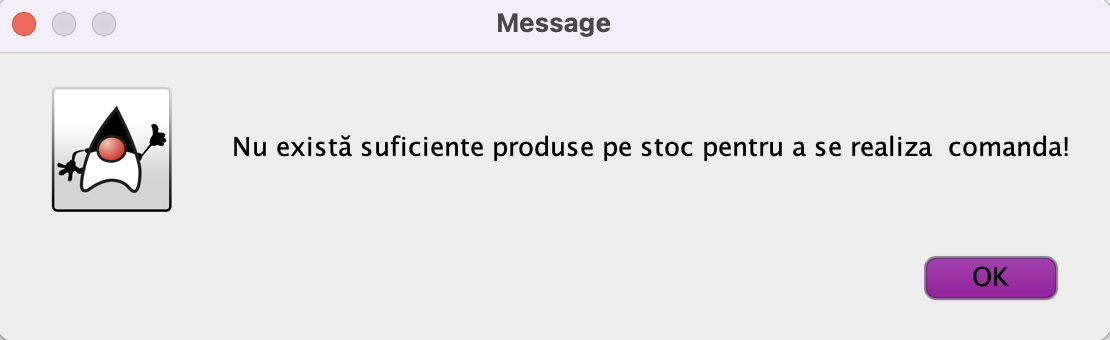
Asa arata interfata pentru comenzi in momentul inserarii :



Iar dupa ce angajatul a apasat butonul Adauga



In caz in care angajatul doreste sa adauge o comanda , dar nu exista sufficient stoc va aparea acest mesaj :



Model-View-Controller (MVC) este un pattern arhitectural folosit pentru a structura și organiza aplicațiile software. În cadrul acestui pattern, aplicația este împărțită în trei componente principale: Model, View și Controller.

Componenta Model reprezintă partea care gestionează datele și logica aplicației. Aceasta încapsulează datele și oferă metode și funcționalități pentru manipularea și accesarea acestora.

Componenta View se ocupă de afișarea datelor utilizatorului și de interacțiunea cu acesta. Ea primește informațiile necesare pentru a fi afișate din Model și le prezintă într-o formă vizuală.

Componenta Controller acționează ca intermediar între Model și View. Fiecare View are asociat un Controller care primește input-ul de la utilizator, cum ar fi evenimentele de la mouse sau tastatură, și le traduce în apeluri către Model sau View. Controller-ul preia acțiunile utilizatorului și le gestionează corespunzător prin interacțiunea cu Modelul și actualizarea View-ului.

Prin utilizarea pattern-ului MVC, se obține o separare clară a responsabilităților în aplicație. Modelul este responsabil de stocarea și manipularea datelor, View-ul se concentrează pe prezentare și interacțiunea cu utilizatorul, iar Controller-ul se ocupă de gestionarea input-ului și coordonarea comunicării între Model și View.

Divizarea in clase:

bll : ClientBLL, OrderBLL, ProductBLL

-aceste clase implementeaza logica pentru clasele ClientDAO, OrderDAO, ProductDAO

validators : ClientValidator, OrderValidator, ProductValidator,Validator ( interfata)

-cele 3 clase verifica datele pentru client/comanda/produs

connection : ConnectionFactory

- face legătura cu baza de date

controllers : ClientController, OrderController, ProductController

- care asimileaza acțiunile angajatului și returnează rezultatul potrivit actiunilor prin interfața grafică

dao : AbstractDAO, ClientDAO, OrderDAO, ProductDAO

-prima clasa are un atribut de tip generic, un constructor și metode care permit crearea generică a obiectelor, generarea de query-uri necesare și afișarea obiectelor pe baza unui ID specificat. De asemenea, include și metode pentru inserarea, actualizarea și ștergerea obiectelor în mod general

-ultimele 3 clase o extend pe prima

models : Client, Order, Product

-aceste clase contin atributele specifice , constructori , set-ere si get-ere

start : Main

- are ca atribute, view-urile și controller-ele si validatorii și dao-urile.

views : ClientView, OrderView, ProductView

- care implementează interfețele

4.Implementare

Clasa Client reprezintă o entitate care are atribute private, cum ar fi un ID de tip int, numele și adresa de tip String. Clasa oferă metode standard de acces (get()) și modificare (set()) pentru aceste atribute, permițând obținerea și setarea valorilor lor.

