DOCUMENTATION

ORDERS MANAGEMENT APPLICATION

ASSIGNMENT *3*

STUDENT NAME: Oltean Darius-Ionuț
GROUP: 30224

CONTENTS

1. ASSIGNMENT OBJECTIVE

The main objective of this project is to design, implement and test an Orders Management Application using reflection mechanisms. To achieve this, I had to go through the following steps:

- Problem analysis, modeling solutions and scenarios and build use cases will be described in detail in the second section.
- OOP design of the application along with the UML package and class diagrams, in section three.
- In the fourth section it will be described the implementation of each class and important methods contained therein.
- Section five will be dedicated to the result and test cases of the application.
- In the last section I will describe a brief conclusion and possible future developments.

2. PROBLEM ANALYSIS, MODELING, SCENARIOS, USE CASES

An orders management application should model clients, products, orders and bills. The user should add new entities and those should be stored in a persistence layer.

To model a client the application should know it's id, age, name. address and an identity card ID. A product should have a name, quantity, price, manufacturer and an ID. A bill should model an ID and a total price. In the end a order should store a client ID, a product ID, an amount and a bill.

There are a few use scenarios. You can add a valid client or a valid product, but you can also insert invalid inputs like a negative quantity for a product or a invalid age for a client, in this case the application would show a error dialog. Inserting an order can be valid only if u select a valid product, a valid client and a correct amount, it should not be negative or above the total quantity of the product.

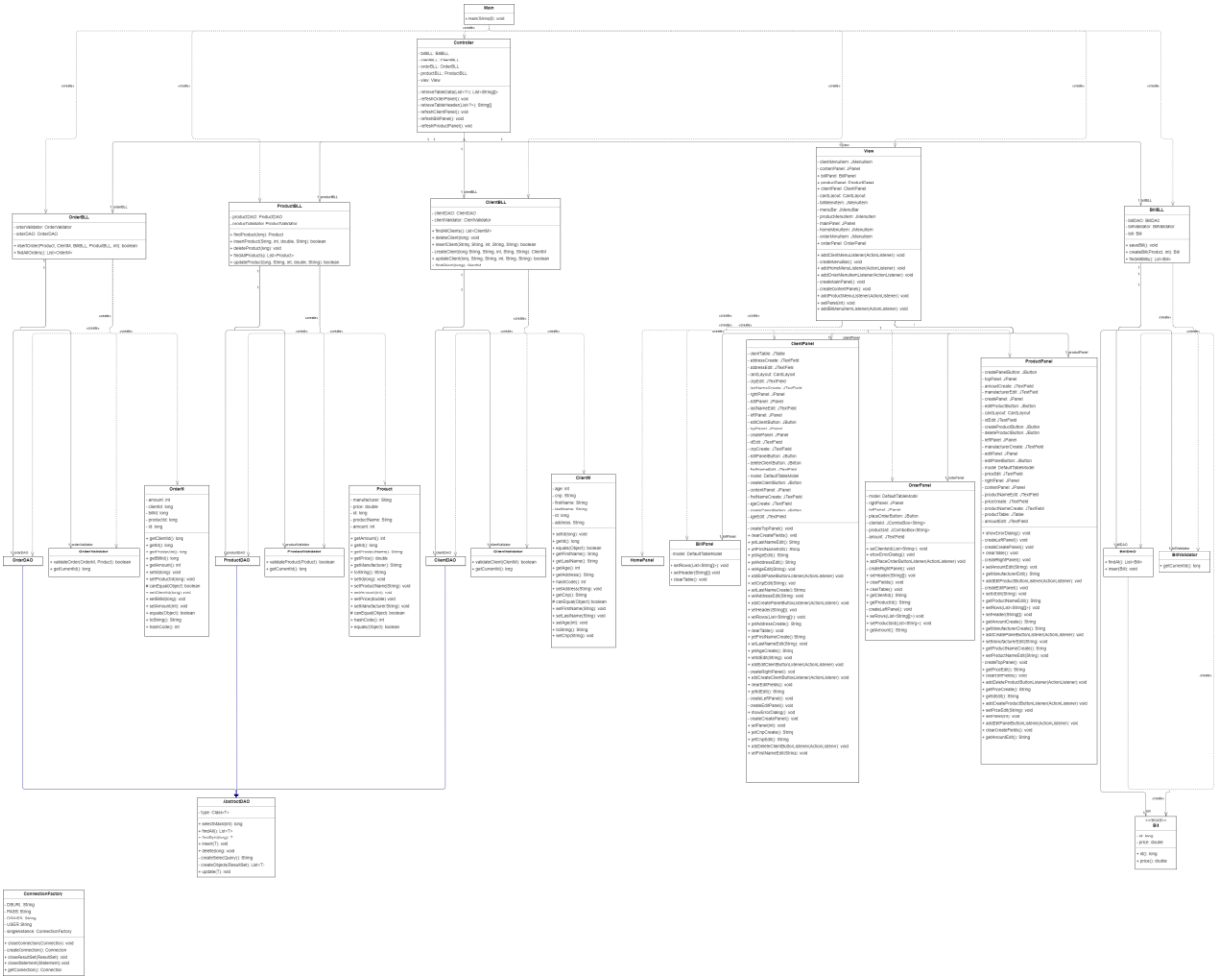
- **Use case description scenario:**
  - **Use case:** Placing an order.
  - **Primary actor:** User.
  - **Success scenario steps:**
    - The user selects a product ID, a client ID and a valid amount.
    - The user press the add button.
    - The application validates the data, stores it in the database and refreshes the orders panel.
  - **Alternative scenarios:**
    - **Invalid input:**
      - ✓ The user inserts an amount which is over the actual quantity of the product.
      - ✓ The application displays an error message and requests the user to insert valid values.
      - ✓ The scenario returns to step 1.

