DOCUMENTATIE

TEMA 3

NUME STUDENT: ERNST ROBERT

GRUPA: 30226

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc95297886)

[3. Proiectare 3](#_Toc95297887)

[4. Implementare 3](#_Toc95297888)

[5. Rezultate 3](#_Toc95297889)

[6. Concluzii 3](#_Toc95297890)

[7. Bibliografie 3](#_Toc95297891)

# Obiectivul temei

Obiectivul acestei teme creerea unei aplicatii care sa simuleze un depozit de produse la care clientii pot face comenzi. Aplicatia este legata de o PostgreSQL prin JDBC.

Pentru a realiza aceasta tema este necesara conexiunea la baza de date, realizarea operatiilor CRUD pentru fiecare tabel: Product, Client, Order si Bill.

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Pentru a putea adauga/ sterge/ updata sau primi date despre un Client, Produs sau Order, este necesara creerea de tabele pentru fiecare clasa model, si creerea layerelor arhitecturale de Repository->Service->Controller->View*.*

Vom avea nevoie si de multe window-uri pentru frontend, una pentru fiecare operatie, pentru fiecare clasa mutabila. View-urile de GET, vor folosi reflexia intr-o functie statica pentru a creea tabele.

# Proiectare

Avem nevoie obligatoriu de niste clase standard pe care le vom folosi ca model. Client, Order si Product, pe care le vom creea cu structure de date simple, cum ar fi long pentru Id, int pentru quantity, String pentru nume sau email-uri si float pentru tot ce tine de pret.

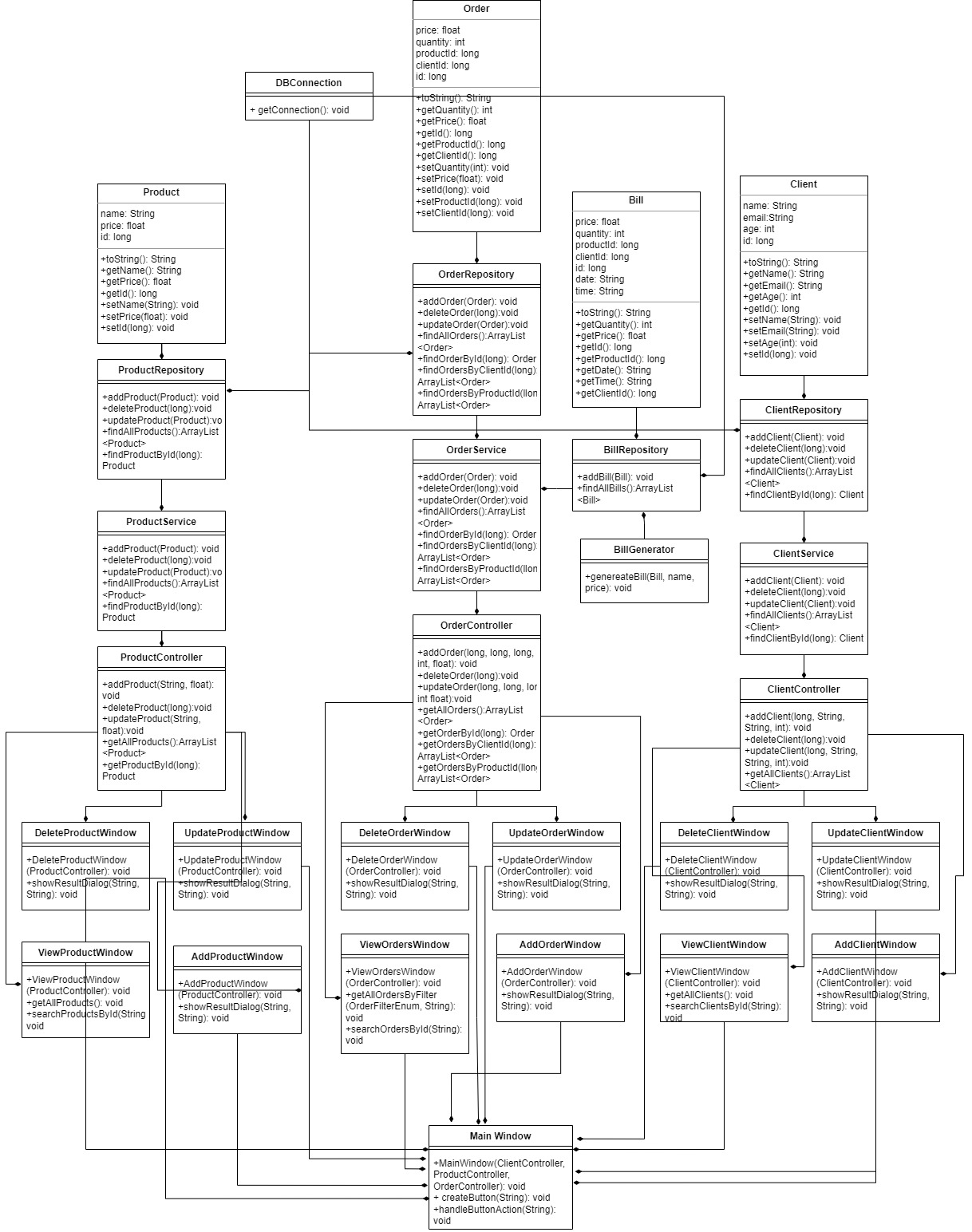
Pentru generatoare ( BillGenereator, TableUtil ) vom folosi o clasa cu o functie statica in ele. Listele le vom creea doar cu ArrayList pentru simplitate si usurinta in munca.

