DOCUMENTATIE

TEMA 3

NUME STUDENT: Pop Tudor Ștefan

GRUPA: 30226

# CUPRINS

[DOCUMENTATIE 1](#_Toc295481402)

[TEMA 3 1](#_Toc895477427)

[CUPRINS 1](#_Toc217426770)

[1. Obiectivul temei 2](#_Toc1656216567)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 2](#_Toc831907749)

[3. Proiectare 6](#_Toc2062242919)

[4. Implementare 7](#_Toc1860538903)

[5. Rezultate 10](#_Toc1663929869)

[6. Concluzii 10](#_Toc2069400119)

# Obiectivul temei

Obiectivul principal al acestei teme este proiectarea și implementarea unei aplicații care gestionează comenzile clienților pentru un depozit. Ca și sub-obiective, putem identifica următoarele:

1. Analiza problemei și identificarea cerințelor
2. Proiectarea aplicației de gestionare a comenzilor
3. Implementarea aplicației de gestionare a comenzilor
4. Testarea acestei aplicații

# Analiza problemei, modelare, scenarii, cazuri de utilizare

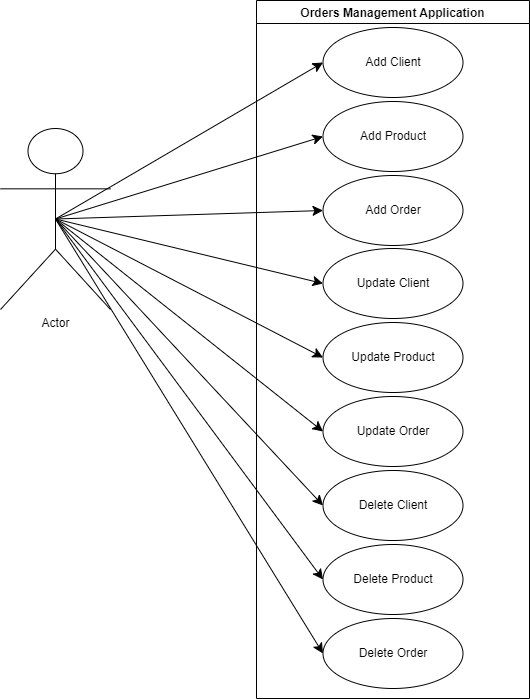
**Cerințele Funcționale:**

1. Aplicația trebuie să permită unui angajat să adauge un nou client
2. Aplicația trebuie să permită unui angajat să adauge un nou produs
3. Aplicația trebuie să permită unui angajat să adauge o nouă comandă
4. Aplicația trebuie să permită unui angajat să actualizeze un client existent
5. Aplicația trebuie să permită unui angajat să actualizeze un produs existent
6. Aplicația trebuie să permită unui angajat să actualizeze o comandă existentă
7. Aplicația trebuie să permită unui angajat să șteargă un client existent
8. Aplicația trebuie să permită unui angajat să șteargă un produs existent
9. Aplicația trebuie să permită unui angajat să șteargă o comandă existentă

**Cerințele Non-Funcționale:**

1. Aplicația trebuie să fie intuitivă și ușoară de utilizat pentru actor.
2. Aplicația trebuie să fie eficientă în raport cu timpul de execuție a comenzilor.

