DOCUMENTATION

ASSIGNMENT 3

STUDENT NAME: Boboș Răzvan
GROUP: 30425

CONTENTS

1. Assignment Objective

Main Objective

Consider an application Orders Management for processing client orders for a warehouse.

Sub-Objectives

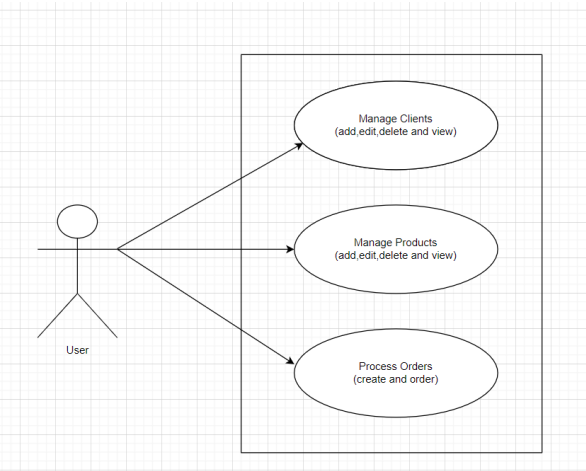
• Analyze the problem and identify requirements

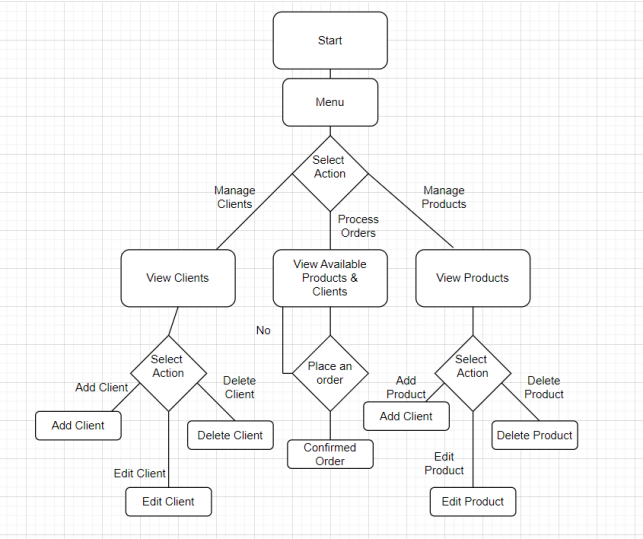
• Design the orders management application

