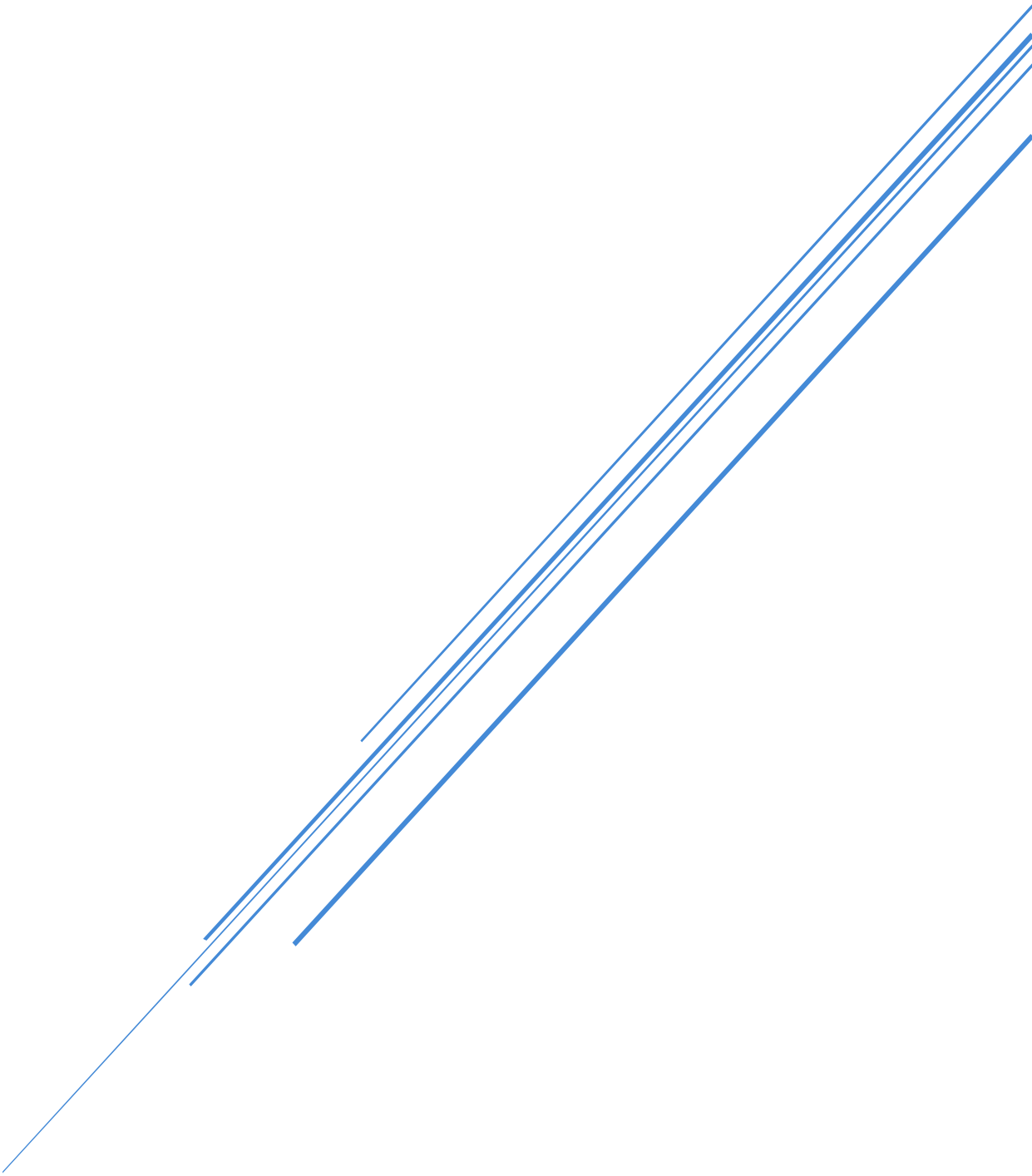
ORDER MANAGEMENT



Fărcaș Alexandru
Tehnici de programare 2019

Cuprins

Obiectivul temei

Tema cu numărul trei de laborator are ca obiectiv principal realizarea unei aplicații de *OrderManagement* pentru procesarea comenzilor clienților unui depozit.

