DOCUMENTATIE

TEMA *3 ORDERS MANAGEMENT APP*

NUME STUDENT: Ricu Alexandru Razvan
GRUPA: 30224

CUPRINS

1. Obiectivul temei

Scopul proiectului este dezvoltarea unei aplicații de gestionare a comenzilor, care să faciliteze procesarea comenzilor clienților pentru un depozit. O baza de date relațională va fi utilizată pentru stocarea produselor, a clienților și a comenzilor, asigurând o gestionare eficientă a datelor.

- *Procesul începe cu analiza nevoilor utilizatorilor și stabilirea cerințelor (se va detalia în capitolul: 2. Analiza problemei, modelare, scenarii, cazuri de utilizare).*
- *Urmează proiectarea detaliată a arhitecturii software și a interfeței. (se va detalia în capitolul: 3. Proiectare)*
- *Implementarea se concentrează pe scrierea codului și integrarea funcționalităților. (se va detalia în capitolul: 4. Implementare)*
- *Testarea este esențială pentru asigurarea corectitudinii și stabilității aplicației. (scenariile pentru testare vor fi prezentate în capitolul: 5. Concluzii)*

Scopul final este dezvoltarea unei aplicații robuste și eficiente, care să permită gestionarea optimă a comenzilor clienților pentru depozit, utilizând baza de date relațională pentru stocarea și manipularea datelor.

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

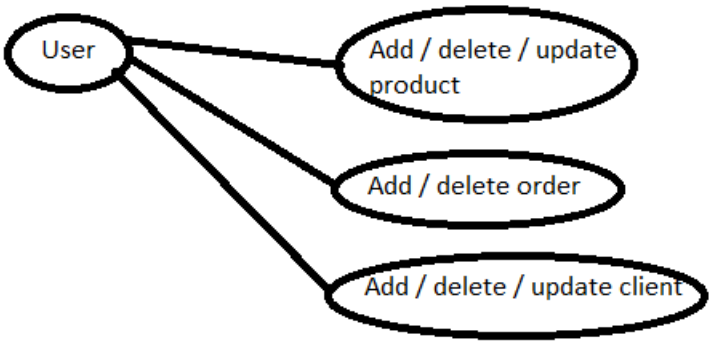
Cerințe Funcționale:

- Insert / Update / Delete Product: Utilizatorul poate introduce un nou produs, sa modifice unul existent sau sa stearga un produs.
- Insert / Update / Delete Client: Utilizatorul poate introduce un nou client, sa modifice unul existent sau sa stearga un client.
- Insert / Delete Order: Utilizatorul poate introduce o noua comanda, sa modifice una existenta sau sa stearga o comanda.
- Interfață Grafică: O interfață grafică intuitivă care permite utilizatorului să manipuleze informatii din baza de date si sa vizualizeze modificarile facute in diferitele tabele.

Cerințe Non-Funcționale:

- Performanță: Aplicația trebuie să fie rapidă și responsive la interacțiunile utilizatorului.

- Ușurință de Utilizare: Interfața grafică trebuie să fie prietenoasă și ușor de înțeles pentru utilizatori de toate nivelurile de experiență.
- Fiabilitate: Aplicația trebuie să fie robustă și să gestioneze corect diversele scenarii de utilizare.



Use-case-urile sunt prezentate mai jos, actorul principal al acestora fiind utilizatorul:

• Add / delete / update client:

Scenariu de succes:

a) Utilizatorul accesează interfața grafică a aplicației de gestionare a clientilor.

b) Utilizatorul introduce datele corespunzătoare clientului

c) Utilizatorul confirmă operatia asupra clientului, făcând clic pe butonul respectiv operatiei dorite.

d) Sistemul validează / înregistrează / modifica clientul în baza de date.

• Add / delete / update product:

Scenariu de succes:

a) Utilizatorul accesează interfața grafică a aplicației de gestionare a produselor.

b) Utilizatorul introduce datele corespunzătoare produsului

c) Utilizatorul confirmă operatia asupra produsului, făcând clic pe butonul respectiv operatiei dorite.

d) Sistemul validează / înregistrează / modifica produsul în baza de date.

• Add / delete order:

Scenariu de succes:

a) Utilizatorul accesează interfața grafică a aplicației de gestionare a comenzilor.

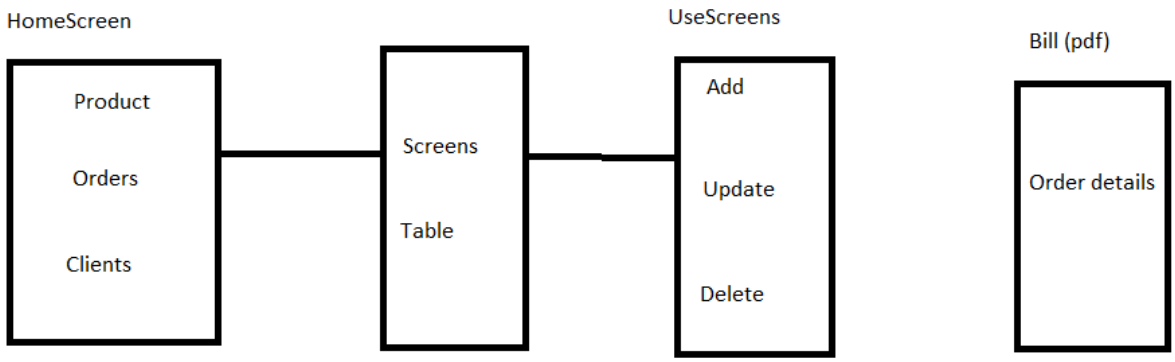
b) Utilizatorul introduce datele corespunzătoare comenzii

c) Utilizatorul confirmă operatia asupra comenzii, făcând clic pe butonul respectiv operatiei dorite.

d) Sistemul validează / înregistrează / modifica comanda în baza de date si genereaza un bill de tip pdf cu datele comenzii.

Scenariul alternativ pentru toate use-case-urile: nu se introduc date valide.

3. Proiectare



Am ales să implementez un model arhitectural Multi Layered Architecture. Acest model împarte aplicația în sapte componente distincte:

- Model (Order, Client, Product, Bill): Fiecare clasă modelează entitățile de bază implicate în gestionarea comenzilor pentru depozit. Clasa Order reprezintă o comandă plasată de un client pentru produsele din depozit. Ea conține informații despre produsele comandate, cantități, precum și detalii despre clientul care a plasat comanda. Clasa Client modelează un client care plasează comenzi în depozit. Ea conține informații precum nume, adresa și alte detalii relevante pentru identificarea și contactarea clientului. Clasa Product reprezintă un produs disponibil în depozit. Ea conține detalii despre produs, cum ar fi nume, descriere, preț și cantitate disponibilă în stoc. Clasa Bill reprezintă o factură generată pentru o comandă plasată de un client. Ea conține informații despre comanda asociată, precum și sumele totale pentru produsele comandate și alte taxe aplicabile.

