**DOCUMENTAȚIE TEMA 3**

**ORDER MANAGEMENT**

**Nume prenume: Copoț Raluca**

**Grupa: 30227**

**Profesor laborator: Mitrea Dan**

Cuprins

1. **Obiectivul temei3**
   1. Obiectivul principal3
   2. Obiective secundare3
2. **Analiza problemei3**

2.1 Analiza principală 3

2.2 Cerințe funcționale 4

2.3 Cerințe non-funcționale 4

1. **Proiectare5**

3.1 Determinarea structurilor de date5

3.2 Împărțirea în clase și pachete5

3.3 Algoritmi6

1. **Implementare6**

4.1 Clase și metode6

4.2 GUI9

1. **Rezultate10**

5.1 Funcționare10

5.2 Testare10

1. **Concluzii10**
2. **Bibliografie11**
3. **Obiectivul temei**
   1. **Obiectivul principal**

Proiectarea și implementare unei aplicații pentru a procesa comenzile clienților pentru un depozit.

* 1. **Obiective secundare**

|  |  |  |
| --- | --- | --- |
| **Obiectiv secundar** | **Descriere** | **Capitol** |
| Analizarea problemei și identificarea cerințelor | Realizarea diagramei use-case și determinarea scenariilor posibile | 2 |
| Determinarea structurilor de date | Pentru ca aplicația să funcționeze corect trebuie să definim tipuri pentru lucrul cu baza de date | 3 |
| Împărțirea în clase si pachete | Determinarea unui scop bine definit pentru fiecare clasă și distribuirea acestora în diferite pachete, în funcție de rol | 3 |
| Determinarea și dezvoltarea algoritmilor | Pentru a putea efectua operații cu baza de date trebuie să găsim metode de conectare si lucru cu aceasta | 3 |
| Implementarea aplicației | Scrierea propriu-zisă a codului, implementarea efectivă a algoritmilor determinați anterior | 4 |
| Testarea | Verificarea corectitudinii aplicației pe cele trei tabele, propuse în îndrumător | 5 |

1. **Analiza problemei**

**2.1. Analiza principală**

Exemplu de use-case pentru adăugarea unui client:

