

DOCUMENTATION

ASSIGNMENT 3

STUDENT NAME: Doia Bogdan-Mihai
GROUP: 30421

CONTENTS

1.	Assignment Objective	3
2.	Problem Analysis, Modeling, Scenarios, Use Cases.....	3
3.	Design	5
4.	Implementation	5
5.	Results.....	6
6.	Conclusions.....	10
7.	Bibliography	10

1. Assignment Objective

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

Sub-objectives:

- Analyze the problem and identify requirements
- Design the orders management application
- Implement the orders management application
- Test the orders management application
- Design a dedicated GUI

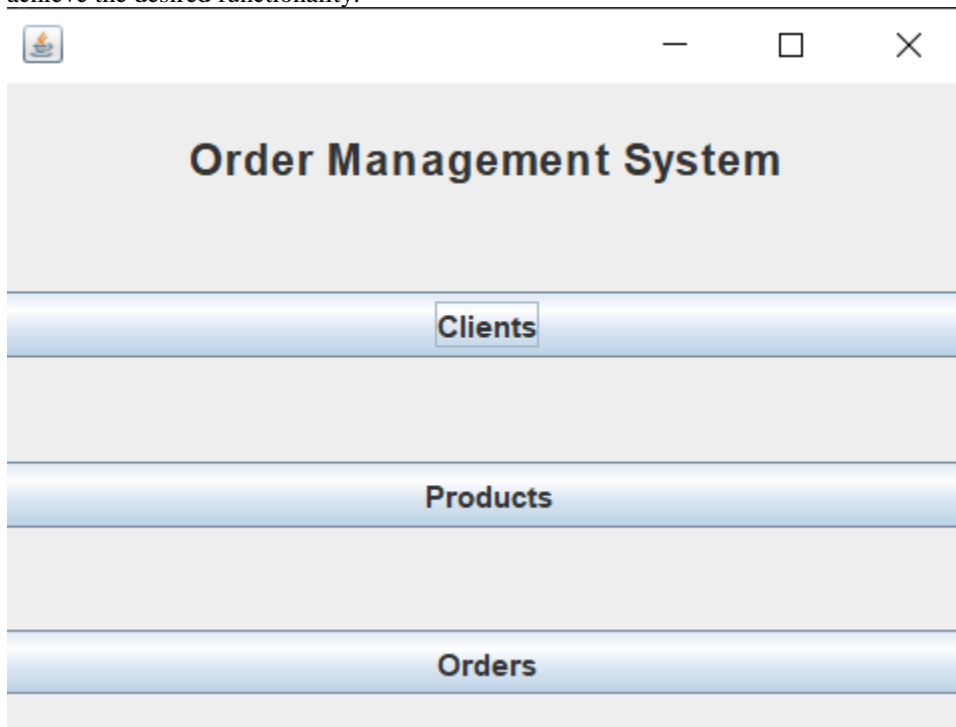
2. Problem Analysis, Modeling, Scenarios, Use Cases

Problem Analysis:

The problem at hand is to develop an Orders Management application for processing client orders in a warehouse. The application will utilize relational databases to store information about products, clients, and orders. The application should follow a layered architecture pattern, consisting of Model classes, Business Logic classes, Presentation classes, and Data access classes..

By implementing the layered architecture pattern and organizing the application into these classes, we can ensure a separation of concerns and maintainability of the codebase. Each layer will have its specific responsibilities, allowing for easier development, testing, and maintenance of the application.

The problem analysis phase sets the foundation for designing and implementing the Orders Management application. It helps identify the key components, their relationships, and their roles within the application. The next steps would involve designing the specific classes, their interfaces, and defining the interactions between them to achieve the desired functionality.



Modeling: The Orders Management application is designed to facilitate the processing of client orders for a warehouse. It follows a layered architecture pattern and utilizes relational databases for storing the necessary data. The application consists of the following key models:

Product Model: Represents the products available in the warehouse. It includes attributes such as product ID, name, quantity, price, and other relevant information.

Client Model: Represents the clients who place orders. It includes attributes such as client ID, name, contact information, and any other necessary details.