- Presentation: Clasa HomeScreen: Afișează ecranul principal al aplicației și furnizează navigarea către alte funcționalități. Clasa ControllerHomeScreen: Gestionează interacțiunile utilizatorului pe ecranul principal și intializeaza paginile corespunzătoare tipului de date selectat prin butoane. Clasa Screens<T>: O clasă generică care afișează datele de tip T pe ecran si actiunile ce se pot face pe acestea. Această clasă poate fi specializată pentru diferite tipuri de date. Clasa ControllerScreens: Gestionează interacțiunile utilizatorului pe ecranele generice și inițiază acțiuni

corespunzătoare pentru tipul specific de date T. Clasa UseScreens<T>: O clasă generică care permite utilizatorului să interacționeze cu datele de tip T și să efectueze operații specific, insert / update / delete. Clasa ControllerUseScreens: Gestionează interacțiunile utilizatorului pe ecranele de utilizare generice și inițiază acțiuni corespunzătoare pentru tipul specific de date T. Clasa InsertScreen: O clasă care permite utilizatorului să introducă și să trimită date de tip Orders în aplicație. Clasa ControllerOrderScreen: Gestionează interacțiunile utilizatorului pe ecranul de inserare a comenzilor.

- BusinessLogicLayer: Clasa ClientBLL: Această clasă conține logica de afaceri asociată entității Client. Ea gestionează operațiile legate de manipularea datelor clientului, cum ar fi adăugarea, actualizarea, ștergerea și obținerea informațiilor despre clienți din baza de date. Clasa OrderBLL: Această clasă conține logica de afaceri asociată entității Order. Ea se ocupă de operațiile legate de gestionarea comenzilor, inclusiv adăugarea de noi comenzi, actualizarea stării comenzilor existente și generarea facturilor asociate. Clasa ProductBLL: Această clasă conține logica de afaceri asociată entității Product. Ea gestionează operațiile legate de manipularea datelor despre produse, inclusiv adăugarea, actualizarea, ștergerea și obținerea informațiilor despre produse din baza de date. Clasa BillBLL: Această clasă conține logica de afaceri asociată entității Bill. Ea se ocupă de operațiile legate de generarea facturilor pentru comenzile plasate de clienți.

- Dao: Clasa AbstractDAO: Această clasă abstractă servește ca șablon pentru clasele DAO specifice fiecărei entități. Ea conține metode generice pentru operațiile de bază de acces la date, cum ar fi adăugarea, actualizarea, ștergerea și obținerea datelor din baza de date. Clasa OrderDAO: Această clasă extinde clasa AbstractDAO și implementează operațiile specifice pentru entitatea Order. Ea conține metode pentru gestionarea comenzilor din baza de date, adaugarea acestora. Clasa ClientDAO: Similar cu OrderDAO, această clasă extinde AbstractDAO și se ocupă de operațiile specifice pentru entitatea Client. Clasa ProductDAO: Similar cu celelalte DAO-uri, această clasă extinde AbstractDAO și implementează operațiile specifice pentru entitatea Product. Clasa BillDAO: Această clasă extinde AbstractDAO se ocupă de operațiile specifice pentru entitatea Bill (Factură). Ea conține metode pentru manipularea datelor despre facturi în baza de date, inclusiv adăugarea.

- Connection (ConnectionFactory): Clasa ConnectionFactory este responsabilă pentru furnizarea și administrarea conexiunii la baza de date. Ea implementează un design pattern factory, permițând crearea și furnizarea eficientă a obiectelor de conexiune către alte clase care necesită acces la baza de date.

- Validators contine: Interfața Validator: Această interfață servește ca șablon pentru clasele de validare specifice. Ea definește o metodă generică de validare a datelor și poate fi implementată de clasele specifice pentru a verifica diverse aspecte ale datelor. Clasa ClientAgeValidator: Această clasă implementează Interfața Validator și este responsabilă pentru validarea vârstei clientului. Ea conține metode pentru a verifica dacă vârsta clientului se încadrează într-un anumit interval specificat de reguli de validare. Clasa EmailValidator: Similar cu ClientAgeValidator, această clasă implementează Interfața Validator și se ocupă de validarea adresei de email a clientului. Ea conține metode pentru a verifica dacă adresa de email respectă un format valid și poate fi utilizată pentru a trimite și primi corespondență. Clasa OrderValidator: Această clasă

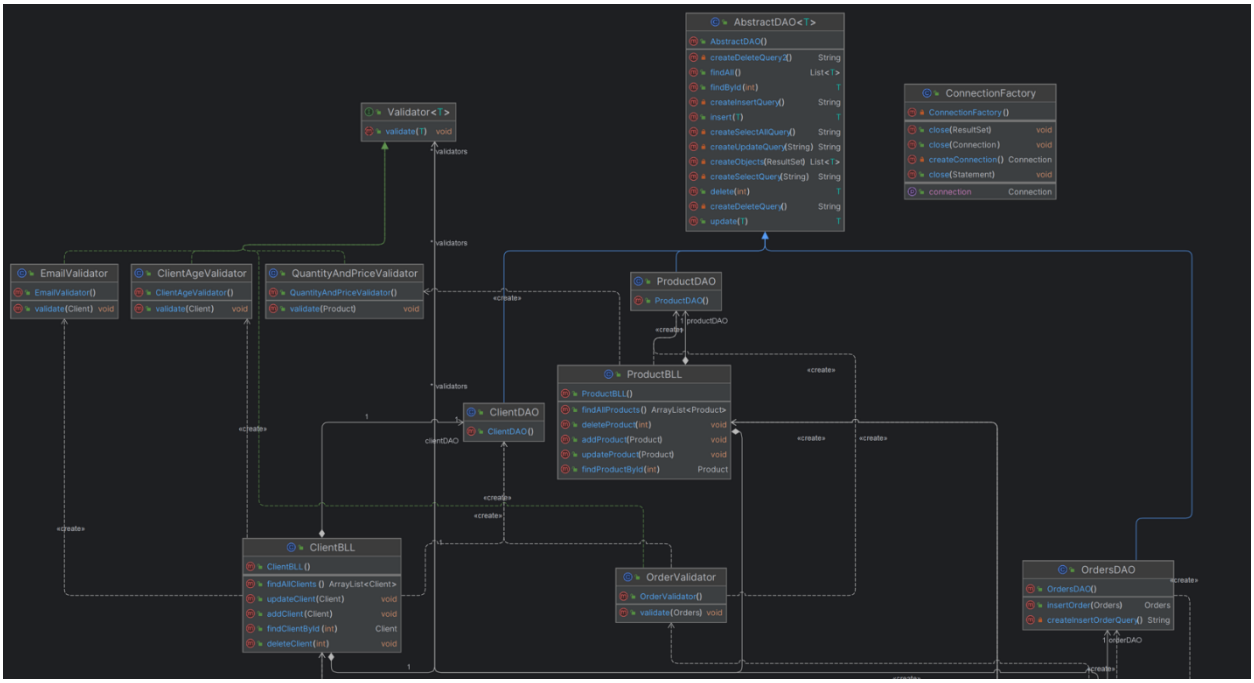
implementează Interfața Validator și este responsabilă pentru validarea comenzilor. Ea conține metode pentru a verifica diferite aspecte ale comenzilor, cum ar fi corectitudinea datelor introduse, existența produselor în stoc și altele. Clasa QuantityAndPriceValidator: Această clasă implementează Interfața Validator și se ocupă de validarea cantităților și prețurilor pentru produse. Ea conține metode pentru a verifica dacă cantitățile și prețurile introduse de utilizator sunt valide și corespund regulilor de validare specificate.

