DOCUMENTATION

ASSIGNMENT 3

STUDENT NAME: Stoica Sergiu
GROUP: 30226

CONTENTS

1. Assignment Objective

The main objective of this assignment is to develop an order management application aiming to manage the products, the clients, and the orders for a warehouse. We are supposed to generate a bill as soon as a new order is created.

Sub-objective	**Description**	**Documentation Section**
Analyze the problem and identify requirements	Identify functional and non-functional requirements for the queue management app.	Section 2: Problem Analysis, Scenario Modeling, Use Cases
Design the orders management application	Create an object-oriented design for the application including UML class and package diagrams.	Section 3: Design
Implement the orders management application	Develop the necessary classes and methods to handle more threads to run simultaneously.	Section 4: Implementation
Testing the orders management application	Conduct testing to ensure that the application behaves correctly.	Section 5: Results

2. Problem Analysis, Modeling, Scenarios, Use Cases

Functional Requirements:

- The order management application should allow users to select which view to select.
- The order management application should allow users to input data into the fields.
- The order management application should create, edit, or delete an entry from the database.
- The order management application should allow users to perform those operations on clients, products, and orders.
- The order management application should generate a bill after each new order is created.
- The order management application should display a table with all the data on the database, and it should update after any operation is performed.

Non - Functional Requirements:

- The order management application must run efficiently, ensuring all operations are processed without delay.
- The user interface of the application should be straightforward, allowing users to easily change between views and select which operation they wish to perform.

Use Cases:

- **Use Case: add operation.**
    - Actor: User
- **Success scenario:**
    1. The user inserts the data in the corresponding fields.
    2. The user clicks on the submit button.
    3. The application validates the data, making sure the ID is unique.
    4. The new record is stored in the database.
- Alternative scenario:
    1. The user inserts an ID that already exists in the database.
    2. The application displays an error message and requests the user to insert another ID.
    3. The scenario returns to step 1.

Use Cases:

- **Use Case: edit operation.**
    - Actor: User
- **Success scenario:**
    1. The user inserts the ID of the entry that should be modified and then the new values in the corresponding fields.
    2. The user clicks on the submit button.
    3. The application validates the data, making sure the ID exists.
    4. The updated record is stored in the database.
- Alternative scenario:
    1. The user inserts an ID that does not exist in the database.
    2. The application displays an error message and requests the user to insert a valid ID.
    3. The scenario returns to step 1.

Use Cases:

- **Use Case: delete operation.**
    - Actor: User
- **Success scenario:**
    1. The user inserts the ID of the entry that should be deleted in the corresponding field.
    2. The user clicks on the submit button.
    3. The application makes sure the ID exists.
    4. The record is deleted from the database.
- Alternative scenario:
    1. The user inserts an ID that does not exist in the database.
    2. The application displays an error message and requests the user to insert a valid ID.
    3. The scenario returns to step 1.

