Tema 3

Order Management

UTCN, Orsan Tudor, Grupa 30225

1. Obiectivul temei:

Obiectivul principal al acestei teme este sa folosim baze de date pentru a stoca informatia, si sa putem sa creeam o conexiune intre ide din java si acea baza de date relationala (precum sql). Noi dorim sa folosim clasele ca si cum ar fi niste tabele, si sa putem sa efectuam operatii de baza pe ele (precum insertul, updateul, deleteul, find, sau find all).

Obiective principale : Sa folosim baze de date relationale pentru stocare informatie, minim cele 3 tabele Client, Produs si Order. Pe langa, sa folosim Javadoc pentru documentarea claselor (deci o comentare mai detaliata). Sa facem GUI in care sa cuprindem mai multe aspecte: Sa avem un window pentru client, in care sa punem toate operatiile specifice lui. La fel sa facem pentru Product, si dupa pentru Order. Insa pentru order avem de ales campuri mai importante: sa selectam id la un client, si un id la produs, si sa stabilim atunci relatia intre cei 2. Adica clientul cumpara un stock de produse. Si dupa ce cumpara acea cantitate mentionata, se decrementeaza din stockul ramas al produselor. In cazul in care nu mai sunt atat de multe produse de cumparat cate doreste clientul, se da un mesaj de understock (si nu poate cumpara produsele). Cand afisam datele din aceste 3 windowuri, sa le afisam cu Jtable, adica sa semene cu tabelele din sql. Dupa GUI, mai avem de folosit java reflection, si acolo sa putem folosi mai multe tabele la aceleasi operatii, prin faptul ca facem metodele generice, si in timpul run-ului, ne dam seama de proprietatile clasei apelante (de fielduri, etc...).

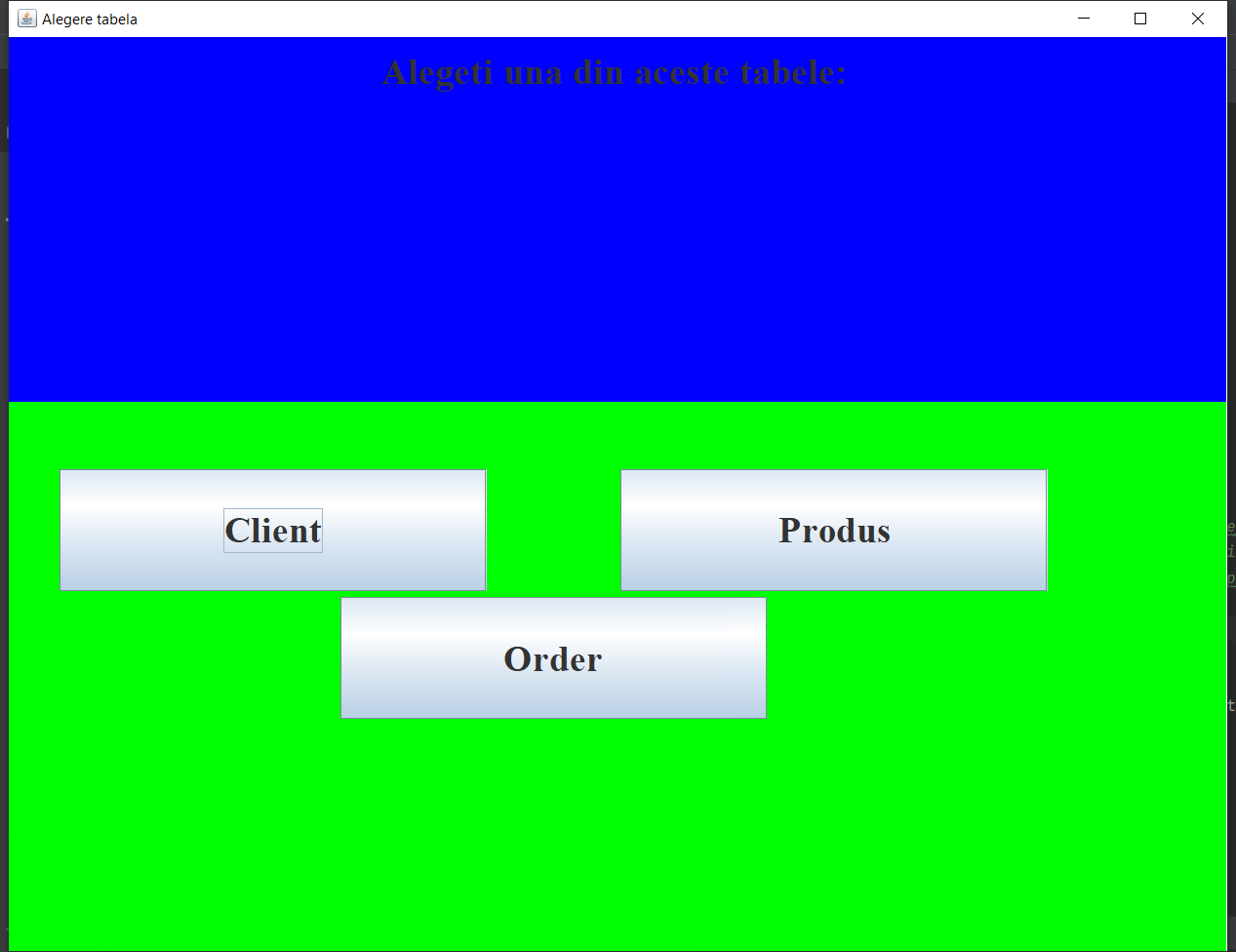
Alte obiective ce pot fi considerate principale, insa le voi considera secundare: Sa avem Layered Architecture, adica sa impartim pe mai multe pachete, in functie de logica fiecarui pachet si de legatura intre ele, conteaza cine are acces la cine. Sa facem un bill, in care afisam informatii despre orderul inserat in tabela. Am decis sa scriu intr-un fisier, care apare in folderul problemei (se genereaza mai mult de 1). Sa folosim java reflection mai in detaliu, si anume sa facem dinamic queriurile, in timpul runului, avand parametrii generici si extragand informatii, dupa cum ziceam, in timp ce se ruleaza programul.

