DOCUMENTATION

ASSIGNMENT *ASSIGNMENT_NUMBER_3*

STUDENT NAME: Feier Catalin Vasile
GROUP: 30424-1

CONTENTS

1. Assignment Objective

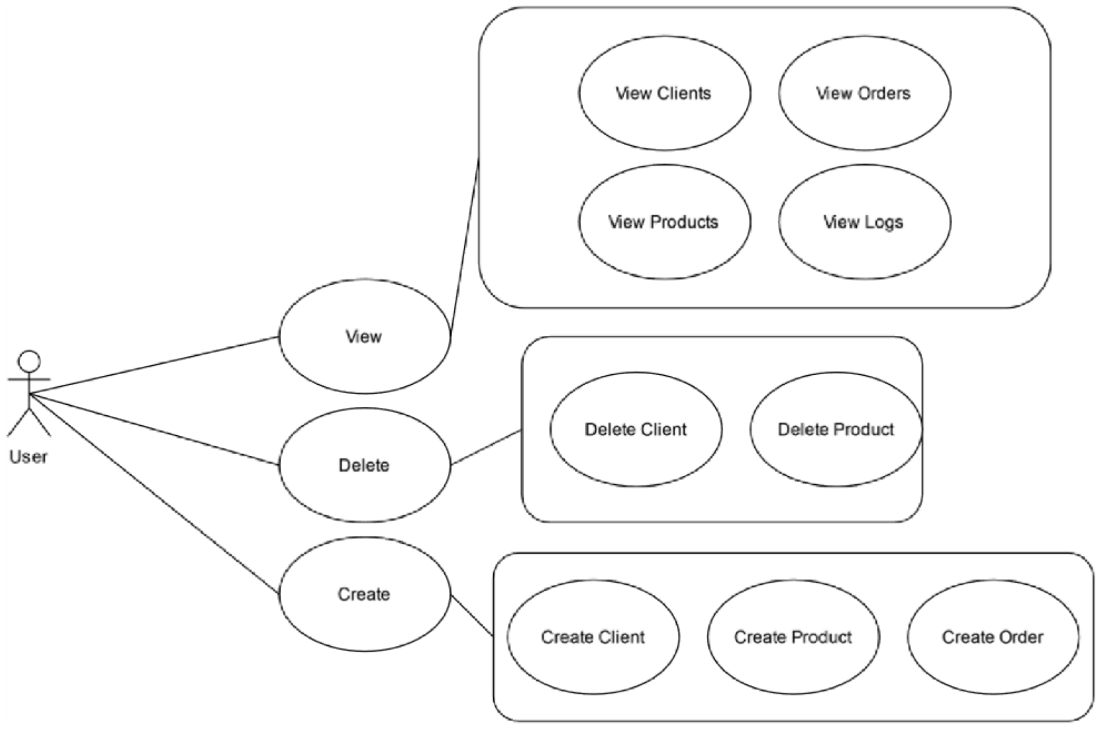
The objective of this assignment is to design and implement an Orders Management application for processing client orders in a warehouse setting. The project necessitates a comprehensive approach to software development, utilizing a layered architecture pattern to ensure separation of concerns and modularity.

The application is structured into four primary packages: model, business, dataAccess, and presentation. Each package plays a distinct role in the application's functionality. The model package contains data models representing entities such as Client, Product, Ordertable, OrderDetail, and Bill. These classes encapsulate the core data and their relationships within the database.

This assignment aims to build a well-architected, maintainable, and user-friendly Orders Management application, leveraging object-oriented principles, reflection techniques, and a structured, layered design.

2. Problem Analysis, Modeling, Scenarios, Use Cases

USE CASE-> Diagram



USE CASE -> Description:

Use Case: For Viewing the Data
Primary Actor: User

Success Scenario Steps:

I. The user selects one of the three options: Client, Order, Product

II. The users select based on the first choice the valid “SEE” option.

III. The Application shows the data.

Alternative Sequences:

a) The database connection fails: go to step I.

Use Case: Data Creation

Primary Actor: User

Success Scenario Steps:

I. The user selects one of the three options: Client, Order, Product.

II. The user completes the required textboxes/checkboxes in the specific window.

III. The user presses the corresponding “CREATE” button.

IV. A message pops up on the screen.

Alternative Sequences:

a) The database connection fails, go to step I.

b) The user enters invalid data: go to step II.

