# DOCUMENTAȚIE

TEMA 3

Order Management

Nume student: VELICEA ANDREEA – IOANA

Grupa: 30228

# Contents

1. Obiectivul temei ..... 4
1.1 Obiective secundare ..... 4
2. Analiza problemei, modelare, scenarii, cazuri de utilizare ..... 4
2.1 Cerinte functionale ..... 4
2.2 Cerinte non functionale ..... 4
2.3 Prezentarea use-case-urilor ..... 4
Use case intefata View: ..... 4
Use case interfata ClientView, ProductView, OrderView: ..... 5
Use case interfata Bill: ..... 6
3. Proiectarea aplicatiei de simulare ..... 6
3.1 Arhitectura Conceptuala ..... 6
3.2 Divizarea in pachete ..... 7
3.3 Divizarea in clase ..... 8
4. Implementarea ..... 8
4.1 Clasa App ..... 8
4.2 Clasa View ..... 9
4.3 Clasa ClientView ..... 9
4.4.Clasa ProductView ..... 11
4.5 Clasa OrderView ..... 13
4.6 Clasa BillView ..... 16
4.7 Clasa Controller ..... 17
4.8 Clasa ConnectionFactory ..... 18
4.9 Clasa Client ..... 19
4.10 Clasa Product ..... 19
4.11 Clasa Order ..... 19
4.12 Clasa Bill ..... 19
4.13 Clasa AbstractDAO ..... 19
4.14 Clasa ClientDAO ..... 23
4.15 Clasa ProductDAO ..... 25
4.16 Clasa OrderDAO ..... 26
4.17 Clasa BillDAO ..... 29
4.18 Clasa ClientBLL ..... 30

4.19 Clasa ProductBLL ..... 31
4.20 Clasa OrderBLL ..... 32
4.21 Clasa BillBLL ..... 34
4.22 Clasa EmailValidator ..... 34
4.23 Clasa QuantityValidator ..... 35
5. Rezultate ..... 35
6. Concluzii ..... 37
7. Bibliografie ..... 37

1. Obiectivul temei
2. Analiza problemei, modelare, scenarii, cazuri de utilizare

## 2.1 Cerinte functionale

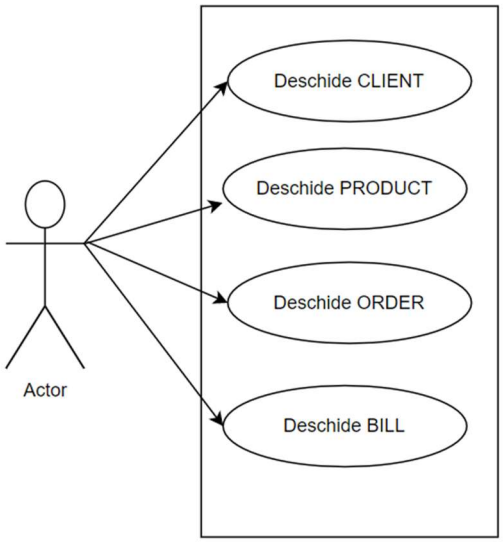
- Aplicatia ar trebui sa permita utilizatorului sa introduca un nou client
- Aplicatia ar trebui sa permita utilizatorului sa introduca un nou produs
- Aplicatia ar trebui sa permita utilizatorului sa introduca o noua comanda
- Aplicatia ar trebui sa permita utilizatorului sa stearga un client
- Aplicatia ar trebui sa permita utilizatorului sa stearga un produs
- Aplicatia ar trebui sa permita utilizatorului sa stearga o comanda
- Aplicatia ar trebui sa permita utilizatorului sa modifice un client
- Aplicatia ar trebui sa permita utilizatorului sa modifice un produs
- Aplicatia ar trebui sa permita utilizatorului sa vizualizeze toti clientii
- Aplicatia ar trebui sa permita utilizatorului sa vizualizeze toate produsele
- Aplicatia ar trebui sa permita utilizatorului sa vizualizeze toate comenzile

## 2.2 Cerinte non functionale

- Aplicatia ar trebui sa fie intuitiva si usor de utilizat de catre utilizator

## 2.3 Prezentarea use-case-urilor

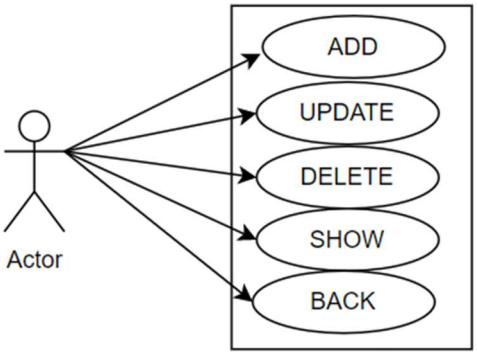
Use case intefata View:




Use case: schimbarea interfetei
Actor principal: Utilizatorul
Scenariu principal de success:

- Utilizatorul selecteaza unul din cele 4 butoane representative fiecarei interfete
- Butonul Client transfera utilizatorul in interfata Client
- Butonul Product transfera utilizatorul in interfata Product
- Butonul Order transfera utilizatorul in interfata Order
- Butonul Bill transfera utilizatorul in interfata Bill

## Use case interfata ClientView, ProductView, OrderView:




Use case: alegerea operatiei

Actor principal: utilizatorul

Scenariul principal de success:

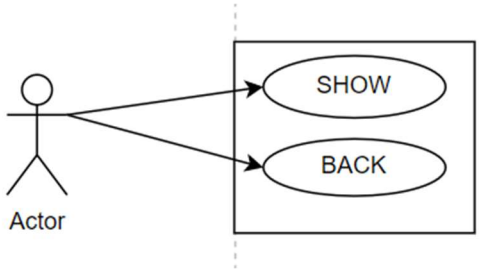
- Utilizatorul insereaza elemente in casutele representative campurilor din enitatea bazei de date din tabelele respective

- Apoi se alege operatie care vrem sa o efectuam: ADD, UPDATE, DELETE.
- Datele sunt verificate si daca apasam SHOW TABLE o sa vedem cum se modifica inregistrarile
- Apoi cand am terminat ceea ce aveam de facut, apasam BACK si ne intoarcem la interfata principala

Scenariu alternativ:

- In momentul in care apasam pe o operatie fara sa adaugam in casete valori valide, o sa apara o exceptie si va trebui sa adaugam din nou.