1. Analiza problemei / cazurile de utilizare:

Pentru ca avem multe windowuri la aceasta tema, apar mai multe use caseuri, care sunt totusi asemanatoare intre ele:

Cazurile de functionare ale programului sunt acestea:

- In primul rand, avem primul window, cel in care alegem intre client, produs si order. Aici nu avem de introdus date de intrare, ci avem doar de ales un buton din acestea 3. Dupa care, vom fi dusi la unul din aceste 3 windowuri.



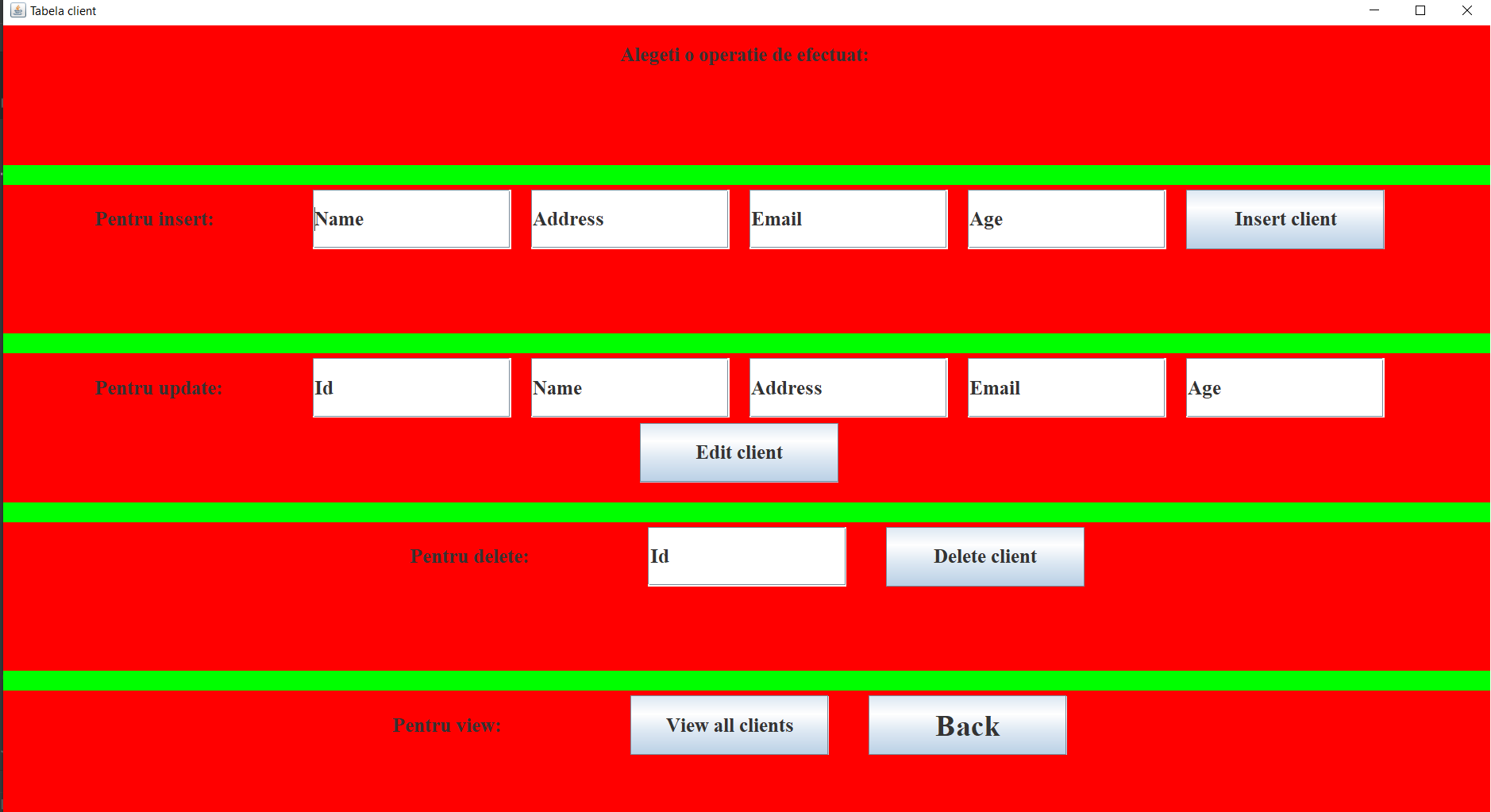
- Urmatoarea situatie este cea in care ne aflam in unul din cele 3 windowuri (care sunt foarte asemanatoare intre ele). Acum vrem ca pe acel window, sa executam o operatie specifica tabelei, si anume add, update, delete, findAll pentru client, produs, si pentru order doar insert (si eventual findAll). Voi intra in detaliu la fiecare din aceste operatii (care sunt acelasi concept la toate 3 windowurile):