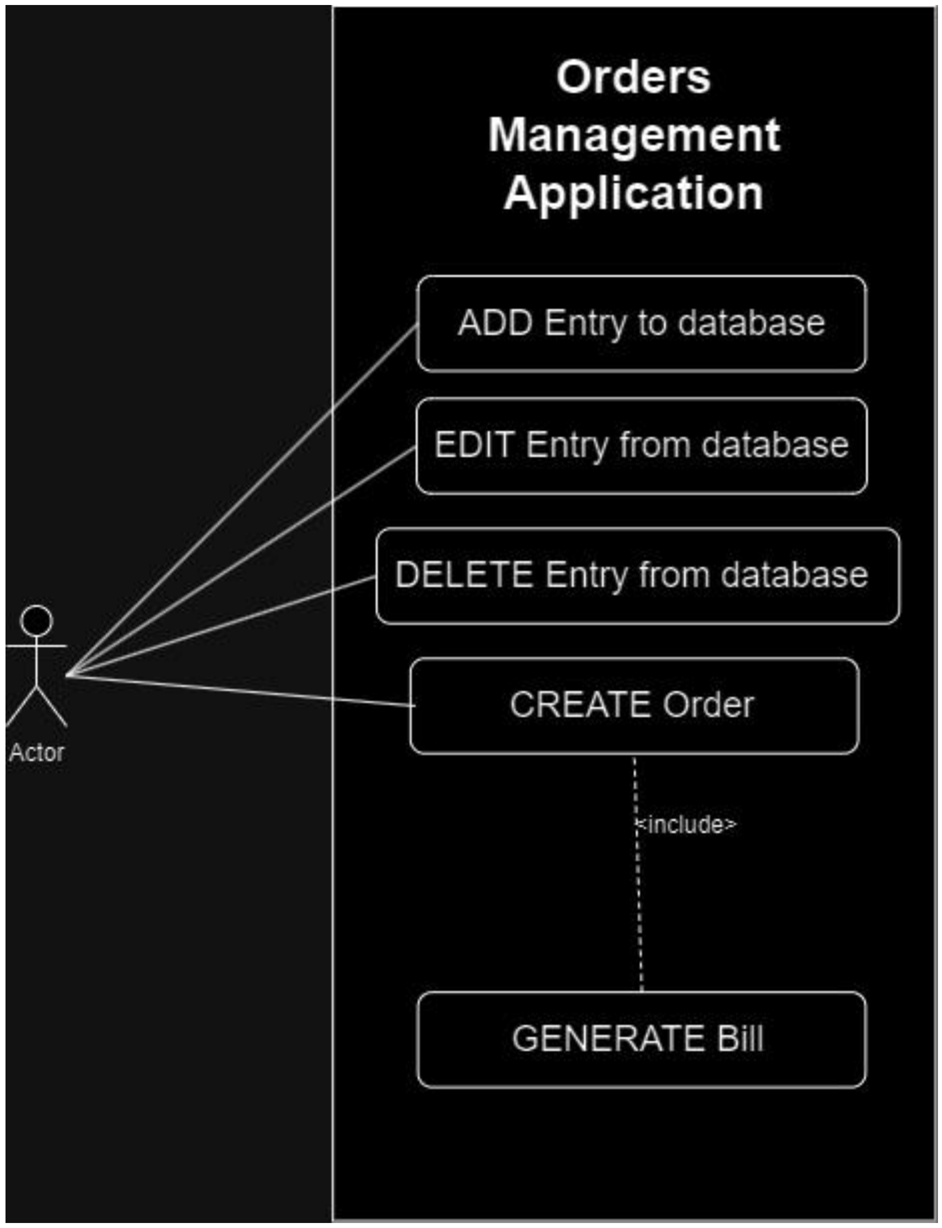
Use Cases:

- **Use Case: create order.**
    - Actor: User
- **Success scenario:**

1. The user inserts the data in the corresponding fields.
2. The user clicks on the submit button.
3. The application validates the data, making sure the IDs exist, and that the stock is enough.
4. The stock is decreased for the said product.
5. The order is saved to the database, and a bill is created.

- Alternative scenario:

1. The user inserts an ID that does not exist in the database, or a quantity that exceeds the stock of the product.
2. The application displays an error message and requests the user to insert a valid ID or a smaller quantity.
3. The scenario returns to step 1.


Orders Management Application
ADD Entry to database
EDIT Entry from database
DELETE Entry from database
CREATE Order
<include>
GENERATE Bill
Actor

3. Design

Conceptual Architecture

The conceptual architecture of the orders management application illustrates a layered software application. This architecture is designed to separate concerns, making the application easier to manage and scale.

Here’s a detailed description of each layer shown in the architecture:

Model

- **Purpose**: The Model layer contains classes that directly map to the database tables. This is where the application's domain logic resides, which includes data and the rules governing the access and modifications of this data.
- **Interaction**: The Model serves as the data structure backbone and interacts with the Business Logic layer, allowing it to retrieve, store, and update information in the database.

Presentation Layer

- **Purpose**: This layer contains classes and resources related to the user interface (UI) of the application. It is responsible for displaying data to the user and interpreting the user's actions such as clicks, keyboard input, and other forms of input.
- **Interaction**: The Presentation Layer sends commands to the Business Logic layer, depending on the user's interaction, and updates the UI based on changes in the data model.

Business Logic

- **Purpose**: The Business Logic layer contains the core computational logic of the application. It handles commands from the Presentation Layer, performs requested operations on the data via the Data Access layer, and returns the data/results back to the Presentation Layer.
- **Interaction**: This layer acts as an intermediary between the Presentation and Data Access layers, ensuring that user inputs are processed, and that data integrity is maintained.