**Cazuri de Utilizare:**

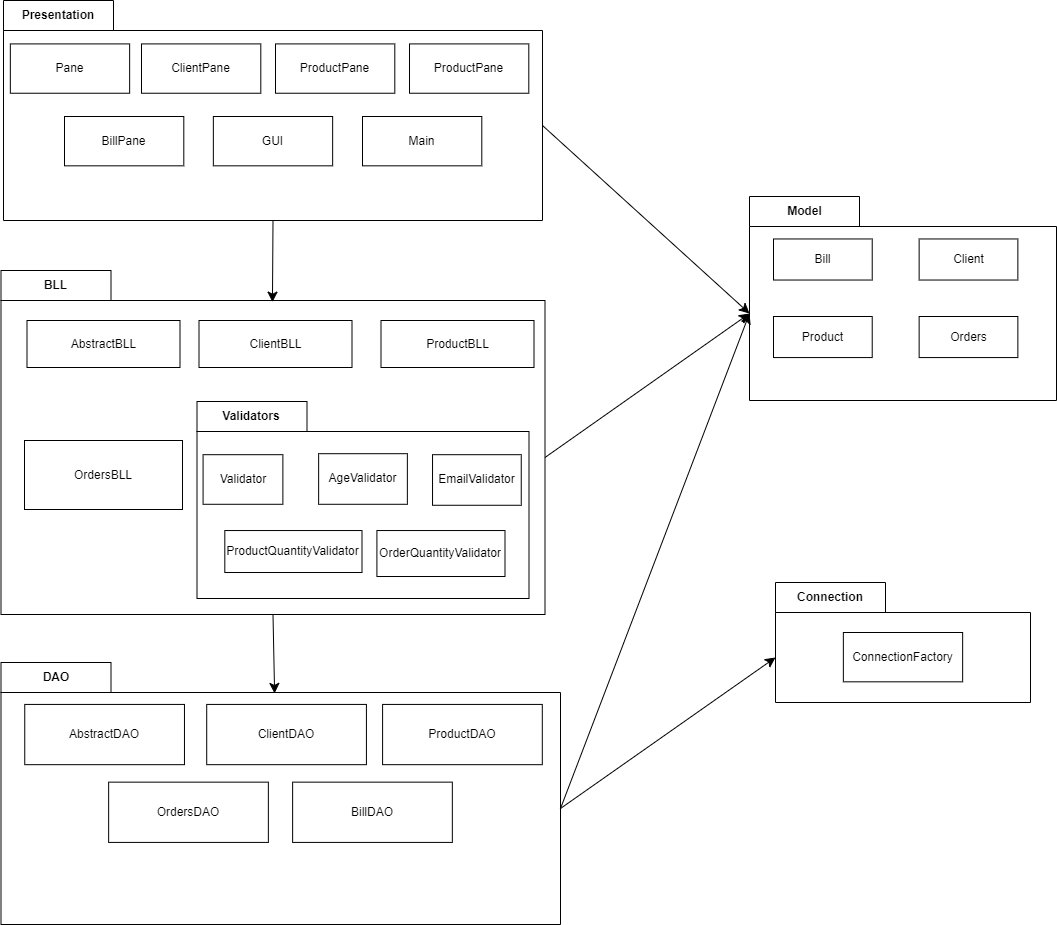


**Figura 1. Cazurile de Utilizare.**

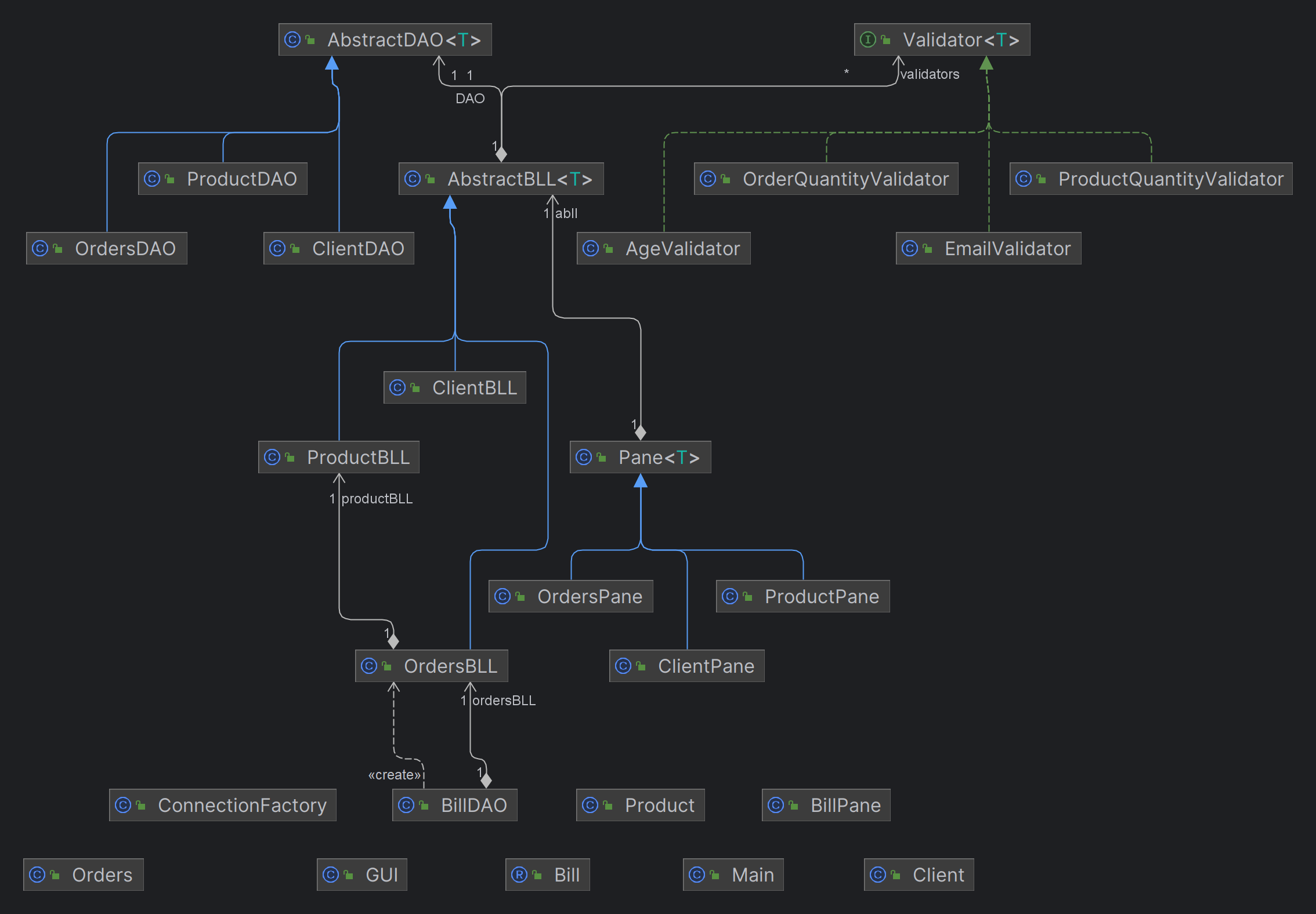
Vom prezenta doar primele trei cazuri de utilizare, deoarece, următoarele șase vor avea o procedură complet echivalentă, datorită tehnicilor de reflexie utilizate.