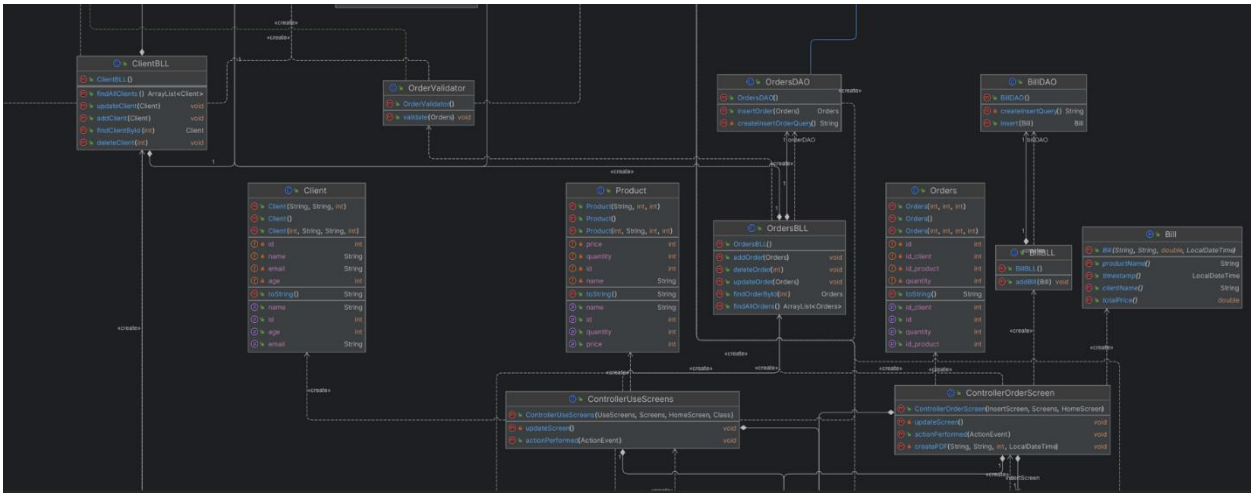
Prin implementarea acestui model arhitectural, putem gestiona eficient interacțiunea utilizatorului cu aplicația, separând clar business logic-ul (BusinessLogicLayerul), DAO-ul si Model-ul de interfața utilizatorului (Presentation) și gestionând intrările utilizatorului prin intermediul mai multor clase ce actioneaza ca si un Controler intre diferitele pagini ale interfetei.

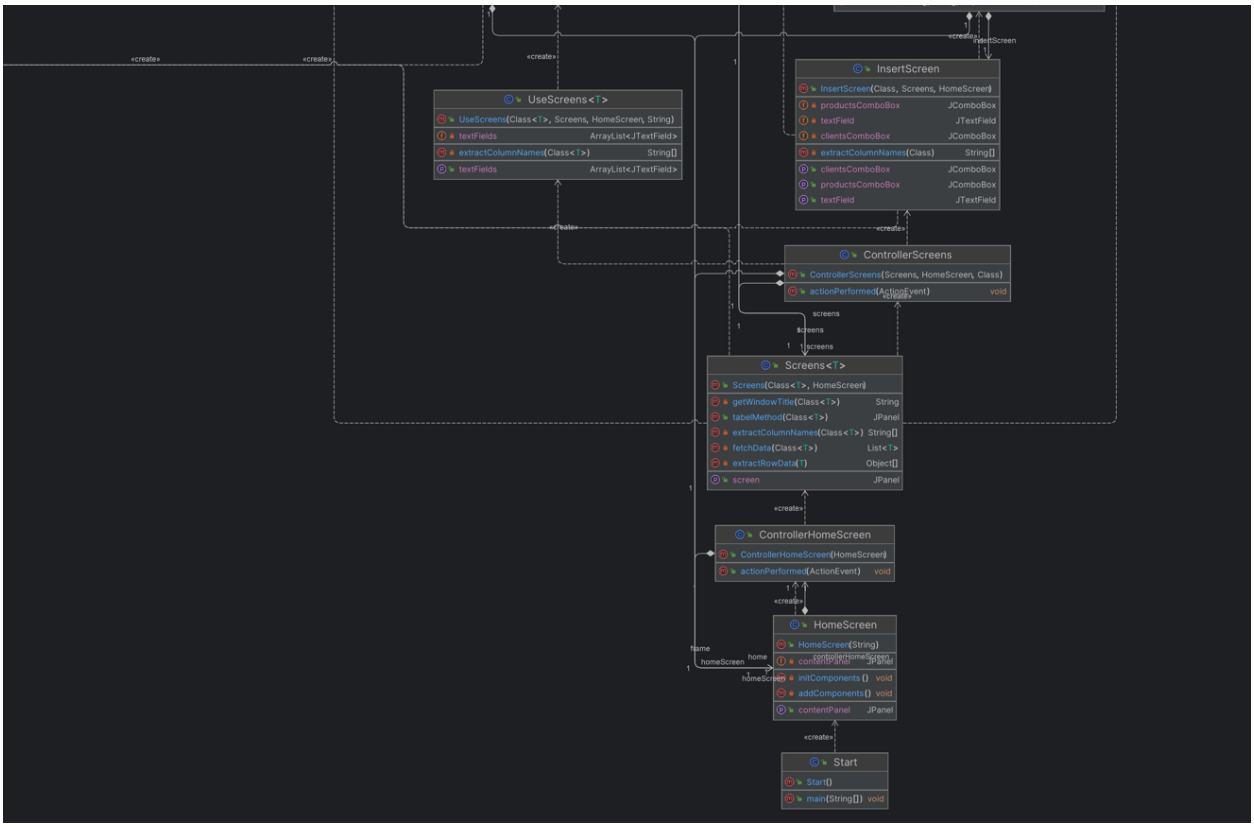
Pe lângă cele sase pachete menționate, am introdus și următorul pachet:

• Start: Clasa Main: În această clasă se găsește punctul de intrare în aplicație, de unde este deschis order management app. Clasa Main este responsabilă pentru inițializarea și lansarea aplicației.

Diagrama UML a proiectului este urmatoarea:







4. Implementare

Pentru a asigura o înțelegere completă a funcționalității Orders Management App, voi descrie detaliat clasele menționate în secțiunile anterioare:

- Clasa Client ;

Descriere: Această clasă reprezintă o entitate client într-un sistem. Definește atributele și funcționalitățile asociate unui client.

Campuri:

- `id`: Un număr întreg care identifică în mod unic clientul (privat).
- `nume`: Un șir de caractere care stochează numele clientului (privat).
- `email`: Un șir de caractere care stochează adresa de e-mail a clientului (privat).
- `vârsta`: Un număr întreg care stochează vârsta clientului (privat).

