Universitatea Tehnică din Cluj-Napoca
Facultatea de Automatică și Calculatoare

# DOCUMENTAȚIE

TEMA NUMĂRUL 3

Nume student: Maria-Magdalena Creț

Grupa 30223

# Cuprins

# 1 Cerința temei

Cerința acestei teme este următoarea:

Proiectați și implementați o aplicație de Gestionare a Comenzilor pentru procesarea comenzilor clienților pentru un depozit. Baze de date relaționale ar trebui să fie utilizate pentru a stoca produsele, clienții și comenzile. Aplicația ar trebui să fie proiectată conform modelului arhitectural pe straturi și ar trebui să utilizeze (minim) următoarele clase:

- Clase de Model - reprezintă modelele de date ale aplicației
- Clase de Logică de Afaceri - conțin logica aplicației
- Clase de Prezentare - clase legate de interfața grafică (GUI)
- Clase de Acces la Date - clase care conțin accesul la baza de date

Alte cerinte:

- Folosirea tehnicilor de reflecție
- Definirea unei clase imutabile Bill în pachetul Model folosind recorduri Java. Un obiect Bill va fi generat pentru fiecare comandă și va fi stocat într-un tabel Log. Facturile pot fi doar inserate și citite din tabelul Log; nu sunt permise actualizările
- Uitilizarea javadoc, documentarea claselor și metodelor realizate.

# 2 Obiectivele temei

- Utilizarea bazelor de date relaționale;
- Proiectarea plicației conform modelului arhitectural pe straturi:
  - Model (Model classes);
  - Logică de Afaceri (Business Logic classes);
  - Prezentare (Presentation classes);
  - Acces la Date (Data access classes - DAO);
- Implementarea interfeței grafice există posibilitatea alegerii între Java Swing sau JavaFX;
- Importanța Modificabilității și Întreținerii Codului;
- Înțelegerea fluxului de lucru pentru gestionarea comenzilor într-un depozit;
- Layered Architecture.

# 3 Analiza problemei, modelare, scenarii, cazuri de utilizare

## 3.1 Analiza problemei

1. Cerințele funcționale:
   (a) utilizatorul poate să introducă/să elimine clienți, produse și să realizeze comenzi pentru un client, toate fiind stocate în baza de date;
   (b) Utilizatorul poate să genereze factura unei comenzi pentru un anumit client;
   (c) Actualizarea informațiilor despre clienți (nume, prenume, număr de telefon, email);
   (d) Vizualizarea unei liste cu toate produsele disponibile;
   (e) Vizualizarea unei liste cu toate comenzile unui client;
   (f) Mecanisme de navigare între diferitele secțiuni ale aplicației (produse, clienți, comenzi);
   (g) Calcularea automată a valorii totale a comenzilor pentru un client;
   (h) Realizarea operațiilor CRUD (Create, Read, Update, Delete) pentru toate entitățile;
   (i) Conectarea la baza de date relațională pentru a salva și a prelua date despre produse, clienți și comenzi;
   (j) Mesaje de eroare și confirmare pentru acțiuni reușite sau nereușite.
2. Cerințele non-funcționale:
   (a) Aplicația trebuie să fie intuitivă și ușor de utilizat;
   (b) Aplicația trebuie să fie capabilă să gestioneze creșterea volumului de date (produse, clienți, comenzi) și a numărului de utilizatori fără a suferi o scădere semnificativă a performanței;
   (c) Codul trebuie să fie bine documentat și organizat pentru a facilita întreținerea și actualizările viitoare. (Use javadoc for documenting classes and generate the corresponding JavaDoc files)

## 3.2 Modelarea problemei pentru scrierea codului Java

Așadar pentru a simplifica rezolvarea problemei și pentru a crea o structură cât mai ușor de analizat, unde se pot realiza într-un mod accesibil și rapid modificări, s-a ales organizarea codului Java în pachete(packages):

- connection: integrează clasa ConnectionFactory care conțin implementarea codului pentru conectarea cu baza de date.
- controller: integrează clasele de tipul Controller pentru a lega logica aplicației de UI și a realiza o aplicație ușor de utilizat și funcțională.

- model:
  - Client, această clasă reține informațiile despre clienți;
  - Product , această clasă reține informațiile despre produse;
  - WareHouseOrder , această clasă reține informațiile despre compenzi;
- dao: pentru clasele de tipul Data Access Object. Data Access Object este un design pattern în Java (și în alte limbaje de programare) utilizat pentru a separa logica de acces la date de restul aplicației. Acesta este folosit pentru a ascunde detaliile implementării persistentei datelor și pentru a oferi o interfață simplă și coerentă pentru operațiunile de acces la date. Principiile sunt - încapsularea detaliilor de acces la date, separarea logicii de accesul la date și faptul că DAO poate fi schimbat sau actualizat fără a afecta restul aplicației, făcând posibilă migrarea între diferite baze de date sau metode de stocare a datelor.
- gui: conține clasele pentru fiecare pagină a UI-lui, interfața a fost realizată în swing GUI. Totodată gui conține un alt pachet numit *util*, care are la rândul lui clase pentru butoane și tabele.
- validate: conține o clasă Validator pentru verificarea anumitor condiții pentru field-uri și afișarea mesajelor de eroare atunci când este cazul: s-au utilizat regex-uri pentru validarea adresei de email, a numărului de telefon și a field-urilor pentru nume, respectiv prenume care trebuie să conțină doar litere mari sau mici.

```java
/**
 * The Validator class provides methods for validating user input in the user interface.
 *  This class includes various static methods to check for common validation requirements.
 *
 */
public class Validator {

    1 usage
    private static final String EMAIL_PATTERN = "(?:(?:\\r\\n)?[ \\t])*(?:(?:(?:[^()<>@,;:\\\\\".\\[\\] \\000-\\031]+
    1 usage
    private static final String LETTER_PATTERN = "^[a-zA-Z]+$";
    1 usage
    private static final String PHONE_PATTERN = "^\\d{10}$";

    /**
     * Validates if the given string that represents email is not null and not empty and respects the standards for a
     * @param mail represents input the string to validate
     * @return true if the string is valid, false otherwise
     */
    2 usages
    public static boolean validateMail(String mail) {
        Pattern pattern = Pattern.compile(EMAIL_PATTERN);
        if (!pattern.matcher(mail).matches()) {
            return false;
        }

        return true;
    }
```