Use Case: Data Update

Success Scenario Steps:

I. The user selects one of the following three options: Client, Order, Product.

II. The user selects the corresponding ID of the item it wants to update.

III. The user completes the corresponding fields.

IV. The user presses the correct “UPDATE” button.

V. A message pops up on the screen regarding the result.

Alternative Sequences:

a) The database connection fails, go to step I.

b) Invalid user input, go to step III,

c) Client/Order/Product doesn’t exist, go to step II.

Use Case: Data Deletion

Success Scenario Steps:

I. The user selects one of the following three options: Client, Order, Product.

II. The user selects the ID of the Client/Product/Order they want to delete.

III. The user presses the “DELETE” button.

IV. The selected model is deleted.

V. A message pops up on the screen regarding the result.

Alternative Sequences:

a) The database connection fails, go to step I.

b) The ID doesn’t exist, go to step II.

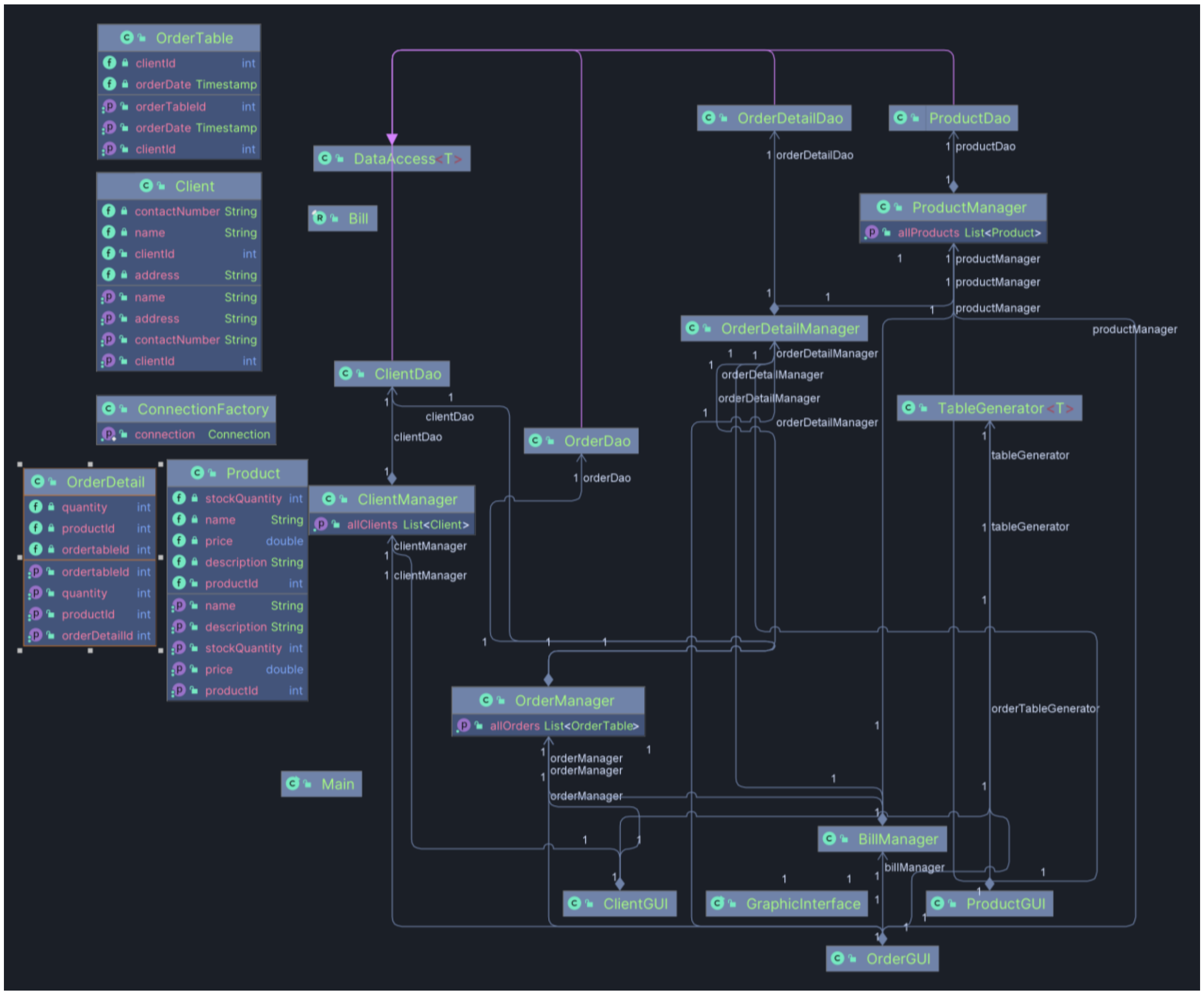
3. Design

The application follows a well-structured layered architecture, which includes a presentation layer, a business logic layer, a data access layer, and a model layer. The **presentation layer** consists of GUI classes responsible for interacting with the user. These classes provide a user-friendly interface for displaying data and capturing user input, ensuring a seamless user experience. The **model layer** contains data models representing the core entities of the application, such as `Bill`, `OrderDetail`, `Client`, `Product`, and `OrderTable`. Each class in this layer encapsulates the properties and behaviors of the corresponding entity.

