ORDER MANAGEMENT

**Student: Vesa Bianca**

**Grupa : 30227**

1. **Obiectivul temei**

**Tema acestui proiect este implementarea unei aplicatii care lucreaza cu o baza de date MySql , reprezentativa pentru un depozit de produse.** . Pentru indeplinirea acestei cerinte am urmat modelul pus la dispozitie in suportul lucrarii de laborator, si am impartit obiectivul principal in mai multe obiective secundare :

* Realizarea legaturii dintre conceptul de tabel in baza de date si cel de clasa in Java. Astfel am definit clasele : Client, Product, Order, OrderItem, care sa fie actorii scenariului definit in cerinta.
* **Identificarea claselor care fac legatura cu baza de date si care lucreaza in mod direct cu aceasta: DBConnect, AbstractDAO, ClientDAO, ProductDAO, OrderDAO, OrderItemDAO.**
* **Crearea claselor care implementeaza operatiile logice definite in clasele mentionate anterior: DataValidation, ClientBLL, ProductBLL, OrderBLL.**
* **Definirea unor clase care sa asigure conversia din fisier cu comenzi in documente PDF care sa contina rezultatul intregii aplicatii: Parser, PDFTable, Bill.**

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

**Cazul de utilizare a acestei aplicatii este procesarea comenzilor clienților pentru un depozit. Baze de date relaționale sunt utilizate pentru a stoca produsele, clienții și comenzile. Aplicația permite procesarea comenzilor dintr-un fișier text dat ca argument, efectuarea operațiunilor solicitate, salvarea datelor in tabele și generarea de rapoarte în format pdf.**

**Dupa cum am mentionat anterior, pentru utilizarea programului se cere sa se dea ca input, un fisier care sa contina comenzi aplicabile pe tabelele bazei de date : insert , delete , select , update.**

**Programul pune la dispozitie urmatorul scenariu de utilizare :**

* **Utilizatorul da ca argument la executarea fisierului .jar fisierul cu comenzi.**
* **Daca acele comenzi sunt valide, se vor executa.**
* **Programul ia in considerare urmatoarele 6 tipuri de comenzi :**
* **Adauga client in baza de date (ex.** Insert client: Ion Popescu, Bucuresti) - Introduce un client nou in baza de date cu cu numele Ion Popescu și adresa Bucuresti
* Sterge client din baza de date (ex. Delete client: Ion Popescu) – Sterge din baza de date clientul cu numele Ion Popescu
* Adauga produs in baza de date (ex. Insert product: apple, 20, 1) - Introduce un produs nou in baza de date cu numele apple, cantitatea 20 si pretul 1
* Sterge produs din baza de date (ex. Delete product: apple) – Sterge produsul apple din baza de date
* Creeaza o comanda in baza de date (ex. Order: Ion Popescu, apple, 5) – Creeaza o comandă pentru Ion Popescu, cu cantitatea de mere 5. Actualizeaza și stocul de mere la 15. Genereaza o factură în format pdf cu comanda și pret total de 5
* Genereaza rapoarte (ex. Report client, Report product, Report order) - Genereaza rapoarte pdf cu toti clienții / toate comenzile / produsele. Rapoartele vor conține informațiile corespunzătoare entitatilor pentru care sunt solicitate (client, comandă sau produs) , returnate din baza de date, afișate într-un tabel într-un fișier PDF

