# DOCUMENTATION

## ASSIGNMENT 3

STUDENT NAME: COMAN ALECSIA-COSTINA
GROUP: 30423

## CONTENTS

## 1. Assignment Objective

The objective of this assignment is to develop an application that facilitates the management of orders within a warehouse setting. This application will utilize a relational database to store information about clients, products, and orders. The database management will be handled through the MySQL Workbench software.

The application provides a user-friendly interface that allows users to create, edit, and delete entries related to clients, products, or orders.

The design of the project adheres to the Layered Architecture pattern, which includes the following packages:

- **Model classes**: These represent the data models within the application.
- **Business Logic classes**: These encapsulate the core logic of the application.
- **Presentation classes**: These include the classes associated with the Graphical User Interface (GUI).
- **Data Access classes**: These represent the classes that interact with the database.

This architectural design promotes clear and efficient communication between the various components of the project.

In addition, the application employs Javadoc for the purpose of documenting the classes. It also utilizes reflection techniques to generate table data and queries. This approach enhances the maintainability and scalability of the application.

## 2. Problem Analysis, Modeling, Scenarios, Use Cases

- <u>PROBLEM ANALYSIS</u>

The primary problem that the application addresses is the management of orders within a warehouse setting. The challenges include tracking the status of orders, maintaining an updated inventory of products, and managing client information. The application aims to streamline these processes by providing a user-friendly interface and efficient database management.

- MODELLING

The application employs a relational database model. The entities in this model include Clients, Products, and Orders. Each entity is represented by a table in the database, with fields corresponding to the attributes of the entity. Relationships between these entities are represented by foreign keys.

**Clients**: This table includes fields such as ClientID, Name, Email, Phone.

**Products**: This table includes fields such as ProductID, Name, Price, and Quantity.

**Orders**: This table includes fields such as OrderID, ClientID, ProductID, Quantity.

Whenever a new order is created, the product's stock is checked, such that it's quantity must never be under 0. If it reaches null, the product must be deleted from the table. Also, when a client or a product is erased, all the orders regarding it must be deleted too. A bill can be generated by the user, creating a pdf which contains the information about all users' orders and their total amount to pay.

- SCENARIOS

Here are a few typical scenarios that the application can handle:

**Scenario 1**: A client places an order for a product. The application checks the availability of the product, updates the quantity of the product in stock, and records the order details.

**Scenario 2**: A new product is added to the inventory. The application allows the user to enter the product details and updates the Products table.

**Scenario 3**: A client's contact information is updated. The application allows the user to edit the client's details and updates the Clients table.