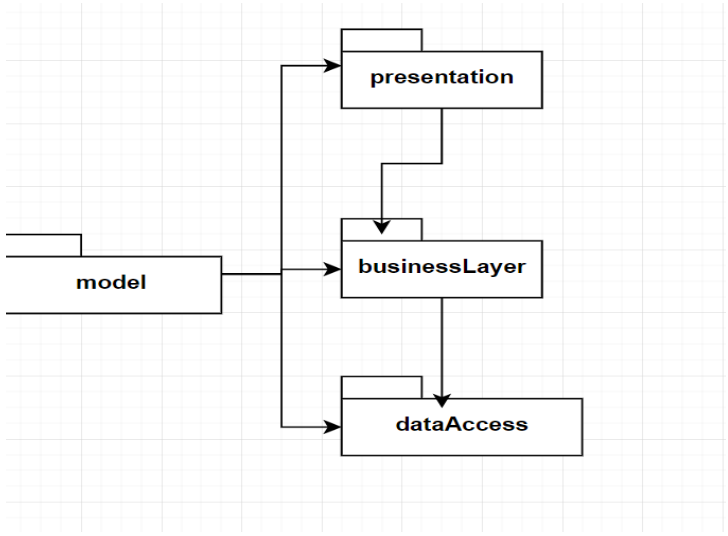
The **data access layer** (DAO) is responsible for database interactions. It includes Data Access Objects (DAOs) like `ClientDao`, `OrderDetailDao`, `OrderDao`, and `ProductDao`, which provide CRUD operations for each entity. The **business logic layer** contains manager classes such as `BillManager`, `ClientManager`, `OrderDetailManager`, `OrderManager`, and `ProductManager`. These classes handle the core operations and business rules, coordinating between the model layer and the data access layer.

The application demonstrates several key OOP design principles, including encapsulation, single responsibility principle (SRP), separation of concerns, and abstraction. Each class has a single responsibility, making the code easier to maintain and extend. Encapsulation is implemented effectively, with classes encapsulating their data and providing public methods for access and modification. The separation of concerns ensures clear delineation of functionality between layers, enhancing maintainability and scalability. Overall, the application is designed to be modular, maintainable, and extensible, adhering to best practices in software engineering and object-oriented design.

CLASS DIAGRAM:



PACKAGE DIAGRAM:



USED DATA STRUCTURES:

ArrayList is extensively used throughout the business logic layer for storing and managing collections of objects. For example, in the BillManager class, an ArrayList is used to hold a list of Bill objects generated from orders. Similarly, the ClientManager, OrderDetailManager, OrderManager, and ProductManager classes use lists to manage collections of clients, order details, orders, and products, respectively. The ArrayList provides dynamic resizing and allows for efficient random access, making it suitable for operations that require frequent reads and iterations.
The Bill class is implemented as a record, a special kind of class in Java that provides a concise syntax for immutable data objects. Records automatically generate constructors, accessors, equals, hashCode, and toString methods, making them an excellent choice for simple data carriers.
In the data access layer, SQL result sets are used to retrieve and manipulate data from the database. While not explicitly shown in the provided code, result sets are typically processed to populate lists of objects in the DAO classes.
Exception handling is used to manage errors and unexpected situations, particularly when interacting with the database. Custom exceptions or SQL exceptions are caught and handled appropriately to ensure the application's robustness.

4. Implementation

Each class will be described (fields, important methods). Also, the implementation of the graphical user interface will be described.

I. The **ConnectionFactory** class is a central part of the data access layer, responsible for managing database connections. Its design uses the Singleton pattern to ensure a single instance is responsible for creating and managing connections, thereby promoting efficient resource management.

Important Fields

1. **LOGGER**: A logger for logging connection-related events and errors.
2. **DRIVER**: The JDBC driver class name.
3. **URL**: The URL of the database.
4. **USER**: The database user name.
5. **PASSWORD**: The database user's password.
6. **singleInstance**: A static instance of **ConnectionFactory**, ensuring only one instance of the class exists.

