DOCUMENTATION

ASSIGNMENT *3*

STUDENT NAME: Tianu Cosmin-Nicolae
GROUP: 30425

CONTENTS

1. Assignment Objective

Design and implement an application for managing the client orders for a warehouse.

I. ***Analyze the problem and identify requirements.***

II. ***Design the the orders management application.***

III. ***Implement the orders management application.***

IV. ***Test the orders management application.***

2. Problem Analysis, Modeling, Scenarios, Use Cases

*The **warehouse management application** has the following **functional requirements**. It should enable the user to manage client orders efficiently by providing functionality for **viewing, adding, updating, and removing clients,** as well as generating **bills** for their **orders**. Additionally, the application should allow the user to **view the current inventory, add or remove products, and update product details.** The user can also **view the orders and bills** generated.*

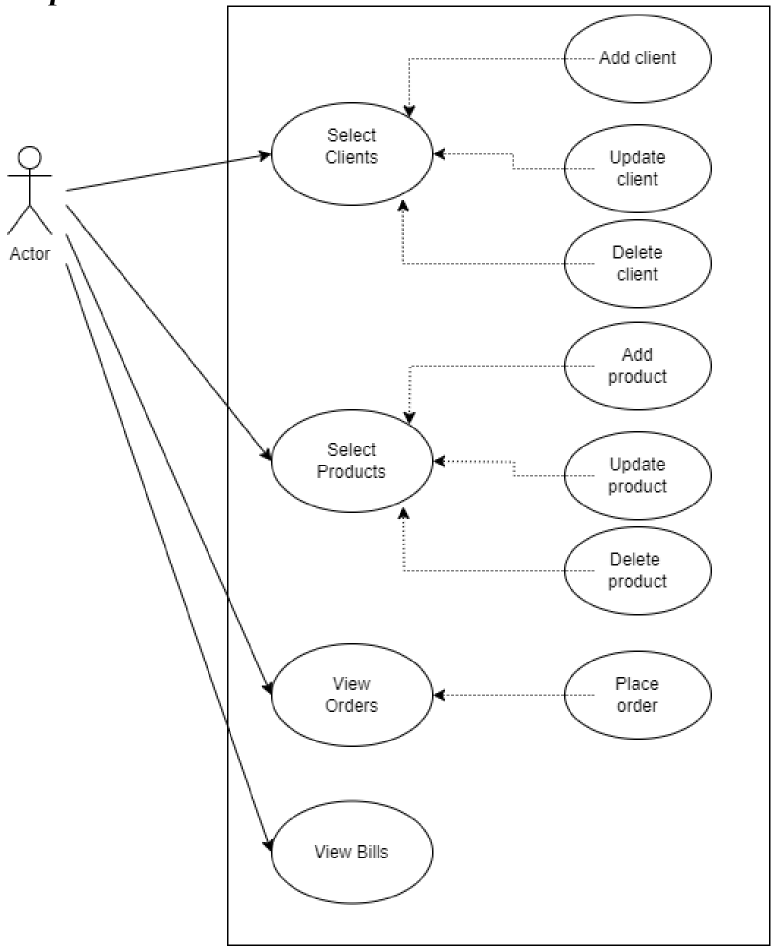


Figure 1: Use case diagram.

*As for the **non-functional requirements,** the order management app of an warehouse should be easy and intuitive to use by the user.*

The uses cases of the warehouse order management application are as follows:

- ***Client:***
  - ***Add a client.** The user representing an admin of the warehouse can add new clients by entering their name, email address and age.*
  - ***Update a client's information**. The user can modify all of a client's information based on its id from the database. If the user does not exist, it must be created first of all.*
  - ***Delete a client.** The admin can delete a current client by using their id from the database.*
  - ***View all the available clients.** The warehouse manager can see all the clients in the database until this moment.*
- ***Product:***
  - ***Add a product.** The user can add new products by entering the name, price and stock of said products.*
  - ***Update a product's information**. The user can modify a product's information based on its id from the database.*
  - ***Delete a client.** The admin can delete a current product by using their id from the database.*
  - ***View all the available clients.** The warehouse manager can see all the products in the database until this moment, together with their stock and price.*
- ***Orders and Bill***
  - ***Create orders.** The admin of the warehouse can create orders based on the available clients and of the products that are present in stock. **A bill** will be generated automatically for each order that consists of only one product. The user can also see both of the orders and bills in the database.*

3. Design

The warehouse management application follows a 4-layer architecture, consisting of the following layers:

***Data Access Layer (DAO):** Responsible for handling data persistence and database operations. Uses DAO design pattern to abstract the interaction with the database. Utilizes DAO classes to perform CRUD (Create, Read, Update, Delete) operations on entities.*

***Model Layer:** Represents the domain model or business entities of the application. Consists of classes representing clients, products, orders, bills. Contains business logic methods specific to each entity and their own constructors, getters and setters.*

***Business Logic Layer (BLL):** Acts as an intermediary between the data access layer and the presentation layer. Implements business logic and rules governing the interactions between different entities. Ensures data integrity, consistency, and validation before data is persisted in the database.*

***Presentation Layer**: Handles the user interface and interaction with the end-user. Utilizes Swing library to create graphical user interface (GUI) elements such as buttons, tables, and frames. Responds to user inputs, triggers appropriate actions, and updates the view based on changes in the model.*