Constructori:

- **Constructor Implicit (`Client()`):** Acest constructor creează un obiect client gol, fără a initializa nicio valoare.
- **Constructor Parametrat cu Toate Câmpurile (`Client(int id, String nume, String email, int vârsta)`):** Acest constructor inițializează un obiect client cu valorile `id`, `nume`, `email` și `vârsta` furnizate.
- **Constructor Parametrat Fără id (`Client(String nume, String email, int vârsta)`):** Acest constructor inițializează un obiect client cu valorile `nume`, `email` și `vârsta` furnizate, dar lasă `id-ul` neasignat.

Metode:

- **Metode Getter:**
  - `getId()` - Returnează id-ul clientului ca număr întreg.
  - `getNume()` - Returnează numele clientului ca șir de caractere.
  - `getEmail()` - Returnează adresa de e-mail a clientului ca șir de caractere.
  - `getVârsta()` - Returnează vârsta clientului ca număr întreg.
- **Metode Setter:**
  - `setId(int id)` - Setează id-ul clientului la valoarea numărului întreg furnizat.
  - `setNume(String nume)` - Setează numele clientului la valoarea șirului de caractere furnizat.
  - `setEmail(String email)` - Setează adresa de e-mail a clientului la valoarea șirului de caractere furnizat.
  - `setVârsta(int vârsta)` - Setează vârsta clientului la valoarea numărului întreg furnizat.
- **Metodă Suprascrisă:**
  - `toString()` - Suprascrie metoda implicită `toString()` pentru a oferi o reprezentare sub forma unui șir de caractere a obiectului `Client`. Acest șir include informații despre `id`, `nume`, `email` și `vârsta` clientului.

- Clasa Bill:

Descriere: Clasa `Bill` reprezintă o factură pentru o tranzacție. Ea încapsulează informații despre client, produs, prețul total și data și ora tranzacției. Această clasă este imutabilă și folosește caracteristica record din Java 14 pentru a-și defini proprietățile în mod succint.

Campuri:

- `clientName`: Un șir de caractere care stochează numele clientului (final).
- `productName`: Un șir de caractere care stochează numele produsului (final).
- `totalPrice`: Un număr real care reprezintă prețul total al facturii (final).
- `timestamp`: Un obiect `LocalDateTime` care stochează data și ora tranzacției (final).

- Clasa Orders:

Descriere: Clasa `Orders` reprezintă o comandă plasată de un client pentru un produs. Ea încapsulează informații despre comandă, cum ar fi identificatorul unic al comenzii, identificatorul clientului, identificatorul produsului și cantitatea.

Campuri:

- `id`: Un număr întreg care reprezintă identificatorul unic al comenzii (privat).
- `id_client`: Un număr întreg care reprezintă identificatorul unic al clientului care plasează comanda (privat).
- `id_product`: Un număr întreg care reprezintă identificatorul unic al produsului comandat (privat).
- `quantity`: Un număr întreg care reprezintă cantitatea produsului comandat (privat).

Constructori:

- **Constructor Implicit (`Orders()`):** Acest constructor creează un obiect de comandă gol, fără a initializa nicio valoare.
- **Constructor Parametrat cu Toate Câmpurile (`Orders(int id, int id_client, int id_product, int quantity)`):** Acest constructor inițializează un obiect de comandă cu valorile `id`, `id_client`, `id_product` și `quantity` furnizate.
- **Constructor Parametrat Fără id (`Orders(int id_client, int id_product, int quantity)`):** Acest constructor inițializează un obiect de comandă cu valorile `id_client`, `id_product` și `quantity` furnizate, dar lasă `id-ul` neasignat.

Metode:

- **Metode Getter:**
  - `getId()` - Returnează identificatorul unic al comenzii.
  - `getId_client()` - Returnează identificatorul unic al clientului care a plasat comanda.
  - `getId_product()` - Returnează identificatorul unic al produsului comandat.
  - `getQuantity()` - Returnează cantitatea produsului comandat.
- **Metode Setter:**
  - `setId(int id)` - Setează identificatorul unic al comenzii.
  - `setId_client(int id_client)` - Setează identificatorul unic al clientului care a plasat comanda.

    - `setId_product(int id_product)` - Setează identificatorul unic al produsului comandat.
    - `setQuantity(int quantity)` - Setează cantitatea produsului comandat.
- **Metodă Suprascrisă:**
    - `toString()` - Suprascrie metoda implicită `toString()` pentru a oferi o reprezentare sub forma unui șir de caractere a obiectului `Orders`. Acest șir include informații despre `id`, `id_client`, `id_product` și `cantitate`.

- Clasa Product:

Descriere: Clasa `Product` reprezintă un produs disponibil pentru vânzare. Ea încapsulează informații despre produs, cum ar fi identificatorul unic, numele, cantitatea disponibilă și prețul.

Campuri:

- `id`: Un număr întreg care reprezintă identificatorul unic al produsului (privat).
- `name`: Un șir de caractere care reprezintă numele produsului (privat).
- `quantity`: Un număr întreg care reprezintă cantitatea disponibilă a produsului (privat).
- `price`: Un număr întreg care reprezintă prețul produsului (privat).

Constructori:

- **Constructor Implicit (`Product()`):** Acest constructor creează un obiect produs gol, fără a initializa nicio valoare.
- **Constructor Parametrat cu Toate Câmpurile (`Product(int id, String name, int quantity, int price)`):** Acest constructor inițializează un obiect produs cu valorile `id`, `name`, `quantity`, și `price` furnizate.
- **Constructor Parametrat Fără id (`Product(String name, int quantity, int price)`):** Acest constructor inițializează un obiect produs cu valorile `name`, `quantity`, și `price` furnizate, dar lasă `id-ul` neasignat.

Metode:

- **Metode Getter:**
    - `getId()` - Returnează identificatorul unic al produsului.
    - `getName()` - Returnează numele produsului.
    - `getQuantity()` - Returnează cantitatea disponibilă a produsului.
    - `getPrice()` - Returnează prețul produsului.
- **Metode Setter:**
    - `setId(int id)` - Setează identificatorul unic al produsului.
    - `setName(String name)` - Setează numele produsului.
    - `setQuantity(int quantity)` - Setează cantitatea disponibilă a produsului.
    - `setPrice(int price)` - Setează prețul produsului.
- **Metodă Suprascrisă:**

- `toString()` - Suprascrie metoda implicită `toString()` pentru a oferi o reprezentare sub forma unui șir de caractere a obiectului `Product`. Acest șir include informații despre `id`, `name`, `quantity`, și `price`.

- Clasa ConnectionFactory:

Descriere: Clasa `ConnectionFactory` oferă metode pentru gestionarea conexiunilor la baza de date. Se ocupă de crearea conexiunilor, închiderea conexiunilor, instrucțiunilor SQL (statement) și seturilor de rezultate (resultSet).

