**DOCUMENTATIE TEMA 3**

**ORDER MANAGEMENT**

**Nume prenume: Pașca Maria**

**Grupa 30229**

**Profesor Laborator Assist Antal Marcel**

Contents

[1. Cerinte Functionale 3](#_Toc476131445)

[2. Obiective 3](#_Toc476131446)

[2.1. Obiectiv Principal: 3](#_Toc476131447)

[2.2. Obective Secundare: 3](#_Toc476131448)

[3. Analiza Problemei 3](#_Toc476131449)

[4. Proiectare 3](#_Toc476131450)

[4.1. Structuri de date 3](#_Toc476131451)

[4.2. Diagrama de clase 3](#_Toc476131452)

[4.3. Algoritmi 3](#_Toc476131453)

[5. Implementare 4](#_Toc476131454)

[6. Testare 4](#_Toc476131455)

[7. Concluzii si Dezvoltari Ulterioare 4](#_Toc476131456)

[8. Bibliografie 4](#_Toc476131457)

# Cerinte Functionale

Cerința a fost dezvoltarea unui program pentru a procesa comenzile primite de un depozit, de asemenea folosind o bază de date pentru stocarea informațiilor despre clienți, produse și comenzi. Utilizator are la dispoziție o interfață grafică prin care poate comunica cu programul și cu baza de date.

Pentru insert product și insert client, informațiile trebuie adăugate de la tastatură, însă pentru insert order, utilizatorul se poate folosi de două combo box-uri pentru a selecta id-ul clientului și id-ul produsul care vor fi adăugate comenzii, cantitatea fiind adăugată de la tastatură. În cazul în care cantitatea cerută este mai mare decât cea de pe stoc, va apărea o eroare care avertizează clientul că nu se poate îndeplini comanda.

# Obiective

## Obiectiv Principal:

Obiectivul principal este proiectarea și implementarea unei aplicații pentru a procesa comenzile primite de un depozit, care relaționează cu o bază de date în care sunt stocați clienții, produsele și comenzile și permite adăugarea, ștergerea, editarea și vizualizarea acestora printr-o interfață grafică.

## Obective Secundare:

|  |  |  |
| --- | --- | --- |
| **Obiectiv Secundar** | **Descriere** | **Capitol** |
| Dezvoltarea de use case-uri si scenarii | Descrierea modului de funcționare a programului în diferite situații | 3 |
| Alegerea structurilor de date | Prezentarea pe scurt a structurilor de date folosite | 4 |
| Diagrama de clase | Diagrama UML a proiectului | 4 |
| Implementarea solutiei | Descrierea programului pe larg, descrierea pachetelor, claselor, metodelor și relației dintre clase și pachete | 5 |
| Testare |  | 6 |

# Analiza Problemei

Use Case Name: Inserarea unui nou client

Actori:

* Utilizatorul
* Program
* Baza de date

Trigger:

* Utilizatorul a dat click pe butonul de Add Client

Precondition:

* Utilizatorul a introdus în câmpuri datele cerute corect

Normal Flow:

1. Utilizatorul introduce corect datele în textFieldurile aferente.
2. Utilizatorul apasă butonul de Add Client.
3. Listenerul butonului de Add Client din ClientController apelează metoda addClient din ClientController.
4. În addClient, se verifică dacă textfieldurile sunt goale sau nu, după care se apelează insertClient din ClientBLL.
5. În metoda insertClient din ClientBLL, se validează informația primită și se apelează metoda insert din DAO.
6. În DAO se stabilește conexiunea cu baza de date, după care se generează query-ul de inserare care se trimite bazei de date.
7. Baza de date execută query-ul și adaugă clientul în tabelul Client, ca pe urmă să trimită înapoi programului id-ul generat.
8. În DAO, id-ul generat din baza de date este setat clientului și se returnează.
9. Din clientBLL, id-ul este returnat mai departe.
10. În ClientController, id-ul returnat este afișat într-un JOptionPane cu mesajul ”The client with id: *id* has been added”.
11. Utilizatorul va vedea mesajul pe ecran.
12. Utilizatorul va închide programul.

Alternate Flow:

4A. Utilizatorul lasă câmpuri goale.

