DOCUMENTATIE

TEMA 3

ORDER MANAGEMENT

Acu Andrei Cosmin

Grupa 30225

CUPRINS

1. Obiectivul temei

-Se va prezenta obiectivul principal al temei printr-o fraza si un tabel sau o lista cu obiectivele secundare. Obiectivele secundare reprezinta pasii care trebuie urmati

pentru indeplinirea obiectivului principal. Fiecare obiectiv secundar va fi descris si

se va indica in care capitol al documentatiei va fi detaliat.

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

-Modelare, scenarii, cazuri de utilizare . Se va prezenta cadrul de cerinte functionale formalizat si cazurile de utilizare ca si diagrame si descrieri de use-case.

Descrierile use-case-urilor se vor face sub forma unui flow-chart ori sub forma unei liste continand pasii executiei fiecarui use-case.

1. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)

-Se va prezenta proiectarea OOP a aplicatiei, diagramele UML de clase si de pachete, structurile de date folosite, interfetele definite si algoritmii folositi (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator).

4. Implementare

-Se va descrie fiecare clasa cu campurile si cu metodele importante. Se va descrie implemantarea interfetei utilizator.

5. Rezultate

-Se vor prezenta scenariile pentru testare cu Junit sau alt framework de testare.

6. Concluzii

-Se vor prezenta concluziile, ce s-a invatat din tema, dezvoltari ulterioare.

7. Bibliografie

-Se vor mentiona resursele bibliografice care au fost folosite pentru dezvoltarea temei.

1. Obiectivul temei

Obiectivul acestei teme este implementarea unei aplicatii OrderManagement care proceseaza comenzile clientilor pentru un depozit. Se folosesc baze de date relationale pentru a stoca produsele, cleintii si comenzile. De asemenea aplicatia trebuie sa fie structurata pe pachete folosind o arhitectura stratificata. Si ar trebui sa urmeze minimal urmatoarele clase:

* Clasele model – modelele de date ale aplicatiei ;
* Clasele pentru logica de afaceri – implementeaza logica aplicatiei ;
* Clasele de prezentare – implementeaza input-ul / output-ul user-ului ;
* Clasele pentru acces de date – implementeaza accesul la baza de date ;

Obiective secundare*:*

* Dezvoltarea de use-case-uri
* Alegerea corecta a structurilor de date
* Impartirea pe clase
* Dezvoltarea algoritmilor
* Implementarea solutiilor
* Testarea programului

