DOCUMENTATION

ASSIGNMENT *3*

STUDENT NAME: Chipirliu Denis
GROUP: 30421

CONTENTS

1. Assignment Objective

This assignment aims to create an Orders Management application using Java and relational databases. The application will facilitate processing client orders for a warehouse, adhering to a layered architecture pattern and utilizing specific class types.

Key Goals:

- **Object-Oriented Programming (OOP):** Implement the application using OOP principles, leveraging classes with clear responsibilities and relationships.
- **Layered Architecture:** Design the application with a layered architecture pattern for better organization and maintainability. The application should include at least four packages:
    - dataAccessLayer: Handles database interaction. ○ businessLayer: Contains the application logic for order creation and stock management.
    - model: Holds the data models (Client, Product, Order, Bill) and functionalities for populating tables.
    - presentation: Provides the graphical user interface (GUI).
- **Database Integration:** Utilize a relational database (e.g., MySQL) to store and manage client, product, and order data.
- **Client Management:** The application should offer functionalities for:
    - Adding new clients.
    - Editing existing client information. ○ Deleting clients.
    - Viewing a list of all clients in a user-friendly table format.
- **Product Management:** The application should provide functionalities for:
    - Adding new products with relevant information. ○ Editing existing product details. ○ Deleting products.
    - Viewing a list of all products in a user-friendly table format.
- **Order Management:** The application should allow users to: ○ Create new orders by selecting an existing client, choosing a product from the inventory, and specifying the desired quantity.
        - The application should validate order creation to ensure sufficient product stock exists and display an "under-stock" message if needed.
        - Upon successful order creation, the application should decrement the product's stock quantity to reflect the order fulfillment.
- **Data Access with Reflection:** Implement a generic class within the dataAccessLayer that utilizes reflection techniques to dynamically generate queries for interacting with the database across various tables (excluding the Log table).
- **Immutable Bill Class:** Define an immutable Bill class (using Java records) within the model package. Each order will generate a Bill object for logging purposes. Bill objects will only be inserted and read from the Log table; no updates are allowed.
- **GUI Design:** Develop a user-friendly GUI using Swing or JavaFX to facilitate user interaction with the application's functionalities.

2. Problem Analysis, Modeling, Scenarios, Use Cases

2.1 ***Problem Analysis***

This project addresses the inefficiency and potential errors associated with manual order processing in a warehouse environment. Manually managing orders can lead to delays, inaccurate stock levels, and difficulty fulfilling orders efficiently. Additionally, tracking client information and product details can become cumbersome with traditional methods.

2.2 ***Modeling***

To address these challenges, we will model the system using the following entities:

- ***Client:*** *A class representing a customer placing orders. Attributes could include client ID, name, contact information.*
- ***Product****: A class representing an item in the warehouse inventory. Attributes could include product ID, name, description, price, stock quantity.*
- ***Order:*** *A class representing a client's purchase request. Attributes could include order ID, client ID, product ID, quantity ordered.*

2.3 ***Scenarios***

These scenarios illustrate potential interactions with the system:

- **Client Management:** ○ A user adds a new client with relevant information. ○ A user edits an existing client's details.
  - A user deletes a client from the system (with confirmation).
- **Product Management:** ○ A user adds a new product to the inventory, specifying details.
  - ○ A user edits an existing product's information.
  - A user deletes a product from the inventory (with confirmation and potentially stock adjustment for existing orders).
- **Order Management:**
  - A user creates a new order by selecting a client, choosing a product, and specifying the desired quantity.
    - The system validates the order by checking if enough stock is available for the chosen product.
    - If sufficient stock exists, the order is created, and the product's stock quantity is decremented.
    - If stock is insufficient, the system displays an "under-stock" message, and the order is not placed.

