

# Documentație warehouse

Fariseu Adriana-Teodora

Facultatea de Automatică și Calculatoare, Grupa 30225

Anul 2020-2021

# **Obiectivul temei**

Se cere realizarea unei aplicații pentru gestionarea comenzilor unui depozit de produse, având în spate o bază de date pentru reținerea informațiilor. De asemenea trebuie să existe opțiuni pentru operații pe clienți, produse și comenzi precum inserări, ștergeri, editări și inserări, toate acestea având loc dintr-o interfață grafică.

Pentru rezolvarea acestei probleme, va trebui să se parcurgă următoarele obiective secundare:

|  |  |  |
| --- | --- | --- |
| Obiectiv | Descriere | Capitol în care este detaliat |
| Analiza problemei | Pentru a transpune problema în cod, trebuie mai întâi să se extragă din enunț informații relevante în legătură cu funcționalitățile și cazurile de utilizare. | [2](#_Analiza_problemei) |
| Proiectarea | Odată extrase informațiile necesare, se poate începe maparea acestora în paradigma de programare pusă la dispoziție (Programare Orientată pe Obiect în Java în cazul acesta) | [3](#_Proiectare) |
| Implementarea | În acest pas se realizează scrierea codului pentru rezolvarea problemei cerute, pe baza mapării realizate anterior. | [4](#_Implementare) |
| Testarea | Pentru a ne asigura că produsul final sau chiar rezultate parțiale necesare pentru o etapă următoare funcționează corect, codul implementat trebuie la rândul lor supus unor teste ce simulează scenarii de utilizare identificate în etapa de analiză. | [5](#_Testare_și_rezultate) |

# **Analiza problemei**

***Cerințe funcționale***

**Persistența informațiilor**. Informațiile legate de clienți, produse și comenzi trebuie să persiste după închiderea aplicației, lucru care face apel la utilizarea unei baze de date relaționale

**Gestionarea operațiilor dintr-o interfață grafică**. Este mai natural ca într-un astfel de scenariu în care utilizatorii nu sunt programatori să se facă o interfață grafică pentru gestionarea operațiilor. Dintr-un meniu, se poate alege daca se doresc operații pe clienți produse sau comenzi.

**Crearea automată a unui tabel**. Se impune afișarea în interfața grafică a informațiilor din baza de date într-un JTable. Mai mult, header-ul acestuia care conține numele atributelor afișate trebuie să fie generate automat prin Reflection

**Validarea datelor**. Acest aspect se referă la verificarea corectitudinii datelor înaintea introducerii lor în baza de date. Atunci când se proiectează o aplicație destinată unui utilizator, șansele sunt destul de mari ca o persoană să greșească atunci când dorește să atribuie valori unor câmpuri, fie din neatenție, fie din pure intenții sociopate. Deoarece câmpurile tabelelor din baza de date depind de implementare, cazurile de date eronate și tratarea concretă acestora vor fi discutate ulterior

***Cerințe non-funcționale***

**Emiterea de chitanță pentru o comandă**. O chitanță ce conține informațiile legate de o comandă oarecare, la fel ca la comenzile din viața reală, ar fi un lucru bun de avut pentru ca un angajat să nu fie nevoit să o facă manual de fiecare dată .

**Automatizarea interogărilor**. Sintaxa interogărilor din MySQL poate fi redusă la un pattern care depinde doar de tipul operației(ștergere, editare, selectare,inserare). Prin urmare, cunoscând tehnici de Java Reflection, ele pot fi ușor generate atâta timp cât se mapează tabelele în Java cu exact aceleași nume ca în MySQL

***Scenarii de utilizare***

Aplicația trebuie concepută pentru utilizarea de către manager-ul unui depozit. În general, pașii de utilizare decurg simplu: utilizatorul deschide meniul principal și selectează opțiunea dorită, care îi va deschide un meniu cu un tabel în care se află informațiile legate de categoria cerută. După aceea, selectează operația dorită (de exemplu, inserare), eventual introduce date suplimentare și apasă pe un buton special. După aceste etape, el se poate întoarce în meniul principal să execute altă operație. Un exemplu de scenariu este următorul:

Scenariu de utilizare: inserarea unui nou client

Actor principal: utilizator (manager-ul depozitului)

Cazul de utilizare principal:

1. Se deschide meniul principal
2. Se apasă butonul pentur opțiunea de clienți
3. Utilizatorul scrie datele pentru clientul de introdus într-o zonă dedicată
4. Se apasă pe un buton pentru confirmarea datelor
5. Se realizează verificarea datelor introduse
6. Se face inserarea în tabelul corespunzător
7. Utilizatorul fie revine în meniul principal pentru a opera pe altceva, fie închide aplicația

Scenarii alternative:

* 1. Se introduc date incorecte
* Nu se introduce nimic în baza de date
* Utilizatorul este avertizat

Majoritatea cazurilor de utilizare se reduc laoperarea pe tabele din baza de date din interfața grafică, verificând în spate corectitudinea informațiilor introduse

Mai detaliat, diagrama cazurilor de utilizare este următoarea:

Diagram

Description automatically generated

# **Proiectarea aplicației**

Diagrama EER.

Deoarece se lucrează cu baze de date, trebuie să se asigure că modelarea ei respectă principiile domeniului acesta (fără relații many to many, o formă de normalizare destul de bună încât să nu creeze probleme operațiilor CRUD)

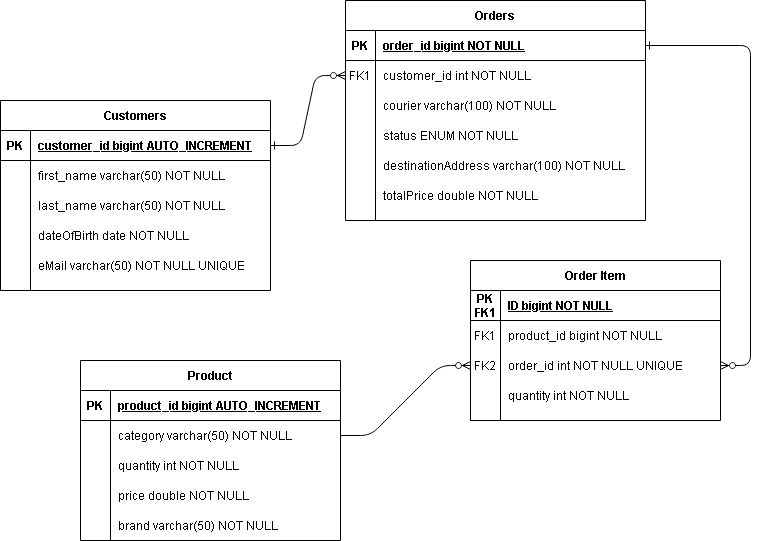


Diagrama UML și a pachetelor

Deși este realizată diagrama EER, aceasta nu cuprinde nici pe departe toate clasele din Java. Alături de clasele pentru entitățile din MySQL, trebuie să existe și clase care se ocupă cu interogările din baza de date, dar și pentru validarea acestora (ca să nu mai vorbim de interfață grafică și controllerele pentru ferestre)

A picture containing engineering drawing

Description automatically generated

Pattern-uri arhitecturale

Singleton Pattern. Este un mod de design OOP care presupune folosirea unei singure instanțe a unui obiect pentru rezolvarea unei probleme. În acest caz, ea a fost folosită pentru management-ul conexiunii aplicației cu baza de date, incluzând stabilirea unei conexiuni și închiderea acesteia sau a altor obiecte precum Statement-uri și ResultSet-uri

Layered Architecture

Pentru o organizare mai bună a claselor, proiectul a fost spart în mai multe pachete, fiecare ocupându-se de câte un aspect diferit al aplicației:

Pachetul model. Conține maparea în Java a entităților și tipurilor de date suplimentare(enum-uri) din baza de date

Pachetul dataAccess. Aici se află clasele responsabile de accesul datelor din baza de date și management-ul conexiunii

Pachetul business. Presupune validarea informațiilor înaintea introducerii lor în baza de date. Datele eronate ce pot fi introduse pentru fiecare clasă sunt următoarele:

|  |  |  |  |
| --- | --- | --- | --- |
| Clients | Orders | Product | OrderItem |
| * Se încearcă modificarea ID-ului cu orice valoare * Data nașterii are format invalid (yyyy-MM-dd) * Email-ul nu respectă un pattern cu litere@adresa.domeniu | * Se încearcă modificarea ID-ului * Se introduce o dată cu format incorect * Se introduce un preț cu pattern incorect: cu mai mult de 2 zecimale și cu mai mult de 15 cifre în total * Se încearcă modificarea manuală a prețului total | * Se încearcă modificarea ID-ului * Se introduce un preț cu pattern incorect: cu mai mult de 2 zecimale și cu mai mult de 15 cifre în total * Se introduce o cantitate care nu e număr întreg | * Se încearcă introducerea unei cantități mai mare decât cea din tabelul de produse din stoc sau a unei cantități care nu e număr natural * Se încearcă modificarea manuală a prețului total |