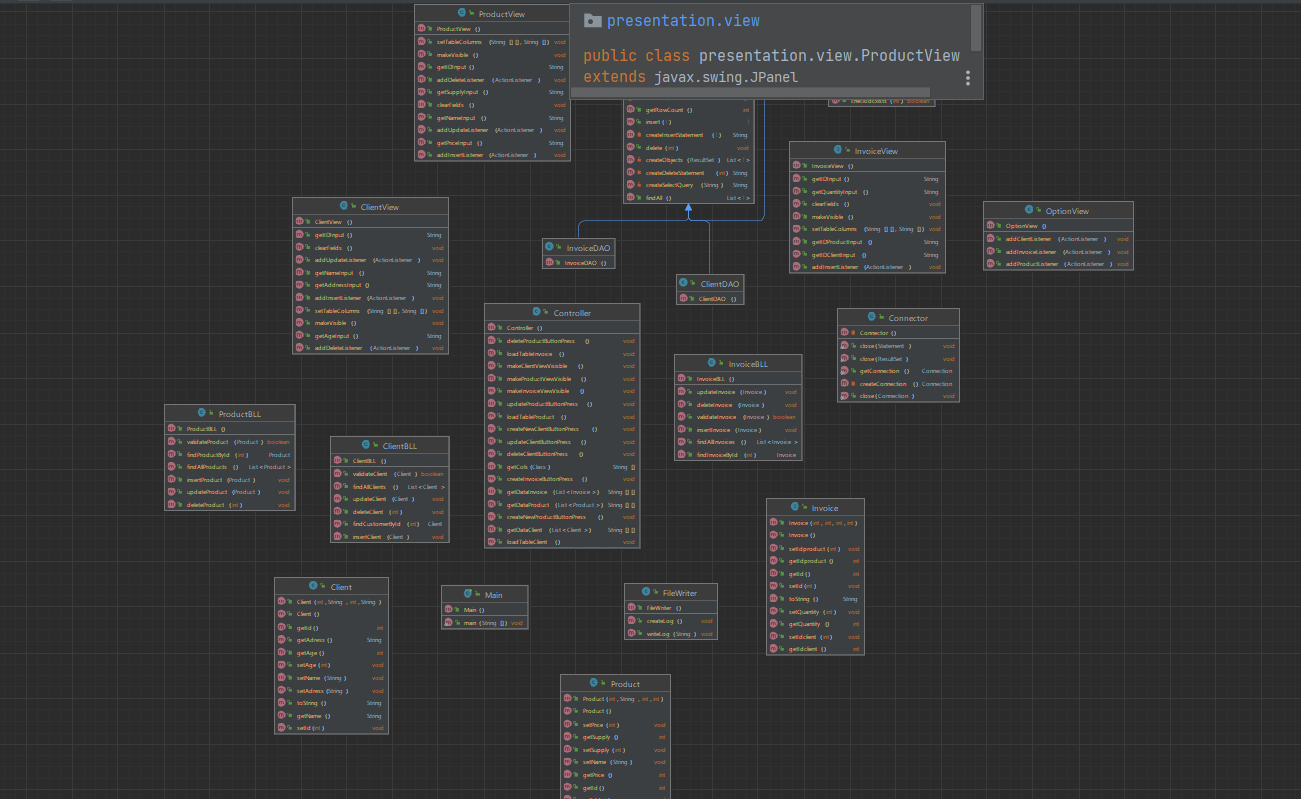
1. Proiectare

* Decizii de proiectare:

Se va folosi tehnica “ reflection “ atat pentru accesul si operatiile cu baza de date cat si pentru scrierea in fisiere. **Java Reflection** este procesul de analizare si modificare a tuturor capabilitatilor unei clase la runtime.

1. Implementare

Diagrama UML:



-Se va descrie fiecare clasa cu campurile si cu metodele importante. Se va descrie implemantarea interfetei utilizator.

Clasa ClientBLL reprezintă logica de business a aplicației pentru gestionarea clienților. Aceasta include metode pentru a găsi clienți (în funcție de ID sau toți clienții), pentru a insera, actualiza sau șterge un client și, de asemenea, o metodă pentru a valida un client înainte de inserare sau actualizare.

Clasa InvoiceBLL gestionează facturile. Are metode similare cu ClientBLL pentru a găsi, insera, actualiza sau șterge o factură și o metodă pentru a valida o factură înainte de inserare sau actualizare. De asemenea, aceasta conține o instanță a FileWriter, ceea ce sugerează că rezultatele operațiilor asupra facturilor sunt înregistrate într-un fișier.

Clasa ProductBLL gestionează produsele. Are metode similare cu ClientBLL și InvoiceBLL pentru a găsi, insera, actualiza sau șterge un produs și o metodă pentru a valida un produs înainte de inserare sau actualizare.

În general, în toate aceste clase, logica de validare asigură că datele care sunt inserate sau actualizate în baza de date respectă anumite reguli de afaceri. De exemplu, în ClientBLL, se verifică dacă clientul nu este nul și dacă acesta are cel puțin 16 ani.

Fiecare din aceste clase BLL utilizează o clasă DAO corespunzătoare (Data Access Object) pentru a executa operațiile efective asupra bazei de date. Aceasta este o bună practică pentru separarea responsabilităților - clasele BLL se ocupă de logica de afaceri, iar clasele DAO se ocupă de logica de acces la date.

DataAccess pachet

Clasa AbstractDAO este o implementare a pattern-ului DAO (Data Access Object). Aceasta abstracție separă codul care accesează baza de date de logica de afaceri, ceea ce face codul mai curat și mai ușor de întreținut.