Creem o singura interfata care foloseste reflexia si definim cum ar trebui sa arate un Repository, cu functiile speicifice.

Layerele Arhitecturale le declaram ca si clase, si pasam Layerul de jos ca si parametru final in constructor, fiind o tehnica numita DependecyInjection. Doar ca aici trebuie sa pasam de mana parametrii in main, nu cu

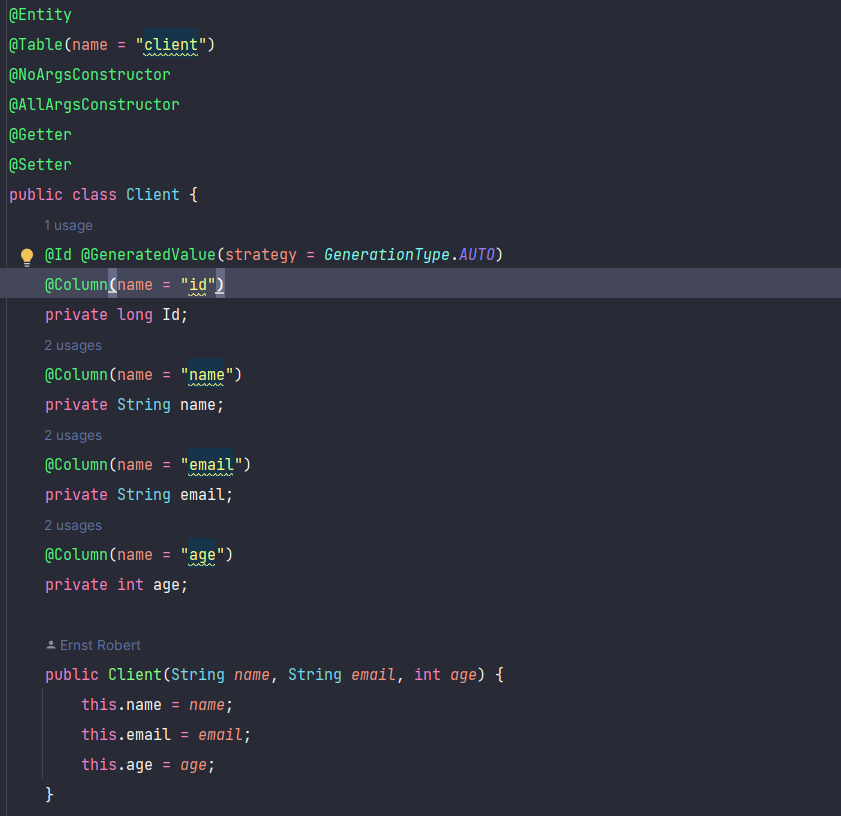
@Autowired cum ar fi fost traditional.

Mai jos prezentam Diagrama UML:



# Implementare

Vom incepe crearea prin crearea bazei de date ORDERS in PostgreSQL, si creerea de tabele, cu statement-urile urmatoare generate de IntelliJ din clasele facute ca model, cu abonatiile specifice:



@Id declara atributul ca primaryKey, iar @GeneratedValue il face sa generez automat id-urile, ca o clasa statica. @Getter si @Setter declara gettere si settere pentru toate atributele, @NoArgsConstructor delcara un constructor care accepta 0 parametrii, si @AllArgsConstructor care accepta toti paramtetrii. @Column semnifica faptul ca acel atribut este o coloana din tabel. Iar IntelliJ ne declara automat statement-urile pentru a delcara tabele, pe care le rulam in baza de date din PostgreSQL.

