DOCUMENTAȚIE

TEMA 3

Nume student: Cotei Ruxandra-Maria

Grupa: 8

Cuprins

Obiectivul temei

Obiectivul principal al temei este de a dezvolta o aplicație de gestionare a comenzilor pentru un depozit/magazin, utilizând baze de date relaționale pentru stocarea produselor, clienților și comenzilor, conform unui model de arhitectură stratificată.

Obiective secundare:

- **Analiza cerințelor și specificațiilor**
    - Identificarea nevoilor și cerințelor utilizatorilor pentru aplicația de gestionare a comenzilor.
- **Proiectarea arhitecturii sistemului**
    - Crearea unui plan detaliat pentru structura și funcționalitățile sistemului, incluzând proiectarea claselor Model, Business Logic, Presentation și Data Access.
- **Dezvoltarea claselor Model**
    - Crearea claselor care reprezintă modelele de date ale aplicației, cum ar fi Product, Client și Order.
- **Dezvoltarea claselor Business Logic**
    - Implementarea logicii de afaceri a aplicației, incluzând gestionarea comenzilor, actualizarea stocurilor și procesarea plăților.
- **Dezvoltarea interfeței utilizator (Presentation)**
    - Realizarea interfeței grafice pentru utilizatori, incluzând pagini pentru plasarea, vizualizarea și gestionarea comenzilor.
- **Dezvoltarea claselor de acces la date (Data Access)**
    - Implementarea claselor care gestionează accesul la baza de date pentru operațiuni CRUD (Create, Read, Update, Delete).
- **Testarea și validarea sistemului**

- Realizarea testelor unitare și de integrare pentru a asigura funcționarea corectă a aplicației.

- **Documentarea sistemului**
  - Crearea documentației tehnice și a manualului de utilizare pentru aplicație.

Analiza problemei, modelare, scenarii, cazuri de utilizare

Analiza Problemei:

- **Cerințe Funcționale:**

  1. **Gestionarea Clienților:**
     - Adăugare, Editare, Ștergere, Vizualizare a clienților într-un tabel.
  2. **Gestionarea Produselor:**
     - Adăugare, Editare, Ștergere, Vizualizare a produselor într-un tabel.
  3. **Plasarea Comenzilor:**
     - Selectarea unui client existent și a unui produs existent.
     - Introducerea cantității dorite pentru produs.
     - Validarea stocului disponibil.
     - Actualizarea stocului și crearea comenzii.

- **Cerințe Non-Funcționale:**

  1. **Interfața Utilizator (UI):**
     - Interfață grafică intuitivă și ușor de utilizat.
     - Utilizarea de tabele pentru afișarea și manipularea datelor (JTable în Java).
  2. **Performanță:**
     - Răspuns rapid la acțiunile utilizatorului.
     - Eficiență în manipularea datelor și interacțiunea cu baza de date.

Cazuri de Utilizare:

1. **Adăugare Client:**
   - Utilizatorul introduce datele noului client în câmpurile corespunzătoare.
   - Apasă butonul "Adăugare".
   - Datele clientului sunt validate și salvate în baza de date.
2. **Editare Client:**
   - Utilizatorul selectează un client din tabel.
   - Modifică datele clientului în câmpurile corespunzătoare.

- Apasă butonul "Editare".
- Datele clientului sunt actualizate în baza de date.

3. **Ștergere Client:**
   - Utilizatorul selectează un client din tabel.
   - Apasă butonul "Ștergere".
   - Clientul este șters din baza de date.
4. **Vizualizare Client:**
   - Utilizatorul vizualizează lista de clienți într-un tabel.
5. **Plasare Comandă:**
   - Utilizatorul selectează un client și un produs existent.
   - Introduce cantitatea dorită pentru produs.
   - Apasă butonul "Plasare Comandă".
   - Se verifică disponibilitatea stocului.
   - Dacă stocul este suficient, se actualizează stocul și se creează comanda.