Clasa Product reprezintă o entitate care are atribute private, cum ar fi un ID și cantitatea de tip int, numele de tip String și prețul de tip double. Această clasă furnizează metode standard de acces (get()) și modificare (set()) pentru aceste atribute, permițând gestionarea și actualizarea informațiilor legate de produs.

Clasa Order reprezintă o entitate care are atribute private, cum ar fi un ID, IDClient, IDProdus și cantitatea de tip int. Această clasă oferă metode standard de acces (get()) și modificare (set()) pentru aceste atribute, permițând manipularea și gestionarea comenzilor, inclusiv informațiile legate de client, produs și cantitatea dorită.

Clasa **AbstractDAO** este o clasă abstractă care servește drept bază pentru clasele DAO (Data Access Object). Aceasta are un atribut privat de tipul Class<T>, care reprezintă tipul de obiect cu care clasa DAO va interacționa. Clasa are un constructor care utilizează reflexia pentru a obține informații despre clasa corespunzătoare și a inițializa obiectele asociate. Această clasă abstractă oferă un schelet comun și funcționalități de bază pentru clasele DAO specifice, permițând manipularea și gestionarea datelor într-un mod generic și flexibil.

*/\*\*  
 \* Constructorul clasei AbstractDAO este acesta :  
 \*/*@SuppressWarnings(**"unchecked"**)  
**public** AbstractDAO() {  
 **this**.**type** = (Class<T>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[0];  
}

Metodele , care folosesc reflexia sunt toate cele din clasa AbstractDAO, cateva exemple:

*/\*\*  
 \* Metoda care creeaza o lista de obiecte de tipul corespunzator tabelelor respective este aceasta :  
 \*  
 \** ***@param resultSet*** *contine field-urile fiecarei tabele  
 \** ***@return*** *o lista de obiecte de tipul corespunzator tabelei corespunzatoare  
 \*/***private** List<T> createObjects(ResultSet resultSet) {  
 List<T> list = **new** ArrayList<T>();  
 Constructor[] c = **type**.getDeclaredConstructors();  
 Constructor ct = **null**;  
 **for** (**int** i = 0; i < c.**length**; i++) {  
 ct = c[i];  
 **if** (ct.getGenericParameterTypes().**length** == 0)  
 **break**;  
 }  
 **try** {  
 **while** (resultSet.next()) {  
 ct.setAccessible(**true**);  
 T instance = (T) ct.newInstance();  
 **for** (Field field : **type**.getDeclaredFields()) {  
 String fieldName = field.getName();  
 Object obj = resultSet.getObject(fieldName);  
 PropertyDescriptor pD = **new** PropertyDescriptor(fieldName, **type**);  
 Method method = pD.getWriteMethod();  
 method.invoke(instance, obj);  
 }  
 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;  
}

*/\*\*  
 \* Metoda care construieste query-ul pentru INSERT  
 \*  
 \** ***@return*** *query-ul obtinut  
 \*/***private** String createInsertQuery() {  
 StringBuilder sb = **new** StringBuilder();  
 sb.append(**"INSERT INTO "**).append(**type**.getSimpleName()).append(**" VALUES ("**);  
 **int** nrAtribute = **type**.getDeclaredFields().**length**;  
 **for** (**int** i = 0; i <= nrAtribute - 2; i++)  
 sb.append(**"?, "**);  
 sb.append(**"?)"**);  
 **return** sb.toString();  
}

*/\*\*  
 \* Metoda care insereaza date intr-o tabela  
 \*  
 \** ***@param t*** *obiectul care trebuie inserat  
 \** ***@return*** *obiectul inserat  
 \*/***public** T insert(T t) {  
 Connection connection = **null**;  
 PreparedStatement statement = **null**;  
 ResultSet resultSet = **null**;  
 **try** {  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(**this**.createInsertQuery());  
 **int** index = 1;  
 **for** (Field f : **type**.getDeclaredFields()) {  
 f.setAccessible(**true**);  
 **if** (f.get(t) **instanceof** Integer) {  
 statement.setInt(index, (Integer) f.get(t));  
 } **else if** (f.get(t) **instanceof** String) {  
 statement.setString(index, (String) f.get(t));  
 }  
 index++;  
 }  
 statement.executeUpdate();  
 **return** t;  
 } **catch** (SQLException | IllegalAccessException ex) {  
 ex.printStackTrace();  
 } **finally** {  
 ConnectionFactory.*close*(resultSet);  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
 **return null**;  
}