**Use case**: adăugarea unui client

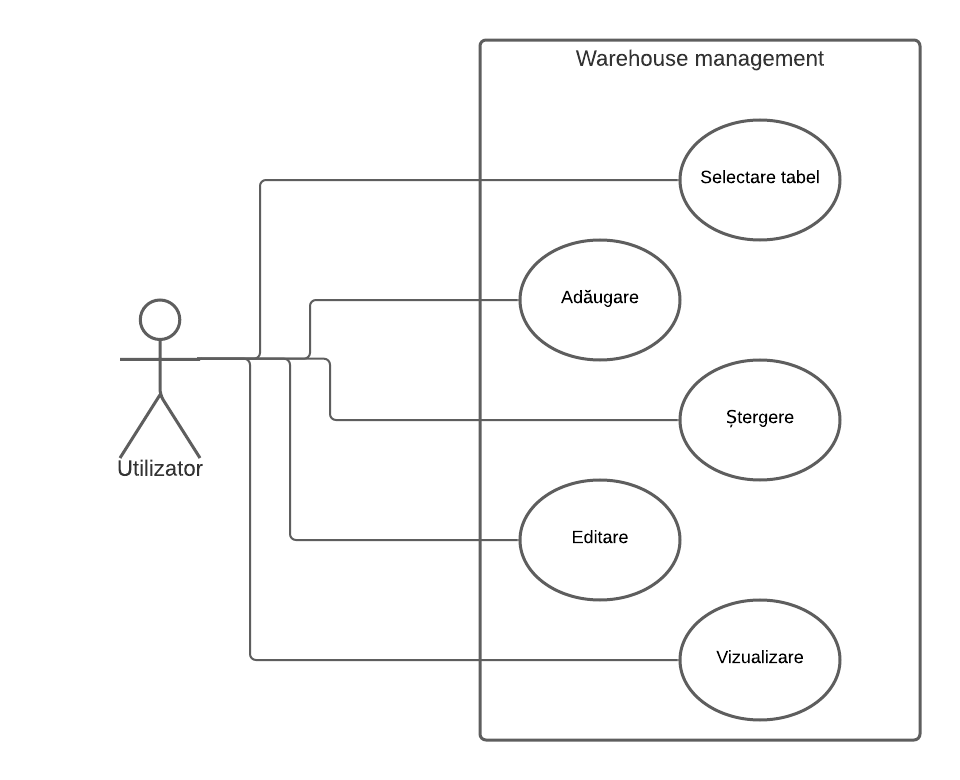
**Actorul primar**: utilizatorul

**Scenariul principal în caz de succes**:

1. Utilizatorul introduce datele de intrare: id-ul clientului, numele și adresa acestuia.
2. Utilizatorul apasă pe butonul pentru inserare, „Insert”.
3. Clientul este creat și adăugat în baza de date.

**Scenariul alternativ**: Introducerea unor date de intrare invalide

* Utilizatorul introduce un interval negativ de timp sau nu introduce un număr
* Aplicația afișează un mesaj de eroare
* Nu este introdus clientul în baza de date
* Scenariul se întoarce la pasul 1

**Diagrama use case:**

**2.2. Cerințe funcționale**

Proiectați și implementați un aplicație de simulare cu o interfață grafică, care ca scop procesarea comenzilor pentru un depozit, folosind baze de date relaționale. Aplicația trebuie să poată realiza:

* Citirea datelor de la tastatură;
* Operații cu cele 3 tabele a bazei de date;
* Utilizarea arhitecturii „layered”;
* Afișarea rezultatelor în interfața grafică;
* Generarea unei chitanțe;

**2.3. Cerințe non-funcționale**

* Aplicația ar trebui să fie intuitivă și ușor de folosit de către utilizator;
* Designul ar trebui să fie unul compact, dar care conțină toate elementele dorite, fără să devină prea încărcat;
* Aplicația ar trebui să atenționeze utilizatorul atunci când introduce date de intrare invalide și să nu modifice datele din baza de date în acest caz;
* Rezultatul afișat ar trebui sa fie lizibil, ușor de înțeles și interpretat;