**Notă: aceleași operații ca la Client sunt efectuate și la Produs*

Proiectare

Proiectare OOP:

- **Diagrama UML de Clase:**
    - Prin intermediul acestei diagrame, se evidențiază structura modulară și relațiile între clase, ceea ce facilitează gestionarea și extinderea aplicației. Utilizarea conceptelor de încapsulare, moștenire și polimorfism contribuie la o arhitectură flexibilă și ușor de întreținut.
- **Diagrama UML de Pachete:**
    - Organizarea în pachete ajută la gestionarea și structurarea logică a codului, ceea ce duce la o dezvoltare mai eficientă și la o navigare mai ușoară în proiect. Modularizarea în pachete permite dezvoltatorilor să lucreze independent la diferite componente ale aplicației.

Structuri de Date Folosite:

- **Liste și Tablouri:**
    - Utilizarea adecvată a listelor și tablourilor contribuie la eficiența în stocarea și manipularea datelor. Listele dinamice pot fi redimensionate la nevoie, în timp ce tablourile sunt ideale pentru afișarea datelor în interfața grafică. Aceste structuri de date optimizează performanța aplicației, asigurând un timp de răspuns rapid pentru utilizatori.

Interfețe definite:

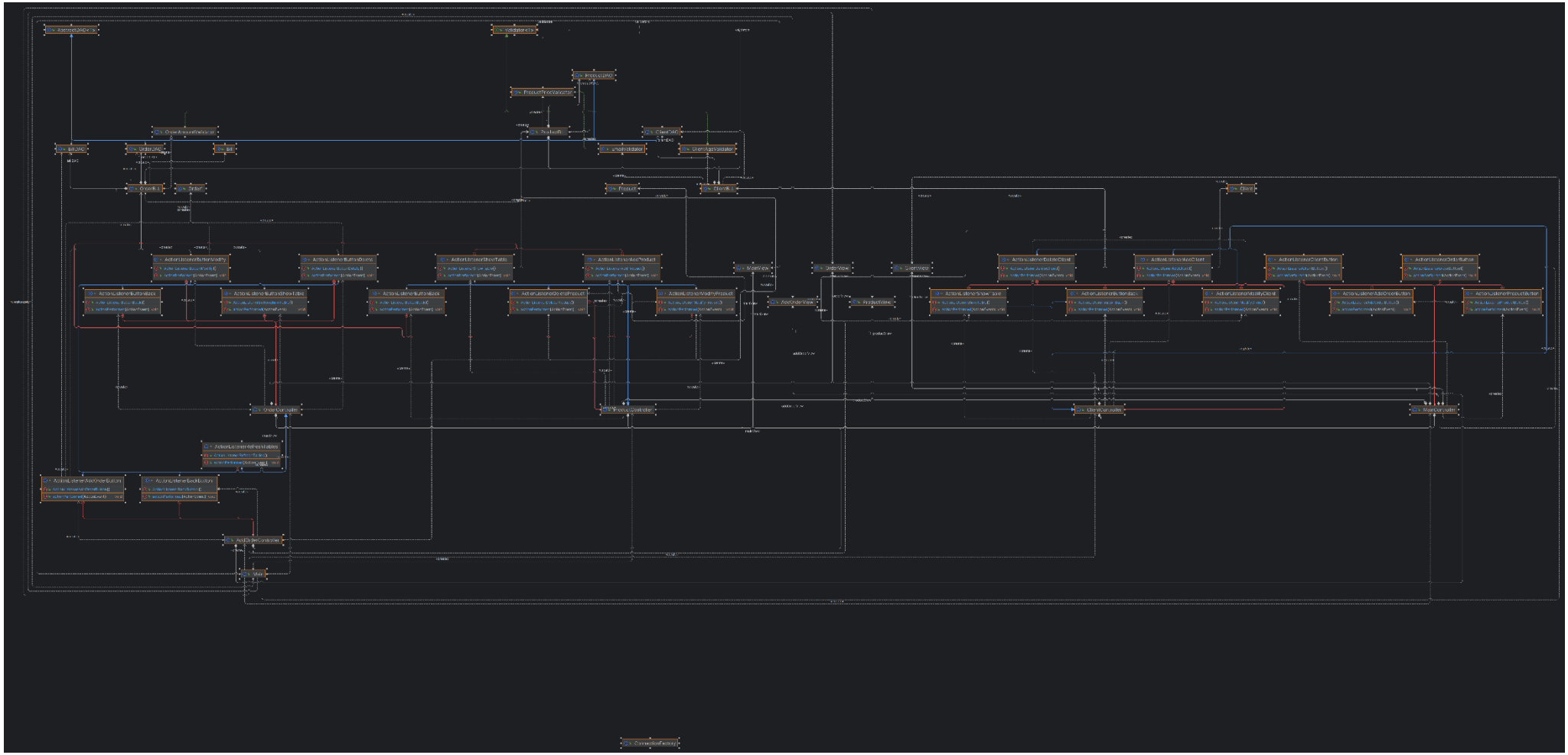
- **Interfața Utilizator (UI) Prietenoasă:**
    - Definirea unei interfețe grafice intuitive și prietenoase asigură o experiență plăcută utilizatorilor. Prin intermediul interfețelor grafice bine proiectate, utilizatorii pot interacționa eficient cu aplicația și pot accesa funcționalitățile dorite cu ușurință.