CREATE DATABASE orders; // pentru creerea bazei de date

CREATE TABLE package

(

Id SERIAL NOT NULL,

clientId BIGINT,

productId BIGINT,

price FLOAT,

quantity INTEGER,

CONSTRAINT pk\_order PRIMARY KEY (Id)

);

//pentru crearea de tabelul de package ( am denumit-o asa deoarece order ii un keyword in Postgre )

CREATE TABLE client

(

id SERIAL NOT NULL,

name VARCHAR(255),

email VARCHAR(255),

age INTEGER,

CONSTRAINT pk\_client PRIMARY KEY (id)

); //pentru crearea de tabelul de client

CREATE TABLE product

(

id SERIAL NOT NULL,

name VARCHAR(255),

price FLOAT,

quantity BIGINT,

CONSTRAINT pk\_product PRIMARY KEY (id)

);

//pentru crearea de tabelul de product

CREATE TABLE bill

(

Id SERIAL NOT NULL,

datee VARCHAR(255),

time VARCHAR(255),

clientId BIGINT,

productId BIGINT,

price FLOAT,

quantity INTEGER,

CONSTRAINT pk\_bill PRIMARY KEY (Id)

);

//pentru crearea tabelului bill

ALTER TABLE package

ADD CONSTRAINT fk\_product

FOREIGN KEY (productid)

REFERENCES product (id);

//cheie straina de la order la product

ALTER TABLE package

ADD CONSTRAINT fk\_client

FOREIGN KEY (clientid)

REFERENCES client (id);

//cheie straina de la order la client

Apoi creem conexiunea JDBC cu baza de date, folosind userul si password-ul bazei de date, si driverul, pe care l am declarat ca dependenta In pom.xml:



Incepem apoi layerele cele mai aproape de bazei de date, unde folosim statement-urile de sql pentru a insera, sterge, updata sau primii datele, fiind cele de Repository:

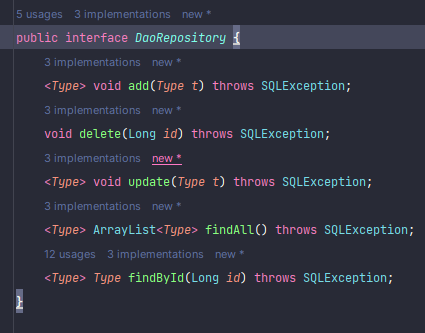
-DaoRepository

-ClientRepository

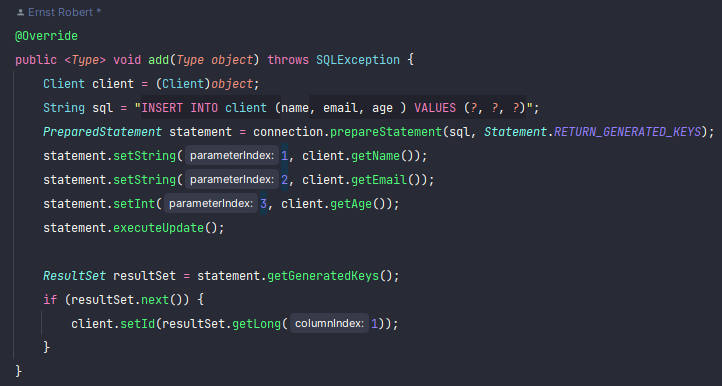
-ProductRepository

-OrderRepository

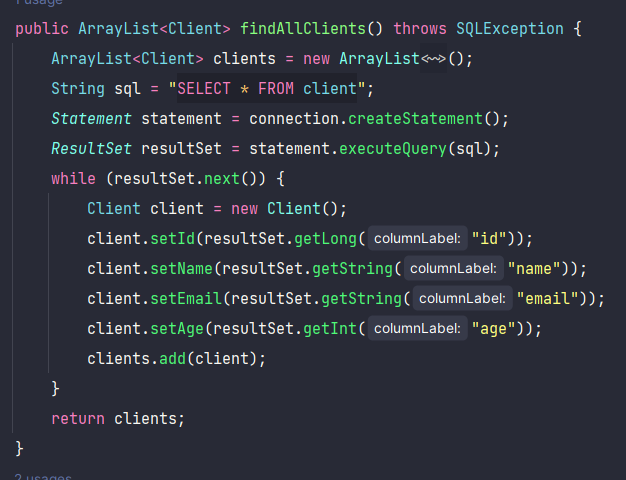
-BillRepository



-folosim reflexia pentru a implementa DaoRepository si a folosi caracteristicile speicifice, toate inafara de BillRepository