1. ***Inserarea unui client nou:***
2. **Cazul de Utilizare:** Inserarea unui client nou
3. **Actorul Principal:** Angajatul la depozit
4. **Scenariul de Succes:** Angajatul inserează date noi în căsuțele text delimitate în interfața grafică de către textul “Insert” și butonul “Insert”. În condițiile în care datele introduse sunt considerate valide, se apasă butonul “Insert”, și tabelul de clienți este actualizat în timp real cu noua entitate introdusă.
5. **Scenariul de eșec:** În cazul în care s-au introdus date invalide, consola va semnala acest lucru foarte concret, și se revine la inserarea datelor noi în căsuța text.
6. ***Actualizarea unui client existent:***
7. **Cazul de Utilizare:** Actualizarea unui client existent
8. **Actorul Principal:** Angajatul la depozit
9. **Scenariul de Succes:** Angajatul inserează date noi în căsuțele text delimitate în interfața grafică de către textul “Update” și butonul “Update”. În condițiile în care datele introduse sunt considerate valide, se apasă butonul “Update”, și tabelul de clienți este actualizat în timp real cu entitatea care a fost modificată.
10. **Scenariul de eșec:** În cazul în care s-au introdus date invalide, consola va semnala acest lucru foarte concret, și se revine la inserarea datelor noi în căsuța text.
11. ***Ștergerea unui client existent:***
12. **Cazul de Utilizare:** Ștergerea unui client existent
13. **Actorul Principal:** Angajatul la depozit
14. **Scenariul de Succes:** Angajatul inserează un identificator unic în ultima căsuță text a aplicației. În condițiile în care identificatorul unic există în baza de date, se apasă butonul “Delete”, și tabelul de clienți este actualizat în timp real cu entitatea care a fost modificată.
15. **Scenariul de eșec:** În cazul în care s-au introdus date invalide, consola va semnala acest lucru foarte concret, și se revine la inserarea datelor noi în căsuța text.

# Proiectare



**Figura 2. Diagrama UML de pachete.**

**Figura 3. Diagrama UML de clase.**

# Implementare

1. **ConnectionFactory.java:**

Constantele “DBURL”, “USER”, și “PASS” reprezintă datele necesare conexiunii cu baza de date MySQL. Constanta “LOGGER” reprezintă un mecanism de a semnala erorile sau avertismentele din program. Metoda “createConnection()” furnizează o conexiune cu baza de date MySQL prin utilizarea celor 3 constante esențiale descrise mai sus. În caz de excepție, se afișează un mesaj de eroare prin intermediul constantei “LOGGER”. Metodele “close(Connection | Statement | ResultSet)” doresc închiderea unei conexiuni, unei instrucțiuni, sau ale unui ResultSet. În caz de eșec, constanta “LOGGER” se ocupă de afișarea mesajului de eroare.