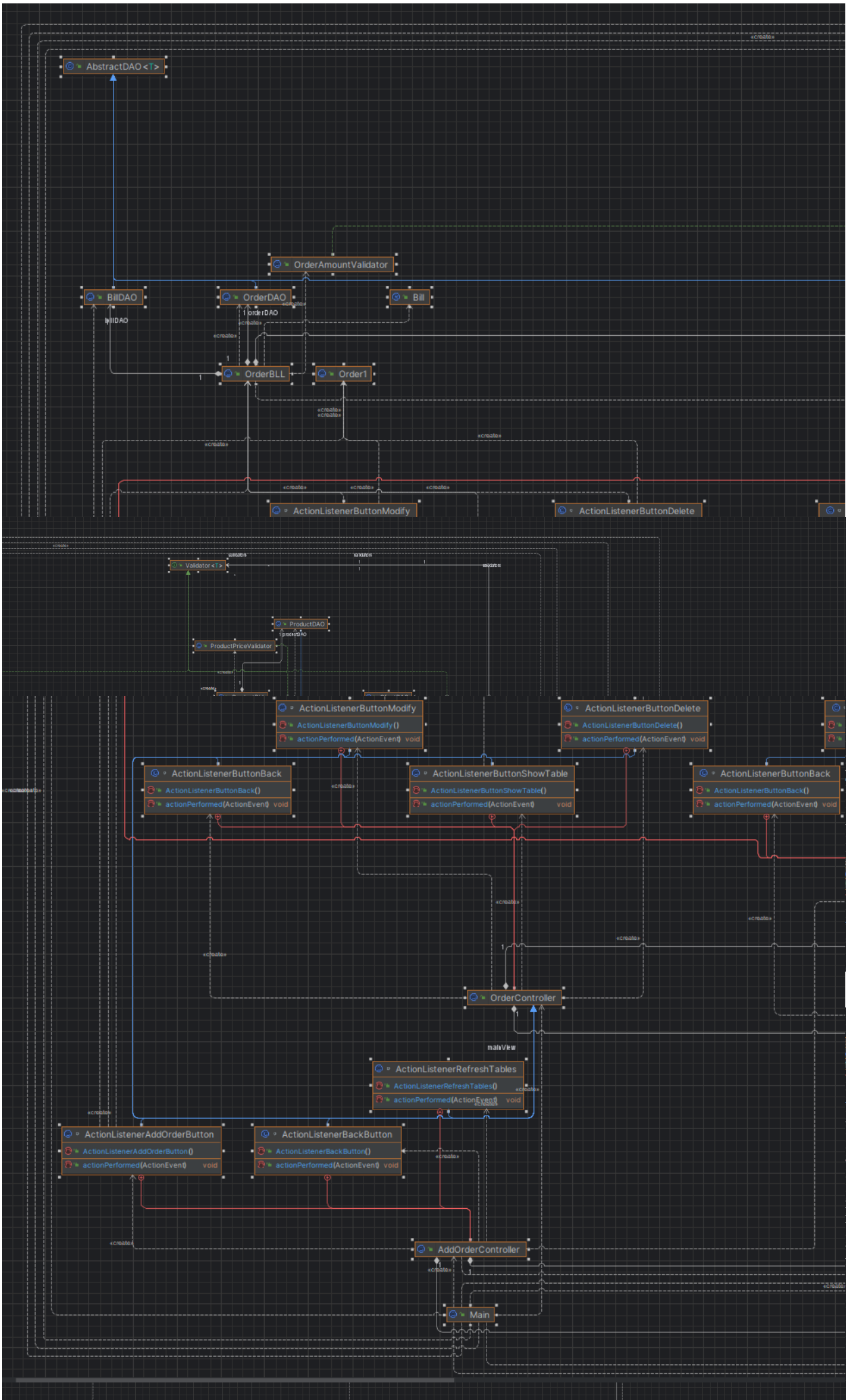
Algoritmi Folosiți:

- **Algoritmi de Validare Eficienți:**
    - Utilizarea algoritmilor de validare eficienți asigură integritatea datelor și minimizează erorile introduse de utilizatori. Validarea datelor înainte de salvarea lor în baza de date reduce necesitatea corecțiilor ulterioare și contribuie la creșterea calității aplicației.
- **Algoritmi de Procesare Optimizați:**
    - Algoritmii optimizați pentru procesarea comenzilor asigură o gestionare eficientă a stocului și a tranzacțiilor. Prin intermediul acestor algoritmi, aplicația poate răspunde rapid la cererile utilizatorilor și poate procesa un volum mare de date într-un timp scurt.

Aceste elemente de proiectare demonstrează nu doar eficiența, ci și avantajele pe care proiectarea OOP le aduce în dezvoltarea unei aplicații. Prin utilizarea unor structuri de date optimizate, interfețe prietenoase și algoritmi eficienți, aplicația devine ușor de întreținut, scalabilă și oferă o experiență plăcută utilizatorilor.

Implementare




AbstractDAO<T>
OrderAmountValidator
BillDAO
OrderDAO
Bill
OrderBLL
Order1
ActionListenerButtonModify
ActionListenerButtonDelete
Validator<T>
ProductDAO
ProductPriceValidator
ActionListenerButtonModify()
actionPerformed(ActionEvent) void
ActionListenerButtonDelete()
ActionListenerButtonBack()
ActionListenerButtonShowTable()
OrderController
mainView
ActionListenerRefreshTables()
ActionListenerAddOrderButton()
ActionListenerBackButton()
AddOrderController
Main

Descrierea claselor:

- **ClientBLL (businessLogic):**

  Metode Importante:

  - findClientById(int id): Caută un client după ID.
  - findAllClients(): Returnează o reprezentare a tuturor clienților.
  - insertClient(Client client): Inserează un nou client.
  - updateClient(Client client): Actualizează un client existent.
  - deleteClient(Client client): Șterge un client existent.

- **ProductBLL (businessLogic):**

  Metode Importante:

  - findProductById(int id): Găsește un produs după ID.
  - findAllProducts(): Returnează o reprezentare a tuturor produselor.
  - insertProduct(Product product): Inserează un nou produs.
  - updateProduct(Product product): Actualizează un produs existent.
  - deleteProduct(Product product): Șterge un produs existent.
  - decrementStock(int productId, int amount): Decrementarea stocului pentru un anumit produs.

- **OrderBLL (businessLogic):**

  Metode Importante:

  - createOrder(Order1 order1): Creează o comandă nouă.
  - updateOrder(Order1 order1): Actualizează o comandă existentă.
  - deleteOrder(Order1 order1): Șterge o comandă existentă.
  - checkStock(Order1 order1): Verifică stocul pentru o comandă dată.
  - processOrder(Order1 order1): Procesează o comandă, inclusiv verificarea stocului, crearea comenzii și facturarea.

Clasele din pachetul Model: (pentru fiecare dintre aceste clase, am creat tabele corespunzătoare în MySQL pentru a stoca informațiile relevante):

- **Client**: Un client este compus dintr-un ID unic, un nume, o adresă și o vârstă.
- **Produs**: Un produs este format dintr-un ID unic, un nume, un preț și o cantitate în stoc.
- **Comandă**: O comandă conține un ID unic, ID-ul produsului comandat, ID-ul clientului care plasează comanda și cantitatea de produse comandate.
- **Factură**: O factură este generată pentru o comandă specifică și conține un ID unic pentru factură, ID-ul comenzii asociate, suma totală și data la care a fost emisă factura.

Clasa AbstractDAO reprezintă un element central al structurii de acces la date din aplicație. Ea facilitează interacțiunea cu baza de date MySQL prin intermediul unor metode generice care pot fi folosite pentru orice clasă specifică. Prin utilizarea reflexiei în Java, această clasă poate opera cu orice tip de obiect, reducând astfel duplicarea codului și oferind o abordare flexibilă și scalabilă pentru manipularea datelor.