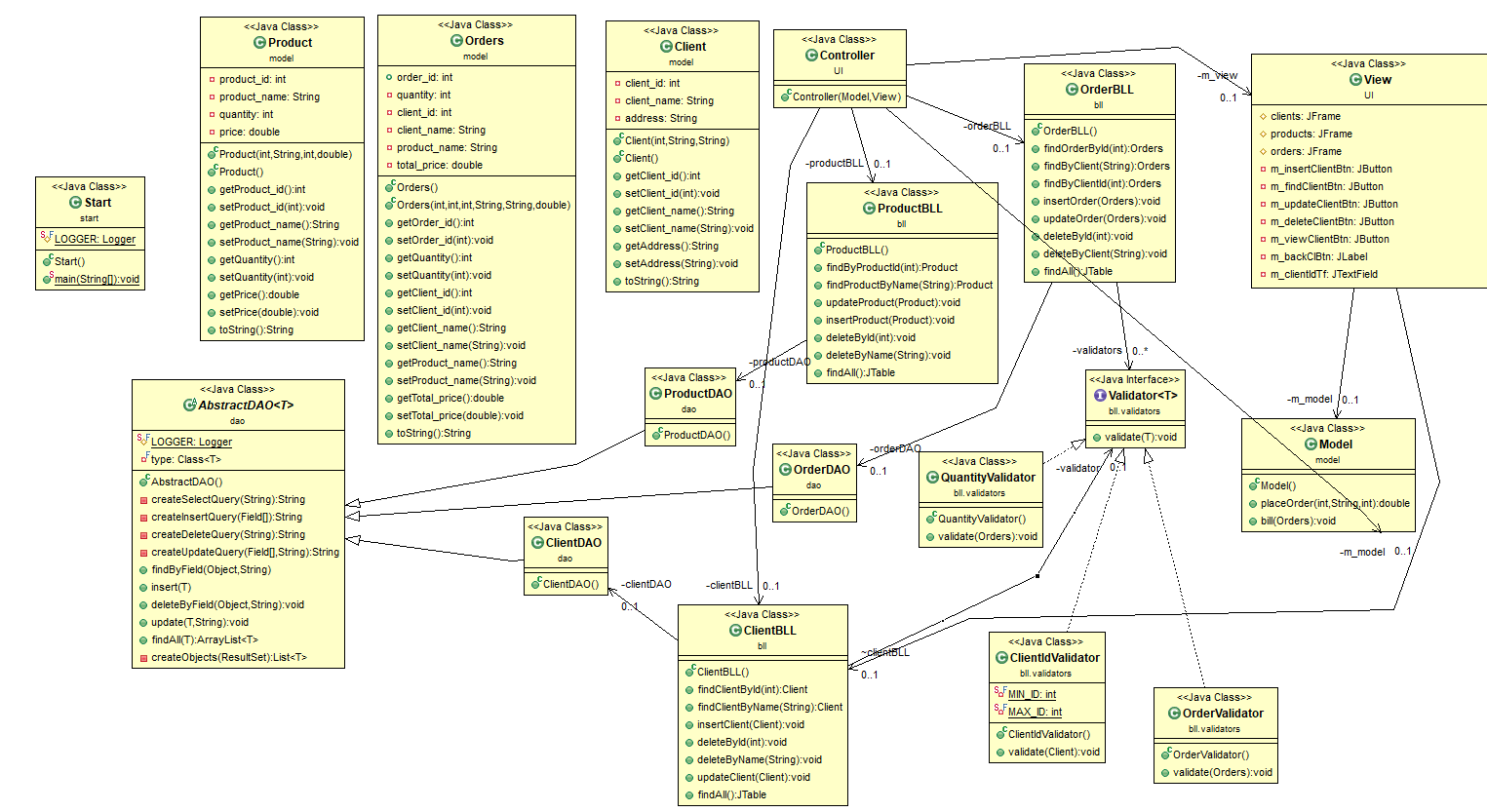
1. **Proiectare**
   1. **Determinarea structurilor de date**

Pentru acest proiect, elementele principale sunt Client, Product și Orders, motiv pentru care am implementat aceste clase. De asemenea, precum este specificat în cerință, trebuie să utilizez tehnica reflexiei, motiv pentru care utilizez clasa „AbstractDAO”, care conține metodele generale și câte o clasa DAO pentru cele trei tabele.

De asemenea, utilizez o intefață „Validator” pentru a îmi crea validatorii pentru datele din tabele. Am decis sî ma limitez la trei validatori, unul fiind pentru clienți și ceilalți pentru comenzi.

* 1. **Împărțirea în clase și pachete**

**Diagrama UML:**



Pasul următor după stabilirea tipurilor de date pe care le voi folosi, este împărțirea în clase, respectiv în pachete, acest lucru fiind ușor datorită suportului .pdf. Astfel am împărțit clasele în pachete după cum urmează:

* start: clasa main
* model: conține logica aplicației, Client, Product, Orders, Model
* dao: partea de lucru cu baza de date, AbstractDAO, ClientDAO, ProductDAO, OrderDAO
* bll: bussiness logic layer, ClientBLL, OrderBLL, ProductBLL, și validatorii
* connection: realizează conexiunea la baza de date, ConnectionFactory
* UI: interfața grafică, Controller și View
  1. **Algoritmi**

**FindByField:** folosindu-mă de un query pentru select, în care folosesc condiția „where field =?”, pot interoga, într-un mod general, oricare tabel din baza de date.

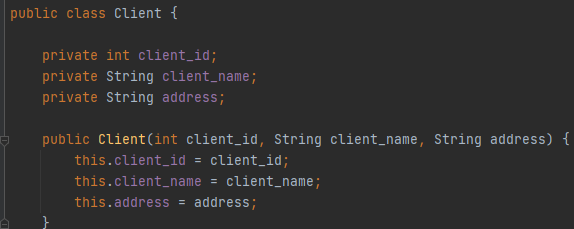
**Insert:** utilizând query-ul specific pentru inserare, pot introduce un nou element în tabel.