Figura 1: Clasa Validator - parte din implementare

- view: conține clasa OrderView, care este o clasă care conține la rândul ei informații despre comenzi și produse și trimite aceste informații componentelor din UI

# 4 Proiectare

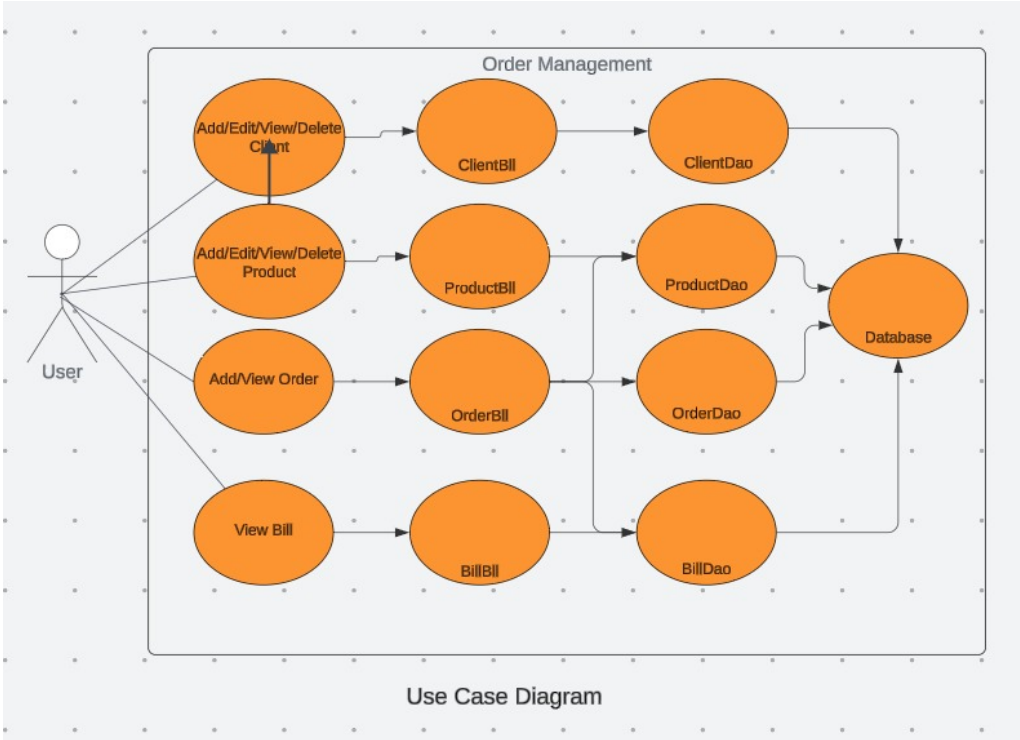


Figura 2: Diagrama Use-Case

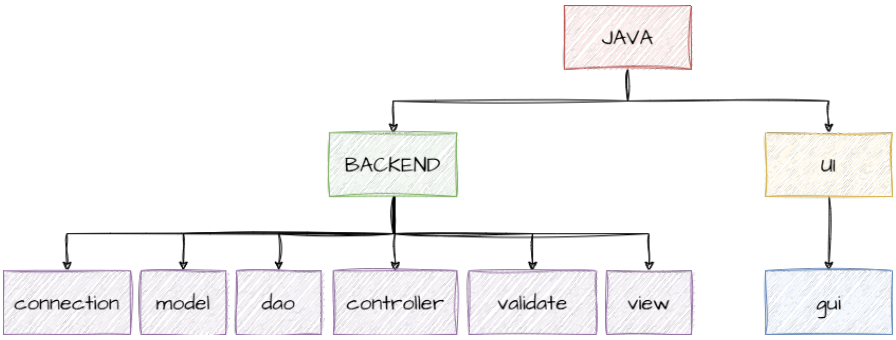


Figura 3: Diagrama Pachete

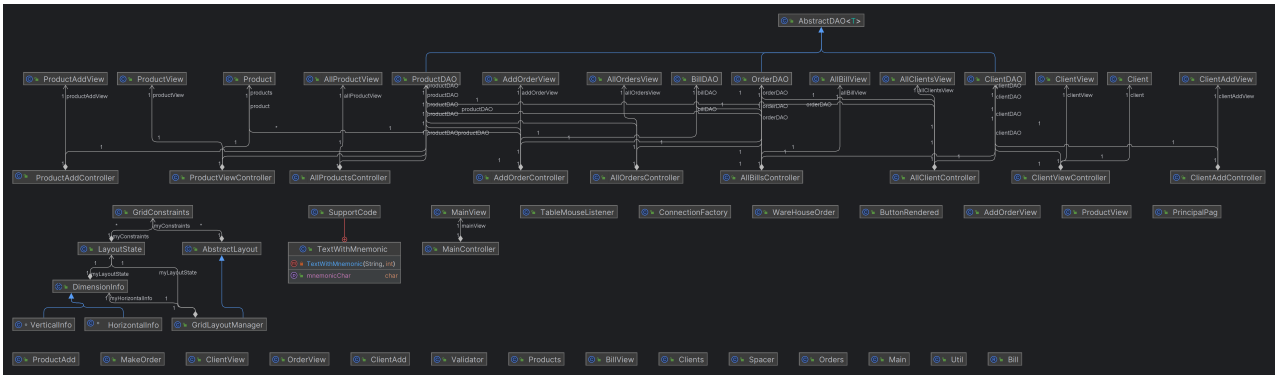

Figura 4: Diagrama UML pentru clase

# 5 Implementare

Implementarea codului s-a realizat structural, pe baza pachetelor menționate mai sus și pachetul de Swing UI Designer: *gui*, unde s-a realizat implementarea pentru interfața grafică. Clasele conținute de proiect în mare, sunt următoarele:

1. Clasa *ConnectionFactory*:conține metodele de conectarea cu baza de date SQL și realizează operațiile de deschidere și inchidere a bazei de date respective. Conține metoda *createConnection*, unde se crează o conexiune cu baza de date pe baza obiectului Logger, care înregistrează informații detaliate despre fiecare încercare de conectare la baza de date, inclusiv succesul sau eșecul acestor încercări. Tot în această clasă sunt metodele statice *close*, unde pe baza argumentului primit de metodă (Apare polimorfismul!) se alege ce exact se va inchide - conexiunea cu baza de date, setul de rezultate obținut în urma executării unei interogări SQL sau un obiect de tipul Statement.

```
1   private ConnectionFactory() {
2         try {
3             Class.forName(DRIVER);
4         } catch (ClassNotFoundException e) {
5             e.printStackTrace();
6         }
7     }
8
9     private Connection createConnection() {
10        Connection connection = null;
11        try {
12            connection = DriverManager.getConnection(DBURL, USER,
                  PASS);
13        } catch (SQLException e) {
14            LOGGER.log(Level.WARNING, "An error occured while
                 trying to connect to the database");
15            e.printStackTrace();
16        }
17        return connection;
18    }
19
20    public static Connection getConnection() {
21        return singleInstance.createConnection();
22    }
23
24    public static void close(Connection connection) {
25        if (connection != null) {
26            try {
27                connection.close();
28            } catch (SQLException e) {
29                LOGGER.log(Level.WARNING, "An error occured while
                      trying to close the connection");
30            }
31        }
32    }
```