Principalele aspecte ale clasei sunt:

1. **Reflection**: în Java, este o caracteristică puternică care permite unui program Java să inspecteze și să manipuleze structurile sale la timpul de execuție. În esență, reflectia îi oferă programatorului posibilitatea de a analiza și de a modifica comportamentul și structura claselor, metodelor și câmpurilor în timpul rulării programului, în loc să fie determinate static la momentul compilării.

Iată câteva aspecte importante despre reflection și modul în care poate fi util:

- **Inspectarea Claselor și a Metodelor**: Reflection permite programatorului să obțină informații despre clase și metode, cum ar fi numele, tipurile de parametri, modificatorii de acces și multe altele. Acest lucru este util în situații în care aveți nevoie să operați cu clase și metode dinamic, fără a cunoaște detaliile acestora la momentul de compilare.
- **Crearea de Obiecte la Timpul de Execuție**: Reflection permite crearea de noi instanțe ale claselor la timpul de execuție. Aceasta este utilă atunci când trebuie să creați obiecte dinamic, bazate pe anumite condiții sau date primite în timpul rulării programului.

- **Accesarea și Modificarea Câmpurilor Private**: Reflection permite accesarea și modificarea câmpurilor private ale unei clase, ceea ce în mod normal nu ar fi posibil din codul exterior clasei. Acest lucru poate fi util în situații în care aveți nevoie să accesați sau să modificați valori private într-o clasă.
- **Implementarea de Framework-uri și Biblioteci Generice**: Reflection este folosit adesea în implementarea de framework-uri și biblioteci generice, unde codul trebuie să lucreze cu clase necunoscute la momentul compilării. Folosirea reflection-ului poate elimina redundanța și poate face codul mai flexibil și mai ușor de întreținut.

Cum am utilizat reflecția în clasa mea:

- În contextul clasei AbstractDAO, reflexia este folosită pentru a inspecta și manipula dinamic obiectele și metodele la timpul de execuție. Prin utilizarea reflexiei, AbstractDAO poate obține informații despre clase și obiecte, accesa și modifica câmpurile acestora, și apela metodele necesare pentru interacțiunea cu baza de date.
- De exemplu, atunci când interogăm baza de date pentru a recupera un obiect de tip T, AbstractDAO utilizează reflexia pentru a crea dinamic o instanță a clasei T și pentru a seta valorile câmpurilor acesteia în funcție de datele obținute din baza de date. Astfel, putem obține un obiect complet și actualizat în mod dinamic, fără a ști detalii specifice despre clasa T la momentul compilării.
- De asemenea, reflexia permite clasei AbstractDAO să ofere metode generice și reutilizabile pentru operații de bază CRUD (Create, Read, Update, Delete). Aceste metode pot fi apoi utilizate pentru orice clasă specifică, eliminând astfel necesitatea de a scrie cod suplimentar pentru fiecare tip de clasă.

2. **Metode Generice**: Metodele clasei sunt definite pentru a fi generice și pot fi folosite pentru orice clasă specifică, eliminând nevoia de a scrie metode separate pentru fiecare tip de clasă. Acest lucru face clasa extrem de utilă și eficientă, deoarece același set de metode poate fi folosit pentru a interacționa cu diferite clase de obiecte.

3. **Operațiuni CRUD**: Clasa oferă operații de bază CRUD (Create, Read, Update, Delete) pentru a interacționa cu baza de date. Aceste operații includ inserția, actualizarea, ștergerea și recuperarea obiectelor din baza de date.
4. **Utilizare în DAOs Specifice**: Clasa AbstractDAO servește ca o clasă părinte abstractă pentru clasele DAO specifice. Prin extinderea acestei clase și specificarea tipului de obiect cu care lucrează fiecare DAO, putem beneficia de toate funcționalitățile generice oferite de AbstractDAO.

În concluzie, clasa AbstractDAO reprezintă o componentă esențială a infrastructurii de acces la date a aplicației noastre. Utilizând reflexia în Java și metode generice, această clasă oferă un mod flexibil, eficient și scalabil de a interacționa cu baza de date MySQL, reducând astfel complexitatea și repetiția codului și facilitând gestionarea datelor în întreaga aplicație.