Order Model: Represents an order placed by a client. It includes attributes such as order ID, client ID (foreign key), product ID (foreign key), quantity, total price, and order date.

Scenarios and use cases:

Use Case: add product

Primary Actor: employee

Main Success Scenario:

1. The employee selects the option to add a new product
2. The application will display a form in which the product details should be inserted
3. The employee inserts the name of the product, its price and current stock
4. The employee clicks on the "Add" button
5. The application stores the product data in the database and displays an acknowledge message

Alternative Sequence: Invalid values for the product's data

- The user inserts a negative value for the stock of the product
- The application displays an error message and requests the user to insert a valid stock
- The scenario returns to step 3

Use Case: Update Product

Primary Actor: employee

Main Success Scenario:

1. The employee selects the option to update a product.
2. The application displays a form showing the existing product details.
3. The employee modifies the name, price, or stock of the product.
4. The employee clicks on the "Update" button.
5. The application updates the product data in the database and displays an acknowledgement message.

Alternative Sequence: The employee inserts invalid values for the product's data (e.g., negative price).

The application displays an error message and requests the user to insert valid data.
The scenario returns to step 3.

Use Case: place order

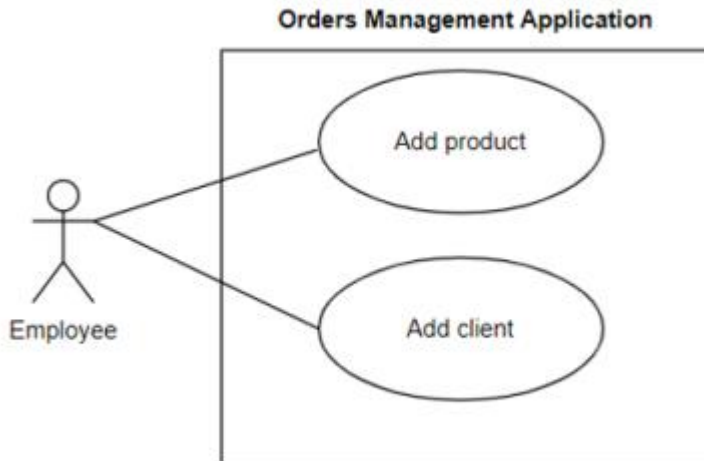
Primary Actor: customer

Main Success Scenario:

1. The customer selects the option to place an order.
2. The application displays a list of available products.
3. The customer selects the desired products and specifies the quantities.
4. The customer clicks on the "Place Order" button.
5. The application validates the order details and calculates the total price.
6. The application creates an order record in the database with the customer's information and order details.
7. The application displays an order confirmation message to the customer.

Alternative Sequence: Insufficient Stock for Products

- The customer selects a quantity for a product that exceeds the available stock.
- The application displays an error message indicating insufficient stock.
- The scenario returns to step 3 for the customer to adjust the quantities or select different products.