În această clasă utilizează JDBC pentru conectarea cu baza de date, unde: DriverManager reprezintă gestionarea setului de drivere de baze de date și oferă o metodă de conectare la baza de date prin intermediul unui driver specific; Driver este o interfața care definește metodele pentru conectarea la baza de date, iar de menționat este ca fiecare bază de date are propriul driver JDBC care implementează această interfață; Connection: reprezintă o conexiune activă la o bază de date și permite crearea de Statement, care este la rândul lui utilizat pentru a executa interogări SQL simple și pentru a primi rezultatele acestora; ResultSet: reprezintă setul de rezultate obținut în urma executării unei interogări SQL și permite navigarea și preluarea datelor din rândurile returnate.

2. Clasele de tip *DAO* -*ClientDAO*,*ProductDAO*, *OrderDAO*- extind clasa de bază *AbstractDAO*, care implementează metodele CRUD(Create, Find All, Update, Delete) pentru entitățiile din baza de date, respectiv metoda *createObjects*, care ia rezultatul din query-ul din baza de date și crează un obiect de tipul java. Se utilizează reflection pentru a construi un obiect T. Se aruncă excepții în cazul în care nu se poate realiza acest lucru. Totodată existe metode by id, care găsesc entității sau șterg entități dupa id.

```
1      protected List<T> createObjects(ResultSet resultSet) {
2          List<T> list = new ArrayList<T>();
3          Constructor[] ctors = type.getDeclaredConstructors();
4          Constructor ctor = null;
5          for (int i = 0; i < ctors.length; i++) {
6              ctor = ctors[i];
7              if (ctor.getGenericParameterTypes().length == 0)
8                  break;
9          }
10         try {
11             while (resultSet.next()) {
12                 ctor.setAccessible(true);
13                 T instance = (T) ctor.newInstance();
14                 for (Field field : type.getDeclaredFields()) {
15                     String fieldName = field.getName();
16                     Object value = resultSet.getObject(fieldName)
                           ;
17                     PropertyDescriptor propertyDescriptor = new
                           PropertyDescriptor(fieldName, type);
18                     Method method = propertyDescriptor.
                           getWriteMethod();
19                     method.invoke(instance, value);
20                 }
21                 list.add(instance);
22             }
23         } catch (InstantiationException e) {
24             e.printStackTrace();
25         } catch (IllegalAccessException e) {
26             e.printStackTrace();
```

```
27          } catch (SecurityException e) {
28              e.printStackTrace();
29          } catch (IllegalArgumentException e) {
30              e.printStackTrace();
31          } catch (InvocationTargetException e) {
32              e.printStackTrace();
33          } catch (SQLException e) {
34              e.printStackTrace();
35          } catch (IntrospectionException e) {
36              e.printStackTrace();
37          }
38          return list;
39      }
```