- Pentru add, avem nevoie de toate campurile, inafara de id tabela, deoarece noi avem AI la tabele selectat, adica atunci cand se insereaza o linie in tabel, se auto incrementeaza idul, deci tot timpul se stie urmatorul id, nu trebuie dat ca parametru (deci nici la constructor nu dorim). Deci avem un buton de add, si text fielduri pentru

- Pentru update, este nevoie de toata informatia de la insert, deoarece vom suprascrie peste datele existente noile date, dar acum avem nevoie si de un ID, deoarece vrem sa stim la cine facem updateul. Deci avem alte text fielduri ca mai sus, dar in plus si un id, pentru ca avem nevoie de acel id.

- Pentru delete, nu avem nevoie de text fielduri de introdus, deoarece noi vrem sa scoatem, nu sa introducem. Si atunci, noi doar dam un id, si asa stim exact pe cine dorim sa scoatem, fiecare client/produs are un id unic.

- Pentru find all, nu avem nevoie de nici o intrare in plus, deoarece cu acest buton vom accesa toate liniile din baza de date de la tabela respectiva, si atunci nu vrem un id specific, sau alte date de intrare. La acest find all, avem un alt view Jtable, care se deschide si arata datele gasite in baza de date. (Nu am buton de find simplu, deoarece nu m-am gandit ca este folositor, cu find all se pot vedea deja toate datele mai rapid)



Cerintele non-functionale ale programului sunt:

-Ar trebui sa fie usor de folosit de catre utilizator, sa fie clar unde trebuie sa introduca date, cum functioneaza si unde se afiseaza rezultatul.

-In caz de erori, sa fie evidentiat faptul ca avem acea eroare, si sa cunoasca utilizatorul faptul ca trebuie sa schimbe valorile de intrare, ca sa corespunda la cerintele problemei.

-Sa poata fi reutilizabil, daca se doresc alte date de intrare, sa se poata face accesibila aceasta optiune cat mai usor.

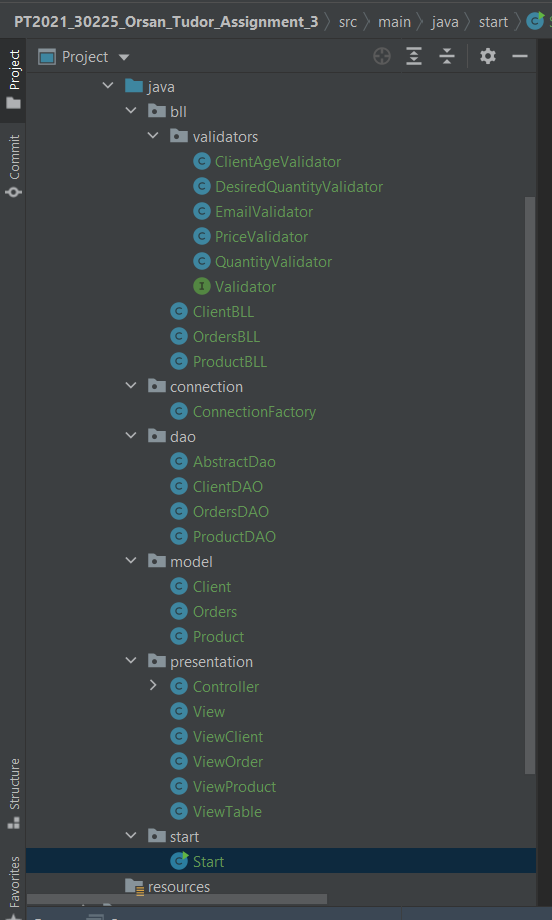
- Claritatea datelor este un mare avantaj la un astfel de GUI, pentru ca avem multe date, si vrem ca ele sa fie cat mai usor de inteles si cat mai usor de accesat de catre utilizatori.