Aici, AbstractDAO este implementat folosind Generics în Java, ceea ce permite codului să funcționeze cu mai multe tipuri de date. În acest caz, T este un tip de date generic.

**Clasa AbstractDAO** furnizează metode pentru operațiile CRUD (Create, Read, Update, Delete) în baza de date:

findAll(): Returnează toate înregistrările din tabelul asociat cu tipul T.

findById(int id): Returnează o înregistrare specifică din tabelul asociat cu tipul T bazat pe id-ul său.

insert(T t): Introduce o nouă înregistrare în tabelul asociat cu tipul T.

update(T t): Actualizează o înregistrare în tabelul asociat cu tipul T.

delete(int id): Șterge o înregistrare specifică din tabelul asociat cu tipul T bazat pe id-ul său.

getRowCount(): Returnează numărul de înregistrări din tabelul asociat cu tipul T.

Metodele createInsertStatement(T t), createUpdateStatement(T t) și createDeleteStatement(int id) sunt metode private care generează interogările SQL pentru insert, update și delete, respectiv.

În plus, createObjects(ResultSet resultSet) este o metodă care transformă un ResultSet într-o listă de obiecte de tip T. Această metodă folosește tehnici de reflectie pentru a crea și a seta valoarea obiectelor.

Clasele specifice DAO precum ClientDAO, InvoiceDAO și ProductDAO probabil extind această clasă AbstractDAO pentru a avea acces la metodele sale. Aceste clase DAO specifice pot avea, de asemenea, metode specifice pentru acele clase de date.

ClientDAO: Acesta extinde AbstractDAO cu tipul Client. Nu există metode adăugate în această clasă, deci toate operațiile care se pot face pe un client sunt cele moștenite de la AbstractDAO: findAll(), findById(int id), insert(Client t), update(Client t), delete(int id) și getRowCount().

InvoiceDAO: Acesta extinde AbstractDAO cu tipul Invoice. Similar cu ClientDAO, nu există metode adăugate în această clasă. Deci toate operațiile care se pot face pe o factură sunt cele moștenite de la AbstractDAO.

ProductDAO: Acesta extinde AbstractDAO cu tipul Product și include și alte două metode:

checkIdExists(int id): Această metodă verifică dacă există un produs cu un anumit id în baza de date. Dacă există, metoda returnează true, altfel returnează false.

getStockForId(int id): Această metodă returnează cantitatea de produse în stoc pentru un anumit id de produs.

Deci, pentru un Product, pe lângă metodele moștenite de la AbstractDAO, putem verifica dacă un anumit id există în baza de date și putem obține cantitatea de produse în stoc pentru un anumit id.

Această implementare a pattern-ului DAO permite o separare clară a responsabilităților, ceea ce face codul mai ușor de înțeles și de întreținut.

**Clasa FileWriter** este o clasă care îndeplinește funcția de a crea și a scrie într-un fișier text denumit LogOfEvents.txt. Acest fișier va stoca o serie de evenimente sau log-uri.

Metodele din această clasă sunt:

createLog(): Aceasta este o metodă care creează fișierul LogOfEvents.txt în directorul curent al aplicației. Dacă fișierul există deja, va afișa un mesaj care spune "Fișierul deja există". Dacă apare o eroare în timpul creării fișierului, aceasta va fi prinsă și gestionată de blocul catch.

writeLog(String log): Aceasta este o metodă care permite scrierea în fișierul LogOfEvents.txt. Log-ul sau șirul de caractere pe care îl primim ca parametru va fi adăugat la sfârșitul fișierului, iar dacă scrierea este realizată cu succes, se va afișa mesajul "S-a scris cu succes în fișier". Dacă apare o eroare în timpul scrierii în fișier, aceasta va fi prinsă și gestionată de blocul catch.