3. Clasele de tip *Controller*: implementează metodele de conectarea a logicii aplicației cu din interfața și mai exact, atunci când se accesează un anumit buton se deschide o altă interfață specifică butonului care a fost accesat, cu funcționalitățiile corespunzătoare. Spre exemplu clasa *AllClientController* are rolul de a implementa logica pentru butonul *Clients* din prima pagină a UI-lui aplicației, care pe vaza metodei *actionPerformed* utilizatorul este trimis la pagina cu afișarea tuturor clienților. Metoda *showClients* ia toți clienții din baza de date și îi afișează.

```
1   public class AllClientController {
2
3       private AllClientsView allClientsView;
4
5       private ClientDAO clientDAO;
6
7       public AllClientController() {
8           allClientsView = new AllClientsView();
9           clientDAO = new ClientDAO();
10
11          allClientsView.getBackButton().addActionListener(new
                ActionListener() {
12              @Override
13              public void actionPerformed(ActionEvent e) {
14                  allClientsView.setVisible(false);
15                  allClientsView.dispose();
16                  MainController mainController = new
                        MainController();
17              }
18          });
19
20
21          allClientsView.getAddButton().addActionListener(new
                ActionListener() {
22              @Override
23              public void actionPerformed(ActionEvent e) {
24                  allClientsView.setVisible(false);
```

```
                allClientsView.dispose();

                ClientAddController clientAddController = new
                    ClientAddController();
            }
        });
        showClients();
    }
```

4. Clasa *Validator*: În acestă clasă se găsesc metodele de verificare a field-urilor din interfață. Există field-uri de nume și prenume pentru client care trebuie validate astfel încât să conțină doar litere mari sau mici, lucru verificat pe baza unui regex, astfel că dacă se aduagă cifre sau altfel de caractere se aruncă mesaje de eroare și nu se poate salva un client cu aceste date. La fel și în cazu adresei de email, care trebuie să fie validă și la fel și în cazul numărului de telefon care este valid doar dacă are 10 cifre și nu are altfel de caractere. Field-urile sunt verificate totodată dacă sunt goale sau nu, utilizatorul nu poate introduce un client sau un produs fără a completa toate field-urile.

```
public class Validator {

    private static final String EMAIL_PATTERN = "(?:(?:\\r\\n)?[
        \\t])*(?:(?:...";
    private static final String LETTER_PATTERN = "^[a-zA-Z]+$";
    private static final String PHONE_PATTERN = "^\\d{10}$";

    /**
     * Validates if the given string that represents email is not
         null and not empty and respects the standards for a
        correct email.
     * @param mail represents input the string to validate
     * @return true if the string is valid, false otherwise
     */
    public static boolean validateMail(String mail) {
        Pattern pattern = Pattern.compile(EMAIL_PATTERN);
        if (!pattern.matcher(mail).matches()) {
            return false;
        }

        return true;
    }

    /**
     * Validates if the text field has just letters
     * @param textField represents the input the text field to
        validate
     * @return true if the text field number is valid, false
        otherwise
     */
    public static boolean validateFieldLetter(JTextField
        textField) {
```

```
27          Pattern pattern = Pattern.compile(LETTER_PATTERN);
28          if (!pattern.matcher(textField.getText()).matches()) {
29              return false;
30          }
31        return true;
32      }
33
34      /**
35       * Validates if the text field for phone number respects the
             standards for phone numbers (exactly 10 digits)
36       * @param textField represents the input the text field to
             validate
37       * @return true if the phone number is valid, false otherwise
38       */
39      public static boolean validateFieldPhone(JTextField textField
           ) {
40          Pattern pattern = Pattern.compile(PHONE_PATTERN);
41          if (!pattern.matcher(textField.getText()).matches()) {
42              return false;
43          }
44          return true;
45      }
46
47      /**
48       * Validates if the given string is not null and not empty.
49       * @param textField input the string to validate
50       * @param errorMessage the error message to set if the
             validation fails
51       * @return true if the string is not null and not empty,
             false otherwise
52       */
53      public static int isTextFieldEmpty(JTextField textField,
           String errorMessage) {
54          if (textField.getText().isEmpty()) {
55              JOptionPane.showMessageDialog(null, errorMessage, "
                   ERROR", JOptionPane.ERROR_MESSAGE);
56              return 1;
57          }
58          return 0;
59      }
60  }
```

5. Clasele din pachetul *gui*: S-a realizat pe baza implementării acestor clase o interfață grafică prietenoasă, ușor de utilizat. Are mai multe pagini care corespund introducerii datelor necesare pentru clienți și produse, pentru vizualizarea clienților, comenziilor unui client și a produselor, pentru adăugarea unei comenzi unui client pentru un anumit produs. Panel-urile sunt de dimensiuni diferite, se deschid in mod dinamic prin accesarea unor butoane în funcție de operația dorită. Pachetul *gui* conține un alt pachet *util*, unde am implementat clasele pentru butoane și tabele, care conțin metodele pentru accesarea

unui buton sau a unui field dintr-un tabel cu clienți, produse și ordere.

```
    public class TableMouseListener extends MouseAdapter {

    private final Object[][] data;
    private final int rowHeight;
    private final List<Integer> buttonIndexes;
    private final JTable table;

    public TableMouseListener(Object[][] data, int rowHeight,
       List<Integer> buttonIndexes, JTable table) {
        this.data = data;
        this.rowHeight = rowHeight;
        this.buttonIndexes = buttonIndexes;
        this.table = table;

    }

    public void mouseClicked(MouseEvent e) {

        Integer y = table.getColumnModel().getColumnIndexAtX(e.
           getX());
        if (buttonIndexes.contains(y)) {
            int row = e.getY() / rowHeight;
            Object value = data[row][y];
            if (value instanceof JButton) {
                //perform a click event
                ((JButton) value).doClick();
            }
        }
    }
}
```