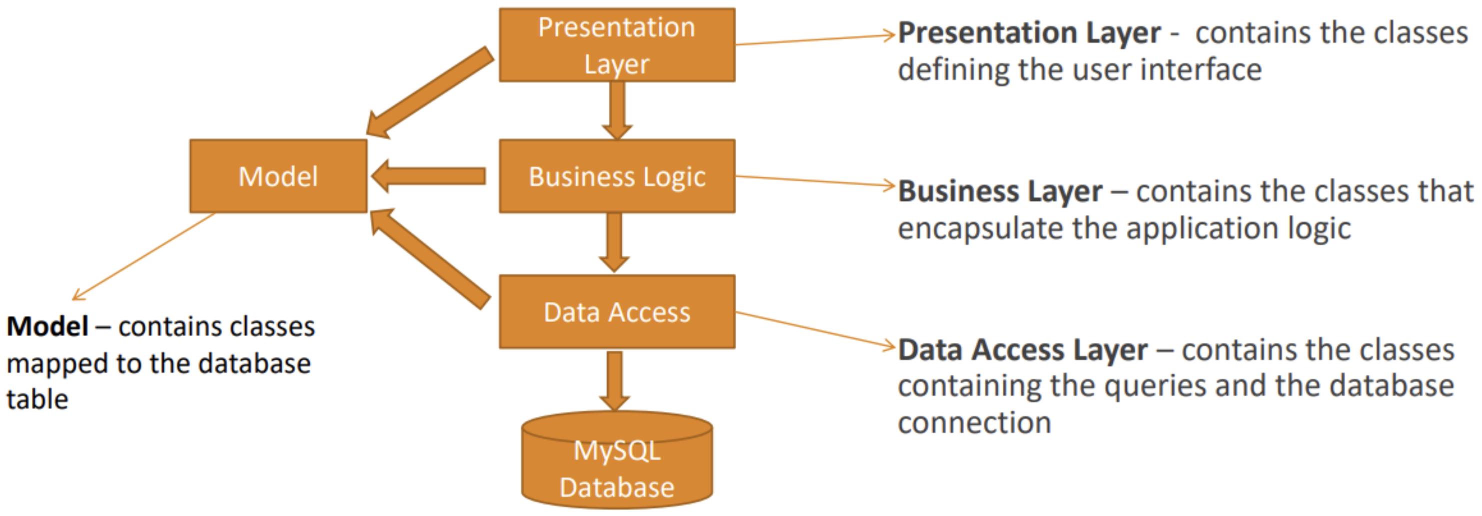
Data Access Layer

- **Purpose**: Also known as the persistence layer, this contains the logic to access the underlying database. It includes SQL queries, database connection management, and data transfer objects.
- **Interaction**: The Data Access Layer directly interacts with the MySQL Database to retrieve, update, and store information. It exposes various methods that the Business Logic layer can use to perform these operations.

MySQL Database

- **Purpose**: This is the storage subsystem of the application. It stores all the application data across various tables.
- **Interaction**: It interacts only with the Data Access Layer and is isolated from direct access by other layers to maintain security, scalability, and manageability.

This architecture is particularly beneficial for complex applications that require robustness, security, and scalability while maintaining a user-friendly interface.


Presentation Layer
Presentation Layer - contains the classes defining the user interface
Model
Business Logic
Business Layer – contains the classes that encapsulate the application logic
Data Access
Data Access Layer – contains the classes containing the queries and the database connection
Model – contains classes mapped to the database table
MySQL Database

Detailed Architecture - UML Package Diagram

Here’s a detailed description of each package shown in the architecture:

Presentation Layer

- **Component**: Main Menu
- **Relationships**:
    - Interacts directly with the Business Logic layer, specifically with services like **MainMenuService**, **ClientService**, **OrderService**, and **ProductService**. This layer sends user input to the Business Logic layer and receives data to display to the users.

Business Logic Layer

- **Components**: **ClientService**, **MainMenuService**, **OrderService**, **ProductService**, **TableUtil**
- **Relationships**:
    - Receives commands from the Presentation layer to perform specific business functions.
    - Uses the Data Access layer to retrieve, update, insert, and delete data in the database through DAOs like **ClientDAO**, **ProductDAO**, **OrderDAO**, etc.
    - May interact with the Model layer to pass and transform data into the format required for business operations or UI display.