Pentru stocarea informațiilor legate de produse, clienți si comenzi se va folosi o baza de date relațională. Totodată, aplicația trebuie sa utilizeze cel puțin următoarele clase:

- **Model** – reprezentând modelul de date al aplicației;
- **Business Logic** – conținând logica aplicației;
- **Presentation** – conținând interfața grafica a aplicației;
- **Data access** – furnizând accesul la baza de date;

Pot fi adăugate si alte clase si pachete pentru implementarea tuturor funcționalităților aplicației.

Cerințe minime pentru nota 5:

- Interfața grafica:
  - Fereastra separata pentru operații asupra clienților (adăugare, editare, ștergere, vizualizare in tabel);
  - Fereastra separata pentru operații asupra produselor (adăugare, editare, ștergere, vizualizare in tabel);
- Posibilitatea de creare a unei noi comenzi prin selectarea unui client si a unui produs existent, introducerea cantității comandate si inserarea noii comenzi in tabelul corespunzător;
- Folosirea a cel puțin trei tabele in baza de date relațională (Clienți, Produse si Comenzi);
- Folosirea tehnicilor de reflexie pentru introducerea datelor într-un tabel;
- Documentație;

Cerințe pentru nota maxima:

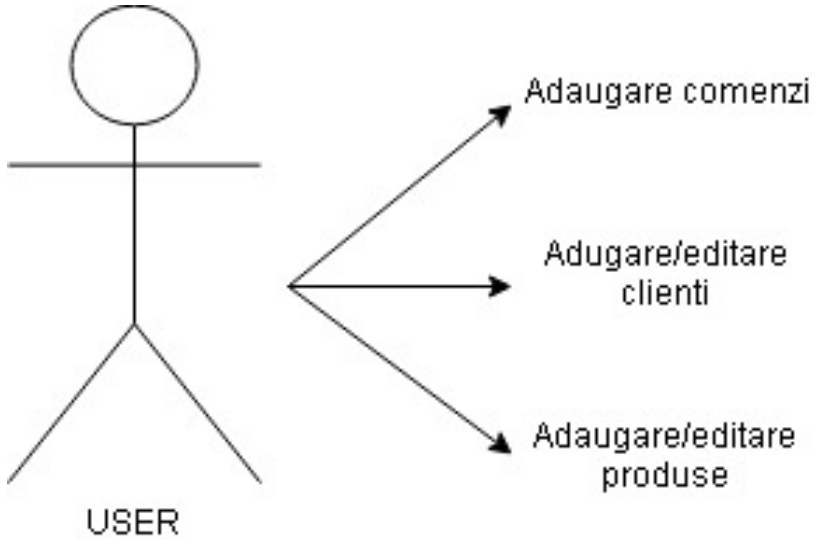
- Crearea unui bon pentru fiecare comanda ca fișier .txt sau .pdf;
- Calitatea documentației;
- *Layered Architecture* (aplicația va conține cel puțin patru pachete: *dataAccessLayer*, *businessLayer*, *model* and *presentation*);

Cerințe pentru puncte suplimentare:

- Structurarea corecta a bazei de date (mai mult de trei tabele);
- Folosirea tehnicilor de reflexie pentru a crea clase generice care conțin metode pentru accesarea bazei de date: creare obiect, editare obiect, ștergere obiect si căutare obiect. Interogările pentru accesarea bazei de date vor fi generate dinamic prin reflexie;

Analiza problemei

După cum este specificat in cerința temei, aplicația trebuie sa furnizeze următoarele funcționalități, prezentate in diagrama de *Use-Case*:



Adăugarea de comenzi trebuie realizata prin selectarea unui produs respectiv a unui client din cei existenți in baza de date. In acest caz a ales dispunerea datelor in trei tabele: clienți, produse si comenzi, care oferă utilizatorului posibilitatea de a selecta datele necesare plasării unei noi comenzi. Totodată, utilizatorului ii este pusa la dispoziție posibilitatea de filtrare a datelor din tabele, conform cerinței, in funcție de fiecare atribut pe care acesta îl selectează.