Use case interfata Bill:




Use case: arata tabel

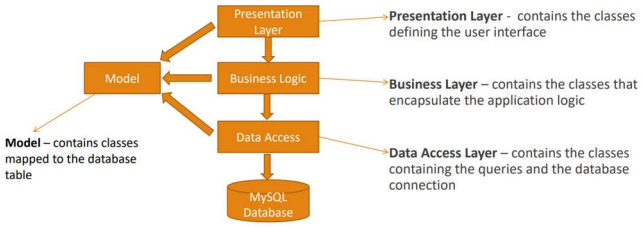
Actor principal: utilizatorul

Scenariu principal de success:

- In momentul in care utilizatorul apasa butonul SHOW TABLE, tabelul cu facturi este afisat pe ecran
- Iar mai apoi daca este apasat butonul de BACK se revine la interfata principala

# 3. Proiectarea aplicatiei de simulare

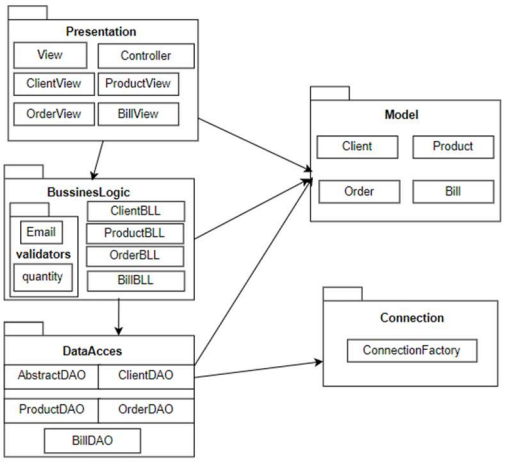
## 3.1 Arhitectura Conceptuala



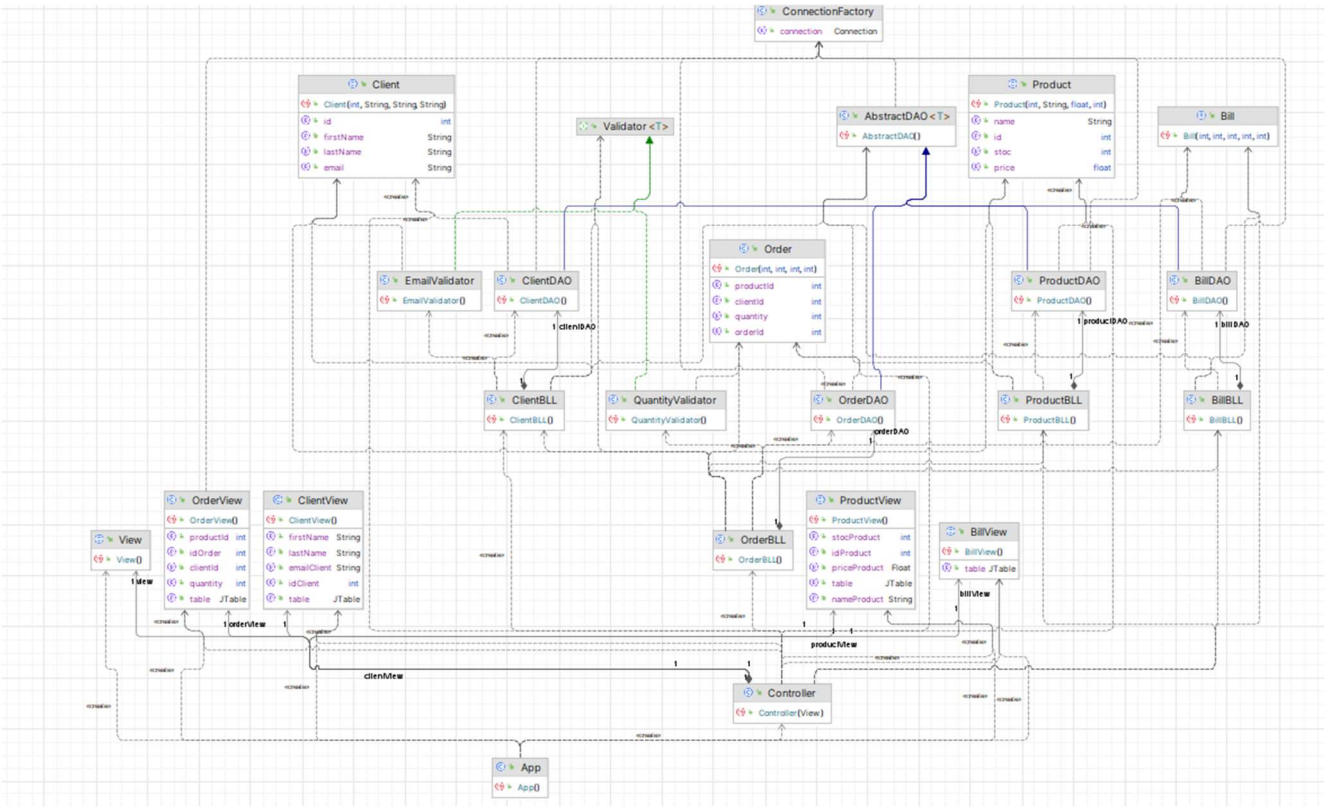
## 3.2 Divizarea in pachete

Am impartit clasele in 5 pachete:

- BussinesLogic
  - Validators
    - EmailValidator
    - QuantityValidator
    - Validator
  - BillBLL
  - ClientBLL
  - ProductBLL
  - ClientBLL
- Connection
  - ConnectionFactory
- DataAcces
  - AbstractDAO
  - BillDAO
  - ClientDAO
  - ProductDAO
  - OrderDAO
- Model
  - Bill
  - Client
  - Product
  - Order
- Presentation
  - BillView
  - ClientView
  - Controller
  - ProductView
  - OrderView
  - View



### 3.3 Divizarea in clase




# 4. Implementarea

### 4.1 Clasa App

- Metoda main in care imi creez interfetele si controller-ul

```
-   public static void main(String[] args) {

        ClientView client = new ClientView();
        ProductView product = new ProductView();
        OrderView order = new OrderView();
        BillView billView = new BillView();
        View view = new View();
        Controller controller = new Controller(view);
        view.setVisible(true);

    }
```

## 4.2 Clasa View

- Fereastra principala de unde pot sa ajung in fiecare tabel: Client, Product, Order
- are ca si atribute:
    - contentPane: JPanel
    - tableClient, tableProduct, tableOrder, showBill: JButton
    - title, chooseTable :JLabel
- Implementeaza metodele:

```
/**
 * Metoda care adauga actionListener pentru butonul de
afisare a inteferetei Client
 * @param actionListener
 */
public void clientShow(ActionListener actionListener) {
    this.tableClient.addActionListener(actionListener);
}

/**
 * Metoda care adauga actionListener pentru butonul de
afisare a intefetei Product
 * @param actionListener
 */
public void productShow(ActionListener actionListener) {
    this.tableProduct.addActionListener(actionListener);
}

/**
 * Metoda care adauga actionListener pentru butonul de
afisare a interfetei Order
 * @param actionListener
 */

public void orderShow(ActionListener actionListener) {
    this.tableOrder.addActionListener(actionListener);
}

/**
 * Metoda care adauga actionListener pentru butonul de
afisare a interfetei Bill
 * @param actionListener
 */
public void billShow(ActionListener actionListener){
    this.showBill.addActionListener(actionListener);
}
```