- USE CASES

**Use Case: Add Product**

**Primary Actor**: Employee

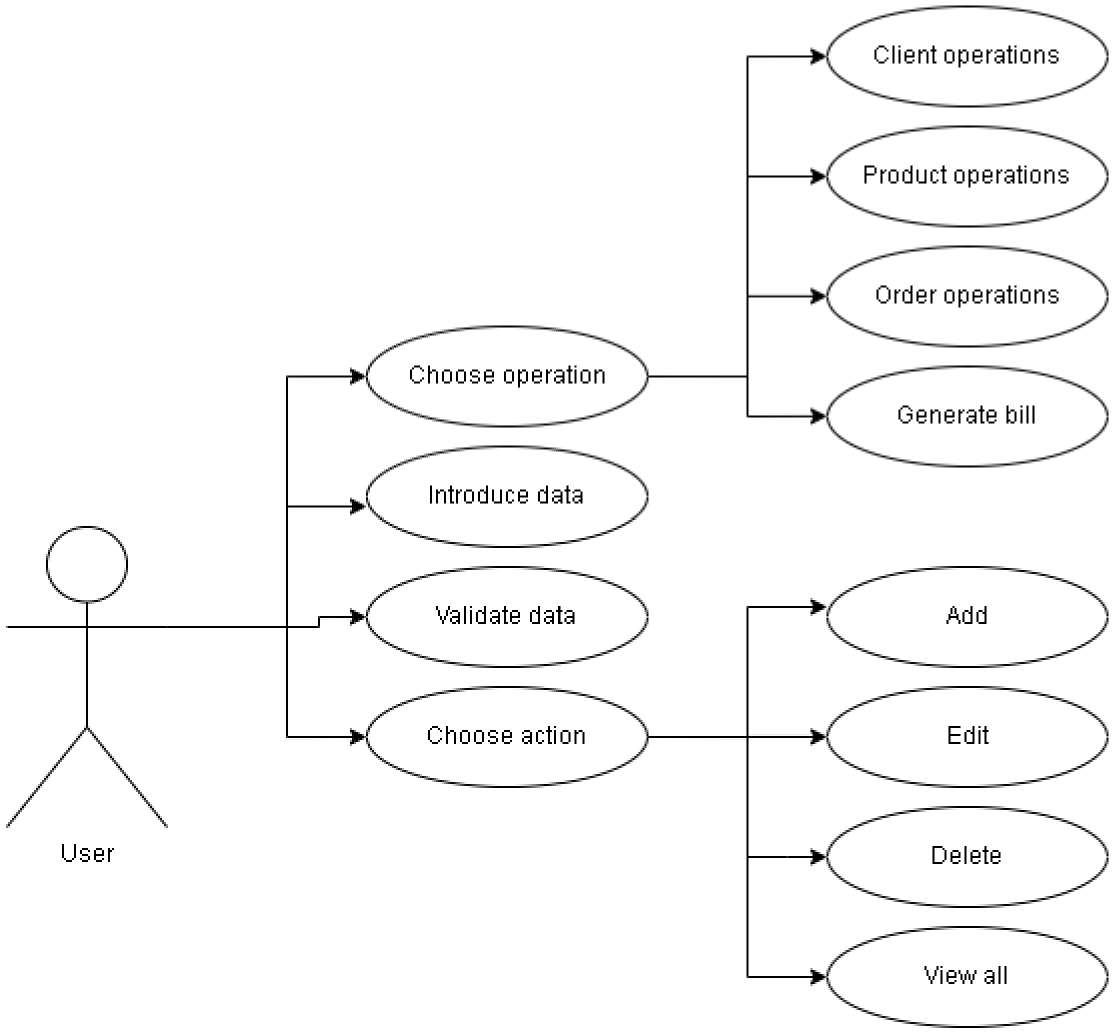
**Main Success Scenario**:

1. The employee selects the option to manage the products.
2. The application displays a form requesting the details of the product.
3. The employee enters the product's name, its price, and the current stock.
4. The employee clicks on the "Add Product" button.
5. The application stores the product data in the database and displays a confirmation message.