In ceea ce privește operațiile asupra clienților, acestea se realizează într-o fereastra separata, unde datele sunt de asemenea dispuse într-un tabel. Aici utilizatorul are posibilitatea de a edita, șterge sau adăuga clienți.

Adăugarea si editarea produselor se urmează modelul prezentat in cazul clienților, într-o fereastra separata.

Proiectare

Problema abordata presupune utilizarea unei baze de date pentru stocarea informațiilor utilizate de către aplicație. Astfel, proiectarea presupune crearea bazei de date prin MySQL, respectiv *MySQL Workbench* si *SERVER*.

Baza de date realizata este compusa din trei tabele care au fost populate cu date necesare pentru testarea aplicației, respectiv cu clienți si produse.

In ceea ce privește partea de JAVA a aplicației, conexiunea cu baza de date trebuie realizata prin utilizarea librăriei externe *mysql-connector-java.*

Structura claselor si a pachetelor respecta formatul impus de cerința temei, astfel aplicația realizata se compune din următoarele pachete:

- businessLayer;
- controller;
- dataAccessLayer;
- model;
- view;

Totodata, tema abordata presupune utilizarea reflexiei. Reflecția ajută programatorii să facă biblioteci de software generice pentru a afișa date, pentru a procesa diferite formate de date, pentru a efectua serializarea sau pentru a deserializa datele pentru comunicare sau pentru a face gruparea și separarea datelor pentru containere sau explozii de comunicare. Utilizarea eficientă a reflecției necesită aproape întotdeauna un plan: un cadru de proiectare, o descriere a codificării, o bibliotecă de obiecte, o hartă a unei baze de date sau relații entitate. Reflecția face o limbă mai potrivită pentru codul orientat spre rețea. De exemplu, asistă limbi precum Java pentru a funcționa bine în rețele, permițând bibliotecilor să facă serializări, grupări și diferite formate de date. Limbile fără reflexie (de ex. C) trebuie să utilizeze compilatoare auxiliare, de ex. pentru Notation Syntax Abstract, pentru a produce cod pentru serializare și grupare. Reflecția poate fi utilizată pentru observarea și modificarea executării programului în timpul rulării. O componentă a programului orientată spre reflexie poate monitoriza execuția unei incinte de cod și poate să se modifice în

funcție de scopul dorit legat de acea incintă. Aceasta este de obicei realizată prin atribuirea dinamică a codului de program la timpul de execuție. În limbile de programare orientate pe obiecte, cum ar fi Java, reflecția permite inspecția claselor, interfețelor, câmpurilor și metodelor în timpul execuției fără a cunoaște numele interfețelor, câmpurilor și metodelor la momentul compilării. De asemenea, permite instantierea obiectelor noi și invocarea metodelor. Reflecția este adesea folosită ca parte a testelor software, cum ar fi crearea / instantierea execuției obiectelor mock. Reflecția este, de asemenea, o strategie cheie pentru meta programarea. În unele limbi de programare orientate pe obiect, cum ar fi C # și Java, reflecția poate fi utilizată pentru a suprascrie regulile de accesibilitate a membrilor. De exemplu, reflecția face posibilă modificarea valorii unui câmp marcat "privat" într-o clasă a bibliotecii unei terțe părți.

