**PROIECT “ORDER MANAGEMENT”**

FILIP LIVIU PAUL

**Conținut:**

1. Obiectivul temei
2. Analiza problemei, modelare, scenarii, cazuri de utilizare
3. Proiectare
4. Implementare
5. Rezultate
6. Concluzii
7. Bibliografie

**Capitolul I. *Obiectivul temei***

Obiectivul temei “Order Management” îl reprezintă proiectarea și implementarea unei aplicații care lucrează cu o bază de date și poate realiza management-ul unui magazin (clienți, produse, angajați, comenzi).

Îndeplinirea obiectivului propus necesită completarea unor obiective secundare și pași intermediari, elemente pe care le vom sintetiza în următorul tabel:

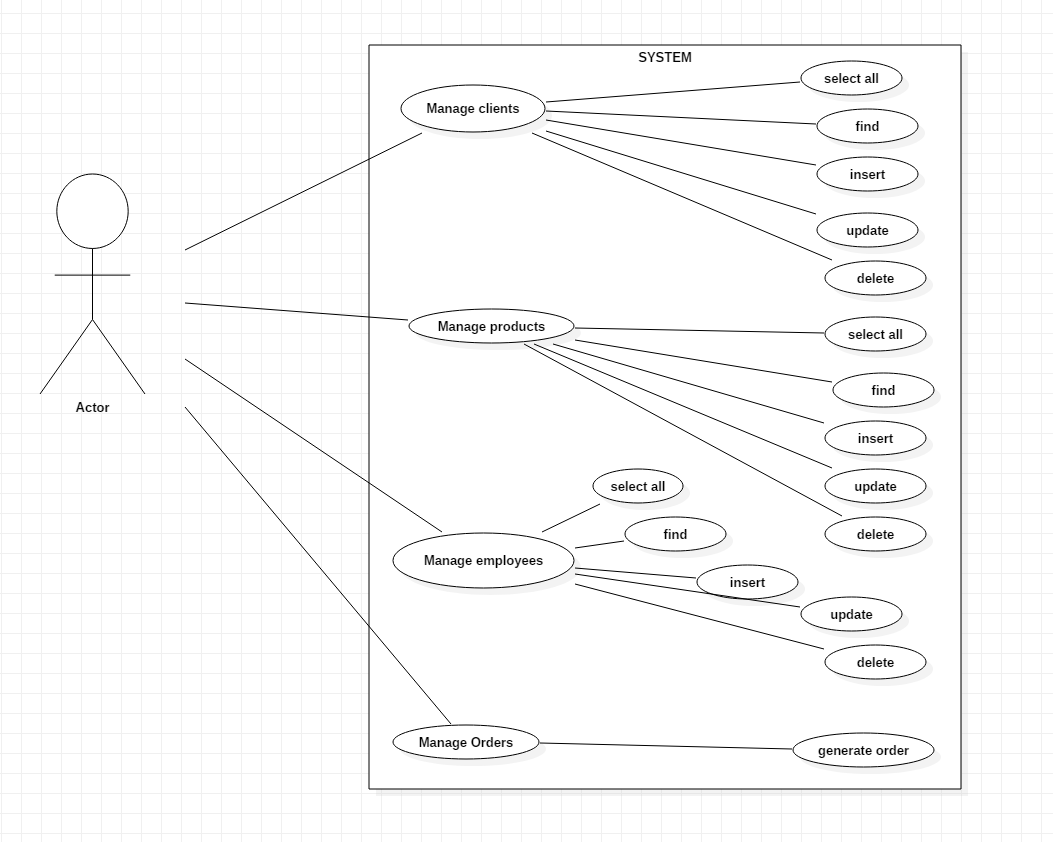
|  |  |
| --- | --- |
| Obiectiv secundar | Descriere |
| Înțelegerea conceptelor identificate în enunțul problemei și transpunerea lor în elemente software care pot manipula astfel de date. | * Se prezintă noțiunile despre bazele de date relaționale și aplicabilitatea lor în dezvoltarea aplicațiilor. * Aceste noțiuni vor fi descrise pe larg în capitolul II. |
| Analizarea problemei, modelarea conceptuală a elementelor ierarhice care constituiesc soluția problemei | * Se vor analiza cerințele problemei, se vor explica acțiunile care trebuie implementate. * Se va explica modelarea soluției pe baza cazurilor de utilizare și a ipotezei problemei. * Aceste sarcini vor fi tratate în capitolul II. |
| Identificarea scenariilor și cazurilor de utilizare a aplicației și rezolvarea eventualelor probleme care pot apărea în urma interacțiunii dintre utilizator și aplicație | * Se vor prezenta cazurile de utilizare sub forma de diagrame și descrieri de use-case. * Descrierile use-case-urilor se vor face sub forma unei liste conținând pașii execușiei pentru fiecare use-case * Se vor identifica eventualele nereguli ce pot apărea la interacțiunea utilizator – program și se va oferi tehnica abordată pentru soluționarea problemei. * Descrierea acestor pași se va face in capitolul II. |
| Alegerea elementelor optime de proiectare corespunzătoare paradigmei de programare orientată pe obiecte  Construirea unei interfețe utilizator prin care interacțiunea dintre sistem și user să fie ușurată, să conțină toate funcționalitățile dorite și să ofere rezultate care pot fi refolosite ulterior | * Se vor expune deciziile de proiectare luate pentru optimizarea aplicației în domeniile timp și memorie. * Se va prezenta proiectarea aplicației în conformitate cu paradigma de programare orientată pe obiecte. * Se prezintă diagramele UML de clase și de pachete pentru proiect. * Se evidențiază structurile de date folosite, algoritmii utilizați și interfețele adăugate. * Se evidențiază structura ierarhică a proiectului, acesta fiind împărțit în pachete, clase și subclase. Relațiile dintre acestea vor fi de asemnea vizualizate cu ajutorul diagramelor UML. * Se va descrie modul de proiectare a interfeței utilizator pentru ca această să satisfacă cerințele utilizatorului, să aibă un aspect “prietenos” și ofere posibilitatea utilizatorului să folosească funcțiile implementate. * Aceste idei vor fi pe larg prezentate în capitolul III. |
| Evidențierea și descrierea etapelor de implementare a problemei. | * Se va prezenta implementarea claselor cu câmpurile și metodele cele mai semnificative. * Se vor evidenția elementele de implementare a unor algoritmi specifici. * Se va descrie modul de implementare a interfeței grafice. * Aceste descrieri vor avea loc în capitolul IV. |
| Interpretarea și evaluarea rezultatelor | * Se vor construi date de intrare, cu ajutorul cărora se vor efectua operațiile implementate în aplicație * Se vor construi atât date valide, cât și invalide pentru evidențierea funcționării aplicației. * Aceste procese se vor prezenta în capitolul V. |
| Concluzii cu privire la temă și dezvoltări ulterioare | * Se vor trage concluzii cu privire la tema abordată, aplicabilitatea ei și eventuale dezvoltări ulterioare. * Acest subiect va fi tratat în capitolul VI. |