Implementarea Interfeței Utilizator:

Interfața utilizatorului este implementată utilizând o interfață grafică cu mai multe ferestre, fiecare dedicată operațiilor cu clienții, produsele și comenzile. Utilizatorul poate adăuga, edita și șterge clienți și produse folosind ferestre dedicate, iar pentru crearea unei comenzi, utilizatorul poate selecta un client existent, un produs și introduce o cantitate dorită pentru a crea o comandă validă. În cazul în care nu există suficiente produse în stoc pentru o comandă, se afișează un mesaj de sub-stoc. După finalizarea comenzii, stocul produsului este decrementat, iar o factură este generată.

Rezultate

Am adăugat comentarii adecvate în cod pentru a permite generarea documentației **JavaDoc** pentru metodele importante. Aceasta va facilita înțelegerea funcționării acestor metode și va oferi o documentație completă și clară pentru dezvoltatori.

De asemenea, am implementat funcționalitatea pentru generarea automată a unor **Fișiere text** în care se afișează detaliile facturii și ale comenzii corespunzătoare în formatul lor toString() după fiecare comandă efectuată. Această abordare oferă o modalitate eficientă de a monitoriza și de a înțelege fluxul de date și operațiile efectuate în cadrul aplicației, ușurând procesul de testare și de depanare.

Scenariile pentru testare pot include:

1. Adăugarea unui nou client și a unui nou produs în baza de date și verificarea corectitudinii acestora prin interogare manuală a bazei de date.
2. Crearea unei comenzi noi pentru un anumit client și un anumit produs și verificarea stocului produsului înainte și după efectuarea comenzii.
3. Actualizarea detaliilor unui client sau a unui produs existent și verificarea actualizării în baza de date.
4. Ștergerea unui client sau a unui produs existent și verificarea eliminării acestora din baza de date.
5. Generarea automată a facturii și a comenzii corespunzătoare pentru fiecare comandă efectuată și verificarea conținutului acestora în fișierele text create.

Aceste scenarii vor acoperi o gamă largă de funcționalități ale aplicației și vor asigura că aceasta funcționează corect și eficient în diferite situații de utilizare. De asemenea, permit testarea și validarea tuturor aspectelor importante ale aplicației noastre, inclusiv interacțiunea cu baza de date, operațiile CRUD, generarea automată a documentelor și multe altele.

Concluzii

În concluzie, tema m-a oferit oportunitatea de a învăța și de a aplica o varietate de concepte și tehnici importante în dezvoltarea de aplicații Java, cu accent pe gestionarea datelor și interacțiunea cu o bază de date MySQL. Iată câteva aspecte semnificative învățate din această temă:

- **Reflection în Java**: Am învățat să utilizez reflection-ul în Java pentru a inspecta și manipula obiectele la timpul de execuție, ceea ce m-a permis să creez o structură de acces la date flexibilă și scalabilă în aplicație. Folosirea reflection-ului mi-a oferit posibilitatea de a lucra cu obiecte de diferite tipuri, reducând astfel duplicarea codului și oferind o abordare mai generică și mai eficientă în interacțiunea cu baza de date.
- **Operarea cu baza de date MySQL**: Am obținut cunoștințe practice despre cum să interacționez cu o bază de date MySQL în Java, inclusiv crearea, citirea, actualizarea și ștergerea datelor din tabelele bazei de date folosind operațiile CRUD. Am învățat să utilizez PreparedStatement-uri pentru a preveni injecțiile SQL și pentru a asigura securitatea datelor.
- **Structurarea codului**: Am învățat să structurez codul aplicației într-o manieră modulară și bine organizată, folosind clase abstracte, interfețe și clase DAO specifice pentru a separa logica de acces la date de logica de afaceri. Această abordare m-a ajutat să creez un cod mai ușor de întreținut și de extins în viitor.

În continuare, **Dezvoltarea ulterioară** a aplicației consta în implementarea unui sistem de autentificare și autorizare pentru utilizatori, extinderea funcționalității pentru a permite gestionarea comenzilor de produse în stoc și gestionarea stocului în timp real, precum și adăugarea de funcționalități de raportare și analiză pentru a genera rapoarte personalizate despre vânzări, stoc etc..