• Implement the orders management application

• Test the orders management application

2. Problem Analysis, Modeling, Scenarios, Use Cases





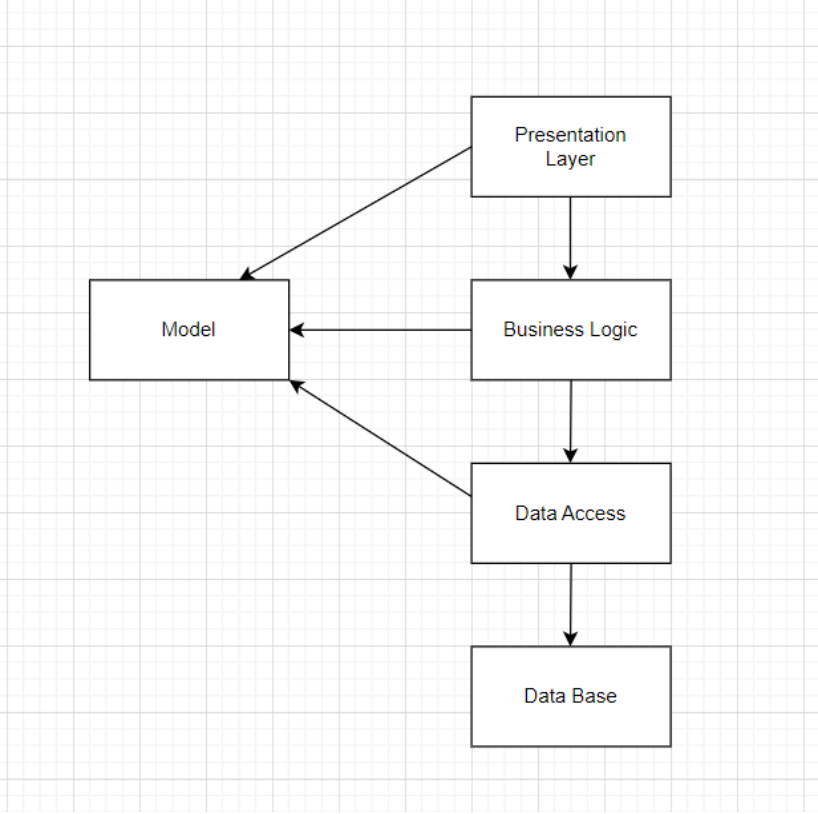
Functional Requirements

- The application should allow an employee to manage clients (add, edit, delete, view all clients in a table)
- The application should allow an employee to manage products (add, edit, delete, view all products in a table)
- The application should allow an user to create an order

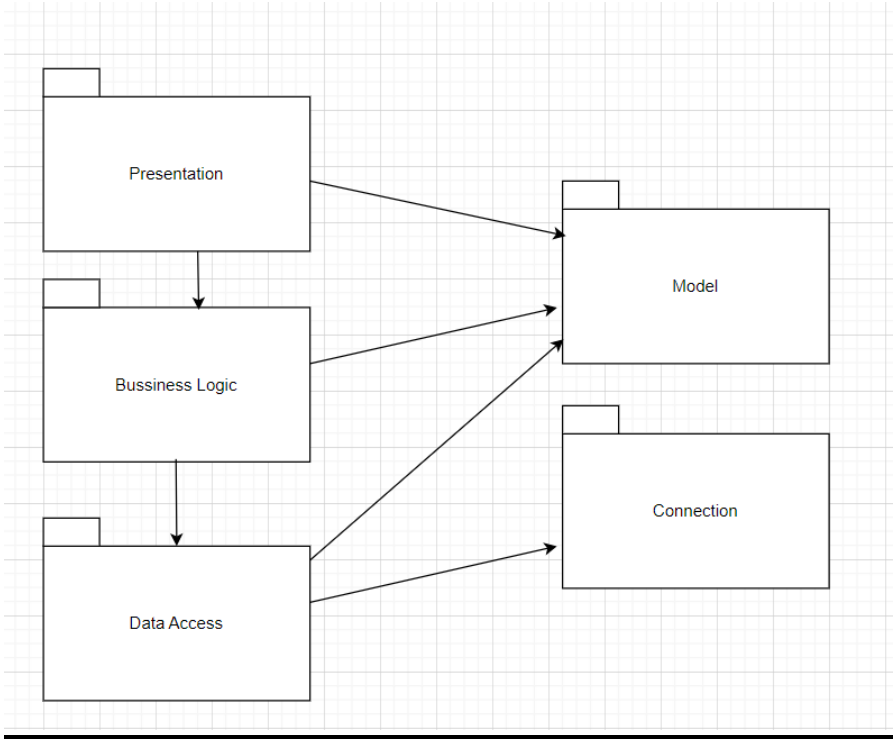
Non-Functional Requirements

- The application should be intuitive and easy to use by the user
- The application should provide tooltips and help guides to assist users in understanding how to use various features.
- The application should include a dashboard with an overview of log metrics (Bill Table)

3. Design



Division into packages



To design the object-oriented architecture of the orders management application, we can break down the functionality into several classes representing different components of the system. Here's a conceptual overview of the OOP design:

AbstractDAO.java

This class provides generic methods for CRUD operations on the database. It encapsulates common database operations such as querying, inserting, updating, and deleting records. It's generic to work with any type of object (e.g., Product, Client, etc.) by using Java Reflection (for queries) to handle object properties dynamically.

ClientDAO.java & ProductDAO.java & OrdersDAO.java

These classes represent Data Access Objects (DAOs) tailored for managing specific entities within the database, including Clients, Orders, and Products, respectively. Each DAO extends the AbstractDAO class, providing generic methods for CRUD operations while focusing on the management of their corresponding entity types.