**DeleteByField:** precum procedez în cazul inserării, în funcție de un anumit câmp al tabelului, spre exemplu id-ul, pot șterge unul sau mai multe elemente din tabel care au id-ul respectiv.

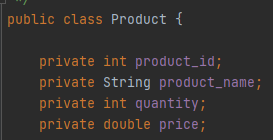
**Update:** și în acest caz, precum la insert, obțin query-ul corespunzător pentru operația de update. Însă, pentru ca acest lucru să fie posibil, voi face update în funcție de id, acesta fiind cheia primară în toate cele trei tabele. Așadar, acesta nu va putea fi modificat, în schimb oricare dintre celelalte coloane putând fi editate.

**FindAll:** efectuează „SELECT \* FROM tabel”, rezultatul returnat fiind convertit într-o listă, care va conține toate elementele din tabel. Voi utiliza această metodă pentru afișare.

**PlaceOrder:** folosesc acest algoritm pentru a obține prețul total al comenzii, acesta fiind produsul dintre prețul unitar și numărul de obiecte cumpărate. De asemenea, actualizez numărul de produse, pe care îl calculez ca suma dintre numărul inițial de produse și diferența dintre cantitatea inițială din comandă, care este 0 în cazul uneia noi și cantitatea efectivă din comandă. În cazul în care acest numar este negativ, atunci nu mai există stoc suficient, ceea ce înseamnă ca nu poate fi plasată comanda.

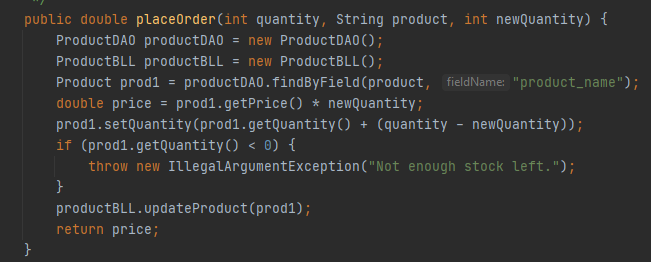
1. **Implementare**
   1. **Clase și metode**

**Client:** conține structura unui client. Atributele sale sunt ID-ul, numele și adresa. Metodele implementate în această clasă sunt cele de bază, respectiv metode de get, set, constructorul cu parametrii și unul fără niciun parametru. De asemenea, am suprascris metoda toString.



**Product:**  conține structura unui produs, conținând atributele: ID-ul, numele și cantitatea și prețul, acesta din urmă fiind de tip double. Metodele utilizate sunt aceleași precum cele din clasa Client. Suprascriind și în acest caz metoda toString.

**Orders: :**  conține structura unei comenzi. Atributele sale sunt: ID-ul, cantitatea, ID-ul și numele clientului care a plasat comanda, numele produsului cumpărat și prețul total, tot de tip double. Precum în celelate două clase din acest pachet, are aceleași metode.



**Model:** conform arhitecturii de tip MVC, aceasta conține logica din spatele aplicației. Metodele implementate sunt „placeOrder” și „bill”. Precum am menționat în secțiunea precedentă, „placeOrder” calculează prețul comenzii și actualizează stocul de produse. Cealaltă metodă este cea care generează chitanța, în format pdf. Titlul reprezintă data exactă la care a fost plasată comanda, iar conținutul acesteia este editat folosind un „pdfWriter” si elemente de tipul „Paragraph” pentru a putea aranja textul în modul dorit.

**ConnectionFactory:**  este clasa cu ajutorul căreia este realizată conexiunea la baza de date. Aceasta implementează design pattern-ul singleton în acest sens, cu metode simple de creare, obținere și închidere a acestor conexiuni.