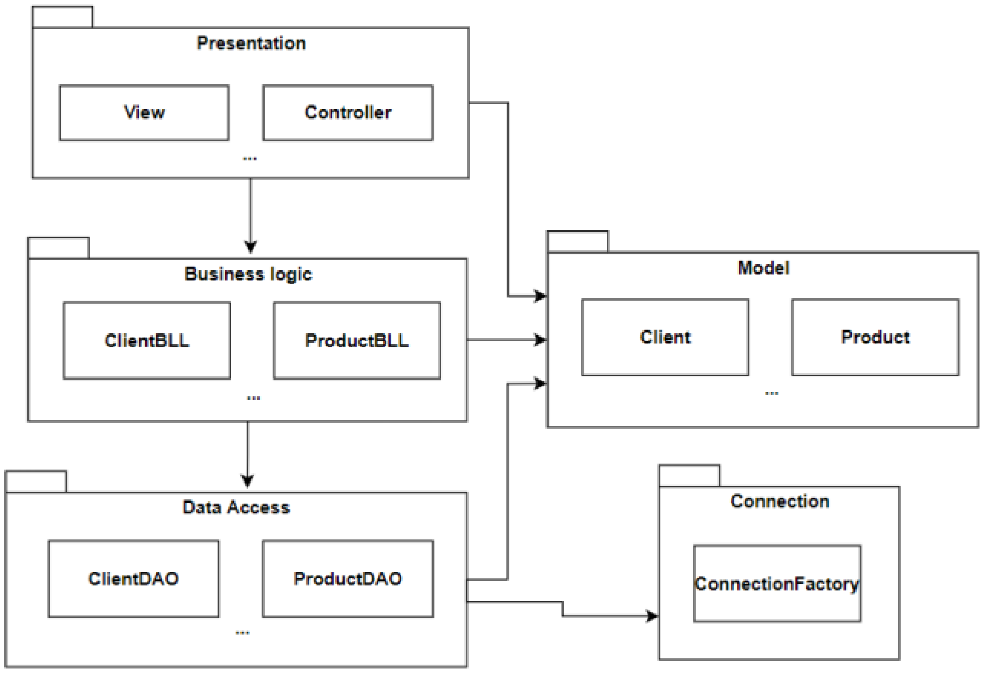


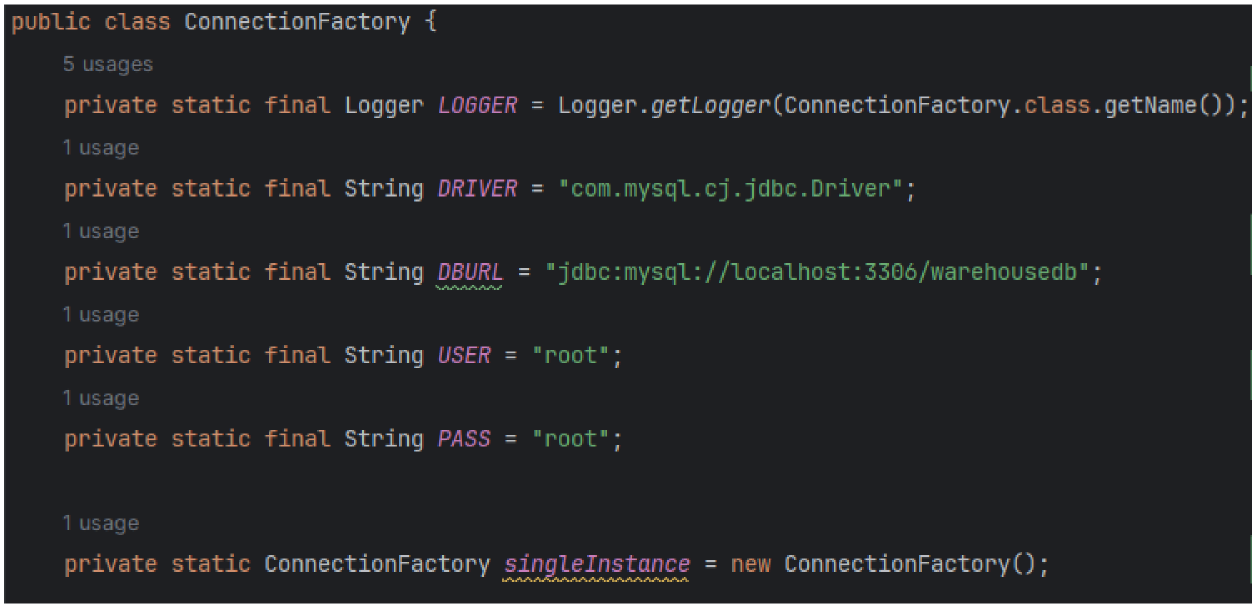
Figure 2: UML package diagram

*The architecture promotes separation of concerns and **modularity**, allowing each layer to focus on its specific **responsibilities**. This design facilitates easier **maintenance**, testing, and **scalability** of the warehouse management application.*

4. Implementation

*The **ConnectionFactory** class provides methods for establishing, managing, and closing database connections in a Java application. Let's discuss the responsibilities and functionality of this class:*

- *Establishes database connections using JDBC (Java Database Connectivity).*
- *Provides methods to create, retrieve, and close database connections, SQL statements, and result sets.*
- *Handles exceptions related to database connection and resource management.*

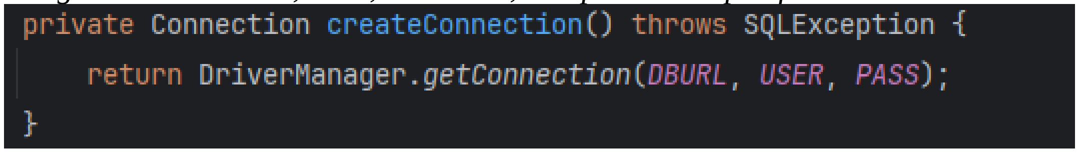


```java
public class ConnectionFactory {
    5 usages
    private static final Logger LOGGER = Logger.getLogger(ConnectionFactory.class.getName());
    1 usage
    private static final String DRIVER = "com.mysql.cj.jdbc.Driver";
    1 usage
    private static final String DBURL = "jdbc:mysql://localhost:3306/warehousedb";
    1 usage
    private static final String USER = "root";
    1 usage
    private static final String PASS = "root";

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

Figure 3: Connection details

Functionality:

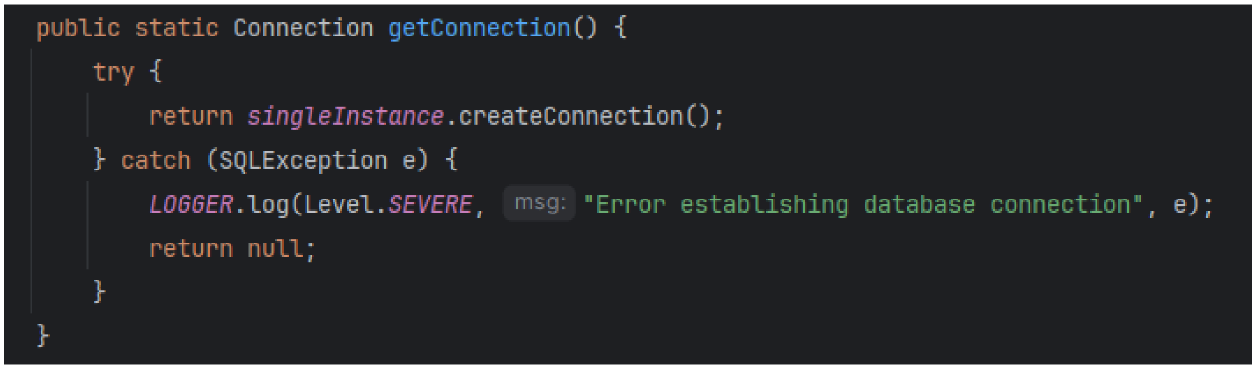
- ***createConnection():*** *Private method responsible for creating a new database connection using the JDBC driver, URL, username, and password specified in the class constants.*



```java
private Connection createConnection() throws SQLException {
    return DriverManager.getConnection(DBURL, USER, PASS);
}
```

Figure 4: method example

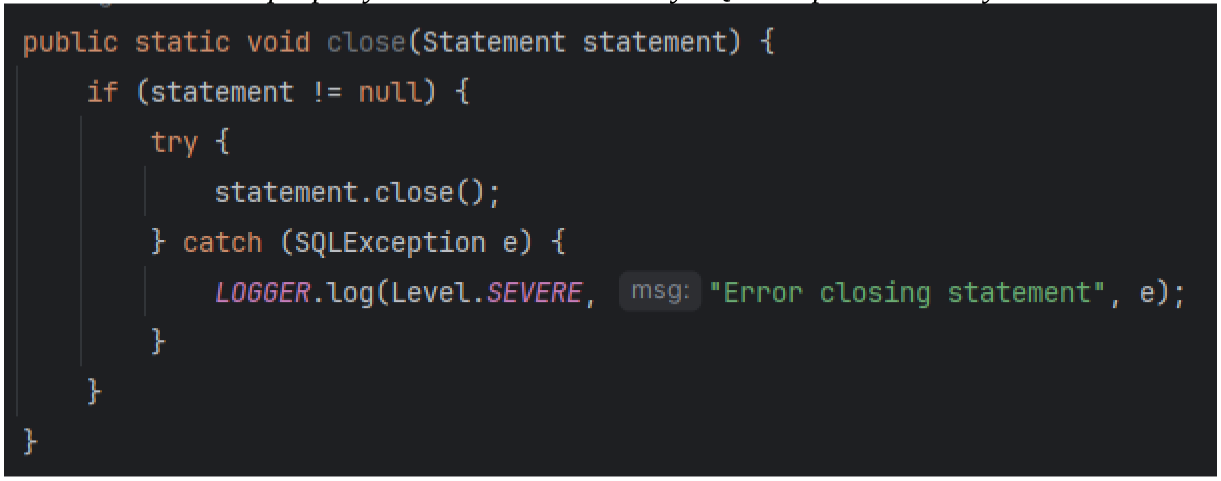
- ***getConnection():*** *Public static method that clients can call to retrieve a database connection. It delegates the creation of the connection to the private createConnection() method*



```java
public static Connection getConnection() {
    try {
        return singleInstance.createConnection();
    } catch (SQLException e) {
        LOGGER.log(Level.SEVERE, msg: "Error establishing database connection", e);
        return null;
    }
}
```

-
- ***close(Connection connection):*** *Public static method for closing a database connection. It handles any SQLException that may occur during the closing process.*

- ***close(Statement statement):*** *Public static method for closing a SQL statement. It ensures that the statement is properly closed and handles any SQLException that may occur.*