BillDAO.java
This class serves as a Data Access Object (DAO) dedicated to managing Bill entities within the database.

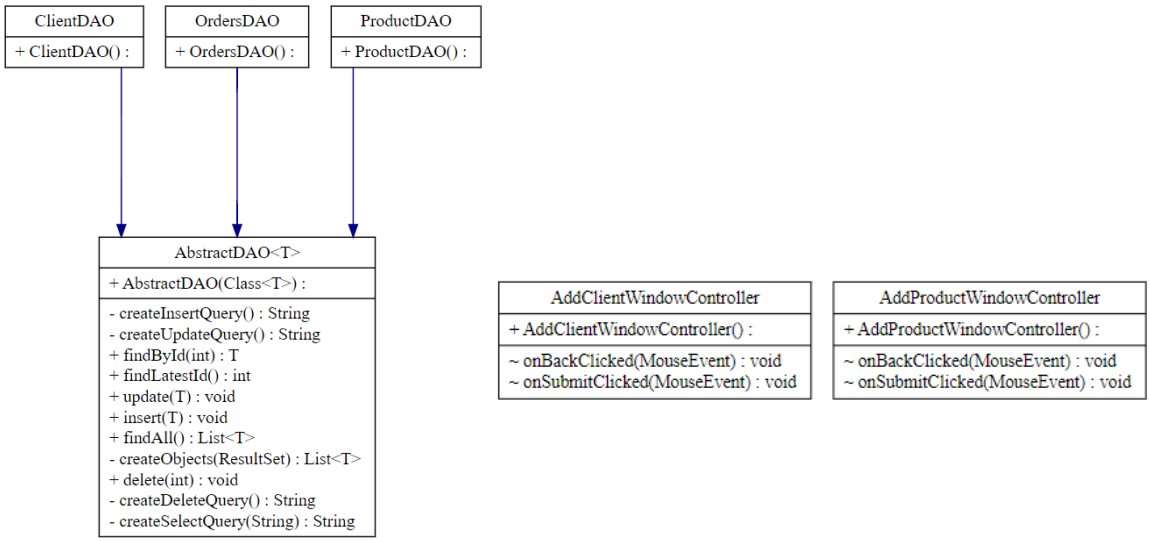
Client.java & Product.java & Orders.java & Bill.java
These classes serve as the fundamental models within the application's architecture, encapsulating various entities such as clients, orders, products, and bills, providing a structured representation of the application's core data.