**Alternative Sequence**: Invalid values for the product's data

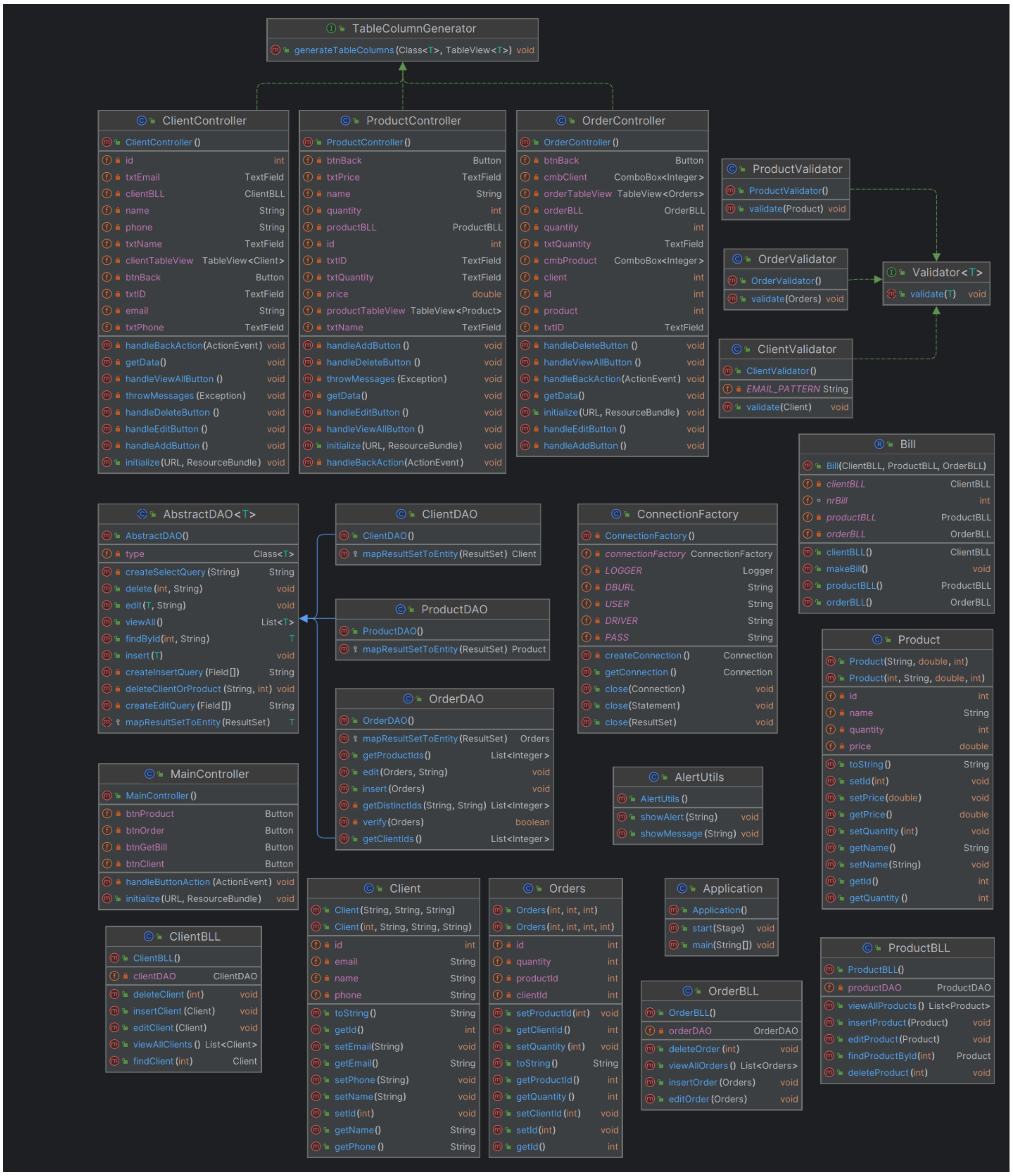
- The employee enters a negative value for the stock of the product.
- The application displays an error message and requests the employee to enter a valid stock value.
- The scenario returns to step 3.

This use case ensures that the application can handle the addition of new products effectively and validates the input data to maintain the integrity of the information in the database. It also provides feedback to the employee about the success or failure of the operation. This contributes to the overall usability and robustness of the application.



## 3. Design

- CLASS DIAGRAM



- OOP DESIGN

The object-oriented programming (OOP) design of the application follows key principles and concepts that are fundamental to software engineering:

*Encapsulation* is achieved by bundling the data (attributes) and methods (behavior) within each class. This ensures that the internal workings of each class are hidden from the outside world, promoting modularity and information hiding. All classes are built through the encapsulation principle, ensuring setters and getters for the fields.

*Inheritance* allows classes to inherit attributes and methods from other classes. This promotes code reusability and allows for the creation of hierarchical relationships between classes. For example, the **AbstractDAO** class serves as a base class for specific Data Access Object (DAO) implementations, while ClientDAO, ProductDAO and OrderDAO extend this class.