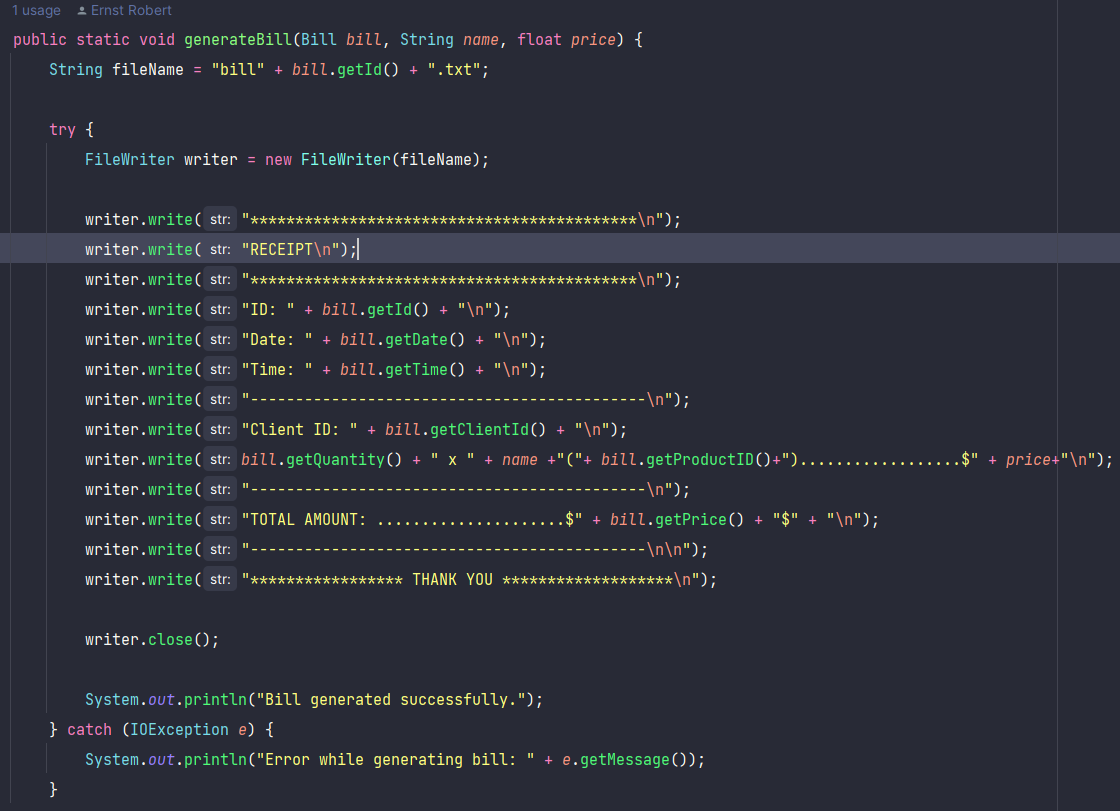


Toate functiile de add/update/delete din Repository-uri, declar instructiunea sql, si adaug parametrii in statement din argumentul de intrare, si daca nu avem erori, setam Id-ul generat. In OrderRepository, dupa ce verificam sa fie totul ok inainte de adaugarea in baza de date, apelam BillGenerator si adaugam in baza de date un Bill nou cu datele comenzii.



Toate functiile de find, care nu au argumente de intrare, declar statementul sql, si il execut, mai apoi parcurgem rezultatul si facem un Obiect din el, pe care il adaugam in lista si returnam;

Bill Generator este o clasa care contine doar o functie statica de a genera un text file dupa Id-ul bill-ului, printr-un fileWriter