*/\*\*  
 \* Metoda care construieste query-ul pentru UPDATE  
 \*  
 \** ***@return*** *query-ul obtinut  
 \*/***private** String createUpdateQuery() {  
 StringBuilder sb = **new** StringBuilder();  
 sb.append(**"UPDATE "**).append(**type**.getSimpleName()).append(**" SET "**);  
 Field[] fields = **type**.getDeclaredFields();  
 **int** nrAtribute = fields.**length**;  
 **for** (**int** i = 1; i < nrAtribute - 1; i++) {  
 sb.append(fields[i].getName()).append(**" = ?, "**);  
 }  
 sb.append(fields[nrAtribute - 1].getName()).append(**" = ? WHERE id = ?"**);  
 **return** sb.toString();  
}

*/\*\*  
 \* Metoda care modifica date dintr-o tabela  
 \*  
 \** ***@param t*** *obiectul care trebuie modificat  
 \** ***@return*** *obiectul modificat  
 \*/***public** T update(T t) {  
 Connection connection = **null**;  
 PreparedStatement statement = **null**;  
 ResultSet resultSet = **null**;  
 **try** {  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(**this**.createUpdateQuery());  
 **int** index = 1;  
 Field[] fields = **type**.getDeclaredFields();  
 **for** (**int** i = 1; i <= fields.**length** - 1; i++) {  
 fields[i].setAccessible(**true**);  
 **if** (fields[i].get(t) **instanceof** Integer) {  
 statement.setInt(index, (Integer) fields[i].get(t));  
 } **else if** (fields[i].get(t) **instanceof** String) {  
 statement.setString(index, (String) fields[i].get(t));  
 }  
 index++;  
 }  
 fields[0].setAccessible(**true**);  
 statement.setInt(index, (Integer) fields[0].get(t));  
 statement.executeUpdate();  
 **return** t;  
 } **catch** (SQLException | IllegalAccessException ex) {  
 ex.printStackTrace();  
 } **finally** {  
 ConnectionFactory.*close*(resultSet);  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
 **return null**;  
}

Clasele **ClientDAO,ProductDAO,OrderDAO** extind clasa exemplificata mai sus.

**public** Client findById(**int** id) {  
 Connection connection = **null**;  
 PreparedStatement statement = **null**;  
 ResultSet resultSet = **null**;  
 String query = **"SELECT \* FROM client WHERE id = ?"**;  
 **try** {  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(query);  
 statement.setInt(1, id);  
 resultSet = statement.executeQuery();  
 **while** (resultSet.next()) {  
 String nume = resultSet.getString(2);  
 String prenume = resultSet.getString(3);  
 **return new** Client(id, nume, prenume);   
 }  
 } **catch** (SQLException e) {  
 e.printStackTrace();  
 } **finally** {  
 ConnectionFactory.*close*(resultSet);  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
 **return null**;  
}

Metoda "findById" din clasa dată are rolul de a căuta un client în baza de date pe baza unui ID specificat și de a returna un obiect Client corespunzător.

În implementarea metodei, se inițializează obiecte Connection, PreparedStatement și ResultSet pentru a interacționa cu baza de date. Se construiește interogarea SQL utilizând parametri de întrebare pentru a filtra rezultatele după ID-ul dorit.

Se încearcă obținerea unei conexiuni la baza de date prin apelul metodei "getConnection()" din clasa ConnectionFactory. Se pregătește instrucțiunea SQL utilizând metoda "prepareStatement()" a obiectului de conexiune și se setează parametrul întrebării cu ID-ul specificat.

Se execută interogarea prin apelul metodei "executeQuery()" pe obiectul PreparedStatement și se obține rezultatul într-un obiect ResultSet.

Se creează și se returnează un nou obiect Client cu Id-ul, numele și prenumele extrase din rezultatul interogării. În cazul în care nu există niciun rezultat sau apare o excepție de tip SQLException și se închid obiectele ResultSet, PreparedStatement și Connection pentru a elibera resursele.

În final, în cazul în care nu se găsește niciun client cu ID-ul specificat sau apare o excepție, metoda returnează valoarea null.

Clasele **ClientValidator, OrderValidator, ProductValidator** verifica cu ajutorul metodelor datele clientului/ comenzii /produsului daca sunt in regula.