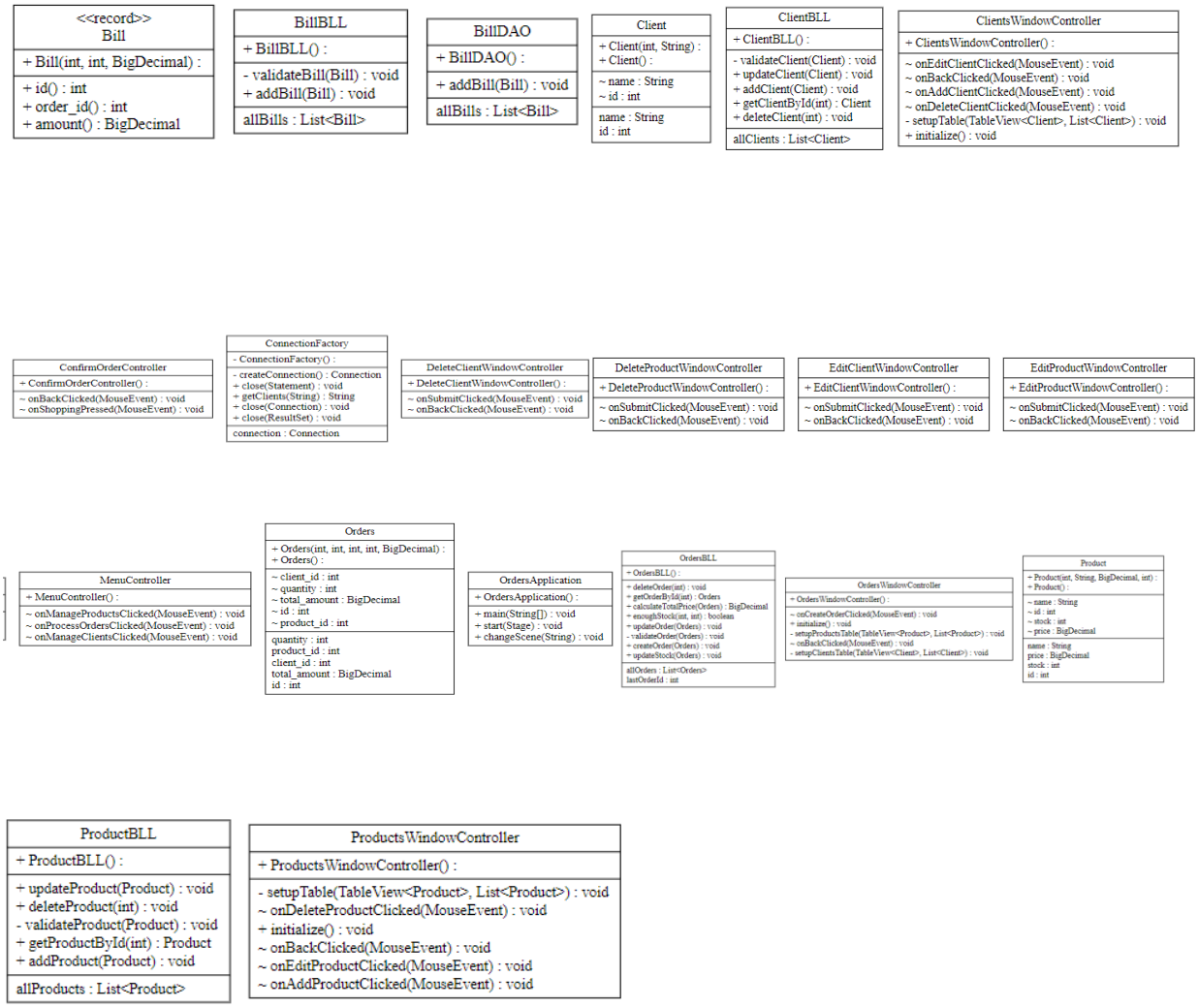
Controllers (MenuController.java, ProductsWindow.java, etc.)
The controllers in the FXML architecture act as intermediaries between the user interface defined in FXML files and the application logic, handling user interactions and updating the view accordingly.

ConnectionFactory.java
The ConnectionFactory class serves as a central hub for creating and managing database connections in the application, offering methods to obtain connections and close resources safely.

ClientBLL.java & OrdersBLL.java & ProductBLL.java & BillBLL.java
The BusinessLogic package encapsulates the core functionality of the application, with classes like ClientBLL, OrdersBLL, ProductBLL, and BillBLL handling operations related to clients, orders, products, and bills, respectively.

Here's how these classes might relate to each other in a basic **UML class diagram**:





4. Implementation

To comprehend the implementation thoroughly, we will delve into each class's implementation in detail, examining how they contribute to the overall functionality of the project.

AbstractDAO.java

The **insert(T object)** method in the **AbstractDAO** class inserts a new object of type T into the database, dynamically constructing an SQL INSERT query and binding the object's values to it, ensuring data integrity and consistency. The **findById(int id)** method in the AbstractDAO class retrieves an object of type T from the database based on its unique identifier.

```java
public T findById(int id) {
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    List<T> items = new ArrayList<>();
    String query = createSelectQuery( field: "id");
    try {
        statement = this.connection.prepareStatement(query);
        statement.setInt( parameterIndex: 1, id);
        resultSet = statement.executeQuery();
        items = createObjects(resultSet);
        if (!items.isEmpty()) {
            return items.get(0);
        }
    } catch (SQLException | InstantiationException | IllegalAccessException
        e.printStackTrace();
    } catch (NoSuchMethodException e) {
        throw new RuntimeException(e);
    } finally {
        ConnectionFactory.close(resultSet);
        ConnectionFactory.close(statement);

    }
    return null;
}
```