Pachetul presentation. Cuprinde partea de interfață grafică și controller-ul care face legătura între utilizator și aceasta

Algoritmi

Nu au fost folosiți algoritmi speciali în afară de parcurgerea variabilelor instanță ale unei clase și eventual verificarea unor Annotations

Structuri de date

Păstrarea obiectelor dintr-un ResultSet a fost realizată într-o listă deoarece nu este cunoscut de dinainte numărul lor

# **Implementare**

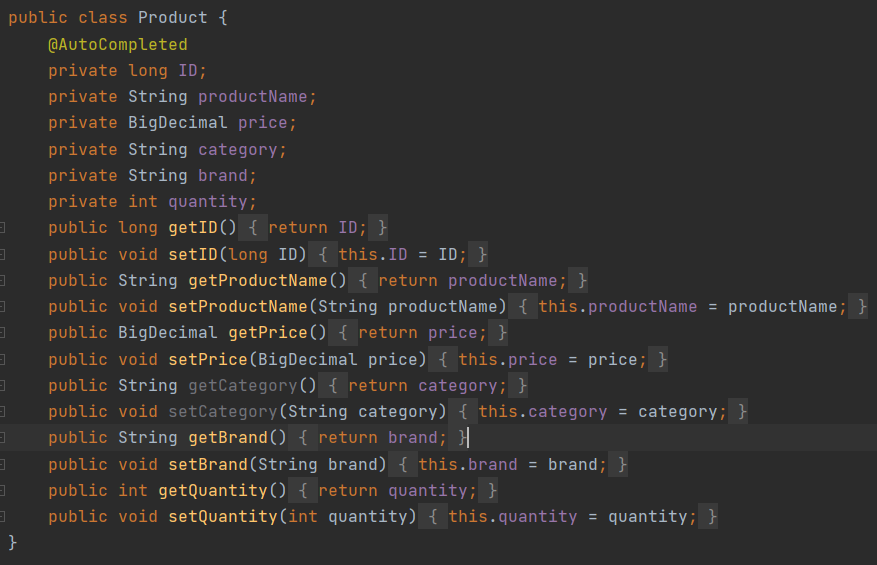
Aplicând ideile de proiectare, am obținut următoarele clase

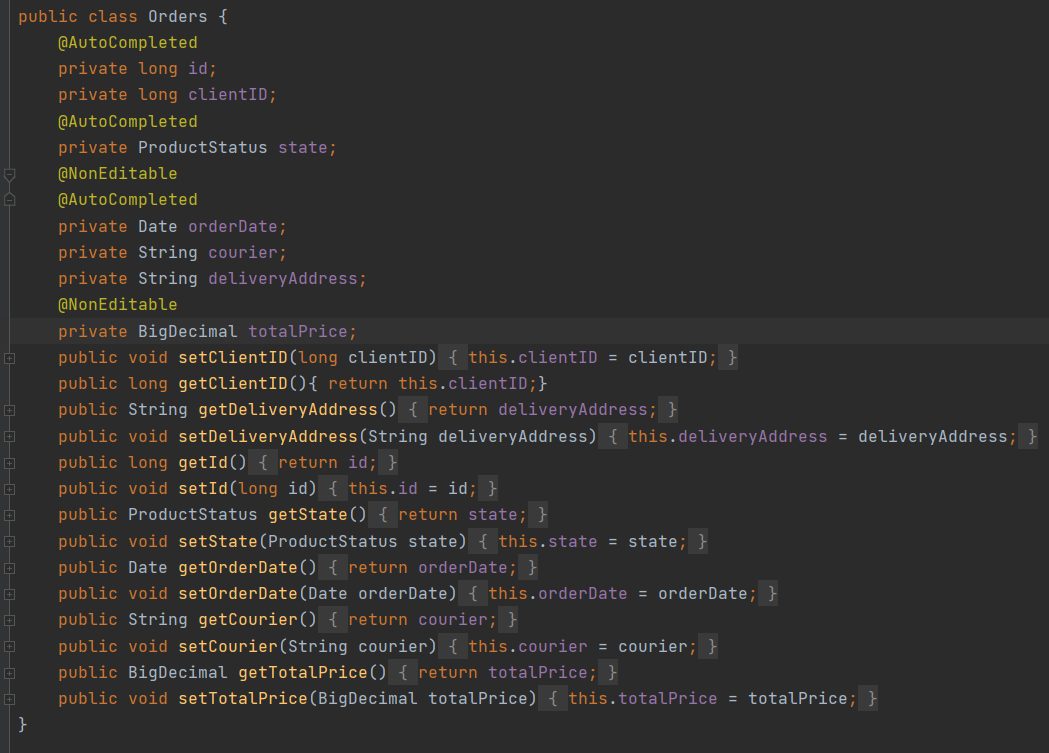
Pachetul model

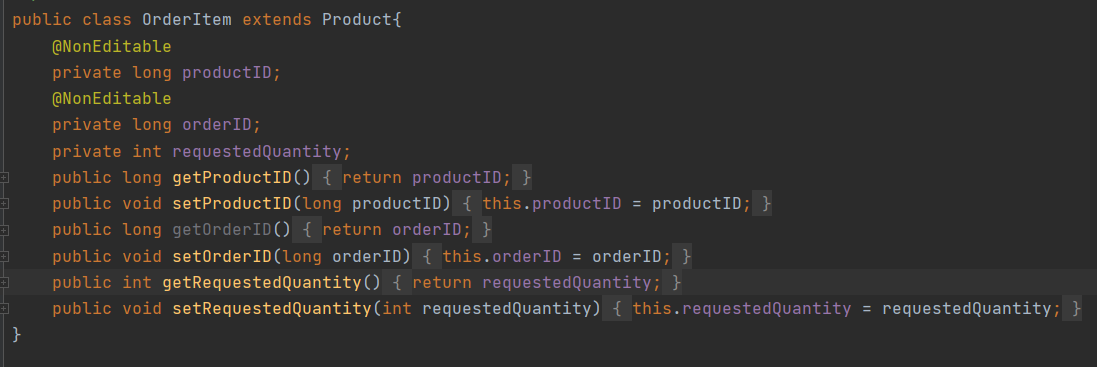
***model.entity***

Clasele corespondente tabelelor din MySQL, conținând câmpuri cu tipuri cât mai corespunzătoare pentru aceasta. Metodele lor sunt doar setters și getters



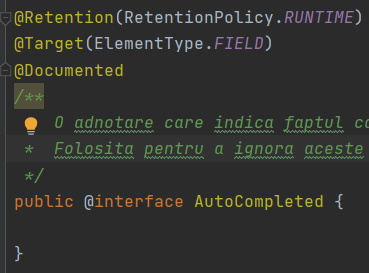




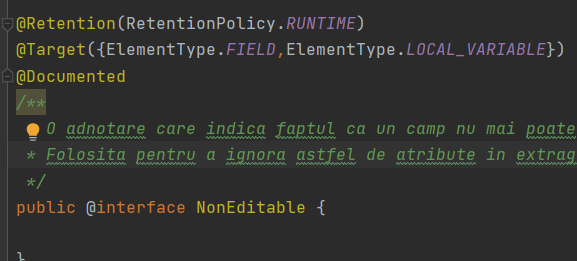


***model.misc***

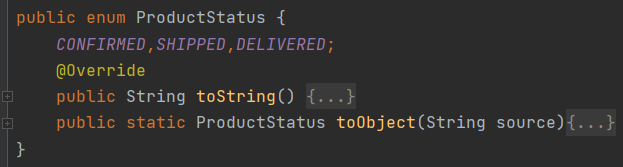
Cuprinde enum-uri și adnotări suplimentare pentru maparea în Java sau marcarea anumitor variabile instanță



Adnotarea AutoCompleted marchează dacă un câmp dintr-un tabel este completat automat (fie prin triggere, fie prin Auto\_incremented)