## 4.3 Clasa ClientView

- Clasa care se ocupa de realizarea interfetei pentru operatii cu client
- Are ca si atribute:
    - contentPane:JPanel
    - id_label, email_label, firstName_label, lastName_label,title: JLabel

- id_textField, email_textField, firstName_textField, lastName_textField: JTextField
- table: JTable
- deteleClient, addClient, updateClient, showTableClient, backButton, clear: JButton
- jScrollPane: JScrollPane

➢ Implementeaza metodele

```
➢  /**
     * Getter ul pentru id-ul introdus de utilizator
     * @return id-ul introdus de utilizator
     */
    public int getIdClient() {
        return Integer.parseInt(this.id_textField.getText());
    }

    /**
     * Getter ul pentru email-ul introdus de utilizator
     * @return email-ul introdus de utilizator
     */
    public String getEmailClient() {
        return this.email_textField.getText();
    }

    /**
     * Getter ul pentru firstName introdus de utilizator
     * @return firstName introdus de utilizator
     */
    public String getFirstName() {
        return this.firstName_textField.getText();
    }

    /**
     * Getter ul pentru lastName introdus de utilizator
     * @return lastName introdus de utilizator
     */
    public String getLastName() {
        return this.lastName_textField.getText();
    }

    /**
     * Getter ul pentru tabel
     * @return tabelul
     */
    public JTable getTable() {
        return table;
    }

    /**
     * Metoda care elibereaza casetele pentru reintroducerea altor date
     */

    public void clear() {
        id_textField.setText("");
        email_textField.setText("");
        firstName_textField.setText("");
        lastName_textField.setText("");
    }
```

```java
    /**
     * Metoda carea adauga actionListener pentru butonul de ADD
     * @param actionListener
     */

    public void addClientListener(ActionListener actionListener) {
        this.addClient.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de UPDATE
     * @param actionListener
     */
    public void updateClientListener(ActionListener actionListener) {
        this.updateClient.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de DELETE
     * @param actionListener
     */
    public void deleteClientListener(ActionListener actionListener) {
        this.deleteClient.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de SHOW TABLE
     * @param actionListener
     */
    public void showTableClientListener(ActionListener actionListener) {
        this.showTableClient.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de BACK
     * @param actionListener
     */
    public void backListener(ActionListener actionListener) {
        this.backButton.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de CLEAR
     * @param actionListener
     */
    public void clearListener(ActionListener actionListener) {
        this.clear.addActionListener(actionListener);
    }
```

## 4.4.Clasa ProductView

- Clasa care se ocupa de realizarea interfetei pentru operatii cu produse
- Are ca si atribute:
    - contentPane: JPanel

- id_label, name_label, price_label, title stoc_label: JLabel
- id_textField, name_textField, price_textField, stoc_textField: JTextField
- table: JTable
- addProduct, deleteProduct, updateProduct, showTabelProduct, clear: JButton
- jScrollPane: JScrollPane

➢ Implementeaza metodele

```
➢  /**
    * Getter ul pentru id-ul introdus de utilizator
    * @return id-ul introdus de utilizator
    */
   public int getIdProduct() {
       return Integer.parseInt(id_textField.getText());
   }

   /**
    * Getter ul pentru name introdus de utilizator
    * @return name introdus de utilizator
    */
   public String getNameProduct() {
       return name_textField.getText();
   }

   /**
    * Getter ul pentru price introdus de utilizator
    * @return price introdus de utilizator
    */
   public Float getPriceProduct() {
       return Float.parseFloat(price_textField.getText());
   }

   /**
    * Getter ul pentru stoc introdus de utilizator
    * @return stoc introdus de utilizator
    */
   public int getStocProduct(){ return
   Integer.parseInt(stoc_textField.getText());}

   /**
    * Getter ul pentru tabel
    * @return tabel
    */

   public JTable getTable() {
       return table;
   }
   /**
    * Metoda care elibereaza casetele pentru reintroducerea altor date
    */

   public void clear() {
       id_textField.setText("");
       name_textField.setText("");
       price_textField.setText("");
       stoc_textField.setText("");
```

```
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de ADD
     * @param actionListener
     */
    public void addProductListener(ActionListener actionListener) {
        this.addProduct.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de DELETE
     * @param actionListener
     */
    public void deleteProductListener(ActionListener actionListener) {
        this.deleteProduct.addActionListener(actionListener);
    }
    /**
     * Metoda carea adauga actionListener pentru butonul de UPDATE
     * @param actionListener
     */

    public void updateProductListener(ActionListener actionListener) {
        this.updateProduct.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de SHOW TABLE
     * @param actionListener
     */
    public void showProductTableListener(ActionListener actionListener) {
        this.showTableProduct.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de BACK
     * @param actionListener
     */
    public void backListener(ActionListener actionListener) {
        this.backButton.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de CLEAR
     * @param actionListener
     */
    public void clearListener(ActionListener actionListener) {
        this.clear.addActionListener(actionListener);
    }
```

## 4.5 Clasa OrderView

- Clasa care se ocupa de realizarea interfetei pentru operatii cu comenzi
- Are ca si atribute

- contentPane: JContentPane
- id_label, clientId_label, productId_label, quantity_label, title: JLabel
- id_textField, clientId_textField, productId_textField, quantity_textField: JText_Field
- table: JTabel
- addOrder, deleteOrder, showOrderTable, back, clear: JButton
- jScrollPane: JScrollPane

➢ Implementeaza metodele:

```
➢  /**
    * Getter ul pentru id-ul comenzii introdus de utilizator
    * @return id-ul comenzii introdus de utilizator
    */

   public int getIdOrder() {
       return Integer.parseInt(id_textField.getText());
   }

   /**
    * Getter ul pentru id-ul produsului introdus de utilizator
    * @return id-ul produsului introdus de utilizator
    */
   public int getProductId() {
       return (int) productId_combo.getSelectedItem();
   }

   /**
    * Getter ul pentru id-ul clientului introdus de utilizator
    * @return id-ul clientului introdus de utilizator
    */

   public int getClientId() {
       return (int) clientId_combo.getSelectedItem();
   }
   /**
    * Getter ul pentru tabel
    * @return tabel
    */

   public JTable getTable() {
       return table;
   }

   /**
    * Metoda care elibereaza casetele pentru reintroducerea altor date
    */

   public void clear() {
       id_textField.setText("");
       quantity_textField.setText("");
   }

   /**
    * Metoda prin care sunt adaugate in comboBox valorile id-ului din
   tabelul client
    */
```

```
public void comboIdClient() {
    try {
        Connection connection = ConnectionFactory.getConnection();
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery("SELECT id FROM client");
        while (rs.next()) {
            clientId_combo.addItem(rs.getInt(1));
        }
        connection.close();
    } catch (SQLException e) {
        JOptionPane.showMessageDialog(null, "Couldn't connect to db");
    }
}

/**
 * Metoda prin care sunt adaugate in comboBox valorile id-ului din
tabelul product
 */
public void comboIdProduct() {
    try {
        Connection connection = ConnectionFactory.getConnection();
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery("SELECT id FROM
product");
        while (rs.next()) {
            System.out.println(rs.getInt(1));
            productId_combo.addItem(rs.getInt(1));
        }
        connection.close();
    } catch (SQLException e) {
        JOptionPane.showMessageDialog(null, "Couldn't connect to db");
    }
}

/**
 * Getter ul pentru quantity introdusa de utilizator
 * @return quantity introdusa de utilizator
 */
public int getQuantity() {
    return Integer.parseInt(quantity_textField.getText());
}

/**
 * Metoda carea adauga actionListener pentru butonul de ADD
 * @param actionListener
 */
public void addOrderListener(ActionListener actionListener) {
    this.addOrder.addActionListener(actionListener);
}

/**
 * Metoda carea adauga actionListener pentru butonul de DELETE
 * @param actionListener
 */

public void deleteOrderListener(ActionListener actionListener) {
```

```
        this.deleteOrder.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de SHOW TABLE
     * @param actionListener
     */
    public void showTableOrderListener(ActionListener actionListener) {
        this.showOrderTable.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de BACK
     * @param actionListener
     */
    public void backListener(ActionListener actionListener) {
        this.back.addActionListener(actionListener);
    }

    /**
     * Metoda carea adauga actionListener pentru butonul de CLEAR
     * @param actionListener
     */
    public void clearListener(ActionListener actionListener) {
        this.clear.addActionListener(actionListener);
    }
```