```
public static void close(Statement statement) {
    if (statement != null) {
        try {
            statement.close();
        } catch (SQLException e) {
            LOGGER.log(Level.SEVERE, msg: "Error closing statement", e);
        }
    }
}
```

Figure 5: method example

- ***close(ResultSet resultSet):*** *Public static method for closing a ResultSet. It ensures that the result set is properly closed and handles any SQLException that may occur.*

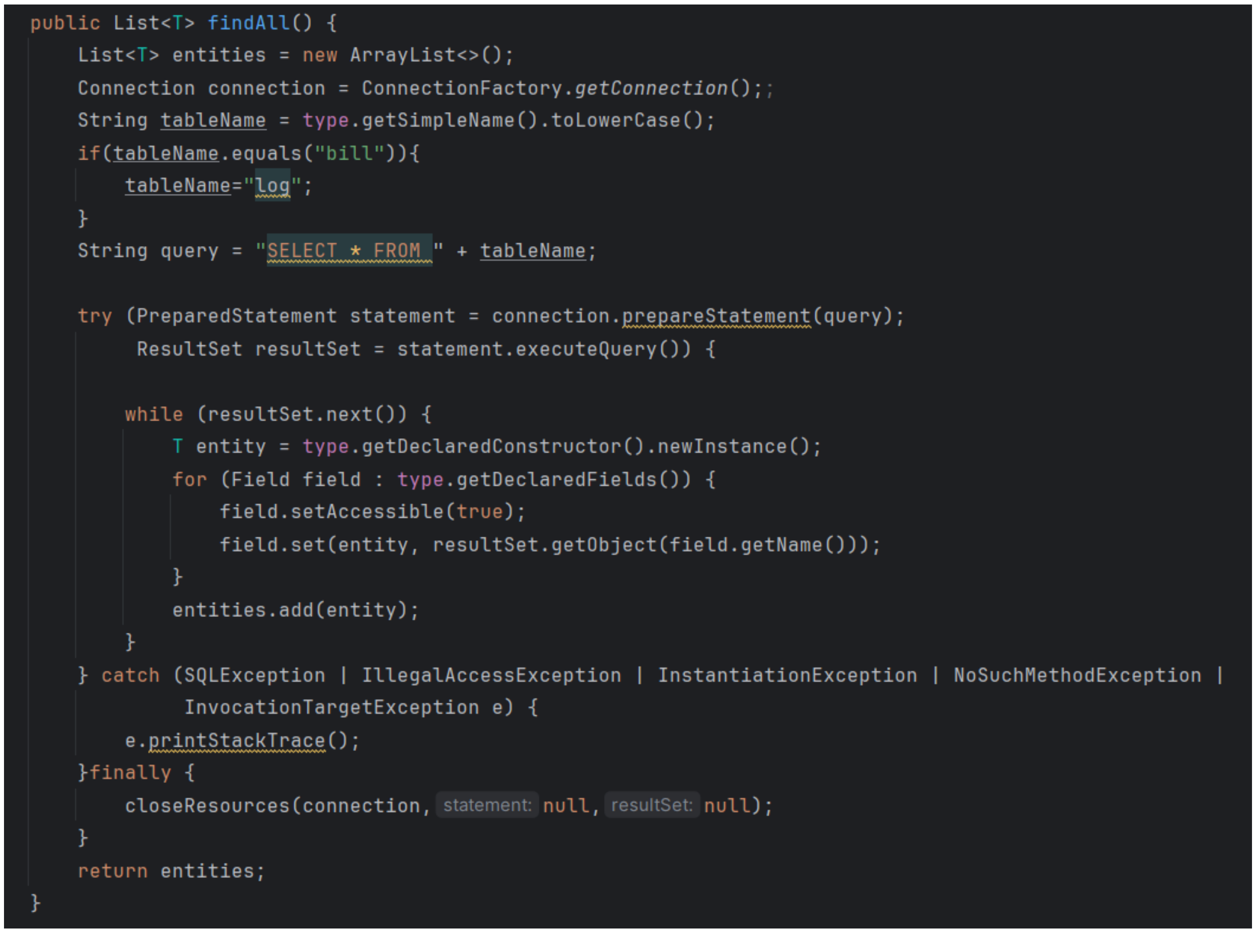
Additional Information:

- *The class follows the Singleton design pattern, ensuring that **only one** instance of ConnectionFactory is created.*

*In the **DAO (Data Access Object)** layer, the AbstractDao class serves as a **foundational** class providing common methods and functionality for interacting with the database. Let's discuss its role and how other DAO classes extend it:*

AbstractDao Class:

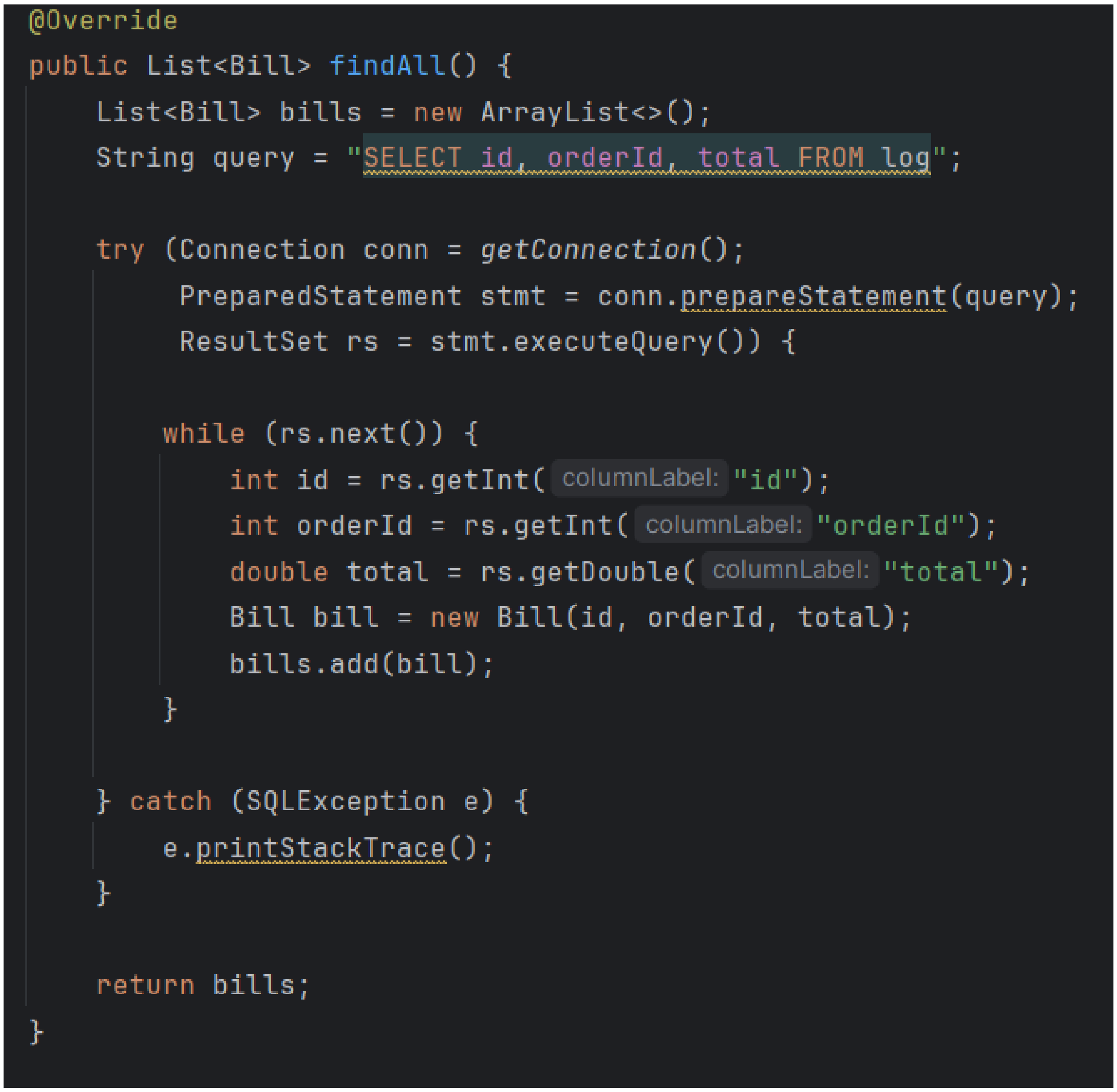
- *Abstracts away common database operations such as **CRUD** (Create, Read, Update, Delete) operations.*



```java
public List<T> findAll() {
    List<T> entities = new ArrayList<>();
    Connection connection = ConnectionFactory.getConnection();;
    String tableName = type.getSimpleName().toLowerCase();
    if(tableName.equals("bill")){
        tableName="log";
    }
    String query = "SELECT * FROM " + tableName;

    try (PreparedStatement statement = connection.prepareStatement(query);
         ResultSet resultSet = statement.executeQuery()) {

        while (resultSet.next()) {
            T entity = type.getDeclaredConstructor().newInstance();
            for (Field field : type.getDeclaredFields()) {
                field.setAccessible(true);
                field.set(entity, resultSet.getObject(field.getName()));
            }
            entities.add(entity);
        }
    } catch (SQLException | IllegalAccessException | InstantiationException | NoSuchMethodException |
             InvocationTargetException e) {
        e.printStackTrace();
    }finally {
        closeResources(connection, statement: null, resultSet: null);
    }
    return entities;
}
```

Figure 6: method example

- *Provides methods for executing **SQL** queries and handling database connections.*
- *Encapsulates the interaction with database entities and shields clients from low-level database details.*
- ***Majority** DAO **classes** in the application extend the AbstractDao class to **inherit** its common functionality and structure.*
- *By extending AbstractDao, specific DAO classes can focus on implementing entity-specific logic and custom queries without **duplicating** boilerplate code for database interaction.*
- ***Only the bill** DAO class overrides abstract methods defined in AbstractDao to provide entity-specific implementations such as reading it from database, it's model being a record.*