2.4 **Use Cases**

These use cases categorize the system's functionalities from a user perspective:

- **Client Management:**
  - Add Client ○
    Edit Client ○
    Delete Client ○
    View All Clients
- **Product Management:**
  - Add Product ○
    Edit Product ○
    Delete Product ○
    View All Products
- **Order Management:**
  - Create Order ○
    View Orders

3. Design

3.1 **OOP Design**

The application will adhere to Object-Oriented Programming (OOP) principles for better organization and maintainability. Here's a breakdown of the key components:

- **Classes:**
    - Client: Represents a customer with attributes like ID, name, contact information, etc.
    - Product: Represents an item in the warehouse with attributes like ID, name, description, price, stock quantity, etc.
    - Order: Represents a client's purchase request with attributes like ID, client ID, product ID, quantity ordered, date, etc.
    - OrderManager: Handles order creation logic, including validation, stock management, and potentially interacting with the DataAccessLayer.
    - ClientManager: Provides functionalities for adding, editing, and deleting clients. o

        ProductManager: Provides functionalities for adding, editing, and deleting products.

    - DataAccessLayer: Manages interaction with the relational database (potentially using JDBC). This layer might contain generic functionalities for CRUD (Create, Read, Update, Delete) operations on various tables.
- **Inheritance:** Depending on the complexity, consider using inheritance if different types of clients or products require specific functionalities.
- **Reflection:** Using it for dynamic database access across various tables and dynamic table filling.

3.2 **UML Package and Class Diagrams**

Here's a high-level overview of the UML package diagram:

OrdersManagementSystem

+- model (package)

| +- Client.java

| +- Product.java

| +- Order.java

+- businessLayer (package)

| +- Validator.java (for every model class)

| +- OrderManager.java

| +- ClientManager.java

| +- ProductManager.java

+- dataAccessLayer (package)
| +- AbstractDAO.java

| +- OrderDAO.java

| +- ProductDAO.java

| +- ClietnDAO.java

+- presentation (package)

| +- Application

| +- Controller

3.3 **Used Data Structures**

- Clients, Products, Orders (potentially stored as objects in collections like ArrayLists or HashMaps depending on access needs).
- Relational database tables (Client, Product, Order) for persistent data storage.

4. Implementation

This section details the implementation aspects of the Orders Management application. We'll delve into the functionalities of key classes and the development of the graphical user interface (GUI).

4.1 Class Descriptions

- **AbstractDAO<T>**
  - **Fields:**
    - connection (Connection): Holds a database connection object.
  - **Methods:**
    - Abstract methods for CRUD (Create, Read, Update, Delete) operations:
    - create(T entity): Creates a new entity of type T in the database.
    - update(T entity): Updates an existing entity of type T in the database.
    - delete(T entity): Deletes an existing entity of type T from the database.
    - find(int id): Finds an entity of type T by its ID from the database.
- **Client**
  - **Fields:**
    - clientID (int): Unique identifier for the client.
    - name (String): Client's name.
    - contactInfo (String): Client's contact information (phone, email, etc.).
  - **Methods:**
    - Getters and setters for all fields.
    - (Optional) toString(): Returns a String representation of the client information.
- **Product**
  - **Fields:**
    - productID (int): Unique identifier for the product.
    - name (String): Product name.
    - description (String): Product description.
    - price (double): Price of the product.
    - stockQuantity (int): Current quantity of the product in stock.
  - **Methods:**
    - Getters and setters for all fields.

- (Optional) toString(): Returns a String representation of the product information.

- **Order** ○ **Fields:**
  - orderID (int): Unique identifier for the order.
  - clientID (int): Foreign key referencing the Client table.
  - productID (int): Foreign key referencing the Product table.
  - quantity (int): Quantity of the product ordered.
  - date (Date): Date the order was placed. ○ **Methods:**
  - Getters and setters for all fields.
  - (Optional) toString(): Returns a String representation of the order details.
- **OrderManager**
  ○ **Methods:**
  - createOrder(Client client, Product product, int quantity): Validates order details using OrderValidator, checks stock availability, creates an Order object, and interacts with OrderDAO to store the order in the database.
  - updateOrder(Order order): Validates updated order details, updates the order in the database using OrderDAO.
  - deleteOrder(Order order): Deletes the order from the database using OrderDAO (with confirmation).
  - viewOrders(): Retrieves and displays existing orders (potentially using OrderDAO).
- **ClientManager**
  ○ **Methods:**
  - addClient(Client client): Validates client information using ClientValidator, adds the client to the database using ClientDAO.
  - editClient(Client client): Validates updated client information, updates the client in the database using ClientDAO.
  - deleteClient(Client client): Deletes the client from the database using ClientDAO (with confirmation).
  - viewClients(): Retrieves and displays all clients (potentially using ClientDAO).
- **ProductManage**
  r ○ **Methods:**
  - addProduct(Product product): Validates product information using ProductValidator, adds the product to the database using ProductDAO.
  - editProduct(Product product): Validates updated product information, updates the product in the database using ProductDAO.
  - deleteProduct(Product product): Deletes the product from the database using ProductDAO (with confirmation and potential stock adjustment).
  - viewProducts(): Retrieves and displays all products (potentially using ProductDAO).

