DOCUMENTATION

ASSIGNMENT *3*

STUDENT NAME: BOTA ALIN ALEXANDRU

GROUP: 30423

# CONTENTS

[1. Assignment Objective 2](#_Toc128043139)

[2. Problem Analysis, Modeling, Scenarios, Use Cases 2](#_Toc128043140)

[3. Design 4](#_Toc128043141)

[4. Implementation 6](#_Toc128043142)

[5. Results 12](#_Toc128043143)

[6. Conclusions 13](#_Toc128043144)

[7. Bibliography 13](#_Toc128043145)

# Assignment Objective

The objective of this assignment is to design an Orders Management application for processing client orders in a warehouse. The application should utilize a relational database to store information about products, clients, and orders. The design should follow the layered architecture pattern and include the following classes:

Model classes: These classes represent the data models of the application. They define the structure and attributes of the entities in the system, such as Product, Client, and Order. Each class should have appropriate properties and methods to manage and manipulate its data.

Business Logic Classes: These classes contain the application logic and implement the business rules. They handle the processing of orders, validation, calculations, and any other operations required to manage the orders. For example, there could be a class called Orders that stores information about the order.

Presentation Classes: These classes are responsible for the graphical user interface (GUI) and user interaction. They provide the screens, forms, and controls for users to interact with the application. For example, there could be a class called OrdersView that handles the display and input of order information.

Data Access Classes: These classes manage access to the database and perform operations such as inserting, updating, and deleting data. They interact with the database and provide methods for other classes to retrieve or modify data. For example, there could be a class called OrderDAO (Data Access Object) that handles database operations related to orders.

# Problem Analysis, Modeling, Scenarios, Use Cases

The problem at hand is to develop an Orders Management application for processing client orders in a warehouse. The application will utilize a relational database to store information about products, clients, and orders. The key objective is to design the application using a layered architecture pattern and implement the required classes to fulfill the functionality.

To model the Orders Management application, we can identify several entities and their relationships:

**Modeling**

Products: Attributes: ID, Name, Price, Category, Quantity

Relationships: Products can be associated with Orders.

Clients: Attributes: ID, Name, Address, Email, Age

Relationships: Clients can place Orders.

Orders: Attributes: ID, ClientID, ProductID.

Relationships: Orders can have multiple Products associated with them, Orders are placed by Clients.

**Scenarios**

*Placing an Order*: The client selects the desired products from the available list.

The system calculates the total amount based on the selected products.

The order is saved in the database with a unique order ID, the clients ID and order ID

*Viewing Order Details*: The system retrieves the order details based on the order ID.

The system displays the order information, including products, quantities, total amount, and client details.

**Use Cases:**

*Place Order:* The client selects products and adds them to the order. The system calculates the total amount.The system saves the order to the database.

*View Order Details:* The client selects the order to view. The system retrieves the order details.

The system displays the order information, client information and product information.

Insert, Update, Delete, Show Clients : The user can interact with Clients database and modify it.

Insert, Update, Delete, Show Products : The user can interact with Products database and modify it.

# Design

A picture containing text, screenshot, diagram, plan

Description automatically generated

The **Controller** Class -> This class is the main core of the project. It interacts with all other classes : GUI classes, data access classes, Business model classes. This is the class that handles all the operations that were created in the other classes.

The **Product View, Game Store View, Client View, Orders View** Classes -> Are the classes that provide the GUI for the application. These classes have direct access with the user and the inputs that are provided.

The **Abstract DAO**  Class -> Is an abstract class, that provides generic methods, using reflection, that contain all the operations available : insert, update, delete that affects directly the database were the information is stored in. By reflection, we can use the same methods with different objects, a very powerful tool. The other classes: **ClientDAO, ProductDAO, OrdersDAO** , are extended from this class. These classes are used only for differential purposes. To be more easily readable, which table will be affected by a specific operation.

The **Client, Product, Orders** Classes -> Have the purpose to store information about a specific model.

The **Connection Factory** Class -> Links the application with the database, where the information is store. In this case is used a My SQL database.

The **Bill** Class -> Is used to create a bill after an order is done.

The **Main** Class -> Has the purpose to start the application.

# Implementation

* 1. Controller Class

A screen shot of a computer program

Description automatically generated with medium confidence

The constructor of this class provides all the information needed to execute the operations, and listens to all the inputs from the GUI.

A screenshot of a computer program

Description automatically generated with low confidence

The method starts by retrieving the selected product and client from the GUI components

(productOrderView.getProductListComboBox() and

productOrderView.getClientListComboBox()).

It also retrieves the quantity entered by the user from productOrderView.getQuantityTextField().

Checking Stock Availability:

The method checks if the requested quantity is greater than the available stock of the selected product.

If the quantity exceeds the stock, an error message is displayed using JOptionPane.showMessageDialog().

Updating Product Stock and Creating Order:

If the requested quantity is available, the method proceeds to update the stock of the selected product by subtracting the purchased quantity.

