DOCUMENTATIE

TEMA 3

ORDER MANAGEMENT

Vamvu Denisa-Elena

Grupa 30229

CUPRINS

1. Obiectivul temei

-Se va prezenta obiectivul principal al temei printr-o fraza si un tabel sau o lista cu obiectivele secundare. Obiectivele secundare reprezinta pasii care trebuie urmati

pentru indeplinirea obiectivului principal. Fiecare obiectiv secundar va fi descris si

se va indica in care capitol al documentatiei va fi detaliat.

1. Analiza problemei, modelare, scenarii, cazuri de utilizare

-Modelare, scenarii, cazuri de utilizare . Se va prezenta cadrul de cerinte functionale formalizat si cazurile de utilizare ca si diagrame si descrieri de use-case.

Descrierile use-case-urilor se vor face sub forma unui flow-chart ori sub forma unei liste continand pasii executiei fiecarui use-case.

1. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)

-Se va prezenta proiectarea OOP a aplicatiei, diagramele UML de clase si de pachete, structurile de date folosite, interfetele definite si algoritmii folositi (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator).

4. Implementare

-Se va descrie fiecare clasa cu campurile si cu metodele importante. Se va descrie implemantarea interfetei utilizator.

5. Rezultate

-Se vor prezenta scenariile pentru testare cu Junit sau alt framework de testare.

6. Concluzii

-Se vor prezenta concluziile, ce s-a invatat din tema, dezvoltari ulterioare.

7. Bibliografie

-Se vor mentiona resursele bibliografice care au fost folosite pentru dezvoltarea temei.

1. Obiectivul temei

Obiectivul acestei teme este implementarea unei aplicatii OrderManagement care proceseaza comenzile clientilor pentru un depozit. Se folosesc baze de date relationale pentru a stoca produsele, cleintii si comenzile. De asemenea aplicatia trebuie sa fie structurata pe pachete folosind o arhitectura stratificata. Si ar trebui sa urmeze minimal urmatoarele clase:

* Clasele model – modelele de date ale aplicatiei ;
* Clasele pentru logica de afaceri – implementeaza logica aplicatiei ;
* Clasele de prezentare – implementeaza input-ul / output-ul user-ului ;
* Clasele pentru acces de date – implementeaza accesul la baza de date ;

Obiective secundare*:*

* Dezvoltarea de use-case-uri
* Alegerea corecta a structurilor de date
* Impartirea pe clase
* Dezvoltarea algoritmilor
* Implementarea solutiilor
* Testarea programului

1. Analiza problemei, modelare, scenarii, cazuri de utilizare

Aplicatia trebuie sa permita procesarea comenzilor dintr-un fisier text dat ca argument in linia de comanda, sa efectueze operatiile cerute, sa salveze modificarile efectuate in baza de date si sa genereze rapoarte in format pdf. Alte clase si pachete pot fi adaugate pentru a implementa functionalitatea completa a aplicatiei.

De asemenea , trebuie implementat si un parser pentru citirea comenzilor din fisier in layer-ul de prezentare ( in loc de interfata grafica standard ) si un generator de fisiere pdf care sa genereze rapoartele.

Operatiile pe care aplicatia le poate procesa:

* Adaugarea unui client in baza de date ;
* Stergerea unui client din baza de date ;
* Adaugarea unui produs in baza de date ;
* Stergerea unui produs din baza de date ;
* Crearea unei comenzi pentru client ;
* Generearea de rapoarte pentru clienti,produse si comenzi;

**Use case**

Programul va fi apelat din linia de comanda cu ajutorul unui fisier .jar ce se afla in folderul proiectului. Se executa comanda “java -jar” urmata de numele fisierului .jar si numele fisierului de comenzi. Ca si rezultat se vor realiza fisierele pdf conforme comenzilor.

Se presupune ca elementele din fisierul de intrare, cat si fisierul, sunt valide si reprezinta cerintele temei.

1. Proiectare

* Decizii de proiectare:

Se va folosi tehnica “ reflection “ atat pentru accesul si operatiile cu baza de date cat si pentru scrierea in fisiere. **Java Reflection** este procesul de analizare si modificare a tuturor capabilitatilor unei clase la runtime.

* Structuri de date folosite

Am folosit o structura de tip List < T > , generica, pentru a retine rezultatele operatiilor de select in clasa AbstractDAO indiferent ce tip ar avea acestea.