```java
@Override
public List<Bill> findAll() {
    List<Bill> bills = new ArrayList<>();
    String query = "SELECT id, orderId, total FROM log";

    try (Connection conn = getConnection();
         PreparedStatement stmt = conn.prepareStatement(query);
         ResultSet rs = stmt.executeQuery()) {

        while (rs.next()) {
            int id = rs.getInt( columnLabel: "id");
            int orderId = rs.getInt( columnLabel: "orderId");
            double total = rs.getDouble( columnLabel: "total");
            Bill bill = new Bill(id, orderId, total);
            bills.add(bill);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }

    return bills;
}
```

Figure 7: method example

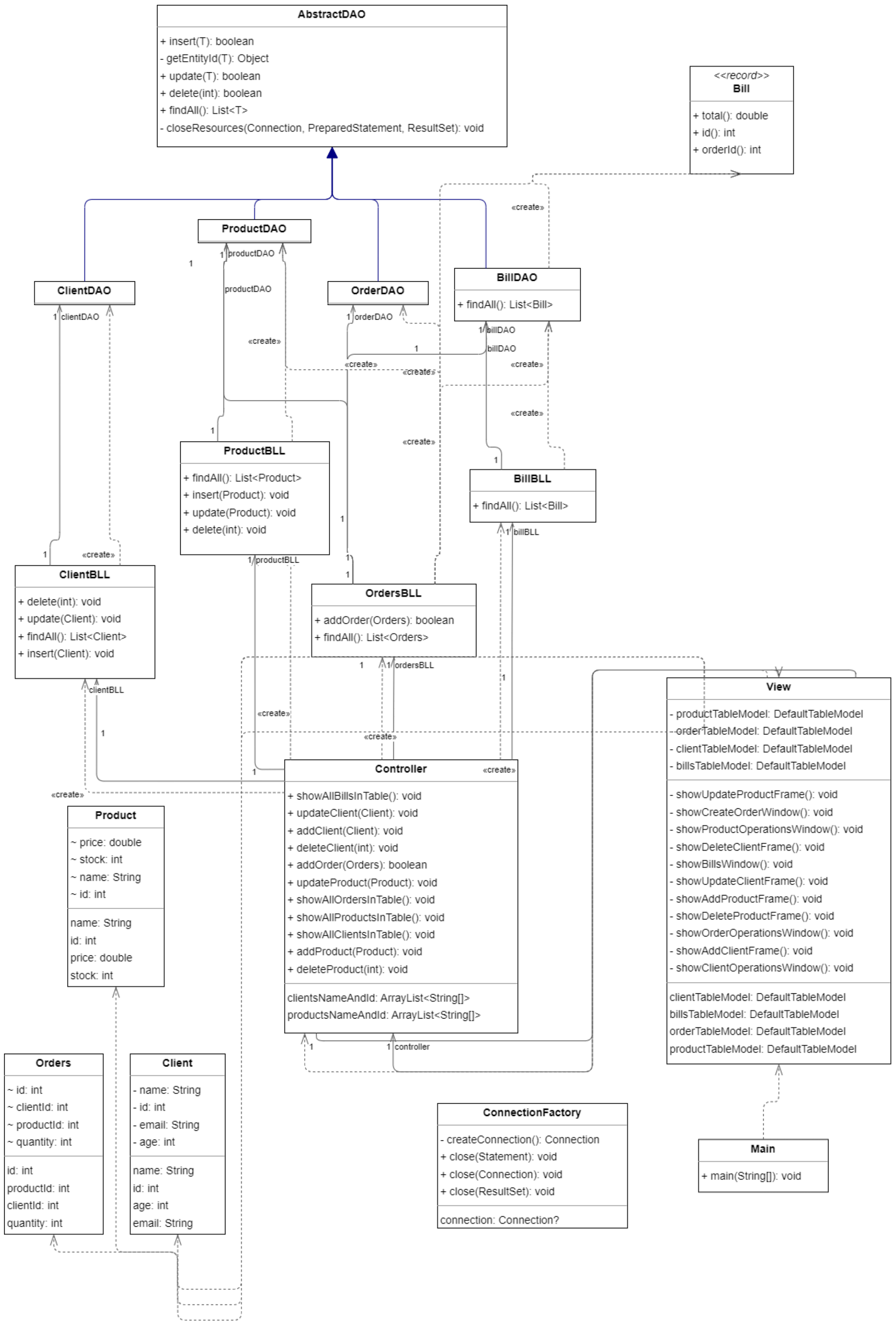
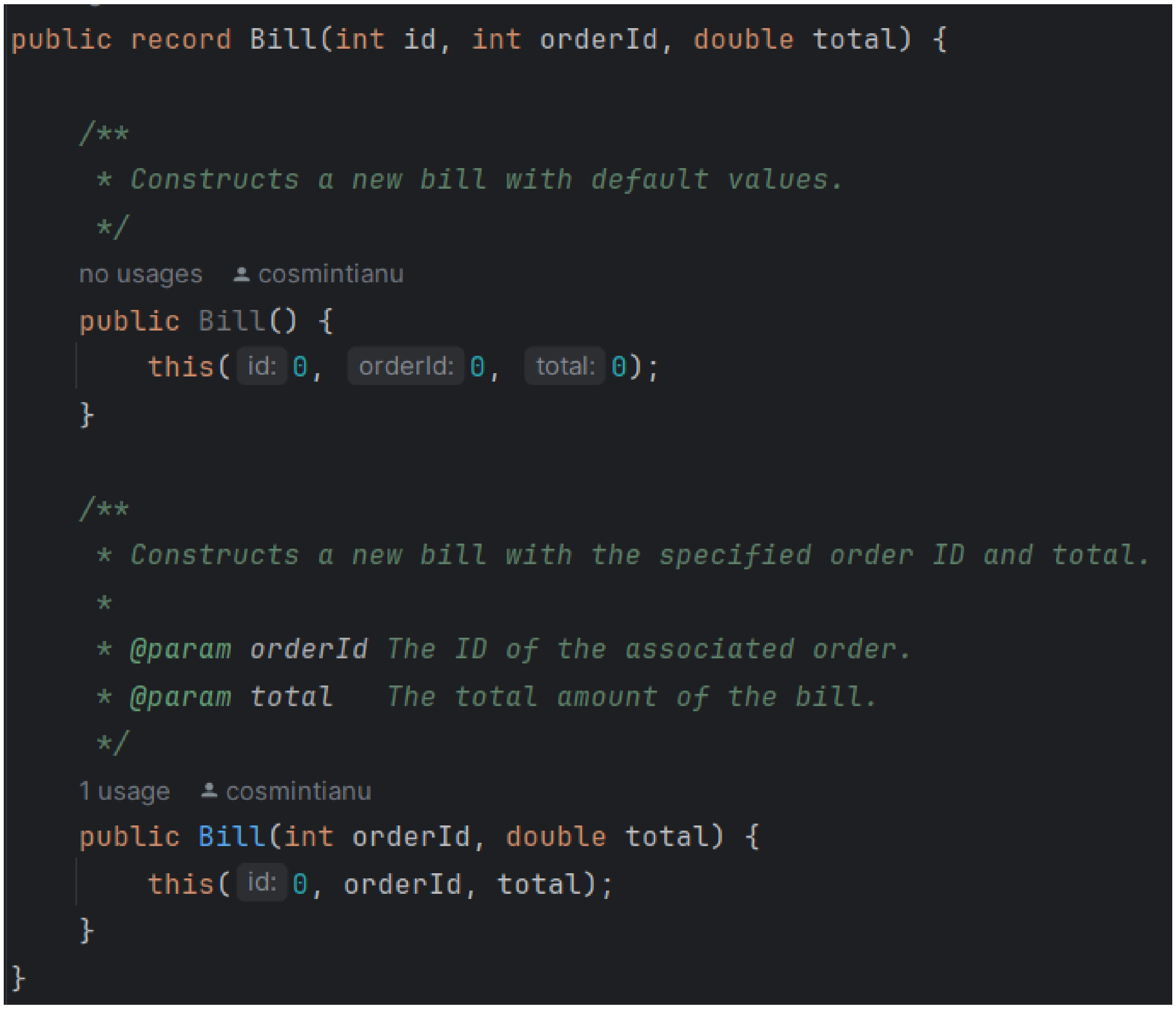

AbstractDAO
+ insert(T): boolean
- getEntityId(T): Object
+ update(T): boolean
+ delete(int): boolean
+ findAll(): List<T>
- closeResources(Connection, PreparedStatement, ResultSet): void
<<record>>
Bill
+ total(): double
+ id(): int
+ orderId(): int
ProductDAO
ClientDAO
OrderDAO
BillDAO
+ findAll(): List<Bill>
ProductBLL
+ findAll(): List<Product>
+ insert(Product): void
+ update(Product): void
+ delete(int): void
BillBLL
+ findAll(): List<Bill>
ClientBLL
+ delete(int): void
+ update(Client): void
+ findAll(): List<Client>
+ insert(Client): void
OrdersBLL
+ addOrder(Orders): boolean
+ findAll(): List<Orders>
View
- productTableModel: DefaultTableModel
- orderTableModel: DefaultTableModel
- clientTableModel: DefaultTableModel
- billsTableModel: DefaultTableModel
- showUpdateProductFrame(): void
- showCreateOrderWindow(): void
- showProductOperationsWindow(): void
- showDeleteClientFrame(): void
- showBillsWindow(): void
- showUpdateClientFrame(): void
- showAddProductFrame(): void
- showDeleteProductFrame(): void
- showOrderOperationsWindow(): void
- showAddClientFrame(): void
- showClientOperationsWindow(): void
clientTableModel: DefaultTableModel
billsTableModel: DefaultTableModel
orderTableModel: DefaultTableModel
productTableModel: DefaultTableModel
Controller
+ showAllBillsInTable(): void
+ updateClient(Client): void
+ addClient(Client): void
+ deleteClient(int): void
+ addOrder(Orders): boolean
+ updateProduct(Product): void
+ showAllOrdersInTable(): void
+ showAllProductsInTable(): void
+ showAllClientsInTable(): void
+ addProduct(Product): void
+ deleteProduct(int): void
clientsNameAndId: ArrayList<String[]>
productsNameAndId: ArrayList<String[]>
Product
~ price: double
~ stock: int
~ name: String
~ id: int
name: String
id: int
price: double
stock: int
Orders
~ id: int
~ clientId: int
~ productId: int
~ quantity: int
id: int
productId: int
clientId: int
quantity: int
Client
- name: String
- id: int
- email: String
- age: int
name: String
id: int
age: int
email: String
ConnectionFactory
- createConnection(): Connection
+ close(Statement): void
+ close(Connection): void
+ close(ResultSet): void
connection: Connection?
Main
+ main(String[]): void
productDAO
clientDAO
orderDAO
billDAO
productBLL
clientBLL
ordersBLL
billBLL
controller
«create»
1

Figure 8: UML Class diagram.

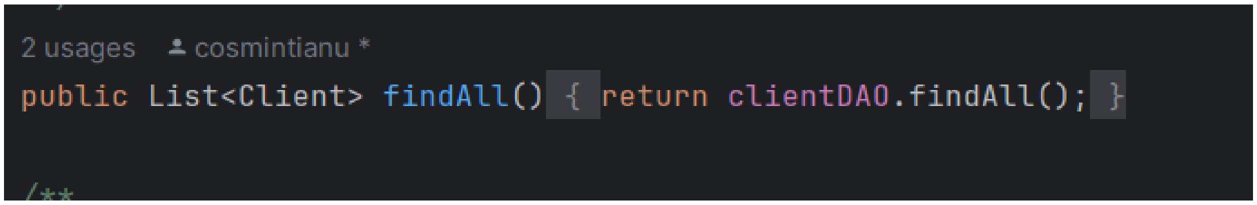
In the ***model layer****, the Bill class represents a record of a bill in the system. It encapsulates information about a bill, such as its ID, client ID, total amount, and date.*
The model classes serve as ***plain Java objects*** *(POJOs) that hold data and provide getters and setters to access and manipulate that data. In this context, the Bill class acts as a data structure to organize bill-related information.*



