Tehnici de Programare – Tema 3

*Gestionarea Comenzilor*

Popovici Ruxandra

*Grupa 30227*

1. Obiectivul Temei

Obiectivul principal al proiectului de fata a fost proiectarea si implementarea unui simulator de cozi dedicat analizei unui sistem de determinare si minimizare a timpului de asteptare a clientilor. Se foloseste o interfata grafica dedicata prin intermediul careia un utilizator va putea introduce de la tastatura numarul clientilor, numarul de cozi(servere) disponibile, timpul maxim de simulare, intervalul de timp in care pot ajunge clientii si intervaul de timp in care acestia pot fi serviti.

Obiectivul principal al proiectului de fata a fost proiectarea si implementarea unei aplicatii dedicate procesarii clientilor dintr-un depozit. Se foloseste o interfata grafica dedicata prin intermediul careia un utilizator va putea alege mai intai, unul din cele trei tabele pe care va dori sa faca operatii, mai apoi, operatiile respective, impreuna cu eventualele date de intrarecorespunzatoare lor, ce ii vor permite sa interactioneze cu baza de date in mod direct.

Obiectivele secundare ale proiectului:

* Analizarea obiectivelor proiectului si a problemelor legate de implementare prin modelarea de scenarii corespunzatoare cazurilor de utilizare, astfel incat sa identificam corect cerintele;
* Proiectarea aplicatiei astfel incat toate necesitatile utilizatorului sa fie indeplinte;
* Implementarea solutiei optime descoperite prin scrierea de cod;
* Testarea aplicatiei, observandu-i si analizandu-i concret comportamentul pentru mai multe cazuri, luand in considerare scenarii diferite de folosire si diferite date de intrare.

2. Analiza problemei (*modelare, scenarii,cazuri de utilizare)*

Prin analiza problemei intelegem examinarea cerintelor, potentialelor probleme si a diferitelor modalitati in care aplicatia noastra ar putea fi folosita, pusa la incercare: introducerea de date (timpii de simulare, numarul de clienti si cozi) care va duce la generarea unui raspuns corespunzator din partea simulatorului in functie de corectitudinea lor, in functie de intervalele de timp si incadrarea caracteristicilor fiecarui client in acestea.

Dorim ca utilizatorul sa fie atentionat cu privire la corectitudinea datelor, astfel incat va primi mesaj de eroare daca acestea nu respecta cerintele(sa fie numere intregi)

De asemenea, programul trebuie sa functioneze in mod adecvat si sa genereze raspunsul dorit, urmand principiul diagramei urmatoare:

Operatie (instructiunea SQL) pe care vrem sa o efectuam

Tabela pe care vrem sa facem operatii

*Order Management*

*~Schema generala a proiectului~*

Rezultat = Factura \*.txt

+ interactiuni valide cu BD

Date de intrare (un tuple ce descrie una din cele 3 tabele)

Diagrama Use Case demostreaza mai bine functionalitatea si ideea intregului proiect:

**User**

Mov – specific tabelelor Client si Product

Verde – specific tuturor tabelelor

Albastru – specific tabelei Order

Dupa cum putem observa si mai sus, intregul sistem reprezinta aplicatia creata, in care User-ul reprezinta, de fapt, actorul / utilizatorul, care se asteapta sa primeasca un raspuns / rezultat dupa ce actioneaza asupra lui.

* Cerinte functionale ale aplicatiei:
* Trebuie sa permita utilizatorului sa aleaga tabela pe care doreste sa o gestioneze;
* Trebuie sa permita utilizatorului sa introduca datele necesare fuctionarii corecet a intstructiunii SQL pe care doreste sa o efectueze asupra tabelelor;
* Trebuie sa permita utilizatorului sa aleaga operatia corespunzatoare instructiunii SQL pe care doreste a o indeplini (add / INSERT, edit / UPDATE, delete / DELETE, view / SELECT \*);
* Trebuie sa permita utilizatorului vizionarea rezultatului cerut la efectuarea unei comenzi – factura, care va contine toate datele tranzactiei efectuate.
* Trebuie sa permita utilizatorului sa vizioneze intr-o noua fereastra tabela pe care acesta a cerut-o pentru afisare.
* Altele:
* Aplicatia trebuie sa fie intuitiva si usor de folosit de catre utilizator.
* Sa efectueze instructiunile SQL cerute in caz de success sau sa afiseze un mesaj de eroare in caz ca datele primite nu au respectat cerintele sau formatul cautat, ori daca ceea ce s-a cautat nu a fost gasit.
* Scenariu de utilizare:

1. Utilizatorul alege tabela cu care vrea sa lucreze: ex. “Order”;
2. Utilizatorul introduce ID-ul unei noi comenzi de la tastatura: ex. 2;
3. Utilizatorul introduce cantitatea de la tastatura: ex: 60;
4. Utilizatorul introduce atat numele clientului sub care se va face tranzactia, cat si numele produsului pe care doreste sa-l obtina: ex. “Daniel” si “Cactus”;
5. Utilizatorul apasa butonul de “ADD ORDER”;
6. Aplicatia va porni conexiunea cu baza de date iar, in caz ca se respecta si va adauga in tabela Order datele introduse de utilizator;
7. Se va genera un fisier \*.txt, reprezentand factura comenzii.
8. Utilizatorul iese din aplicatie.

Un scenariu alternativ de utilizare:

1. Utilizatorul introduce numele unui client inexistent, ori numele unui produs inexistent, ori o cantitate mai mare decat cea disponibila in depozitul virtual;
2. Aplicatia afiseaza un mesaj de eroare si sterge datele introduse;
3. Se revine la primii pasi din scenariul anterior.
4. Proiectare *(decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)*

Proiectarea in limbaj OOP a presupus mai multi pasi, in functie de deciziile luate: am urmarit implementarea folosind modelul MCV (Model-View-Controller), fiind un model tip sistem interactiv si repartizarea lor in pachete, asa cum se poate observa si in diagrama de mai jos:

**presentation**

**start**

BillGenerator

Controller

View

Main

Validator

EmailValidator

ClientBLL

ProductBLL

OrderBLL

**bll.validators**

**model**

**connection**

**dao**

Model

Client

Product

WarehouseOrder

AbstractDAO

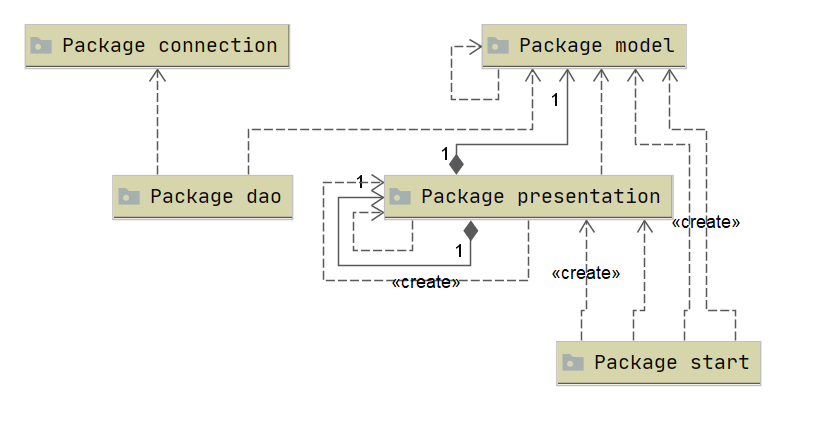
ClientDAO

OrderDAO

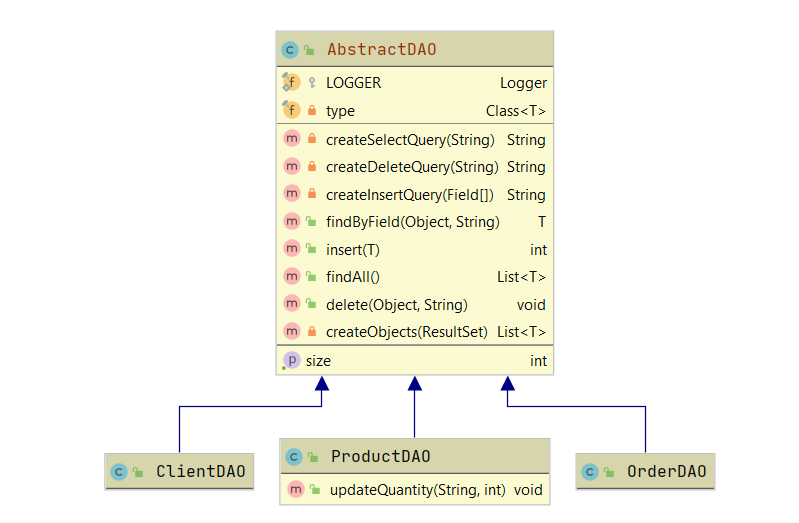
ProductDAO

ConnectionFactory

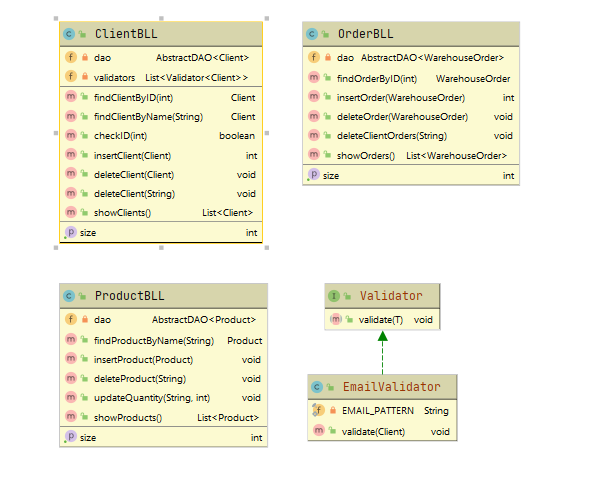
Diagrama UML a proiectului ajuta la o mai buna si amanuntita intelegere ai analiza a claselor, pachetelor, metodelor implementate in fiecare din acestea, cum se “leaga” si “comunica” intre ele:



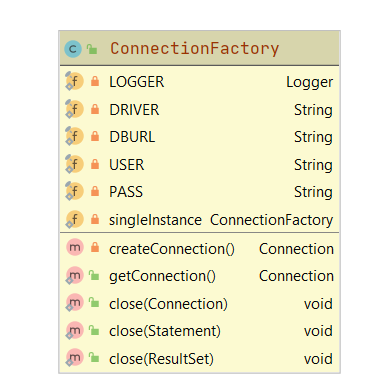
***Pachetul dao:***



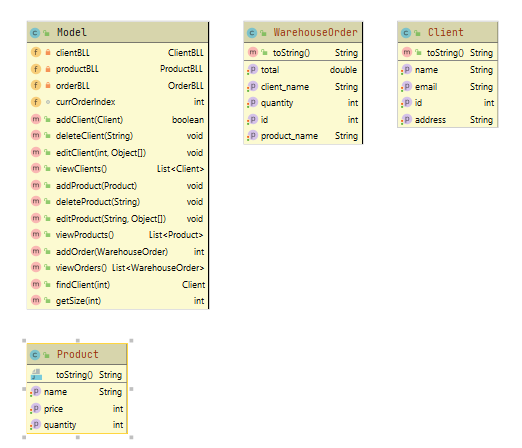
***Pachetul bll.validators:***

******

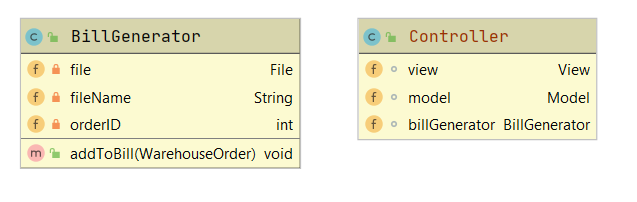
***Pachetul connection:***

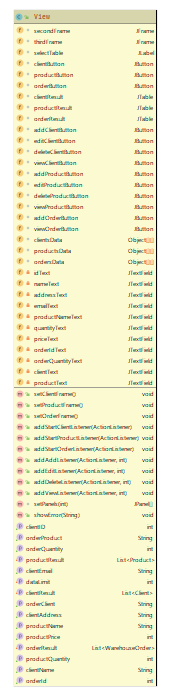
******

***Pachetul model:***

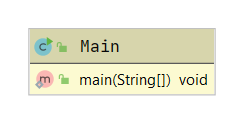
******

***Pachetul presentation:***

******

******

***Pachetul start:***

******

Structura de tip MCV separa fiecare clasa in functie de functionalitatile metodelor din ele.

“model” contine clase ce reprezinta “structurile” datelor la care se asteapta aplicatia: Client, Product, WarehouseOrder, reprezentand clientii, produsele si comenzile corespunzatoare tabelelor din baza de date;

“presentation” le contine pe cele care au de a face cu interactiunea dintre utilizator si aplicatia propriu-zisa: GUI, JFrame, butoanele cu ActionListenere-le respective, etc..

“dao”, “bll.validators”, “connection” are ca si clase chiar logica din spatele aplicatiei ce se ocupa cu “comunicarea” cu baza de date si gestionarea tabelelor in functie de necesitatile utilizatorului.

“main” reprezinta clasa care porneste aplicatia si care leaga toate celelalte pachete si clase impreuna.