The updated product information is saved to the database using productDAO.update().

A new Orders object is created with the client ID, product ID, and quantity.

The order is inserted into the database using ordersDAO.insert().

Updating GUI and Notifying User:

After successfully placing the order, the method retrieves all orders from the database using ordersDAO.findAll().

The GUI table is updated with the latest order information using the populate() method.

Other GUI tables are updated through the populateTables() method (not shown in the code snippet).

A message dialog is displayed to inform the user that the order was successfully placed.

Creating Bill:

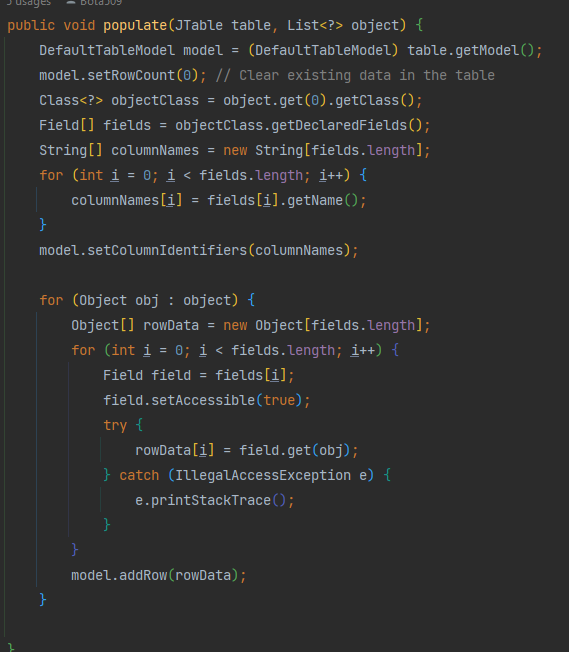
The method proceeds to create a bill for the order using the createBill() method, passing the order, client, and product objects.

The createBill() method is responsible for generating and displaying the bill.

A screen shot of a computer code

Description automatically generated with low confidence

The populateComboBoxes() method retrieves the client and product data from the database, creates DefaultComboBoxModel objects, populates them with the respective names, and sets the models for the client and product combo boxes in the GUI. This ensures that the combo boxes are populated with the available clients and products for the user to select from.



The populate() method dynamically populates a JTable with data from a list of objects. It clears any existing data, retrieves the class and field information, sets column names, and adds rows with the object data to the table model. This allows displaying the object data in the table with appropriate column labels.

* 1. The AbstractDAO class

A screen shot of a computer program

Description automatically generated with low confidence

The method starts by declaring and initializing the necessary variables: connection, statement, and resultSet.

The connection variable is initialized using ConnectionFactory.getConnection(), which is assumed to establish a connection to the database.

The SQL query is constructed to select all records from the table corresponding to the type of objects.

A prepared statement is created using the SQL query.

The prepared statement is executed using statement.executeQuery(), which returns a resultSet containing the retrieved records from the database.

The createObjects(resultSet) method is called to process the result set and create a list of objects based on the retrieved data.

The implementation of createObjects(resultSet) is not provided in the given code snippet, but it is assumed to iterate over the result set, extract the data, and create objects of the appropriate type.

If any SQLException occurs during the database operations, an error message is logged using a LOGGER.

Finally, the database resources (resultSet, statement, and connection) are closed using ConnectionFactory.close() to release the resources and ensure proper cleanup.

A screen shot of a computer code

Description automatically generated with low confidence

The steps for implementing the following methods in this class are similar. The findByNmae(String name) method returns an object of generic type when finds a line in the database that has the key equal to the name passed at the parameter.

A screenshot of a computer program

Description automatically generated with medium confidence

The method starts by declaring and initializing the necessary variables: dbConnection, statement, and rs.

The dbConnection variable is initialized using ConnectionFactory.getConnection(), which is assumed to establish a connection to the database.

The SQL query is constructed to insert data into the table corresponding to the type of objects.

Two loops are used to iterate over the fields of the object's class.

In the first loop, the field names are appended to the querry string, except for the "id" field.

In the second loop, placeholders for the field values in the query are added using "?".

The constructed query is finalized by removing the trailing comma and closing parentheses.

The prepared statement is created using the constructed query with dbConnection.prepareStatement(querry).

A counter variable i is used to keep track of the parameter index in the prepared statement.

Another loop iterates over the fields to set the corresponding values in the prepared statement.

The "id" field is skipped since it's assumed to be automatically generated by the database.

The field.setAccessible(true) allows accessing private fields if necessary.

The field value from the provided object t is retrieved using field.get(t).

The field value is set in the prepared statement using statement.setObject(i, field.get(t)).

The INSERT statement is executed using statement.executeUpdate(), which inserts the record into the database table.

If any exception occurs during the database operations, an error message is logged using a LOGGER.

Finally, the database resources (rs, statement, and dbConnection) are closed using ConnectionFactory.close() to release the resources and ensure proper cleanup.