Un lucru important de menționat este că această clasă FileWriter din pachetul fileWriter este diferită de FileWriter din pachetul java.io. Dacă doriți să evitați confuzia, puteți redenumi această clasă în ceva diferit, cum ar fi LogFileWriter.

**Clasele Client**, Invoice și Product sunt componente de bază ale modelului de date din aplicație. Acestea conțin informațiile specifice despre fiecare entitate pe care o manipulați. Iată o scurtă descriere a fiecăreia:

Client: Această clasă reprezintă o entitate "Client" în aplicația dvs. Fiecare client are un id, un name (nume), age (vârstă) și adress (adresă). Aveți gettere și settere pentru toate aceste câmpuri, ceea ce înseamnă că puteți obține și modifica valorile acestora. De asemenea, aveți o metodă toString() care returnează o reprezentare a obiectului Client ca un șir de caractere.

Invoice: Această clasă reprezintă o entitate "Invoice" (Factură) în aplicația dvs. Fiecare factură are un id, un idclient, un idproduct și o quantity (cantitate). Există, de asemenea, gettere și settere pentru aceste câmpuri, ceea ce înseamnă că puteți obține și modifica valorile acestora. Metoda toString() returnează o reprezentare a obiectului Invoice ca un șir de caractere.

Product: Această clasă reprezintă o entitate "Product" (Produs) în aplicația dvs. Fiecare produs are un id, un name (nume), un price (preț) și o supply (cantitatea disponibilă). Există gettere și settere pentru aceste câmpuri, ceea ce înseamnă că puteți obține și modifica valorile acestora.

Toate aceste clase urmează tiparul Java Beans, ceea ce le face ușor de folosit cu diverse biblioteci Java care lucrează cu acest tip de obiecte. De exemplu, aceste clase ar putea fi folosite cu biblioteci de ORM (Object-Relational Mapping) pentru a stoca și a recupera date dintr-o bază de date

**Clasa Connector** din pachetul dataAccess.connectionFactory este o clasă folosită pentru a stabili o conexiune la baza de date MySQL. Aceasta utilizează JDBC (Java Database Connectivity), o tehnologie care permite aplicațiilor Java să interacționeze cu sisteme de gestiune a bazelor de date.

Conexiunea la baza de date se face folosind URL-ul bazei de date, numele de utilizator și parola. Acestea sunt definite în constantele DRIVER, DBURL, USER și PASS.

Clasa Connector este implementată ca un Singleton, ceea ce înseamnă că există doar o instanță a acestei clase în întreaga aplicație. Aceasta se realizează prin declararea unui membru static singleInstance și prin asigurarea faptului că constructorul este privat, astfel încât alte clase nu pot crea noi instanțe.

Metoda getConnection() este folosită pentru a obține o conexiune la baza de date. Aceasta folosește metoda createConnection() pentru a crea o nouă conexiune.

Clasa oferă, de asemenea, trei metode close(), care sunt folosite pentru a închide conexiunile, instrucțiunile SQL și rezultatele interogărilor (ResultSet). Aceasta este o bună practică, deoarece resursele asociate cu aceste obiecte pot fi costisitoare și este important să fie eliberate atunci când nu mai sunt necesare.

În cazul în care apar excepții, acestea sunt înregistrate într-un obiect Logger și sunt afișate în consolă folosind e.printStackTrace(). Acesta este un mod simplu, dar eficient, de gestionare a erorilor într-o aplicație Java.

Clasa Controller din pachetul presentation.controller pare să fie componenta centrală a modelului Model-View-Controller (MVC) al aplicației tale. Acesta coordonează logica de business și interacțiunea cu utilizatorul.

La crearea instanței de Controller, se inițializează atât View (vizualizări) cât și BLL (Business Logic Layer). Fiecare View este asociată cu o anumită funcționalitate a aplicației: gestionarea clienților, a facturilor și a produselor. BLL conține logica de business pentru fiecare din aceste entități.

Apoi, se setează ascultătorii (listeners) pentru evenimente în fiecare View. Acestea sunt acțiunile care se vor întâmpla când un utilizator face o anumită acțiune, cum ar fi apăsarea unui buton. Aceste ascultători sunt implementați ca clase interne în Controller și fiecare implementează interfața ActionListener.