*Polymorphism* allows objects of different types to be treated as objects of a common superclass. This is achieved through method overriding and method overloading. For instance, the **mapResultSetToEntity()** method in the **AbstractDAO** class is overridden by subclasses to adapt to specific entity types, for each table.

*Composition* is the concept of creating complex objects by combining simpler ones. In your project, composition is utilized when classes contain instances of other classes as attributes. For example, the **ClientController** class composes an instance of the **ClientBLL** class to handle business logic operations related to clients. This logic is met in every BLL class.

- PACKAGES

Layered Architecture, also known as Layered Architecture or N-tier Architecture, is a software design pattern that organizes an application's components into distinct layers based on their responsibilities and functionalities. Each layer represents a different aspect of the application's logic and interacts with adjacent layers in a predefined manner. Here's a brief description of each layer:

1. **Presentation Layer:**

    - This layer is responsible for presenting information to the user and receiving user inputs.

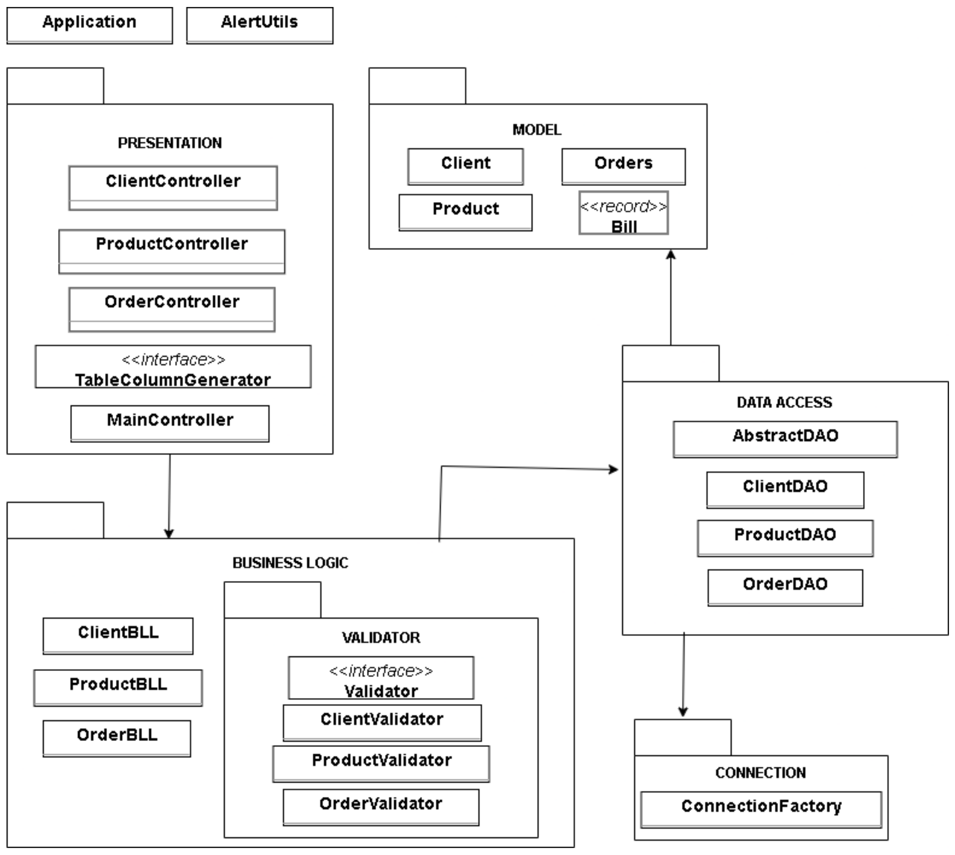
- It typically consists of graphical user interfaces (UI), including web pages, desktop applications, or mobile interfaces.
- The presentation layer communicates with the business layer to retrieve data and execute actions based on user interactions.