```
public record Bill(int id, int orderId, double total) {

    /**
     * Constructs a new bill with default values.
     */
    no usages  cosmintianu
    public Bill() {
        this( id: 0,  orderId: 0,  total: 0);
    }

    /**
     * Constructs a new bill with the specified order ID and total.
     *
     * @param orderId The ID of the associated order.
     * @param total   The total amount of the bill.
     */
    1 usage  cosmintianu
    public Bill(int orderId, double total) {
        this( id: 0, orderId, total);
    }
}
```

Figure 9: method example

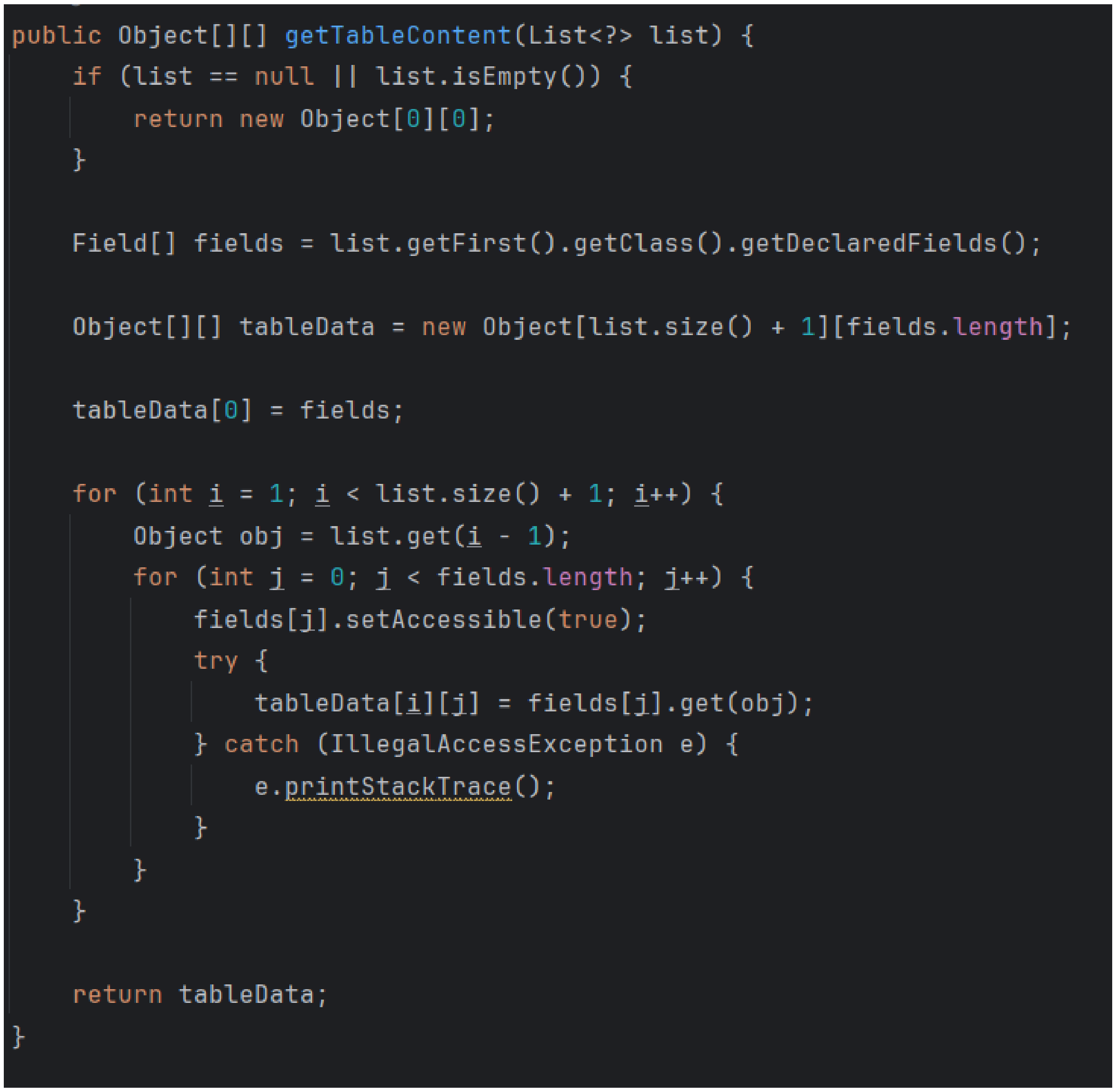
The ***presentation layer*** *has the* ***Controller*** *class in the presentation layer acts as the* ***bridge*** *between the graphical user interface (GUI) and the* ***business logic layer****. It manages user interactions, coordinating actions between the view and business logic, and updating the GUI components accordingly.*