Data Access Layer

- **Components**: **ClientDAO**, **ProductDAO**, **LogDAO**, **OrderDAO**, **GenericDAO**
- **Relationships**:
    - Communicates directly with the Model layer to query or persist objects like **Clients**, **Orders**, **Products**, and **Bills**.
    - Uses the Connection layer (**Database Connection**) to establish and manage database connections necessary for executing SQL commands.

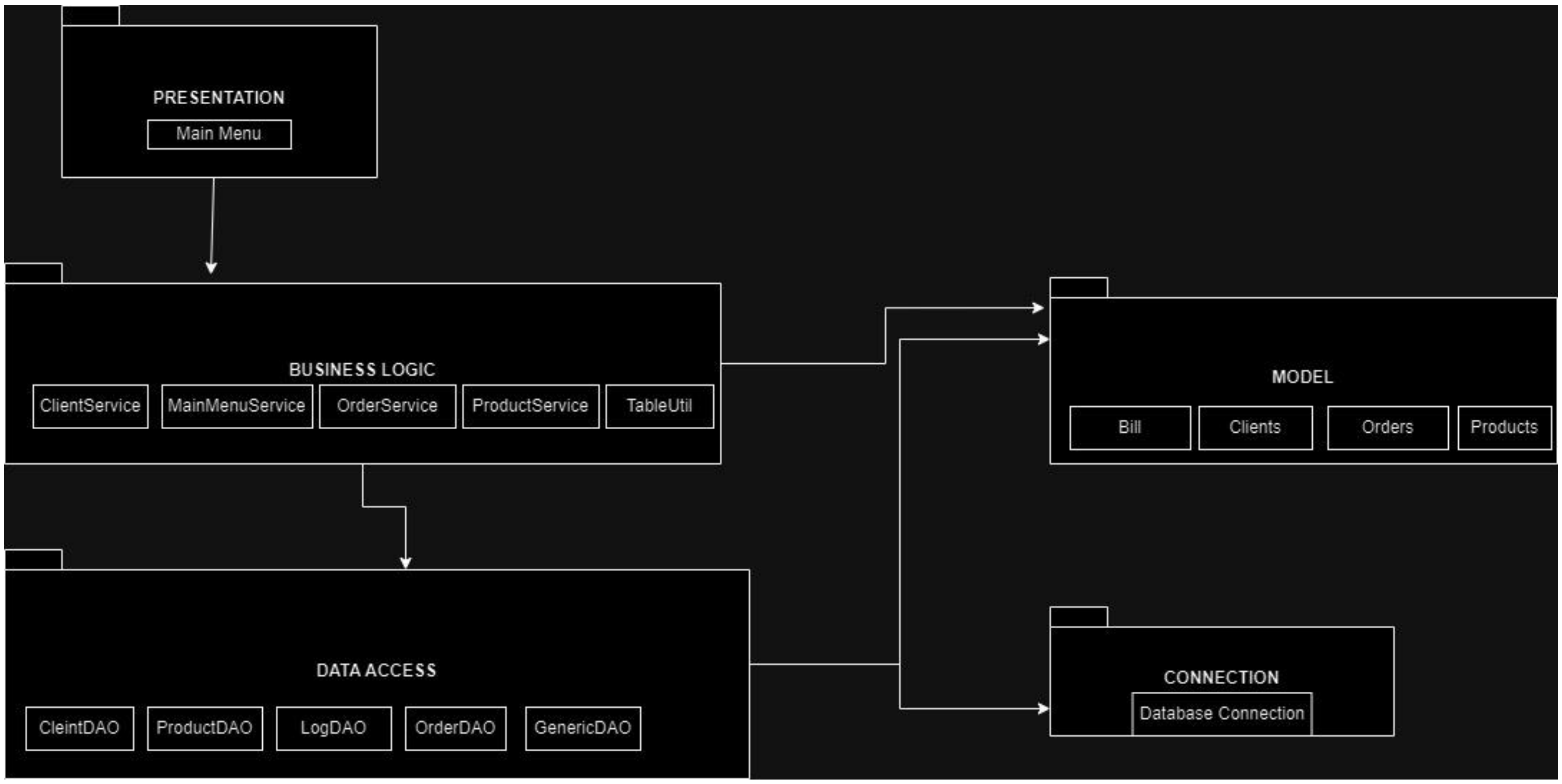
Model Layer

- **Components**: **Bill**, **Clients**, **Orders**, **Products**
- **Relationships**:

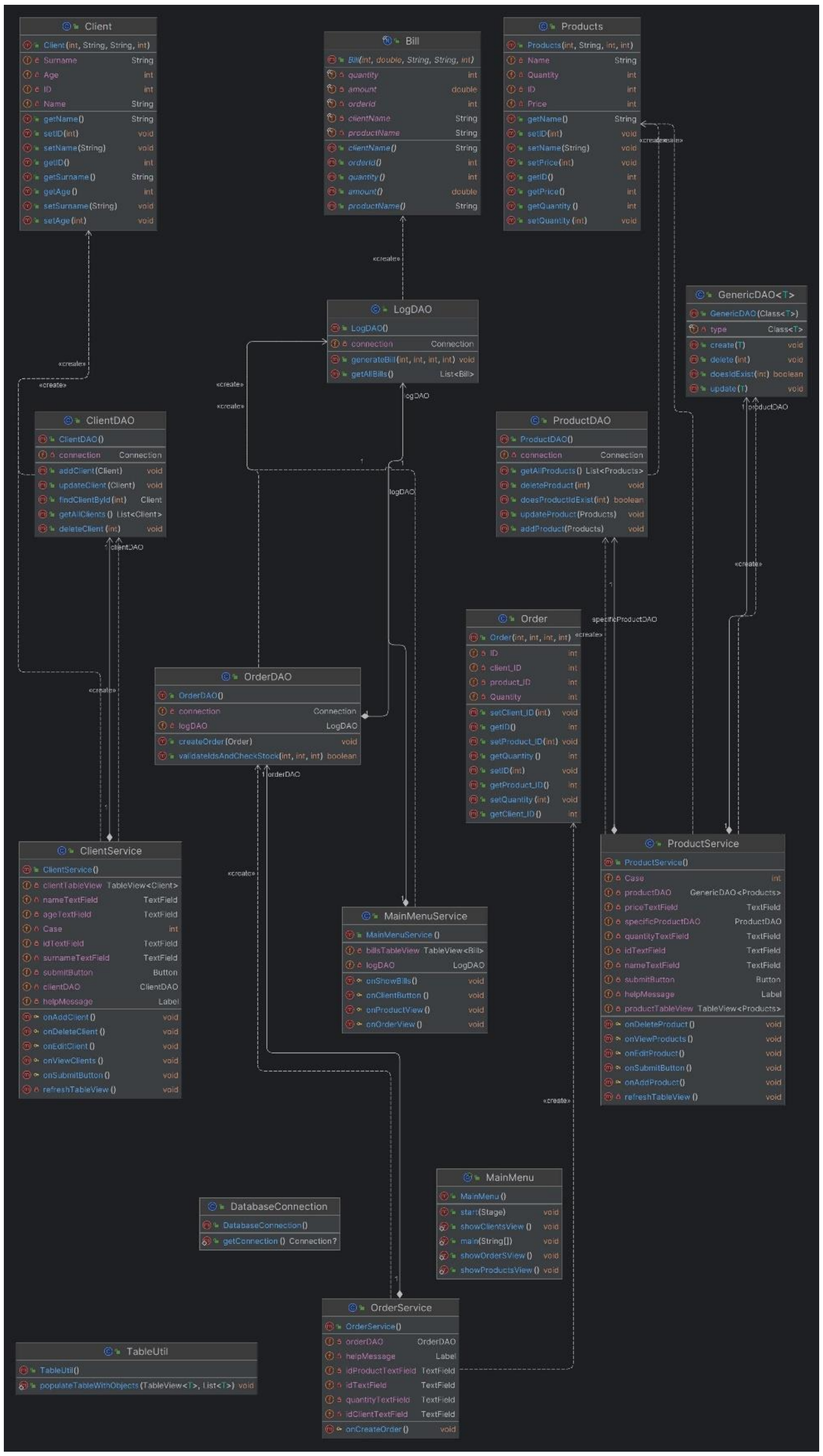
- Acts as the structure defining the business objects being manipulated and viewed. This layer is manipulated by the Data Access layer and may also be directly interacted with by the Business Logic layer for retrieving or storing data in memory during runtime.