6. Clasa *Bill* din pechetul *Model*: este o clasă imutabilă utilizată pentru a genera facturi pentru comenzile tuturor clienților. O clasă imutabilă este o clasă ale cărei instanțe nu pot fi modificate după ce au fost create. În alte cuvinte, odată ce un obiect imutabil este instanțiat, starea sa internă rămâne constantă pe toată durata vieții sale. Totodată această clasă are și corespundenta sa DAO, care oferă o implementare a pattern-ului Data Access Object (DAO). Această clasă definește operațiuni (Create, Find All) pentru facturi. Această clasă a fost implementată separat cu metodele create și find all și nu a extins clasa AbstractDAO, ca celelalte clase DAO, pentru a preveni actualizările și ștergerile să fie efectuate pe ea.

```
public class BillDAO {

    protected static final Logger LOGGER = Logger.getLogger(
       AbstractDAO.class.getName());
    private static final String INSERT_BILL = "INSERT INTO log (
       orderid, generatedDate) VALUES (?, ?)";
    private static final String GET_ALL_BILL = "SELECT * FROM log
```

```
        ";
6
7       /**
8        * Creates a new bill in the database.
9        * @param bill
10       */
11
12      public void create(Bill bill) {
13          Connection connection = null;
14          PreparedStatement statement = null;
15          ResultSet resultSet = null;
16          try {
17
18              connection = ConnectionFactory.getConnection();
19              statement = connection.prepareStatement(INSERT_BILL);
20
21              statement.setObject(1, bill.orderId());
22              statement.setObject(2, bill.generatedDate());
23              statement.executeUpdate();
24              return;
25          } catch (Exception e) {
26              LOGGER.log(Level.WARNING, "BillDAO:create " + e.
                    getMessage());
27          } finally {
28              ConnectionFactory.close(resultSet);
29              ConnectionFactory.close(statement);
30              ConnectionFactory.close(connection);
31          }
32      }
33
34      /**
35       * Retrieves all bills from the database.
36       * @return
37       */
38      public List<Bill> findAll() {
39          Connection connection = null;
40          PreparedStatement statement = null;
41          ResultSet resultSet = null;
42
43          try {
44              connection = ConnectionFactory.getConnection();
45              statement = connection.prepareStatement(GET_ALL_BILL)
                    ;
46              resultSet = statement.executeQuery();
47
48              List<Bill> bills = new ArrayList<>();
49
50              while (resultSet.next()) {
51                  Integer idOrder = (Integer) resultSet.getObject("
```

```
                    orderid");
52                  LocalDateTime generatedDate = (LocalDateTime)
                        resultSet.getObject("generatedDate");
53
54                  bills.add(new Bill(idOrder, generatedDate));
55              }
56
57              return bills;
58          } catch (SQLException e) {
59              LOGGER.log(Level.WARNING, "BillDAO:findAll,query=" +
                    GET_ALL_BILL + "  " + e.getMessage());
60          } finally {
61              ConnectionFactory.close(resultSet);
62              ConnectionFactory.close(statement);
63              ConnectionFactory.close(connection);
64          }
65          return null;
66      }
67  }
```

7. De asemenea există pachetul *model* cu clasele pentru un client, un produs și o comandă, care conțin getter și settere pentru fiecare client, produs sau comandă și proprietățiile pentru un obiect de tipul fiecărei clase. Clasele *Product* și *WareHoseOrder* implementează metoda toString pentru reprezentare sub formă de string a proprietățiilor unui obiect de tipul acestor clase. Clasa *Client* conține și câmpul de gender, care este utilizat în UI de utilizator la crearea unui client pentru a decide ce gen are, pentru că se adaugă în funcție de gen o imagine cu un bărbat/femeie pentru a face interfața mai plăcută vizual și interactivă. De exemplu clasa *Product* are următoarea implementare:

```
1   public class Product {
2
3       protected Integer id;
4       private String name;
5       private String description;
6       private Integer quantity;
7       private Double price;
8
9       public Integer getId() {
10          return id;
11      }
12
13      public String getName() {
14          return name;
15      }
16
17        public Integer getQuantity() {
18          return quantity;
19      }
20
21      public Double getPrice() {
```

```
22          return price;
23      }
24
25      public void setId(Integer id) {
26          this.id = id;
27      }
28
29      public void setName(String name) {
30          this.name = name;
31      }
32
33      public void setQuantity(Integer quantity) {
34          this.quantity = quantity;
35      }
36
37      public void setPrice(Double price) {
38          this.price = price;
39      }
40
41      public String getDescription() {
42          return description;
43      }
44
45      public void setDescription(String description) {
46          this.description = description;
47      }
48      @Override
49      public String toString() {
50          return "Product{" +
51                  "id=" + id +
52                  ", name='" + name + '\'' +
53                  ", quantity=" + quantity +
54                  ", price=" + price +
55                  '}';
56      }
57  }
```