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/pt3";  
private static final String *USER* = "root";  
private static final String *PASS* = "TudorStefan#1";

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 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");  
 }  
 }  
}

1. **AbstractDAO.java:**

Precum în clasa descrisă anterior, constanta "LOGGER” este utilizată pentru a gestiona erorile/avertismentele aplicației în timpul utilizării sale. Variabila “type” stochează clasa exactă a tipului parametrizat T. Constructorul “AbstractDAO()” setează o valoare explicită acestei variabile “type”. După cum îi spune numele, metoda “createSelectQuery()” crează o interogare de tip “Select all from <tableName>” asupra bazei de date MySQL. Metoda “findAll()” reprezintă rezultatul fizic din baza de date relațională al unei interogări “Select all”. Metoda “createObjects()” crează o listă de instanțe ale clasei concrete a tipului parametrizat T. Metoda “findById()” este rezultatul fizic al execuției unei interogări de forma “Select all from <tableName> where id = <desiredID>”. Metoda “getValues()” extrage valorile atributelor unui obiect de tip T. Metodele “insert() | update() | delete()” codifică, prin intermediul interogărilor SQL, operațiile de inserare | modificare | ștergere a unui obiect de tip T în tabelul său corespondent.

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];  
}

private String createSelectQuery(String field) {  
 StringBuilder sb = new StringBuilder();  
 sb.append("SELECT ");  
 sb.append("\* ");  
 sb.append("FROM ");  
 sb.append(type.getSimpleName());  
 sb.append(" WHERE ").append(field).append("=?");  
 return sb.toString();  
}  
public List<T> findAll() {  
 Connection connection = null;  
 PreparedStatement statement = null;  
 ResultSet resultSet = null;  
 String query = "SELECT \* FROM " + type.getSimpleName();  
 try {  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(query);  
 resultSet = statement.executeQuery();  
 return createObjects(resultSet);  
 } catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, type.getName() + "DAO:findAll " + e.getMessage());  
 } finally {  
 ConnectionFactory.*close*(resultSet);  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
 return new ArrayList<>();  
}  
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).getFirst();  
 } catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, type.getName() + "DAO:findById " + e.getMessage());  
 } finally {  
 ConnectionFactory.*close*(resultSet);  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
 return null;  
}  
  
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 | IllegalAccessException | SecurityException | IllegalArgumentException |  
 InvocationTargetException | SQLException | IntrospectionException e) {  
 e.printStackTrace();  
 }  
 return list;  
}  
  
private Map<String,String> getValues(T t){  
 Map<String, String> result = new HashMap<>();  
 try {  
 for(Field f: type.getDeclaredFields()) {  
 String fieldName = f.getName();  
 PropertyDescriptor propertyDescriptor = new PropertyDescriptor(fieldName, type);  
 Method method = propertyDescriptor.getReadMethod(); // age -> getAge  
 if(f.getType() == String.class){  
 result.put(fieldName, "'" + method.invoke(t).toString() + "'");  
 }else {  
 result.put(fieldName, method.invoke(t).toString());  
 }  
 }  
 } catch (IntrospectionException | IllegalAccessException | InvocationTargetException e) {  
 e.printStackTrace();  
 }  
 return result;  
}  
public void insert(T t) {  
 Map<String, String> fields = getValues(t);  
 StringBuilder query = new StringBuilder();  
 query.append("INSERT INTO ").append(type.getSimpleName()).append(" (");  
 for (String key : fields.keySet()){  
 query.append(key).append(", ");  
 }  
 query.deleteCharAt(query.length()-1);  
 query.deleteCharAt(query.length()-1);  
 query.append(") ");  
 query.append("VALUES (");  
 for (String value : fields.values()){  
 query.append(value).append(", ");  
 }  
 query.deleteCharAt(query.length()-1);  
 query.deleteCharAt(query.length()-1);  
 query.append(");");  
 Connection connection = null;  
 PreparedStatement statement = null;  
 ResultSet resultSet = null;  
 try {  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(query.toString());  
 statement.execute();  
 } catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, type.getName() + "DAO:insert " + e.getMessage());  
 } finally {  
 ConnectionFactory.*close*(resultSet);  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
}  
public void update(T t) {  
 Map<String, String> fields = getValues(t);  
 StringBuilder query = new StringBuilder();  
 query.append("UPDATE ").append(type.getSimpleName()).append(" SET ");  
 for (Map.Entry<String, String> entry : fields.entrySet()) {  
 query.append(entry.getKey()).append(" = ").append(entry.getValue()).append(", ");  
 }  
 query.deleteCharAt(query.length()-1);  
 query.deleteCharAt(query.length()-1);  
 query.append(" WHERE id = ").append(fields.get("id")).append(";");  
 Connection connection = null;  
 PreparedStatement statement = null;  
 ResultSet resultSet = null;  
 try {  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(query.toString());  
 statement.execute();  
 } catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, type.getName() + "DAO:update " + e.getMessage());  
 } finally {  
 ConnectionFactory.*close*(resultSet);  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
}  
public void delete(T t){  
 Map<String, String> fields = getValues(t);  
 StringBuilder query = new StringBuilder();  
 query.append("DELETE FROM ").append(type.getSimpleName()).append(" WHERE id = ").append(fields.get("id")).append(";");  
 Connection connection = null;  
 PreparedStatement statement = null;  
 ResultSet resultSet = null;  
 try {  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(query.toString());  
 statement.execute();  
 } catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, type.getName() + "DAO:delete " + e.getMessage());  
 } finally {  
 ConnectionFactory.*close*(resultSet);  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
}