Product.java & Client.java & Orders.java

They use Setters and getters which are essential methods in the model classes of the application, facilitating the encapsulation of data.

```java
Razvan
public void setId(int id) { this.id = id; }

Razvan
public void setName(String name) { this.name = name; }

2 usages  Razvan
public void setPrice(BigDecimal price) { this.price = price; }

3 usages  Razvan
public void setStock(int stock) { this.stock = stock; }

Razvan
public int getId() { return id; }

Razvan
public String getName() { return name; }
```

ConnectionFactory.java

The **ConnectionFactory** class is implemented as a **Singleton**, ensuring that only one instance of it exists throughout the application's lifecycle.

```
1 usage
private static ConnectionFactory singleInstance = new ConnectionFactory();


1 usage  Razvan
private ConnectionFactory() {

    try {
        Class.forName( className: "org.postgresql.Driver");
    } catch (ClassNotFoundException e) {
        LOGGER.log(Level.SEVERE,  msg: "PostgreSQL JDBC Driver not found!", e);
    }
}
```

ClientsWindowController.java & ProductsWindowController.java

In these classes, reflection is utilized within the **setupTable** method to dynamically set up the columns of a JavaFX TableView based on the fields of the provided model class. By introspecting the fields of the model class, the method creates TableColumn instances.

```
1 usage  Razvan
private void setupTable(TableView<Client> table, List<Client> items) {
    if (items == null || items.isEmpty()) return;
    table.getColumns().clear();

    Field[] fields = items.get(0).getClass().getDeclaredFields();

    for (Field field : fields) {
        TableColumn<Client, String> column = new TableColumn<>(field.getName());
        column.setCellValueFactory(new PropertyValueFactory<>(field.getName()));
        table.getColumns().add(column);
    }
    table.setItems(FXCollections.observableArrayList(items));

}
```

ClientBLL.java & ProductBLL.java & OrdersBLL.java & BillBLL.java

The validate methods in the BLL (Business Logic Layer) classes ensure that the data being processed meets certain criteria or constraints before proceeding with operations.

```java
private void validateBill(Bill bill) {
    if (bill.amount().compareTo(BigDecimal.ZERO) < 0) {
        throw new IllegalArgumentException("Bill amount cannot be negative.");
    }
    if (bill.order_id() <= 0) {
        throw new IllegalArgumentException("Bill must be associated with a valid order.");
    }

}
```

.fxml Files

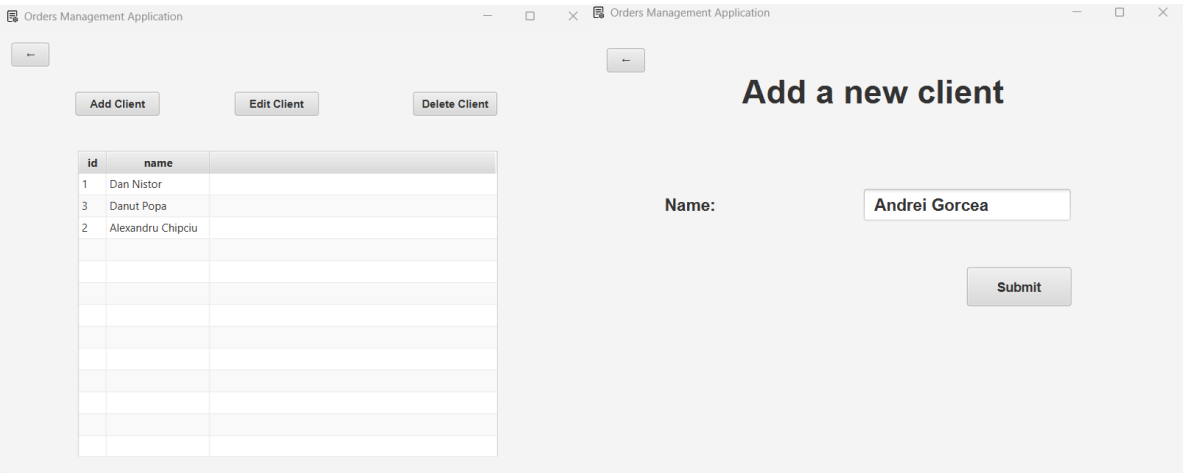
FXML files, like the one shown, define the structure and appearance of JavaFX user interfaces, while tools such as Scene Builder streamline the process by offering visual design capabilities and property editing functionalities.