Campuri:

- `LOGGER`: Un obiect `Logger` folosit pentru înregistrarea evenimentelor (private static final).
- `DRIVER`: Un șir de caractere care conține numele driverului JDBC pentru MySQL (private static final).
- `DBURL`: Un șir de caractere care conține URL-ul bazei de date (private static final).
- `USER`: Un șir de caractere care conține numele de utilizator pentru baza de date (private static final).
- `PASS`: Un șir de caractere care conține parola pentru baza de date (private static final).
- `singleInstance`: O instanță statică privată a clasei `ConnectionFactory` (singleton)

Constructor Privat:

- `ConnectionFactory()`: Acest constructor privat este apelat automat atunci când este creată prima instanță a clasei `ConnectionFactory`. Încarcă driverul bazei de date folosind metoda `Class.forName()`.

Metode Publice Statice:

- `getConnection()`: Returnează o conexiune la baza de date. Metoda apelează metoda `createConnection` pentru a crea o conexiune nouă dacă nu există deja una disponibilă.
- `close(Connection connection)`: Închide o conexiune la baza de date furnizată.
- `close(Statement statement)`: Închide o instrucțiune SQL (statement) furnizată.
- `close(ResultSet resultSet)`: Închide un set de rezultate (resultSet) furnizat.

Metode Private:

- `createConnection()`: Creează o conexiune nouă la baza de date utilizând informațiile de conexiune stocate în câmpurile clasei (`DBURL`, `USER`, `PASS`). În caz de eroare, se înregistrează evenimentul cu nivelul `WARNING` folosind obiectul `LOGGER`.

Clasa AbstractDAO:

Descriere: Clasa `AbstractDAO` oferă operațiuni generice de acces la date pentru entitățile din baza de date. Include metode pentru operațiuni CRUD (Creare, Citire, Actualizare, Ștergere) și maparea obiectelor. Clasa este generică, ceea ce înseamnă că poate fi utilizată cu diferite tipuri de entități (clase) care reprezintă tabele din baza de date.

Câmpuri:

- `LOGGER`: Un obiect `Logger` folosit pentru înregistrarea evenimentelor (private static final).
- `type`: Un obiect `Class<T>` care reprezintă tipul entității gestionate de DAO (private final).

Constructor:

- `AbstractDAO()`: Constructorul implicit al clasei. Se folosește tipul generic pentru a determina tipul real al entității DAO (`T`).

Metode Private:

- `createSelectQuery(String field)`: Creează o interogare SQL de tip SELECT pentru a recupera o entitate după un anumit câmp (field).
- `createSelectAllQuery()`: Creează o interogare SQL de tip SELECT pentru a recupera toate entitățile.
- `createObjects(ResultSet resultSet)`: Creează o listă de obiecte de tipul entității DAO (`T`) dintr-un ResultSet obținut de la baza de date. Folosește reflection pentru a apela constructori și metode setter pe obiecte.
- `createInsertQuery()`: Creează o interogare SQL de tip INSERT pentru a insera o nouă entitate în baza de date.
- `createUpdateQuery(String field)`: Creează o interogare SQL de tip UPDATE pentru a actualiza o entitate existentă în baza de date, specificând câmpul după care se face identificarea entității.
- `createDeleteQuery()`: Creează o interogare SQL de tip DELETE pentru a șterge o entitate din tabela Orders, folosind id-ul entității ca referință.
- `createDeleteQuery2()`: Creează o interogare SQL de tip DELETE pentru a șterge o entitate din tabela corespunzătoare tipului entității DAO (`T`), folosind id-ul entității ca referință.

Metode Publice:

- `findAll()`: Recuperează o listă cu toate entitățile de tipul entității DAO (`T`) din baza de date.
- `findById(int id)`: Recuperează o entitate de tipul entității DAO (`T`) din baza de date după id. Returnează entitatea găsită sau null dacă nu este găsită.
- `insert(T t)`: Inserează o nouă entitate de tipul entității DAO (`T`) în baza de date. Returnează entitatea inserată.

- `update(T t)`: Actualizează o entitate existentă de tipul entității DAO (`T`) în baza de date. Returnează entitatea actualizată.
- `delete(int id)`: Șterge o entitate de tipul entității DAO (`T`) din baza de date după id.

- Clasa BillDAO

Descriere: Clasa `BillDAO` oferă operațiuni de acces la date specifice pentru entitatea `Bill` (Factura). Se extinde pe clasa abstractă `AbstractDAO` pentru a moșteni funcționalitatea CRUD generică pentru entități.

Câmpuri:

- `type`: Un obiect `Class` care reprezintă clasa `Bill` (private final).

Constructor:

- `BillDAO()`: Constructorul implicit al clasei. Setează câmpul `type` la clasa `Bill`.

Metode Private:

- `createInsertQuery()`: Creează o interogare SQL de tip INSERT pentru a insera o nouă factură (entitate `Bill`) în baza de date.

Metode Publice:

- `insert(Bill t)`: Inserează o nouă factură (entitate `Bill`) în baza de date. Folosește metoda moștenită `createInsertQuery` pentru a genera interogarea SQL și apoi execută interogarea pe baza de date. Returnează factura inserată.

- Clasa ClientDAO

Descriere: Clasa `ClientDAO` extinde clasa abstractă `AbstractDAO` și oferă operațiuni specifice de acces la date pentru entitatea `Client`.

Moștenire:

- Clasa moștenește operațiunile CRUD (Create, Read, Update, Delete) generice din clasa `AbstractDAO`.

Constructor:

- `ClientDAO()`: Constructorul implicit al clasei. Nu necesită argumente deoarece clasa părinte (`AbstractDAO`) se ocupă de initializarea tipului entității (`Client`) folosind reflection.

- Clasa OrdersDAO

Descriere: Clasa `OrdersDAO` extinde clasa abstractă `AbstractDAO` și oferă operațiuni specifice de acces la date pentru entitatea `Orders` (comenzi).

Moștenire:

- Clasa moștenește operațiunile CRUD (Create, Read, Update, Delete) generice din clasa `AbstractDAO`.

Câmpuri:

- `type2`: Un obiect `Class<Orders>` care reprezintă clasa `Orders` (private final). Se obține din tipul generic al clasei utilizând reflection.
- `prod_id`: Un câmp pentru a stoca temporar id-ul produsului din comanda (private int).
- `order_quantity`: Un câmp pentru a stoca temporar cantitatea din comanda (private int).

Constructor:

- `OrdersDAO()`: Constructorul implicit al clasei. Se folosește reflection pentru a determina tipul real al entității DAO (`Orders`) din tipul generic al clasei părinte.

Metode Private:

- `createInsertOrderQuery()`: Creează o interogare SQL de tip INSERT pentru a insera o nouă comandă (entitate `Orders`) în baza de date.

Metode Publice:

- `insertOrder(Orders t)`: Inserează o nouă comandă (entitate `Orders`) în baza de date.
    - Pe lângă inserarea comenzii, actualizează și cantitatea produsului comandat.

- Clasa ProductDAO

Descriere: Clasa `ProductDAO` extinde clasa abstractă `AbstractDAO` și oferă operațiuni specifice de acces la date pentru entitatea `Product`.

Moștenire:

- Clasa moștenește operațiunile CRUD (Create, Read, Update, Delete) generice din clasa `AbstractDAO`.