## 4.6 Clasa BillView

- Clasa care se ocupa de realizarea interfetei pentru facturi
- Are ca si atribute:
  - contentPane: JPanel
  - table: JTabel
  - showBillTable, backButton: JButton
  - jScrollPane: jScrollPane
- Are ca si metode:

```
    /**
     * Metoda care adauga actionListener pentru butonul SHOW TABLE
     * @param actionListener
     */
    public void showTable(ActionListener actionListener){
        this.showBillTable.addActionListener(actionListener);
    }

    /**
     * Metoda care adauga actionListener pentru butonul BACK
     * @param actionListener
     */
    public void backListener(ActionListener actionListener){
        this.backButton.addActionListener(actionListener);
    }

    /**
```

```
     * Getter ul pentru tabel
     * @return tabel
     */
    public JTable getTable() {
        return table;
    }
```

## 4.7 Clasa Controller

- Clasa care controleaza intreaga aplicatie. Aici sunt implementate rolurile butoanelor.
- Are ca si atribute:
  - clientView: ClientView
  - productView: ProductView
  - orderView: OrderView
  - billView: BillView
- In constructorul clasei sunt implementate rolurile fiecaror butoane
- Implementeaza metoda care foloseste tehnica Reflection pentru afisarea in tabel a elementelor din baza de date:

```
➢   private void TableUsingReflection(List<Object> list, JTable table) {
        DefaultTableModel tableModel = new DefaultTableModel();
        if (list.size() != 0) {
            Object obj = list.get(0);
            for (Field field : obj.getClass().getDeclaredFields()) {
                field.setAccessible(true);
                try {
                    tableModel.addColumn(field.getName());
                } catch (IllegalArgumentException ex) {
                    ex.printStackTrace();
                }
            }
            for (Object o : list) {
                List<Object> attr = new ArrayList<>();
                for (Field field : o.getClass().getDeclaredFields()) {
                    field.setAccessible(true);
                    Object value;
                    try {
                        value = field.get(o);
                        attr.add(value);
                    } catch (IllegalArgumentException |
    IllegalAccessException ex) {
                        ex.printStackTrace();
                    }
                }
                tableModel.addRow(attr.toArray());
            }
            table.setModel(tableModel);
        } else {
            JOptionPane.showMessageDialog(null, "tabelul este gol!");
            table.setModel(tableModel);
        }
    }
```

## 4.8 Clasa ConnectionFactory

- Clasa care se ocupa de conectarea la baza de date
- Are ca si atribute :
  - LOGGER = Logger.getLogger(ConnectionFactory.class.getName());
  - DRIVER = "com.mysql.cj.jdbc.Driver";
  - DBURL="jdbc:mysql://localhost:3306/databasetp";
  - USER="root";
  - PASS="root";
- Implementeaza metodele:

```
/**
 * Metoda care creeaza conexiunea
 * @return conexiunea
 */
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;
}

/**
 * Getter ul pentru conexiune
 * @return conexiunea
 */
public static Connection getConnection() {
    return singleInstance.createConnection();
}

/**
 * Metoda prin care este inchisa conexiunea
 * @param connection obiectul de tip conexiune
 */
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");
        }
    }
}

/**
 * Metoda care inchide statement ul
 * @param statement obiectul de tip statement
```

```
     */
    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");
            }
        }
    }

    /**
     * Metoda care inchide resulSet
     * @param resultSet obiectul de tip resultSet
     */
    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");
            }
        }
    }
```

### 4.9 Clasa Client

- Clasa care modeleaza entitatea client si are aceleasi campuri ca si tabelul din baza de date.
- Are ca si atribute :
  - Id, email, firstName, lastName

### 4.10 Clasa Product

- Clasa care modeleaza entitatea product si are aceleasi campuri ca si tabelul din baza de date.
- Are ca si atribute:
  - Id, name, price, stoc;

### 4.11 Clasa Order

- Clasa care modeleaza entitatea client si are aceleasi campuri ca si tabelul din baza de date.
- Are ca si atribute :
  - OrderId, clientId, productId, quantity

### 4.12 Clasa Bill

- Clasa immutable Bill care modeleaza entitatea bill

```
➢   public record Bill(int billId, int orderId, int clientId, int
    productId, int quantity) {

    }
```

### 4.13 Clasa AbstractDAO

- Clasa generica DAO pentru interactiunea directa cu baza de date
- Are ca si atribute :