Important MethodsI

1. **Constructor (private ConnectionFactory())**:
    - **Description**: The constructor is private to prevent instantiation from other classes. It loads the JDBC driver class.
    - **Implementation Details**: Uses **Class.forName(DRIVER)** to load the JDBC driver.
2. **createConnection()**:
    - **Description**: Establishes a new database connection.

    - **Implementation Details**: Uses **DriverManager.getConnection(URL, USER, PASSWORD)** to create a connection. Logs an error if the connection fails.
    - **Return Type**: **Connection**
3. **getConnection()**:
    - **Description**: Provides a global access point to get a database connection.
    - **Implementation Details**: Calls **singleInstance.createConnection()** to obtain a connection.
    - **Return Type**: **Connection**
4. **close(Connection connection)**:
    - **Description**: Closes the given database connection.
    - **Implementation Details**: Checks if the connection is not null, then attempts to close it and logs an error if it fails.
    - **Parameters**: **Connection connection**
5. **closeStatement(Statement statement)**:
    - **Description**: Closes the given SQL statement.
    - **Implementation Details**: Checks if the statement is not null, then attempts to close it and logs an error if it fails.
    - **Parameters**: **Statement statement**
6. **closeResultSet(ResultSet resultSet)**:
    - **Description**: Closes the given result set.
    - **Implementation Details**: Checks if the result set is not null, then attempts to close it and logs an error if it fails.
    - **Parameters**: **ResultSet resultSet**

II. The DataAccess class in the dataAccess package is a generic class that manages data access operations for various entities. It uses reflection to dynamically handle different entity types, allowing for a flexible and reusable data access layer. This class relies on the ConnectionFactory to establish and manage connections to the database.

Important Fields

1. **connection: A Connection object that represents a connection to the database.**
2. **type: A Class object representing the type of the entity managed by this DataAccess instance.**

Important Methods

1. **Constructor (DataAccess()):**
    - **Description: Initializes the type field and establishes a database connection using ConnectionFactory.**
    - **Implementation Details: Uses reflection to determine the entity type at runtime.**
2. **create(T t):**
    - **Description: Inserts a new entity into the database.**
    - **Implementation Details: Generates an SQL INSERT statement using the entity's fields, sets the parameters in the PreparedStatement, and executes the update. It also retrieves the generated key and sets it in the entity.**
    - **Return Type: T (the created entity)**
    - **Exception Handling: Catches SQLException, IllegalAccessException, NoSuchMethodException, and InvocationTargetException.**
3. **readAll():**
    - **Description: Retrieves all entities from the database.**

    - **Implementation Details: Executes an SQL SELECT statement and extracts each entity from the ResultSet.**
    - **Return Type: List<T> (a list of all entities)**
    - **Exception Handling: Catches SQLException.**
4. **read(int id):**
    - **Description: Retrieves an entity by its ID from the database.**
    - **Implementation Details: Executes an SQL SELECT statement with a WHERE clause for the entity ID and extracts the entity from the ResultSet.**
    - **Return Type: T (the entity with the given ID)**
    - **Exception Handling: Catches SQLException.**
5. **update(T t):**
    - **Description: Updates an existing entity in the database.**
    - **Implementation Details: Generates an SQL UPDATE statement using the entity's fields, sets the parameters in the PreparedStatement, and executes the update.**
    - **Return Type: T (the updated entity)**
    - **Exception Handling: Catches SQLException.**
6. **delete(int id):**
    - **Description: Deletes an entity by its ID from the database.**
    - **Implementation Details: Executes an SQL DELETE statement with a WHERE clause for the entity ID.**
    - **Return Type: void**
    - **Exception Handling: Catches SQLException.**

III. The **OrderDetailDao** class in the **dataAccess** package extends the generic **DataAccess** class to manage **OrderDetail** entities. It inherits CRUD functionality from **DataAccess** and adds methods for querying order details by specific criteria like order ID and product ID. This class provides specialized data access methods while leveraging the generic setup for common operations.

Important Fields

- **connection**: Inherited from **DataAccess**, this field represents the connection to the database, ensuring that all database operations are performed on an active connection.