Use caseurile le-am explicat (mai pe liber) in randurile de deasupra, deci nu voi intra din nou in detalii. In general, avem userul, care interactioneaza cu text fielduri si butoane, pentru a gestiona in mod cat mai eficient baza de date. Detaliile legate de butoane, operatii, text fielduri sunt deja explicate.

1. Proiectarea problemei:

Pentru proiectarea problemei, am sa intru in mai multe aspecte:

Primul aspect: Pachetele si structura lor:

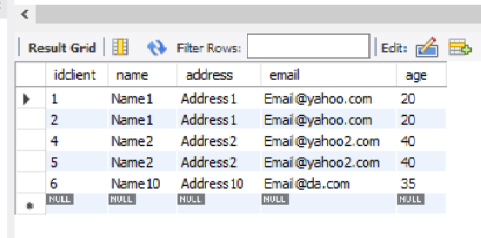


Ca de obicei, avem mai multe nivele de functionare. Avem o structura denumita Layered Architecture, ce beneficiaza de urmatoarele mari bucati: dataAccessLayer, unde se face conexiunea cu baza de date, si se extrag datele folosind reflection, cel mai jos nivel, avem business layer, unde ne focusam pe claritatea si testarea datelor, adica pe datele extrase la data acces layer, dupa care avem modelul, adica tabelele cu care lucram. Ele, de asemenea sunt folosite de toata lumea. Ca sa lucrezi pe tabele, trebuie sa cunosti de dinainte tabelele, si chiar daca avem metode bazate pe java reflection, pana la urma trebuie sa stim cine apeleaza, deci sa stim tabela anume. In final, pentru prezentarea datelor cu ajutorul GUI, avem nevoie de cam toate datele de mai sus. Avem nevoie de tabele, de logica cu care am extras si testat datele, pentru a putea sa le afisam cat de eficient si corect putem. (Deci view are acces la partea logica, iar cea mai high acces class este Controller, pentru ca avem acces la logica, dar si la view). Cu aceste 4 mari parti, putem creea o structura flexibila, dar si cu o buna comunicare intre componentele sale.

Al doilea aspect: Fieldurile

Fieldurile sunt private, in majoritatea claselor, pentru a putea exista o incapsulare mai puternica a codului. Pentru toate fieldurile importante si folosite si altundeva decat in clasa proprie, am creat setteri si getteri. Pentru java reflection, este destul de important faptul ca am creeat setteri, pentru ca fara ei nu am fi putut sa accesam si sa schimbam datele si sa le schimbam (cu invoke).

Al treilea aspect: Baza de date relationala:



Pentru aceasta tema, avem nevoie nu doar de clase, ci si de tabelele din baza de date relationala. Noi in aceste tabele dorim sa stocam datele, clasele sunt doar o modalitate de a mapa direct datele (si deci de a le introduce mai usor). Pentru ca mapam datele, avem nevoie de acelasi nume pentru numele clasei si numele tabelei, cat si la fieldurile lor. Folosim acest aspect cel mai tare la java reflection, pentru a putea gasi result seturile pe datele noastre exacte (care au si acelasi tip), dar aceasta mapare face si mai lizibila legatura intre cele 2. O baza de date ne ajuta cel mai tare cu spatiul de stocare, deoarece acesta este scopul ei principal. Noi dorim sa avem multa informatie stocata intr-o asemenea structura, iar lucrul cu clase ne ajuta sa simplificam doar procesul de stocare si citire al informatiei (prin acel GUI, ce folosest statemeturi si queryes in mod mai rapid).