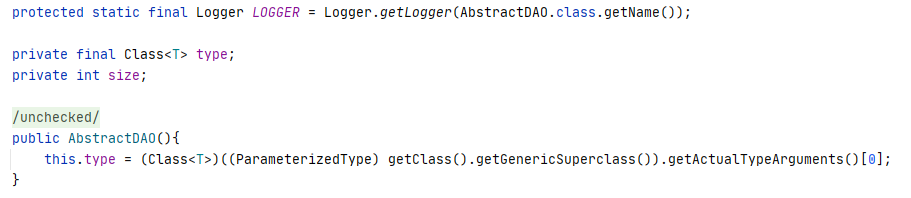
1. Implementare

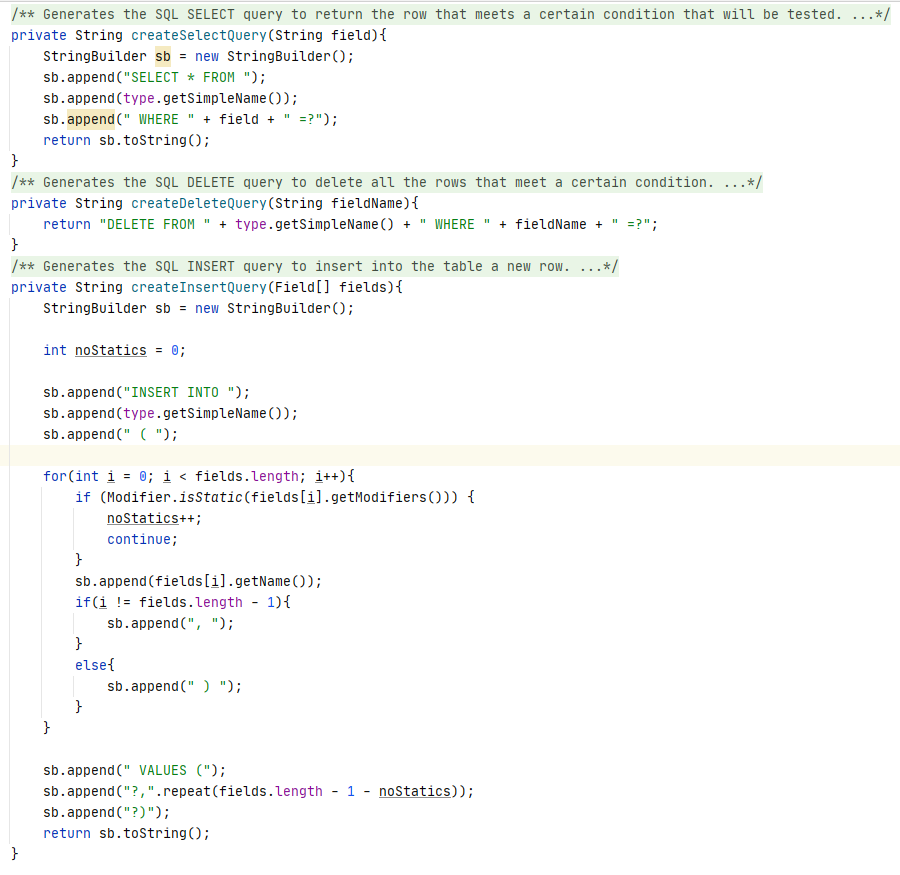
* *Pachetul “dao”*

***Clasa AbstractDAO***

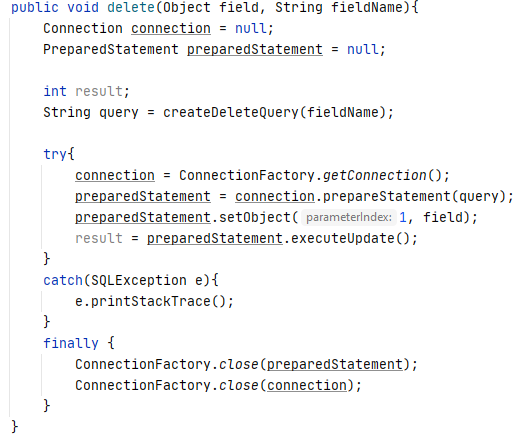
Pentru o mai usoara intelegere si implementare a codului, aceasta clasa abstracta se va ocupa cu “comunicarea” propriu-zisa cu baza de date “ordermanagementdb” – obiectul de tip “Class<T>” va reprezenta tipul clasei din pachetul “model”.

In acest sens, aici vom crea interogarile de tip SQL prin metode private ce vor fi apelate de catre cele publice, folosite inafara clasei de catre instantele \*BLL. Acestea vor crea in mod dinamic atat conexiunea cu baza de date, cat si interogarile, le vor apela pe tabela corespunzatoare si vor genera / returna un ResultSet in cazul in care totul a decurs conform asteptarilor.



**

**

** **

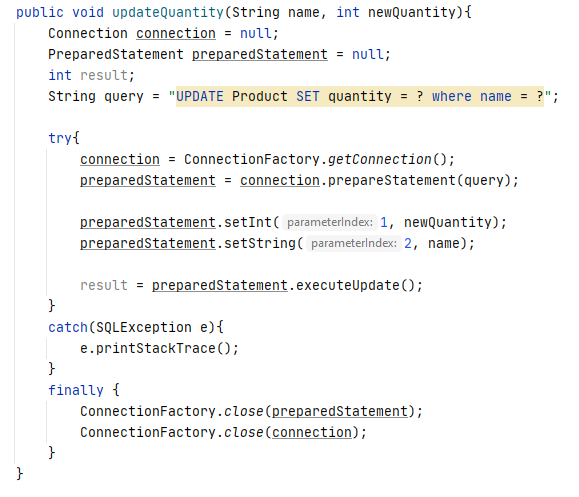
Pentru o folosire mai usoara a rezultatului reutnat de interogari, functia createObjects va returna o lista de obiecte de tip T corespunzatoare uneia din cele 3 clase, primind ca si parametru un obiect de tip ResultSet generat de catre functiile cu care am interactionat cu baza noastra de date.

**

***Clasele Client\*, Order\*, ProductDAO***

Primele doua doar vor mosteni toate metodele clasei absracte, inlocuindu-se, astfel, obiectul de tip clasa “T” cu clasa cu care se vrea a se lucra: “Client”, respectiv “WarehouseOrder”.

Cea din urma, in schimb, are o metoda specifica prin care se va putea actualiza cantitatea unui produs in baza de date: atunci cand ori se insereaza unul care deja exista, crescand cantitatea, ori cand se efectueaza o comanda valida, iar stocul trebuie decrementat.

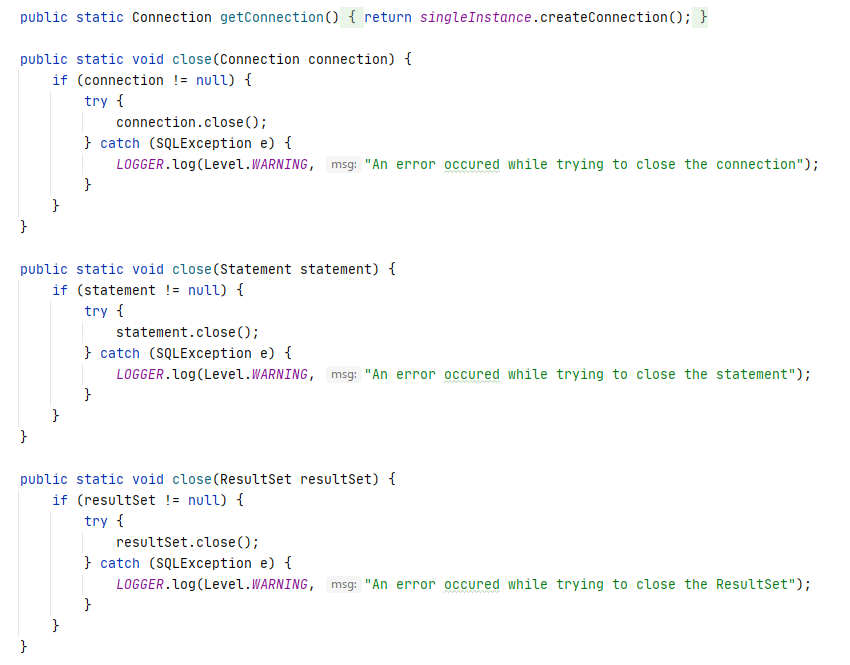


* *Pachetul “connection”*

***Clasa “ConnectionFactory”***

Va gestiona conexiunea aplicatiei cu baza de date: va realiza si initia conexiunea cu aceasta pentru a putea lucra cu tabelele puse la dispozitie, se va ocupa cu “curatenia”, prin metodele de inchidere atat a instructiunilor create in DAO, cat si a obiectului ResultSet, astfel incat vom continua lucrul cu o lista de obiecte de tip T si a conexiunii propriu-zise.