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

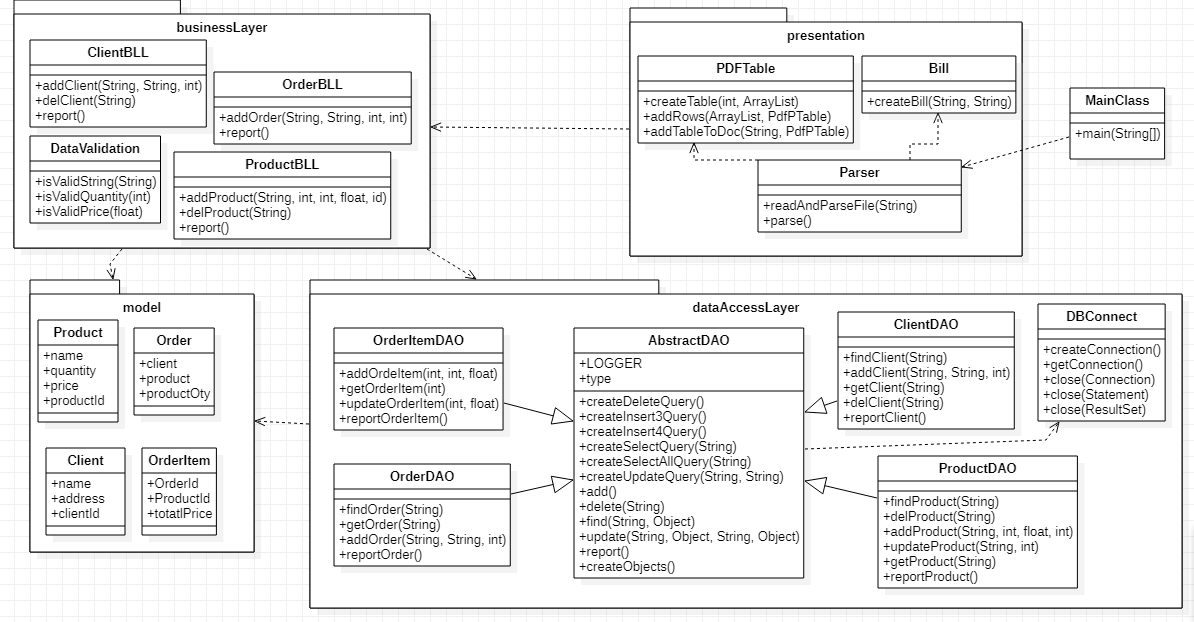
Pentru a realiza legatura cu baza de date este nevoie ca proiectul sa contina clase Java ce au acelasi nume cu tabelele continute. Astfel, o decizie de proiectare a fost sa creez 4 clase in acest scop : Client, Product, Order si OrderItem. Ultima mentionata are rolul de a face legatura dintre celelalte trei, pastrand id-ul clientului, id-ul comenzii si pretul total pe care un client trebuie sa il plateasca pentru toate comenzile plasate.

Aplicatia urmeaza modelul layered architecture, deci este structurata in 4 pachete, fiecare avand un scop diferit:

* model : contine clasele mai sus precizate
* dataAccessLayer : contine clasele care au scopul de a realiza legatura dintre proiectul Java si baza de date relationala. In acest pachet este prezent conceptul de reflection, pe care l-am utilizat la nivelul claselor ClientDAO, ProductDAO, OrderDAO, OrderItemDAO. Aceste patru clase mostenesc clasa AbstractDAO generica, ce ajuta la evitarea codului duplicat, precum si la generalizarea operatiilor pentru o gama mai larga de date de lucru (nu doar cele prezente in aceasta cerinta).
* businessLayer : conține clasele care încapsulează logica aplicației. Acestea utilizeaza pachetul dataAccessLayer, ocupandu-se de manipularea lor in scopul indeplinirii cerintelor impuse
* presentation : contine clasele care realizeaza comunicarea cu utilizatorul prin citirea de fisiere de intrare cu comenzi de aplicat bazei de date si generarea de rapoarte in format PDF

In ceea ce priveste structurile de date, am folosit instante ale colectiei List , pentru a manui cu usurinta un numar mare si variabil de clienti, produse si comenzi. Fiind o structura dinamica, putem sa adaugam, sa eliminam obiecte din lista, precum si sa o parcurgem eficient , cu ajutorul metodelor pe care le pune la dispozitie.

In cele din urma , voi atasa diagrama UML aferenta proiectului :



1. **Implementare**

* **Clasele Client, Product, Order, OrderItem**

Campurile acestora sunt incapsulate si au vedere private , fiind disponibile prin intermediul metodelor getter. Parametri primiti la crearea obiectelor sunt :

Client - clientId (numar unic de identificare) , name (numele clientului) , address (adresa unui client).

Product – name (numele produsului), quantity (cantitatea produsului), price (pretul produsului).