Al patrulea aspect: Java Reflection:

Probabil cel mai important aspect al acestei teme probabil ca il constituie java reflection, deoarece cu operatiile de baza add, update, etc... ne-am mai intalnit si in trecut, dar java reflection este un concept nou. Faptul ca nu trebuie sa cunosti de dinainte multe informatii ca sa poti scoate datele importante dintr-un obiect este un avantaj imens, sau cel putin o modalitate buna de a gandi. Cu aceasta modalitate, putem sa luam doar numele tabelelor, si sa le mapam rapid, in timpul executiei programului, extragand metadate din clasa. Noi avand clasele creeate, cu fielduri anume, nu trebuie sa dam explicit valori acelor fielduri, sau sa le dam ca parametrii. Putem doar sa dam numele tabelei in acea bucata de cod dataAccess, care extrage datele din tabela si le pune in fielduri, sau invers. Este o metoda ingenioasa, care permite lucrul ceva mai flexibil decat inainte, si o noua metoda interesanta de a gandi.

1. Implementarea problemei:

Voi intra in detalii vorbind despre fiecare clasa in parte:

In pachetul connection, avem:

Connection factory:

-Este o clasa cu rol evident, sa se conecteze la baza de date. Nu se lucreaza nimic in aceasta clasa decat conectarea la driver, cu serverul bazei de date, dupa care, atunci cand executam o citire sau scriere in baza de date, se deschide (si la final inchide) conexiunea cu ea (avem si stateme si result set in aceeasi pozitie, facand parte din partea de executie si conexiune)

Din pachetul model:

Clasa Client, Product si Orders: (order este cuvant cheie in sql, deci am luat Orders peste tot)

- Aceste clase sunt simple, sunt doar modelul tabelelor, adica acea mapare. Ce este de remarcat este ce metode avem in ele: Avem 3 constructori, pentru ca avem nevoie de constructori fara id, cu id si fara nimic pentru diferitele operatii si metode de reflection, si avem toti setterii si getterii, pentru acces puternic la date (si din nou, pentru fielduri, si lucrul cu acestea in reflection).

In pachetul DAO:

Clasa AbstractDAO:

-Probabil cea mai importanta clasa, avem clasa in care ne ocupam cel mai tare de java reflection. Aici totul este generic, T, si va fi folosit pentru toate tabelele, de la client la product la order. Aici creeam queriurile de la 0, folosind fieldurile gasite dinamic din clase, iar daca vrem sa scriem in clasa, sa punem date noi, vom folosi invocarea de setteri pentru a seta valorile in statementuri. Ne folosim de aceste metode de lucru pentru a jongla cu scrierea si citirea datelor in baza de date, si astfel facem cat de general putem. Scopul ar fi sa putem lucra cu orice tabela, nu sa facem cate o operatie pe fiecare tabela, ci sa refolosim operatia de add de exemplu pentru toate care o folosesc (chiar daca au fielduri diferite, sau si numarul de fielduri diferit).

Clasele ClientDAO, OrdersDAO si ProductDAO:

-Acestea doar apeleaza explicit metode din clasa AbstractDAO, adica o extind, si deci acum doar mapam direct numele tabelei peste acele operatii. Din numele tabelei, ne vom da seama cu java reflection de fielduri, si astfel vom avea rezolvarea operatiilor anume. Toate aceste clase functioneaza la fel, depinde cat de multe operatii implementeaza doar.

Despre pachetul BLL:

Cele 5 clase validatoare (impreuna cu interfata pe care o extind toate)

-Aici doar verificam daca datele de intrare in baza de date sunt corecte. Daca sunt corecte, facem operatia, daca nu sunt corecte, vom da throw la o exceptie pe care o prindem in gui, si o afisam, pentru a se cunoaste eroarea facuta de utilizator. Aceste functii fac doar logica de verificare, nimic mai mult.

Clasele Client, Product, OrdersBLL:

-Avem aceleasi metode ca in clasele DAO, dar acum validam inainte datele ce vrem a fi scrise, si asa stim sigur ca introducem ce dorim, nu ce nu trebuie. Totodata, avem returnari de obiecte, int, depinde de metoda, ca sa ne verificam daca a fost bine inserat, sters, etc... sau sa folosim un obiect anume, care a fost creeat cu reflection (precum cazul cu find). Aceste clase vor fi des folosite ca reprezentari finale ale rezultatului. (daca a fost facuta operatia sau nu)

Pachetul Presentation:

Clasa View:

* Aceasta clasa se ocupa de view-ul original, cel in care alegem una din cele 3 clase. Avem butoane la acele windowuri

Clasa View client, product si order:

* Acestea sunt windowurile unde executam operatiile, cu butoane si text fielduri. Avem si un buton de revenire la pagina principala, dar in mare folosim getterrii pentru valorile obtinute din GUI, si le folosim ca date de intrare mai departe.

Clasa View table:

* O clasa mai speciala, implementam acel Jtable, folosit acest views de toate 3 butoanele de findAll (pentru toate 3 clasele). Am refolosit-o deoarece nu poate fi deschisa de 2 clase deodata. Aici avem cea mai asemanatoare forma de afisare pentru tabele.