**

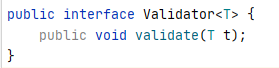
**

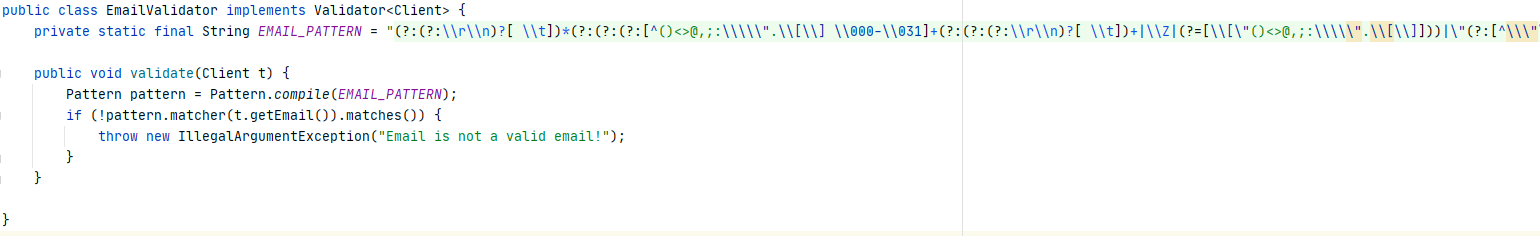
* *Pachetul “bll.validators”*

***Interfata “Validator” si clasa “Email Validator” care o implementeaza***

Ne foloseste in a valida datele de intrare pe care utilizatorul le furnizeaza de la tastatura.

Interfata contine o metoda “add” care va fi implementata in EmailValidator cu un regex prin care vom vedea daca adresa de email a clientului va corespunde cu formatul cerut.



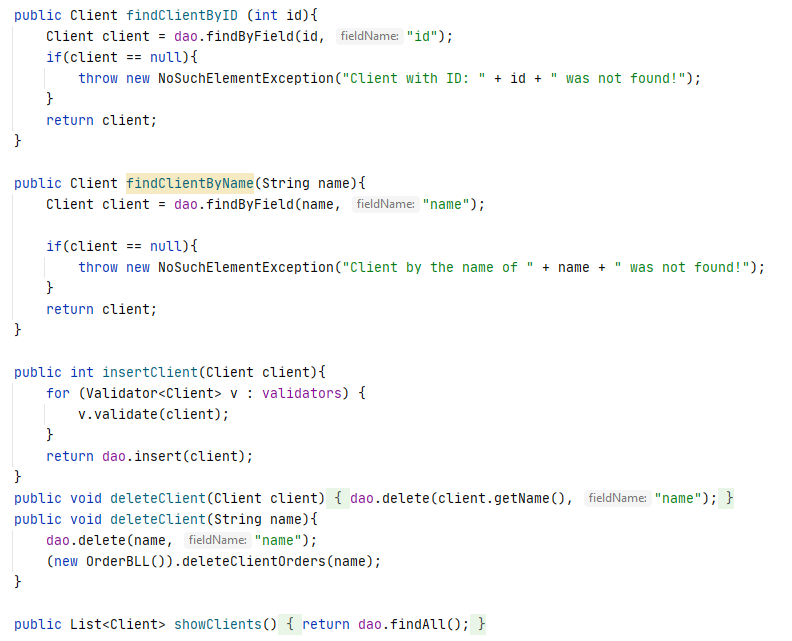


***Clasa “ClientBLL”***

Implementeaza logica operatiilor de insert, delete, update, view din DAO pentru tipul de obiect / tabela Client, prin intermediul unui obiect de tip ClientDAO.

Pentru fiecare operatie se vor face verificariel necesare (ori date returnate din functiile apelate inauntru, ori verificari suplimentare cu exceptiile aruncate).

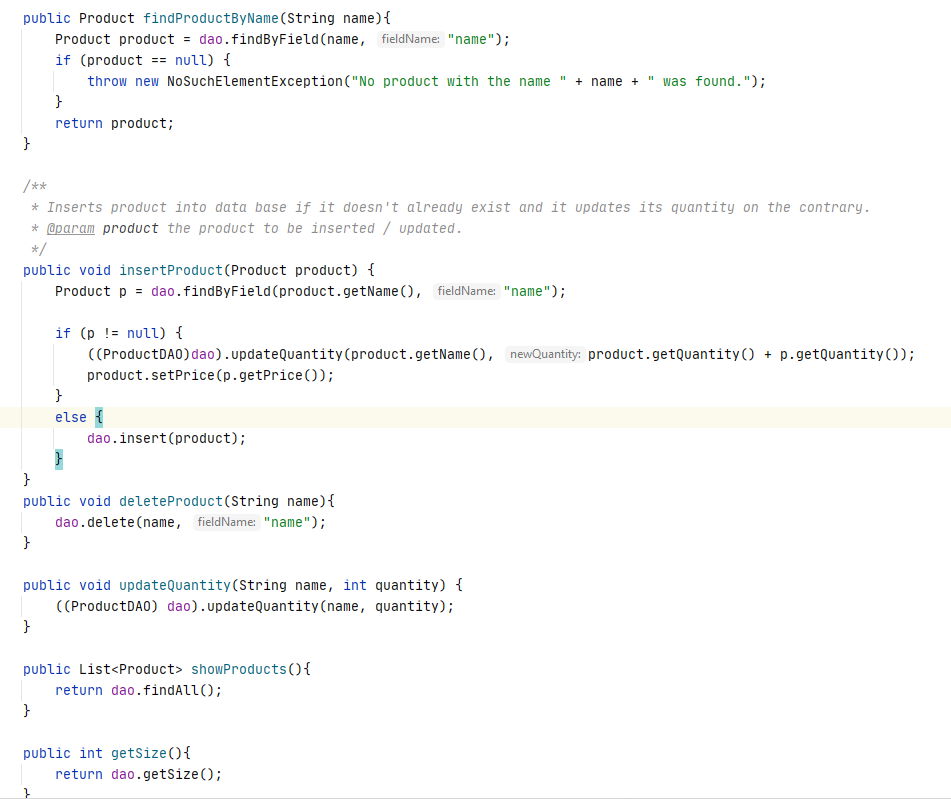
Pe langa acestea, metoda “insertClient” are, in plus, optiunea prin care se valideaza campurile de inserat in tabela (respectiv numai validatorul pentru email).