Order – client (numele clientului), product (numele produsului), prodQty (cantitatea produsului), orderId (id-ul comenzii).

OrderItem – orderId (id-ul comenzii), clientId (id-ul clientului), totalPrice (pretul total al comenzilor plasate de un client).

Metodele puse la dispozitie in aceste clasa , exceptand gettere si settere , sunt : clientIdToString(), prodIdToString(), priceToString(), quantityToString(), orderIdToString(), prodQtyToString(), toate avand rolul de a converti variabilele primitive la String pentru a putea fi scrise ca text in documentul pdf.

* Clasa AbstractDAO

Aceasta clasa implementeaza conceptul de reflection. Acest concept este util pentru a comunica cu baza de date astfel incat sa nu se tina cont de natura datelor cu care se lucreaza. AbstractDAO este o clasa cu parametri generici, astfel ca orice instanta a claselor din pachetul model o poate utiliza, ajutandu-ne sa aducem valorile din baza de date si sa le convertim in obiectele corespunzatoare Java.

Metodele createDeleteQuery(), createInsert3Query(), createInsert4Query(), createSelectAllQuery(), createSelectQuery(), createUpdateQuery() construiesc obiecte String care reprezinta statement-uri in limbajul SQL. Statement-urile sub forma de String sunt mai apoi executate cu ajutorul pachetului *java.sql* in interiorul metodelor : *add, delete, update, report* si *find,* care reprezinta operatii de baza CRUD in SQL.

Mai departe, consider ca cea mai importanta metoda prezenta este *createObjects,* care ilustreaza cel mai bine tehnica reflection. In interiorul acestei metode, ResultSet-ul returnat in urma executarii statement-urilor SELECT este convertit in obiecte ale claselor pachetului model. In functie de clasa DAO care mosteneste AbstractDAO, valorile din tabelele bazei de date vor reprezenta variabile instanta ale claselor Client, Product, Order sau OrderItem.

Valoarea returnata de aceasta metoda este un List cu obiecte, instante ale clasei care corespunde numelui tabelului pe care s-a executat query-ul.

* Clasa DBConnect

DBConnect realizeaza conexiunea cu baza de date cu care dorim sa lucram.

Metoda createConnection() incearca sa conecteze programul la baza de date corespunzatoare variabilelor instanta DRIVER, DBURL, USER, PASS. In caz de reusita, aceasta conexiune va fi disponibila ca valoare returnata de metoda getConnection().

Cele trei metode close cu parametri diferiti incearca sa inchida un Connection, un Statement, respectiv un ResultSet.

* **Clasele ClientDAO, ProductDAO, OrderDAO, OrderItemDAO**

**Consider ca aceste patru clase trebuie mentionate impreuna deoarece ele particularizeaza fiecare statement SQL executat in AbstractDAO, in functie de operatiile pe care le executam pe cele patru tabele ale bazei de date.**

**Metodele care vor indica prin mostenire executia unei operatii de tip INSERT sunt addClient(), addProduct(), addOrder() si addOrderItem(). Fiecare dintre acestea primeste ca parametri valorile pe care dorim sa le adaugam in tabelele Client, Product, Order, repectiv OrderItem.**

**Pentru operatiuni de tip DELETE avem metodele delClient(), delProduct(), care sterg din tabele cate un client si un produs, care corespund cu numele pe care le primesc fiecare ca parametru.**

**Operatiunea UPDATE este utila pentru a modifica cantitatea unui produs atunci cand stocul este suplimentat sau cand se accepta o comanda a acestuia, dar si pretul total al unui order item, pentru situatia in care un singur client plaseaza mai multe comenzi, ce vor fi acumulate intr-un singur cost total. Pentru a realiza aceste lucruri utilizam metodele updateProduct() si updateOrderItem().**

**In ceea ce priveste statement-urile de tipul SELECT \* FROM … WHERE …, ele vor fi utile pentru a cauta un client, produs, order sau order item deja existent in baza de date si a lucra mai apoi cu valorile sale. In acest scop, am definit metodele findClient(), getClient(), findOrder(), getOrder(), findProduct(), getProduct() si getOrderItem().**