Clasa Controller:

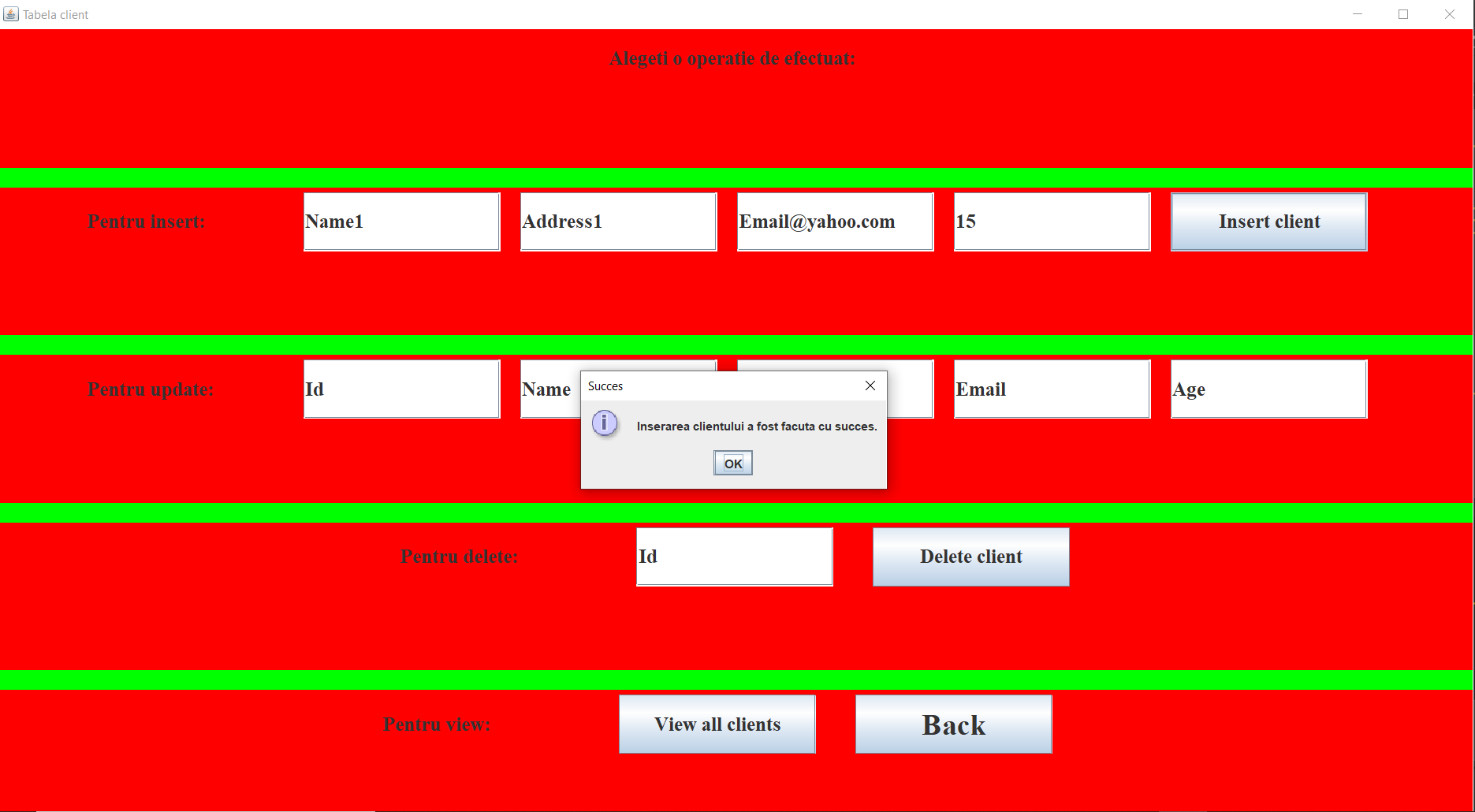
* Avem o clasa ce face un management al view-urilor si butoanelor asociate lor. Avem butoanele de la fiecare view, si in plus avem logica din clasele BLL care ne da rezultat bun sau ra. Daca operatiile executate de BLL sunt bune, avem mesaje de succes si operatii facute pe baza de date, daca nu, avem mesaje de eroare, si asa cunoaste utilizatorul cu ce a gresit. Important sa folosim controllerul in modul in care a fost menit sa fie folosit.