Connection Layer

- **Component: Database Connection**
- **Relationships**:
    - Provides database connectivity to the Data Access layer. This is the only layer that directly interacts with the database, encapsulating all the configuration and mechanics of database interaction, thereby isolating the rest of the application from direct SQL database management tasks.



Class Diagram



4. Implementation

```java
public static <T> void populateTableWithObjects(TableView<T> table, List<T>
objects) {
    if (objects == null || objects.isEmpty()) {
        return;
    }

    Class<?> clazz = objects.get(0).getClass();
    Field[] fields = clazz.getDeclaredFields();

    table.getColumns().clear();

    for (Field field : fields) {
        field.setAccessible(true);
        TableColumn<T, Object> column = new TableColumn<>(field.getName());
        column.setCellValueFactory(cellData -> {
            try {
                return new
ReadOnlyObjectWrapper<>(field.get(cellData.getValue()));
            } catch (IllegalAccessException e) {
                e.printStackTrace();
                return null;
            }
        });
        table.getColumns().add(column);
    }

    ObservableList<T> data = FXCollections.observableArrayList(objects);
    table.setItems(data);
}
```

This method automates the process of populating a **TableView** with data. It dynamically creates columns based on the fields of the objects in the provided list and sets the values of those columns accordingly. This approach allows for a flexible and reusable way to display data in a table without manually defining the table columns and their corresponding values for each type of object.

```java
protected void onShowBills() throws IOException {
    System.out.println("Show Bills !");

    List<Bill> bills = new ArrayList<>();

    billsTableView.setVisible(true);

    try {
        bills = logDAO.getAllBills();
        System.out.println("All bills retrieved successfully!");

        TableUtil.populateTableWithObjects(billsTableView, bills);

    } catch (SQLException e) {
        System.err.println("Error retrieving clients from the database.");
        e.printStackTrace();
    }
}
```

This method effectively bridges the data from the backend (database) to the frontend (UI) by retrieving the necessary information and dynamically updating the user interface.

```java
public void create(T object) throws SQLException {
    StringBuilder sql = new StringBuilder("INSERT INTO ");
    sql.append(type.getSimpleName()).append(" (");
    Field[] fields = type.getDeclaredFields();

    for (Field field : fields) {
        sql.append(field.getName()).append(",");
    }

    sql.setLength(sql.length() - 1);
    sql.append(") VALUES (");

    for (int i = 0; i < fields.length; i++) {
        sql.append("?,");
    }

    sql.setLength(sql.length() - 1);
    sql.append(")");

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement ps = conn.prepareStatement(sql.toString())) {

        for (int i = 0; i < fields.length; i++) {
            fields[i].setAccessible(true);
            ps.setObject(i + 1, fields[i].get(object));
        }

        ps.executeUpdate();
    } catch (IllegalAccessException e) {
        throw new RuntimeException(e);
    }
}
```

The **create** method dynamically constructs an SQL **INSERT** statement to insert an object of a generic type **T** into a database. It:

1. Constructs the SQL statement based on the fields of the object's class.
2. Sets up a database connection and prepares the statement.
3. Sets the parameters of the **PreparedStatement** with the values from the object's fields.
4. Executes the statement to insert the data into the database.
5. Handles exceptions that might occur during reflection or SQL operations.

```java
public void update(T object) throws SQLException {
    StringBuilder sql = new StringBuilder("UPDATE ");
    sql.append(type.getSimpleName()).append(" SET ");
    Field[] fields = type.getDeclaredFields();

    for (Field field : fields) {
        sql.append(field.getName()).append(" = ?,");
    }

    sql.setLength(sql.length() - 1);
    sql.append(" WHERE id = ?");

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement ps = conn.prepareStatement(sql.toString())) {

        for (int i = 0; i < fields.length; i++) {
            fields[i].setAccessible(true);
            ps.setObject(i + 1, fields[i].get(object));
        }

        Field idField = type.getDeclaredField("ID");
        idField.setAccessible(true);
        ps.setObject(fields.length + 1, idField.get(object));

        ps.executeUpdate();
    } catch (IllegalAccessException | NoSuchFieldException e) {
        throw new RuntimeException(e);
    }
}
```