Important Methods

1. **Constructor (OrderDetailDao())**: Initializes the **OrderDetailDao** by calling the superclass constructor, setting up the necessary configurations to manage **OrderDetail** entities without requiring additional setup.
2. **extractFromResultSet(ResultSet rs)**: Converts a **ResultSet** row into an **OrderDetail** object by extracting fields like **orderdetailId**, **ordertableId**, **productId**, and **quantity**, and using them to construct a new **OrderDetail**.
3. **getOrderDetailsByOrderId(int orderId)**: Retrieves a list of **OrderDetail** entities by the order ID, executing a **SELECT** statement with a **WHERE** clause for **ordertableId**, and adding each **OrderDetail** to a list.
4. **getOrderDetailsByProductId(int productId)**: Retrieves a list of **OrderDetail** entities by the product ID, executing a **SELECT** statement with a **WHERE** clause for **productId**, and adding each **OrderDetail** to a list.

IV. The **ProductManager** class in the **businessLayer** package is responsible for managing product-related operations within the application. It acts as a bridge between the data access layer and the business logic layer, utilizing the **ProductDao** class to interact with the database.

Important Fields

- **productDao**: This field holds an instance of **ProductDao**, which is used to perform CRUD operations on **Product** entities. It ensures that all database interactions related to products are handled efficiently.

Important Methods

1. **Constructor (ProductManager())**: Initializes a new instance of **ProductManager** by creating a new **ProductDao** object. This setup ensures that the **ProductManager** has access to the necessary data access functionalities.
2. **createProduct(String name, String description, double price, int stockQuantity)**: Creates a new **Product** object with the provided details and uses the **ProductDao** to insert it into the database. This method returns the newly created product.
3. **updateProduct(int productId, String name, String description, double price, int stockQuantity)**: Updates an existing **Product** object with the given ID by setting new values for its properties. It then uses the **ProductDao** to update the product in the database, returning the updated product.
4. **deleteProduct(int productId)**: Deletes the product with the specified ID by calling the **ProductDao**'s delete method. This operation ensures the removal of the product from the database.
5. **checkStock(int productId)**: Retrieves the stock quantity of a product by its ID. This method calls **ProductDao** to fetch the product from the database and returns its stock quantity.
6. **getAllProducts()**: Retrieves a list of all products by calling the **ProductDao**'s **readAll** method. This method returns a list of all **Product** objects in the database.
7. **getProductById(int productId)**: Fetches a product by its ID using the **ProductDao**'s **read** method. This method returns the **Product** object corresponding to the given ID.
8. **checkStock(int productId, int quantity)**: Checks if there is enough stock for a product with the given ID and quantity. It calls **ProductDao** to read the product and compares its stock quantity with the requested quantity, returning **true** if there is sufficient stock and **false** otherwise.

V. The **OrderManager** class is part of the business layer and is responsible for managing orders within the application. It interacts with **OrderDao**, **ClientDao**, and **OrderDetailManager** to perform its operations. The class ensures that orders are created, updated, and deleted correctly while maintaining the integrity of related data.

Important Fields

- **orderDao**: An instance of **OrderDao** to perform CRUD operations on **OrderTable** entities.
- **clientDao**: An instance of **ClientDao** to verify the existence of clients.
- **orderDetailManager**: An instance of **OrderDetailManager** to manage order details related to orders.

Important Methods

1. **Constructor (OrderManager())**: Initializes a new **OrderManager** with instances of **OrderDao**, **ClientDao**, and **OrderDetailManager**. This setup ensures that the **OrderManager** has access to all necessary data access and management functionalities.
2. **createOrder(int clientId, Timestamp orderDate)**: Creates a new **OrderTable** object if the specified client exists. It throws an **SQLException** if the client does not exist. This method returns the newly created order.

3. **updateOrder(int orderId, int clientId, Timestamp orderDate)**: Updates an existing **OrderTable** object if the specified client exists. It throws an **SQLException** if the client does not exist. This method returns the updated order.
4. **deleteOrder(int orderId)**: Deletes an order and its related order details. It first deletes all order details associated with the order, then deletes the order itself. This ensures data consistency.
5. **deleteOrdersWithClientId(int clientId)**: Deletes all orders associated with a specific client ID. It fetches all orders for the client and deletes each order and its details.
6. **getAllOrders()**: Retrieves a list of all orders in the application. This method calls **OrderDao**'s **readAll** method and returns a list of **OrderTable** objects.

VI. The **OrderDetailManager** class handles the management of order details in the application. It interacts with **OrderDetailDao** to perform CRUD operations on **OrderDetail** entities and uses **ProductManager** to validate and manage product-related data.

Important Fields

- **orderDetailDao**: An instance of **OrderDetailDao** for performing CRUD operations on order details.
- **productManager**: An instance of **ProductManager** to manage product information and ensure stock availability.