Aplicația trebuie sa aiba o structura pe niveluri. Arhitectura pe trei niveluri este un model de arhitectură software client-server în care interfața utilizator (prezentare), logica proceselor funcționale ("business rules"), stocarea datelor și accesul la date sunt dezvoltate și menținute ca module independente, cel mai adesea pe platforme separate [10]. Acesta a fost dezvoltat de John J. Donovan în Open Environment Corporation (OEC), o companie de instrumente pe care a fondat-o la Cambridge, Massachusetts. În afară de avantajele obișnuite ale software-ului modular cu interfețe bine definite, arhitectura pe trei niveluri este destinată să permită oricare dintre cele trei niveluri să fie modernizate sau înlocuite în mod independent ca răspuns la modificările cerințelor sau tehnologiei. De exemplu, o schimbare a sistemului de operare în nivelul de prezentare ar afecta numai codul interfeței utilizator. În mod obișnuit, interfața cu utilizatorul rulează pe un PC desktop sau o stație de lucru și utilizează o interfață grafică standard a utilizatorului, o logică a proceselor funcționale care poate consta dintr-unul sau mai multe module separate care rulează pe o stație de lucru sau un server de aplicații și un RDBMS pe un server de bază de date sau un mainframe conține logica de stocare a datelor computerizate. Nivelul de mijloc poate fi multiplicat în sine (caz în care arhitectura generală se numește arhitectură n-tier). Nivel de prezentare Acesta este nivelul cel mai de sus al aplicației. Nivelul de prezentare afișează informații referitoare la servicii cum ar fi răsfoirea mărfurilor, a cumpărăturilor și a conținutului coșului de cumpărături. Ea comunică cu alte niveluri prin care afișează rezultatele la nivelul browser / client și la toate celelalte niveluri din rețea. În termeni simpli, este un strat pe

care utilizatorii pot avea acces direct (cum ar fi o pagină Web sau un GUI al unui sistem de operare). Nivelul aplicației (logică de afaceri, nivel logic sau nivel mediu) Nivelul logic este scos din nivelul de prezentare și, ca strat propriu, controlează funcționalitatea unei aplicații prin efectuarea unei procesări detaliate. Nivelul de date Nivelul de date include mecanismele de persistență a datelor (servere de baze de date, acțiuni de fișiere etc.) și stratul de acces la date care încapsulează mecanismele de persistență și expune datele. Stratul de acces la date trebuie să furnizeze un API pentru nivelul aplicațiilor care expune metode de gestionare a datelor stocate fără a expune sau a crea dependențe de mecanismele de stocare a datelor. Evitarea dependențelor de mecanismele de stocare permite actualizări sau modificări fără ca clienții de nivel aplicație să fie afectați sau chiar conștienți de schimbare. Ca și în cazul separării oricărui nivel, există costuri pentru implementare și, adesea, costuri pentru performanță în schimbul unei scalabilități și a unei mai bune întrețineri.

Implementare

Clasa DatabaseConnection.java

```
package dataAccessLayer;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

public class DatabaseConnection {
    private static final Logger LOGGER =
Logger.getLogger(DatabaseConnection.class.getName());
    private static final String DRIVER = "com.mysql.jdbc.Driver";
    private static final String URL =
"jdbc:mysql://localhost:3306/order_management?useSSL=false";
    private static final String USER = "root";
    private static final String PASSWORD = "avion2015";

    private static DatabaseConnection singleInstance = new DatabaseConnection();

    private DatabaseConnection(){. . .}

    private Connection createConnection(){. . .}

    public static Connection getConnection() {. . .}

    public static void close(Connection connection) {. . .}

    public static void close(Statement statement) {. . .}

    public static void close(ResultSet resultSet) {. . .}
}
```

Conexiunea la baza de date se realizează după cum a precizat prin intermediul librăriei externe *com.mysql.jdbc.* Astfel clasa care realizează conexiunea conține o metoda de *getConnection(),* care apelează constructorul privat pentru a realiză conexiunea. Totodată, aceasta clasa mai conține si metode de închidere a conexiunilor, necesare pentru interogări.

Clasa AbstractDAO.java

```
package dataAccessLayer;

import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;
```

```java
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public abstract class AbstractDAO<T> {
    protected final static Logger LOGGER =
Logger.getLogger(AbstractDAO.class.getName());

    private final Class<T> type;

    public AbstractDAO(){. . .}

    public abstract String createSelectQuery(String field);

    public T findById(int id){. . .}

    protected List<T> createObjects(ResultSet resultSet) {. . .}

    public List<T> findAll() {. . .}

    public abstract void update(T object);

    public void delete(int id){. . .}
}
```

Clasa responsabila de interogările furnizate bazei de date este de tip abstract, fiind extinsa de către toate tipurile de obiecte cu care operează aplicația implementata, si care la rândul lor au implementate metodele necesare. Unele metode sunt de tip abstract la rândul lor, deoarece datele furnizate de către acestea diferă de la o implementare la cealaltă, însă sunt necesare pentru funcționarea corecta a celorlalte metode deja implementate.