- LOGGER = Logger.getLogger(AbstractDAO.class.getName());
- Type: Class<T>

➢ Implementeaza metodele:

```
➢  /**
    * Metoda care creeaza interogarea pentru inserarea unui obiect
    * @param t
    * @return string ul specific interogarii de inserare
    * @throws IllegalAccessException
    */
   private String insertQuery(T t) throws IllegalAccessException {
       StringBuilder sb = new StringBuilder();
       sb.append("INSERT ");
       sb.append(" INTO ");
       sb.append(type.getSimpleName());
       sb.append(" VALUES (");

       for(Field field : t.getClass().getDeclaredFields()) {
           field.setAccessible(true);
           if(field.get(t) instanceof Integer) {
               sb.append(field.get(t));
               sb.append(",");
           }
           else {
               sb.append("'");
               sb.append(field.get(t));
               sb.append("',");
           }
       }
       sb.deleteCharAt(sb.length()-1);
       sb.append(");");
       System.out.println(sb.toString());
       return sb.toString();
   }


   /**
    * Metoda care inserareaza un obiect in baza de date folosind
   interogarea creata in metoda insertQuery
    * @param t
    */

   public void insert(T t) {

       Connection connection = null;
       PreparedStatement statement = null;

       try {
           connection = ConnectionFactory.getConnection();
           String query=insertQuery(t);
           statement = connection.prepareStatement(query);
           statement.executeUpdate();

       }catch(SQLException | IllegalAccessException e){
           e.printStackTrace();
       }finally {
```

```
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }
    }

    /**
     * Metoda care creeaza interogarea pentru modificarea unui obiect
     * @param t
     * @param fields
     * @param id
     * @return
     * @throws IllegalAccessException
     */

    private String updateQuery(T t,String[] fields, String id) throws
    IllegalAccessException{
        StringBuilder sb = new StringBuilder();
        sb.append("UPDATE ");
        sb.append(type.getSimpleName());
        sb.append(" SET ");
        StringBuilder s= new StringBuilder();
        s.append("");
        int i=0 ;
        for(Field field : t.getClass().getDeclaredFields()) {
            field.setAccessible(true);

            if(field.get(t) instanceof Integer) {

                sb.append(" "+fields[i]+" = "+field.get(t));
                if(i==0)
                    s.append(field.get(t));
                sb.append(",");
                i++;
            }
            else {
                sb.append(fields[i]+" = '"+field.get(t));
                sb.append("',");
                i++;
            }
        }
        sb.deleteCharAt(sb.length()-1);
        sb.append(" WHERE "+id+" = " + s);
        System.out.println(sb.toString());
        return sb.toString();

    }


    /**
     * Metoda care modifica un obiect din baza de date
     * @param t
     * @param field
     * @return
     */
    public T update(T t,String field) {
        Connection connection = null;
        PreparedStatement statement = null;
```

```
        try {
            Field[] fields=type.getDeclaredFields();
            String[] fieldNames=new String[fields.length];
            for(int i=0;i<fields.length;i++){
                fieldNames[i]=fields[i].getName();
            }
            String query=updateQuery(t,fieldNames,field);
            connection = ConnectionFactory.getConnection();
            statement = connection.prepareStatement(query);

            statement.executeUpdate();
        } catch (SQLException|IllegalAccessException e) {
            e.printStackTrace();
        }
        finally {
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }
        return t;
    }

    /**
     * Metoda care creeaza interogarea pentru stergerea unui obiect
     * @param field
     * @return
     */
    private String createDeleteQuery(String field) {
        StringBuilder sb = new StringBuilder();
        sb.append("DELETE ");
        sb.append(" FROM ");
        sb.append(type.getSimpleName());
        sb.append(" WHERE "+ field + " = " + " ?");
        return sb.toString();
    }

    /**
     * Metoda care sterge un obiect din baza de date
     * @param id
     * @param field
     */
    public void delete(int id, String field){
        Connection connection = null;
        PreparedStatement statement = null;
        String query = createDeleteQuery(field);
        try {
            System.out.println(query);
            connection = ConnectionFactory.getConnection();
            statement = connection.prepareStatement(query);
            statement.setInt(1, id);
            statement.executeUpdate();

        }
        catch (SQLException e) {
            LOGGER.log(Level.WARNING, type.getName()+"Dao:findById" +
    e.getMessage());
        } finally {
            ConnectionFactory.close(statement);
```

```
            ConnectionFactory.close(connection);
        }

    }

    /**
     * Metoda prin care se face fetch obiectelor
     * @param resultSet
     * @return
     */


    private List<T> createObjects(ResultSet resultSet) {
        List<T> list = new ArrayList<T>();
        Constructor[] ctors = type.getDeclaredConstructors();
        Constructor ctor = null;
        for (int i = 0; i < ctors.length; i++) {
            ctor = ctors[i];
            if (ctor.getGenericParameterTypes().length == 0)
                break;
        }
        try {
            while (resultSet.next()) {
                ctor.setAccessible(true);
                T instance = (T)ctor.newInstance();
                for (Field field : type.getDeclaredFields()) {
                    String fieldName = field.getName();
                    Object value = resultSet.getObject(fieldName);
                    PropertyDescriptor propertyDescriptor = new
    PropertyDescriptor(fieldName, type);
                    Method method = propertyDescriptor.getWriteMethod();
                    method.invoke(instance, value);
                }
                list.add(instance);
            }
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } catch (IntrospectionException e) {
            e.printStackTrace();
        }
        return list;
    }
```

## 4.14 Clasa ClientDAO

- Clasa care extinda AbstractDAO si permite crearea obiectului ClientDAO
- Implementeaza metodele

```java
/**
 * Metoda prin care sunt extrasi toti clientii din baza de date
 * @return lista de clienti
 */
public List<Client> findAll() {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    String query = "SELECT * FROM client";
    List<Client> lista = new ArrayList<>();
    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        resultSet = statement.executeQuery();
        while (resultSet.next()) {
            int id = resultSet.getInt(1);
            String email = resultSet.getString(2);
            String firstName = resultSet.getString(3);
            String lastName = resultSet.getString(4);
            lista.add(new Client(id, email, firstName,lastName)); //
adaugare client in lista de clienti
        }
    } catch (SQLException e) {

        e.printStackTrace();
    } finally {
        ConnectionFactory.close(resultSet);
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }

    return lista;
}

/**
 * Metoda prin care este cautat un client dupa id ul lui
 * @param id
 * @return clientul cu id ul cautat
 */
public Client findById(int id) {

    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    String query = "SELECT * FROM client WHERE id = ?";

    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.setInt(1, id);
        resultSet = statement.executeQuery();

        while (resultSet.next()) {

            String email = resultSet.getString(2);
            String firstName = resultSet.getString(3);
            String lastName = resultSet.getString(4);
```

```
                return new Client(id, email, firstName,lastName);
            }

        } catch (SQLException e) {

            e.printStackTrace();
        } finally {
            ConnectionFactory.close(resultSet);
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }

        return null;
    }
```

## 4.15 Clasa ProductDAO

- Clasa care extinda AbstractDAO si permite crearea obiectului ProductDAO
- Implementeaza metodele:

```
    /**
     * Metoda prin care sunt extrase toate produsele din baza de date
     * @return lista de produse
     */
    public List<Product> findAll() {
        Connection connection = null;
        PreparedStatement statement = null;
        ResultSet resultSet = null;
        String query = "SELECT * FROM product";
        List<Product> lista = new ArrayList<>();
        try {
            connection = ConnectionFactory.getConnection();
            statement = connection.prepareStatement(query);
            resultSet = statement.executeQuery();
            while (resultSet.next()) {
                int id = resultSet.getInt(1);
                String name = resultSet.getString(2);
                float price = resultSet.getFloat(3);
                int stoc = resultSet.getInt(4);
                lista.add(new Product(id, name, price,stoc));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            ConnectionFactory.close(resultSet);
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }

        return lista;
    }

    /**
     * Metoda prin care este cautat un produs dupa id ul lui
     * @param id
     * @return produsul cu id ul cautat
```

```
     */
    public Product findById(int id) {

        Connection connection = null;
        PreparedStatement statement = null;
        ResultSet resultSet = null;
        String query = "SELECT * FROM product WHERE id = ?";

        try {
            connection = ConnectionFactory.getConnection();
            statement = connection.prepareStatement(query);
            statement.setInt(1, id);
            resultSet = statement.executeQuery();

            while (resultSet.next()) {

                String name = resultSet.getString(2);
                float price = resultSet.getFloat(3);
                int stoc = resultSet.getInt(4);


                return new Product(id, name, price,stoc);
            }

        } catch (SQLException e) {

            e.printStackTrace();
        } finally {
            ConnectionFactory.close(resultSet);
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }

        return null;
    }
```