The **update** method dynamically constructs an SQL **UPDATE** statement to update an object of a generic type **T** in a database. It:

1. Constructs the SQL statement based on the fields of the object's class.
2. Sets up a database connection and prepares the statement.
3. Sets the parameters of the **PreparedStatement** with the new values from the object's fields.
4. Sets the **id** field to ensure the correct record is updated.
5. Executes the statement to update the data in the database.
6. Handles exceptions that might occur during reflection or SQL operations.

```java
public boolean doesIdExist(int id) throws SQLException {
    Connection connection = DatabaseConnection.getConnection();
    String query = "SELECT COUNT(id) FROM " + type.getSimpleName() + " WHERE
id = ?";
    PreparedStatement statement = connection.prepareStatement(query);
    statement.setInt(1, id);
    ResultSet resultSet = statement.executeQuery();
    if (resultSet.next()) {
        return resultSet.getInt(1) > 0;
    }
    return false;
}
```

The **doesIdExist** method performs the following actions to determine if a specific **id** exists in a table:

1. Establishes a connection to the database.
2. Constructs an SQL **SELECT** query to count records with the specified **id**.
3. Prepares the SQL statement with the constructed query.
4. Sets the **id** parameter in the prepared statement.
5. Executes the query and processes the result set to determine if the **id** exists.

```java
public boolean validateIdsAndCheckStock(int clientId, int productId, int
quantity) throws SQLException {
    String checkClientQuery = "SELECT COUNT(1) FROM clients WHERE id = ?";
    String checkProductQuery = "SELECT quantity FROM products WHERE id = ?";
    String updateProductQuery = "UPDATE products SET quantity = quantity - ?
WHERE id = ? AND quantity >= ?";

    try (PreparedStatement checkClientStmt =
connection.prepareStatement(checkClientQuery);
         PreparedStatement checkProductStmt =
connection.prepareStatement(checkProductQuery);
         PreparedStatement updateProductStmt =
connection.prepareStatement(updateProductQuery)) {

        connection.setAutoCommit(false);

        checkClientStmt.setInt(1, clientId);
        ResultSet clientResult = checkClientStmt.executeQuery();
        if (!clientResult.next() || clientResult.getInt(1) == 0) {
            connection.rollback();
            return false;
        }

        // Check if product ID exists and has enough stock
        checkProductStmt.setInt(1, productId);
        ResultSet productResult = checkProductStmt.executeQuery();
        if (productResult.next() && productResult.getInt("quantity") >=
quantity) {
            // Update stock
            updateProductStmt.setInt(1, quantity);
            updateProductStmt.setInt(2, productId);
            updateProductStmt.setInt(3, quantity);
            int affectedRows = updateProductStmt.executeUpdate();
            if (affectedRows == 1) {
                connection.commit();
                return true;
            }
        }
        connection.rollback();
        return false;
    } catch (SQLException e) {
        connection.rollback();
        throw e;
    } finally {
        connection.setAutoCommit(true);
    }
}
```

The **validateIdsAndCheckStock** method performs the following actions:

1. Starts a transaction by disabling auto-commit.
2. Validates the existence of the client ID.
3. Validates the existence of the product ID and checks if there is enough stock.
4. Updates the product stock if both validations pass.
5. Commits the transaction if successful, otherwise rolls back.
6. Handles exceptions and ensures that the connection's auto-commit mode is reset.

```
public void generateBill(int orderId, int clientId, int productId, int
quantity) throws SQLException {
    String query = "SELECT p.name AS product_name, p.price, c.name AS
client_name " +
            "FROM products p, clients c WHERE p.id = ? AND c.id = ?";
    PreparedStatement stmt = connection.prepareStatement(query);
    stmt.setInt(1, productId);
    stmt.setInt(2, clientId);
    ResultSet rs = stmt.executeQuery();
    if (rs.next()) {
        String productName = rs.getString("product_name");
        double price = rs.getDouble("price");
        String clientName = rs.getString("client_name");
        double amount = price * quantity;

        String insertLogQuery = "INSERT INTO Log (order_id, amount,
client_name, product_name, quantity) VALUES (?, ?, ?, ?, ?)";
        PreparedStatement logStmt =
connection.prepareStatement(insertLogQuery);
        logStmt.setInt(1, orderId);
        logStmt.setDouble(2, amount);
        logStmt.setString(3, clientName);
        logStmt.setString(4, productName);
        logStmt.setInt(5, quantity);
        logStmt.executeUpdate();
        logStmt.close();
    }
    stmt.close();
}
```