1. Va apărea pe ecran un popup cu mesajul ”ClientView: field *câmpul\_respectiv* cannot be empty.”
2. Utilizatorul apasă ok și introduce și informațiile lipsă.

5A. Adresa de email nu respectă formatul cerut.

1. Va apărea în consolă eroarea ”Not a valid email!”
2. Programul își va înceta execuția.

5B. Telefonul nu respectă formatul cerut.

1. Va apărea în consolă eroarea ”Not a valid telephone number!”
2. Progrmaul își va înceta execuția.

6A. Serverul este închis.

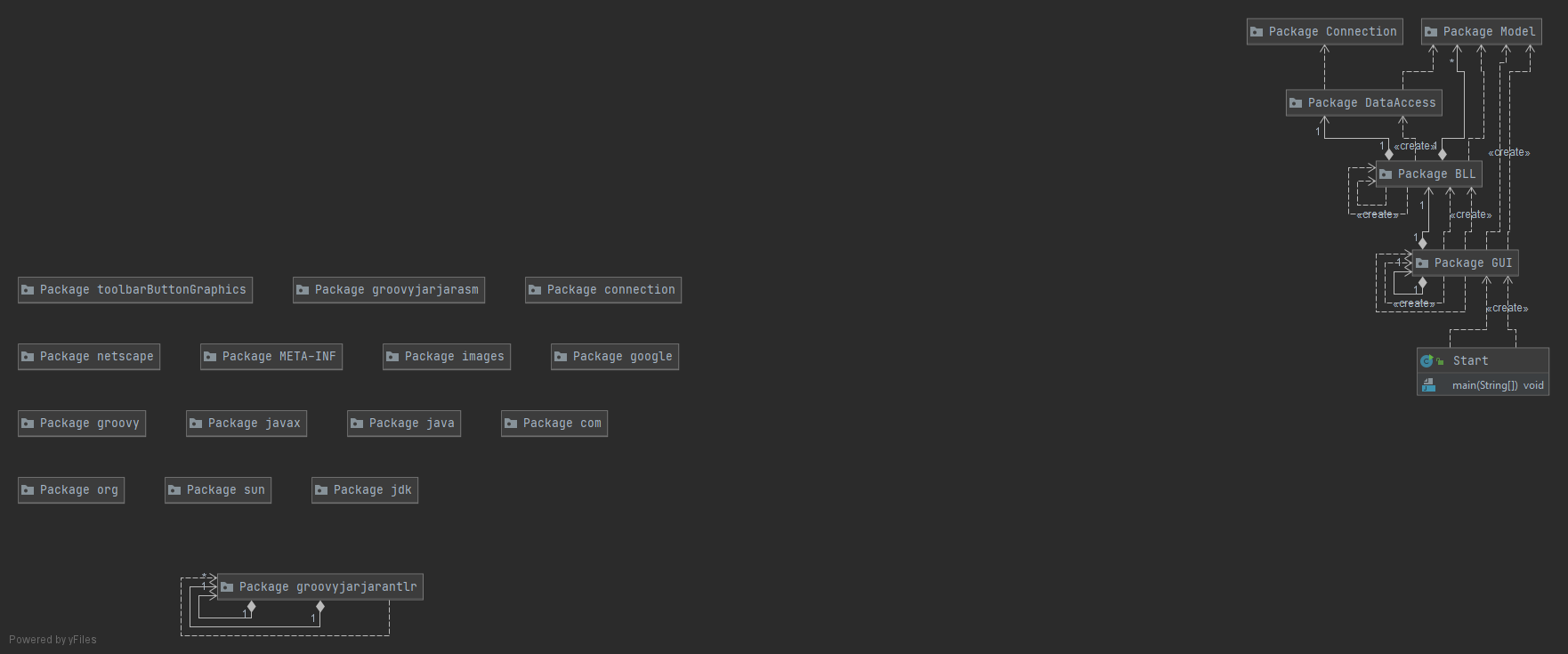
1. Va apărea în consolă o eroare în acest sens.
2. Programul își va înceta execuția.