## 4.16 Clasa OrderDAO

- Clasa care extinda AbstractDAO si permite crearea obiectului OrderDAO
- Implementeaza metodele:

```
    /**
     * Metoda prin care sunt extrase toate comenzile din baza de date
     * @return lista de comenzi
     */
    public List<Order> findAll() {
        Connection connection = null;
        PreparedStatement statement = null;
        ResultSet resultSet = null;
        String query = "SELECT * FROM databasetp.order";
        List<Order> lista = new ArrayList<>();
        try {
            connection = ConnectionFactory.getConnection();
            statement = connection.prepareStatement(query);
            resultSet = statement.executeQuery();
            while (resultSet.next()) {
                int orderId = resultSet.getInt(1);
                int clientId = resultSet.getInt(2);
```

```
            int productId = resultSet.getInt(3);
            int quantity = resultSet.getInt(4);
            lista.add(new Order(orderId, clientId,
productId,quantity));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        ConnectionFactory.close(resultSet);
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }

    return lista;
}

/**
 * Metoda prin care este cautata o comanda dupa id ul ei
 * @param id
 * @return comanda cu id ul cautat
 */

public Order findById(int id) {

    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    String query = "SELECT * FROM databasetp.order WHERE orderId = ?";

    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.setInt(1, id);
        resultSet = statement.executeQuery();

        while (resultSet.next()) {
            int clientId = resultSet.getInt(2);
            int productId = resultSet.getInt(3);
            int quantity = resultSet.getInt(4);
            return new Order(id, clientId, productId,quantity);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        ConnectionFactory.close(resultSet);
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }
    return null;
}

/**
 * Metoda prin care este creata interogarea de inserare a unui obiect
in baza de date
 * @param order
 * @return
 * @throws IllegalAccessException
```

```
     */
    private String insertQuery(Order order) throws IllegalAccessException {
        StringBuilder sb = new StringBuilder();
        sb.append("INSERT ");
        sb.append(" INTO ");
        sb.append("databasetp.order");
        sb.append(" VALUES (");

        for(Field field : order.getClass().getDeclaredFields()) {
            field.setAccessible(true);
            if(field.get(order) instanceof Integer) {
                sb.append(field.get(order));
                sb.append(",");
            }
            else {
                sb.append("'");
                sb.append(field.get(order));
                sb.append("',");
            }
        }
        sb.deleteCharAt(sb.length()-1);
        sb.append(");");
        System.out.println(sb.toString());
        return sb.toString();
    }

    /**
     * Metoda prin care se adauga o comanda in baza de date
     * @param order
     */
    public void insert(Order order) {

        Connection connection = null;
        PreparedStatement statement = null;

        try {
            connection = ConnectionFactory.getConnection();
            String query=insertQuery(order);
            statement = connection.prepareStatement(query);
            statement.executeUpdate();

        }catch(SQLException e){
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            throw new RuntimeException(e);
        } finally {
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }
    }

    /**
     * Metoda prin care este creata interogarea de stergere a unei comenzi
    din baza de date
     * @param field
     * @return
     */
```

```
    private String createDeleteQuery(String field) {
        StringBuilder sb = new StringBuilder();
        sb.append("DELETE ");
        sb.append(" FROM ");
        sb.append(" databasetp.order ");
        sb.append(" WHERE "+ field + " = " + " ?");
        return sb.toString();
    }

    /**
     * Metoda prin care se sterge un obiect din baza de date
     * @param id
     * @param field
     */

    public void delete(int id, String field){
        Connection connection = null;
        PreparedStatement statement = null;
        String query = createDeleteQuery(field);
        try {
            System.out.println(query);
            connection = ConnectionFactory.getConnection();
            statement = connection.prepareStatement(query);
            statement.setInt(1, id);
            statement.executeUpdate();

        }
        catch (SQLException e) {
            LOGGER.log(Level.WARNING, "Order"+"Dao:findById" +
    e.getMessage());
        } finally {
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }

    }
```

## 4.17 Clasa BillDAO

- Clasa care extinda AbstractDAO si permite crearea obiectului BillDAO
- Implementeaza metoda:

```
    /**
     * Metoda prin care sunt extrase toate facturile din baza de date
     * @return lista de facturi
     */
    public List<Bill> findAll() {
        Connection connection = null;
        PreparedStatement statement = null;
        ResultSet resultSet = null;
        String query = "SELECT * FROM bill";
        List<Bill> lista = new ArrayList<>();
        try {
            connection = ConnectionFactory.getConnection();
            statement = connection.prepareStatement(query);
            resultSet = statement.executeQuery();
            while (resultSet.next()) {
                int billId = resultSet.getInt(1);
```

```
            int orderId = resultSet.getInt(2);
            int clientId = resultSet.getInt(3);
            int productId = resultSet.getInt(4);
            int quantity = resultSet.getInt(5);
            lista.add(new Bill(billId, orderId,
clientId,productId,quantity));
        }
    } catch (SQLException e) {

        e.printStackTrace();
    } finally {
        ConnectionFactory.close(resultSet);
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }

    return lista;
}
```

## 4.18 Clasa ClientBLL

- Clasa care se ocupa de logica aplicatiei pentru modelul Client
- Are ca si atribute :
  - clientDAO : ClientDAO
  - validators : List<Validator<Client >>
- Implementeaza metodele :

```
/**
 * Metoda apeleaza DAO pentru gasirea unui client dupa id
 * @param id
 * @return clientul cu id ul cautat
 * @throws Exception
 */
public static Client findClientById(int id) throws Exception {
    Client cl = clientDAO.findById(id);
    if (cl == null) {
        throw new Exception("Clientul cu id =" + id + " nu a fost gasit
!");
    }
    return cl;
}

/**
 * Metoda apeleaza DAO pentru inserarea unui client in baza de date
 * @param client
 */
public static void insertClient(Client client){
    try {
        for (Validator<Client> validator : validators) {
            validator.validate(client);
        }
        clientDAO.insert(client);
    }catch(Exception e){
        JOptionPane.showMessageDialog(null,"email invalid!");
    }

}
```

```
    /**
     * Metoda apeleaza DAO pentru a gasi toate inregistrarile din tabelul
    Client
     * @return lista de clienti
     */
    public static List<Client> findClientAll(){
        return clientDAO.findAll();
    }

    /**
     * Metoda apeleaza DAO pentru stergerea unui client din baza de date
    dupa id
     * @param id
     */
    public static void deleteClientById(int id){
        clientDAO.delete(id,"id");
    }

    /**
     * Metoda apeleaza DAO pentru modificarea unui client din baza de date
    dupa id
     * @param client
     */
    public static void updateClientById(Client client ){
        clientDAO.update(client,"id");
    }
```