**In cele din urma, in scopul realizarii unui raport care sa ne dea o imagine a continutului tabelelor bazei de date, vom avea nevoie de un query de forma SELECT \* FROM … . Pentru a face acest lucru, metodele reportClient(), reportProduct si reportOrder(), vor returna liste de obiecte Client, Product si Order, care sunt echivalente valorilor din tabele.**

* **Clasa DataValidation**

**Am creat aceasta clasa pentru a asigura o forma valida a datelor de intrare, care sa corespunda cu realitatea.**

**In metodele isValidString(), isValidQuantity() si isValidPrice() m-am asigurat ca numele produselor, ale clientilor, precum si adresa unui client sa nu contina diferite caractere sau cifre, iar cantitatea produslui si pretul sau sa nu fie numere negative.**

* **Clasele ClientBLL, ProductBLL, OrderBLL**

**Cele trei clase au sarcina de a realiza logica aplicatiei, pe care ulterior clasele DAO o vor pune in practica.**

**Metodele claselor DAO sunt apelate in fiecare metoda din clasele BLL, cu adaugarea diferitor conditii. Aceste conditii asigura o desfasurare corecta a scenariului impus de comenzile din fisierul de intrare.**

**Voi lua treptat fiecare operatiune si o voi patriculariza pentru cele trei clase : Client, Product, Order (mentionez faptul ca nu am creat o clasa BLL si pentru OrderItem, deoarece operatiile pe acest tabel se realizeaza doar la adaugarea de valori in Order):**

* **add :**

**public static void addProduct(String name, int quantity, float price, int id) {**

**if (DataValidation.isValidString(name) && DataValidation.isValidQuantity(quantity) && DataValidation.isValidPrice(price)) { // produsul este adaugat doar daca numele, cantitatea si pretul acestuia sunt valide**

**if (!ProductDAO.findProduct(name)) { // daca produsul nu exista deja in baza de date va fi adaugat**

**ProductDAO.addProduct(name, quantity, price, id);**

**} else { // daca produsul exista deja in baza de date, cantitatea acestuia va fi suplimentata**

**Product product = ProductDAO.getProduct(name);**

**int qty = quantity + product.getQuantity();**

**ProductDAO.updateProduct(name, qty);**

**}**

**}**

**}**

**public static void addOrder(String name, String product, int quantity, int id) {**

**if (DataValidation.isValidString(name) && DataValidation.isValidString(product) && DataValidation.isValidQuantity(quantity)) { // daca numele clientului, cantitatea produsului si pretul acestuia sunt valide**

**Product prod = ProductDAO.getProduct(product); // se cauta produsul in baza de date**

**if (prod.getQuantity() - quantity >= 0) {//daca exista cantitate suficienta din acest produs**

**ProductDAO.updateProduct(product, prod.getQuantity() - quantity); // cantitatea produsului este scazuta**

**if (!OrderDAO.findOrder(name)) { //daca nu exista deja o comanda pe numele dat**

**Client client = ClientDAO.getClient(name);**

**OrderDAO.addOrder(name, product, quantity, id); //clientului cu numele *name* ii este plasata o comanda**

**OrderItemDAO.addOrderItem(id, client.getClientId(), (prod.getPrice() \* quantity));//clientului cu numele *name* ii este plasata o comanda totala**

**} else {/ /daca nu exista deja o comanda pe numele dat**

**int orderId = OrderDAO.getOrder(name).getOrderId(); // se cauta ID-ul clientului deja existent**

**OrderDAO.addOrder(name, product, quantity, orderId); //se plaseaza o comanda noua cu numele sau**

**OrderItem orderItem = OrderItemDAO.getOrderItem(orderId);//se cauta comanda totala a acestuia**

**OrderItemDAO.updateOrderItem(orderId, orderItem.getTotalPrice() + (prod.getPrice() \* quantity));//la care se adauga costul suplimentar corespunzator noii comenzi**