3. Design

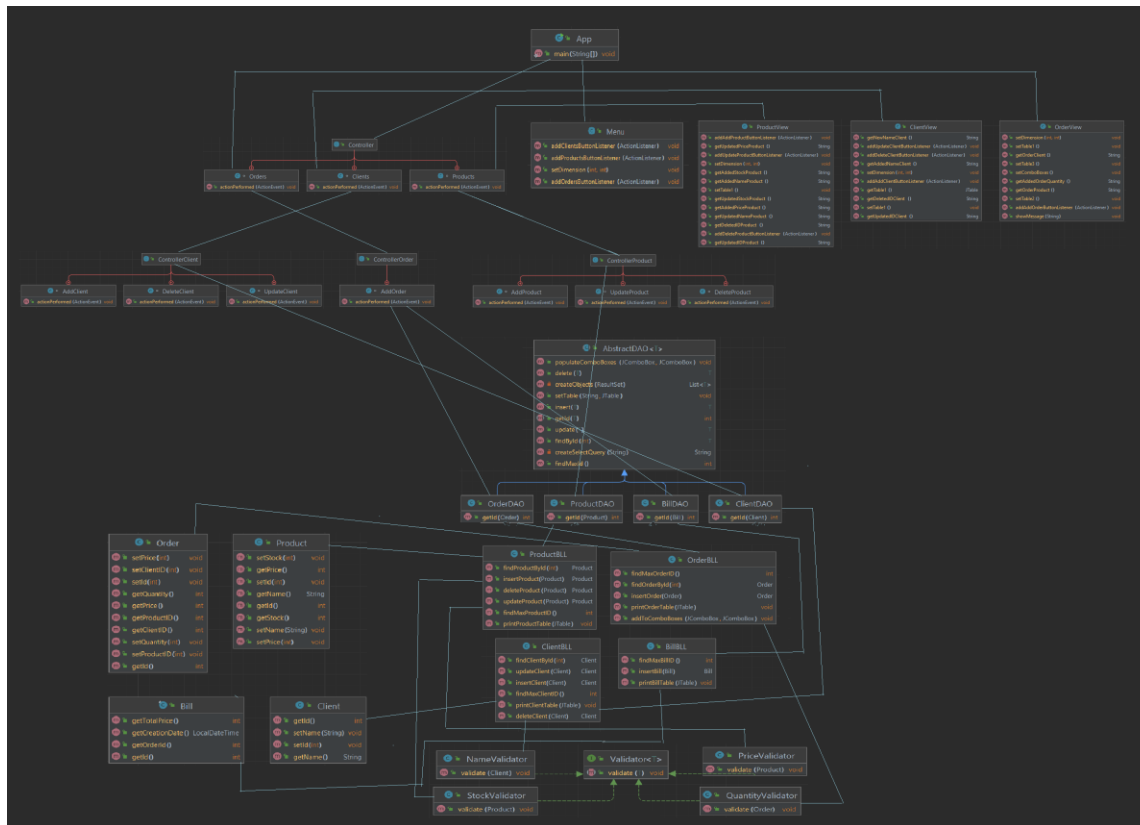
OOP design of the application

The application is designed using Object-Oriented Programming (OOP) principles to ensure modularity, reusability, and maintainability. The design follows a layered architecture pattern, separating different concerns into distinct classes.

The OOP design promotes encapsulation, where each class encapsulates its own data and behavior. It also facilitates code reuse through inheritance and composition, allowing for extensibility and flexibility in adding new features or functionalities. Additionally, the design adheres to SOLID principles, promoting loose coupling and high cohesion between classes.

Overall, the OOP design of the application ensures a modular and scalable structure, making it easier to maintain and enhance the system as requirements evolve over time..

UML classes diagram:



4. Implementation

App: The App class serves as the entry point for the application. It initializes the necessary components and starts the execution of the program. In this class, the main method is responsible for instantiating the Menu view and making it visible to the user. It also creates an instance of the Controller class and associates it with the Menu view.

Validator: The Validator interface represents a contract for validating objects of type T. It provides a single method validate that accepts an object and performs the validation logic specific to that object type. Implementations of this interface will define their own validation rules and logic. The **NameValidator** class implements the Validator interface and is responsible for validating the name of a Client object. It uses a regular expression pattern to check if the name consists of only letters and spaces. If the name is not valid, an IllegalArgumentException is thrown with an appropriate error message. The **PriceValidator** class is responsible for validating the price of a Product object. It implements the Validator interface. The validate method takes a Product object as a parameter and checks if the price falls within the specified range. If the price is not within the valid range, an IllegalArgumentException is thrown with an error message indicating that the price range is not respected. The minimum price is set to 0 and the maximum price is set to 100. The **QuantityValidator** class is responsible for validating the quantity of an Order object. It implements the Validator interface. The validate method takes an Order object as a parameter and checks if the quantity falls within the specified range. If the quantity is not within the valid range, an IllegalArgumentException is thrown with an error message indicating that the quantity capacity is not respected. The minimum quantity is set to 0 and the maximum quantity is set to 100. The **StockValidator** class is responsible for validating the stock of a Product object. It implements the Validator interface. The validate method takes a Product object as a parameter and checks if the stock falls within the specified range. If the stock is not within the valid range, an IllegalArgumentException is thrown with an error message indicating that the stock capacity is not respected. The minimum stock is set to 0 and the maximum stock is set to 100.