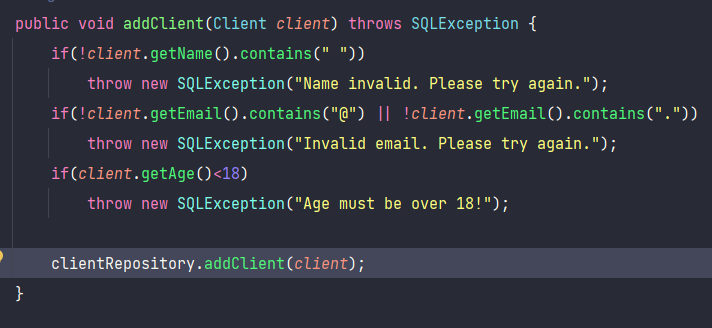


Functiile de Service, sunt layerele de business logic, unde verificam daca conditiile sunt acceptate, daca nu, facem verificari in cazul updatarilor care sunt dependente ( e.g. cand updatam quantity la order, sa updatam si quantity-ul produsului), trimitem o eroare cu mesajul corespunzator si continuam sa trimitem datele catre Repository, prin dependency injection. Mai avem si un getter al repository-ului pentru a ne ajuta la Frontend. Functiile de Service sunt:

-ClientService

-ProductService

-OrderService

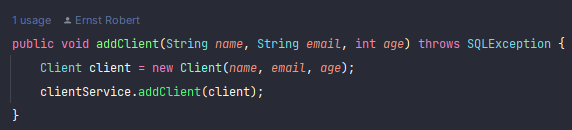


Functiile de Controller, sunt layerele cele mai apropiate de Fronend, unde creem obiecte din datele trimise din View-uri si apelam mai departe Service-urile. Mai avem si un getter al Service-ului aferent care ne ajuta pentru a apela repository-ul in Frontend. Functiile de Controller sunt:

-ClientController

-ProductController

-OrderController



Ajungand la partea de frontend, folosim MainWindow, unde avem niste butoane pentru toate operatiile CRUD pentru toate clasele noastre ( inafara de Bill ). Butoanele conduc fiecare catre o clasa separata ( AddClientWindow, UpdateClientWindow, DeleteClientWindow, ViewClientsWindow, etc. )

Toate clasele de Add{Clasa}Window contin titluri si Field-uri pentru ca utilizatorul sa introduca datele, care sunt extrase si trimise mai departe catre Controller-ul aferent, si un buton pentru a efectua operatia. In cazul in care operatia esueaza, afisam un dialog cu mesajul erorii, iar daca este succssesful, afisam un dialog cu textul “{Clasa} added succssesfully”.



Functiile de AddWindow sunt:

-AddClientWindow

-AddProductWindow

-AddOrderWindow

Clasele de update sunt asemanatoare cu cele de add, un plus au un set de text si Field-uri pentru Id, pentru a sti ce element din baza de date sa updatam. La fel, la apasarea butonului, se apeleaza functia de update din Controller, si dialogurile aferente rezultatului operatiei.



Functiile de UpdateWindow sunt:

-UpdateClientWindow

-UpdateProductWindow

-UpdateOrderWindow

Clasele de Delete{Clasa}Window au doar un titlu si un field pentru Id, pentru a introduce Id-ul aferent celui dorit stergerii. La apasarea butonului, se va apela functia de stergere din Controller-ul aferent si se vor afisa dialog-uri rezultatului operatiei.



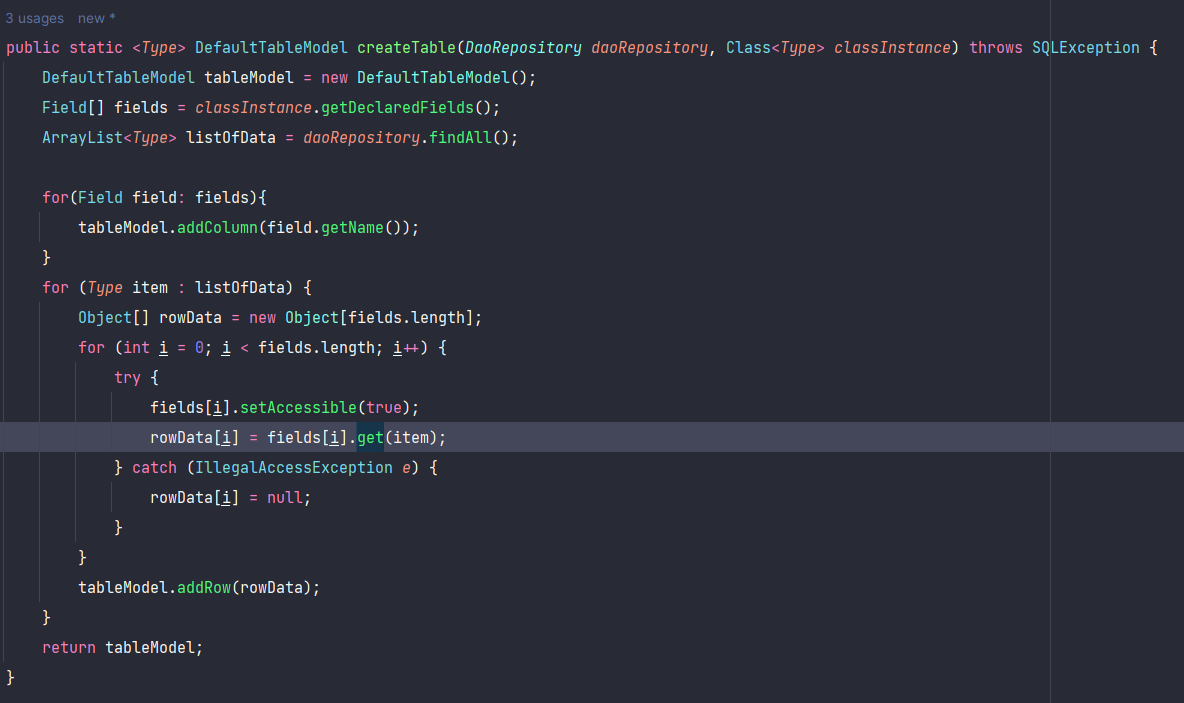
Functiile de DeleteWindow sunt:

-DeleteClientsWindow

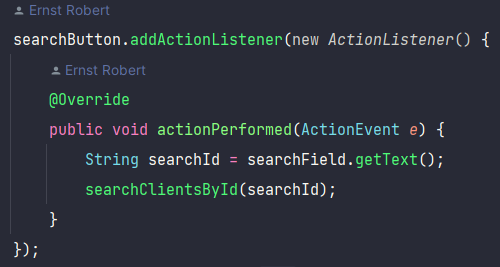
-DeleteProductWindow

-DeleteOrderWindow

Clasele de View{Clasa}Window folosesc o clasa de TableUtil, care primeste ca parametru un DaoRepository si o clasa, si returneaza un DefaultTableModel, cu datele din functia de findAll() a DaoRepository, folosind reflexia.

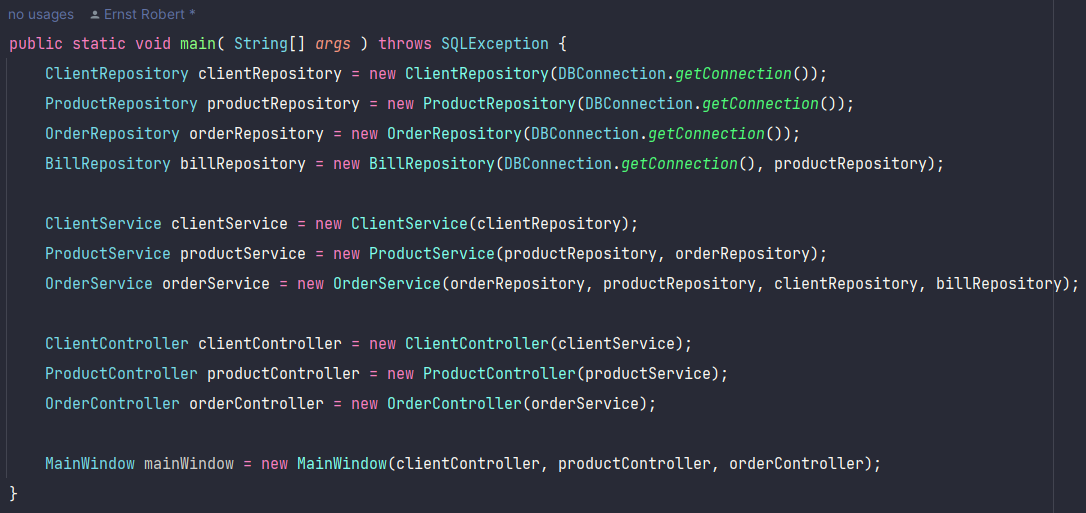


Clasele de genul View contin si un searchbar si un button de search pentru a putea folosia partea de findById() a fiecare clase, avand o functie speciala care se ocupa de asta.



Functia de searchById() apeleaza functia din controller si updateaza row-urile tabelului returnat de TableUtil.