# 6 Interfața grafică

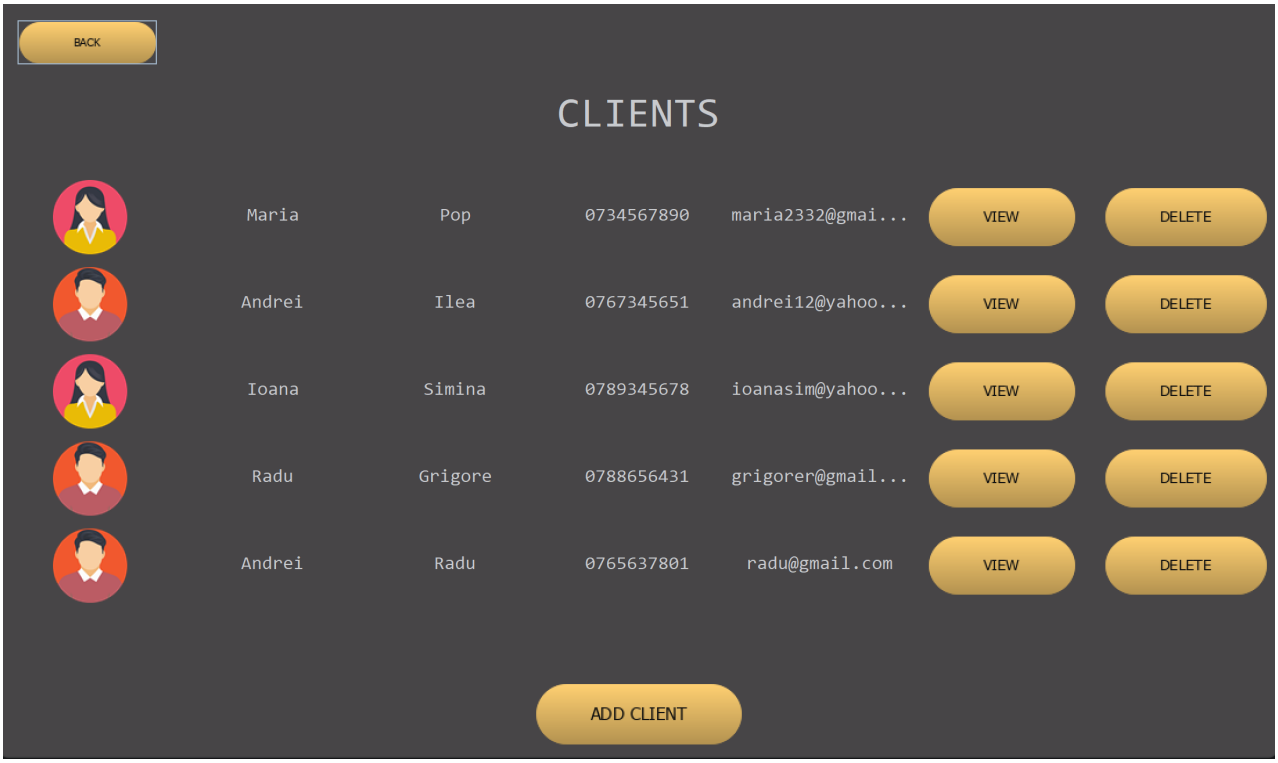


Figura 5: Afișarea clienților

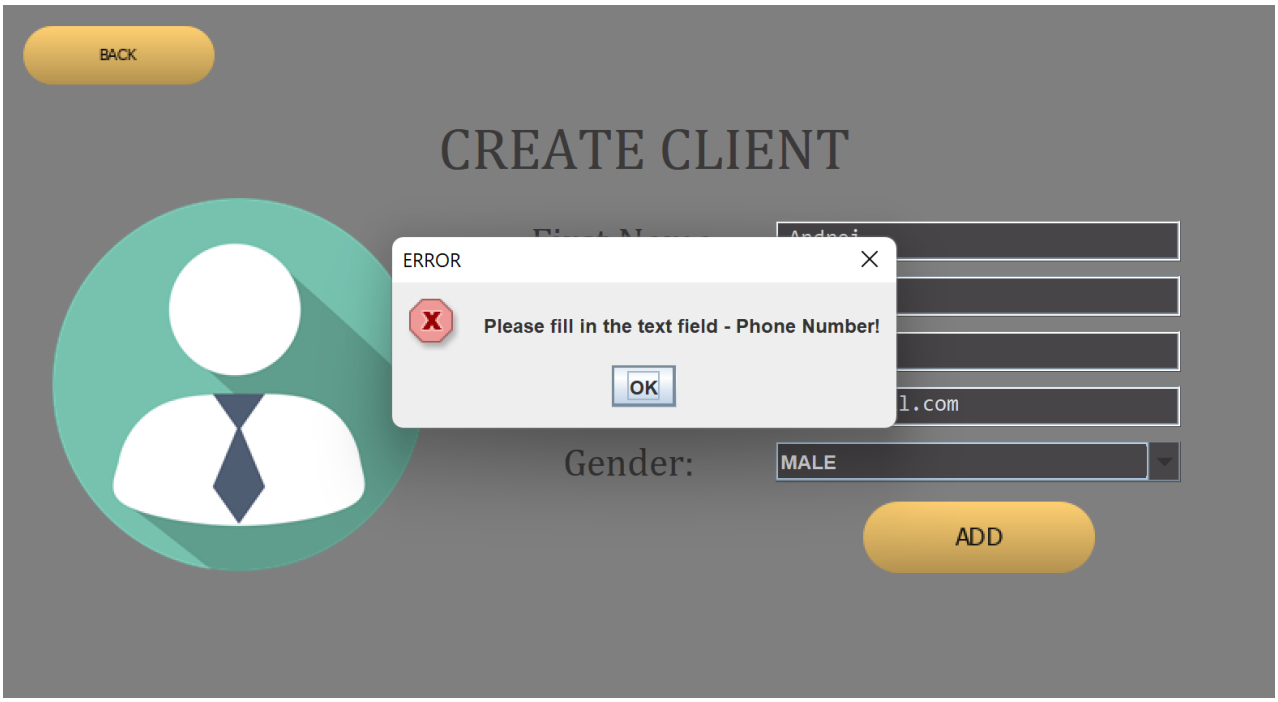


Figura 6: Prezența erorilor pe baza validărilor făcute pentru field-uri la adăugarea unui client - validare număr de telefon