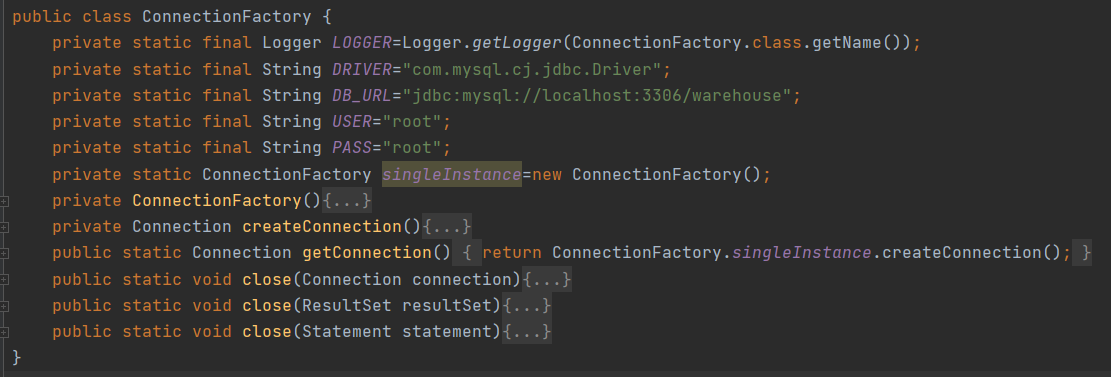


Marchează dacă un câmp se mai poate edita (de exemplu, nu ar avea sens să se poată modifica data plasării unei comenzi)

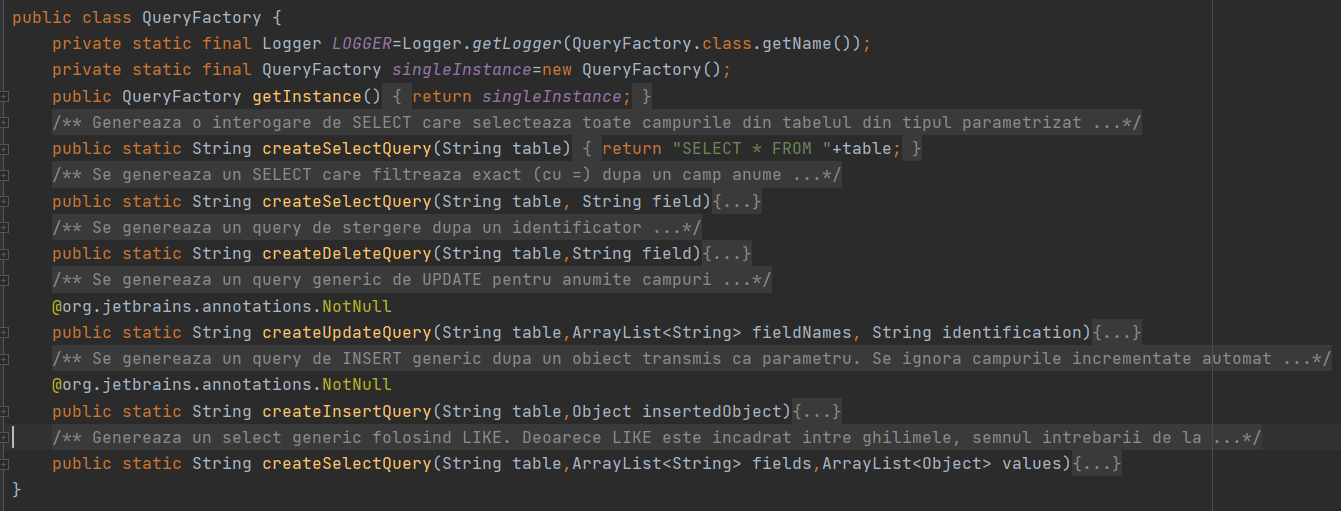


Enumerația corespunzătoare enum-ului din MySQL pentru starea comenzii

Pachetul dataAccess



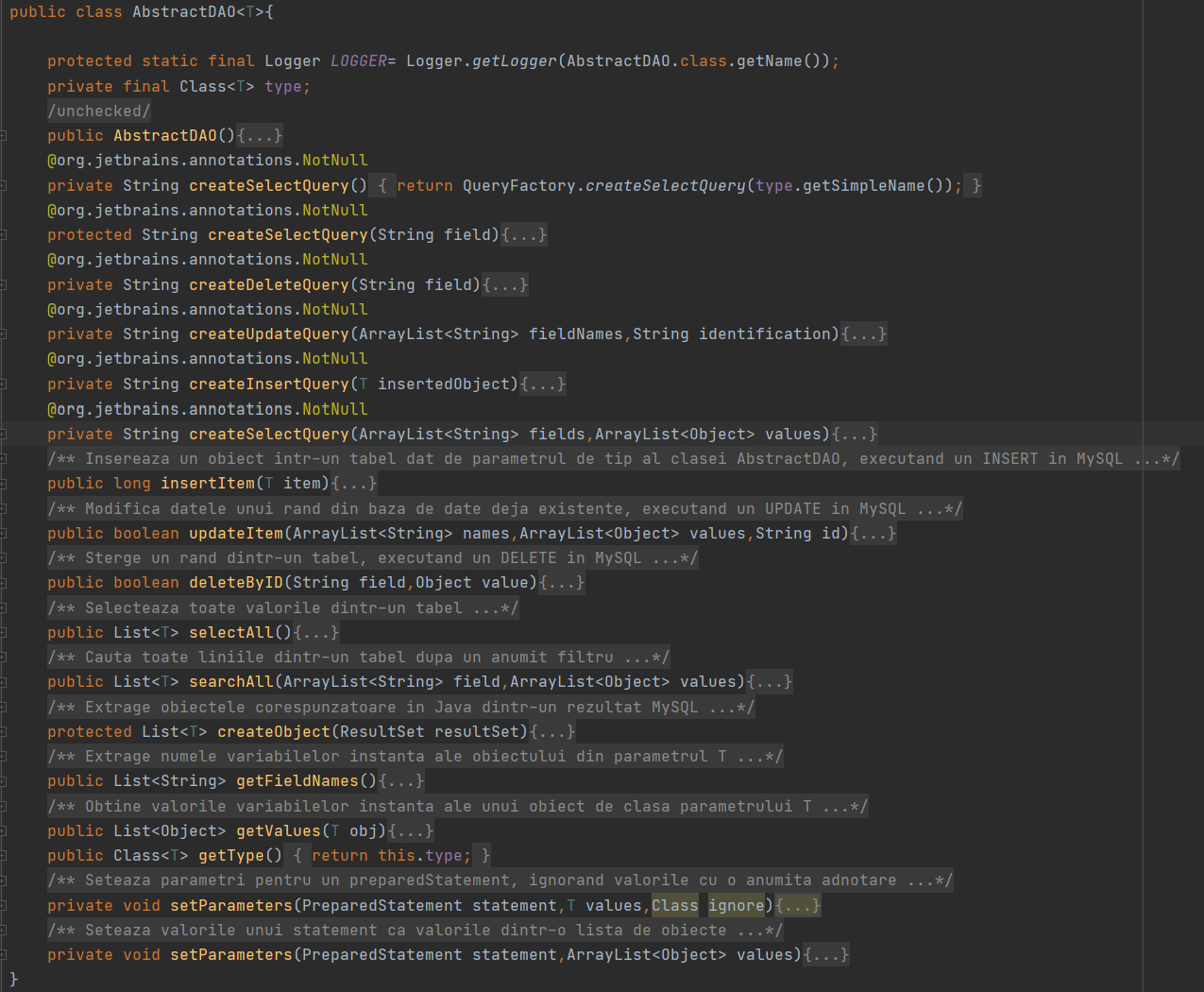
ConnectionFactory – este clasa organizată după Singleton Pattern. Ea păstrează instanța obiectului unic care se va ocupa de crearea unei conexiuni și închiderea acesteia sau a altor lucruri precum Statement-uri sau ResultSet-uri



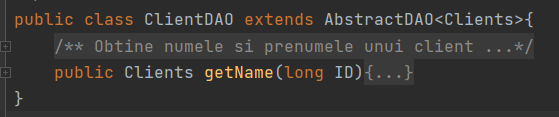
Clasă care se ocupă de generarea de interogări MySQL pe baza sintaxei SQL, fiecare metodă fiind denumită cât mai sugestiv.