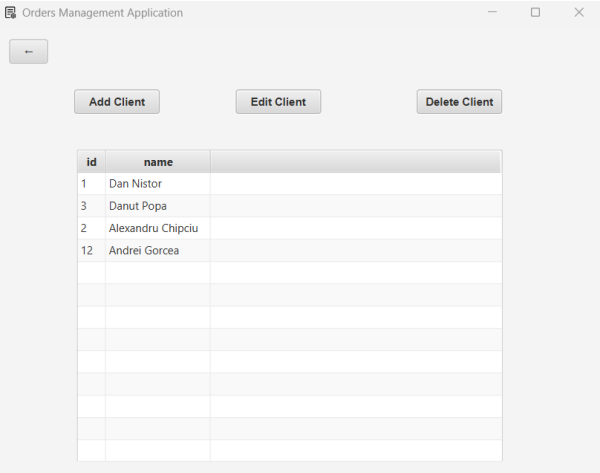
```
<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
  <children>
    <Button fx:id="manageClientsButton" layoutX="95.0" layoutY="102.0" mnemonicParsing="false"
      <font>
        <Font name="Arial Bold" size="14.0" />
      </font></Button>
    <Button fx:id="manageProductsButton" layoutX="95.0" layoutY="217.0" mnemonicParsing="false"
      <font>
        <Font name="Arial Bold" size="14.0" />
      </font></Button>
    <Button fx:id="processOrdersButton" layoutX="95.0" layoutY="322.0" mnemonicParsing="false"
      <font>
        <Font name="Arial Bold" size="14.0" />
      </font></Button>
    <Label layoutX="357.0" layoutY="54.0" prefHeight="216.0" prefWidth="182.0" text="Welcome to
      <font>
        <Font name="Arial Bold" size="24.0" />
      </font>
    </Label>
    <ImageView fitHeight="158.0" fitWidth="151.0" layoutX="363.0" layoutY="243.0">
```

5. Results

It is important to highlight here that every action involving product or client management dynamically updates the respective table. When a new product or client is added, it reflects in the table; likewise, when a client is edited or deleted, the table updates accordingly. Additionally, when creating an order, the stock for the specific product is automatically adjusted to reflect the quantity ordered, ensuring accurate inventory management.

Example: Adding a client





7. Conclusions

Conclusions: The project effectively employs a four-layered architecture to ensure separation of concerns across the user interface, business logic, data management, and database operations, enhancing maintainability and scalability. The use of reflection in the persistence layer promotes flexibility and reduces code redundancy by enabling dynamic database operations across various entity types.

Learnings: Through the project, I learned to modularize an application into a four-layer architecture, which significantly organizes the flow between user interaction, business processes, data manipulation, and database handling, ensuring a clean separation of responsibilities. I also gained insights into using reflection for dynamically generating SQL queries and handling ORM operations, alongside understanding the critical role of the Business Logic Layer (BLL) in encapsulating business rules and data validations.

Future Developments: For future developments, adding a dedicated bills table would enhance financial tracking and management within the system. Expanding the fields for clients, products, and orders can make these entities more specialized and tailored to specific business needs, such as addressing unique client requirements or product specifications. Additionally, enabling the creation of orders with multiple products would significantly enhance the system's capability to handle complex transactions, improving both usability and functionality for users.

8. Bibliography

- Fundamental PROGRAMMING TECHNIQUES LABORATORY GUIDE
- OOP – Marius Joldoș
- https://youtu.be/IZCwawKILsk?si=1a1T8uV8iof2JGQE
- https://www.youtube.com/@Randomcode_0
- https://stackoverflow.com/
- https://www.geeksforgeeks.org/
- https://youtu.be/mrU01yjxHAc?si=ZYxHauFaBf9CPnSN