**Capitolul II. *Analiza problemei, modelare, scenarii, cazuri de utilizare***

Bazele de date relaționale sunt folosite pentru a stoca tabele ce conțin diverse informații. Fiecare înregistrare din table are asociată o coloană care are rol de “PRIMARY KEY”, adiă oferă fiecărei înregistrări un număr de identificare unic. Între tabelele unei baze de date se pot forma relații de tip Primary Key – Foreign Key ce ajută la eficientizarea operațiilor și a memoriei.

Aplicația implementată va avea 4 tabele și anume: Client, Produs, Angajat și comenzi, iar utilizatorul poate efectua operații de afișare, găsire, adăugare, ștergere, actualizare a tabelelor Client, Produs și Angajat și poate de asemenea să emită comenzi, selectând un client, un produs, un angajat și cantitatea dorită.

Vom prezenta în continuare diagrama USE-CASE a aplicației. Rolul actorului este luat de către utilizatorul aplicației, acesta poate fi un student, un profesor sau orice altă persoană care dorește să efectueze o simulare a acestui proces cu baza de date. În cele ce urmează, vom descrie fiecare use case sub forma unei liste conținând pașii execuției fiecărui use case.



1. Rularea aplicației pentru operare pe tabelul Client

Use case: Selectare operare pe tabel Client + efectuare operații + vizualizare operații

Primary actor: Client

Main succes scenario:

* Clientul alege opțiunea Manage Clients
* Clientul introduce datele în câmpurile specificate
* Clientul alege operația dorită
* Sistemul verifică dacă datele introduse sunt corecte
* În caz afirmativ, în fereastră, se observă în tabel eventualele modificări efecutate
* Tabelul se va actualiza după fiecare modificare făcută

Alternative sequences:

* Nu sunt introduși toți parametrii sau aceștia sunt invalizi pentru începerea simulării. În acest caz, se va afișa o fereastră de eroare.

Pentru tabelele Product și Employee, se respect același scenario ca pentru tabelul Client.

1. Rularea aplicației pentru generarea comenzilor

Use case: Selectare operare pe tabel Oder + efectuare operații + vizualizare operații

Primary actor: Client

Main succes scenario:

* Clientul alege opțiunea Manage Orders
* Clientul introduce datele în câmpurile specificate
* Clientul alege operația Generate Order
* Sistemul verifică dacă datele introduse sunt corecte
* La final, se va vedea factura în format PDF.

Alternative sequences:

* Nu sunt introduși toți parametrii sau aceștia sunt invalizi pentru începerea simulării. În acest caz, se va afișa o fereastră de eroare.
* Dacă cantitatea introdusă este mai mare decât stocul existent, se va afișa o fereastră cu un mesaj de eroare.

**Capitolul III. *Proiectare***

În acest capitol, vom prezenta pe larg deciziile de proiectare luate în concordanță cu paradigma programării orientate pe obiecte, vom expune diagramele UML de clase, structura ierarhică a proiectului și strucuturile de date folosite.

Aplicația este structurată în pachete, cele principale fiind pachetele BLL, BLLValidators, Connection, DAO, Modele și Presentation în care sunt implementate atât structurile de date în care stocăm informația despre tabelele din BD, generearea și repartizarea informațiilor în simulare cât și în care avem implementată interfața grafică.

Diagrama UML pentru pachete :

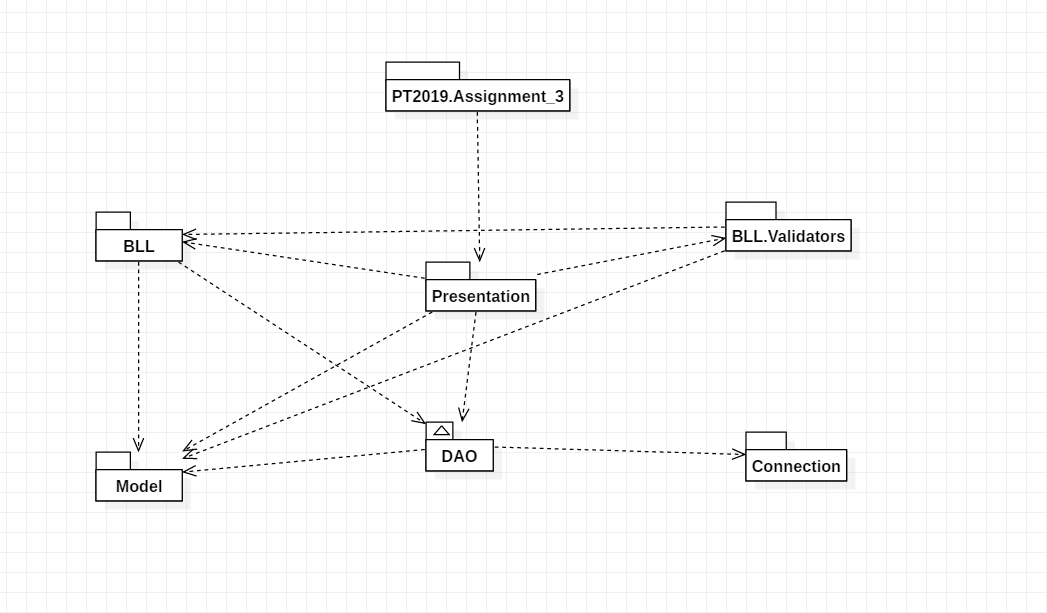


Diagrama UML pentru pachetul BLL:

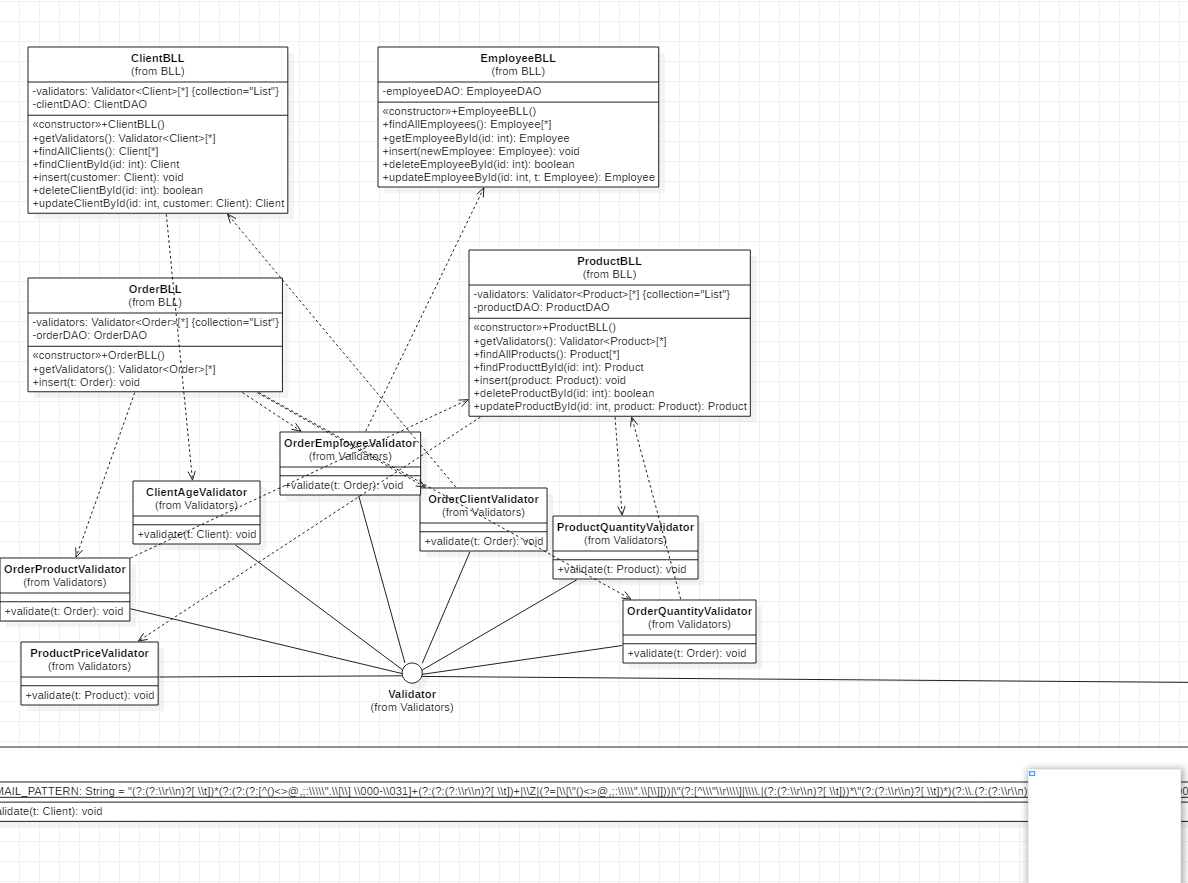


Diagrama UML pentru pachetul Connection:

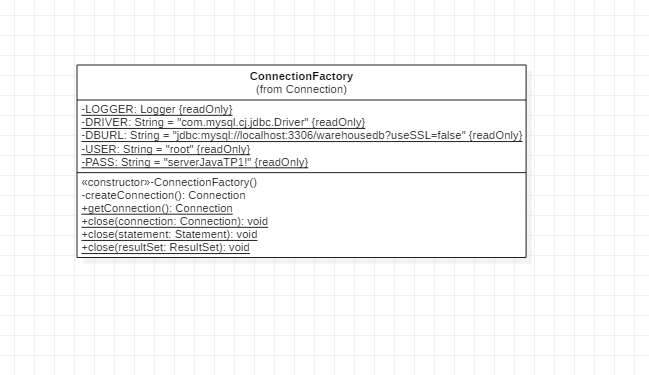


Diagrama UML pentru pachetul DAO :