ClientValidator verifica daca s-au introdus date in campurile nume,prenume

*/\*\*  
 \* Validează atributele specifice unui client.  
 \*  
 \** ***@param client*** *Clientul pentru care se vor efectua validările.  
 \** ***@throws*** *Exception Dacă datele clientului sunt invalide.  
 \*/* @Override  
 **public void** validate(Client client) **throws** Exception {  
  
 **if** (client.getNume().equals(**""**)) {  
 **throw new** Exception(**"Numele clientului este invalid!"**);  
 }  
  
 **if** (client.getPrenume().equals(**""**)) {  
 **throw new** Exception(**"Prenumele clientului este invalid!"**);  
 }  
 }  
}

OrderValidator verifica daca respectiva cantitate este mai mare decat 1

*/\*\*  
 \* Validează atributele specifice unei comenzi.  
 \*  
 \** ***@param order*** *Comanda pentru care se vor efectua validările.  
 \** ***@throws*** *Exception Dacă datele comenzii sunt invalide.  
 \*/*@Override  
**public void** validate(Order order) **throws** Exception {  
  
 **if** (order.getCantitate() < 1) {  
 **throw new** Exception(**"Cantitatea comenzii este invalidă!"**);  
 }  
}

ProductValidator verifica sa fie introdus un nume al produsului,pretul si cantitatea sa nu fie mai mici decat 1

**public void** validate(Product product) **throws** Exception {  
  
 **if** (product.getNume().equals(**""**)) {  
 **throw new** Exception(**"Numele produsului este invalid!"**);  
 }  
  
 **if** (product.getPret() < 1) {  
 **throw new** Exception(**"Prețul produsului este invalid!"**);  
 }  
  
 **if** (product.getCantitate() < 1) {  
 **throw new** Exception(**"Cantitatea produsului este invalidă!"**);  
 }  
}

**ClientBLL,ProductBLL,OrderBLL**

ClientBLL,ProductBLL se aseamana foarte mult.

Clasa ClientBLL reprezintă un nivel de business logic (logica de afaceri) asociată cu entitatea Client. Această clasă are atribute private, cum ar fi un validator (validarea datelor pentru client) și un clientDAO (accesul la datele clientului în baza de date). Clasa include un constructor care primește aceste atribute și metode care facilitează conectarea logicii din spatele operațiilor din DAO cu baza de date.

Atributele private ale clasei ClientBLL, validatorul și clientDAO, sunt utilizate pentru a valida datele introduse de utilizator și pentru a efectua operațiuni(insert,update,delete)  
*/\*\*  
 \* Metoda în care se adaugă un client în tabelul Client.  
 \*  
 \** ***@param client*** *Clientul care trebuie adăugat în tabelul Client.  
 \** ***@return*** *Clientul care a fost adăugat.  
 \** ***@throws*** *Exception Dacă există deja un client cu același ID.  
 \*/***public** Client insert(Client client) **throws** Exception {  
 **this**.**validator**.validate(client);  
  
 **if** (**this**.**clientDAO**.findById(client.getId()) != **null**)  
 **throw new** Exception(**"Există deja un client cu acest ID!"**);  
  
 **return this**.**clientDAO**.insert(client);  
}

*/\*\*  
 \* Metoda în care se șterge un client din tabelul Client.  
 \*  
 \** ***@param client*** *Clientul care trebuie șters din tabelul Client.  
 \** ***@return*** *Clientul care a fost șters.  
 \** ***@throws*** *Exception Dacă clientul respectiv nu există.  
 \*/***public** Client delete(Client client) **throws** Exception {  
 **if** (**this**.**clientDAO**.findById(client.getId()) == **null**)  
 **throw new** Exception(**"Nu există acest client!"**);  
  
 Client rezultat = **this**.**clientDAO**.delete(client);  
  
 **return** rezultat;  
}

*/\*\*  
 \* Metoda în care se adaugă un produs în tabela Product.  
 \*  
 \** ***@param product*** *Produsul care trebuie să fie adăugat în tabela Product.  
 \** ***@return*** *Produsul care a fost adăugat.  
 \** ***@throws*** *Exception Dacă există deja un produs cu acel Id sau dacă datele produsului sunt invalide.  
 \*/***public** Product insert(Product product) **throws** Exception {  
 **this**.**validator**.validate(product);  
  
 **if** (**this**.**productDAO**.findById(product.getId()) != **null**)  
 **throw new** Exception(**"Există deja un produs cu acest Id!"**);  
  