The **generateBill** method performs the following actions:

1. Constructs an SQL query to retrieve the product name, price, and client name for the specified **productId** and **clientId**.
2. Executes the query and processes the result set to extract the required information.
3. Calculates the total amount based on the product price and quantity.
4. Constructs an SQL **INSERT** query to log the bill details into the **Log** table.
5. Sets the parameters and executes the insert statement to log the details.
6. Closes the prepared statements to release resources.

5. Results

The order management application successfully achieved its objectives by implementing key functionalities across various components. The main outcomes of the project are as follows:

1. **Order Processing**:
    - Enabled the creation, updating, deletion, and viewing of orders.
    - Integrated automatic bill generation upon order creation.
2. **Client and Product Management**:
    - Implemented features to add, update, delete, and view client and product information.
    - Ensured accurate inventory tracking and management.
3. **Dynamic User Interface**:
    - Developed a user-friendly interface using JavaFX, allowing dynamic table generation and real-time data updates.
    - Utilized reflection to populate table columns based on object fields, enhancing flexibility and reusability.
4. **Database Integration**:
    - Successfully connected the application to a MySQL database, ensuring robust data handling.
    - Utilized a Data Access Layer (DAO) for efficient and secure CRUD operations.
5. **Transaction Management**:
    - Ensured data integrity through effective transaction management, particularly during complex operations like order processing and stock updates.

Overall, the project delivered a robust and scalable order management system, providing a solid foundation for future enhancements and integrations. The application demonstrates strong modularity, making it maintainable and extensible for evolving business needs.

6. Conclusions

The development of the order management application has been a comprehensive and enlightening experience. Throughout this project, several key functionalities were implemented, and numerous technical and conceptual lessons were learned. This section summarizes the achievements, insights gained, and future enhancements that could be considered.

Lessons Learned

1. **Importance of Modularity**:
    - Learned the value of a modular architecture, which made the application easier to develop, test, and maintain.
    - By separating concerns into different layers (Presentation, Business Logic, Data Access), it was possible to isolate and address issues more effectively.
2. **Working with JavaFX**:
    - Gained experience in creating dynamic and interactive user interfaces using JavaFX.
    - Learned how to bind data to UI components, making use of properties and observable lists to handle real-time data updates.
3. **Database Management**:
    - Understood the intricacies of database connections, transaction management, and the importance of prepared statements to prevent SQL injection.
    - Learned how to efficiently perform CRUD operations and manage relationships between different entities (e.g., clients, products, orders).
4. **Error Handling and Robustness**:
    - Implemented comprehensive error handling strategies to ensure the application remains stable under various conditions.
    - Gained insights into the importance of rolling back transactions in case of errors to maintain data integrity.

5. **Reflection and Dynamic Code**:
    - Utilized Java Reflection to dynamically handle different types of objects, making the application more flexible and extensible.
    - Learned how to use reflection to access object fields and methods, and the associated challenges and performance considerations.

The project successfully demonstrated the importance of a well-structured architecture, effective data management, and a user-centric approach to design. The lessons learned and the foundation laid by this project will serve as a steppingstone for future developments and improvements. The application is well-positioned to evolve and adapt to meet future requirements and challenges.

7. Bibliography

Connect to MySQL from a Java application.

- https://www.baeldung.com/java-jdbc
- http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/

Layered architectures.

- https://dzone.com/articles/layers-standard-enterprise

Reflection in Java

- http://tutorials.jenkov.com/java-reflection/index.html

Creating PDF files in Java

- https://www.baeldung.com/java-pdf-creation

JAVADOC

- https://www.baeldung.com/javadoc

SQL dump file generation

- https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html