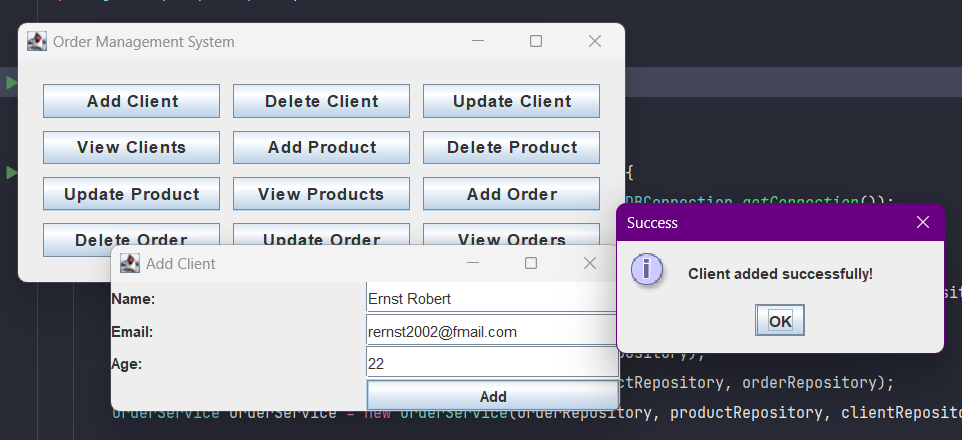
In main instantiem Repository-urile, Service-urile, Controller-ele si un MainWindow, pe care le trimitem cu ajutorul dependency injection.



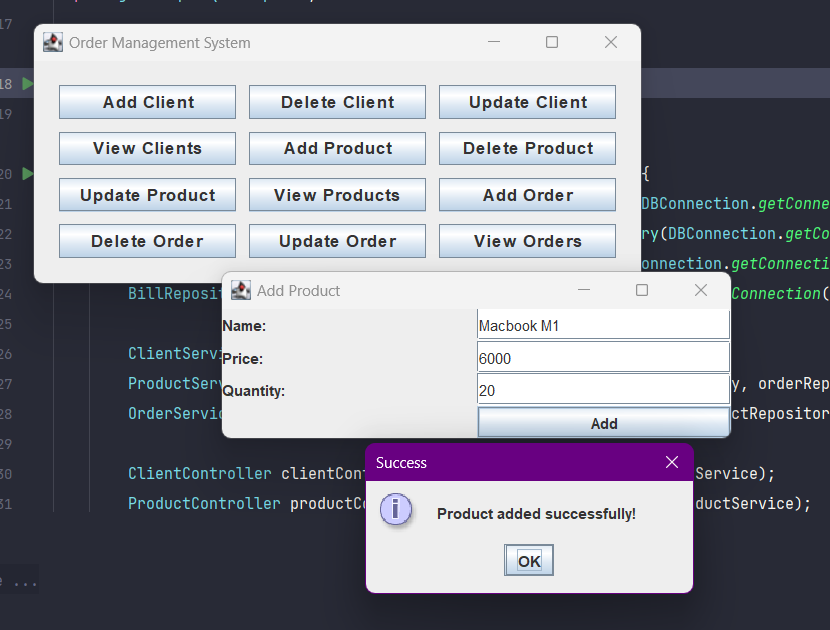
# Rezultate

Toate functiile de add, update, delete si view functioneaza la fel.

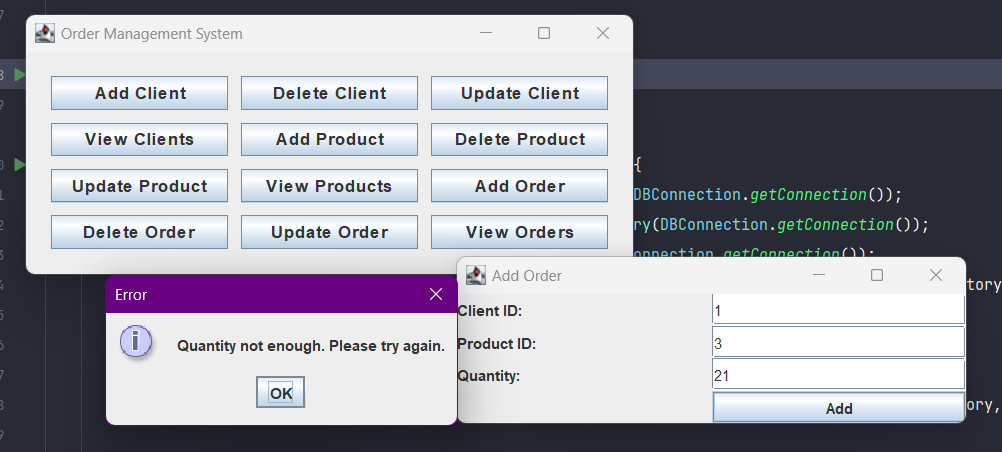
La AddClient nu sunt ce errori sa apara. Adaugam un Client nou.