 **return this**.**productDAO**.insert(product);  
}

*/\*\*  
 \* Metoda în care se șterge un produs din tabela Product.  
 \*  
 \** ***@param product*** *Produsul care trebuie să fie șters din tabela Product.  
 \** ***@return*** *Produsul care a fost șters.  
 \** ***@throws*** *Exception Dacă nu există produsul respectiv.  
 \*/***public** Product delete(Product product) **throws** Exception {  
 **if** (**this**.**productDAO**.findById(product.getId()) == **null**)  
 **throw new** Exception(**"Nu există acest produs!"**);  
  
 Product rezultat = **this**.**productDAO**.delete(product);  
  
 **return** rezultat;  
}

Clasa OrderBLL se aseamana si ea mult , dar este putin mai diferita , deoarece la momentul inserarii unui produs trebuie sa se verifice daca mai exista o comanda cu acelasi Id,daca exista acel client/produs si daca se insereaza o cantitate mai mica sau egala din stocul produsului, iar in final trebuie ca stocul sa fie actualizat.

*/\*\*  
 \* Metoda în care se adaugă o comandă în tabela Order.  
 \*  
 \** ***@param order*** *Comanda care trebuie să fie adăugată în tabela Order.  
 \** ***@return*** *Comanda care a fost adăugată.  
 \** ***@throws*** *Exception Dacă există deja o comandă cu acel Id sau dacă datele comenzii sunt invalide.  
 \*/***public** Order insert(Order order) **throws** Exception {  
 **this**.**validator**.validate(order); *// Validare comandă* **if** (**this**.**orderDAO**.findById(order.getId()) != **null**)  
 **throw new** Exception(**"Există deja o comandă cu acest Id!"**);  
  
 **if** (**this**.**clientDAO**.findById(order.getIdClient()) == **null**)  
 **throw new** Exception(**"Nu există acest client!"**);  
  
 **if** (**this**.**productDAO**.findById(order.getIdProdus()) == **null**)  
 **throw new** Exception(**"Nu există acest produs!"**);  
  
 Product product = **this**.**productDAO**.findById(order.getIdProdus());  
  
 **if** (order.getCantitate() > product.getCantitate())  
 **throw new** Exception(**"Nu există suficiente produse pe stoc pentru a se realiza comanda!"**);  
  
 product.setCantitate(product.getCantitate() - order.getCantitate());  
 **productDAO**.update(product);  
  
 **return this**.**orderDAO**.insert(order);  
}

**Clasele ClientView,OrderView si ProductView**

sunt responsabile de afișarea interfeței grafice utilizatorului. Ele conțin atributele necesare pentru casetele de text, etichetele corespunzătoare acestora și butoanele de adăugare, modificare, ștergere și afișare.

În constructorul claselor, sunt configurate și poziționate toate elementele interfeței grafice, cum ar fi casetele de text pentru introducerea datelor, etichetele care descriu fiecare câmp, precum și butoanele pentru acțiunile specifice.

Pe lângă metodele standard de acces (get()) și modificare (set()) pentru casetele de text, clasele View oferă și metode pentru adăugarea de listeners. Acești listeners sunt responsabili de verificarea dacă unul dintre butoanele de adăugare, modificare, ștergere sau afișare a fost apăsat. În cazul în care un astfel de eveniment are loc, se trimite un apel către controller pentru a afișa rezultatul corespunzător.

Exemple ClientView

**public class** ClientView **extends** JFrame {  
 **private** JTextField **idTextField**;  
 **private** JTextField **numeTextField**;  
 **private** JTextField **prenumeTextField**;  
 **private** JLabel **titleLabel**;  
 **private** JLabel **idLabel**;  
 **private** JLabel **numeLabel**;  
 **private** JLabel **prenumeLabel**;  
 **private** JLabel **dateLabel**;  
  
 **private** JTable **clientTable**;  
   
 **private** JScrollPane **tableScroll**;  
 **private** JButton **adaugaButton**;  
 **private** JButton **stergeButton**;  
 **private** JButton **modificaButton**;  
 **private** JButton **afisareButton**;

Exemple OrderView

*/\*\*  
 \* Metoda de get() pentru caseta ID  
 \*  
 \** ***@return*** *caseta ID  
 \*/***public** JTextField getIdTextField() {  
 **return idTextField**;  
}  
  
*/\*\*  
 \* Metoda de get() pentru caseta cantitate  
 \*  
 \** ***@return*** *caseta cantitate  
 \*/***public** JTextField getCantitateTextField() {  
 **return cantitateTextField**;  
}  
  