Important Methods

1. **Constructor (OrderDetailManager())**: Initializes a new **OrderDetailManager** with instances of **OrderDetailDao** and **ProductManager**. This setup ensures that the **OrderDetailManager** has access to all necessary data access and product management functionalities.
2. **createOrderDetail(int orderdetailId, int ordertableId, int productId, int quantity)**: Creates a new **OrderDetail** object if the specified product exists and there is enough stock. It throws an **SQLException** if the product does not exist or if there is insufficient stock. This method updates the product's stock quantity and returns the newly created order detail.
3. **updateOrderDetail(int orderdetailId, int ordertableId, int productId, int quantity)**: Updates an existing **OrderDetail** object. This method validates the input parameters and uses **OrderDetailDao** to perform the update operation, returning the updated order detail.
4. **deleteOrderDetail(int orderdetailId)**: Deletes an order detail identified by its ID. This method calls **OrderDetailDao**'s **delete** method to remove the order detail from the database.
5. **getOrderDetailsByOrderId(int orderId)**: Retrieves a list of all order details associated with a specific order ID. This method calls **OrderDetailDao**'s **getOrderDetailsByOrderId** method and returns a list of **OrderDetail** objects.
6. **getOrderDetailsByProductId(int productId)**: Retrieves a list of all order details associated with a specific product ID. This method calls **OrderDetailDao**'s **getOrderDetailsByProductId** method and returns a list of **OrderDetail** objects.
7. **deleteOrderDetailsWithProductId(int productId)**: Deletes all order details associated with a specific product ID. This method fetches all relevant order details and deletes each one individually to ensure data consistency.
8. **deleteOrderDetailsWithOrderTableId(int ordertableId)**: Deletes all order details associated with a specific order table ID. This method fetches all relevant order details and deletes each one individually to maintain data integrity.

VII. The **ClientManager** class is responsible for managing client-related operations within the application. It provides methods to create, update, delete, and retrieve client information. The class uses **ClientDao** to interact with the database.

Important Fields

- **clientDao**: An instance of **ClientDao** used to perform CRUD operations on **Client** entities.

Important Methods

1. **Constructor (ClientManager())**: Initializes a new **ClientManager** with an instance of **ClientDao**. This setup ensures that the **ClientManager** can access all necessary data access functionalities for client management.
2. **createClient(String name, String address, String contactNumber)**: Creates a new **Client** object with the specified name, address, and contact number. It throws an **SQLException** if there is an error during the creation process. The method returns the newly created client.
3. **updateClient(int clientId, String name, String address, String contactNumber)**: Updates an existing **Client** object identified by the provided client ID with the new name, address, and contact number. It throws an **SQLException** if there is an error during the update process. The method returns the updated client.
4. **deleteClient(int clientId)**: Deletes a client identified by its ID. It calls the **delete** method of **ClientDao** to remove the client from the database and throws an **SQLException** if there is an error during the deletion process.
5. **getClient(int clientId)**: Retrieves a **Client** object identified by its ID. It calls the **read** method of **ClientDao** and returns the client object. An **SQLException** is thrown if there is an error during the retrieval process.
6. **getAllClients()**: Retrieves a list of all **Client** objects in the application. It calls the **readAll** method of **ClientDao** and returns a list of clients. An **SQLException** is thrown if there is an error during the retrieval process.

VIII. The **BillManager** class is responsible for managing the generation of bills within the application. It utilizes the **OrderManager**, **OrderDetailManager**, and **ProductManager** classes to gather necessary data and compute billing information.

Important Fields

- **orderManager**: An instance of **OrderManager** used to manage and retrieve order information.
- **orderDetailManager**: An instance of **OrderDetailManager** used to manage and retrieve order detail information.
- **productManager**: An instance of **ProductManager** used to manage and retrieve product information.

Important Methods

1. **Constructor (BillManager())**: Initializes a new **BillManager** with instances of **OrderManager**, **OrderDetailManager**, and **ProductManager**. This setup ensures that the **BillManager** can access all necessary data to generate bills.
2. **generateBills()**: Generates a list of all bills in the application. It retrieves all orders, fetches corresponding order details, and calculates the total number of products and the total price for each order. This method throws an **SQLException** if any database interaction fails. The method returns a list of **Bill** objects.