***Clasa “ProductBLL”***

Asemanatoare clasei “ClientBLL”, cu diferentele ca:

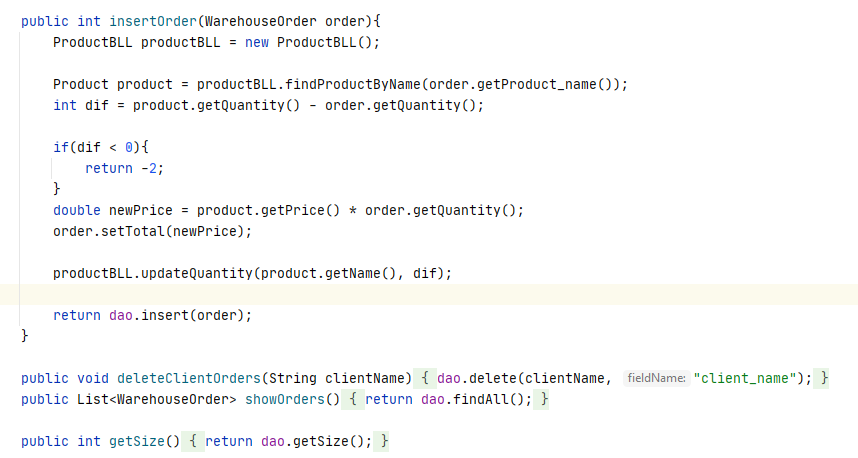
* insertProduct va adauga produsul in baza de date in cazul in care acesta nu exista si va creste cantitaatea in caz contrar;
* avem functia updateQuantity care va actualiza cantitatea unui produs dat.



***Clasa “OrderBLL”***

La fel ca si cele doua clase de mai sus, cu alte doua mari diferente:

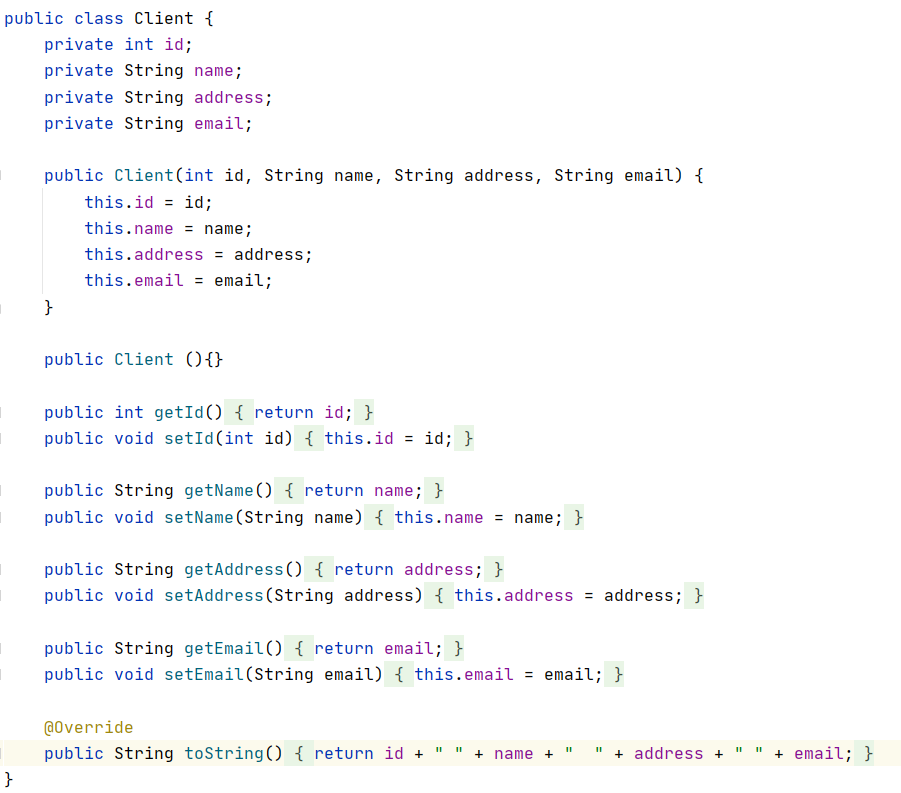
* metoda “insertOrder” va verifica, mai intai, daca exista suficiente resurse / produse pentru a putea fi achizitiozate (verifica cantitatea din tabela “Product” si verifica daca cantitatea ceruta este mai mica), iar mai apoi va calcula pretul (care nu este dat de la tastatura de catre utilizator, ci va fi initializat la valoarea produsului dintre “quantity” si “price”);
* nu vom sterge comenzile in functie de un camp al unui rand din tabela, ci o data cu eliminarea unui client din baza de date se vor sterge si facturile / comenzile acestuia din tabela “order”.



* *Pachetul “model”*

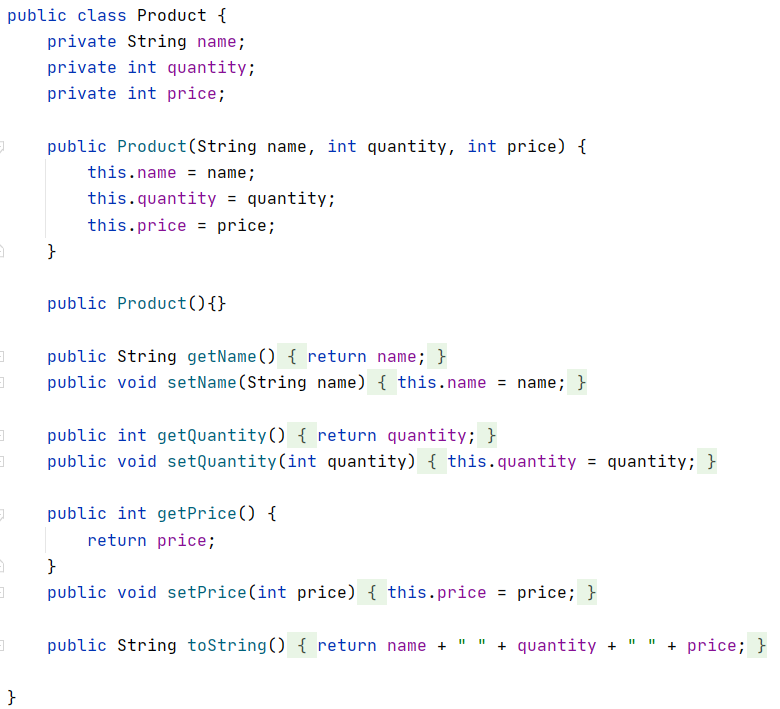
***Clasa “Client”***

Este reprezentarea sub forma de clasa Java a tabelei corespunzatoare din baza noastra de date “Client”, caracterizat prin urmatoarele campuri / coloane: *id(“ID-ul clientului”), name (“numele clientului”), address(“adresa clientului”), email(“email-ul clientului”)*.