***dataAccess.dao***

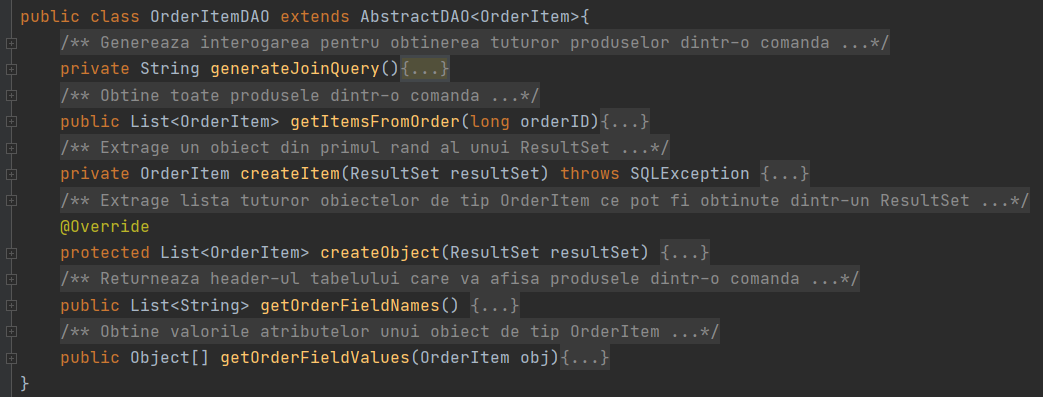
Sunt clasele care se ocupă cu interogările din baza de date



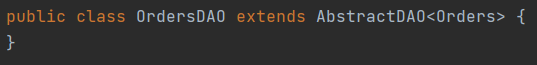
AbstractDAO – clasa parametrizată care conține metode pentru interogări generale (de exemplu SELECT \* FROM tabel) și care realizează o nouă instanță a unei clase de Java dintr-un ResultSet din MySQL. De asemenea, aici se află metodele care generează header-ul unui JTable, alături de informațiile de afișat pe un rând în acesta



Clasa de DAO specifică pentru clienți. Singura metodă este selectarea a doar numelui și prenumelui pentru completarea detaliilor dintr-o comandă



Clasa DAO specifică pentru operații pe tabelul OrderItem. În ea se găsesc mai multe metode particularizate deoarece ele mereu se selectează doar în contextul unei anumite comenzi. De asemenea, pentru ea este nevoie să se extragă separat câmpurile din superclasă, care nu sunt pur si simplu returnate prin getDeclaredFields()



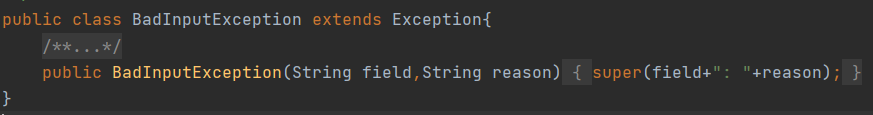
Clasa DAO pentru operații particulare pe tabelul Orders. Momentan toate acestea sunt acoperite de operații generale



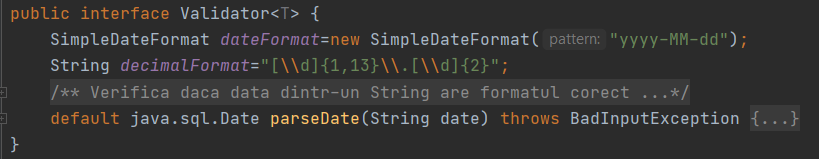
Clasa DAO pentru operații particulare pe tabela Products de produse în stoc. Și acestea sunt acoperite de operații generale

Pachetul business

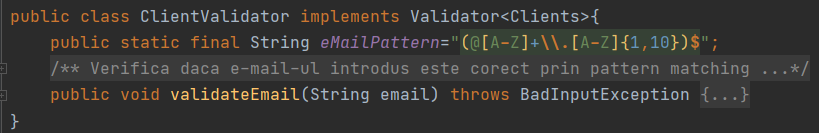
Se ocupă de validarea datelor, după corectitudinea discutată anterior



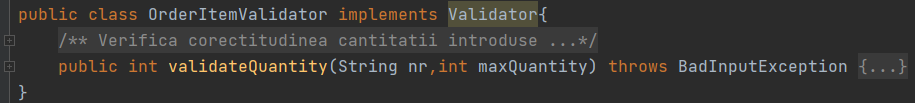
Clasa pentru excepțiile interceptate când datele introduse nu sunt corecte



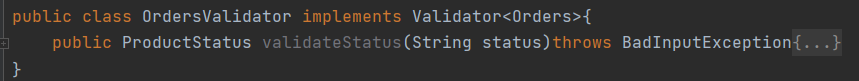
Interfața care se ocupă cu validarea datelor. Conține pattern-uri comune de verificat și o metodă default pentru verificarea formatului de dată



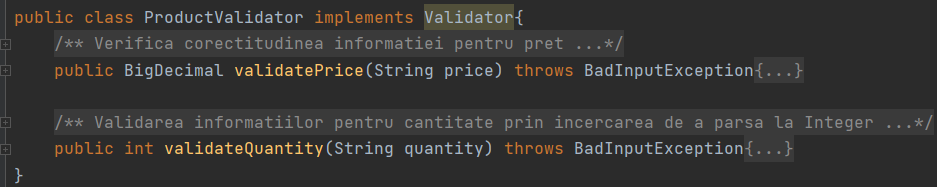
Clasa pentru validarea datelor pentru clienți. Din moment ce data nașterii este acoperită de metoda default din Interfață, rămâne doar verificarea e-mail-ului.



Clasa pentru validarea datelor pentru produsele comandate. Din moment ce circumstanțele fac ca prețul total să fie întotdeauna corect, rămâne doar verificarea cantității, inclusiv depășirea celei din stoc



Validează datele de introdus în Orders. Din nou, data e acoperită de metoda default



Validează datele de introdus în tabela de produse în stoc. Trebuie verificate formatul prețului și cantitatea

Pachetul presentation

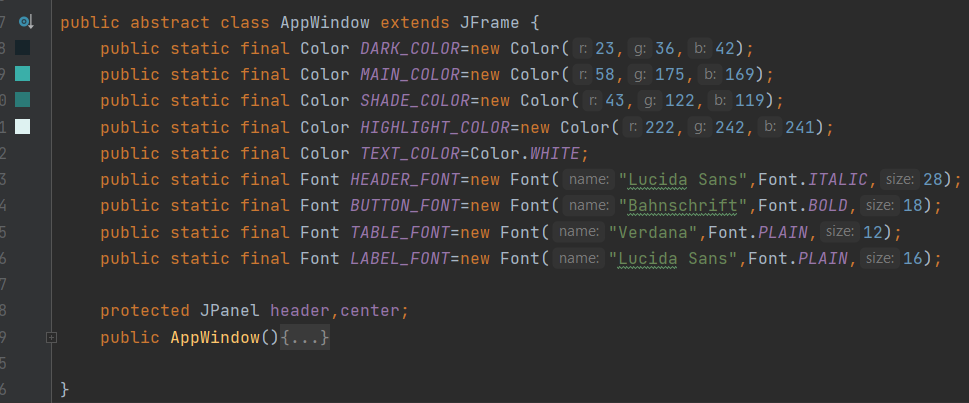
Clasele pentru interfața grafică, inclusiv controller-ul

***presentation.view***

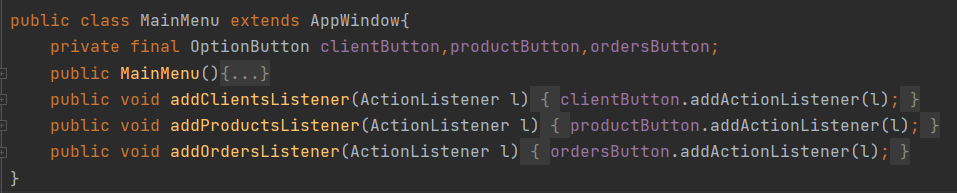
Clasele pentru ferestre și componente speciale

***presentation.view.windows***

Ferestrele pentru interfața grafică, incluzând și un pachet pentru ferestrele de opțiuni



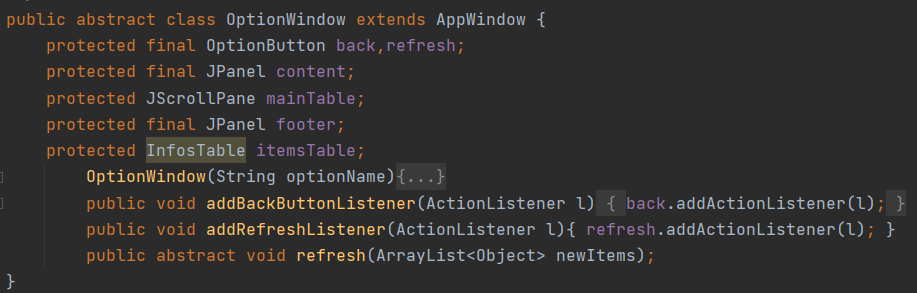
Fereastră generică pentru aplicație, conținând inițializarea și constante de culori și font