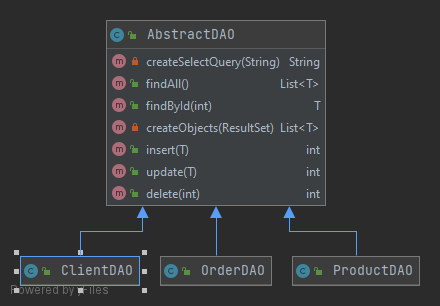
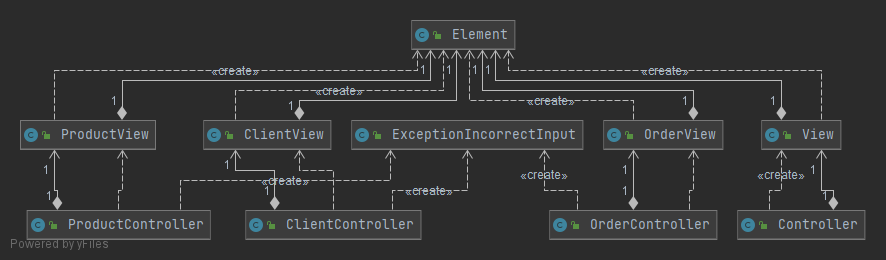
# Proiectare

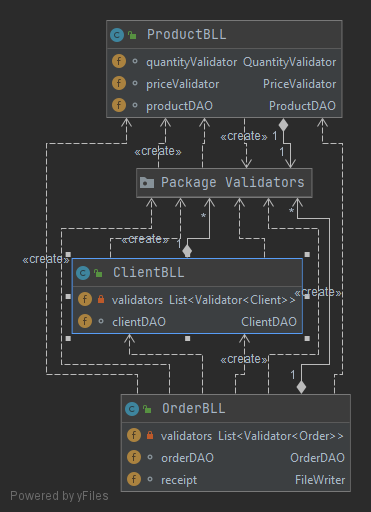
## Structuri de date

Structura de date cea mai răspândită în proiect este Singly-LinkedList-ul. Singly-LinkedLists sunt folosiți în acest proiect atât pentru a stoca și returna clienții, produsele și comenzile atunci când se apelează findAll, cât și pentru stocarea validatorilor pentru clienți, produse și comenzi.

## Diagrama de clase







# Implementare

Pentru acest proiect am folosit layered architecture, în cadrul căreia am pachetele BLL, care conține logica de lucru a aplicației, Validators (în cadrul pachetului BLL), care conține validatorii prin care programul se asigură că informația introdusă este corectă și respectă formatul cerut, Connection, care conține clasa prin care se face legătura proiectului la baza de date, DataAccess, prin care se realizează accesul la baza de date, GUI, care conține interfața proiectului, și Model, care conține modelele de date ale aplicației.

Pachetul Model conține clasele Client, Product și Order, numite identic după tabelele din baza de date warehouseDB. Metodele din clasele Client, Product și Order sunt cele de bază, gettere și settere și toString, deoarece acestea sunt folosite în mare parte doar pentru stocarea datelor extrase din sau care urmează să fie introduse în baza de date.

Pachetul DataAccess conține clasele ClientDAO, ProductDAO, OrderDAO și AbstractDAO, prin care se trimit comenzi la baza de date legată la aplicație. AbstractDAO este clasa care conține toate metodele care fac posibilă transmiterea comenzilor la baza de date. Aceasta conține metoda createObjects, care creează o listă de obiecte (clienți, produse, comenzi, în funcție de ce i se cere) din resultStatement-ul trimis ca parametru. Metoda findById construiește un query pe care îl trimite bazei de date pentru a returna obiectul (client, product, order) cu id-ul dat. Metoda findAll, pe de altă parte, îi trimite bazei de date un query pentru a returna toate obiectele din tabelul specificat, transmițând mai departe o listă de obiecte. Metoda insert construiește un query pentru a insera în tabelul specificat din baza de date obiectul trimis ca parametru. În schimb, metoda returnează un număr întreg care reprezintă id-ul obiectului proaspăt introdus în baza de date. Metoda update construiește un query pentru a updata informațiile unui obiect identificat după id. Acesta returnează de asemenea un întreg care reprezintă id-ul obiectului updatat. Metoda delete construiește un query pentru a șterge din tabelul specificat din baza de date un obiect al cărui id este trimis ca parametru. Această metodă returnează tot un întreg care reprezintă id-ul obiectului proaspăt șters din baza de date. Clasele ClientDAO, ProductDAO și OrderDAO extind AbstractDAO și, deși sunt clase goale, acestea se folosesc pentru specificarea obiectului cu care urmează să lucreze AbstractDAO.