1. **ClientDAO.java, ProductDAO.java, OrdersDAO.java:**

Implementările concrete ale clasei abstracte “AbstractDAO”.

1. **Client.java:**

Variabilele “id”, “age”, “name”, “address”, “email” reprezintă tupla care definește un client. Fiecare client este unic identificat prin ID-ul său. Constructorul implicit cu corp nul “Client()” servește un scop foarte important în contextul apelării metodei “getDeclaredConstructor()” care cere ca și argumente constructori impliciți. Metoda “toString()” traduce obiectul de tip Client într-o variantă afișabilă și inteligibilă pentru programator, în contextul depanării programului. Metodele ce nu au fost discutate sunt: constructorul cu parametri, și metode mutatoare și accesoare pentru fiecare dintre variabilele instanță.

private Integer id, age;  
private String name, address, email;

public Client(){}

@Override  
public String toString() {  
 return "Client:\nID: " + id + "\nName: " + name + "\nAddress: " + address + "\nE-Mail: " + email + "\nAge: " + age + "\n";  
}

1. **Product.java:**

Un produs este definit ca tripletul format din identificatorul său unic, cantitatea în limita stocului disponibil, și numele său. Constructorul implicit “Product()” are o importanță motivată analog cu cea a constructorului implicit “Client()”, analog pentru metoda “toString()”.

private Integer id, quantity;  
private String name;

public Product() {}

@Override  
public String toString() {  
 return "Product:\nID: " + id + "\nName: " + name + "\nQuantity: " + quantity + "\n";  
}

1. **Orders.java:**

Procesul de implementare a acestei clase este complet analog cu cel al celor două clase p

private Integer id, clientID, productID, quantity;

public Orders() {}

@Override  
public String toString() {  
 return "Order\nID: " + id + "\nClient ID: " + clientID + "\nProduct ID: " + productID + "\nQuantity: " + quantity + "\n";  
}

1. **Bill.java:**

Utilizând “java records” (“public record <record\_name>”), putem declara tipuri noi de obiecte imutabile. Recordul “Bill” este definit ca un patruplet format din aceleași date ca și cele ale unei comenzi. Scopul său este de a ține o referință către toate comenzile depuse până în momentul actual, și după ce acestea sunt șterse.

public record Bill(int order, int client, int product, int quantity) {}

1. **ClientDAO.java, ProductDAO.java, OrdersDAO.java:**

Implementările concrete ale clasei abstracte “AbstractDAO<T>”, unde T se înlocuiește cu fiecare tip respectiv.

public class ClientDAO extends AbstractDAO<Client> {  
}

public class OrdersDAO extends AbstractDAO<Orders> {  
}

public class ProductDAO extends AbstractDAO<Product> {  
}

1. **Validator.java:**

O interfață care conține doar metoda “public boolean validate(T t)”, care va testa dacă variabila de tip T dată respectă o mână de condiții pe care le vom implementa diferit în clasele care implementează această interfață.