The original object t is returned after the insertion is completed.

A screenshot of a computer program

Description automatically generated with medium confidence

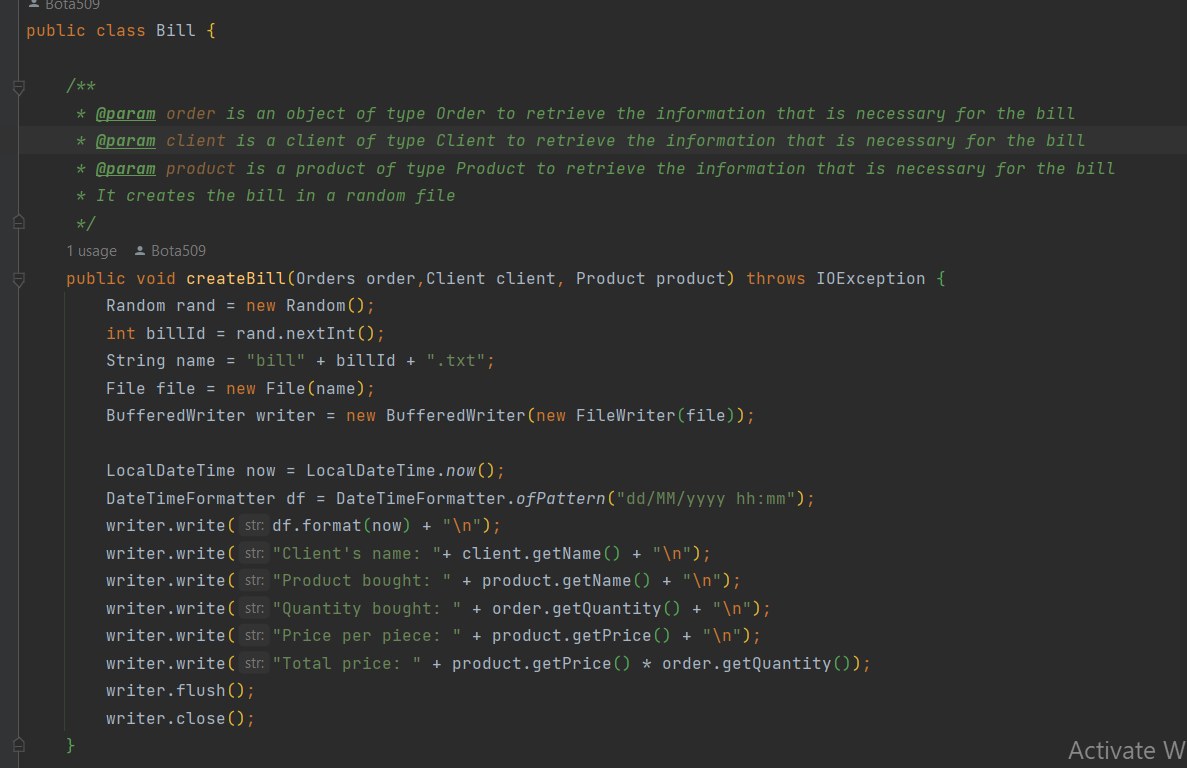
The difference is the SQL command that is provided -> the querry String statement will be constructed differently.

A picture containing text, screenshot, software

Description automatically generated

The delete(int id) method constructs an SQL DELETE query based on the provided ID and deletes the corresponding record from the database table. The method handles exception logging, resource cleanup, and displays a JOptionPane message if the deletion cannot be performed due to dependencies on other objects.

* 1. The bill class



The createBill(Orders order, Client client, Product product) method generates a bill for a client's order by writing relevant information to a text file. The bill includes the date, client's name, product details, quantity, price per piece, and the total price.

# Results

# Conclusions

In conclusion, the assignment objective was to design an application for order management in a warehouse. The application is based on a layered architecture pattern and utilizes relational databases to store data related to products, clients, and orders. The following classes are identified as minimal components of the application:

Model classes: These classes represent the data models of the application, including entities such as products, clients, and orders.

Business Logic classes: These classes contain the application logic, handling processes such as order creation, stock management, and business rules associated with order management.

Presentation classes: These classes are responsible for the GUI-related aspects of the application, including user interfaces, forms, and interactions with users.

Data access classes: These classes provide the functionality to access and interact with the relational databases. They handle tasks such as retrieving data, inserting new records, and deleting records from the database.

By following the layered architecture pattern and organizing the application into these classes, it promotes a modular and maintainable design. Other classes and packages can be added as needed to implement additional functionality and features required by the application.

# Bibliography

<https://www.youtube.com/watch?v=bhhMJSKNCQY>

<https://stackoverflow.com/questions/1238145/how-to-run-a-jar-file>

<https://www.baeldung.com/javadoc>

<https://www.youtube.com/watch?v=fAQB556HtiI&t=490s>

https://www.youtube.com/watch?v=3OrEsC-QjUA&t=1704s