- **ProductValidato**
  r ○ **Methods:**
  - validateProductName(String name): Validates product name following specific rules (e.g., not empty).
  - validateProductPrice(double price): Validates product price to ensure a positive value.
  - (Optional) Additional methods for other product data validations.

- **ClientValidator**
  - **Methods:**
    - validateClientName(String name): Validates client name following specific rules.
    - validateClientContactInfo(String contactInfo): Validates client contact information format (e.g., email format).
    - (Optional) Additional methods for other client data validations.
- **OrderValidator**
  - **Methods:**
    - validateOrderQuantity(int quantity): Validates order quantity to ensure a positive value.
    - checkStockAvailability(Product product, int quantity): Checks if sufficient stock is available for the ordered quantity.
    - (Optional) Additional methods for order data validations.
- **OrderDAO** (Extends AbstractDAO<Order>)
  - **Methods:**
    - Implements CRUD operations specifically for Order entities:
    - createOrder(Order order): Creates a new order in the database.
    - updateOrder(Order order): Updates an existing order in the database.
    - deleteOrder(Order order): Deletes an order from the database.
    - findOrderById(int id): Finds an order by its ID from the database.
- **ClientDAO** (Extends AbstractDAO<Client>)
  - **Methods:**
    - Implements CRUD operations specifically for Client entities.
- **ProductDAO** (Extends AbstractDAO<Product>)
  - **Methods:**
    - Implements CRUD operations specifically for Product entities.
- **Application**
  - **Methods:**
    - (Optional) Main method: Entry point for the application.
    - initializeConnection(): Establishes a database connection.
    - createManagers(): Creates instances of manager classes (e.g., OrderManager, ClientManager).
    - startUI(): Starts the graphical user interface (GUI).
- **Controller**
  - **Methods:**
    - Handles user interactions with the GUI (e.g., button clicks, form submissions).
    - Delegates tasks to manager classes (e.g., createOrder(), updateClient()).
- **ConnectionFactory**
  - **Methods:**
    - getConnection(): Creates and returns a database connection object.
- **Logger**
  - **Methods:**

- info(String message): Logs

5. Conclusions

This project provided valuable insights into designing and implementing an Orders Management system. Here are some key takeaways:

- ***Object-Oriented Design:*** *The utilization of classes and their relationships promotes modularity, code reusability, and maintainability.*
- ***Data Access Layer:*** *Separating data access logic from the business layer ensures cleaner code and simplifies database interaction.*
- ***Business Layer:*** *Implementing a dedicated business layer facilitates the management of core functionalities like order creation, client management, and product management.*
- ***Validation:*** *Incorporating data validation ensures data integrity and prevents invalid entries in the system.*
- ***Graphical User Interface (GUI):*** *Developing a user-friendly GUI allows users to interact with the system efficiently.*

Learnings

Throughout this assignment, I:

- *Gained a deeper understanding of the design principles involved in building an Orders Management system.*
- *Improved my ability to analyze class diagrams and their relationships.*
- *Explored concepts like data access layers and business layer functionalities within an application.*
- *Recognized the importance of data validation for maintaining data integrity.*

Future Developments

This is a foundational design for an Orders Management system. Here are some potential areas for future development:

- ***Security:*** *Implementing user authentication and authorization to restrict access to sensitive data.*
- ***Reporting:*** *Generating reports on sales, inventory levels, and customer trends.*
- ***Integration with Payment Gateways:*** *Enabling online payments for a more streamlined checkout process.*
- ***Inventory Management:*** *Implementing features for managing stock levels, setting reorder points, and handling low stock situations.*
- ***Advanced Search and Filtering:*** *Enabling users to search for orders, clients, and products based on specific criteria.*

By incorporating these enhancements, the Orders Management system can become a more comprehensive and robust solution for managing orders, inventory, and customer relationships

6. 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