## 4.19 Clasa ProductBLL

- Clasa care se ocupa de logica aplicatiei pentru modelul Product
- Are ca si atribute :
  - productDAO : ProductDAO
- Implementeaza metodele :

```
    /**
     * Metoda care apeleaza DAO pentru gasirea unui produs dupa id
     * @param id
     * @return produsul cu id ul cautat
     * @throws Exception
     */
    public static Product findProductById(int id) throws Exception {
        Product pr = productDAO.findById(id);
        if (pr == null) {
            throw new Exception("Produsul cu id =" + id + " nu a fost gasit
    !");
        }
        return pr;
    }

    /**
     * Metoda apeleaza DAO pentru inserarea unui produs in baza de date
     * @param product
     */

    public static void insertProduct(Product product){
        productDAO.insert(product);
```

```
    }

    /**
     * Metoda apeleaza DAO pentru a gasi toate inregistrarile din tabelul
    Product
     * @return lista de produse
     */
    public static List<Product> findProductAll(){
        return productDAO.findAll();
    }

    /**
     * Metoda apeleaza DAO pentru a sterge un produs dupa id
     * @param id
     */
    public static void deleteProductById(int id){
        productDAO.delete(id,"id");
    }

    /**
     * Metoda apeleaza DAO pentru modificarea unui produs dupa id
     * @param product
     */
    public static void updateProductById(Product product ){
        productDAO.update(product,"id");
    }
```

## 4.20 Clasa OrderBLL

- Clasa care se ocupa de logica aplicatiei pentru modelul Order
- Are ca si atribute :
  - orderDAO : OrderDAO
  - validators : List<Validator<Order >>
- Implementeaza metodele :

```
    /**
     * Metoda apeleaza DAO pentru a gasi toate inregistrarile din tabelul
    Order
     * @return lista de comenzi
     */
    public static List<Order> findOrdersAll(){
        return orderDAO.findAll();
    }

    /**
     * Metoda apeleaza DAO pentru a gasi o comanda dupa id
     * @param id
     * @return comanda cu id ul cautat
     * @return comanda cu id ul cautat
     * @throws Exception
     */

    public static Order findOrderById(int id) throws Exception {
        Order or = orderDAO.findById(id);
        if (or == null) {
            throw new Exception("Comanda cu id =" + id + " nu a fost gasita
    !");
```

```
        }
        return or;
    }

    /**
     * Metoda care apeleaza DAO pentru inserarea unei comenzi in baza de
    date
     * @param or
     * @throws Exception
     */
    public static void insertOrders(Order or) throws Exception {
        Product product = ProductBLL.findProductById(or.getProductId());
        if(product == null)
            return;
        Client client = ClientBLL.findClientById(or.getClientId());
        if(client==null)
            return;
        if(product.getStoc()<or.getQuantity()) {
            JOptionPane.showMessageDialog(null, "Nu exista suficiente
    produse in stoc");
            return;
        }
        else
            product.setStoc(product.getStoc()-or.getQuantity());
        createBill(or);
        ProductBLL.updateProductById(product);

        try{
            for(Validator<Order> validator : validators) {
                validator.validate(or);
            }
            orderDAO.insert(or);
        }catch(Exception e){
            JOptionPane.showMessageDialog(null,e.getMessage());
        }

    }

    /**
     * Metoda care apeleaza DAO pentru stergerea unei comenzi din baza de
    date
     * @param or
     * @throws Exception
     */
    public static void deleteOrder(Order or) throws Exception {
        Product product = ProductBLL.findProductById(or.getProductId());
        if(product == null)
            return;
        product.setStoc(product.getStoc()+or.getQuantity());
        ProductBLL.updateProductById(product);
        Bill bill = new Bill(or.getOrderId(),or.getOrderId(),
    or.getClientId(), or.getProductId(), or.getQuantity());
        BillBLL.deleteBill(bill);
        orderDAO.delete(or.getOrderId(),"orderId");
    }

    /**
```

```
     * Metoda care apeleaza DAO pentru crearea unei facturi si adaugarea
    acesteia in baza de date
     * @param orders
     */
    public static void createBill( Order orders)  {
        Bill bill = new Bill(orders.getOrderId(),orders.getOrderId(),
    orders.getClientId(), orders.getProductId(), orders.getQuantity());
        BillBLL.insertBill(bill);
    }
```

## 4.21 Clasa BillBLL

- Clasa care se ocupa de logica aplicatiei pentru modelul Bill
- Are ca si atribute :
  - billDAO : BillDAO
- Implementeaza metodele :

```
    /**
     * Metoda apeleaza DAO pentru inserarea unei facturi in baza de date
     * @param bill
     */
    public static void insertBill(Bill bill){
        billDAO.insert(bill);
    }

    /**
     * Metoda apeleaza DAO pentru stergerea unei facturi din baza de date
     * @param bill
     */
    public static void deleteBill(Bill bill){
        billDAO.delete(bill.billId(), "billId");
    }

    /**
     * Metoda apeleaza DAO pentru a gasi toate inregistrarile din tabelul
    bill
     * @return
     */
    public static List<Bill> showAllBils(){
        return billDAO.findAll();
    }
```

## 4.22 Clasa EmailValidator

- Clasa care implementeaza metoda din interfata Validator pentru validarea email-ului la operatia de insert client
- Implementeaza metoda:

```
    public void validate(Client client) {
        Pattern pattern = Pattern.compile(EMAIL_PATTERN);
        if (!pattern.matcher(client.getEmail()).matches()) {
            throw new IllegalArgumentException("Email is not a valid
    email!");
        }

    }
```