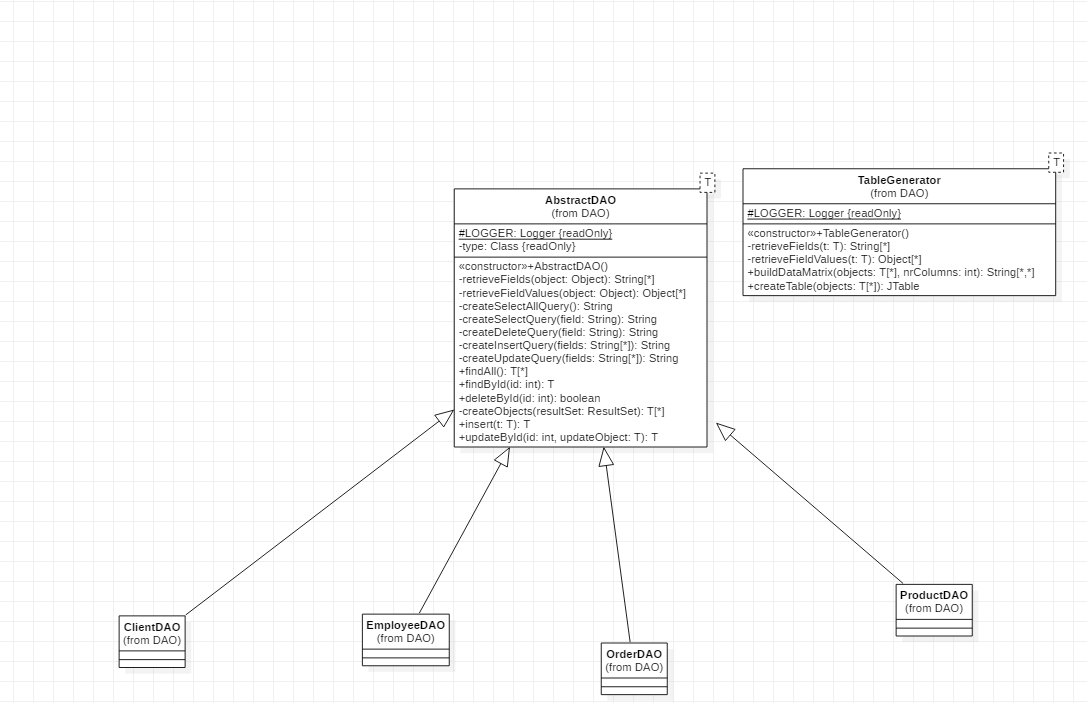


Diagrama UML pentru pachetul Model:

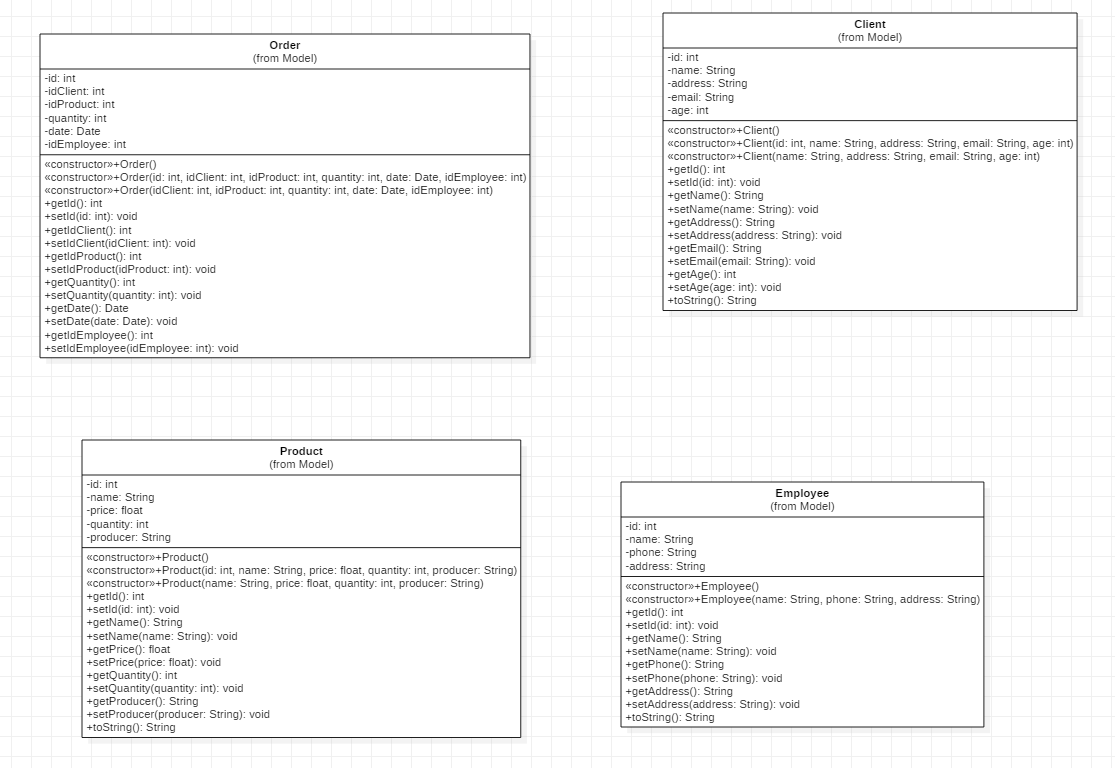


Diagrama UML pentru pachetul Presentation:

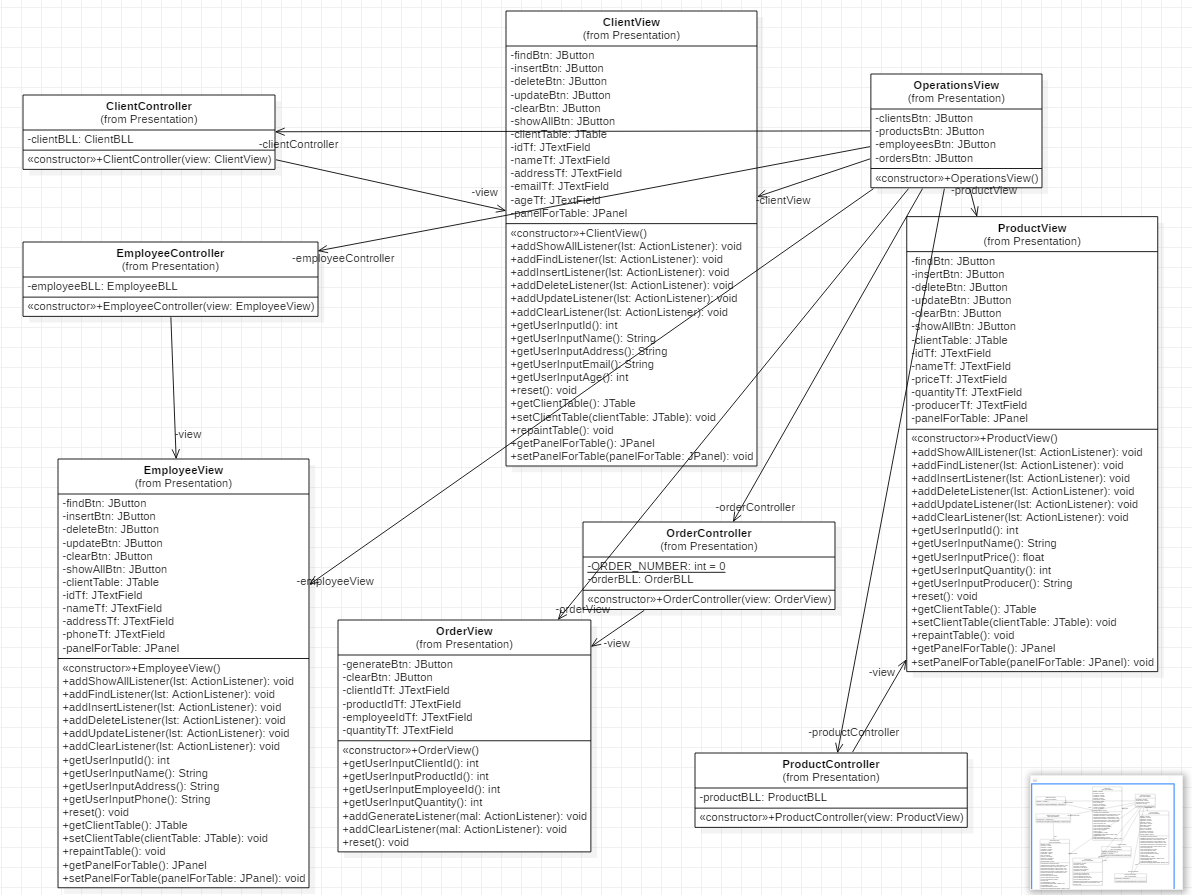
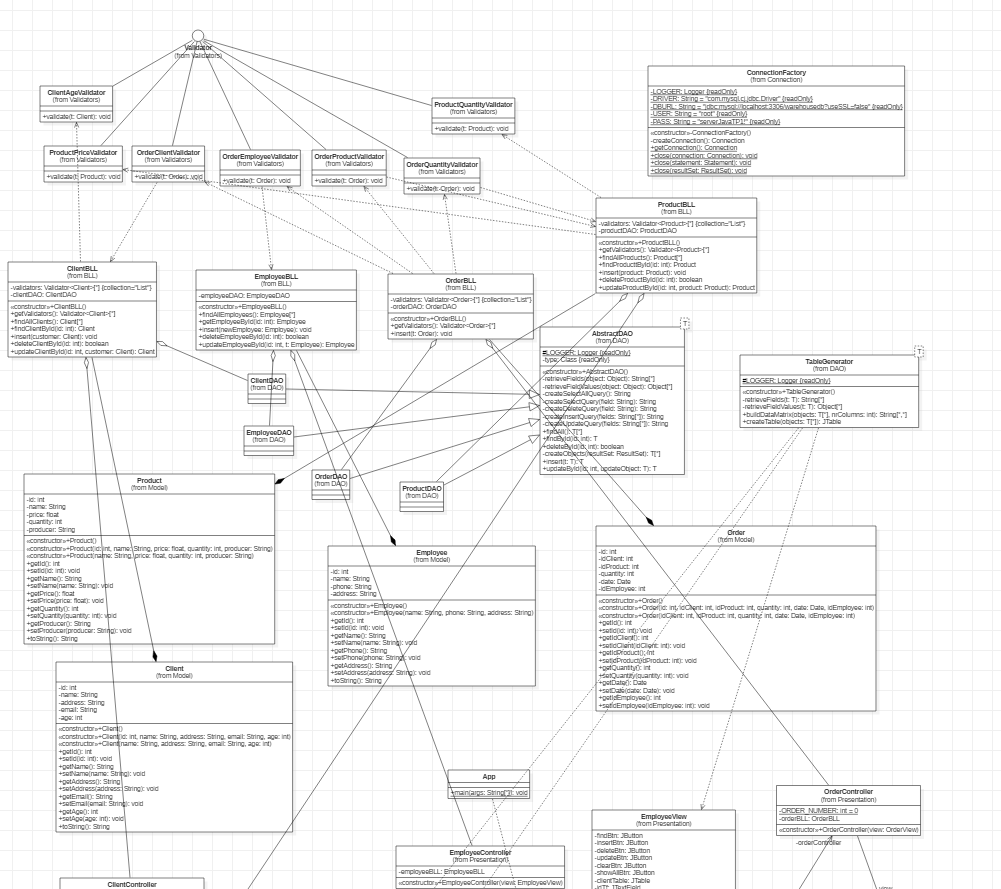


Diagrama UML de clase pentru întreg proiectul:





**Capitolul IV.  *Implementare***

Aplicația este strucutrată ierarhic, pe pachecte, urmărind arhitectura 3-tier, în care fiecare layer are un scop special și apelează funții din layer-ele inferioare.

Aplicația este structurată în următoarele Layer: Model – modelează tabelele din baza de date, Data Access layer – conține clasele cu interogări și conexiunea la BD, Business Layer – conține clasele care încapsulează logica aplicației și Presentation Layer – conține clasele care implementează interfața grafică.

Pentru a ușura întelegerea implementării aplicației, fiecare funcție va avea un javadoc în care este explicată utilizarea funcției.