Diagrama Use-Case:

A close up of text on a white background

Description automatically generated

Diagrama UML de pachete:

A picture containing clock, bird

Description automatically generated

Diagrama UML de clase:

A close up of a map

Description automatically generated

1. Implementare

In continuare voi prezenta clasele si pachetele pe care le-am folosit in acest proiect, dar si metodele cele mai importante din fiecare clasa.

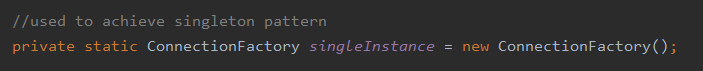
* **ConnectionFactory**

Clasa ConnectionFactory face conexiunea cu baza de date urmand pattern-ul Singleton. Singleton este un design software care restrictioneaza instantierea unei clase la un singur obiect si este folositor arunci cand exact un singur obiect este necesar pentru a coordona actiuni prin sistem.

**ConnectionFactory** face parte din pachetul **DataAccess** pentru ca ne ofera conexiunea cu baza de date.

Clasa contine driverul mysql catre baza de date, userul, parola, locatia bazei.

Cel mai important atribut al clasei este singleInstance.



Acesta, alaturi de constructor ofera o cale pentru clasa de a-si accesa propriul obiect fara a fi nevoie de o instantiere.

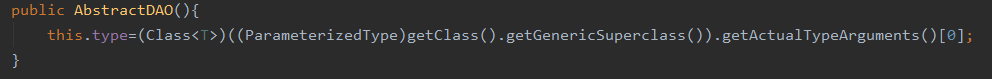
* **Model**

Pachetul **Model** contine clasele care reprezinta tabelele din baza de date. Toate atributele sunt mapate pe coloanele tabelelor . Acestea contin constructori, setters si getters.

* Clasa **Client** contine
* id – numarul de ordine al clientului incrementat automat cand acesta este introdus in baza de date, de asemenea cheie primara pentru tabelul Clients din baza de date assignment3db
* client\_name – numele clientului
* address – adresa la care locuieste clientul
* Clasa **Order** contine
* client\_name – numele clientului care plaseasa comanda
* product\_name – numele produsului dorit
* quantity – cantitatea dorita, un intreg
* Clasa **Product** contine
* id – numarul de ordine al produsului incrementat automat cand acesta este introdus in baza de date, de asemenea cheie primara pentru tabelul Products din baza de date
* product\_name – numele produsului
* stock\_quantity – cantitatea disponibila in stock-ul depozitului
* price – pretul per unitate al produsului, un numar real
* Clasa **OrderItem** contine
* client\_id – id-ul clientului care a plasat comanda, cheie straina
* product\_id – id-ul produsului care este comandat, cheie straina
* **BusinessLogic**

Pachetul **BusinessLogic** contine logica aplicatiei, deci clasele care fac toate metodele si operatiile asupra bazei de date.

* Clasa **AbstractDAO** foloseste tehnica reflection siare ca si atribute un logger, un obiect de tip reportGenerator pentru a permite apelarea functiilor de scriere in pdf si un type, generic. In constructor, se ia clasa tipului generic T cu care se creeaza obiectul de tip AbstractDAO.



Pentru fiecare operatie care este necesara asupra bazei de date in urma primirii unei comenzi din fisierul de intrare txt, clasa contine o metoda privata pentru a returna sub forma de string query-ul sql corespunzator. Se foloseste un obiect de tip StringBuilder si se apeleaza metoda append() pentru a concatena string-ului datele necesare. Metoda **createSelectQuery**() returneaza spre exemplu un obiect de tip string de forma: “select \* from”+ “where”. Numele tabelului este gasit prin intermediul parametrului type si a apelarii metodei getSimpleName(), iar conditia pentru select este data de cei doi parametrii ai functiei, field-ul care ne intereseaza si valoarea acestuia care este cautata. Similar functioneaza si metodele **createUpdateQuery**(), **createDeleteQuery**() si **createSelectAllQuery**() care returneaza un string reprezentand un query de update al unui field, de delete a unei inregistrari dupa un anume criteriu sau de selectare a tuturor elementelor dintr-un tabel necesara scrierii de reporturi. O diferenta ce apare la unele dintre aceste metode este parcurgerea field-urilor fiecarui obiect generic si setarea acestora ca si accesibile pentru a putea folosi valorile si numele in scopul crearii statementurilor.