  - 

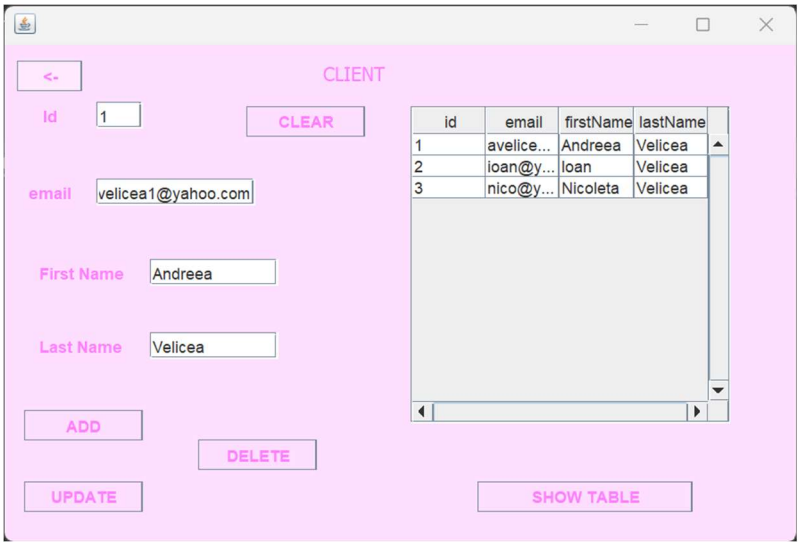
### 4.23 Clasa QuantityValidator

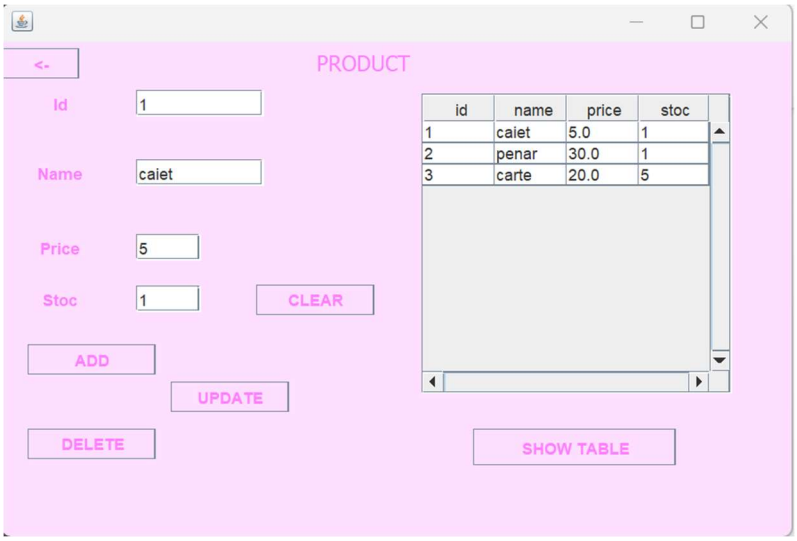
- Clasa care implementeaza metoda din interfata Validator pentru validarea cantitatii la operatia de insert order
- Implementeaza metoda:

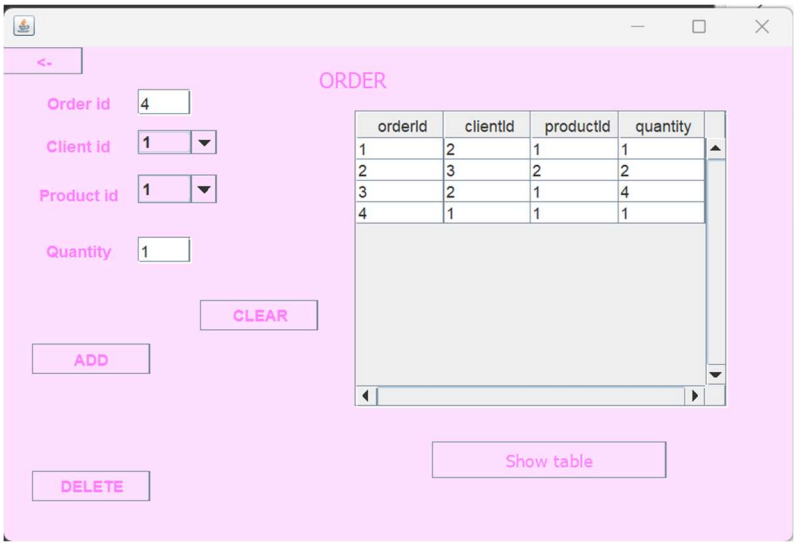
```
public void validate(Order or)
{
    if(or.getQuantity() < MIN_Q || or.getQuantity()> MAX_Q)
    {
        throw new IllegalArgumentException("The quantity is not
respected!");
    }
}
```

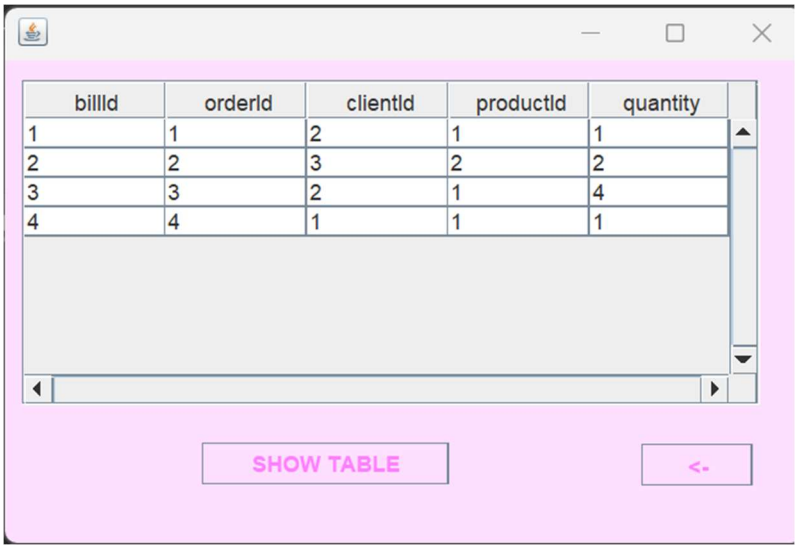
## 5. Rezultate

In urma implementarii claselor, pentru testarea si verificarea aplicatiei s-au introdus cate un produs mai apoi un client, pentru verificarea corectitudinii s-au si listat, iar numai mai apoi s-a creat o comanda cu ajutorul clientului si a produsului creat iar pentru fiecare a fost afisata mai jos cate o imagine. Fiecare comanda se adauga intr un tabel afisat in ViewBill




<-
PRODUCT
Id
1
Name
caiet
Price
5
Stoc
1
CLEAR
ADD
UPDATE
DELETE
SHOW TABLE
id name price stoc
1 caiet 5.0 1
2 penar 30.0 1
3 carte 20.0 5


<-
ORDER
Order id
4
Client id
1
Product id
1
Quantity
1
CLEAR
ADD
DELETE
Show table
orderId clientId productId quantity
1 2 1 1
2 3 2 2
3 2 1 4
4 1 1 1


billId orderId clientId productId quantity
1 1 2 1 1
2 2 3 2 2
3 3 2 1 4
4 4 1 1 1
SHOW TABLE
<-

# 6. Concluzii

Din aceasta tema am invatat cum sa conectez baza de date la o aplicatie java si cum sa stochez si sa preiau elemente din baza de date.

O actualizare ulterioara a acestui proiect este, afisarea facturii ca un pdf unde sunt precizate toate detaliile despre comanda, clientul care a efectuat comanda si produsele comandate

# 7. Bibliografie


- https://dsrl.eu/courses/pt/materials/PT2023_A3_S1.pdf
- https://dsrl.eu/courses/pt/materials/PT2023_A3_S2.pdf
- https://gitlab.com/utcn_dsrl/pt-layered-architecture