De exemplu, CreateNewActionClient este un ascultător pentru un eveniment care se produce atunci când utilizatorul dorește să creeze un nou client. Când se întâmplă acest lucru, metoda createNewClientButtonPress() este apelată.

Clasa Controller include, de asemenea, metode pentru încărcarea datelor în tabelele vizualizărilor. Acest lucru se realizează prin extragerea datelor din BLL, apoi setarea acestor date în View. În acest scop, se folosește tehnologia de reflectie Java pentru a obține numele câmpurilor și valorile acestora pentru obiectele Client, Product și Invoice.

În concluzie, clasa Controller gestionează fluxul de date între modelul logic de business și prezentare, procesând toate solicitările de acțiuni ale utilizatorului.

**6.Concluzii**

Aceasta tema este una dintre cele mai mari proiecte pe care le-am avut in ultima perioada, poate cea mai complexa provocare in ceea ce priveste limbajul java cu paradigma oop -ului. Cred ca m-a ajutat sa pot fi capabil sa pot face aplicatii care sunt complexe, pe mai multe planuri, sa impart o problema pe zone si m-a pus in fata unor noi concept de poo cu care nu am putut avea altfel access decat daca faceam un curs online sau daca se putea sa imi vina mie sa fac o aplicatie de genul acesteia. Din punctul meu de vedere este o aplicatie cam prea facuta pe fuga, astfel ca intentionez, la fel pentru fiecare aplicatie pe care am avut de facut, sa o refac mai bine si mai complex, deoarece nu mi-am dat interesul sa fie mai mult frumoasa decat functionala, sa o pot imbunatati prin adaugarea unor componente de grafica mult mai apetisante. Am facut exact ce am considerat pentru a primi un punctaj satisfacator pe ea.

In legatura cu Dezvoltarile ulterioare:

1. Pagină de date personale și mai multe atribute pentru angajat, admin, client. Ca în aplicații care deja există pe piață, ar fi frumos să existe o fereastră pentru vizualizarea acestor date. Ar putea exista chiar și imagini de profil.
2. Operarea pe date personale pentru admin. În general, adminul este cineva care are un control foarte mare în întreținerea sistemului informatic. De asemenea, există și cazuri în care se doresc modificări de date personale. De aceea, ar fi bine ca adminul să aibă și o pagină de operare pe utilizatori și datele lor personale.
3. Dezvoltarea unui sistem mai complex de notificări. Acest lucru implică primirea de notificări pentur mai multe evenimente și implementarea pattern-ului de observer și pentru clienți. Ar fi foarte util ca angajatul să poată schimba statusul unei comenzi, iar clientul să fie notificat de acest lucru.
4. Interfață de notificări mai bună pentru angajat. Din cauza timpului limitat, am implementat interfața pentru admin doar pentru a se observa pattern-ul Observer. În viitor ar fi mai bine ca notificările să fie panouri care durează puțin mai mult timp, nu doar ferestre de dialog care se închid imediat

Dezvoltarea interfeței pentru formular. Formularul pentru creare de client, produs și căutare de produs și chiar logare sunt făcute doar ca să fie funcționale, însă ele ar putea fi și mai bune dacă ar avea o opțiune de resetare a tuturor input-urilor. De asemenea, codul lor ar fi mult mai ușor de scris prin refl

**7.Bibliografie**

HYPERLINK "https://www.geeksforgeeks.org/composite-design-pattern/" https://www.geeksforgeeks.org/composite-design-pattern/

HYPERLINK "https://www.confluent.io/blog/stream-processing-part-1-tutorial-developing-streaming-applications/" https://www.confluent.io/blog/stream-processing-part-1-tutorial-developing-streaming-applications/

HYPERLINK "https://www.w3schools.com/java/java\_lambda.asp" https://www.w3schools.com/java/java\_lambda.asp

HYPERLINK "https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html" https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html