*/\*\*  
 \* Metoda de get() pentru tabel clienti  
 \*  
 \** ***@return*** *tabelul clienti  
 \*/***public** JTable getClientTable() {  
 **return clientTable**;  
}  
  
*/\*\*  
 \* Metoda de get() pentru tabel produse  
 \*  
 \** ***@return*** *tabelul produselor  
 \*/***public** JTable getProdusTable() {  
 **return produsTable**;  
}  
  
*/\*\*  
 \* Metoda de get() pentru tabelul comenzilor  
 \*  
 \** ***@return*** *tabelul comenzilor  
 \*/***public** JTable getOrderTable() {  
 **return orderTable**;  
}  
  
*/\*\*  
 \* Metoda de adaugare a unui listener pe butonul de afisare  
 \*  
 \** ***@param actionListener*** *actionListener  
 \*/***public void** addAfisareListener(ActionListener actionListener) {  
  
 **this**.**afisareButton**.addActionListener(actionListener);  
}  
  
*/\*\*  
 \* Metoda de adaugare a unui listener pe butonul de adaugare  
 \*  
 \** ***@param actionListener*** *actionListener  
 \*/***public void** addAdaugareListener(ActionListener actionListener) {  
  
 **this**.**adaugaButton**.addActionListener(actionListener);  
}

Exemple ProductView

*// Marginile ferestrei interfetei***this**.setBounds(200, 200, 800, 500);  
**this**.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);  
**this**.getContentPane().setLayout(**null**);  
  
*// Etichete***titleLabel** = **new** JLabel(**"Produs"**);  
**titleLabel**.setFont(**new** Font(**"Serif"**, Font.***PLAIN***, 15));  
**titleLabel**.setHorizontalAlignment(SwingConstants.***CENTER***);  
**titleLabel**.setBounds(300, 20, 200, 50);  
**titleLabel**.setForeground(Color.***BLUE***);  
**this**.getContentPane().add(**titleLabel**);

**numeTextField** = **new** JTextField();  
**numeTextField**.setColumns(10);  
**numeTextField**.setBounds(160, 150, 150, 30);  
**this**.getContentPane().add(**numeTextField**);  
  
**pretTextField** = **new** JTextField();  
**pretTextField**.setColumns(10);  
**pretTextField**.setBounds(160, 200, 150, 30);  
**this**.getContentPane().add(**pretTextField**);

*// Tabel***produsTable** = **new** JTable();  
**this**.**tableScroll** = **new** JScrollPane(**this**.**produsTable**, JScrollPane.***VERTICAL\_SCROLLBAR\_ALWAYS***, JScrollPane.***HORIZONTAL\_SCROLLBAR\_ALWAYS***);  
**this**.**tableScroll**.setBounds(350, 100, 400, 250);  
**this**.**tableScroll**.setViewportView(**this**.**produsTable**);  
**this**.getContentPane().add(**this**.**tableScroll**);  
  
*// Butoane***adaugaButton** = **new** JButton(**"Adauga"**);  
**adaugaButton**.setFont(**new** Font(**"Serif"**, Font.***PLAIN***, 15));  
**adaugaButton**.setBounds(50, 300, 120, 40);  
**this**.getContentPane().add(**adaugaButton**);

**cantitateLabel** = **new** JLabel(**"Cantitate:"**);  
**cantitateLabel**.setHorizontalAlignment(SwingConstants.***CENTER***);  
**cantitateLabel**.setFont(**new** Font(**"Serif"**, Font.***PLAIN***, 15));  
**cantitateLabel**.setBounds(50, 250, 100, 30);  
**cantitateLabel**.setForeground((Color.***yellow***));  
**this**.getContentPane().add(**cantitateLabel**);

Clasa **ClientController/ProductController/OrderController**  este responsabilă de gestionarea acțiunilor utilizatorului din interfața grafică și de comunicarea cu componentele View și ClientBLL/ProductBLL/OrderBLL . Această clasă primește un apel când utilizatorul apasă unul dintre butoanele de pe interfață, executând operația corespunzătoare și returnând rezultatul în tabel (în cazul butonului Afisare) sau afișând un mesaj corespunzător (pentru celelalte butoane).

Constructorul clasei primește ca parametri un obiect View și un obiect ClientBLL. Acesta are rolul de a conecta butoanele din interfață cu metodele care apelează operațiile corespunzătoare în ClientBLL.