Clasa Client.java

```java
package model;

public class Client {
    private int id;
    private String firstName;
    private String name;
    private String address;

    public Client() {. . .}

    public Client(int id, String firstName, String name, String address) {. . .}
```

```
    public int getId() {. . .}

    public void setId(int id) {. . .}

    public String getFirstName() {. . .}

    public void setFirstName(String firstName) {. . .}

    public String getName() {. . .}

    public void setName(String name) {. . .}

    public String getAddress() {. . .}

    public void setAddress(String address) {. . .}
}
```

Fiecare tabel din baza de date are o clasa corespondenta *.java*. In fiecare din aceste cazuri, metodele implementate sunt reprezentate de *getters* si de *setters*, precum si de constructorii necesari.

Clasa Reflection.java

```
package businessLayer;

import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import java.lang.reflect.Field;
import javafx.collections.ObservableList;
import javafx.scene.control.cell.PropertyValueFactory;

public class Reflection {
    public static void objectPrint(Object object){. . .}

    public static void dataToTable(Object object, TableView table){. . .}

    public static void emptyTable(TableView table){. . .}
}
```

Aceasta clasa este responsabila de popularea tabelelor. Acest lucru este realizat prin tehnica reflexiei, după cum sugerează si numele clasei. Metodele sunt de tip static si primesc ca si parametrii obiecte, iar in funcție de tipul acestora tabelele furnizate sunt populate in mod corespunzător cu datele memorate in câmpurile fiecărui obiect.