The **ClientBLL** class represents the business logic for handling Client operations. It contains methods for finding a client by ID, inserting, updating, and deleting a client. It also provides methods for finding the maximum client ID and printing the client table to a JTable. The class utilizes a list of validators to validate client data before performing operations. It interacts with the ClientDAO class for database access. The class is responsible for throwing appropriate exceptions when necessary. The **BillBLL** class represents the business logic for handling Bill operations. It contains methods for inserting a new bill, printing the bill table to a JTable, and finding the maximum bill ID. The class utilizes a list of validators, although it is currently empty in the provided code. It interacts with the BillDAO class for database access. The class is responsible for throwing appropriate exceptions when necessary. The **OrderBLL** class represents the business logic for handling Order operations. It contains methods for finding an order by ID, inserting a new order, printing the order table to a JTable, adding client and product data to JComboBoxes, and finding the maximum order ID. The class utilizes a list of validators, specifically a QuantityValidator, to validate the order before insertion. It interacts with the OrderDAO class for database access. The class is responsible for throwing appropriate exceptions when necessary. The **ProductBLL** class represents the business logic for handling Product operations. It contains methods for finding a product by ID, inserting a new product, updating a product, deleting a product, finding the maximum product ID, and printing the product table to a JTable. The class utilizes a list of validators, specifically a PriceValidator and a StockValidator, to validate the product before insertion or update. It interacts with the ProductDAO class for database access. The class is responsible for throwing appropriate exceptions when necessary.

AbstractDAO class is an abstract class that provides common functionality for accessing and manipulating database tables. It is a generic class that can be extended by specific DAO (Data Access Object) classes for different entities. The **BillDAO** class extends the AbstractDAO class and provides a concrete implementation for the getId method specific to the Bill entity. This class can be used to perform database operations related to bills. The **ClientDAO** class is a data access layer class that handles the retrieval and manipulation of Client data in the database. It extends the AbstractDAO class and provides specialized functionality for the Client model. The class includes a method to retrieve the ID of a Client object. The **OrderDAO** class is a data access layer class that handles the retrieval and manipulation of Order data in the database. It extends the AbstractDAO class and provides specialized functionality for the Order model. The **ProductDAO** class is a data access layer class that handles the retrieval and manipulation of Product data in the database. It extends the AbstractDAO class and provides specialized functionality for the Product model.

The **Bill** class represents a bill object in the system. It contains information such as the bill ID, associated order ID, creation date, and total price. The class provides methods to retrieve the bill's attributes. The **Client** class represents a client object in the system. It contains information such as the client ID and name. The class provides methods to retrieve and modify the client's attributes. The **Order** class represents an order object in the system. It contains information such as the order ID, client ID, product ID, quantity, and total price. The class provides methods to retrieve and modify the order's attributes. The **Product** class represents a product in the system and provides methods to retrieve and modify its attributes, such as the product ID, name, price, and stock quantity.

The **ControllerClient** class handles the interactions between the ClientView and the ClientBLL. It listens to events triggered by the buttons in the ClientView and performs corresponding actions such as adding, deleting, and updating clients in the database. The **Controller** class handles the user interface interactions by listening to events triggered by the menu buttons. It creates the corresponding views and controllers for clients, products, and orders. The **ControllerOrder** class is responsible for handling interactions between the OrderView and the business layer components. It listens to events triggered by buttons in the OrderView and performs corresponding actions. It allows adding new orders by creating Order objects and inserting them into the database through the OrderBLL. It also handles the creation of bills and updates the product stock based on the ordered quantity. The class ensures that there is enough stock available before placing an order and provides feedback to the user if stock is insufficient. The **ControllerProduct** class, which acts as a controller between the ProductView and the business layer components. It provides event listeners for adding, deleting, and updating products based on the user's interactions with the ProductView. The class retrieves the necessary data from the view, performs the corresponding actions using the ProductBLL class, and updates the table in the ProductView to reflect the changes made to the product data.