Clasa ClientController/ProductController/OrderController asociază butoanele din interfață cu metode specifice, astfel încât atunci când utilizatorul apasă un buton, se declanșează metoda corespunzătoare din ClientController/ProductController/OrderController . Această metodă, la rândul ei, apelează metodele corespunzătoare din obiectul ClientBLL pentru a efectua operațiile dorite.

Exemple din clasa ClientController

*/\*\*  
 \* Clasa pentru implementarea inserarii datelor in tabela Client in interfata corespunzatoare  
 \*/***class** AdaugareListener **implements** ActionListener {  
  
 @Override  
 **public void** actionPerformed(ActionEvent e) {  
 **try** {  
 **int** id = Integer.*parseInt*(**view**.getIdTextField().getText());  
 String nume = **view**.getNumeTextField().getText();  
 String prenume = **view**.getPrenumeTextField().getText();  
 Client client = **new** Client(id, nume, prenume);  
 **clientBLL**.insert(client);  
 JOptionPane.*showMessageDialog*(**null**, **" Client adaugat**

**!"**);  
 } **catch** (NumberFormatException ex) {  
 JOptionPane.*showMessageDialog*(**null**, **"ID invalid!"**);  
 } **catch** (Exception ex) {  
 JOptionPane.*showMessageDialog*(**null**, ex.getMessage());  
 }  
 }  
}

OrderController

*/\*\*  
 \* Clasa pentru implementarea inserarii datelor in tabela Order in interfata corespunzatoare  
 \*/***class** AdaugareListener **implements** ActionListener {  
 @Override  
 **public void** actionPerformed(ActionEvent e) {  
 **try** {  
 **int** id = Integer.*parseInt*(**view**.getIdTextField().getText());  
 **int** cantitate = Integer.*parseInt*(**view**.getCantitateTextField().getText());  
 **int** randSelectatClienti = **view**.getClientTable().getSelectedRow();  
 **int** randSelectatProduse = **view**.getProdusTable().getSelectedRow();  
 String idClientString = **view**.getClientTable().getModel().getValueAt(randSelectatClienti, 0).toString();  
 String idProdusString = **view**.getProdusTable().getModel().getValueAt(randSelectatProduse, 0).toString();  
 **int** idClient = Integer.*parseInt*(idClientString);  
 **int** idProdus = Integer.*parseInt*(idProdusString);  
 Order order = **new** Order(id, idClient, idProdus, cantitate);  
 **orderBLL**.insert(order);  
 JOptionPane.*showMessageDialog*(**null**, **"Adaugare efectuata cu succes!"**);  
 } **catch** (NumberFormatException ex) {  
 JOptionPane.*showMessageDialog*(**null**, **"Id sau Cantitate invalida!"**);  
 } **catch** (Exception ex) {  
 JOptionPane.*showMessageDialog*(**null**, ex.getMessage());  
 }  
 }  
}

ProductController

- constructorul primește ca parametru view-ul și productBLL, iar acesta are rolul de a lega butoanele de metoda care apelează operația corepunzătoare.

*/\*\*  
 \* Constructor pentru clasa ProductController  
 \*  
 \** ***@param view*** *View-ul pentru clienti  
 \** ***@param productBLL*** *Bll-ul pentru produse  
 \*/***public** ProductController(ProductView view, ProductBLL productBLL) {  
 **this**.**view** = view;  
 **this**.**productBLL** = productBLL;  
 **this**.**view**.addAfisareListener(**new** AfisareListener());  
 **this**.**view**.addAdaugareListener(**new** AdaugareListener());  
 **this**.**view**.addStergereListener(**new** StergereListener());  
 **this**.**view**.addModificareListener(**new** ModificareListener());  
}

Metoda care foloseste tehnica de reflexie pentru a afisa tabelul  
*/\*\*  
 \* Metoda pentru creearea capurilor de tabel  
 \*  
 \** ***@param objectList*** *Lista de obiecte din care se va crea capul de tabel  
 \** ***@param table*** *Tabelul obtinut  
 \*/***private void** reflectionTableHeaders(List<Object> objectList, JTable table) {  
 DefaultTableModel tableModel = **new** DefaultTableModel();  
  
 **if** (!objectList.isEmpty()) {  
 Object obj = objectList.get(0);  
 Field[] fields = obj.getClass().getDeclaredFields();  
  
 **for** (Field field : fields) {  
 field.setAccessible(**true**);  
 String fieldName = field.getName();  
 tableModel.addColumn(fieldName);  
 }  
  