Constructor:

- `ProductDAO()`: Constructorul implicit al clasei. Nu necesită argumente deoarece clasa părinte (`AbstractDAO`) se ocupă de initializarea tipului entității (`Product`) folosind reflection.

- Clasa BillBLL

Descriere: Clasa `BillBLL` (Business Logic Layer) se ocupă de logica de business legată de entitatea `Bill` (Factura). Oferă metode pentru a interactiona cu baza de date prin intermediul clasei `BillDAO` pentru operațiuni CRUD (Create, Read, Update, Delete).

Dependențe:

- Clasa depinde de clasa `BillDAO` pentru accesul la date.

Câmpuri:

- `billDAO`: Un obiect din clasa `BillDAO` folosit pentru a interactiona cu baza de date (private final).

Constructor:

- `BillBLL()`: Constructorul implicit al clasei. Initializează obiectul `billDAO`.

Metode Publice:

- `addBill(Bill bill)`: Adaugă o factură (entitate `Bill`) în baza de date. Apela metoda `insert` din clasa `billDAO` pentru a insera factura.

- Clasa ClientBLL

Descriere: Clasa `ClientBLL` (Business Logic Layer) se ocupă de logica de business legată de entitatea `Client`. Oferă metode pentru a interactiona cu baza de date prin intermediul clasei `ClientDAO` pentru operațiuni CRUD (Create, Read, Update, Delete). Separă logica de business de accesul la date.

Dependențe:

- Clasa depinde de clasa `ClientDAO` pentru accesul la date.
- Clasa folosește clase din pachetul `bll.validators` pentru validarea clientului.

Câmpuri:

- `validators`: O listă de obiecte din clasele de validare (Validator) a clientului (private final).

- `clientDAO`: Un obiect din clasa `ClientDAO` folosit pentru a interactiona cu baza de date (private final).

Constructor:

- `ClientBLL()`: Constructorul implicit al clasei.
  - Initializează lista de validatori cu validatori pentru email și vârsta clientului.
  - Initializează obiectul `clientDAO`.

Metode Publice:

- `findClientById(int id)`: Găsește un client după id. Aruncă o excepție (`NoSuchElementException`) dacă nu găsește clientul.
- `findAllClients()`: Recuperează toți clienții din baza de date. Returnează o listă cu toți clienții.
- `addClient(Client client)`: Adaugă un client nou în baza de date. Validează clientul înainte de inserare.
- `deleteClient(int id)`: Șterge un client din baza de date după id. Aruncă o excepție (`NoSuchElementException`) dacă nu găsește clientul.
- `updateClient(Client client)`: Actualizează un client existent în baza de date. Validează clientul înainte de actualizare. Aruncă o excepție (`NoSuchElementException`) dacă nu găsește clientul.

- Clasa OrdersBLL

Descriere: Clasa `OrdersBLL` (Business Logic Layer) se ocupă de logica de business legată de entitatea `Orders` (comenzi). Oferă metode pentru a interactiona cu baza de date prin intermediul clasei `OrdersDAO` pentru operațiuni CRUD (Create, Read, Update, Delete). Separă logica de business de accesul la date.

Dependențe:

- Clasa depinde de clasa `OrdersDAO` pentru accesul la date.
- Clasa folosește o clasă din pachetul `bll.validators` pentru validarea comenzii.

Câmpuri:

- `validators`: O listă de obiecte din clasa de validare (Validator) a comenzii (private final).
- `orderDAO`: Un obiect din clasa `OrdersDAO` folosit pentru a interactiona cu baza de date (private final).

Constructor:

- `OrdersBLL()`: Constructorul implicit al clasei.

- Initializează lista de validatori cu un validator pentru comanda.
- Initializează obiectul `orderDAO`.

Metode Publice:

- `findOrderById(int id)`: Găsește o comandă după id. Aruncă o excepție (`NoSuchElementException`) dacă nu găsește comanda.
- `findAllOrders()`: Recuperează toate comenzile din baza de date. Returnează o listă cu toate comenzile.
- `addOrder(Orders order)`: Adaugă o comandă nouă în baza de date. Validează comanda înainte de inserare.
- `deleteOrder(int id)`: Șterge o comandă din baza de date după id. Aruncă o excepție (`NoSuchElementException`) dacă nu găsește comanda.
- `updateOrder(Orders order)`: Actualizează o comandă existentă în baza de date. Validează comanda înainte de actualizare. Aruncă o excepție (`NoSuchElementException`) dacă nu găsește comanda.

- Clasa ProductBLL

Descriere: Clasa `ProductBLL` (Business Logic Layer) se ocupă de logica de business legată de entitatea `Product`. Oferă metode pentru a interactiona cu baza de date prin intermediul clasei `ProductDAO` pentru operațiuni CRUD (Create, Read, Update, Delete). Separă logica de business de accesul la date.

Dependențe:

- Clasa depinde de clasa `ProductDAO` pentru accesul la date.
- Clasa folosește o clasă din pachetul `bll.validators` pentru validarea produsului.

Câmpuri:

- `validators`: O listă de obiecte din clasa de validare (Validator) a produsului (private final).
- `productDAO`: Un obiect din clasa `ProductDAO` folosit pentru a interactiona cu baza de date (private final).

Constructor:

- `ProductBLL()`: Constructorul implicit al clasei.
  - Initializează lista de validatori cu un validator pentru cantitatea și prețul produsului.
  - Initializează obiectul `productDAO`.

Metode Publice:

- `findProductById(int id)`: Găsește un produs după id. Aruncă o excepție (`NoSuchElementException`) dacă nu găsește produsul.
- `findAllProducts()`: Recuperează toate produsele din baza de date. Returnează o listă cu toate produsele.
- `addProduct(Product product)`: Adaugă un produs nou în baza de date. Validează produsul înainte de inserare.
- `deleteProduct(int id)`: Șterge un produs din baza de date după id. Aruncă o excepție (`NoSuchElementException`) dacă nu găsește produsul.
- `updateProduct(Product product)`: Actualizează un produs existent în baza de date. Validează produsul înainte de actualizare. Aruncă o excepție (`NoSuchElementException`) dacă nu găsește produsul.

- Clasa HomeScreen

Descriere: Clasa `HomeScreen` reprezintă ecranul principal al aplicației. Oferă butoane pentru a naviga către diferite funcționalități ale aplicației.

Câmpuri:

- `clientsButton`: Un obiect JButton pentru butonul "Clienți" (private).
- `productsButton`: Un obiect JButton pentru butonul "Produse" (private).
- `ordersButton`: Un obiect JButton pentru butonul "Comenzi" (private).
- `contentPanel`: Un obiect JPanel care reprezintă panoul principal al ferestrei (private).
- `controllerHomeScreen`: Un obiect din clasa `ControllerHomeScreen` folosit pentru gestionarea evenimentelor butoanelor (private).