3. DESIGN

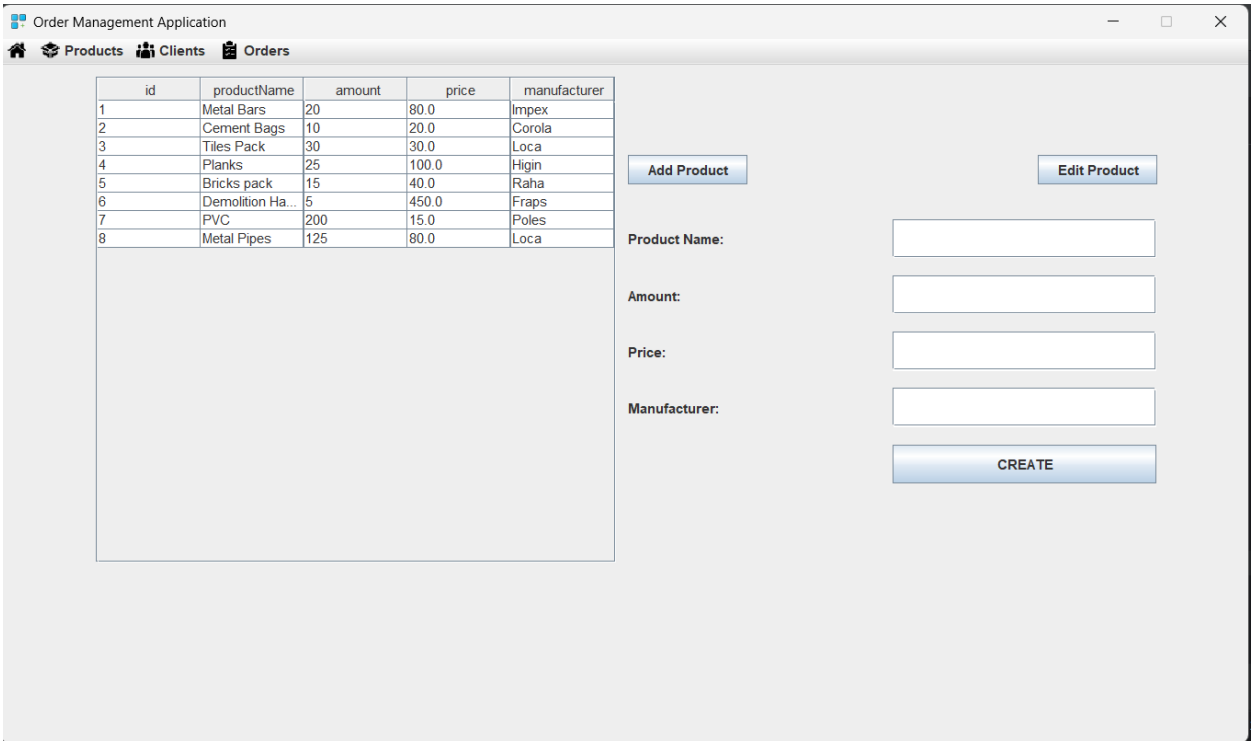
The OOP design of the application consists in 6 packages. The graphical user interface is implemented in the View package which contains View class, Controller class and another five classes which extends JPanel for each model and a home screen. Data models are described in the Model package. Logic package implements the business logic of the application, Connection package has only a helper class, ConnectionFactory, and retrieves the