**AbstractDAO:** este o clasă abstractă, care implementează metodele CRUD pentru instanțe ale claselor Client, Produs, respectiv Orders. Această clasă conține metode care generează query-uri pentru SELECT, INSERT, DELETE și UPDATE. De asemenea, implementează metode generalizate pentru a efectua aceste operații, după cum este menționat în secțiunea anterioară. Iar metoda „createObjects” este cea care convertește elementele din tipul „ResultSet”, în „List”, pentru ca rezultatele interogărilor sa poată fi utilizate ulterior.

**ClientDAO:** moștenește clasa „AbstractDAO”, implementând implicit metodele din aceasta și particularizându-le pentru elemente de tipul „Client”.

**ProductDAO:** și aceastamoștenește clasa „AbstractDAO”, implementând implicit metodele din aceasta și particularizându-le pentru elemente de tipul „Product”.

**OrderDAO:** care, de asemene moștenește clasa „AbstractDAO, de această data metodele fiind particularizate pentru elemente de tipul „Orders”.

**ClientBLL:** implementează business logic, atât pentru clasa cât și pentru tabelul de clienți. Conține un validator, pentru id și un obiect de tipul „ClientDAO” cu ajutorul căruia pot particulariza operațiile pe baza de date. Astfel, am implementat metode precum „findClientById”, „findClientByName”, „insertClient”, „DeleteById” și „updateClient”. Metoda „findAll” generează un tabel, ale cărui date sunt obțiunte apelând metoda cu același nume din clientDAO și returnează un JTable, acesta urmând să fie afișat în interfață.

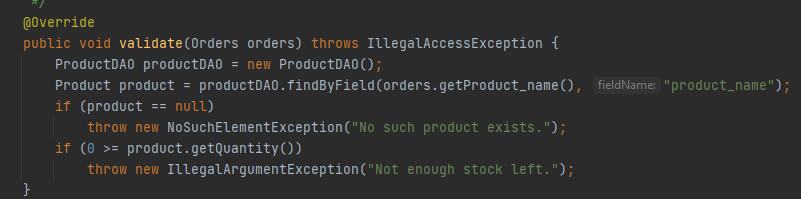
**OrderBLL:** precum „ClientBLL”, implementeză business logic, dar pentru comenzi. În această clasă folosesc doi validatori, unul pentru a verifica existența id-ului clientului care plasează comanda și unul pentru a verifica existența produsul și dacă acesta mai este în stoc. Apelez aceți validatori la inserarea unui noi comenzi. Precum în cazul clasei precedente, am particularizat metode pentru căutare, inserare, ștergere și editare, dar și metoda „findAll”, care și în acest caz returnează un JTable.

**ProductBLL:** și această clasă implementează business logic, dar de această dată pentru produse. Nu am considerat că ar fi necesara crearea unui validator pentru acestea, motiv pentru care nu folosesc unul. Așadar singurul atribut din această clasă este productDAO. La fel ca pentru celelalte două clase BLL, particularizez operațiile cu baza de date și metoda „findAll”, pentru a genera tabelul cu toate elementele, pe care îl voi afișa.

**Validator:** este o interfață, care va fi implementată de către validatori. Singura metoda este „validate”, aplicată pentru orice obiect de tipul „T”. În cazul în care obiectul nu este valid, metoda va arunca o excepție de tipul „IllegalAccessException”.

**ClientIdValidator:** clasă care implementează interfața „Validator”, pentru obiecte de tipul „Client”. În cazul în care id-ul clientului nu se află în intervalul definit de „MIN\_ID”, respectiv „MAX\_ID”, atunci excepția este aruncată.

**OrderValidator:** o altă clasă care implementează interfața. În acest caz, metoda „validate” verifică existența id-ului clientului care dorește să plaseze comanda. În cazul în care acesta nu există în tabelul de clienți, excepția este aruncată.



**QuantityValidator:** la rândul ei, această clasă implementează interfața, respectiv metoda de validare. În cazul acestui validator, este verificată existența produsului pe care utilizatorul dorește să îl cumpere, în baza de date. Dacă nu există, este aruncată excepția cu un mesaj corespunzător. De asemenea, este verificată și cantitatea rămasă în stoc. Există posibilitatea ca execuția programului să treacă de testele efectuate în clasa „Model”, deși mai există zero bucăți din acel produs. Astfel, verific dacă există o cantitate mai mare strict decât zero, iar dacă această condiție nu este îndeplinită, atunci este aruncată excepția și comanda nu va putea fi plasată.