Tot pentru fiecare operatie se defineste o metoda publica ce executa query-ul de care are nevoie. Toate acestea realizeaza conexiunea la baza de date, isi pregatesc statementul si executa query-ul intr-un try-catch. Pentru cazul in care nu este posibil se arunca o exeptie de tipul SQLException.

Metoda **findByName**() primeste ca si parametrii field-ul care se cauta si valoarea acestuia, isi creeaza query-ul cu ajutorul metodei private, tine intr-un ResultSet (generat cu ajutorul metodei **executeQuery**()) ceea ce se returneaza si, cu ajutorul metodei **createObjects**() returneaza o Lista generica de obiecte. Similar cu aceasta metoda functioneaza si metoda **selectAll**() cu exceptia ca aceasta nu mai are o conditie si returneaza tot tabelul corespunzator tipului obiectului. Metodele **insert**() ,**update**() si **delete**() sunt similare, doar ca nu returneaza nimic. Acestea apeleaza metoda **executeUpdate**() si nu folosesc un resultSet pentru ca nu este nevoie.

Metoda privata **createObjects**() primeste ca si parametru un obiect de tip ResultSet, ca si rezultat al unui sql query si il parcurge, iar pentru fiecare “line” al acestuia, cu un for each se trece prin field-uri, se creeaza o instanta de tip T, iar cu ajutorul metodei invoke() se apeleaza metoda Method pe obiectul instanta cu parametrul valoare si astfel se creeaza obiectele generice. Obiectele se adauga in lista iar aceasta se returneaza.

Clasele ClientDAO, OrderDAO si ProductDAO extend fiecare clasa AbstractDAO cu parametrul corespunzator numelor lor: Cleint, Order si Product, pentru a se putea apela functiile din clasa parinte.

* Pachetul **PresentationLayer** contine clasele pentru user input/output interface
* Clasa **Parser** contine un String ce va fi numele fisierului de unde se vor extrage datele de intrare, un obiect de tip ReportGenerator pentru scrierea in pdf si 3 variabile de tip int care vor reprezenta numerele de ordine ale reporturilor din fiecare tip.

Metoda **readFromFile**() deshide fisierul primit ca si parametru in linia de comanda cu care obiectul de tip Parser este creat, defineste un scanner pentru citire, si cat timp exista linii scrise in fisier, se vor lua pe rand si se vor trimite celor doua functii de parsare: **checkEasyCase**() si **checkOtherCases**().

Metoda **checkEasyCase**() verifica daca stringul introdus este pentru generarea unui report. Se iau pe rand cele 3 cazuri: “Report client”, “Report order” si “Report product” iar in caz ca se gaseste unul dintre ele, functia creeaza un obiect de tip ClientDAO, OrderDAO sau ProductDAO, apeleaza functia **selectAll** din clasa AbstractDAO (care genereaza si pdf-ul de report cu numarul corespuzator) transmisa prin mostenire si celorlalte 3, si returneaza true. Daca nu este unul dintre cazurile cu report, se returneaza false.

Metoda **checkOtherCases**() primeste o linie a fisierului si foloseste regex pentru doua impartiri: dupa caracterul “ : “ si dupa caracterul “ , ”. Apoi verifica pe rand cazurile: inserare client, inserare produs, stergere client, stergere produs, si “Order”. Pentru fiecare dintre acestea este definite o functie privata care apeleaza functia necesara din AbstractDAO.

Metoda **insertClient**() primeste ca si parametru doua stringuri reprezentand numele si adresa unui client, creeaza un client nou si apeleaza metoda de inserare din AbstractDAO ce primeste ca parametru un Object si il insereaza in tabelul corespunzator.

Metoda **deleteClient**() primeste ca si parametru numele clientului care se doreste a fi sters, creeaza un obiect de tip ClientDAO() si unul de tip OrderDAO(), retine intr-un obiect de tip Client rezultatul dat de functia findByName() ce cauta in tabel clientul cu numele dat si intr-un obiect de tip Order rezultatul dat de aceeasi functie, cautand comenzile clientului respectiv. Astfel, inainte sa se apeleze functia de delete pentru client, se sterg prima datac cu un while toate comenzile sale.