Pachetul Model conține clasele Client, Product, Employee, Order și acestea se mapează direct cu tabelelel din BD. Exemplu:

**public** **class** Client {

**private** **int** id;

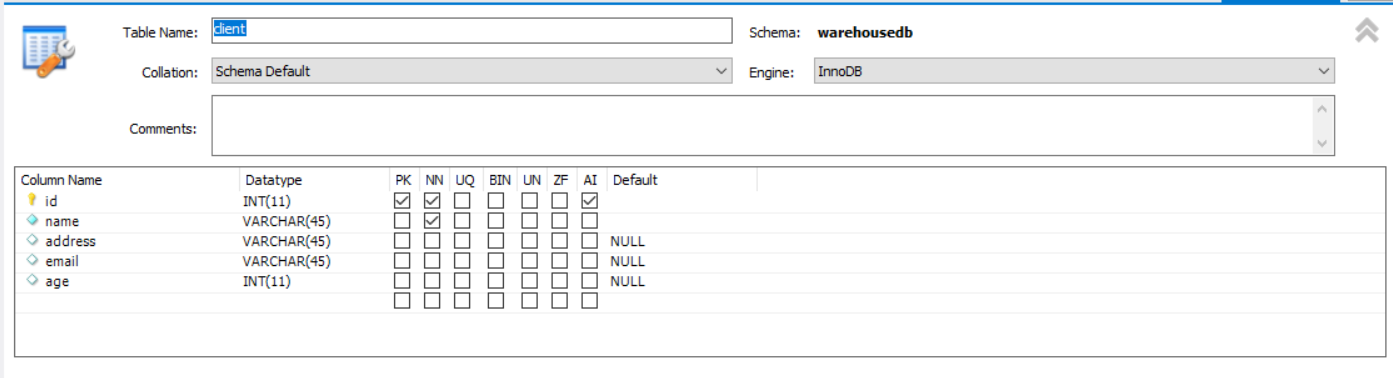
**private** String name;

**private** String address;

**private** String email;

**private** **int** age;

Atributele clasei corespund cu câmpurile tabelului din BD:



În pachetul Presentation, implementăm interfața grafică și logica interfeței pentru fiecare table, dar și un meniu principal de unde putem select ace dorim să facem. Următorul fragment de cod este implementarea acestui tip de meniu:

package Presentation;

import java.awt.GridLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

/\*\*

\* Clasa pentru GUI meniu principal

\*

\* @author user

\*

\*/

public class OperationsView extends JFrame {

private JButton clientsBtn = new JButton("Manage Clients");

private JButton productsBtn = new JButton("Manage Products");

private JButton employeesBtn = new JButton("Manage Employees");

private JButton ordersBtn = new JButton("Manage Orders");

private ClientView clientView;

private ClientController clientController;

private ProductView productView;

private ProductController productController;

private OrderView orderView;

private OrderController orderController;

private EmployeeView employeeView;

private EmployeeController employeeController;

public OperationsView() {

this.setTitle("Main Frame");

this.setSize(600, 300);

this.setResizable(false);

this.setLayout(new GridLayout(4, 0));

this.add(clientsBtn);

this.add(productsBtn);

this.add(employeesBtn);

this.add(ordersBtn);

clientsBtn.addActionListener(new ClientsListener());

productsBtn.addActionListener(new ProductsListener());

employeesBtn.addActionListener(new EmployeesListener());

ordersBtn.addActionListener(new OrdersListener());

this.setVisible(true);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

class ClientsListener implements ActionListener {

public void actionPerformed(ActionEvent e) {

// TODO Auto-generated method stub

clientView = new ClientView();

clientController = new ClientController(clientView);

}

}

class ProductsListener implements ActionListener {

public void actionPerformed(ActionEvent e) {

// TODO Auto-generated method stub

productView = new ProductView();

productController = new ProductController(productView);

}

}

class OrdersListener implements ActionListener {

public void actionPerformed(ActionEvent e) {

// TODO Auto-generated method stub

orderView = new OrderView();

orderController = new OrderController(orderView);

}

}

class EmployeesListener implements ActionListener {

public void actionPerformed(ActionEvent e) {

// TODO Auto-generated method stub

employeeView = new EmployeeView();

employeeController = new EmployeeController(employeeView);

}

}

}

În pachetul DAO, clasa AbstractDAO este o clasă care utilizează tehnicile de reflexie pentru a efectua operațiile de find, insert, update, delete pentru orice obiect care se dă ca parametru.

**public** **class** AbstractDAO<T> {

**protected** **static** **final** Logger ***LOGGER*** = Logger.*getLogger*(AbstractDAO.**class**.getName());

**private** **final** Class<T> type;

@SuppressWarnings("unchecked")

**public** AbstractDAO() {

**this**.type = (Class<T>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[0];

}

/\*\*

\*

\* **@param** object

\* un obiect

\* **@return** o lista de String ce reprezinta numele campurilor obiectului

\*/

**private** ArrayList<String> retrieveFields(Object object) {

ArrayList<String> list = **new** ArrayList<String>();

**for** (Field field : object.getClass().getDeclaredFields()) {

**if** (!field.getName().equals("id")) {

list.add(field.getName());

}

}

**return** list;

}

/\*\*

\*

\* **@param** object

\* un obiect

\* **@return** un ArrayList de obiecte ce reprezinta valorile atributelor

\* obiectului

\*/

**private** ArrayList<Object> retrieveFieldValues(Object object) {

ArrayList<Object> list = **new** ArrayList<Object>();

**for** (Field field : object.getClass().getDeclaredFields()) {

**if** (!field.getName().equals("id")) {

field.setAccessible(**true**);

Object value;

**try** {

value = field.get(object);

list.add(value);

} **catch** (IllegalArgumentException e) {

e.printStackTrace();

} **catch** (IllegalAccessException e) {

e.printStackTrace();

}

}

}

**return** list;

}

Aceste metode sunt folosite pentru a extrage câmpurile și valorile pentru obiecte și acestea vor fi folosite pentru construirea query-urilor. De exemplu, pentru find by id:

/\*\*

\*

\* **@param** field

\* string care reprezinta campul dupa care se face selectia din

\* tabelul aflat in BD

\* **@return** un string ce creeaza query-ul aferent

\*/

**private** String createSelectQuery(String field) {

StringBuilder sb = **new** StringBuilder();

sb.append("SELECT ");

sb.append(" \* ");

sb.append(" FROM ");

sb.append(type.getSimpleName());

sb.append(" WHERE " + field + " =?");