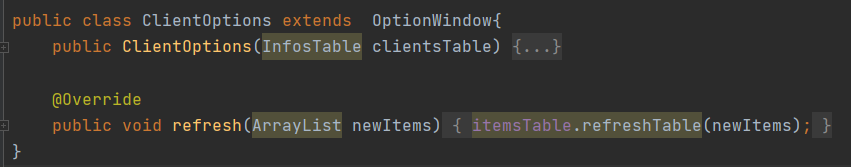
Meniul principal de unde se selectează tabelul pe care se dorește să se opereze

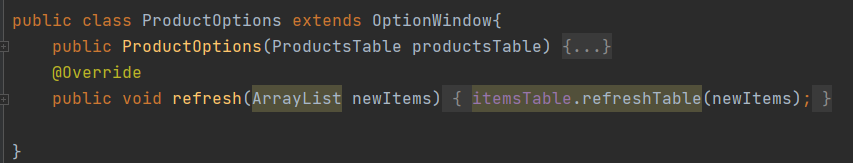
***presentation.view.optionMenus***

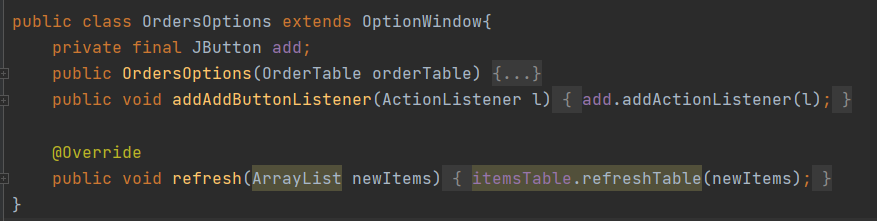
Pachet ce include ferestrele de operare pe tabele + câteva ferestre ajutătoare



Fereastră generică pentru gestionarea operațiilor. Restul ferestrelor din acest pachet sunt doar derivate de la aceasta. Metoda refresh are rolul de a reîmprospăta tabelul central pe care se fac operații

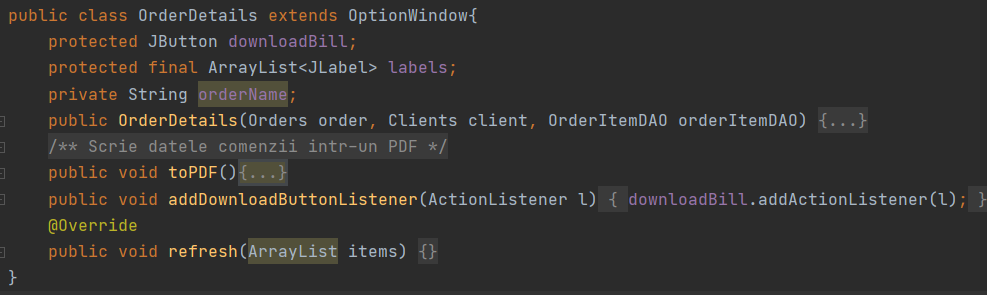








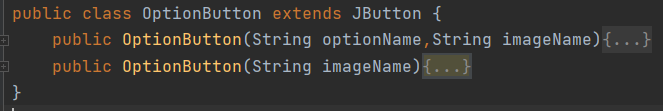
Fereastră specială pentru a adăuga o comandă, conținând mai multe tabele



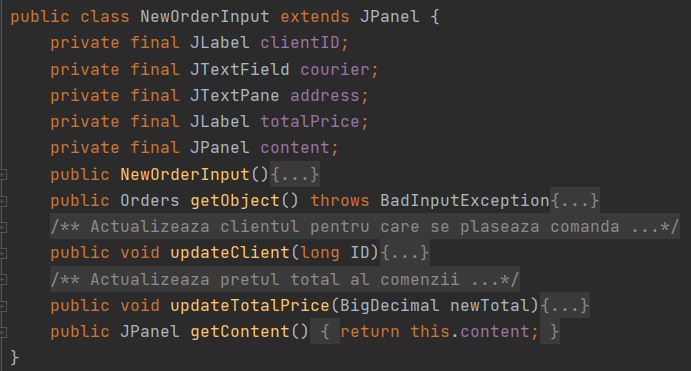
Fereastră pentru detaliile unei comenzi, având un buton pentru descărcarea chitanței

***presentation.view.utility***

Câteva componente grafice mai speciale

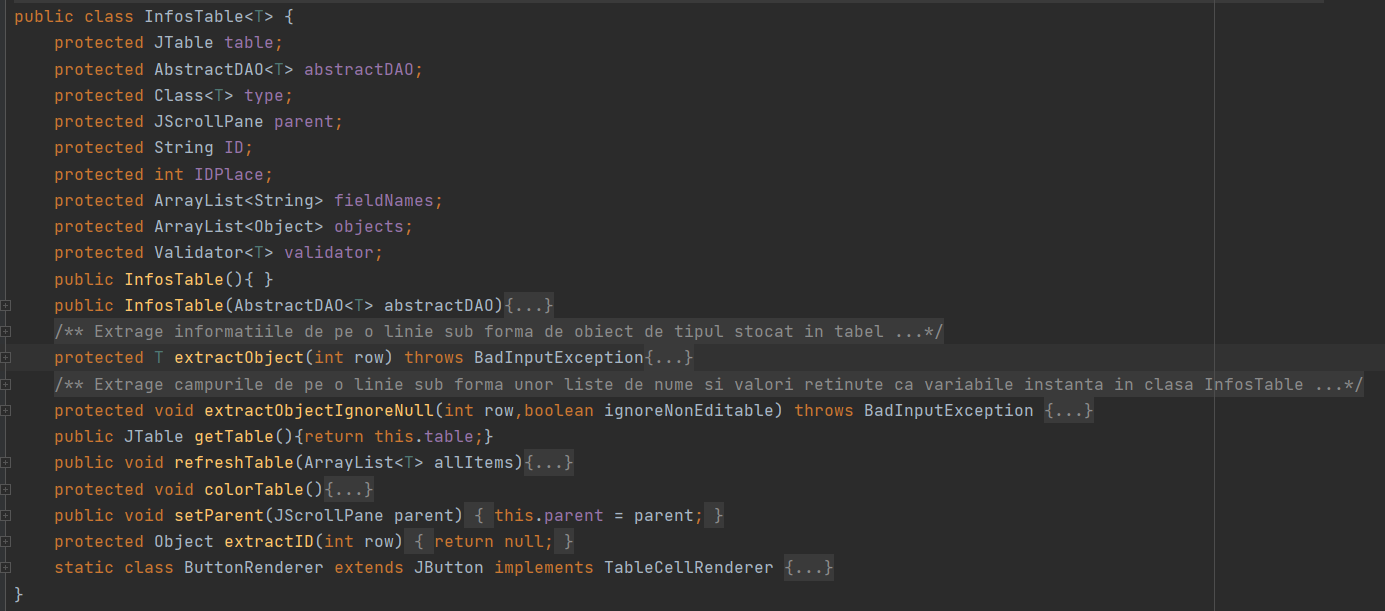


Butoanele pentru opțiunile din meniul principal + butoanele de navigație (back, refresh)

