DOCUMENTATION

ASSIGNMENT *3*

STUDENT NAME: Borzan Călina-Annemary
GROUP:30424

CONTENTS

1. Assignment Objective

Main Objective

The assignment's main objective is to develop and execute an application that manages the client orders for a warehouse by simulating three tables: client one, product one, order one, in which we can see how clients/product are being added, how an order is made and how the products quantity changes live.

Sub-Objectives

Sub-Objective	Description	Section
Analyze the problem and Identify requirements	This phase involves a comprehensive analysis of the requirements outlined for the queues management application. The goal is to dissect and understand the intricate details of the problem statement, focusing on key aspects such as client relational database, layered architecture, and system optimization. Through this analysis, we aim to identify the core functionalities and constraints of the application, including the simulation of the product quantity, bill generation , and order management strategies.	Problem Analysis, Modeling, Scenarios, Use Cases
Design the orders management application	Outline the way the architecture of the system will look like using object-oriented design principles: UML package, class diagrams, data structure, interfaces and designing the database schema.	Design
Implement the orders management application	In this phase, the focus is on implementing the core functionality of the warehouse application. We will decide on the approach for handling various aspects such as using reflection and generic in creating a method that contains the DB operations:add, edit, delete, find includes developing the Model classes, Business Logic classes, Presentation classes, and Data Access classes.	Implement

Develop the graphical interface	Create the GUI with which the user will interact.	Implement
Test the orders management application	Analyze the outcome of the project and find certain future developments that could be implemented.	Conclusion

2. Problem Analysis, Modeling, Scenarios, Use Cases

2.1. Functional Requirements:

- The system should manage the clients, products, and orders, which will have bill generated automatically.
- The system shall allow users to select a client from the client list.
- The system shall provide a form pre-populated with the selected client's current details.
- The system shall allow users to modify the client details.
- The system shall validate the modified data before saving.
- The system shall update the client information in the database and refresh the client list in the GUI.
- The system shall display a confirmation message upon successful update.
- The system shall allow users to select a client from the client list.
- The system shall prompt for confirmation before deleting the client.
- The system shall delete the client from the database if no references exist in other tables.
- The system shall update the client list in the GUI to remove the deleted client.
- The system shall display an error message if the client cannot be deleted due to database constraints.
- The system shall provide a form to input product details including product name, quantity, and price.
- The system shall validate the input data (e.g., price as a positive decimal, quantity as a positive integer).
- The system shall save the new product information to the database upon successful validation.
- The system shall update the product list in the GUI to include the new product.
- The system shall display a confirmation message upon successful addition.
- The system shall allow users to select a product from the product list.

- The system shall provide a form pre-populated with the selected product's current details.
- The system shall allow users to modify the product details.
- The system shall validate the modified data before saving.
- The system shall update the product information in the database and refresh the product list in the GUI.
- The system shall display a confirmation message upon successful update.
- Delete Product
- The system shall allow users to select a product from the product list.
- The system shall prompt for confirmation before deleting the product.
- The system shall delete the product from the database if no references exist in other tables.
- The system shall update the product list in the GUI to remove the deleted product.
- The system shall display an error message if the product cannot be deleted due to database constraints.
- The system shall provide a form to input order details including client ID, product ID, and quantity.
- The system shall validate the input data (e.g., quantity as a positive integer).
- The system shall check the availability of the specified product quantity.
- The system shall create a new order in the database upon successful validation.
- The system shall update the order list in the GUI to include the new order.
- The system shall update the product quantity in the database.
- The system shall generate a bill for the order and save it to the database.
- The system shall display a confirmation message upon successful order creation.
- The system shall provide a view to list all orders.
- The system shall allow users to view all bills in a separate window.