**return** sb.toString();

}

/\*\*

\*

\* **@param** id

\* id-ul inregistrarii din tabel

\* **@return** obiectul cu atributele din tabel daca exista si null daca nu

\*/

**public** T findById(**int** id) {

Connection connection = **null**;

PreparedStatement statement = **null**;

ResultSet resultSet = **null**;

String query = createSelectQuery("id");

**try** {

connection = ConnectionFactory.*getConnection*();

statement = connection.prepareStatement(query);

statement.setInt(1, id);

resultSet = statement.executeQuery();

**return** createObjects(resultSet).get(0);

} **catch** (SQLException e) {

***LOGGER***.log(Level.***WARNING***, type.getName() + "DAO:findById " + e.getMessage());

} **catch** (IndexOutOfBoundsException e) {

***LOGGER***.log(Level.***WARNING***, type.getName() + "DAO:findById " + e.getMessage());

} **finally** {

ConnectionFactory.*close*(resultSet);

ConnectionFactory.*close*(statement);

ConnectionFactory.*close*(connection);

}

**return** **null**;

}

În pachetul Connection, avem implementată clasa ConnectionFactory care realizează conexiunea cu BD:

**public** **class** ConnectionFactory {

**private** **static** **final** Logger ***LOGGER*** = Logger.*getLogger*(ConnectionFactory.**class**.getName());

**private** **static** **final** String ***DRIVER*** = "com.mysql.cj.jdbc.Driver";

**private** **static** **final** String ***DBURL*** = "jdbc:mysql://localhost:3306/warehousedb?useSSL=false";

**private** **static** **final** String ***USER*** = "root";

**private** **static** **final** String ***PASS*** = "serverJavaTP1!";

**private** **static** ConnectionFactory *singleInstance* = **new** ConnectionFactory();

**private** ConnectionFactory() {

**try** {

Class.*forName*(***DRIVER***);

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

}

}

**private** Connection createConnection() {

Connection connection = **null**;

**try** {

connection = DriverManager.*getConnection*(***DBURL***, ***USER***, ***PASS***);

} **catch** (SQLException e) {

***LOGGER***.log(Level.***WARNING***, "An error occured while trying to connect to the database");

e.printStackTrace();

}

**return** connection;

}

**public** **static** Connection getConnection() {

**return** *singleInstance*.createConnection();

}

**public** **static** **void** close(Connection connection) {

**if** (connection != **null**) {

**try** {

connection.close();

} **catch** (SQLException e) {

***LOGGER***.log(Level.***WARNING***, "An error occured while trying to close the connection");

}

}

}

**public** **static** **void** close(Statement statement) {

**if** (statement != **null**) {

**try** {

statement.close();

} **catch** (SQLException e) {

***LOGGER***.log(Level.***WARNING***, "An error occured while trying to close the statement");

}

}

}

**public** **static** **void** close(ResultSet resultSet) {

**if** (resultSet != **null**) {

**try** {

resultSet.close();

} **catch** (SQLException e) {

***LOGGER***.log(Level.***WARNING***, "An error occured while trying to close the ResultSet");

}

}

}

În pachetul BLL.Validators avem definite clase care implementează interfața Validator pentru certificarea datelor introduse.

/\*\*

\* Interfata pentru validarea parametrilor in vederea operatiilor cu BD

\*

\* **@author** user

\*

\* **@param** <T>

\* clasa obiectului pentru care se face validarea

\*/

**public** **interface** Validator<T> {

**public** **void** validate(T t);

}

/\*\*

\* implementeaza validator, verifica daca un client are varsta eligibila pentru

\* a putea fi stocat in BD

\*

\* **@author** user

\*

\*/

**public** **class** ClientAgeValidator **implements** Validator<Client> {

**public** **void** validate(Client t) {

// **TODO** Auto-generated method stub

**if** (t.getAge() < 18 || t.getAge() > 85) {

**throw** **new** IllegalArgumentException("Age constraint violated");

}

}

}

Clasele din pachetul BLL reprezintă clasele de business pentru aplicația noastră. De exemplu, pentru tabelul Order, avem:

**public** **class** OrderBLL {

**private** List<Validator<Order>> validators;

**private** OrderDAO orderDAO;

**public** OrderBLL() {

validators = **new** ArrayList<Validator<Order>>();

validators.add(**new** OrderClientValidator());

validators.add(**new** OrderProductValidator());

validators.add(**new** OrderEmployeeValidator());

validators.add(**new** OrderQuantityValidator());

orderDAO = **new** OrderDAO();

}

**public** List<Validator<Order>> getValidators() {

**return** validators;

}

**public** **void** insert(Order t) {

Order ord = orderDAO.insert(t);