2. **Business Layer (or Logic Layer):**

- The business layer contains the core logic and rules of the application.
- It processes and validates data, executes business rules, and coordinates the application's workflow.
- This layer is independent of any specific user interface or data storage mechanism, making it reusable and adaptable to different presentation and data access layers.

3. **Data Access Layer**

- The data access layer manages the interaction with the underlying data storage systems, such as databases or external services.
- It includes components responsible for querying, updating, and persisting data to and from the data storage.
- This layer shields the rest of the application from the complexities of data storage mechanisms and provides a standardized interface for accessing data.



## 4. Implementation

**Client Class:**

- Represents a client entity with attributes such as ID, name, email, and phone.
- Provides constructors to initialize a client with or without an ID.
- Implements methods to retrieve and modify client attributes.
- Overrides the toString() method to provide a string representation of the client.

**Product Class & Orders Class:**

- Are build having the same logic as the Client Class, differentiating through the fields.
- The product has the following attributes: ID, name, price, quantity.
- The orders have the following properties: ID, product ID, client ID, quantity.

**Bill Class:**

- Generates a bill by fetching client, product, and order data using respective Business Logic Layer objects.
- Utilizes the iText library to create a PDF document for the bill.
- Iterates through client, order, and product lists to compile bill information.
- Calculates the total price for each client's orders and generates a PDF bill.
- Keeps track of the bill number for each generated bill.

**AbstractDAO Class:**

- An abstract Data Access Object (DAO) class responsible for handling CRUD (Create, Read, Update, Delete) operations on entities.
- Utilizes generics to operate on different types of entities, facilitating inheritance and code reusability.

- Provides methods for finding entities by ID, inserting new entities, updating existing entities, and deleting entities.
- Uses reflection to dynamically determine the type of the entity class.
- Implements methods to generate SQL queries for CRUD operations based on entity fields.
- Offers a method to map database query results to entity objects, which is override in each class, to adapt the entity to each specific attributes.
- Includes exception handling for database operations and utilizes an alert utility for user notification.
- Supports retrieving all entities from the database.

**ProductBLL, ClientBLL, OrderBLL:**

- Manages business logic related to products, such as finding, inserting, editing, and deleting products.
- Utilizes a **ProductDAO/ClientDAO/OrderDAO** object to interact with the data access layer for database operations.
- Implements methods for finding a product by ID, inserting a new product, editing an existing product, deleting a product, and viewing all products.
- Performs validation using a **ProductValidator/ClientValidator/OrderValidator** before inserting or editing a product.
- Throws a **NoSuchElementException** if a product is not found by ID during retrieval.
- Integrates with the **ProductDAO/ClientDAO/OrderDAO** to execute CRUD operations on the **Product** entity.
- Provides exception handling for cases where products are not found or validation fails.

**Validator Interface:**

- A generic interface for validating objects of any type.

**ClientValidator Class:**

- Implements the **Validator** interface for validating **Client** objects.

- Defines regular expression patterns for validating email addresses and phone numbers.
- Provides a method **validate(T t)** to validate client data.
- Utilizes regular expressions to check if the email and phone number meet the required criteria.
- Throws **IllegalArgumentException** with appropriate messages if validation fails.
- Uses **AlertUtils.showAlert()** to display alerts for invalid email or phone number

## 5. Results

As final results, the user can generate a bill which incorporates all clients' bills. It contains the client's name, products ordered, their price and total amount to pay. This utility assures a better understanding of the database.

## 6. Conclusions

This assignment aims to teach students how to use reflection and work with database by incorporating a graphical user interface (GUI), the goal is to create a more interactive database that is easier to use. If it included more complex queries and had better stability, it could even be utilized by small businesses, as it operates similarly to monthly subscription applications that store owner's purchase.

## 7. Bibliography

Programming Techniques – Lectures of prof. Cristina POP

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

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

Creating PDF files in Java: https://www.baeldung.com/java-pdf-creation

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