Pachetul Connection conține clasa ConnectionFactory, care realizează conexiunea cu baza de date specificată. Aceasta conține metoda createConnection, care este privată deoarece este apelată doar în cadrul acestei clase. Aceasta stabilește conexiunea la baza de date prin apelul metodei DriverManager.getConnection(DBURL, USER, PASS). Deoarece se folosește o singură instanță a acestei clase, se realizează în cadrul clasei un singleton care este trimis la restul aplicației. Metoda getConnection() returnează conexiunea creată anterior, anume acel singleton. Clasa mai are 3 metode de close, una pentru închiderea conexiunii la baza de date, una pentru închiderea statement-ului și una pentru închiderea resultSet-ului.

Pachetul BLL conține clasele ClientBLL, ProductBLL, OrderBLL și pachetul Validators. Clasa ClientBLL conține metoda findClientById, care primește ca parametru id-ul clientului căutat și returnează o instanță a clasei Client care reprezintă chiar clientul căutat. După ce apelează metoda findById din DAO, verifică dacă obiectul există sau nu, caz în care afișează un mesaj de eroare. Metoda insertClient trece obiectul Client trimis ca parametru prin lista de validatori și, în cazul în care sunt îndeplinite toate condițiile, se apelează metoda insert din DAO, care-l primește ca parametru pe obiectul respectiv și returnează id-ul nou generat al obiectului proaspăt adăugat în baza de date. Metoda update este similară celei anterioare, din nou clientul este trimis ca parametrul validatorilor din listă și, în cazul în care se îndeplinesc toate condițiile, este trimis mai departe ca parametru metode update din DAO, pe urmă returnând id-ul acestuia. Metoda deleteClient primește ca parametru id-ul obiectului care urmează să fie șters și apelează metoda delete din DAO. Metoda showAllClients apelează metoda findAll din DAO și returnează toate obiectele din tabelul Client sub formă de Singly linked list. Clasele ProductBLL și OrderBLL conțin aceleași metode, însă cu alte nume și puțin schimbate să se potrivească pe obiectul de lucru al fiecăreia. Singura diferență totuși se găsește la clasa OrderBLL care conține în plus metoda printReceipt, care generează un text file care conține detaliile comenzii și a clientului care a comandat și a produsului care a fost comandat.

Pachetul Validators conține interfața Validator și clasele ClientIDValidator, ProductIDValidator, EmailValidator, PriceValidator, QuantityValidator și TelephoneValidator, care extind interfața Validator. ClientIDValidator și ProductIDValidator verifică în baza de date dacă există obiectele cu id-ul dat ca parametru. Aceste două clase se folosesc în cadrul metodei insertOrder și updateOrder din clasa OrderBLL. PriceValidator verifică ca prețul introdus să fie mai mare ca 0 și QuantityValidator verifică ca și cantitatea să fie mai mare ca 0 și, în plus, în cazul unei comenzi, se asigură ca și cantitatea cerută să nu fie mai mare decât cantitatea care e pe stoc. Ambele clase se folosesc în cadrul metodei insertProduct și updateProduct din clasa ProductBLL. EmailValidator se asigură că email-ul introdus este corect din punct de vedere al formatului unui email, iar TelephoneValidator de asemenea verifică dacă telefonul introdus respectă formatul unui număr de telefon. Ambele clase folosesc regex patterns și sunt folosite în cadrul metodelor insertClient și updateClient din clasa ClientBLL.

Pachetul GUI conține View-urile și Controllere adiacente acestora. Astfel, pachetul conține View și Controller, ClientView și ClientController, ProductView și ProductController, OrderView și OrderController, dar și ExceptionIncorrectInput și Element.

ExceptionIncorrectInput este o clasă care extinde Exception și este folosită pentru a afișa un mesaj de eroare în cazul în care un câmp este lăsat gol deși nu trebuie. Clasa Element conține elementele necesare pentru aspectul interfețelor, precum culori și fonturi.