Crearea bazei de date

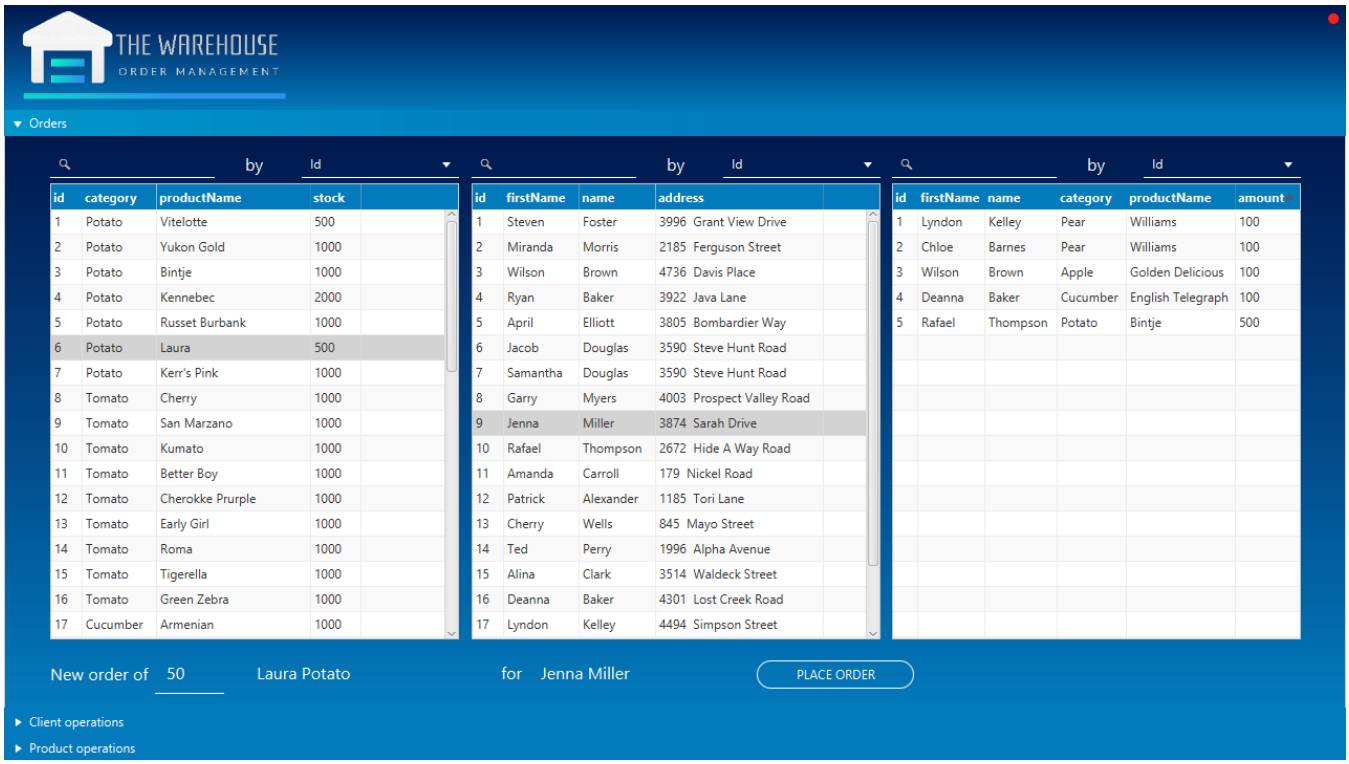
```
DROP SCHEMA if EXISTS order_management;
CREATE SCHEMA order_management;
USE order_management;
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'avion2015';

CREATE TABLE order_management.client (
id INT NOT NULL AUTO_INCREMENT,
firstName VARCHAR(45) NULL,
name VARCHAR(45) NULL,
address VARCHAR(45) NULL,
PRIMARY KEY (id),
UNIQUE INDEX id_UNIQUE (id ASC) VISIBLE);

CREATE TABLE order_management.product (
id INT NOT NULL AUTO_INCREMENT,
category VARCHAR(45) NULL,
productName VARCHAR(45) NULL,
stock INT NULL,
PRIMARY KEY (id),
UNIQUE INDEX id_UNIQUE (id ASC) VISIBLE);

CREATE TABLE order_management.order (
id INT NOT NULL AUTO_INCREMENT,
productId INT NULL,
amount INT NULL,
clientId INT NULL,
PRIMARY KEY (id),
UNIQUE INDEX id_UNIQUE (id ASC) VISIBLE);
```

Interfața grafica



Interfața grafica este realizata folosind JAVAFX si CSS pentru un aspect cat mai prietenos cu utilizatorul. Acesta are la dispoziție multiple filtre de căutare precum si cinci tabele pentru vizualizarea datelor. Alegerea culorilor a fost realizata cu atenție pentru a respecta o tema cat mai minimalista si cu aspect modern. Totodată aplicația mai furnizează in momentul creării comenzilor un bon fiscal cu datele acesteia, care este generat într-un fișier .pdf.

ORDER BILL

ORDER ID.....5
PRODUCT CATEGORY.....Potato
PRODUCT NAME.....Bintje
AMOUNT.....500
CUSTOMER.....Thompson Rafael

Sun May 05 13:45:58 EEST 2019

Rezultate

Aplicația obținuta poate fi ușor adaptata pentru a satisface nevoile unei companii de furnizare de produse. Datele folosite sunt abstracte, astfel putând fi înlocuite cu ușurința, iar utilizarea tehnicilor de reflexie fac posibila utilizarea oricăror structuri de date pentru aplicația implementata.

Concluzii

Tema realizata are ca scop principal dobândirea de informații, iar in acest sens am obținut o multitudine de cunoștințe folositoare, precum tehnica de reflexie si utilizarea bazelor de date pentru stocarea datelor unei aplicații. Aceste informații cresc capacitatea studenților de a face fata la diferite proiecte viitoare.

Bibliografie

→ https://utcn_dsrl@bitbucket.org/utcn_dsrl/pt-layered-architecture.git
→ https://utcn_dsrl@bitbucket.org/utcn_dsrl/pt-reflection-example.git
→ http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/HW3_Tema3/