Metoda **insertProduct**() primeste ca si parametru numele produsului, pretul si cantitatea. Se procedeaza similar ca pentru inserarea unul client, doar ca se folosesc obiecte de tip Product si ProductDAO pentru apelarea functiilor generice, iar daca produsul se afla deja in baza de date, se face update la field-ul stock, adunandu-se valoarea citita la cea existenta si apeland metoda update() din AbstractDAO.

Metoda **deleteProduct**() primeste ca si parametru numele produsului care sa fie sters din baza de date. Procedura este similara cu cea pentru stergerea unui client (cum ar trebui sa si fie, pentru ca am folosit reflection). Se foloseste metoda findByName() si o structura repetitive pentru a sterge mai intai toate comenzile din baza de date in care exista produsul si apoi pe acesta .

Metoda **takeOrder**() presupune faptul ca s-a gasit comanda “Order” in fisier si creeaza un obiect de acest tip cu ajutorul parametriilor: numele clientului, numele produsului si cantitatea dorita. Se creeaza si un produs, extragandu-se din baza de date cu metoda findByName() pentru a fi folosit in urmatorul scop: pentru a lua pretul si a calcula totalul care trebuie platit de catre client, pentru a verifica daca este stoc suficient pentru procesarea comenzii iar in caz negativ sa se genereze un pdf care spune cantitatea curenta din produsul respectiv, si pentru a ajuta la actualizarea stocului produsului, scazandu-se cantitatea din comanda din cantitatea de pe stoc, folosind de asemenea metoda update(). Tot in aceasta functie se genereaza in format pdf o factura apeland generateBill() a clasei ReportGenerator.

* Clasa **ReportGenerator** este o clasa generica ce creeaza pdf-urile corespunzatoare reporturilor, facturilor si mesajelor de eroare. Se foloseste reflection.

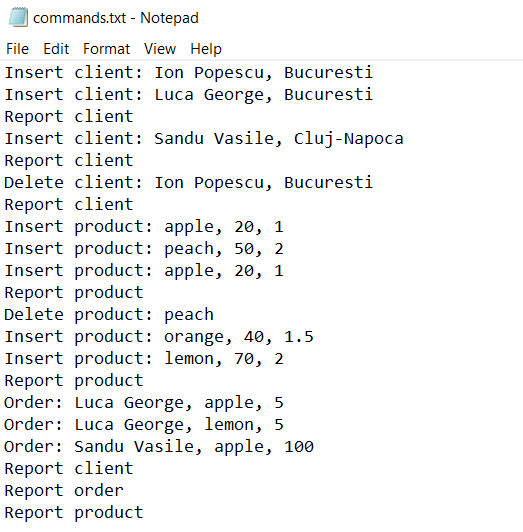
Metoda privata **addTableHeader**() adauga in pdf-ul primit ca si parametru numele field-urilor obiectului de tip T ca si capul tabelului ce urmeaza a si creat.

Metoda **createReport**() creeaza un fisier de tip pdf, parcurge lista de obiecte primita ca si parametru si insereaza in tabel valorile fiecarui field prin aceeasi metoda intalnita in AbstractDAO.

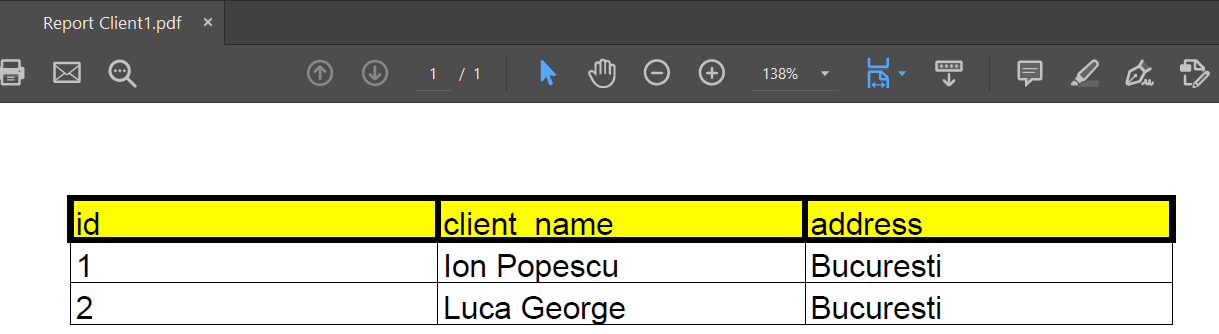
Metoda **generateBill**() creeaza un fisier pdf cu numele pe care il primeste ca parametru si scrie in el un string, care poate fi cel de stoc insuficient sau factura (ce este generat in Parser, in metoda makeOrder)