```
2 usages   cosmintianu *
public List<Client> findAll() { return clientDAO.findAll(); }
```

Figure 10: method example

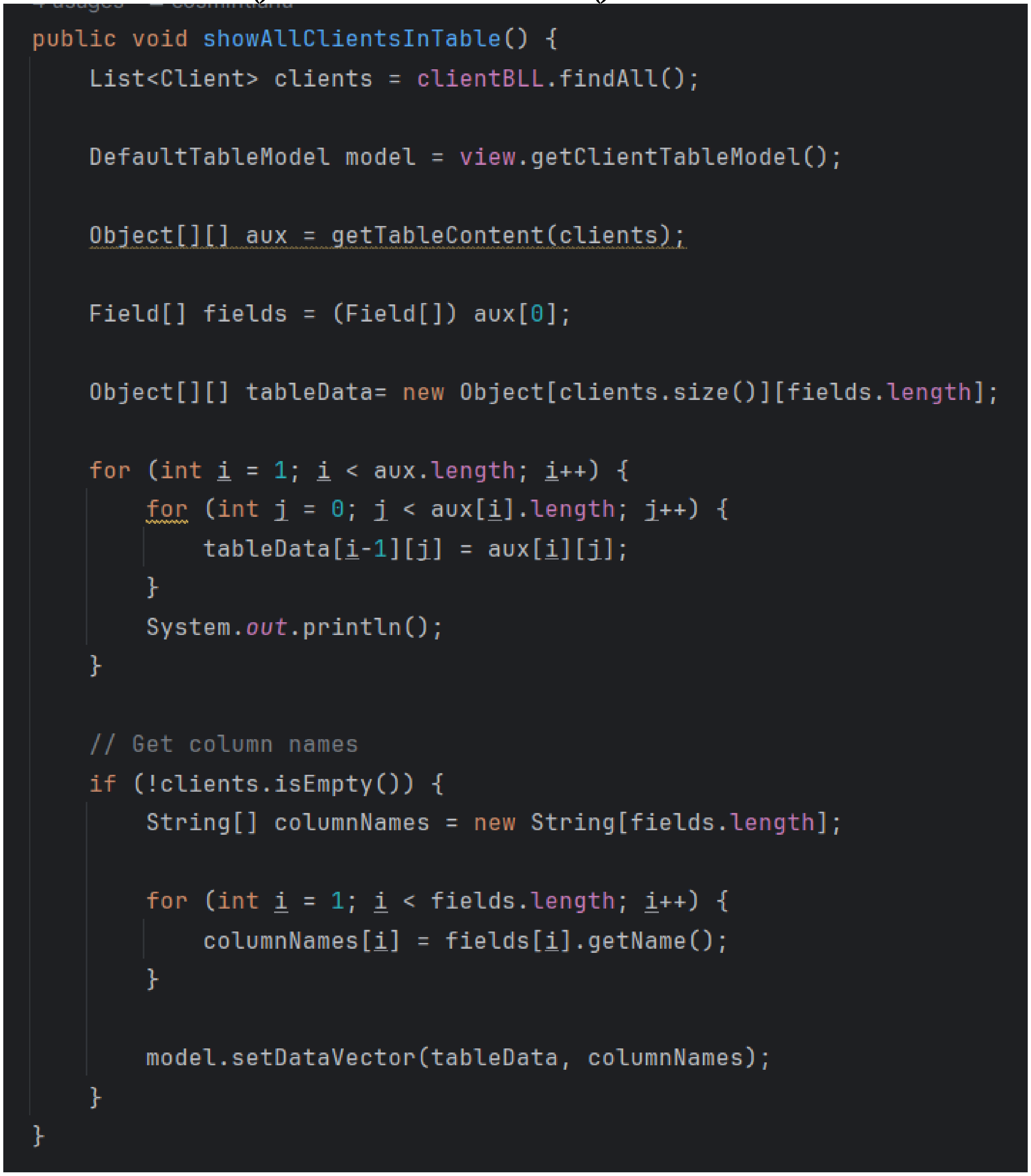
For client operations, methods like ***showAllClientsInTable(), addClient(), updateClient(), and deleteClient()*** *handle fetching, adding, updating, and deleting clients in the database using the* ***BLL layer****, which has not a lot of functionality just check if there is* ***available stock****. These methods ensure the client table in the GUI is updated with the* ***latest data using reflection****. Similarly,* ***showAllProductsInTable(), addProduct(), updateProduct()****, and* ***deleteProduct()*** *manage product-related operations, updating the product table in the GUI with data retrieved from* ***ProductBLL****.*



```
public Object[][] getTableContent(List<?> list) {
    if (list == null || list.isEmpty()) {
        return new Object[0][0];
    }

    Field[] fields = list.getFirst().getClass().getDeclaredFields();

    Object[][] tableData = new Object[list.size() + 1][fields.length];

    tableData[0] = fields;

    for (int i = 1; i < list.size() + 1; i++) {
        Object obj = list.get(i - 1);
        for (int j = 0; j < fields.length; j++) {
            fields[j].setAccessible(true);
            try {
                tableData[i][j] = fields[j].get(obj);
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
        }
    }

    return tableData;
}
```

Figure 11: getTable() Content example

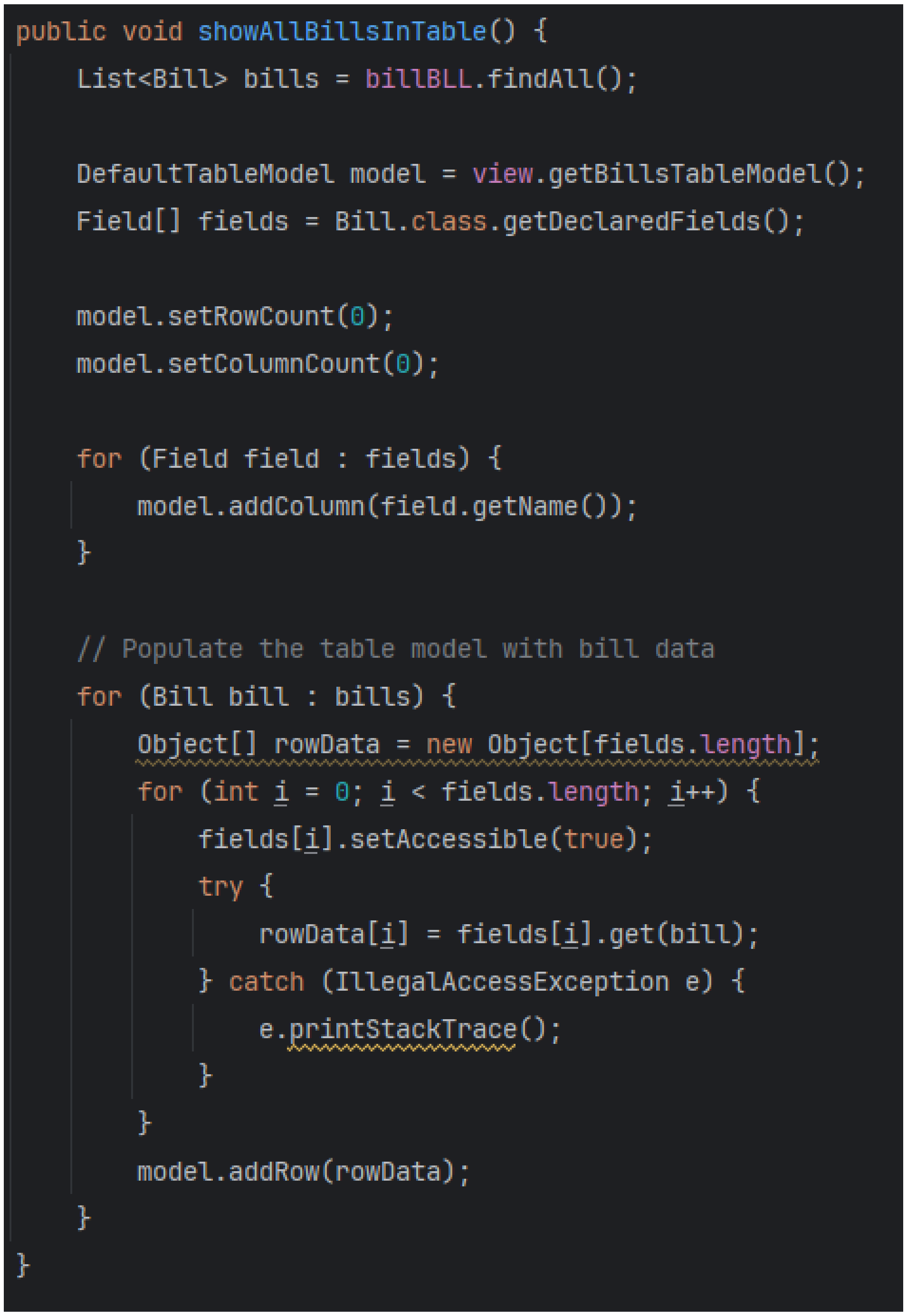
This method gets as a parameter a list of objects and places them in a matrix of objects, on the first row the ***Fields*** *and after that the rest of the contents of the table. And it is used in the* ***showAllClientsInTable() and showAllProductsInTable().***