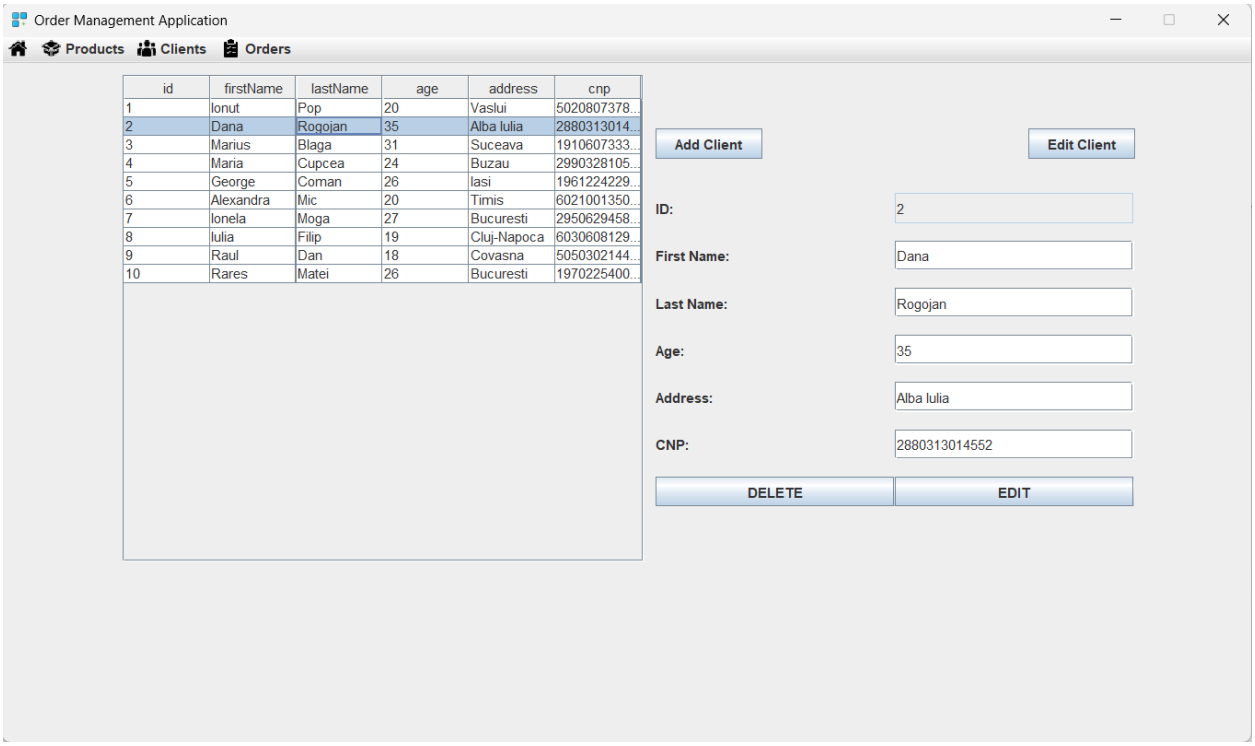
connection with the database. Data access package implements the objects for accessing the database. Validator package also contains a few helper classes which provides IDs for models and validates the user input.