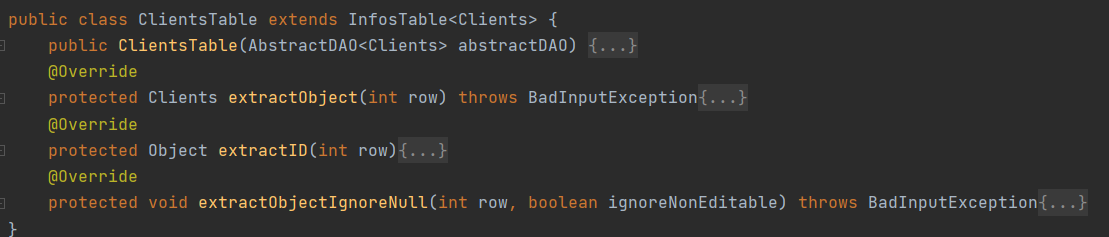


Clasă specială pentru introducerea datelor atunci când se plasează o comandă

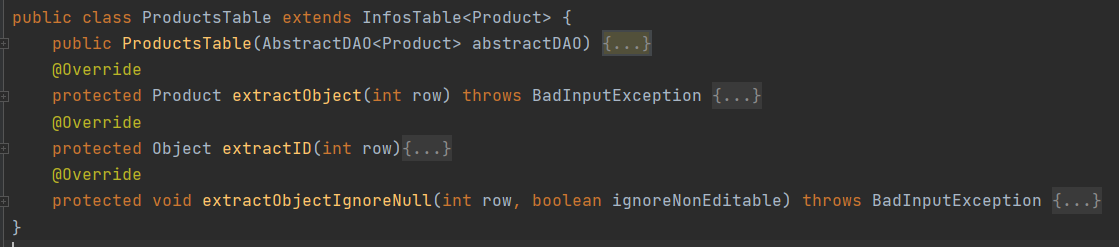
***presentation.view.utility.tables***



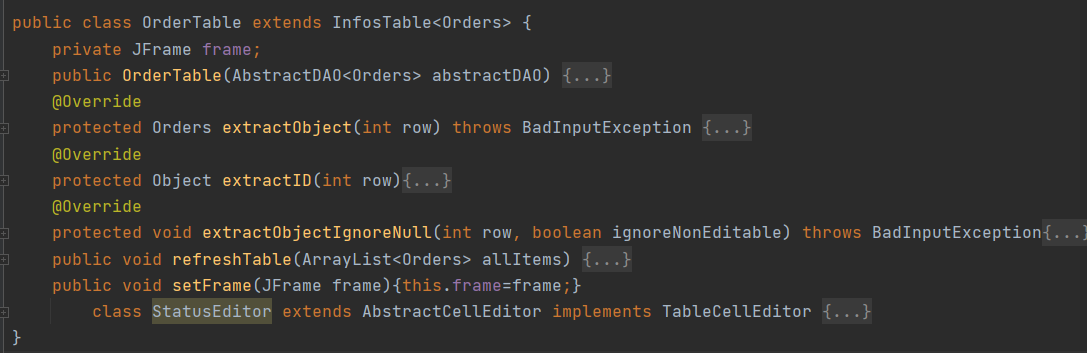
Tabel general pe baza căruia se personalizează altele



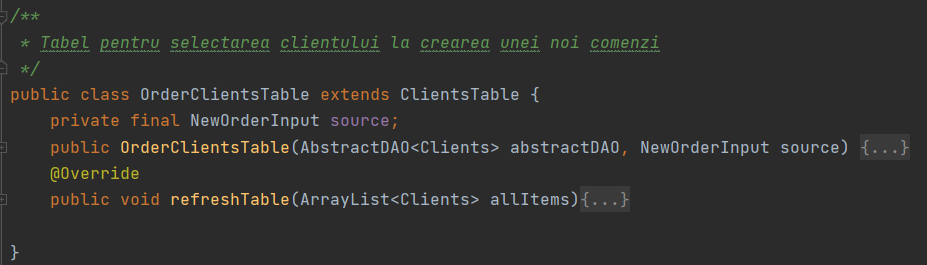
Tabelul pentru gestionarea tuturor clienților. Doar se suprascriu metode de extragere a obiectului pentur validarea datelor

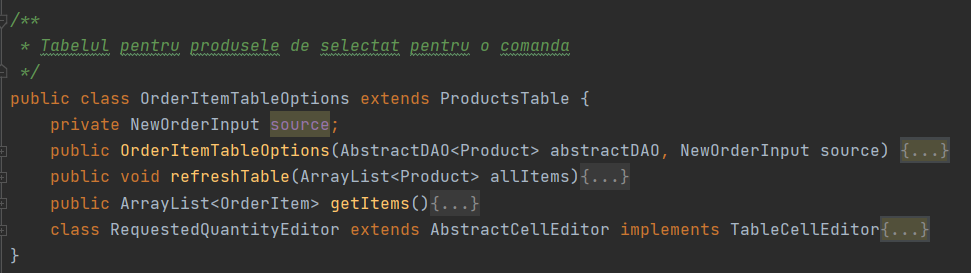


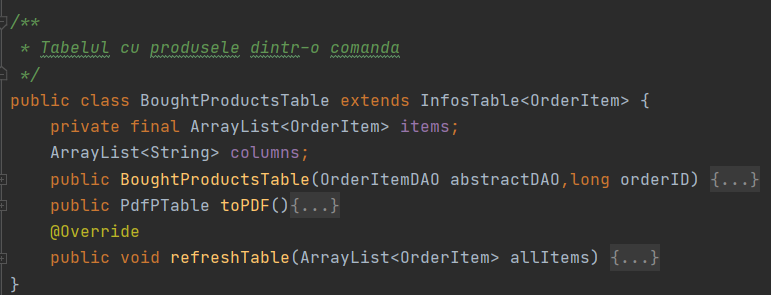
La fel ca mai sus, doar că pentru produse în stoc



Tabelul pentru gestionarea comenzilor (în afară de adăugare, care se face dintr-o fereastră separată)

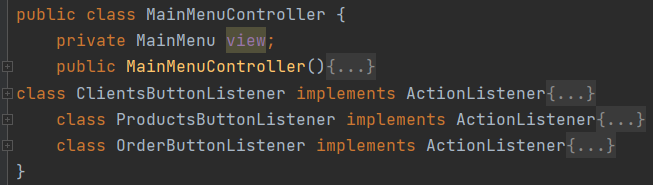






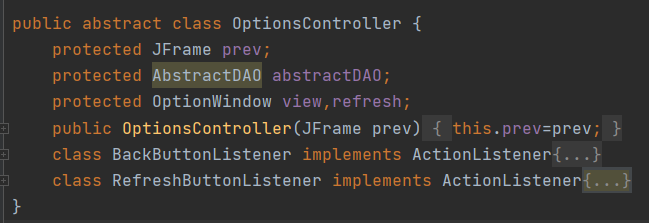
***presentation.controller***

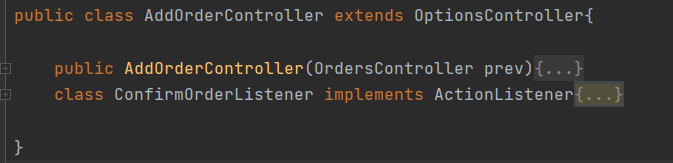
Conține toate controller-ele corespunzătoare fiecărei ferestre

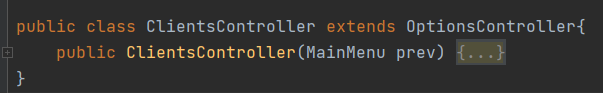


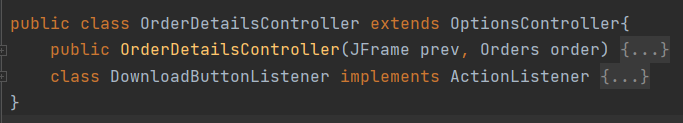
***presentation.controller.optionsControllers***

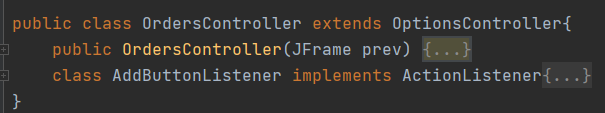
Controller-ele din ferestrele de opțiuni, din moment ce doar sunt derivate de la aceeași clasă abstractă