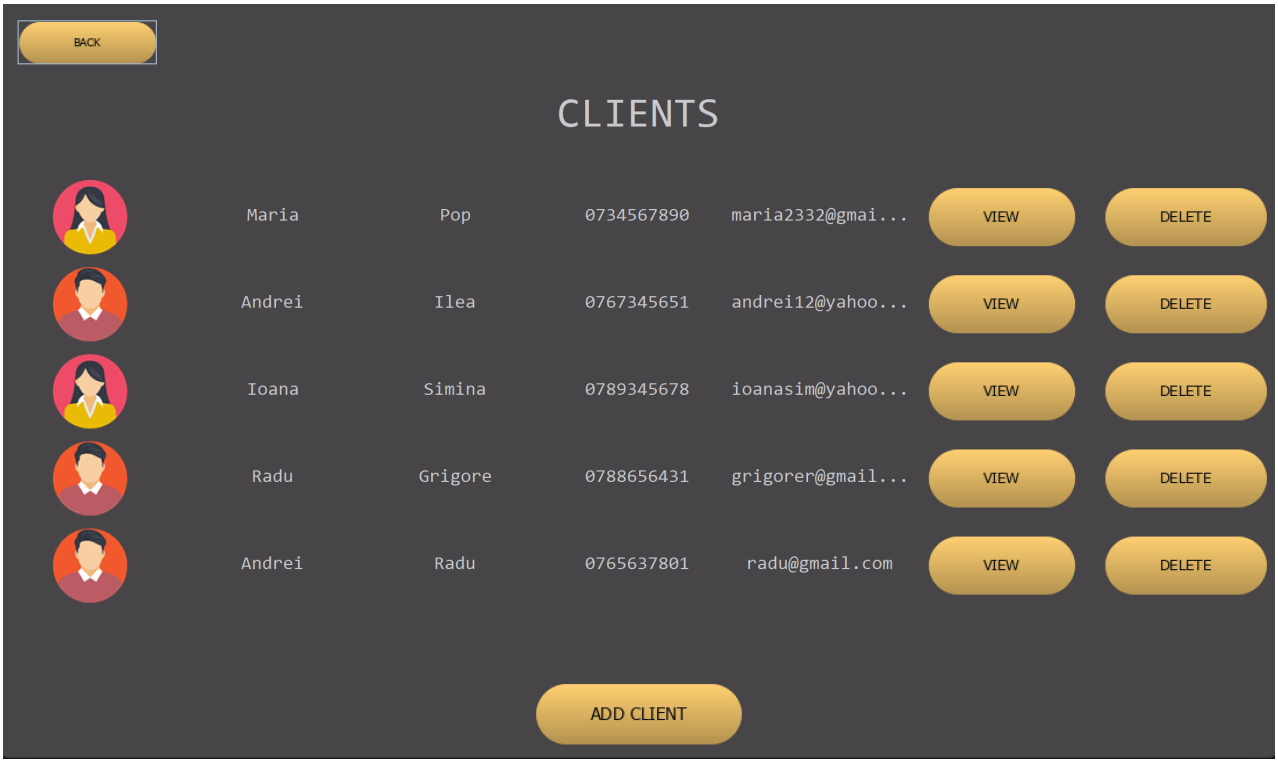


Figura 7: Prezența erorilor pe baza validărilor făcute pentru field-uri la adăugarea unui client - validare email


Figura 8: Prezența erorilor pe baza validărilor făcute pentru field-uri la editarea unui client - adăugarea field gol