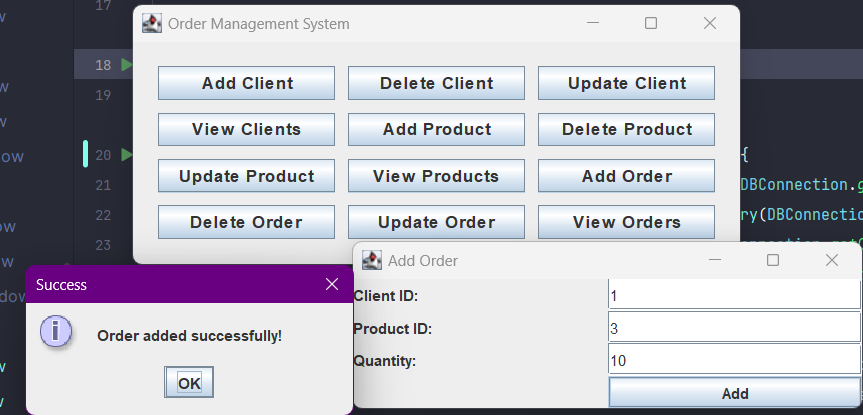
Nici la AddProduct. Adaugam un produs nou.



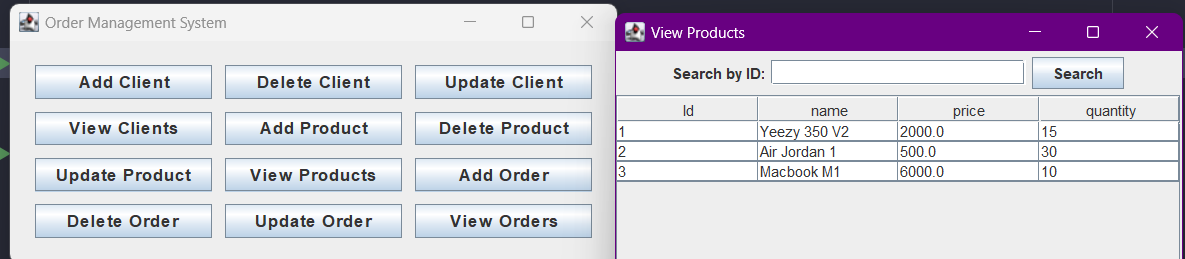
Dar daca incercam sa facem o comanda cu quantity mai mare decat cel al produsului, ne apare eroarea corespunzatoare.



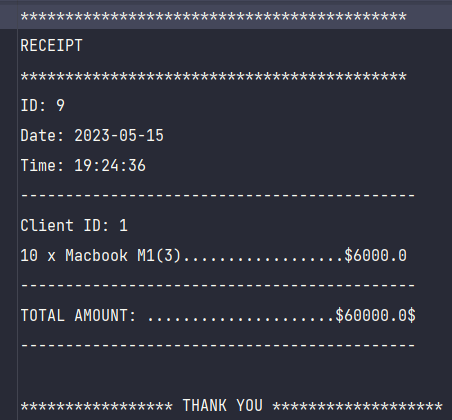
Iar daca incercam un quantity mai mic decat cel al produsului, este ok si se adauga in baza de date.



Si daca dorim sa vedem dupa Produsele, apare quantity-ul de 10 ( celelalte produse sunt facute pentru alte teste). Ca si rezultatul pe care l-am dorit.



Putem observa si bill-ul generat de BiillGenerator cand am plasat order-ul:



# Concluzii

Aceasta a fost una dintre cele mai frumoase teme, deoarece foloseste multe principii care se folosesc in WebDev si in backend, cum ar fi JDBC, LayeredArhitecute, Reflexion, etc. Ca dezvoltari ulterioare, se poate transforma intr-un proiect web, trecand din simplu Maven in Spring, transformand Controllerele in Mapping uri pentru URL si folosind ceva mai friendly pentru baza de date, ca JPA, si un frontend mai usor de customizat, de ex. React/ Angular.

# Bibliografie

<https://www.javatpoint.com/java-swing>

<https://www.oracle.com/technical-resources/articles/java/javareflection.html>

<https://stackoverflow.com/questions/38527998/how-to-connect-postgresql-database-to-a-maven-project>

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