1. **AgeValidator.java:**

Prima clasă care implementează interfața Validator. Aceasta verifică dacă vârsta clientului se încadrează între MIN\_AGE și MAX\_AGE, și îl validează sau invalidează în funcție de caz.

1. **EmailValidator.java:**

A doua clasă care implementează interfața Validator. Aceasta verifică dacă E-Mail-ul clientului respectă modelul impus de către o expresie regulată foarte sofisticată, prea lungă pentru scopurile acestei documentații.

1. **ProductQuantityValidator.java:**

A treia clasă care implementează interfața Validator. Aceasta verifică dacă numărul de bucăți rămase în stoc ale unui produs (cantitatea acestuia) este un număr pozitiv sau 0, precum ar trebui să fie. În caz contrar, se semnalează un caz de eroare, care va fi interceptat în Main.java ca fiind un caz excepțional al programului.

1. **OrderQuantityValidator.java:**

A patra, respectiv ultima, clasă care implementează interfața Validator. Aceasta verifică dacă numărul de bucăți comandate este un număr pozitiv, și, dacă nu se depășește limita stocului disponibil a produsului respectiv. În caz contrar, se semnalează o situație excepțională.

1. **AbstractBLL.java:**

Clasa de tip Business Logic Layer care va fi concretizată de către moștenitorii săi ClientBLL.java, ProductBLL.java, și OrdersBLL.java. Această clasă conține ca și atribute variabilele “validators" și “DAO”. Metoda “checkValidators()” verifică dacă toate datele de intrare ale unui obiect sunt valide conform fiecăruia dintre validatori. Metodele “findAll()”, “findById()”, “insert()”, “update()”, și “delete()” sunt implementate pentru a le pasa următorului strat din această arhitectură pe straturi.

private final List<Validator<T>> validators;  
private final AbstractDAO<T> DAO;

protected void checkValidators(T t){  
 for (Validator<T> validator : validators)  
 validator.validate(t);  
}

public List<T> findAll() {  
 return DAO.findAll();  
}

public T findById(int id) throws NoSuchElementException {  
 T t = DAO.findById(id);  
 if (t == null)  
 throw new NoSuchElementException(DAO.getType().getName() + " with ID " + id + " not found!");  
 return t;  
}

public void insert(T t) {  
 checkValidators(t);  
 DAO.insert(t);  
}

public void update(T t) {  
 checkValidators(t);  
 DAO.update(t);  
}

public void delete(T t) {  
 DAO.delete(t);  
}

1. **ClientBLL.java, ProductBLL.java, OrdersBLL.java:**

Moștenitorii concreți ai clasei AbstractBLL.java. Însă, clasa OrdersBLL.java este puțin mai delicată decât celelalte două. În momentul în care se depune o comandă, se cere o anumită cantitate dintr-un anumit produs. Acea cantitate va trebui să fie scăzută din stocul total al acelui produs. Prin urmare, inserarea și actualizarea unei comenzi vor influența și intrări din tabelul de produse, așadar, implementările metodelor “insert()” și “update()” vor trebui modificate în contextul acestei clase. O altă consecință a necesității de a modifica cantitatea unui anumit produs este nevoia de o variabilă de “business logic layer” pentru gestionarea produselor; să o numim “productBLL”.

private final ProductBLL productBLL;

@Override  
public void insert(Orders order) {  
 checkValidators(order);  
 Product p = productBLL.findById(order.getProductID());  
 if (order.getQuantity() > p.getQuantity())  
 throw new IllegalArgumentException("Not enough items in stock!");  
 p.setQuantity(p.getQuantity() - order.getQuantity());  
 productBLL.update(p);  
 super.insert(order);  
}

@Override  
public void update(Orders order) {  
 checkValidators(order);  
 super.update(order);  
 Product newProduct = productBLL.findById(order.getProductID());  
 newProduct.setQuantity(newProduct.getQuantity() - order.getQuantity());  
 productBLL.update(newProduct);  
}

1. **Pane.java:**

Cea mai importantă clasă din descrierea interfeței utilizator. Aceasta reprezintă un obiect de tip “JSplitPane”, implementat utilizând tehnici de reflexie, în care definim și inițializările secțiunilor din panou, și acțiunile de inserare/actualizare/ștergere în funcție de datele introduse. Această clasă guvernează comportamentul tuturor celor 3 moștenitori ai săi: ClientPane.java, ProductPane.java, OrdersPane.java. Pentru a implementa corect funcționalitățile dorite, sunt necesare o serie de variabile instanță prin care putem apela toate metodele aferente.