**}**

**} else //daca nu exista cantitate suficienta din produsul comandat se genereaza o chitanta cu textul aferent**

**Bill.CreateBill("order" + id + ".pdf", "We are out of stock for this product!");**

**}**

**}**

**public static void addClient(String name, String address, int id) {**

**if (DataValidation.isValidString(name) && DataValidation.isValidString(address)) {// daca numele si adresa clientului sunt valide**

**if (!ClientDAO.findClient(name)) {// daca clientul nu exista deja in baza de date, este adaugat**

**ClientDAO.addClient(name, address, id);**

**}**

**}**

**}**

* **delete:**

**public static void delProduct(String name) {**

**if (DataValidation.isValidString(name)) { // daca numele produsului este valid, este sters**

**ProductDAO.delProduct(name);**

**}**

**}**

**public static void delClient(String name) {**

**if (DataValidation.isValidString(name)) { // daca numele clientului este valid, va fi sters**

**ClientDAO.delClient(name);**

**}**

**}**

* **report**

**// se apeleaza metoda report din clasa DAO corespunzatoare**

**public static List<Client> report() {**

**return ClientDAO.reportClient();**

**}**

**public static List<Product> report() {**

**return ProductDAO.reportProduct();**

**}**

**public static List<Order> report() {**

**return OrderDAO.reportOrder();**

**}**

* **Clasa Parser**

**Metodele readAndParseFile() si parse() din Parser vor realiza comunicarea cu utilizatorul. Prima citeste un fisier linie cu linie si o apeleaza pe cea de-a doua, care le converteste.**

**Fiecare linie primita ca parametru este parcursa si convertita in operatii pe clasele BLL, care mai apoi se vor transforma in statement-uri SQL.**

**-In cazul in care comanda citita este 'report', creeaza documente PDF cu continuturile fiecarul tabel din baza de date.**

**-In cazul in care comanda citita este 'order', creeaza documente PDF cu cate o chitanta pentru fiecare order.**

* **Clasa PDFTable**

**Am decis sa separ logica lucrului cu tabele pentru documente PDF in aceasta clasa, care contine trei metode:**

**-createTable() : creeaza un tabel de inserat intr-un document PDF, cu parametri dati.**

**-addRows(): adauga randuri noi in tabel**

**-addTableToDoc(): creeaza un document PDF, in care adauga un tabel.**

* **Clasa Bill**

**O ultima decizie de implementare luata a fost sa definesc clasa Bill, care cu ajutorul metodei createBill() creeaza un document PDF tip text, cu numele dat, in care se scrie fiecare order plasat de clientii din baza de date.**

1. **Rezultate**

Dupa cum se poate observa , programul primeste ca argument orice fisier de intrare cu comenzi de aplicat pe baza de date.

Dupa finalizarea aplicatiei , algoritmul implementat a fost testat cu comenzile din fisierul commands.txt pus la dispozitie in indrumatorul de laborator. In urma executarii fisierului .jar s-au creat atat rapoarte pentru fiecare tabel, cat si chitante format pdf pentru fiecare comanda existenta in fisier.

1. **Concluzii**

In urma finisarii acestui proiect pot spune ca am invatat sa lucrez cu baze de date relationale, care se imbina in cadrul acestei aplicatii cu programarea orientata pe obiect. De asemenea implementarea tehnicii reflection m-a ajutat sa dobandesc o gandire mai generica pentru structurarea proiectului si pentru utilizarea ulterioara a acestuia.

In ceea ce priveste dezoltarile posibile, consider ca algoritmul poate fi imbunatatit prin adaugarea de noi clase care sa reflecte o baza de date mai vasta. In acelasi timp, clasa Parser ar putea fi impartita in mai multe task-uri pentru a evita unele bucati de cod duplicat.

1. **Bibliografie**

* **Reflection:**

<https://www.geeksforgeeks.org/reflection-in-java/>

* Java & SQL:

<https://www.javatpoint.com/example-to-connect-to-the-mysql-database>

<https://javaconceptoftheday.com/difference-between-executequery-executeupdate-execute-in-jdbc/>

* Javadoc:

<https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#defaultconstructors>

* **iText:**

<https://www.concretepage.com/itext/create-pdf-with-text-list-table-in-java-using-itext>