1. Rezultate

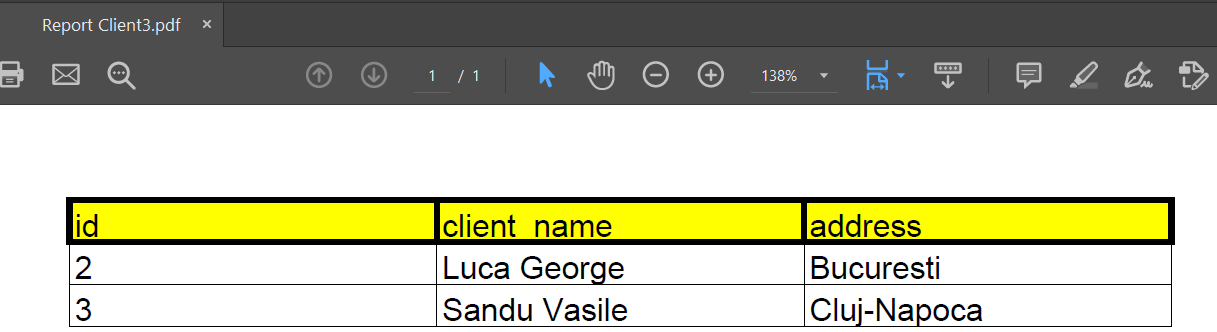
Se foloseste o baza de date initial goala. In momentul in care se citesc pe rand comenzi din fisier, aceasta sufera modificari datorita metodelor care sunt apelate in clasa Parser. In continuare, se prezinta comenzile ce pot fi date aplicatiei si rezultatele acestora:



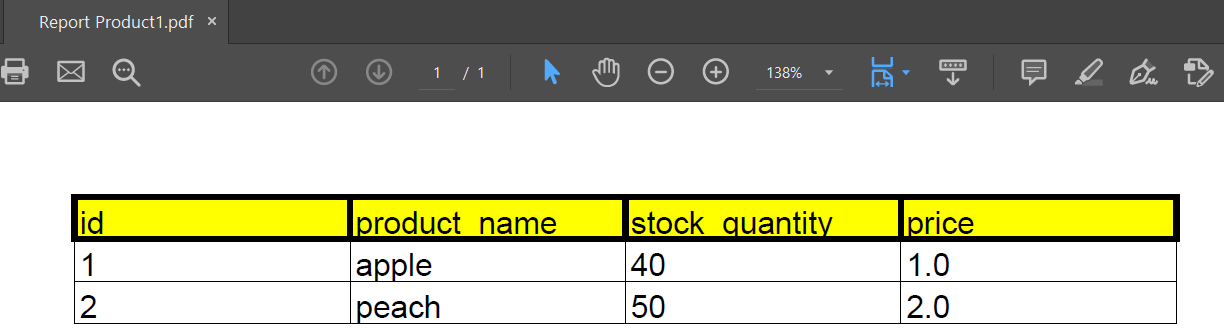
* **Report client** genereaza un pdf cu tot continutul tabelului Clients
* **Report order** genereaza un pdf cu tot continutul tabelului Orders
* **Report product** genereaza un pdf cu tot continutul tabelului Products
* **Insert client** contine numele si adresa clientului care se doreste a fi inserat in baza de date. Dupa introducerea primelor doua comenzi, se poate verifica starea tabelului de client prin comanda Report client**;**



* **Delete client** contine numele clientului care se doreste a fi sters din baza de date a depozitului. In cazul in care clientul are comenzi existente, se sterg si acestea. De asemenea, pentru verificare se poate genera din nou un fisier pdf, care va fi numerotat corespunzator. Dupa inserarea clientului Sandu Vasile si stergerea clientului Ion Popescu, pdf-ul generat (si tabelul din baza de date) vor contine urmatoarele inregistrari:

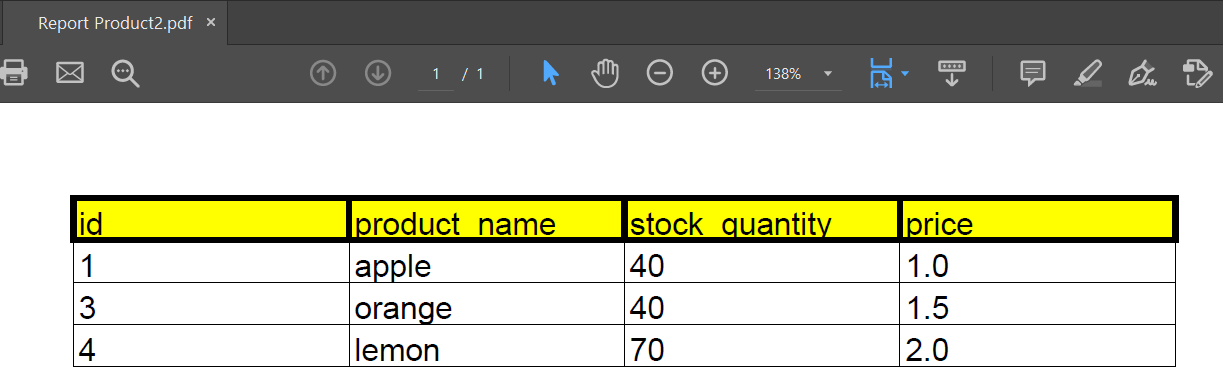


* **Insert product** – se va insera in tabela de produse un produs nou caracterizat prin nume, cantitatea din stoc si pret. In cazul in care se va incerca inserarea unui produs care deja exista, cantitatea acestuia va fi acutalizata prin adunare. Se presupune ca un produs care are acelasi nume cu unul deja existent, are si aceslasi pret. Dupa inserarea a 3 produse, “apple”, “peach” si din nou “apple”, apeland comanda “Report product”, starea tabelului va fi afisata intr-un pdf nou ca fiind:

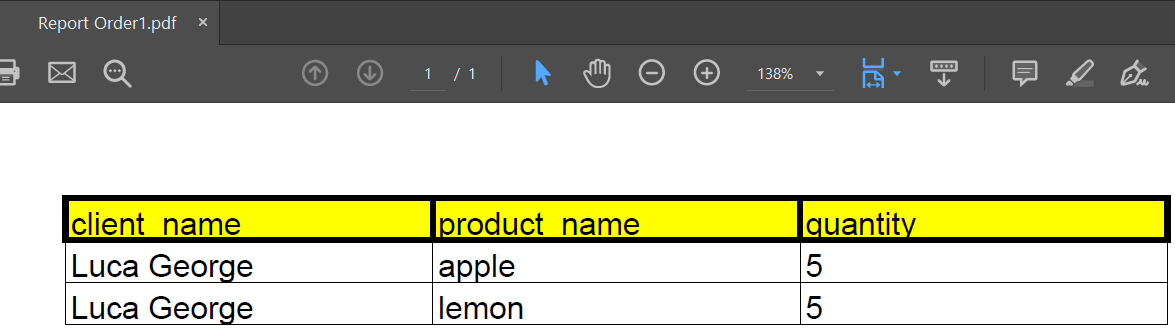
****

Se observa ca produsul “apple” are ca si cantitate suma celor doua cantitati introduse la momente diferite de timp.

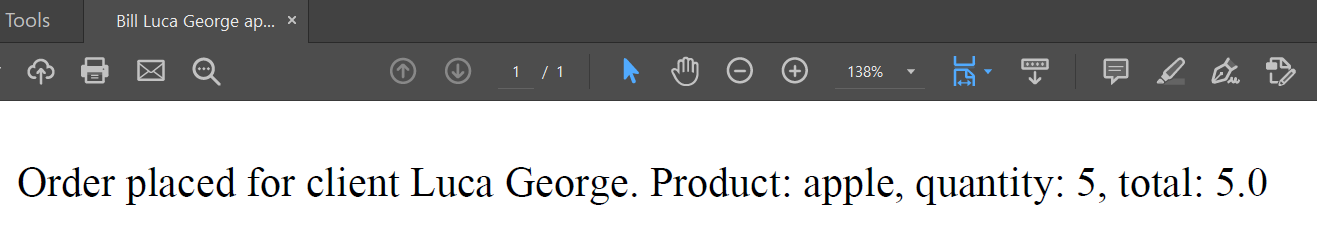
* **Delete product** – sterge din baza de date produsul cu numele primit. De asemenea, daca exista comenzi cu produsul respectiv, se sterg si acestea. Al doilea apel de “Report product” dupa inserarea produselor “orange” si “lemon” si stergerea produsului “peach”, va genera urmatorul pdf:



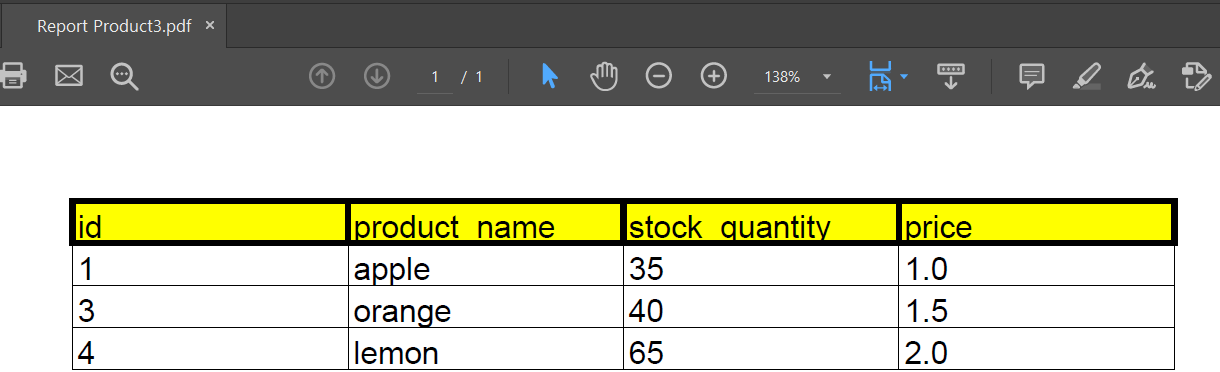
* **Order –** plaseaza comanda pentru un client, dupa numele acestuia, de o anumita cantitate dintr-un produs. Tabelul Orders va arata ca in pdf-ul generat dupa comanda “Report order”.

****

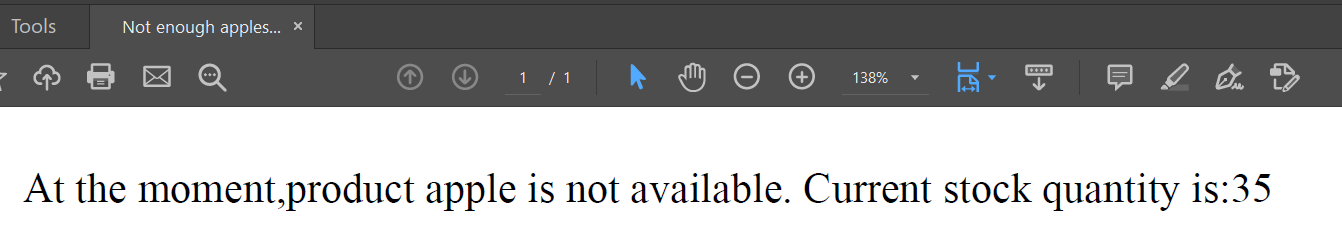
O data cu plasarea unei comenzi se genereaza si o factura in format pdf.



Cand o comanda este procesata, stocul de produse este actualizat:



In cazul in care stocul nu este suficient pentru o comanda, se va genera un mesaj de eroare tot sub forma unui pdf, cum este in cazul comenzii “Order: Sandu Vasile, apple, 100”. Mesajul contine si stocul actual de produse de acel tip.



1. Concluzii

O posibila imbunatatire poate fi adusa la capitolul eficientei. Ca si dezvoltare ulterioara as dori sa introduc o factura in care s-ar pune toate produsele comandate de un client, nu unul singur, cu un total corespunzator.

1. Bibliografie

https://www.javatpoint.com/example-to-connect-to-the-mysql-database

https://www.mysql.com

https://howtodoinjava.com/library/read-generate-pdf-java-itext/

* 1. https://en.wikipedia.org/wiki/Parsing