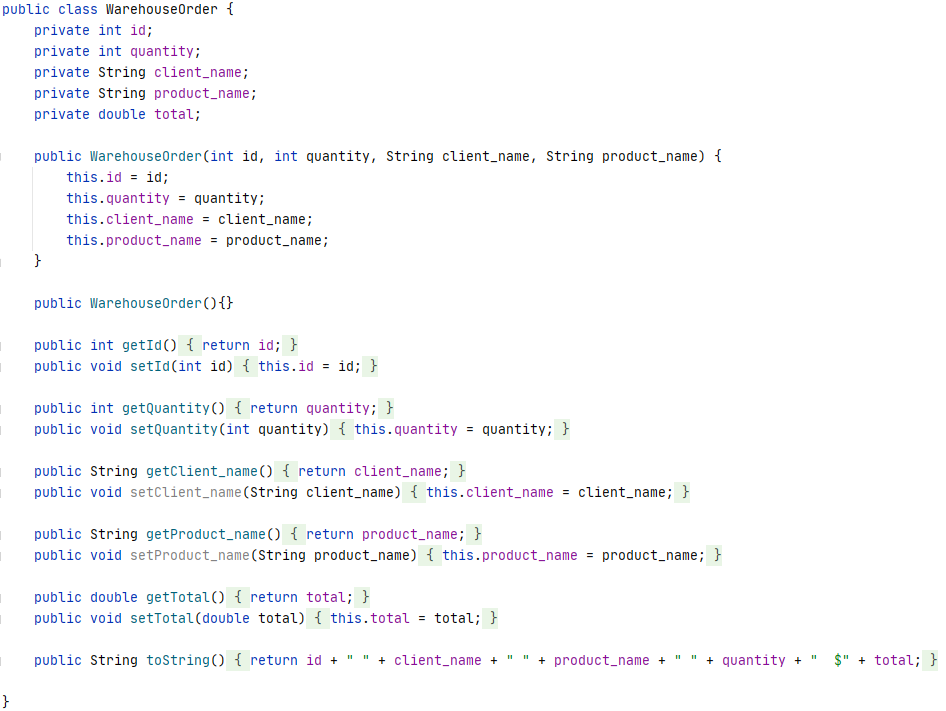
***Clasa “Product”***

Este reprezentarea sub forma de clasa Java a tabelei corespunzatoare din baza noastra de date “Product”, reprezentand produsul care va putea fi achizitionat, caracterizat prin urmatoarele campuri / coloane: *name(“numele produsului”), quantity (“cantitatea din BD”), price(“pretul corespunzator”).*



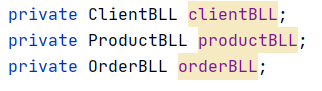
***Clasa “WarehouseOrder”***

Este reprezentarea sub forma de clasa Java a tabelei corespunzatoare din baza noastra de date “Order”, reprezentand comanda care va fi efectuata in depozitul nostru virtual, caracterizata prin urmatoarele campuri / coloane: *id (“ID-ul comenzii”), quantity (“cantitatea produsului comandat”), client\_name (“numele clientului”), product\_name (“numele clientului”), total (“pretul total al comenzii”) = cantitatea ceruta \* pretul produsului .*

**

***Clasa “Model”***

Contine 3 instante diferite pentru cele 3 bll-uri care se vor ocupa de logica din spatele operatiilor pe fiecare tip de date in parte (Client, Product, WarehouseOrder).

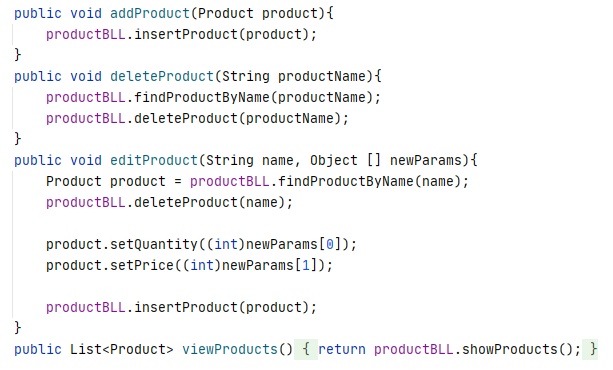


Cu ajutorul instantelor \*BLL, vor face operatiile “cerute” pe baza de date: in functie de tipul obiectului (ceea ce este notat cu \*), va adauga, edita, sterge si returna o lista de obiecte, facand si verificarile necesare si transmitand, prin ***return***, informatia mai departe pentru a fi interpretata (afisare de erori sau validare de date).

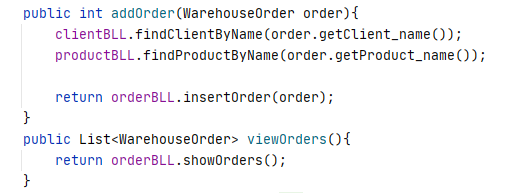
\*= client

**

*\*=* product



\*= order

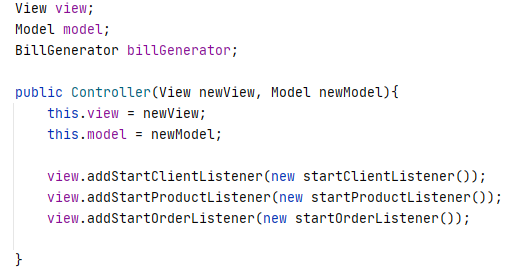


* *Pachetul “presentation”*

***Clasa “Controller”***

Contine un obiect de tip “Model”, unul de tip “View” si unul de tip “BillGenerator”.

Aici se face, de fapt, legatura dintre view (interfata grafica) si model (simularea propriu-zisa).

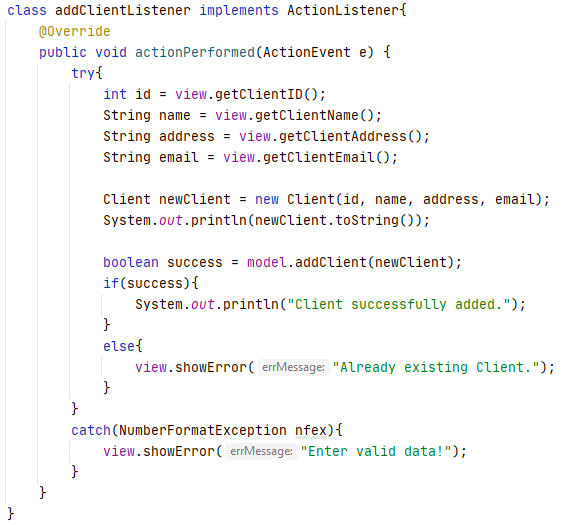


Constructorul acestei clase va adauga Listener-e la actionarea carora se va genera una din cele 3 ferestre corespunzatoare tabelelor puse la dispozitie de baza de date:



Tot in aceasta clasa se gasest ActionListener-e pentru fiecare operatie pe care dorim sa o efectuam pe tabele; prin intermediul acestora se vor putea, de asemenea, genera si eventualele mesaje de eroare.

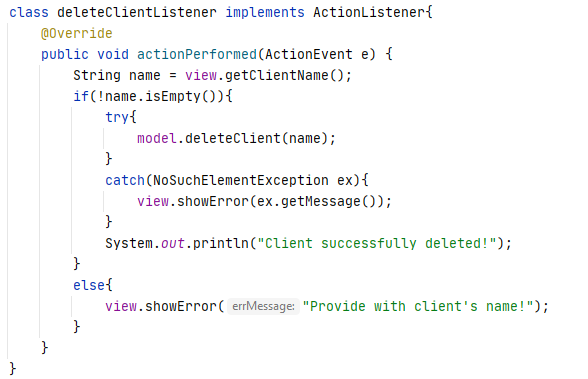
Ex. pentru ***add***:



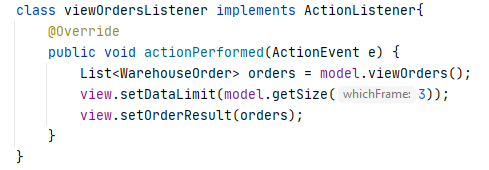
Ex. pentru ***edit***:



Ex. pentru ***delete***:



Ex. pentru ***view***:

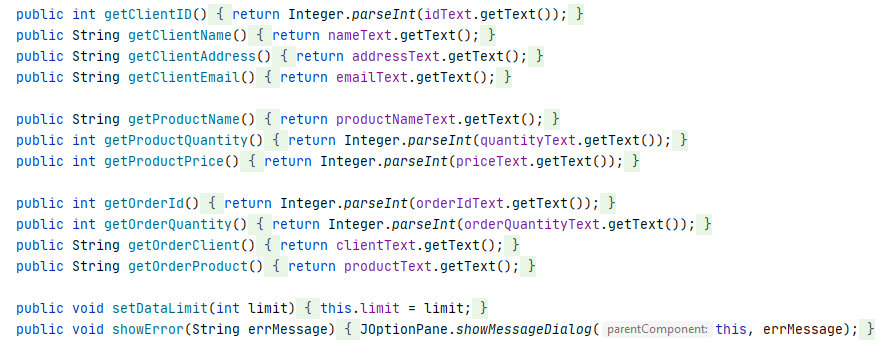


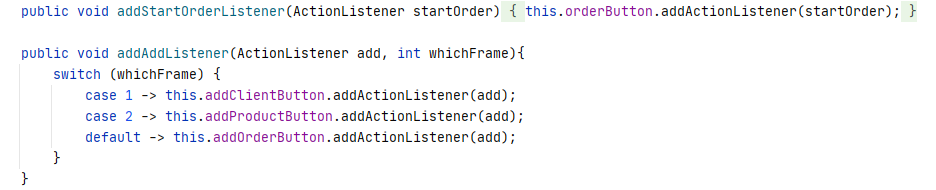
***Clasa “View”***

Aici se va crea interfata grafica cu care va interactiona utilizatorul.

Prin intermediul ei, utilizatorul nu trebuie sa se gandeasca cum se va ajunge / cum trebuie sa se ajunga la un rezultat, el doar va introduce date (prin TextField-uri), va trimite comenzi (prin butoane) si va primi rezultate, adica un raspuns de la aplicatie.

Diferite metode, mai ales cele de la sfarsit, ajuta programatorul in a face legatura cu celelalte clase si componente ale MCV-ului (transmiterea datelor, adica aparametrilor corespunzatori simularii - ex. “*getClientID*”, “*getProductPrice*”, “*getOrderClient*”, adaugarea de actionListener-e corespunzatoare – *ex. “add\*\*\*Listener”*-, afisarea de erori in functie de raspunsul primit de la Controller – *ex. showError* -, stabilirea numarului de coloane pentru afisarea tabelelor – *ex. setDataLimit* -, etc.):

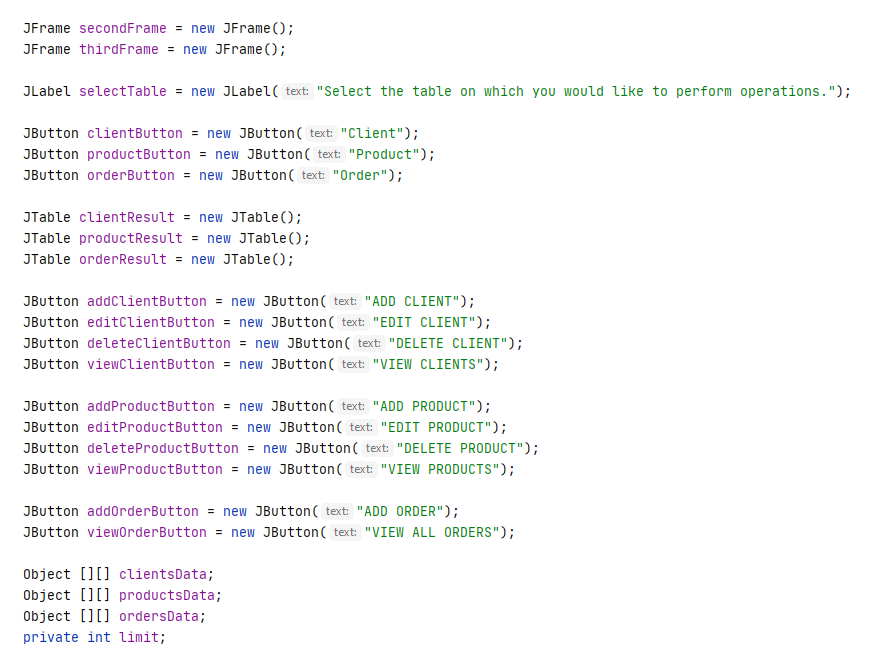




Toate acestea sunt posibile datorita faptului ca “View” mosteneste clasa JFrame:

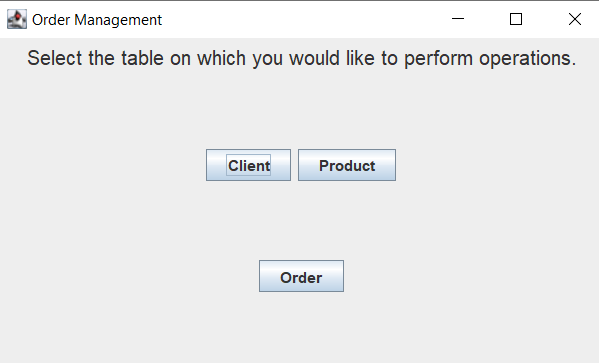


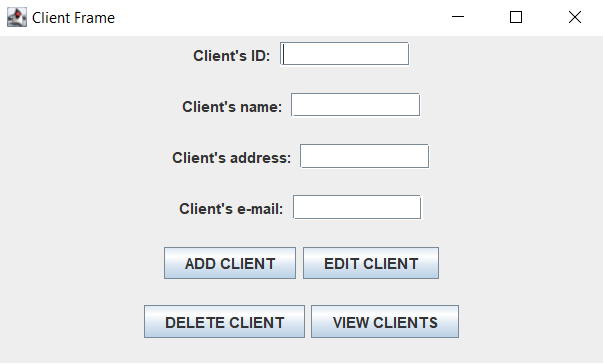
astfel incat constructorul clasei noastre creeaza insasi fereastra cu care va interactiona utilizatorul (interfata). Vom folosi tot felul de obiecte din pachetele “swing” si “awt” de tip JLabel, JTextField, JButton, si JPanel de care un user se va folosi pentru transmiterea de date si informatii, iar un programator pentru prelucrarea si modelarea lor.

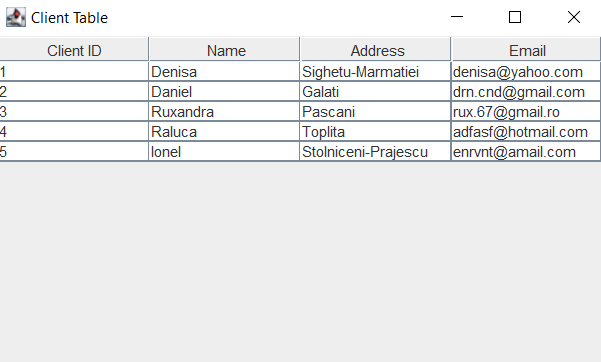


Am impartit atat Frame-ul principal, cat si pe cele secundare, in mai multe panouri (panel-uri) prin functia interna “setPanels”, pentru o mai buna organizare a componentelor din GUI, folosind metode de tip setLayout, add, pack, setAlignment.