Method Details

generateBills():

- **Retrieves All Orders**: The method starts by calling **orderManager.getAllOrders()** to fetch a list of all orders.
- **Iterates Through Orders**: For each order in the list, it retrieves the associated order details using **orderDetailManager.getOrderDetailsByOrderId(order.getOrderTableId())**.
- **Calculates Bill Details**:
    - **Number of Products**: Initializes **noProducts** to zero and iterates through the order details to accumulate the total quantity of products.
    - **Total Price**: Initializes **totalPrice** to zero and iterates through the order details to compute the total cost based on product price and quantity.
- **Creates Bill Objects**: For each order, a new **Bill** object is created with the calculated number of products and total price, along with the order ID and client ID.
- **Adds to Bill List**: Each generated **Bill** is added to the list of bills.
- **Returns Bill List**: Finally, the method returns the complete list of bills.

IX. The **ClientGUI** class is responsible for providing a graphical user interface (GUI) for managing clients in the application. It leverages **ClientManager** and **OrderManager** to handle the necessary business logic for client operations. This class extends **JFrame** from the Swing library to create a window-based application.

Important Fields

- **clientManager**: An instance of **ClientManager** used for managing client data.
- **orderManager**: An instance of **OrderManager** used for managing order data related to clients.
- **tableGenerator**: An instance of **TableGenerator<Client>** used for generating tables from lists of clients.

Constructor

The constructor sets up the GUI components and their event listeners:

- **Title and Window Settings**: Sets the title, size, default close operation, and centers the window on the screen.
- **Input Fields**: Creates **JTextField** components for entering client information.
- **Combo Boxes**: Creates **JComboBox** components for selecting clients during update and delete operations.
- **Buttons**: Creates and configures buttons for creating, viewing, updating, and deleting clients, including setting their sizes and attaching **ActionListener** objects to handle button clicks.

X. The **GraphicInterface** class serves as the main graphical user interface (GUI) for the application. It extends **JFrame** and sets up a window with buttons that navigate to different parts of the application, such as client management, order management, and product management.

Important Fields and Components

- **Title and Window Settings**: Configures the window's title, size, default close operation, and centers it on the screen.
- **Title Label**: A **JLabel** component that displays the title "ORDER MANAGEMENT" at the top of the window.
- **Buttons**: Three **JButton** components for navigating to the **ClientGUI**, **OrderGUI**, and **ProductGUI** interfaces.

Constructor

The constructor initializes the GUI components and sets up their layout and behavior:

- **Title Label**: Creates a **JLabel** for the title, sets its font and alignment, and adds it to the panel.
- **Client Button**: Creates a button labeled "Client", sets its preferred size, and attaches an **ActionListener** that opens the **ClientGUI** when clicked.
- **Order Button**: Creates a button labeled "Order", sets its preferred size, and attaches an **ActionListener** that opens the **OrderGUI** when clicked.
- **Product Button**: Creates a button labeled "Product", sets its preferred size, and attaches an **ActionListener** that opens the **ProductGUI** when clicked.
- **Panel Layout**: Uses a **BoxLayout** for the main panel to arrange components vertically. Adds rigid areas to create space between components for better visual separation.
- **Grid Panel**: Wraps the main panel in a **JPanel** with **GridBagLayout** to center the panel in the window.
- **Visibility**: Sets the visibility of the frame to true at the end of the constructor to display the GUI.

XI. The OrderGUI class provides a graphical user interface (GUI) for managing orders within an application. This class extends JFrame and integrates various business layer components to handle order creation, updating, viewing, and deletion.

Important Fields and Components

- **Manager Instances: Instances of OrderManager, OrderDetailManager, ProductManager, ClientManager, and BillManager to handle business logic.**
- **Table Generators: Instances of TableGenerator for generating tables from lists of OrderTable, OrderDetail, and Bill objects.**
- **GUI Components: Various Swing components like JComboBox, JCheckBox, JTextField, JButton, and JLabel to interact with the user.**

Constructor

The constructor initializes the GUI components, sets up their layout, and defines the behavior for user interactions:

1. **Initialization:**
    - **Instances of manager classes and table generators are created.**
    - **The frame's title, size, close operation, and location are set.**
2. **Label and ComboBoxes:**
    - **A title label is created.**
    - **ComboBoxes for selecting orders, clients, and products are initialized and populated with data from the database.**
3. **Buttons and Their Actions:**
    - **Create Order Button: Creates a new order, optionally with an associated product, and checks stock availability.**
    - **See Orders Button: Displays all orders and their details in tables.**
    - **See Bills Button: Generates and displays bills for all orders.**
    - **Update Order Button: Updates an order by adding a new product to it, checking stock before updating.**
    - **Delete Order Button: Deletes the selected order from the database.**
4. **Panel Layout:**
    - **A main panel is created using BoxLayout to arrange components vertically.**
    - **Spacing is added between components for better visual separation.**
    - **Components are added to the panel in an organized manner.**
5. **Adding Panel to Frame:**

- **The main panel is added to a GridBagLayout panel for centering within the frame.**
- **The frame is set to visible at the end.**

XII. The **ProductGUI** class provides a graphical user interface for managing products within an application. It extends **JFrame** and integrates various business layer components to handle product creation, updating, viewing, and deletion.