**if** (t == **null**) {

**throw** **new** NoSuchElementException("Insert error");

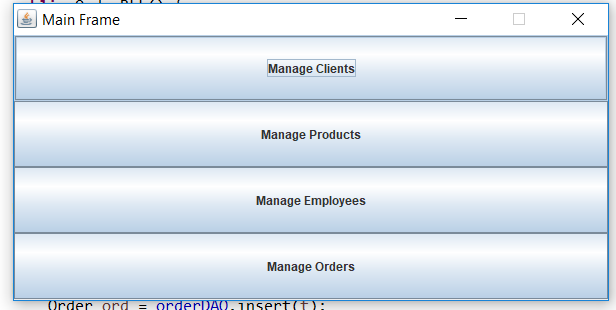
}

}

}

**Capitolul V. *Rezultate***

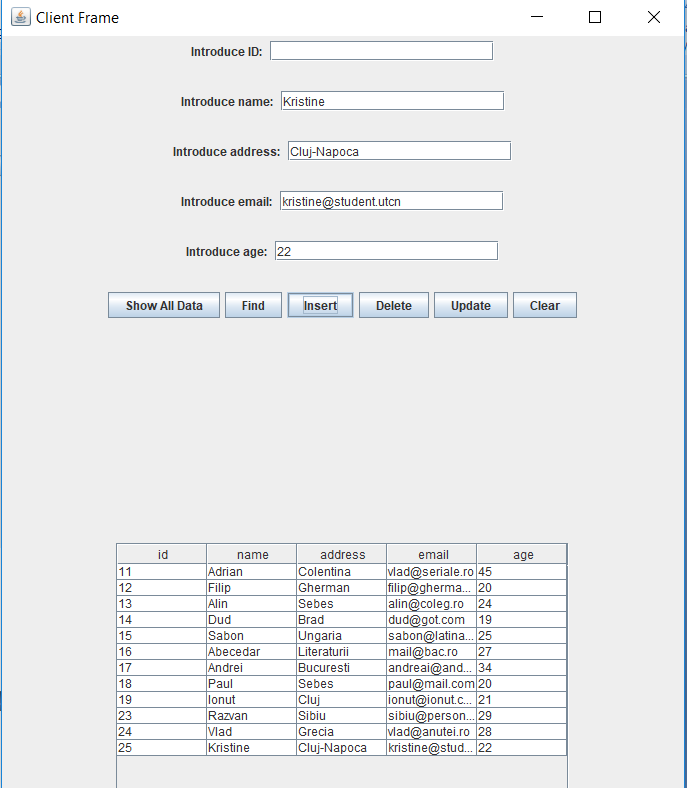
Prima dată , vom alege o opțiune din meniu:



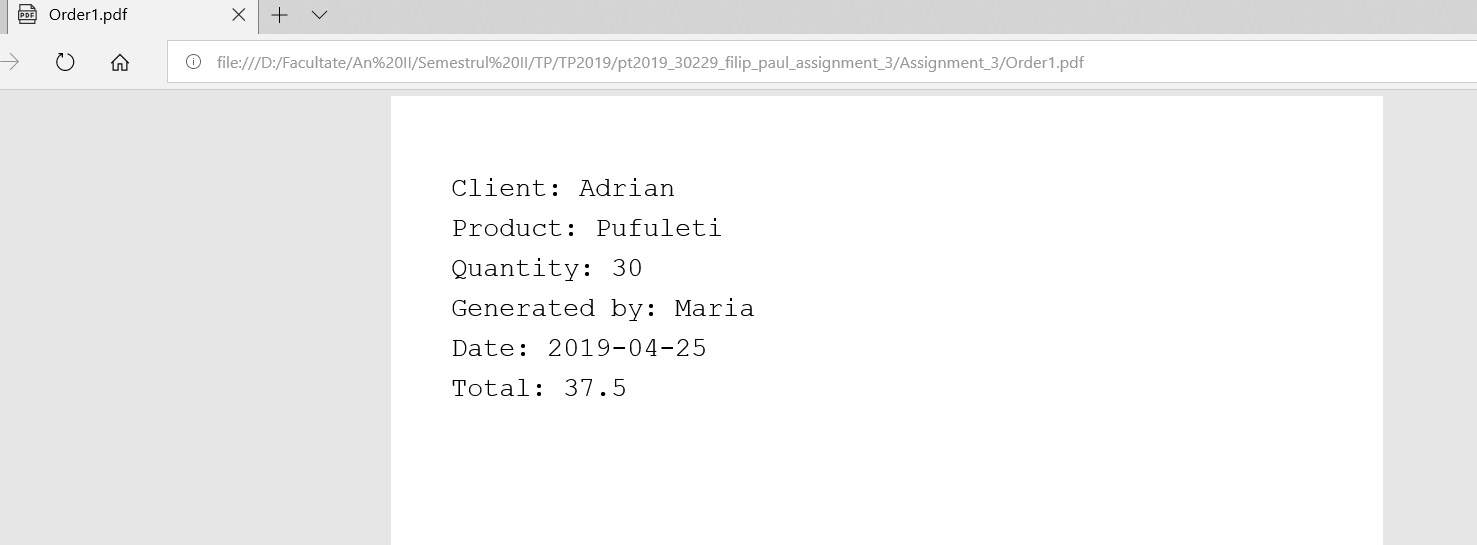
Alegând ManageClients, ni se va afișa frame-ul pentru lucrul cu clienți



Introducem date și apăsam Insert. Se va observa modificarea tabelului:



În mod identic, se poate lucra și cu celelalte tabele. Pentru tabelul Order, se va genera, în schimb un PDF cu comanda, astfel:



**Capitolul VI. *Concluzii***

Tema prezentată impune cunoașterea și aplicarea noțiunilor din paradigma programării orientate pe obiect. Așadar, rezolvarea cerinței permite aprofundarea și dezvoltarea capacităților de a scrie cod Java, identificarea cauzelor și tratarea problemelor ce pot apărea în procesul de implementare. Totodată, datorită implementării și a unei interfețe grafice, programatorul trebuie să conceapă o interfață inteligibilă prin care utilizatorul poate experimenta toate funcționalitățile puse în evidență.

Această temă este importantă deoarece utilizează concepte din teoria bazelor de date și implementarea aplicațiilor cu baze de date intergrate prin Java. Totodată, această problemă stă la baza modelării proceselor din viața reală și rezolvarea acesteia oferă un avantaj atât în evidențierea trăsăturilor programării cu baze de date, cât și în abstractizarea cât mai eficientă a problemelor din viața reală pentru implementare software.

Aplicația curentă poate urma diferite etape de dezvoltare, atât pe parte grafică, cât și pe parte de eficientizare și algoritmică sau mapare pe probleme de gestiune din viața reală.

**Capitolul VII. *Bibliografie***

[1] <http://users.utcluj.ro/~crisb_pop/pt2018.html>

[2] <http://users.utcluj.ro/~igiosan/teaching_poo.html>

[3] <https://www.tutorialspoint.com/>