Constructor:

- `HomeScreen(String title)`: Constructorul clasei.
  - Setează titlul ferestrei.
  - Inițializează comportamentul la închidere (EXIT_ON_CLOSE).
  - Apeleaza metoda `initComponents` pentru inițializarea componentelor.
  - Adaugă ascultători pentru evenimentele butoanelor (se apelează metoda din controller).
  - Adaugă panoul principal la fereastră.
  - Ajustează dimensiunea ferestrei și o setează vizibilă.

Metode Private:

- `initComponents()`: Inițializează componentele ecranului principal.
  - Creează o instanță a controlerului (`ControllerHomeScreen`).
  - Creează panoul principal (`contentPanel`) cu layout GridBagLayout.
  - Creează butoanele pentru "Clienți", "Produse" și "Comenzi" și le adaugă în panoul principal.

    - Setează dimensiunea preferată a panoului principal.
- `getContentPanel()`: Returnează panoul principal al ferestrei.
- `addComponents()`: Adaugă panoul principal la containerul principal al ferestrei (`getContentPane`).

- Clasa Screens

Descriere: Clasa `Screens` este o clasă generică reutilizabilă care reprezintă un panou (`JPanel`) ce afișează date dintr-o tabelă a bazei de date. Oferă metode pentru preluarea datelor, extragerea numelor coloanelor și a datelor din rânduri și generarea unui tabel pentru vizualizare.

Parametrii Generici:

- `<T>`: Tipul de date ce urmează să fie afișat pe ecran.

Câmpuri:

- `dataList`: O listă ce conține datele preluate (private).
- `contentType`: Tipul clasei ce reprezintă datele afișate (private final).
- `topPanel`: Panoul de sus al ecranului ce conține butoane (private final).

Constructor:

- `Screens(Class<T> contentType, HomeScreen frame)`: Constructorul clasei.

Metode Private:

- `fetchData(Class<T> contentType)`: Prelucrează datele din baza de date în funcție de tipul conținutului.
    - Folosește o instrucțiune `switch` pentru a apela metoda corespunzătoareBLL (ProductBLL, ClientBLL sau OrdersBLL) pentru preluarea tuturor datelor.
- `getWindowTitle(Class<T> contentType)`: Generează titlul ferestrei în funcție de tipul conținutului.
    - Folosește o instrucțiune `switch` pentru a returna un titlu sugestiv ("Produse", "Clienți", "Comenzi").
- `tabelMethod(Class<T> contentType)`: Generează un panou (`JPanel`) ce conține un tabel (`JTable`) pentru afișarea datelor.
- `extractColumnNames(Class<T> contentType)`: Extrahe numele coloanelor din câmpurile clasei.
- `extractRowData(T item)`: Extrahe datele unui rând din obiectul de tip T.
- `getScreen()`: Reîmprospătează ecranul cu conținutul actualizat.

- Clasa UseScreens

Descriere: Clasa `UseScreens` reprezintă un panou (`JPanel`) utilizat pentru formulare de introducere sau ecrane de modificare asociate cu operațiuni din baza de date, cum ar fi inserare, actualizare sau ștergere. Oferă un layout generic pentru formulare cu etichete și câmpuri text.

Parametrii Generici:

- `<T>`: Tipul de date asociat cu ecranul.

Câmpuri:

- `textFields`: O listă ce conține toate câmpurile text din formular (private final).

Constructor:

- `UseScreens(Class<T> contentType, Screens screens, HomeScreen homeScreen, String command)`: Constructorul clasei.

Metode Private:

- `extractColumnNames(Class<T> contentType)`: Extrahe numele coloanelor din câmpurile clasei (similar cu clasa `Screens`).

Metode Publice:

- `getTextFields()`: Returnează lista de câmpuri text din formular.

- Clasa InsertScreen

Descriere: Clasa `InsertScreen` reprezintă un panou (`JPanel`) utilizat special pentru inserarea de date în sistem. Oferă un formular generic cu etichete și câmpuri text, combo box-uri pentru selectarea clientului și produsului din listele existente.

Câmpuri:

- `textField`: Un câmp text pentru introducerea unei cantități (private).
- `clientsComboBox`: Un combo box pentru selectarea clientului (private).
- `productsComboBox`: Un combo box pentru selectarea produsului (private).

Constructor:

- `InsertScreen(Class contentType, Screens screens, HomeScreen homeScreen)`: Constructorul clasei.

Metode Private:

- `extractColumnNames(Class contentType)`: Similar cu clasa `Screens`.

Metode Publice:

- `getTextField()`: Returnează câmpul text pentru cantitate.
- `getClientsComboBox()`: Returnează combo box-ul pentru selectarea clientului.
- `getProductsComboBox()`: Returnează combo box-ul pentru selectarea produsului.

- ControllerHomeScreen

Câmpuri:

- `home`: Referință către ecranul principal (`HomeScreen`) (private final).

Constructor:

- `ControllerHomeScreen(HomeScreen home)`: Constructorul clasei.
  - Primește ca parametru ecranul principal (`HomeScreen`) și îl stochează într-un câmp.

Metode Publice:

- `actionPerformed(ActionEvent e)`: Această metodă este apelată automat când are loc o acțiune pe un buton (`JButton`) din cadrul ecranului principal.

- ControllerScreens

Câmpuri:

- `screens`: Referință către ecranul curent de tip `Screens` (private final).
- `frame`: Referință către ecranul principal (`HomeScreen`) (private final).
- `contentType`: Tipul de date asociat ecranului curent (private final).

Constructor:

- `ControllerScreens(Screens screens, HomeScreen frame, Class contentType)`: Constructorul clasei.
  - Primește ca parametri ecranul curent (`screens`), ecranul principal (`HomeScreen`) și tipul de date asociat ecranului (`contentType`) și le stochează în câmpuri separate.

Metode Publice:

- `actionPerformed(ActionEvent e)`: Această metodă este apelată automat când are loc o acțiune pe un buton (`JButton`) din cadrul ecranului curent.

- ControllerOrderScreen

Câmpuri:

- `screens`: Referință către ecranul curent de tip `Screens` (private final).
- `insertScreen`: Referință către ecranul pentru inserarea comenzilor (`InsertScreen`) (private final).
- `homeScreen`: Referință către ecranul principal (`HomeScreen`) (private final).

Constructor:

- `ControllerOrderScreen(InsertScreen insertScreen, Screens screens, HomeScreen homeScreen)`: Constructorul clasei.
    - Primește ca parametri ecranul pentru inserarea comenzilor (`insertScreen`), ecranul curent (`screens`) și ecranul principal (`homeScreen`) și le stochează în câmpuri separate.

Metode Publice:

- `actionPerformed(ActionEvent e)`: Această metodă este apelată automat când are loc o acțiune pe un buton (`JButton`) din cadrul ecranului pentru inserarea comenzilor.

Metode Private:

- `createPDF(String client_name, String product_name, int totalPrice, LocalDateTime timestamp)`: Generează o factură PDF cu detaliile comenzii.
- `updateScreen()`: Actualizează interfața grafică a ecranului principal după finalizarea operațiunii.

- ControllerUseScreens

Câmpuri:

- `screens`: Referință către ecranul curent de tip `Screens` (private final).
- `homeScreen`: Referință către ecranul principal (`HomeScreen`) (private final).
- `contentType`: Tipul de date asociat ecranului curent (`contentType`) (private final).
- `textFields`: Listă de câmpuri text preluate din ecranul curent (`textFields`) (private final).