 **for** (Object o : objectList) {  
 List<Object> rowData = **new** ArrayList<>();  
  
 **for** (Field field : fields) {  
 field.setAccessible(**true**);  
 **try** {  
 Object value = field.get(o);  
 rowData.add(value);  
 } **catch** (IllegalAccessException ex) {  
 ex.printStackTrace();  
 }  
 }  
 tableModel.addRow(rowData.toArray());  
 }  
 }  
  
 table.setModel(tableModel);  
 table.setShowGrid(**true**);  
 table.setGridColor(Color.***BLACK***);  
}

Clasa **ConnectionFactory** face legatura cu baza de date a carei date au fost scrise

Aici trebuie scrise datele bazei de date pentru a creea o legatura de succes

**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/schooldb"**;  
**private static final** String ***USER*** = **"root"**;  
**private static final** String ***PASS*** = **"Cluj01veh"**;

Cateva secvente de cod care afiseaza si erori in caz de insucces

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

Ultima clasa este Main

Clasa Main este punctul de intrare în program, în care se execută aplicația. Aceasta instantiază obiectele necesare pentru fiecare entitate (Client, Product, Order) și le asociază cu validatorii, DAO-urile și BLL-urile corespunzătoare.

În metoda main(), se creează un obiect Validator<Client> pentru validarea datelor clientului și un obiect ClientDAO pentru accesul la datele clientului în baza de date. Aceste obiecte sunt apoi utilizate pentru a construi un obiect ClientBLL care conține logica de afaceri asociată entității Client. Un obiect ClientView este creat pentru afișarea interfeței grafice asociate clienților. În final, un obiect ClientController este creat, având asociate ClientView și ClientBLL.

Același proces se aplică și pentru entitățile Product și Order. Se creează obiecte Validator<Product> și Validator<Order> pentru validarea datelor, obiecte DAO corespunzătoare și obiecte BLL care conțin logica de afaceri. De asemenea, se creează obiecte View și Controller asociate cu fiecare entitate.  
  
*/\*\*  
 \* Clasa in care se va executa programul.  
 \*/***public class** Main {  
  
 */\*\*  
 \* Metoda in care se va executa programul.  
 \** ***@param args*** *Argumentele din linia de comanda.  
 \*/* **public static void** main(String[] args) {  
  
 Validator<Client> clientValidator = **new** ClientValidator();  
 ClientDAO clientDAO = **new** ClientDAO();  
 ClientBLL clientBLL = **new** ClientBLL(clientValidator, clientDAO);  
 ClientView clientView = **new** ClientView();  
 ClientController clientController = **new** ClientController(clientView, clientBLL);  
  
 Validator<Product> productValidator = **new** ProductValidator();  
 ProductDAO productDAO = **new** ProductDAO();  
 ProductBLL productBLL = **new** ProductBLL(productValidator, productDAO);  
 ProductView productView = **new** ProductView();  
 ProductController productController = **new** ProductController(productView, productBLL);  
  
 Validator<Order> orderValidator = **new** OrderValidator();  
 OrderDAO orderDAO = **new** OrderDAO();  
 OrderBLL orderBLL = **new** OrderBLL(orderValidator, clientDAO, productDAO, orderDAO);  
 OrderView orderView = **new** OrderView();  
 OrderController orderController = **new** OrderController(orderView, clientBLL, productBLL, orderBLL);  
  
 }  
}

5.Rezultate

Aplicatia functioneaza correct si implineste toate conditiile.

A screenshot of a message

Description automatically generated with medium confidence



A screenshot of a message

Description automatically generated with medium confidence

A screenshot of a message

Description automatically generated with medium confidence

6.Concluzii

Cu ajutorul acestui proiect am invatat mult mai multe lucruri in Java , dar si SQL. Mi-a placut sa integrez SQL in acest proiect si am dobandit foarte multe cunostinte in realizarea interfetelor , dar si in utilizarea metodelor care utilizeaza tehnicii de reflexie.Am invatat sa folosesc DAO-uri pentru a accesa si manipula datele, sa realizez clasele Validator pentru a ma asigura ca datele introduse sunt valide si pot sa le utilizez.

7.Bibliografie

<https://www.w3schools.com/sql/sql_intro.asp>

<https://dsrl.eu/courses/pt/materials/PT2023_A3_S2.pdf>

<https://docs.oracle.com/javaee/7/api/javax/validation/Validator.html>

<https://www.geeksforgeeks.org/business-logic-layer/>