```java
public void showAllClientsInTable() {
    List<Client> clients = clientBLL.findAll();

    DefaultTableModel model = view.getClientTableModel();

    Object[][] aux = getTableContent(clients);

    Field[] fields = (Field[]) aux[0];

    Object[][] tableData= new Object[clients.size()][fields.length];

    for (int i = 1; i < aux.length; i++) {
        for (int j = 0; j < aux[i].length; j++) {
            tableData[i-1][j] = aux[i][j];
        }
        System.out.println();
    }

    // Get column names
    if (!clients.isEmpty()) {
        String[] columnNames = new String[fields.length];

        for (int i = 1; i < fields.length; i++) {
            columnNames[i] = fields[i].getName();
        }

        model.setDataVector(tableData, columnNames);
    }
}
```

Figure 11: showAllClientsInTable() Content example

*The Controller also manages **orders and bills**. Methods like **showAllOrdersInTable**() and **showAllBillsInTable**() update the orders and bills tables in the GUI with data from OrdersBLL and BillBLL.*



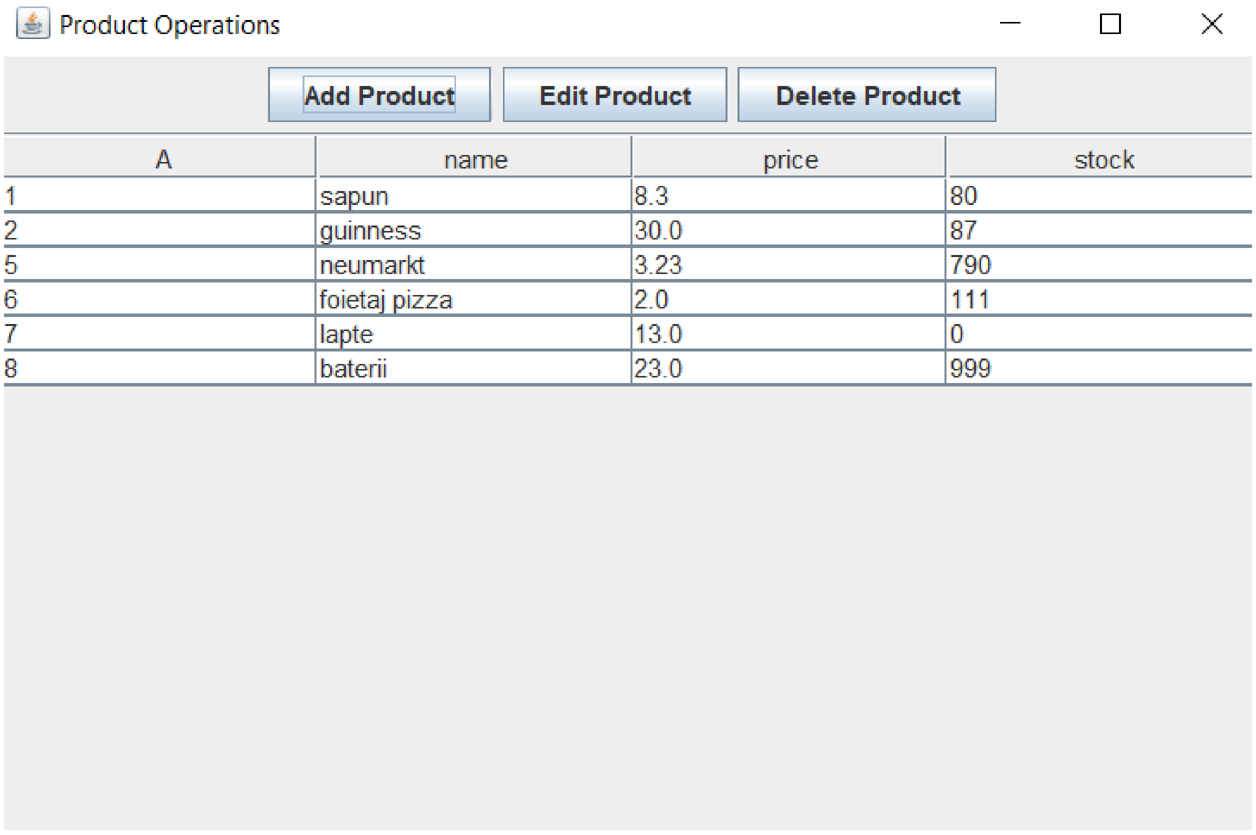
```java
public void showAllBillsInTable() {
    List<Bill> bills = billBLL.findAll();

    DefaultTableModel model = view.getBillsTableModel();
    Field[] fields = Bill.class.getDeclaredFields();

    model.setRowCount(0);
    model.setColumnCount(0);