The **ClientView** class is a graphical user interface (GUI) component that allows for the management of clients. It extends the JFrame class and provides various methods to interact with the client-related components and retrieve user input. This class is responsible for displaying client information in a table, retrieving client data from the ClientBLL class, and capturing user actions such as adding, deleting, and updating clients. It serves as the user interface for managing clients in the application. The **Menu** class represents the main menu of the application's graphical user interface (GUI). It extends the JFrame class and provides buttons for navigating to different sections of the application, such as clients, products, and orders. The **OrderView** class represents the view for managing orders in the application's graphical user interface (GUI). It provides various components such as tables, combo boxes, and buttons to interact with orders. Users can add new orders, view order details, and perform related operations. The **ProductView** class represents the view for managing products in the application's graphical user interface (GUI). It provides various components such as tables, buttons, and text fields to interact with products. Users can add new products, delete existing products, update product information, and view product details.


5. Results

The menu appears first and the user can select one of the 3 options by clicking the buttons Clients, Products, Orders, upon pressing one of them a new interface will appear, in which the buttons on the left represent the options. Then the user can input the data he wants to add, update or delete and he can visualize the database table on the right of the interfaces in a table.

First menu:



The Clients interface:



—

□

×

Add

Client to add name:

Delete

Client to delete ID:

Update


Client to update ID:

New name:

Clients

id	name
1	Ana
4	Carlos
2	Bobby

The products interface:



—

□

×

Add

Product to add name:

Product to add price:

Product to add stock:

Delete

Product to delete ID:

Update

Product to update ID:

New name:

New price:

New stock:

Product

id	name	price	stock
2	Apa minerala	4	10
4	Ciocolata alba	8	5
1	Apa plata	5	4
6	Cartofi dulci	12	8

The orders interface:

The interface is titled "Orders" and contains a form for adding new orders. The form includes a "Client" dropdown menu with "1" selected, a "Product" dropdown menu with "2" selected, and a "Quantity" input field. An "Add" button is located below the form. To the right of the form is a table titled "Orders" with columns: id, clientid, productid, quantity, and total_price. Below the form is a table titled "Bills" with columns: id, orderid, creation_date, and total_price. To the right of the "Add" button is a table titled "Products" with columns: id, name, price, and stock.

id	clientid	productid	quantity	total_price
1	2	2	2	8
2	1	4	10	80
3	4	1	6	30
4	1	6	3	36
5	1	6	4	48
6	2	6	2	24

id	orderid	creation_date	total_price
1	1	2023-05-19 0...	8
2	2	2023-05-19 0...	80
3	3	2023-05-19 0...	30
4	6	2023-05-19 0...	24

id	name	price	stock
2	Apa minerala	4	10
4	Ciocolata alba	8	5
1	Apa plata	5	4
6	Cartofi dulci	12	8

6. Conclusions

While working on this assignment I had the opportunity to consolidate my Java and OOP knowledge and also discover new features that IntelliJ and the Java programming language provide the user. I also learned and understood a lot more about the concept of databases, valuable in many fields and for many subjects at the university too. It has also taught me how to better manage my time and how to approach a project, things very valuable in any field or circumstance not just programming.

There are several potential future developments that could enhance the order managing system:

- **User Authentication and Authorization:** Implement a user authentication and authorization system to secure the application. This would involve user registration, login functionality, and role-based access control to ensure that only authorized users can access and modify the order data.
- **Enhanced Data Validation:** Improve data validation and error handling mechanisms to provide more specific and informative error messages. Validate user inputs to ensure they meet the required format and constraints, such as validating order quantities, client details, and product information.
- **Order Status Tracking:** Implement functionality to track the status of orders throughout their lifecycle. This could include different stages such as "pending," "processing," "shipped," and "delivered," allowing users to easily monitor and manage the progress of each order.
- **Reporting and Analytics:** Add reporting and analytics capabilities to generate insights from order data. Provide features such as order statistics, sales reports, and inventory management reports. This would enable users to make data-driven decisions and identify trends, patterns, and potential areas for improvement.

7. Bibliography

1. <https://jenkov.com/tutorials/java-reflection/index.html>
2. <https://www.baeldung.com/javadoc>
3. <https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html>
4. <https://mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>
5. <https://regex101.com/>