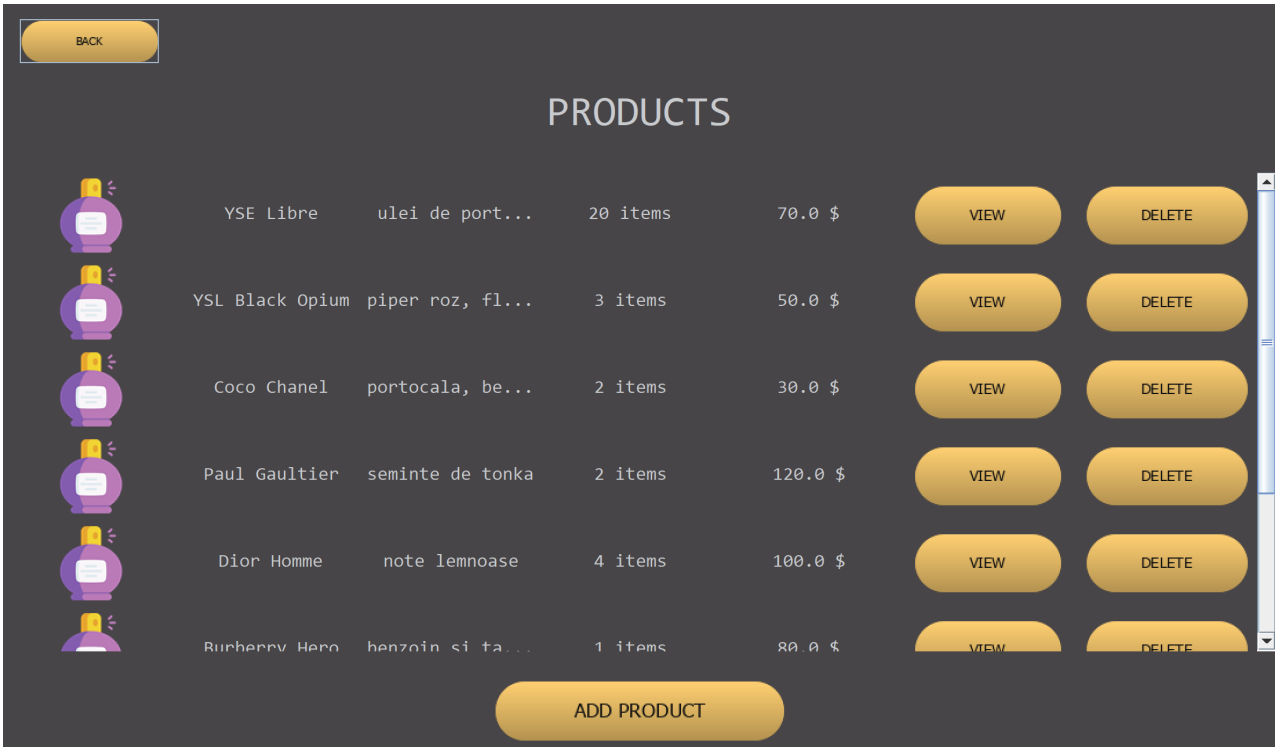


Figura 9: Afișarea produselor

# 7 JavaDoc

JavaDoc este un instrument de documentare pentru limbajul de programare Java, folosit pentru a genera documentație API într-un format HTML din comentariile specializate din codul sursă. Comentariile JavaDoc sunt plasate de obicei deasupra definițiilor de clase, metode și câmpuri și sunt marcate cu caractere speciale pentru a indica diferite tipuri de informații. Am utilizat JavaDoc pentru a documenta clasele și metodele importante din proiectul JAVA. Fiecare clasă și metodele importante din această clasă (mai puțin partea de design din pachetul gui, conțin comentarii care generează javadoc) pentru o înțelegerea mai bună a utilizatorului asupra codului. Comentariile javadoc arată ca în exemplu:

```
/**
 * The BillDAO class provides an implementation of the Data Access Object (DAO) pattern.
 * This class definesoperations (Create, Find All) for bills.
 * This class was implemented separately with the create and find all methods
 * and did not extend the class AbstractDAO, like the other DAO classes, in order to prevent updates an
 */
 MagdalenaCret
public class BillDAO {
```

Figura 10: JavaDoc

Am salvat fișierele JavaDoc într-un director seprarat din proiect *javadoc* unde se generează și html-ul corespunzător documentație codulului.

# 8 Concluzii

În cele din urmă, se poate spune că, deși, crearea unui aplicației pentru managementul unui depozit, unde se realizează comenzi pentru un client, poate părea la prima vedere o sarcină complicată, prin utilizarea tehnicilor de programare orientate pe Obiect, se poate obține o aplicație eficientă și utilă.
Îmbunătățiri ulterioare pot fi:

- Adăugarea unor strategy patterns, care într-o astfeaplicație de gestionare a unui depozit cu comenzi pentru clienți poate aduce flexibilitate și modularitate în implementarea diferitelor comportamente și strategii de manipulare a comenzilor, livrare, inventariere etc.
- Adăugarea unor animații pentru evoluția comenzilor unui client, pentru cum scade numărul de produse din depozit, etc.
- Interfață grafică îmbunătățită: pentru a fi mai intuitivă și ușor de utilizat. Adăugarea unor opțiuni de personalizare, cum ar fi selectarea culorilor temei, adăugarea a mai multor proprietăți pentru clienți, produse sau comenzi, adăugarea unui coș de cumpărături intermediar cu realizarea comenzii a mai multor produse în același timp.
- Îmbunătățiri priviind realizarea unei facturi pentru o comandă.
- Transmiterea unui email odată cu realizarea comenzii.
- Posibilitatea de logare a clientului, existența opțiunii de creare a unui cont de client.
- O posibilă îmbunătățire priviind organizarea codului.

# 9 Bibliografie (Webografie)


1. Cursuri OOP Anul2, Semestrul 1
2. https://dsrl.eu/courses/pt/materials/lectures/
3. https://stackoverflow.com/questions/14852719/double-click-listener-on-jtable-in-java
4. https://stackoverflow.com
5. https://stackoverflow.com/questions/4685500/regular-expression-for-10-digit-number-without-any-special-characters
6. https://www.tutorialspoint.com/how-can-we-make-jtextfield-accept-only-numbers-in-java
7. https://regex101.com/
8. https://gitlab.com/utcn_dsrl/pt-layered-architecture
9. https://www.overleaf.com/learn/latex
10. https://libguides.eur.nl/overleaf/lists-tables-images-labelling
11. https://www.jetbrains.com/help/idea/javadocs.htmlcustom-tags-javadocs
12. https://www.baeldung.com/java-date-to-localdate-and-localdatetime
13. https://www.flaticon.com/free-icons/order