Clasa View conține 4 JPanel-uri principale, unul cu meniul principal, unul cu meniul pentru clienți, unul cu meniul pentru produse și unul cu meniul pentru comenzi. Meniul principal are 3 JButtons, unul care deschide meniul pentru Clienți, unul care deschide meniul pentru Produse și unul care deschide meniul pentru comenzi, meniul pentru clienți conține la rândul lui 5 JButtons, unul pentru inserare, unul pentru editare, unul pentru ștergere, unul pentru afișarea tuturor clienților și unul de back, prin care se revine la meniul principal. Celelalte două meniuri, pentru produse și pentru comenzi, conțin aceleași butoane precum meniul pentru clienți. Apăsarea oricăruia dintre aceste butoane, mai puțin al celui de back, va deschide o nouă fereastră prin apelarea clientController-ului care realizează o nouă instanță de clientView.

ClientView conține 4 JPanels, unul pentru adăugare, unul pentru updatare, unul pentru ștergere și unul pentru vizualizare a tuturor obiectelor. Panelurile pentru adăugare și updatare sunt aproape identice; ele conțin JTextBox-uri pentru introducerea noii informații și un JButton prin care se apelează în ClientController metoda de insert, respectiv de update din ClientBLL. Panelul de ștergere conține un JTextBox în care se introduce de la tastatură id-ul clientului care urmează să fie șters și un buton prin apăsarea căruia se apelează metoda delete din ClientBLL. Panelul de vizualizare a tuturor obiectelor conține un JTable care este populat cu clienții din baza de date prin apelarea metodei showAllClients din ClientBLL. Celelalte două view-uri sunt în principal identice, singura diferență fiind la OrderView, în panelul insert, unde de această dată, pentru câmpurile clientId și productId, în loc de JTextFields sunt JComboBox-uri care conțin doar id-urile existente în baza de date, pentru a preveni crearea unei comenzi pe un client fictiv sau pentru un produs fictiv.

View-urile au fost realizate folosind Java Swing și toate clasele, mai puțin cele ce țin de UI, au până în 300 de linii, iar metodele până în 30 de linii.

# Testare

Testarea proiectului se face prin UI, unde utilizatorul are oportunitatea să testeze toate funcționalitățile programului și să constate dacă este vreo eroare sau vreun rezultat greșit returnat.

# Concluzii si Dezvoltari Ulterioare

Programul reușește să realizeze toate funcționalitățile dorite, anume să insereze, să updateze, să șteargă sau să afișeze clienții, produsele sau comenzile, în funcție de dorințele utilizatorului, de asemenea reușind și să genereze chitanța pentru fiecare nouă comandă plasată și să o salveze local sub forma de text file.

Dezvoltările pe care le văd să fie făcute în viitor sunt, în primul rând, împrospătarea aspectului UI-ului și schimbarea în cadrul OrderView-ului, în panelul de updatare, a textfieldurilor de unde introduce utilizatorul clientId-ul și productId-ul în ComboBox-uri, precum în panelul de insert, pentru a ne asigura că produsul respectiv clientul există. De asemenea, în cadrul panelurilor de delete din ClientView, ProductView și OrderView, văd modificate TextField-urile pentru Id-ul obiectului care vine șters cu ComboBox-uri, din nou pentru a avea certitudinea că obiectele respective există și nu se încearcă ștergerea din baza de date a unui obiect care nu există. De asemenea, o altă implementare pe care o mai văd realizată este ștergerea după nume; utilizatorul să introducă un nume al unui obiect pe care vrea să-l șteargă din baza de date, însă prima oară să genereze o listă într-un combobox cu toate obiectele care poartă acel nume.

O altă implementare este adăugarea unei funcționalități de căutare după Id, respectiv după nume; utilizatorul să introducă id-ul respectiv numele în textbox, iar programul să returneze obiectul cu id-ul respectiv sau toate obiectele care au numele introdus de utilizator.

În afară de aceste aspecte tehnice, prin acest proiect am reușit să stăpânesc mai bine limbajul Java și crearea programelor prin implementarea layered architecture, prin folosirea reflection technique pentru comunicarea cu baza de date și prin folosirea javadoc-ului.

# Bibliografie

<https://www.baeldung.com/java-jdbc>

<http://tutorials.jenkov.com/java-reflection/index.html>

<https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html>