Important Fields and Components

- **Manager Instances**: Instances of **ProductManager** and **OrderDetailManager** to handle business logic related to products and their order details.
- **Table Generator**: Instance of **TableGenerator** for generating tables from lists of **Product** objects.
- **GUI Components**: Various Swing components like **JTextField**, **JComboBox**, **JButton**, and **JLabel** to interact with the user.

Constructor

The constructor initializes the GUI components, sets up their layout, and defines the behavior for user interactions:

1. **Initialization**:
   - Instances of manager classes and the table generator are created.
   - The frame's title, size, close operation, and location are set.
2. **Label and ComboBoxes**:
   - A title label is created.
   - ComboBoxes for selecting products are initialized and populated with data from the database.
3. **Text Fields for User Input**:
   - Text fields for entering product details (name, description, price, stock quantity) for both creation and updating of products.
4. **Buttons and Their Actions**:
   - **Create Product Button**: Creates a new product using the details entered in the text fields.
   - **See Products Button**: Displays all products in a table.
   - **Update Product Button**: Updates the selected product with new details entered in the text fields.
   - **Delete Product Button**: Deletes the selected product from the database.
5. **Panel Layout**:
   - A main panel is created using **BoxLayout** to arrange components vertically.
   - Spacing is added between components for better visual separation.
   - Components are added to the panel in an organized manner.
6. **Adding Panel to Frame**:
   - The main panel is added to a **GridBagLayout** panel for centering within the frame.
   - The frame is set to visible at the end.

XIII. The **TableGenerator** class is a utility designed to create a table (**JTable** in Java Swing) from a list of objects of any type. This is especially useful in graphical user interfaces where you want to display a list of items in a tabular format.

Key Concepts

1. **Generics**: The class uses Java generics, which means it can handle lists of any object type (**T**). This makes it versatile and reusable across different parts of an application.
2. **Reflection**: The class utilizes Java's reflection API to inspect the fields of the objects in the list. Reflection allows the program to examine and modify the runtime behavior of applications, including accessing private fields.

1. **Determine Table Headers**:
    - The class inspects the fields of the first object in the list to determine the column headers for the table. For example, if the objects have fields **id**, **name**, and **price**, these will become the table headers.
    - Only non-static fields (those that belong to instances rather than the class itself) are included.
2. **Generate Table Data**:
    - For each object in the list, the class retrieves the values of its fields and organizes these values into rows.
    - Again, it only considers non-static fields and uses reflection to access their values.
3. **Create and Return the JTable**:
    - The class then creates a **JTable** using the collected headers and row data.
    - This table can then be used in a Swing application to display the data in a structured, tabular format.

5. Conclusions

The integration of a database through data access objects (DAOs) reflects a well-structured approach to data management. Each DAO class is responsible for CRUD operations related to its respective entity, promoting a clear separation between the business logic and data access layers. This design pattern not only enhances modularity but also simplifies testing and debugging by isolating database-related code.

The application demonstrates a good practice of exception handling, particularly in scenarios involving database interactions. By catching and handling SQL exceptions, the application ensures robustness and reliability, preventing crashes and providing meaningful error messages to users. This approach is critical for maintaining data integrity and providing a smooth user experience.

The design and implementation of the application are geared towards scalability and maintainability. The modular structure allows for easy addition of new features or entities without significant changes to existing code. For example, adding a new entity like Supplier would only require creating a new class and corresponding manager and DAO classes, without impacting other parts of the application. Additionally, the clear separation of concerns and adherence to OOP principles ensure that the codebase remains manageable as the application grows.

Overall, the assignment demonstrates a comprehensive understanding of software development best practices, including OOP design, efficient data structure usage, exception handling, and database integration. The application is well-architected, providing a solid foundation for further enhancements and scalability. The thoughtful design and implementation choices reflect a high level of competency in developing maintainable, robust, and efficient software solutions.

6. Bibliography

https://dsrl.eu/courses/pt/
https://dsrl.eu/courses/pt/materials/PT2024_A3.pdf
https://dsrl.eu/courses/pt/materials/PT2024_A3_S1.pdf
https://dsrl.eu/courses/pt/materials/PT2024_A3_S2.pdf