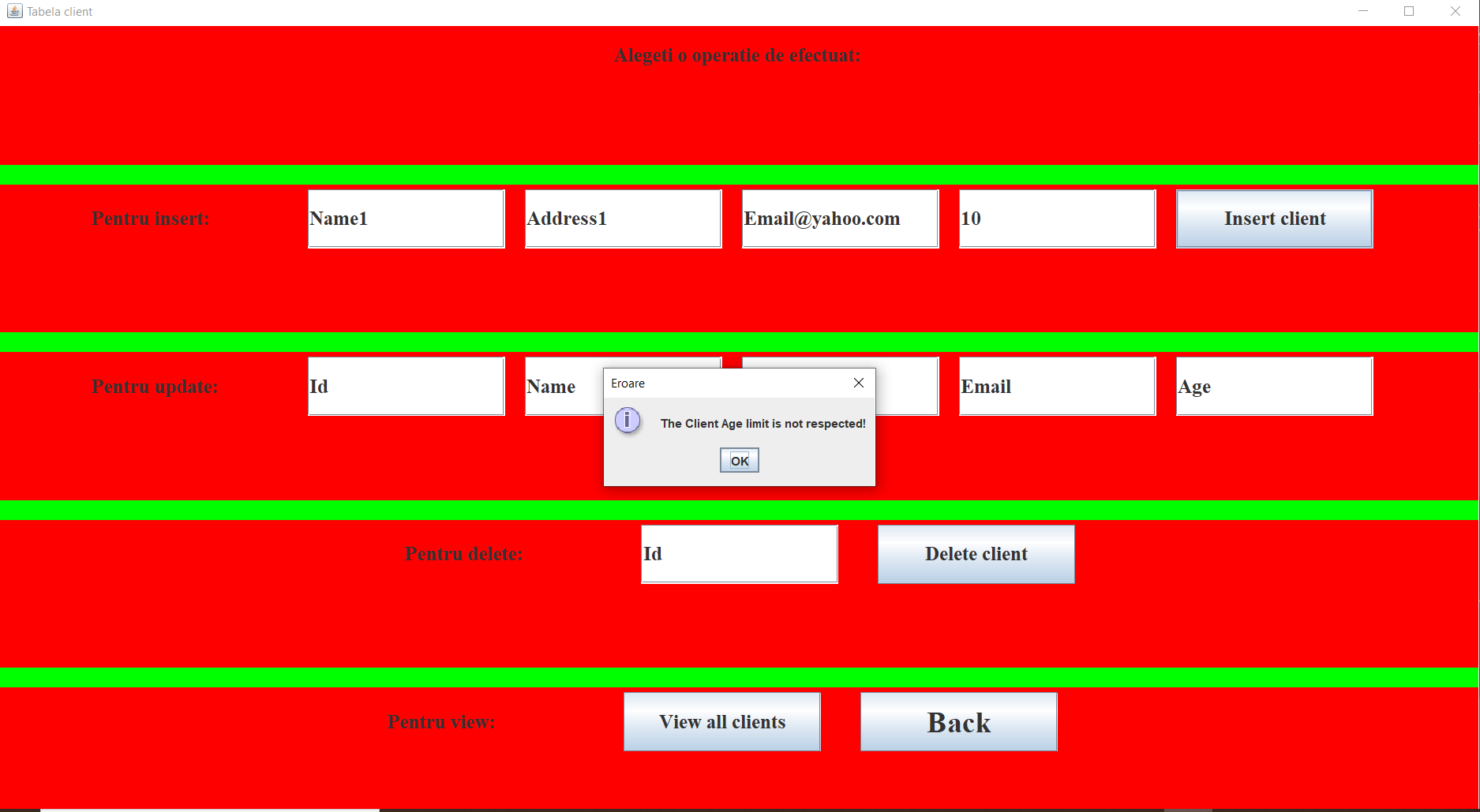
Clasa start (main):

* Doar o clasa in care initializam elementele de MVC, adica modelul, view si controllerul, si dupa logica se intampla intre clase. Nu avem alta logica suplimentara in main. (doar pentru inceperea programului).