private JTable t;  
*/\*\**  
 *\* The general model that the table should respect.*  
 *\*/*  
private DefaultTableModel model;  
*/\*\**  
 *\* An Abstract Business Logic Layer which handles all the operations.*  
 *\*/*  
private AbstractBLL<T> abll;  
*/\*\**  
 *\* The component left pane and right pane.*  
 *\*/*  
private JPanel leftPane, rightPane;  
*/\*\**  
 *\* The aforementioned buttons.*  
 *\*/*  
private JButton insertButton, updateButton, deleteButton, refreshButton;  
*/\*\**  
 *\* The concrete type of the parameterized type T.*  
 *\*/*  
private Class<T> type;  
*/\*\**  
 *\* Maps describing the text fields for each of the insertion / modification sections.*  
 *\*/*  
private Map<String, JTextField> insert\_textFields, update\_textFields;  
*/\*\**  
 *\* The text field associated with deletion. Deletion will be performed by IDs.*  
 *\*/*  
private JTextField deleteTextField;

public Pane(AbstractBLL<T> abll) {  
 this.type = (Class<T>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[0];  
 leftPane = new JPanel();  
 rightPane = new JPanel();  
 initTable();  
 leftPane.setLayout(new BoxLayout(leftPane, BoxLayout.*Y\_AXIS*));  
 rightPane.setLayout(new BoxLayout(rightPane, BoxLayout.*Y\_AXIS*));  
 model = (DefaultTableModel) t.getModel();  
 insertButton = new JButton("Insert");  
 updateButton = new JButton("Update");  
 deleteButton = new JButton("Delete");  
 refreshButton = new JButton("Refresh");  
 insertButton.setFont(new Font("Comic Sans MS", Font.*ITALIC*, 12));  
 updateButton.setFont(new Font("Comic Sans MS", Font.*ITALIC*, 12));  
 deleteButton.setFont(new Font("Comic Sans MS", Font.*ITALIC*, 12));  
 refreshButton.setFont(new Font("Comic Sans MS", Font.*ITALIC*, 12));  
 refreshButton.addActionListener(this);  
 setLeftComponent(leftPane);  
 rightPane.add(new JScrollPane(t));  
 rightPane.add(refreshButton);  
 setRightComponent(rightPane);  
 this.abll = abll;  
 initTableInsert();  
 initTableUpdate();  
 initTableDelete();  
 refreshTable();  
}

public void initTableInsert() {  
 JLabel insLabel = new JLabel("Insert new " + type.getSimpleName());  
 insLabel.setFont(new Font("Comic Sans MS", Font.*BOLD*, 20));  
 leftPane.add(insLabel);  
 insert\_textFields = new HashMap<>();  
 for (Field f : type.getDeclaredFields()) {  
 JLabel label = new JLabel(f.getName());  
 label.setFont(new Font("Comic Sans MS", Font.*BOLD*, 12));  
 leftPane.add(label);  
 JTextField textField = new JTextField(10);  
 insert\_textFields.put(f.getName(), textField);  
 leftPane.add(textField);  
 }  
 insertButton.addActionListener(this);  
 leftPane.add(insertButton);  
}

public void initTableUpdate() {  
 JLabel updLabel = new JLabel("Update " + type.getSimpleName());  
 updLabel.setFont(new Font("Comic Sans MS", Font.*BOLD*, 20));  
 leftPane.add(updLabel);  
 update\_textFields = new HashMap<>();  
 for (Field f : type.getDeclaredFields()) {  
 JLabel label = new JLabel(f.getName());  
 label.setFont(new Font("Comic Sans MS", Font.*BOLD*, 12));  
 leftPane.add(label);  
 JTextField textField = new JTextField(10);  
 update\_textFields.put(f.getName(), textField);  
 leftPane.add(textField);  
 }  
 updateButton.addActionListener(this);  
 leftPane.add(updateButton);  
}

public void initTableDelete() {  
 JLabel delLabel = new JLabel("Delete " + type.getSimpleName());  
 delLabel.setFont(new Font("Comic Sans MS", Font.*BOLD*, 20));  
 leftPane.add(delLabel);  
 JLabel idLabel = new JLabel("id");  
 idLabel.setFont(new Font("Comic Sans MS", Font.*BOLD*, 12));  
 leftPane.add(idLabel);  
 deleteTextField = new JTextField(20);  
 leftPane.add(deleteTextField);  
 deleteButton.addActionListener(this);  
 leftPane.add(deleteButton);  
}