    for (Field field : fields) {
        model.addColumn(field.getName());
    }

    // Populate the table model with bill data
    for (Bill bill : bills) {
        Object[] rowData = new Object[fields.length];
        for (int i = 0; i < fields.length; i++) {
            fields[i].setAccessible(true);
            try {
                rowData[i] = fields[i].get(bill);
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
        }
        model.addRow(rowData);
    }
}
```

Figure 12: method example

*Additionally, methods like **getClientsNameAndId**() and **getProductsNameAndId**() retrieve client and product names and IDs for use **in the GUI** for selecting the wanted product and client when creating an order.*

Also here is present the ***View*** *class creates and manages the* ***graphical interface*** *of the warehouse management system. It* ***constructs windows*** *for tasks like adding, updating, and deleting clients and products, managing orders, and viewing bills. Using Java Swing components, it facilitates user interaction by displaying buttons, text fields, tables, and frames. Interfacing with the Controller class,* ***it updates the GUI*** *with data fetched from the database, enabling efficient warehouse resource management.*

Here, it an example of the ***Product Operations Frame,*** *it has the Product CRUD.*



A	name	price	stock
1	sapun	8.3	80
2	guinness	30.0	87
5	neumarkt	3.23	790
6	foietaj pizza	2.0	111
7	lapte	13.0	0
8	baterii	23.0	999

Figure 13: Product operations frame.

The available actions are Creating, Reading, Updating and Deleting ***Clients*** *and the same principles apply for the* ***Products.***

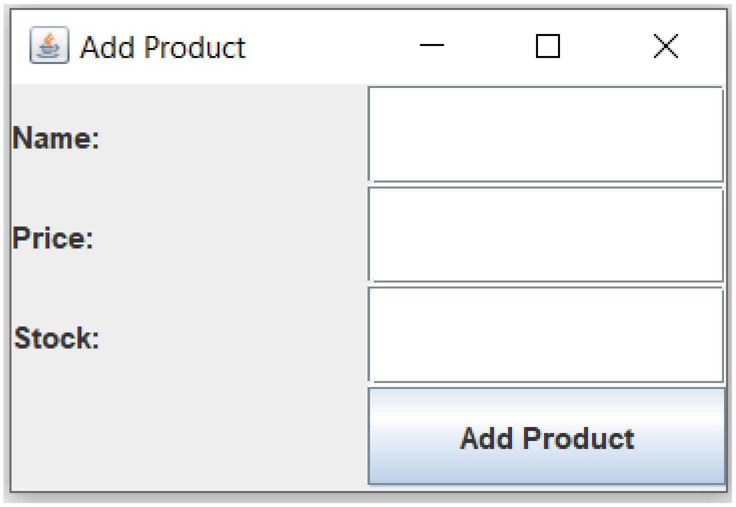


Figure 14: Add product frame.

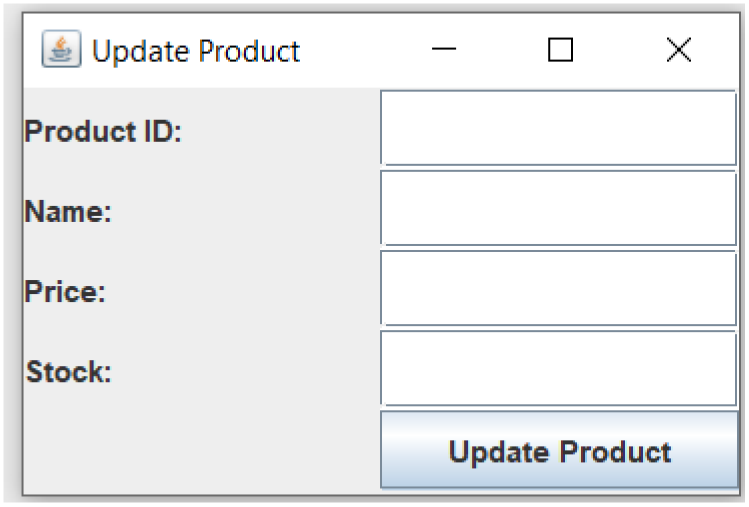


Figure 15: Update product frame.

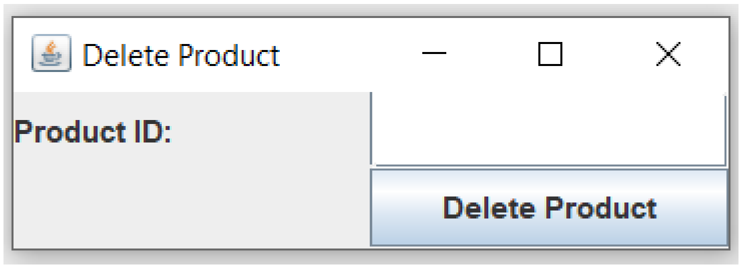
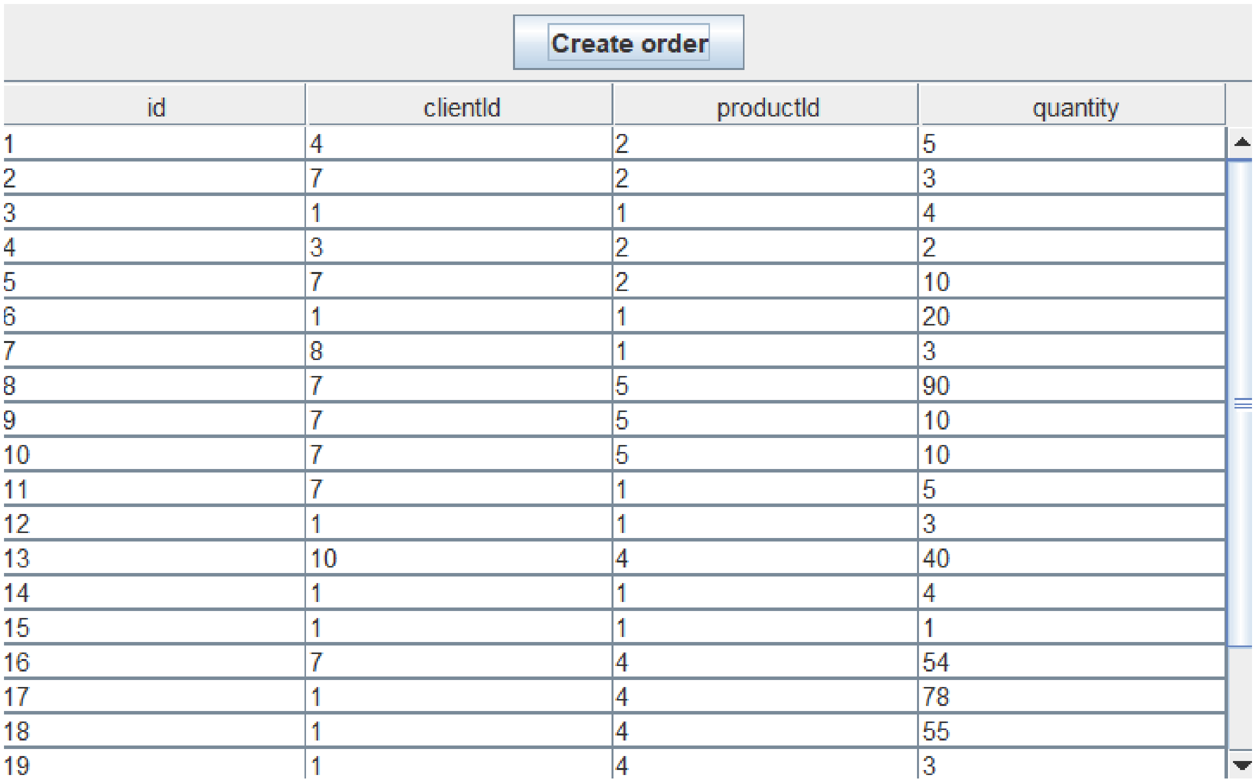


Figure 16: Delete product frame.

*The same frame are available for the **Clients,** but containing their own attributes.*

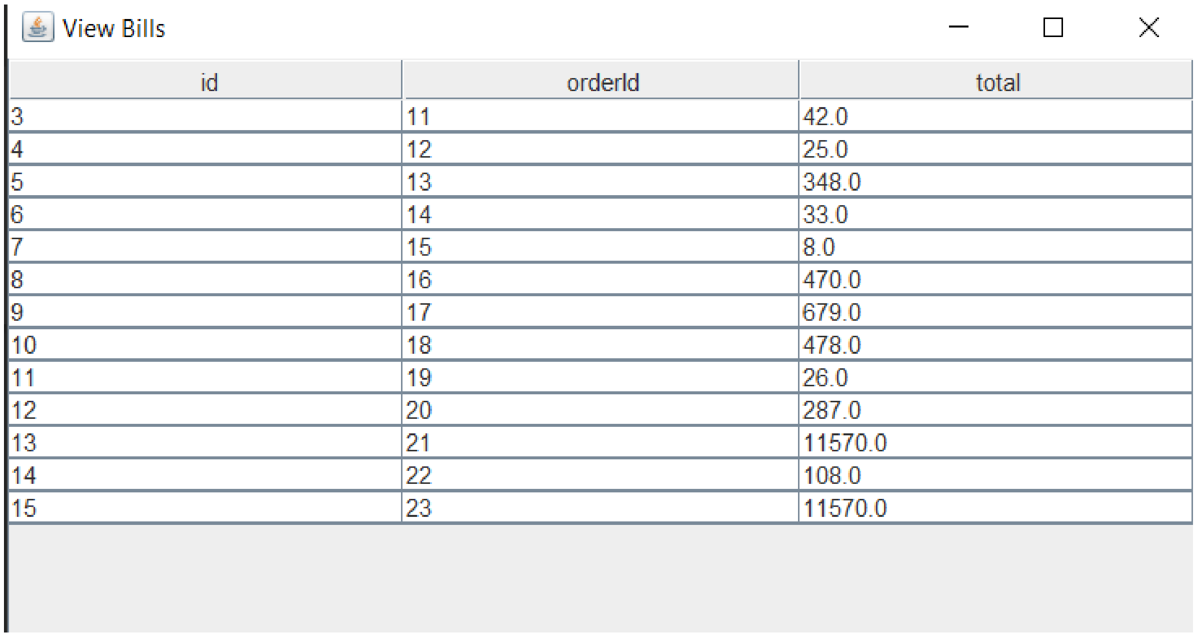
*This is the **Orders Frame,** where the user can create an order and see all the past orders with their id, the id of the client that made the order and the id of the product he bought.*



id	clientId	productId	quantity
1	4	2	5
2	7	2	3
3	1	1	4
4	3	2	2
5	7	2	10
6	1	1	20
7	8	1	3
8	7	5	90
9	7	5	10
10	7	5	10
11	7	1	5
12	1	1	3
13	10	4	40
14	1	1	4
15	1	1	1
16	7	4	54
17	1	4	78
18	1	4	55
19	1	4	3

Figure 13: Product operations frame.

*A **bill** is automatically generated for an order if succesfully placed.*



id	orderId	total
3	11	42.0
4	12	25.0
5	13	348.0
6	14	33.0
7	15	8.0
8	16	470.0
9	17	679.0
10	18	478.0
11	19	26.0
12	20	287.0
13	21	11570.0
14	22	108.0
15	23	11570.0

Figure 13: Product operations frame.

5. Conclusions

In ***conclusion****, the order warehouse management system plays a crucial role in streamlining inventory operations, facilitating client management, and ensuring efficient order processing. This project has provided invaluable insights into database management, GUI development using Java Swing, and the MVC architecture. Personally, I've gained hands-on experience in implementing CRUD operations, handling user interactions, and maintaining data integrity. Moving forward, potential enhancements could focus on refining the GUI to enhance user experience and incorporating additional features for more comprehensive warehouse management capabilities.*

6. Bibliography

1. *https://dsrl.eu/courses/pt/materials/PT2024_A3.pdf*
2. *https://dsrl.eu/courses/pt/materials/PT_2024_A3_S1.pdf*
3. *https://dsrl.eu/courses/pt/materials/PT_2024_A3_S2.pdf*