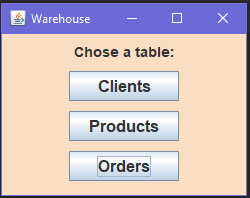
**Start:** reprezintă clasa de „main” a programului, cea în care are loc execuția propriu-zisă. Aceasta conține un Logger, pentru conexiunea la baza de date. Iar, pentru că folosesc arhitectura de tip MVC, conține si un obiecte de tipul „Model”, „Controller” și „View”.

**Controller:** controlează logica aplicației, coordonând View-ul și Modelul. Acesta primește un input prin intermediul vederii, procesează datele cu ajutorul modelului si returnează rezultatul vederii. Acesta poate fi astfel ușor de interpretat pentru utilizator.

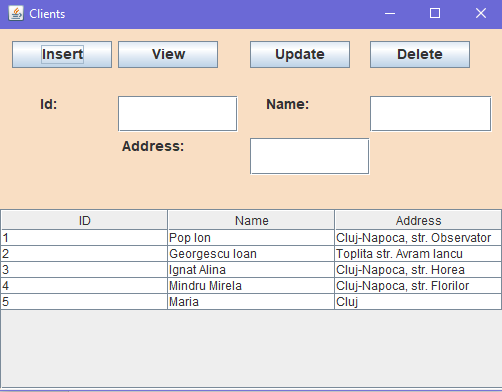
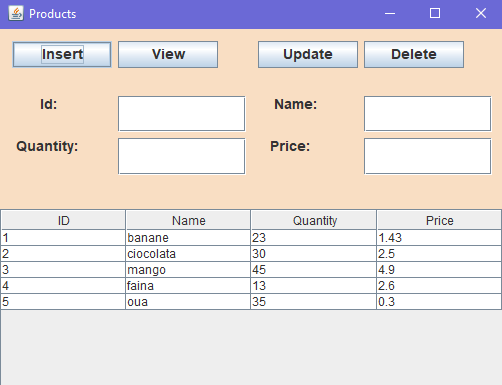
**View:** conține elementele care formează interfața grafică, precum butoanele sau text-field-urile, dar și câte un ActionListener pentru fiecare buton.

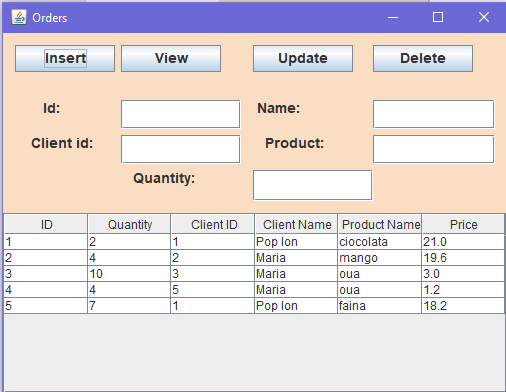
**4.2. GUI**

GUI (Graphical User Interface) din Java reprezintă mecanismul cu ajutorul căruia utilizatorul poate interacționa cu un program. Acesta conține componente grafice, cum ar fi butoane, etichete, ferestre etc., pe care utilizatorul le poate folosi pentru a interacționa cu ușurință cu aplicația. GUI are un rol important în construirea cu ușurință a unei interfețe pentru aplicațiile Java.

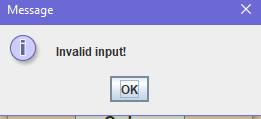
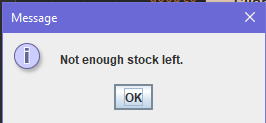


**View:** precum am menționat în secțiunea anterioară, în această clasă se află elementele care formează interfața grafică. Pentru această aplicație utilizez patru „JFrame-uri”. Panoul principal, care devine vizibil odată cu începerea execuției programului este cel din imaginea alăturată. Atunci când acesta este închis, se încheie și execuția programului.