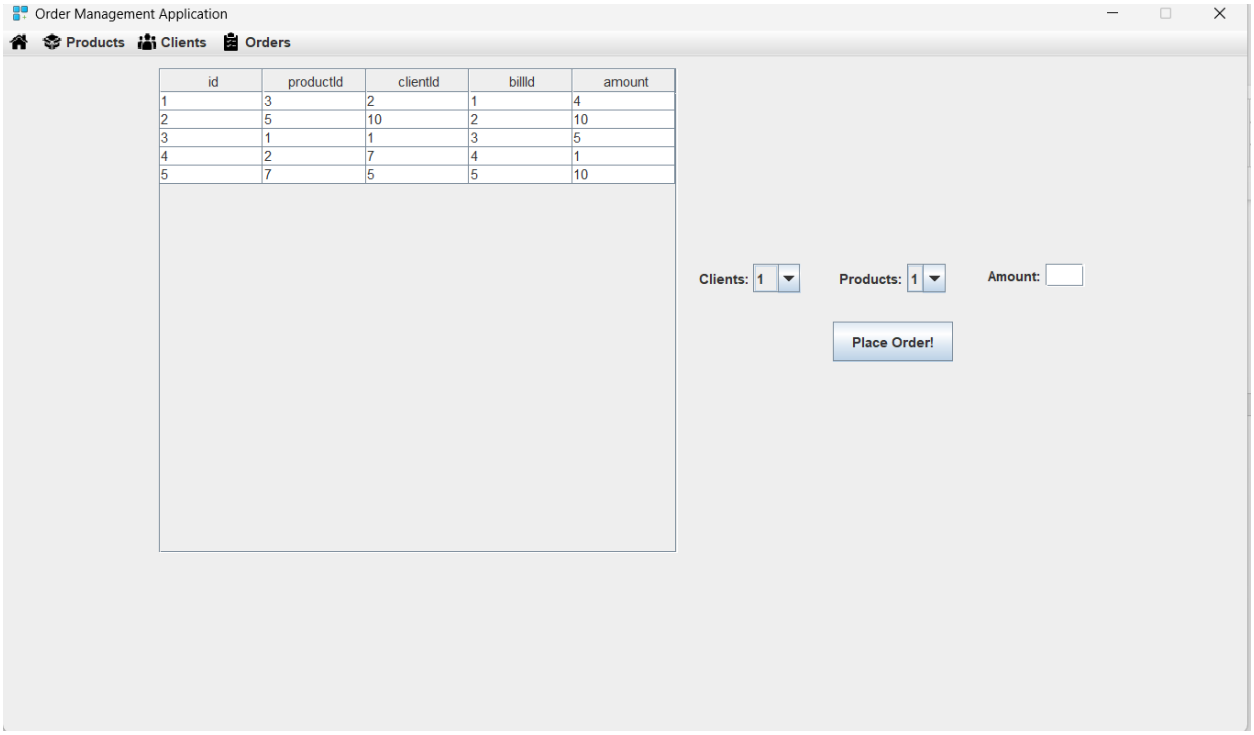
UML class diagram:



Graphical user interface:







4. IMPLEMENTATION

- ConnectionFactory class:

  It is a helper class for retrieving connections and control to the database. Implements a getter to a connection and a close connection, statement and result set.

- AbstractDAO class:

  Implements a various method for accessing the database using reflection techniques. It holds a type of variable for the T class. The main methods are finding an object by id, finding all the objects contained in a table, updating an object into the database, inserting and deleting objects into the database. It also implements a creating objects method which returns a list of objects.

```java
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) {
        e.printStackTrace();
    } finally {
        ConnectionFactory.closeStatement(statement);
        ConnectionFactory.closeResultSet(resultSet);
        ConnectionFactory.closeConnection(connection);
    }

    return null;
}
```

```
public void delete(long id) {
    Connection connection = null;
    PreparedStatement statement = null;
    String query = "DELETE FROM " +
            type.getSimpleName() +
            " WHERE id = ?";

    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.setLong( parameterIndex: 1, id);
        statement.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        ConnectionFactory.closeStatement(statement);
        ConnectionFactory.closeConnection(connection);
    }
}
```

- BillDAO class:

  Is the data access class of the Bill model.

- ProductDAO class:

  Is the data access class of the Product model. Extends AbstractDAO.

- ClientDAO class:

  Is the data access class of the Client. Extends AbstractDAO.

- OrderDAO class:

  Is the data access class of the Order. Extends AbstractDAO.

- BillBLL class:

  Implements the business logic of the Bill class.

- ProductBLL class:

  Implements the business logic of the Product class.

- OrderBLL class:

  Implements the business logic of the Order class.

- ClientBLL:

  Implements the business logic of the Client class

- Bill class:

  Models the bills.

- Product class:

  Models the products.

- ClientM class:

  Models the clients.

- OrderM class:

  Models the orders.

- ClientValidator class:

  Validates and provides IDs for new clients.

- BillValidator class:

  Validates and provides IDs for new bills.

- ProductValidator class:

  Validates and provides IDs for new products.

- OrderValidator class:

  Validates and provides IDs for new orders.

- View class:

  Implements the graphical user interface.

5. RESULTS

Besides the manual testing I exported SQL dumps for testing the application on a already generated database.

The source code includes JavaDoc.

6. CONCLUSIONS

- Conclusion:
  - The application is functional and can perform any required operation (inserting new clients/products and place orders).

- Learned skills:
  - I delved into Reflection and learned a new type of software architecture. I upskilled my abilities to organize my code.

7. BIBLIOGRAPHY

1. *Regex:* Tutorials List - Javatpoint
2. Stack Overflow - Where Developers Learn, Share, & Build Careers
3. diagrams.net
4. *www.baeldung.com/java-reflection#6-implemented-interfaces - Reflection*
5. *www.flaticon.com – Icons*
6. *www.baeldung.com*