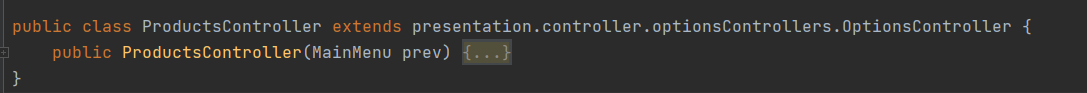






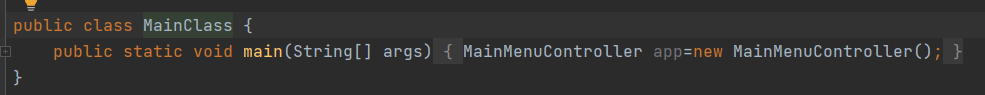






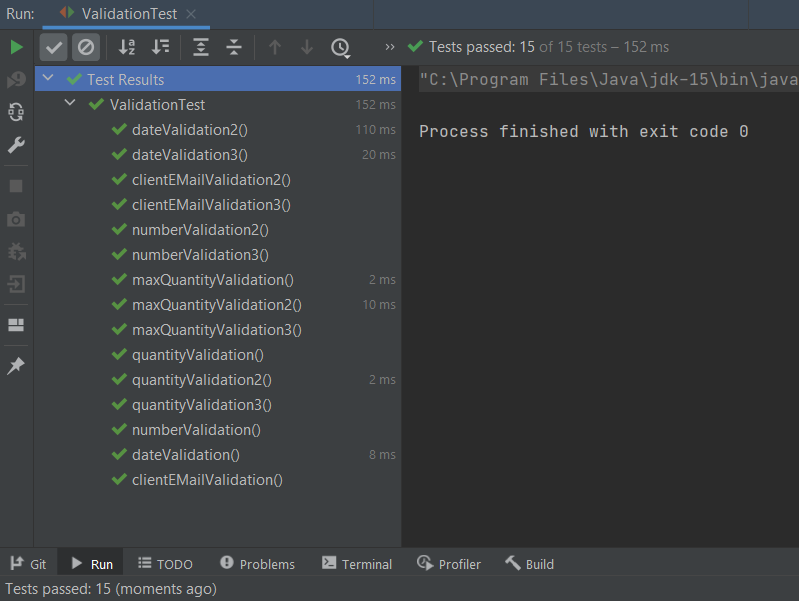
Pachetul default

Aici se află doar clasa care inițializează aplicația într-o metodă main

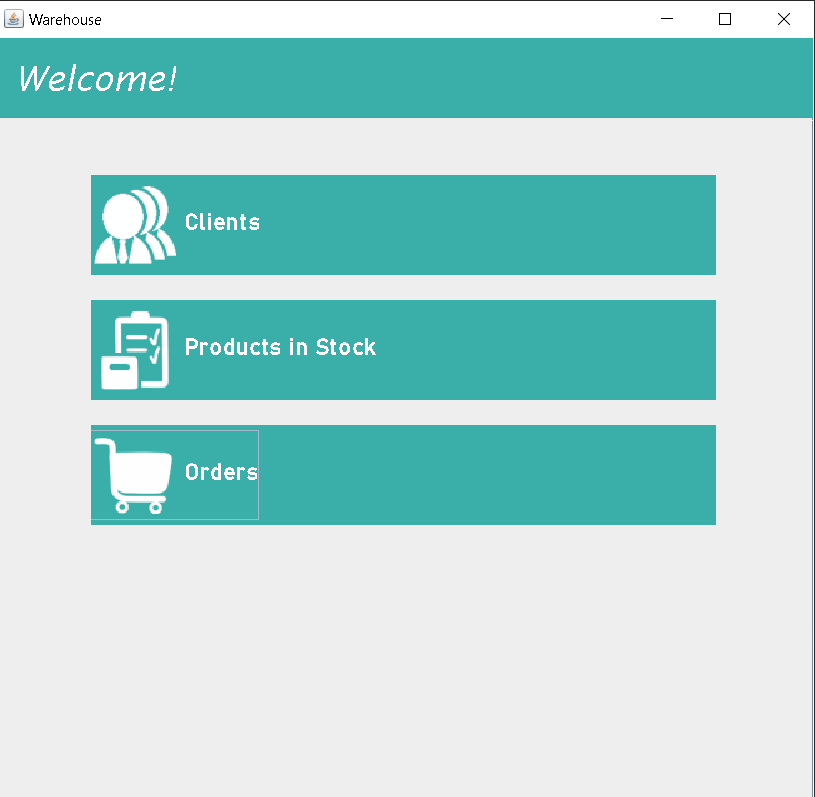


# Testare și rezultate

Înainte de a pune în funcțiune aplicația finală, s-au realizat mai multe teste cu Junit. În principiu s-au testat clasele de validare din business logic pentru asigurarea că datele sunt introduse corect în baza de date, realiz



La rularea aplicației, se va deschide următorul meniu principal:

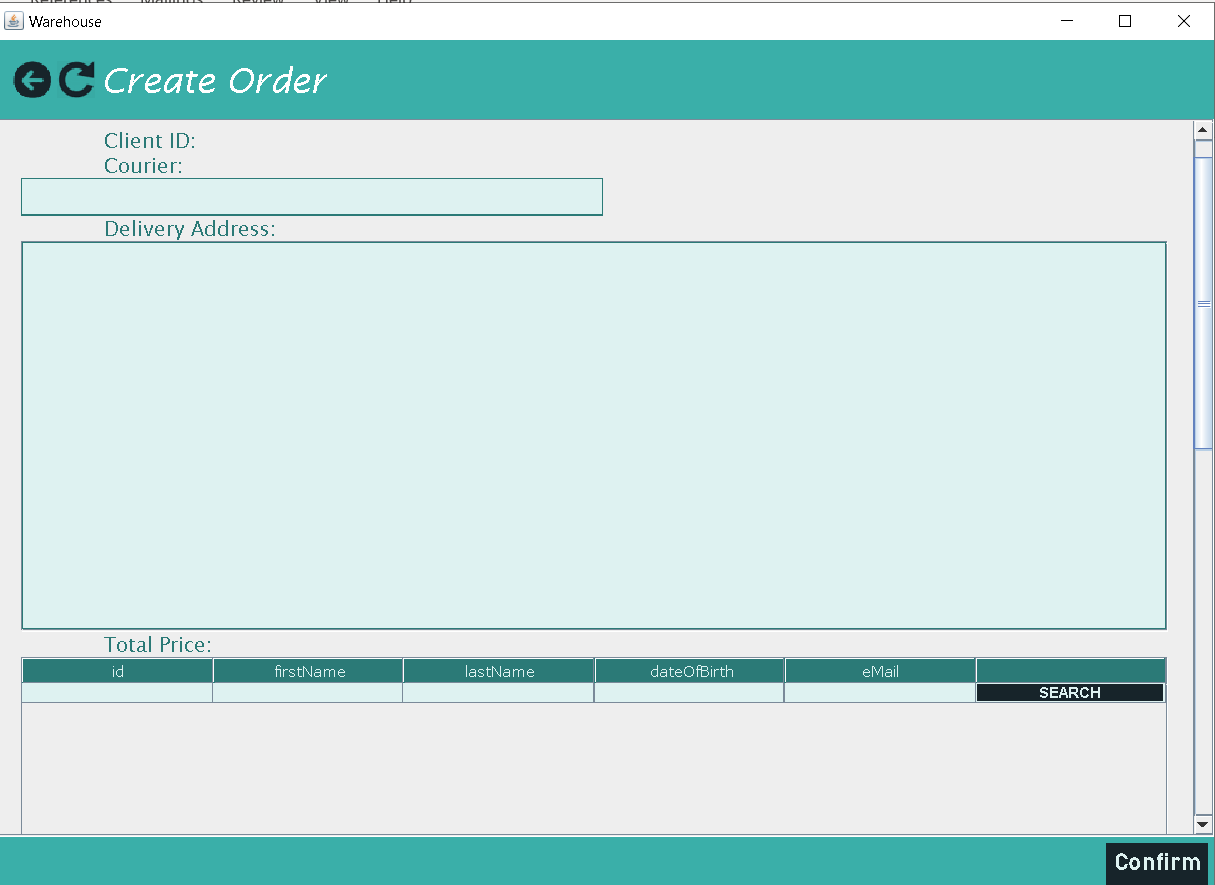


Pentru operarea pe un tabel, se va apăsa pe opțiunea corespunzătoare. De exemplu, dacă se vrea gestionarea clienților, se apasă pe butonul de Clients, care duce la fereastra:

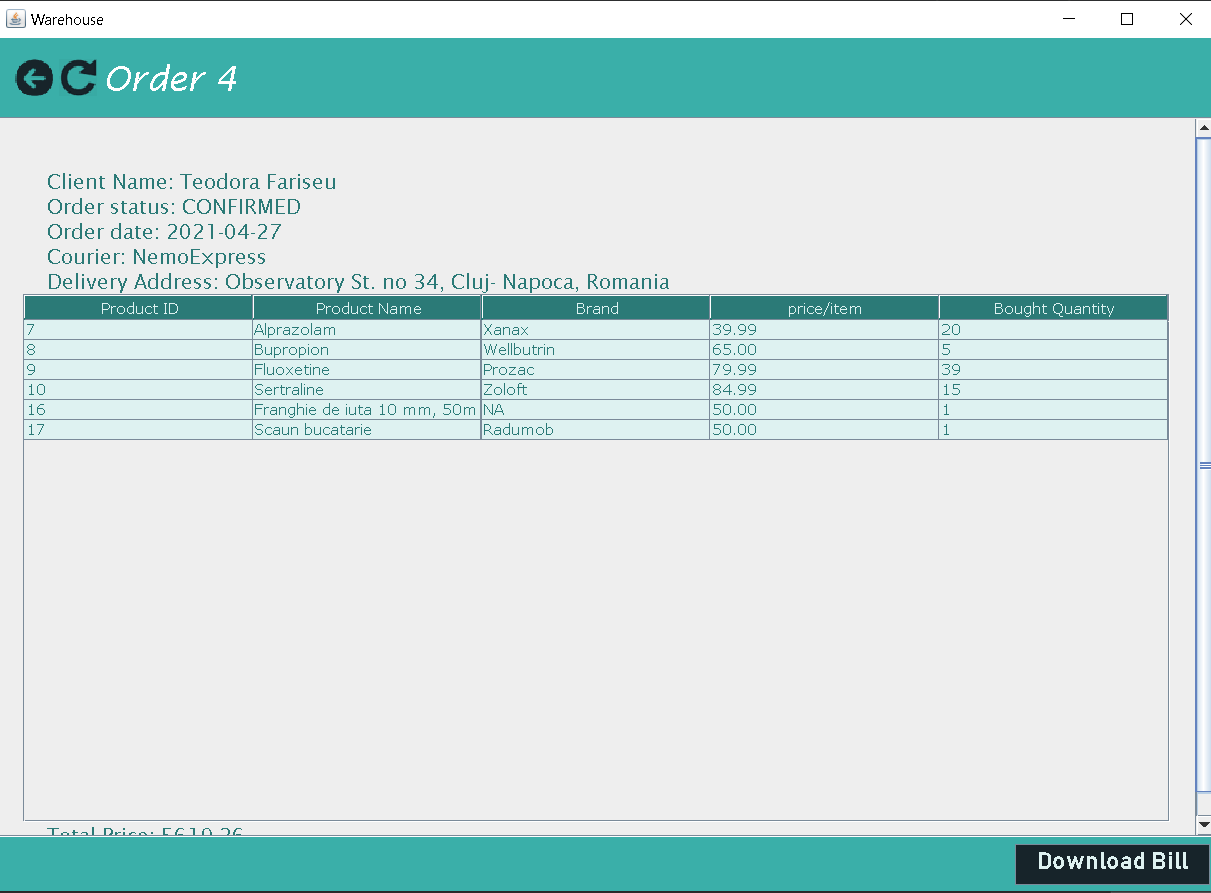


Pentru comenzi, există 2 meniuri adiționale:

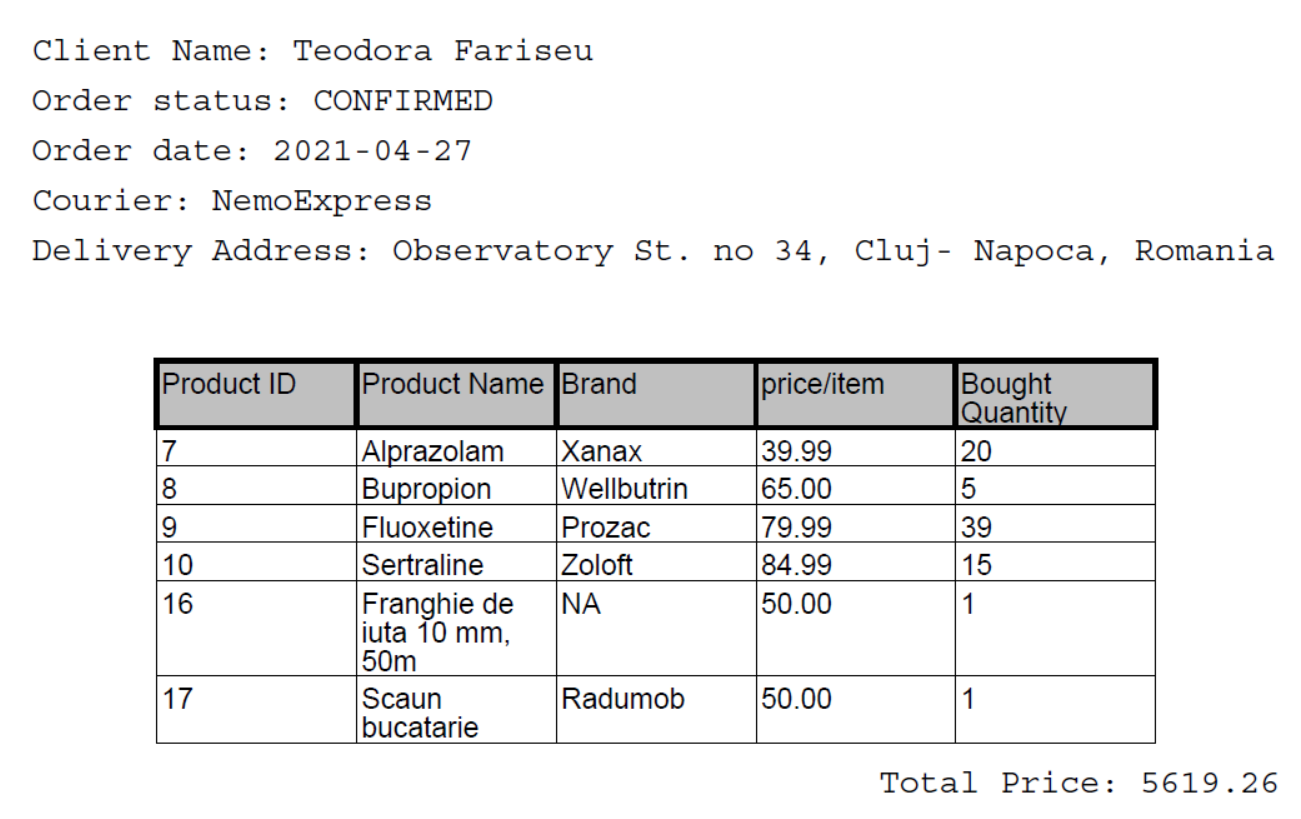
Meniul de adăugare de comandă



Meniul de detalii comandă, de unde se poate descărca și chitanță:



Exemplu generare chitanță:



# Concluzii

Acest proiect a fost un ajutor extrem de mare în obișnuirea cu Java Reflection. Java Reflection este o tehnică unică, specifică pentru Java, însă foarte utilă, permițând până și abstractizarea extragerii unei clase dintr-un ResultSet. Invocarea dinamică a metodelor și obținerea anumitor marcaje puse pe o variabilă instanță au o foarte mare aplicabilitate în proiectele cu anumite lucruri foarte repetitive, precum o aplicație pe baze de date, unde interogările și obținerea unui obiect din aceasta au un pattern similar

Un al lucru dobândit din acest proiect este lucrul cu Annotations, care din nou, sunt foarte utile, în special folosite în Reflection pentru a marca anumite lucruri despre o variabila instanță, metodă sau clasă.

În ceea ce privește lucrul cu o bază de date din Java mi-am reîmprospătat memoria legată de crearea și închiderea unei conexiuni cu o bază de date, alături de cunoștințe de bază despre interogări (structura lor, apelul lor din Java, crearea de triggere)

De asemenea, am realizat ca JTable-urile sunt mult mai complexe și mai versatile decât credeam. În încercarea de a personaliza JTable-urile cu TableCellRenderer și TableCellEditor, am realizat că acestea pot înlocui sau pot face editarea mult mai ușoară decât JPanel cu mai multe componente editabile

Și crearea de documente PDF a fost ceva interesant. Deși în trecut am lucrat cu PdfBox (care e puțin mai low-level în sensul că nu are metode pentru tabele de la sine, fiind nevoie de încă [o dependență Maven pentru asta](https://github.com/vandeseer/easytable)), proiectul acesta m-a facut să realizez că nu am nevoie de licență de iText ca să-l folosesc la proiecte.

Validând datele de introdus, am fost ținută la curent și cu regex și moduri de extragere a unei date

Un regret este faptul că nu am stat să studiez mai aproape Java Annotations. Adnotările au foarte mare potențial, putând avea și valori atribuite în spate, iar dacă mai analizam puțin, poate reușeam și să abstractizez extragerea unui OrderItem din tabel

# Dezvoltări ulterioare

**Interfața grafică**. Din nou, lucrând în grabă, nu prea am avut timp să mă axez așa mult pe interfață, în afară de a urmări ușurința utilizării sale. De asemenea, dacă nu se impunea lucrul cu JTable, aș fi încercat să fac casete de componente editabile

**Multithreading**. Deoarece se realizează mai multe lucruri simultan care chiar pot fi costisitoare(comunicarea cu baza de date, gestionarea interfeței grafice) și având în vedere că am cunoștințe legate de thread-uri din tema trecută, aș fi putut să organizez aplicația pe mai multe fire de lucru. Din păcate, nu am avut timp suficient pentru acest lucru.

**O organizare mai naturală a bazei de date.**Nu am încercat să fac baza de date prea complexă deoarece nu acesta a fost scopul proiectului

# Bibliografie

<https://docs.oracle.com/javase/tutorial/uiswing/components/table.html>

<https://www.baeldung.com/java-custom-annotation>

<https://www.tutorialspoint.com/itext/itext_adding_paragraph.htm>

<https://www.codejava.net/java-se/swing/jtable-simple-renderer-example>

<https://docs.oracle.com/javase/7/docs/api/javax/swing/table/TableCellEditor.html>