public void initTable() {  
 Vector<String> fields = new Vector<>();  
 for (Field f : type.getDeclaredFields()) {  
 fields.add(f.getName());  
 }  
 DefaultTableModel tableModel = new DefaultTableModel(fields, 30);  
 t = new JTable(tableModel);  
}

public void refreshTable() {  
 model.setRowCount(0);  
 for (T t : abll.findAll()) {  
 List<Object> lineData = new ArrayList<>();  
 for (Field f : t.getClass().getDeclaredFields()) {  
 try {  
 String fieldName = f.getName();  
 PropertyDescriptor propertyDescriptor = new PropertyDescriptor(fieldName, type);  
 Method method = propertyDescriptor.getReadMethod();  
 lineData.add(method.invoke(t));  
 } catch (IntrospectionException | InvocationTargetException | IllegalAccessException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 model.addRow(lineData.toArray());  
 System.*out*.println(lineData);  
 }  
}

1. **BillPane.java (Funcționalitate Incompletă):**

Ultima parte componentă a interfeței grafice. Aceasta nu poate fi moștenitoare a clasei Pane.java, deoarece Pane conține metode de actualizare și ștergere a datelor dintr-o tabelă. Aceste funcționalități sunt strict interzise pentru tabelul de note de plată.

private JTable table;  
*/\*\**  
 *\* The Log table's model.*  
 *\*/*  
private DefaultTableModel tableModel;  
  
*/\*\**  
 *\* The constructor for the pane itself.*  
 *\*/*  
public BillPane() {  
 table = new JTable();  
 initTable();  
}  
  
*/\*\**  
 *\* An initializer for the Log table.*  
 *\*/*  
public void initTable() {  
 Vector<String> fields = new Vector<>();  
 for (Field f : BillPane.class.getDeclaredFields()) {  
 fields.add(f.getName());  
 }  
 tableModel = new DefaultTableModel(fields, 50);  
 table = new JTable(tableModel);  
}

1. **GUI.java:**

Clasa care unește cele patru părți componente ale interfeței grafice utilizator. Se crează un obiect de tip “JTabbedPane”, ale cărui 4 componente vor fi ClientPane, ProductPane, OrdersPane, și BillPane.

package Presentation;  
  
import Model.Bill;  
  
import javax.swing.\*;  
  
*/\*\**  
 *\* The GUI which implements all the heirs of the {@link Presentation.Pane} class into a tabbed pane structure.*  
 *\*/*  
public class GUI {  
 public GUI() {  
 JFrame frame = new JFrame();  
 frame.setSize(800,700);  
 frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 JTabbedPane tabbedPane = new JTabbedPane();  
 tabbedPane.setSize(800, 700);  
  
 tabbedPane.add("Client", new ClientPane());  
 tabbedPane.add("Product", new ProductPane());  
 tabbedPane.add("Orders", new OrdersPane());  
 tabbedPane.add("Bill", new BillPane());  
  
 frame.setContentPane(tabbedPane);  
 frame.setVisible(true);  
 }  
}

1. **Main.java:**

Clasa în care se rulează aplicația implementată.

package Presentation;  
  
import DAO.ClientDAO;  
import DAO.OrdersDAO;  
import DAO.ProductDAO;  
import Model.Client;  
import Model.Orders;  
import Model.Product;  
  
*/\*\**  
 *\* The main class.*  
 *\*/*  
public class Main {  
  
 public static void main(String[] args) {  
 new GUI();  
 }  
}

# Rezultate

Validatoarele sunt complet funcționale. Orice dată introdusă corect este acceptată, și orice dată introdusă greșit va fi respinsă, și se va afișa un mesaj de eroare în cadrul consolei programului.

# Concluzii

În concluzie, din această temă s-a învățat utilizarea tehnicilor de reflexie în programarea orientată pe obiect, importanța acestor tehnici în a reduce bucăți de cod repetitive, respectiv, s-a învățat metoda eficientă Java pentru conexiunea cu o bază de date MySQL. Clasa Bill și funcționalitățile sale nu sunt implementate în întregime, așadar, acestea s-ar putea considera o posibilitate de dezvoltare ulterioară.