Constructor:

- `ControllerUseScreens(UseScreens useScreens, Screens screens, HomeScreen homeScreen, Class contentType)`: Constructorul clasei.

Metode Publice:

- `actionPerformed(ActionEvent e)`: Această metodă este apelată automat când are loc o acțiune pe un buton (`JButton`) din cadrul ecranului specific pentru operațiuni CRUD.

Metode Private:

- `updateScreen()`: Actualizează interfața grafică a ecranului principal după finalizarea operațiunii.

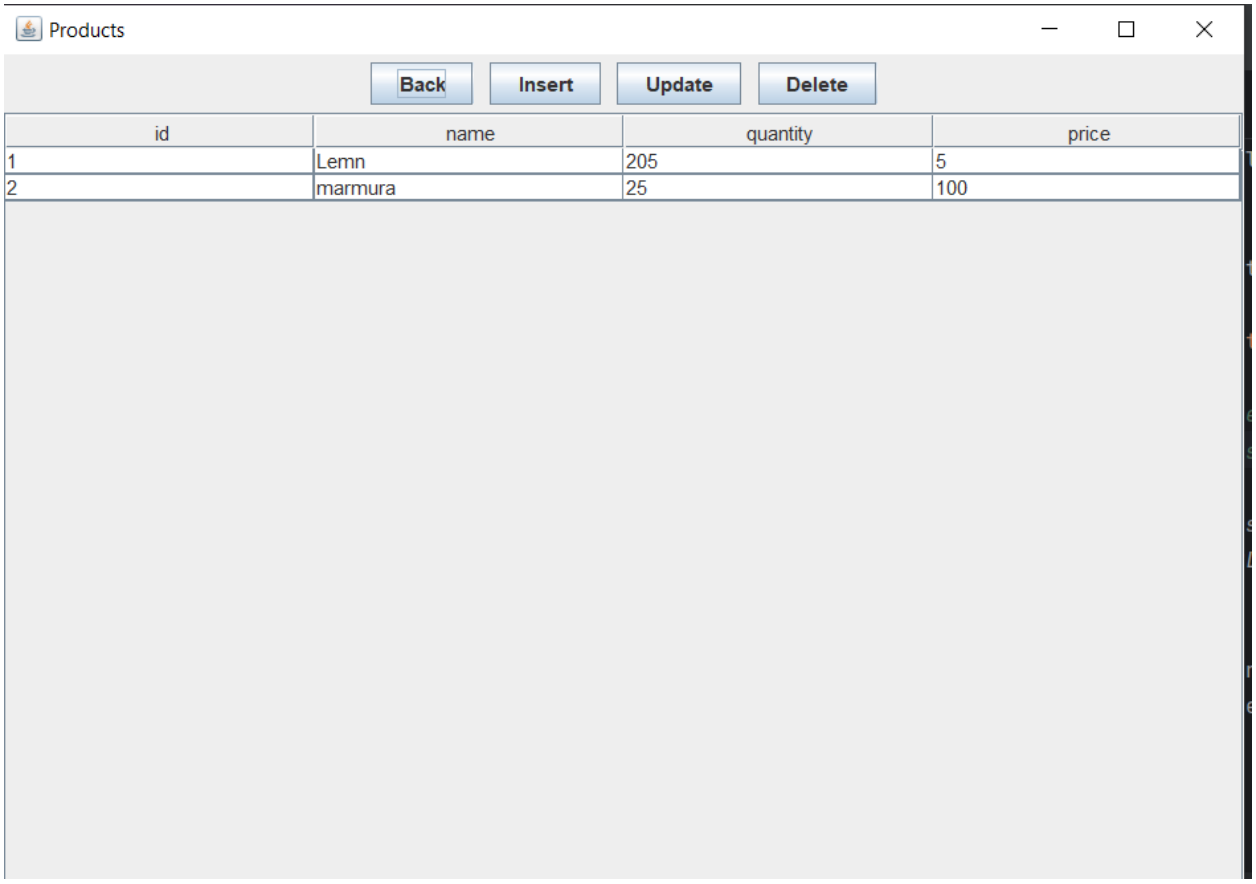
5. Rezultate

Testarea funcționalităților proiectului s-a desfășurat folosind operații pe date predefinite, care au fost concepute pentru a acoperi diverse scenarii și cazuri de utilizare. Aceste operații au implicat interacțiunea cu diferite funcționalități ale aplicației, cum ar fi adăugarea si ștergerea comenzilor, gestionarea stocurilor de produse și procesarea clienților.

Fiecare operație a fost concepută pentru a verifica coerența și corectitudinea funcționalităților în diverse situații de utilizare. De exemplu, în timpul testării adăugării comenzilor, s-au verificat aspecte precum validarea datelor introduse de utilizator, actualizarea corectă a stocurilor de produse și înregistrarea comenzilor în baza de date.

Rezultatele fiecărei operații au fost înregistrate în baza de date pentru a permite o analiză ulterioară și pentru a asigura integritatea datelor. Aceste înregistrări au inclus detalii despre operațiile efectuate, cum ar fi timpul de execuție, parametrii utilizați și eventualele erori sau excepții întâlnite în timpul procesării.

Afisarea datelor din baza de date sub forma unui tabel a facilitat procesul de corectare.



6. Concluzii

Experiența acumulată în timpul implementării acestui proiect a adus la lumină câteva concluzii și idei relevante, care pot fi de folos în viitoarele proiecte:

• Structurarea obiectivelor în sub-obiective a fost esențială pentru progresul eficient și gestionarea adecvată a proiectului. Împărțirea obiectivelor mari în etape mai mici și mai ușor de gestionat ne-a permis să ne concentrăm pe realizarea pașilor necesari pentru atingerea obiectivului final.

• Alegerea structurilor de date potrivite a avut un impact semnificativ asupra performanței și eficienței aplicației. Utilizarea adecvată a structurilor de date, cum ar fi bazele de date relaționale pentru stocarea și gestionarea comenzilor și produselor, a dus la o gestionare mai eficientă a informațiilor și a redus timpul necesar pentru procesarea comenzilor.

• Abstractizarea codului prin intermediul genericelor a permis o dezvoltare mai flexibilă și extensibilă a aplicației. Definirea unor interfețe clare pentru operațiile de bază și utilizarea

genericelor pentru a implementa comportamente comune au facilitat integrarea și extinderea funcționalităților fără a afecta structura existentă a codului.

• Menținerea lizibilității codului a fost un aspect crucial în dezvoltarea și întreținerea aplicației. Limitarea dimensiunii metodelor și claselor, precum și respectarea unor convenții de denumire și organizare a codului, au fost esențiale pentru înțelegerea și gestionarea eficientă a acestuia pe parcursul proiectului.

Bibliografie

1. *https://dsrl.eu/courses/pt/*
2. *https://mvnrepository.com/artifact/org.apache.pdfbox/pdfbox*
3. *https://www.baeldung.com/java-generics*