De la fereastra principala se vor putea porni alte inca 2, in fucntie de cerintele utilizatorului: pe a doua vom gasi structurile necesare transmiterii de informatii bazei de date, iar pe a treia, rezultatul afisarii tabelei; exemplu pentru tabela “Client”:







De la o fereastra la cealalta se trece apeland metodele:

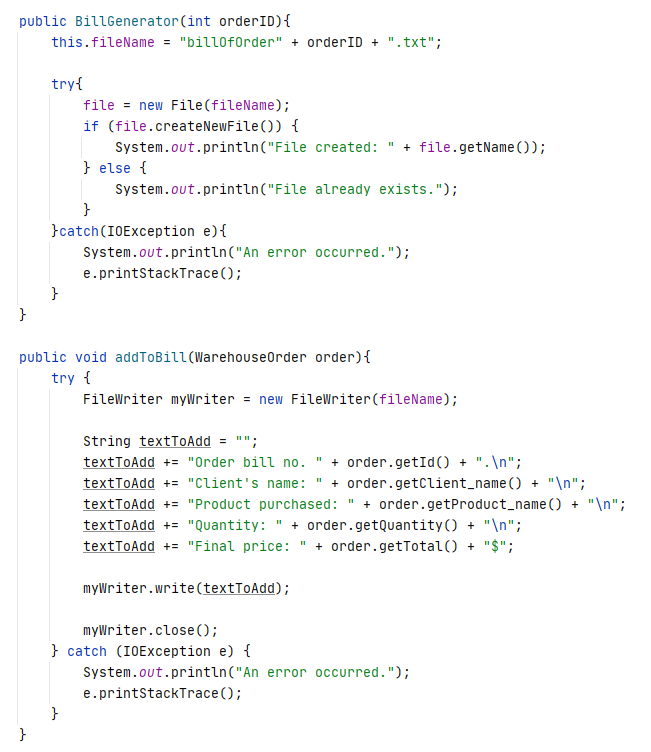




***Clasa “BillGenerator”***

Va genera factura corespunzatoare comenzii plasate, cu conditia ca aceasta s-a efectuat cu succes.

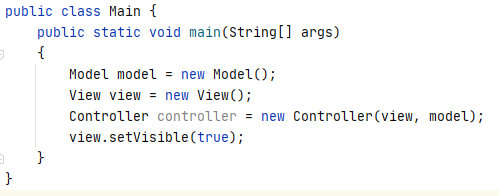
Costructorul va initializa un fisier \*.txt in care vom scrie cu ajutorul metodei “addToBill” ce va primi ca parametru un obiect de tip “WarehouseOrder” de a carui date se va folosi:



* *Pachetul “start”*

***Clasa “Main”***

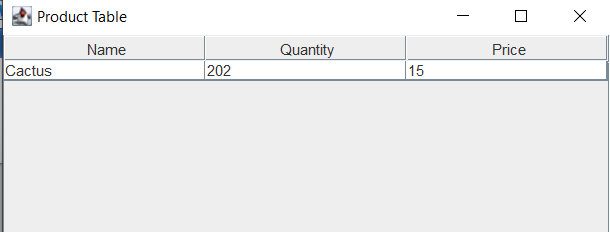
Aceasta clasa porneste aplicatia si permite lucrul cu GUI, facand legatura intre toate celelalte clase si stabilind relatiile care vor duce la buna functionare a interfetei grafice si a programului din spatele acesteia.



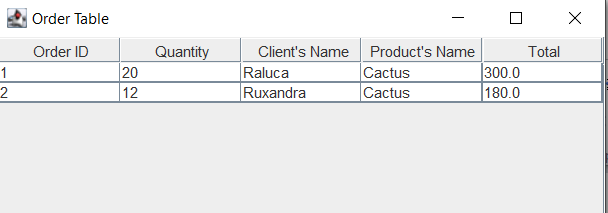
1. Rezultate

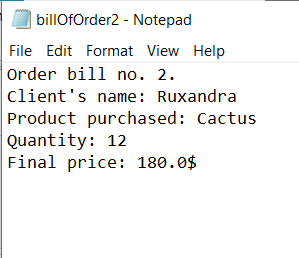
Am testat aplicatia adaugand, mai intai, atat clienti, cat si produse in baza de date, pentru a ma putea folosi de aceste date in plasarea unei comenzi si generarea unei noi facturi.

Cu diferite date de intrare, am ajuns atat la afisarea de erori (introducand numere in loc de String-uri, adrese de email gresite, clienti care nu respectau principiul cheii primare, alegand sa sterg date fara sa introduc campul necesar operatiei “delete”, etc.), cat si la rezultate corecte si succese:



Mai apoi, am ajuns sa generez o nou factura, dupa ce am facut testele necesare pentru ca aplicatia sa nu “crape”, in cazul in care se va ajunge la eventuale exceptii.



Factura: 

1. Concluzii

Prin implementarea acestui proiect s-a urmarit, in principiu, lucrul cu o baza de date, crearea unei aplicatii care se poate folosi de o asemenea componenta externa in proiectul sau, prin stabilirea unei conexiuni si apelarea de metode ce sa extraga sau sa modifice informatii din aceasta.

Am reusit sa creez modele arhitecturale de Layers, sa folosesc tehnica reflectiei in Java pentru a interactiona cu o baza de date externa printr-o interfata grafica GUI si sa generez javadoc-uri pentru clasele proiectului si metodele existente in acestea.

As fi putut imbunatati modul in care se face editarea / update-ul datelor din baza de date astfel incat aceasteia sa-i fie necesare doar datele prin care sa gaseasca obiectul respectiv si doar cele care vor trebui actualizate (ex: doar numele produsului si una, ori ambele campuri care trebuie schimbate, in functie de ce doreste utilizatorul sa modifice).

1. Bibliografie

[*https://www.youtube.com/watch?v=m6rFp-Yxgsk*](https://www.youtube.com/watch?v=m6rFp-Yxgsk)

[*https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/ParameterizedType.html*](https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/ParameterizedType.html)

[*https://www.youtube.com/watch?v=FIACGmzibAM*](https://www.youtube.com/watch?v=FIACGmzibAM)

[*https://www.baeldung.com/java-reflection*](https://www.baeldung.com/java-reflection)

[*https://www.infoworld.com/article/3388036/what-is-jdbc-introduction-to-java-database-connectivity.html*](https://www.infoworld.com/article/3388036/what-is-jdbc-introduction-to-java-database-connectivity.html)

[*https://www.javatpoint.com/steps-to-connect-to-the-database-in-java*](https://www.javatpoint.com/steps-to-connect-to-the-database-in-java)

[*https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html*](https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html)

[*https://www.oracle.com/ro/technical-resources/articles/java/javadoc-tool.html*](https://www.oracle.com/ro/technical-resources/articles/java/javadoc-tool.html)

[*https://gitlab.com/utcn\_dsrl/pt-reflection-example/-/tree/master/src/main/java*](https://gitlab.com/utcn_dsrl/pt-reflection-example/-/tree/master/src/main/java)

*FUNDAMENTAL PROGRAMMING TECHNIQUES*

*ASSIGNMENT 3 – SUPPORT PRESENTATION -Teams-*