Din acest moment, utilizatorul poate apăsa pe unul dintre cele trei butoane pentru a alege tabelul cu care își dorește să lucreze. Luându-le în ordine, panoul pentru clienți este cel alăturat. Cu ajutorul acestuia, utilizatorul poate atât să vizualizez conținutul tabelului, apăsând pe butonul de „View”, cât și să efectueze operații cu baza de date, în cazul în care introduce inputul corespunzător. Panourile pentru produse și comenzi sunt asemănătoare, singurul lucru care diferă fiind conținutul tabelelor și textfield-urile, după cum se poate observa:



De asemenea, mesajele de eroare care pot apărea sunt în cazul datelor de intrare invalide și dacă nu mai există suficiente produse pe stoc pentru a plasa comanda, după cum se poate observa din cele doua imagini:



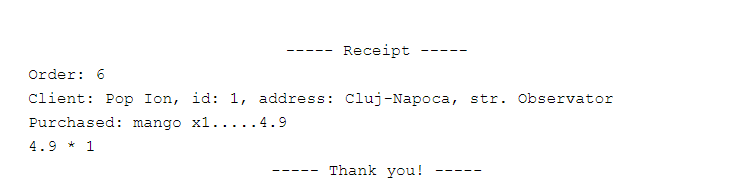
**Controller:** conține logica aflată la baza interfeței grafice, îmbinând view-ul cu modelul, motiv pentru care le conține ca atribute. În această clasă am implementat ActionListener pentru fiecare dintre butoanele vizibile în cele 4 JPanel-uri. Aceastea suprascriu metoda „ActionPerformed” în funcție de acțiunea pe care o execută. Spre exemplu, butoanele de pe primul JPanel, „Clients”, „Produscts” și „Orders” fac vizibile JPanelurile corespunzătoare, în timp ce restul efectuează operații care implică si baza de date.

1. **Rezultate**
   1. **Funcționare**

Utilizatorul trebuie să aleagă pentru început tabelul cu care își dorește să lucreze. După aceea, el poate fie să apese doar pe butonul de view pentru a vizualiza conținutul tabelului, caz în care nu trebuie introduse date. Dacă iși dorește sa editeze conținutul tabelului, trebuie sa introducă date de intrare valide, în caz contrat fiind afișate mesaje de eroare. Iar, fiind mult mai intuitiv astfel, odată cu terminarea lucrul cu un tabel, poate apăsa pe „x” și alege un alt tabel. Există și posibilitatea ca toate cele patru JPanel-uri sa fie active în același timp.

* 1. **Testare**

Pentru acest proiect nu a trebuit să efectuez teste cu JUnit, în schimb corectitudinea programului reiese din modul în care sunt modficate datele din baza de date. Un alt mod bun de verificare este chitanța, datele de pe aceasta și cele din baza de date fiind identice.



1. **Concluzii**

În cadrul acestui proiect am îmbinat pentru prima data Java cu bazele de date și am învațat sa folosesc tehnica reflexiei. De asemenea, am încercat pentru prima dată sa scriu într-un fișier de output în format .pdf. Sunt mulțumită de modul în care am aranjat textul, dar și de corectitudinea generală a programului.

Însă, un lucru pe care l-aș modifica este actualizarea automată a tabelului din JPanel, eliminând necesitatea butonului de view. Însă acesta a fost menționat în cerință și nici nu am mai avut timp pentru a implementa acest detaliu.

1. **Bibliografie**

* Definiție GUI (13.04.2021 21:30): [https://www.guru99.com/java-swing-gui.html definiție GUI 17.03](https://www.guru99.com/java-swing-gui.html%20definiție%20GUI%2017.03)
* Layout (13.04.18:00): <https://docs.oracle.com/javase/tutorial/uiswing/layout/group.html>
* PDF (10.05 19:00):
  + <https://www.vogella.com/tutorials/JavaPDF/article.html#format>
  + <https://www.baeldung.com/java-pdf-creation>
* Javadoc (12.05 13:00): <https://www.jetbrains.com/help/idea/working-with-code-documentation.html#generate-javadoc>