1. Rezultatele obtinute:

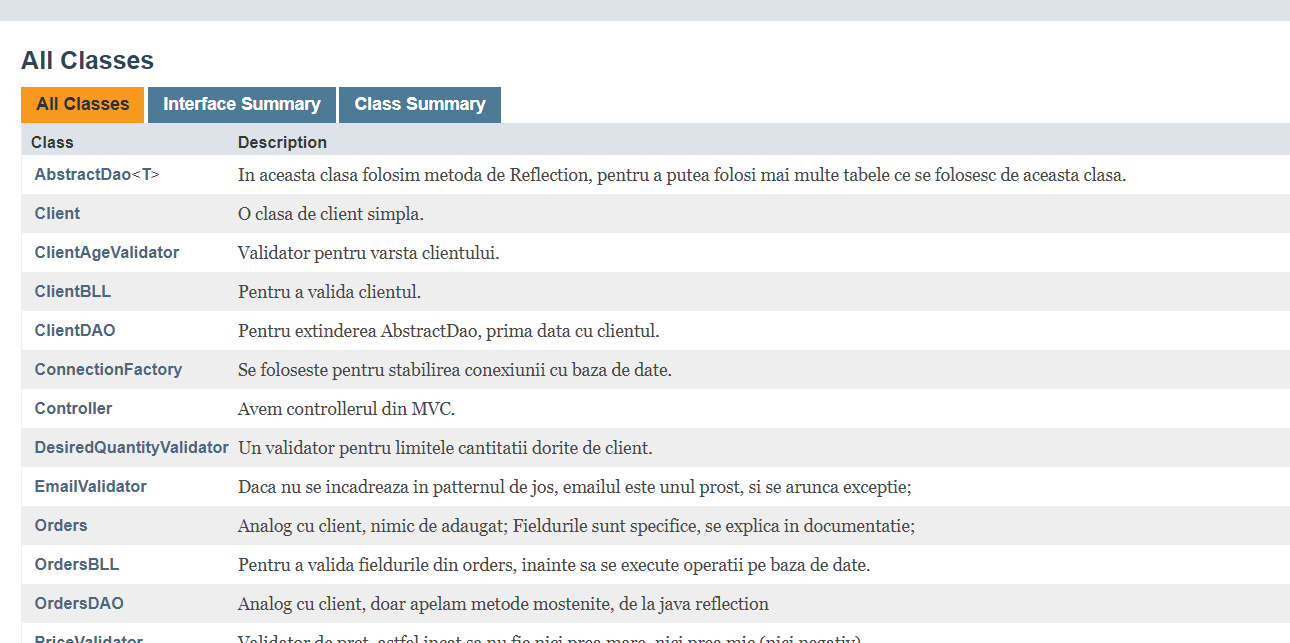
Deoarece am lucrat cu GUI, avem multe rezultate ce pot fi obtinute. In mare, avem mesaje de succes si mesaje de eroare.



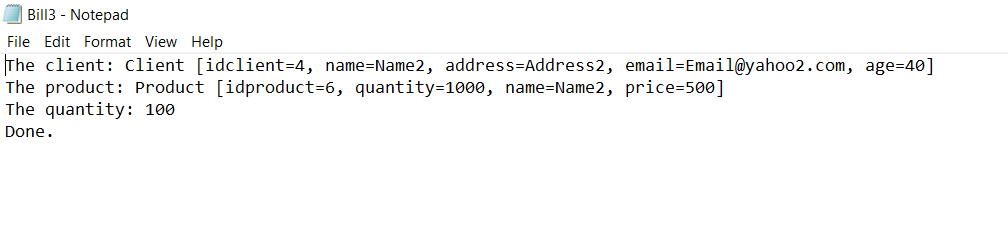


Mesajele de succes vin atunci cand avem datele scrise in mod corect si logic, si operatiile se pot executa fara probleme. Exceptiile si erorile vin atunci cand nu se respecta una din validari, sau nu se dau tipuri corecte de a tastatura, etc... sunt multe, si am tratat o mare parte din ele in program.

Cat despre alte rezultate avute, dupa ce am generat fisierele JavaDoc, am putut sa observ mai detaliat informatiile despre clasele mele, si programul meu in general. Voi studia in viitor mai mult aceste fisiere, pentru ca pare ca pot oferi informatii de mare ajutor.



In final, dupa fiecare inserare in orders, un bill este generat in folderul proiectului. Acel bill contine informatii legate de clientul ce a cumparat produsul anume, si cat de mare cantitate a cumparat. Acest bill este generat doar daca produsul nu este under stock sau nu sunt alte erori (precum: id client si id product trebuie sa existe, altfel nu se poate efectua acest order, tratat cazurile aici).



1. Concluzii:

De la aceasta tema, am invatat importanta folosirii generalitatii cand lucram cu orice fel de obiecte, fisiere, etc.. .Daca avem ceva general, atunci vom putea sa il folosim in mod flexibil, si vom putea lega foarte usor lucrurile intre ele. Nu vom avea nevoie de mult cod actual, pentru ca deja codul este facut in spate, pentru a se face maparea corecta. Aceasta este marea idee pe care mi-am facut-o, in rest mi se pare ca a fost o tema destul de lunga, insa a iesit mai bine decat credeam, si au fost multe elemente interesante legate de aceasta tema.

1. Bibliografie:

* Materialul ajutator oferit pentru aceasta tema.
* <https://www.baeldung.com/javadoc>
* https://www.baeldung.com/java-jdbc