2.2. Use Case

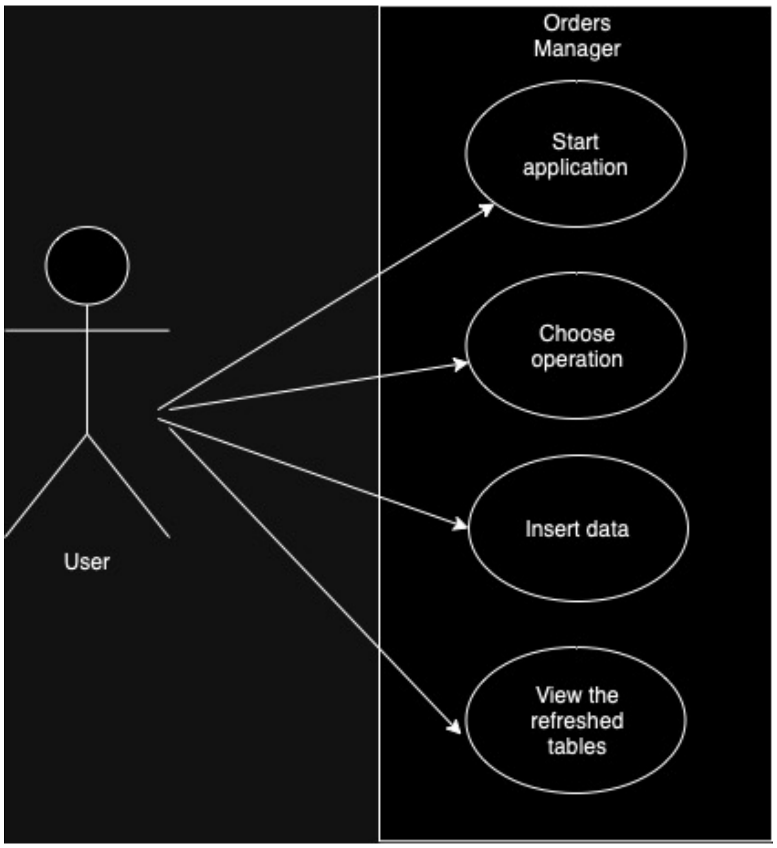


Figure 1:Use Case Diagram

Use Case Description:

1. **Start Application:**

Goal: Initialize the application and display the main user interface.
Explanation: Launch the application, setting up all UI components and preparing for user interactions.

2. **Choose Operation:**

Goal: Select the desired operation (e.g., add, edit, delete) for clients, products, or orders.
Explanation: Navigate through the UI to choose the specific action to perform.

3. **Insert Data:**

Goal: Enter the necessary information into the provided form fields.
Explanation: Fill out the form with relevant data required for the selected operation.

4. **View the Refreshed Tables:**

Goal: Update the display to reflect the latest data changes.
Explanation: Refresh the tables in the UI to show the most recent information, confirming the successful completion of the operation.

3. Design

In the orders management application, a layered architecture is employed to ensure modularity, reusability, and ease of maintenance. The application is structured into distinct layers, each responsible for a specific aspect of the application's functionality. This layered approach enhances separation of concerns and promotes a clean and maintainable codebase.

Layered Architecture Overview

The application is divided into several layers:

Presentation Layer: This layer handles the user interface and user interactions.

Business Logic Layer: This layer encapsulates the core functionality and business rules.

Data Access Layer: This layer manages data storage and retrieval operations.

Model Layer: This layer defines the data structures used throughout the application.

3.1. OOP Design

Encapsulation

Encapsulation is the mechanism of bundling data and methods that operate on the data into a single class while restricting access to some of the object's components. Each layer in the application encapsulates related data and behavior within its scope.

Model Layer: Classes like Client, Product, Order, and Bill encapsulate attributes and methods relevant to their respective entities. These classes expose methods like getFirstName(), getLastName(), and getEmail() in the Client class, allowing controlled interaction with the client's data.

Data Access Layer: DAO (Data Access Object) classes such as ClientDAO, ProductDAO, OrderDAO, and BillDAO encapsulate the logic for interacting with the database. They provide methods like insert(), delete(), and findById(), controlling access to the underlying data store.

Business Logic Layer: Classes like ClientBL, ProductBL, and OrderBL encapsulate business rules and logic. They interact with the DAO classes to perform operations like adding, editing, and deleting clients, products, and orders while ensuring that business rules are enforced.

Presentation Layer: The View and Controller classes encapsulate the GUI and user interaction logic. For example, the View class manages the layout and display of data, while the Controller class handles user actions and coordinates between the view and business logic layers.

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling code to operate on objects of various types without needing to know their specific class.

Validation: In the business logic layer, polymorphism is used through interfaces and their implementations. For instance, the Validator interface defines a contract for validation. Classes like EmailValidator, ClientValidator, ProductValidator, and QuantityValidator implement this interface, providing specific validation logic.

Strategy Pattern: If there were different strategies for processing orders, they could be defined using an interface and multiple implementations. This would allow the OrderBL class to use different order processing strategies interchangeably.

Composition

Composition is a design principle where a class contains objects of other classes as member variables, allowing for building complex objects by combining simpler ones.

Business Logic Layer: The OrderBL class demonstrates composition by creating instances of OrderDAO, ProductDAO, ClientDAO, and BillDAO. These instances are used to perform various operations related to orders, products, clients, and bills.

Presentation Layer: The View class contains instances of JFrame, JTable, JButton, and other UI components, composing the overall user interface from these individual components.

Modularity

Modularity refers to the division of the application into distinct modules or packages that can be developed, maintained, and understood independently.

Package Structure: The application is divided into multiple packages:

model package contains the data models (e.g., Client, Product, Order, Bill).

data package contains the DAO classes for data access (e.g., ClientDAO, ProductDAO, OrderDAO, BillDAO).

business package contains the business logic classes (e.g., ClientBL, ProductBL, OrderBL).

presentation package contains the GUI classes (e.g., View, Controller).

This modular structure allows for easier maintenance and scalability, as changes in one package do not affect others.

3.2. UML Packages

Packages play a critical role in the architectural design of a software application by allowing better organization and enhancing modularity. In UML diagrams, packages are depicted with their contained elements, classes, and interfaces to avoid clutter. In our program, they helped organize the application into six main parts:

1. **Model Package**
   This package represents the foundation of the orders management system, containing the essential data structures like Client, Product, Order, and Bill classes. It serves as the basis for the system, containing the classes that are the core representations

2. **Data Package**
   This package manages data storage and retrieval operations through Data Access Objects (DAOs).

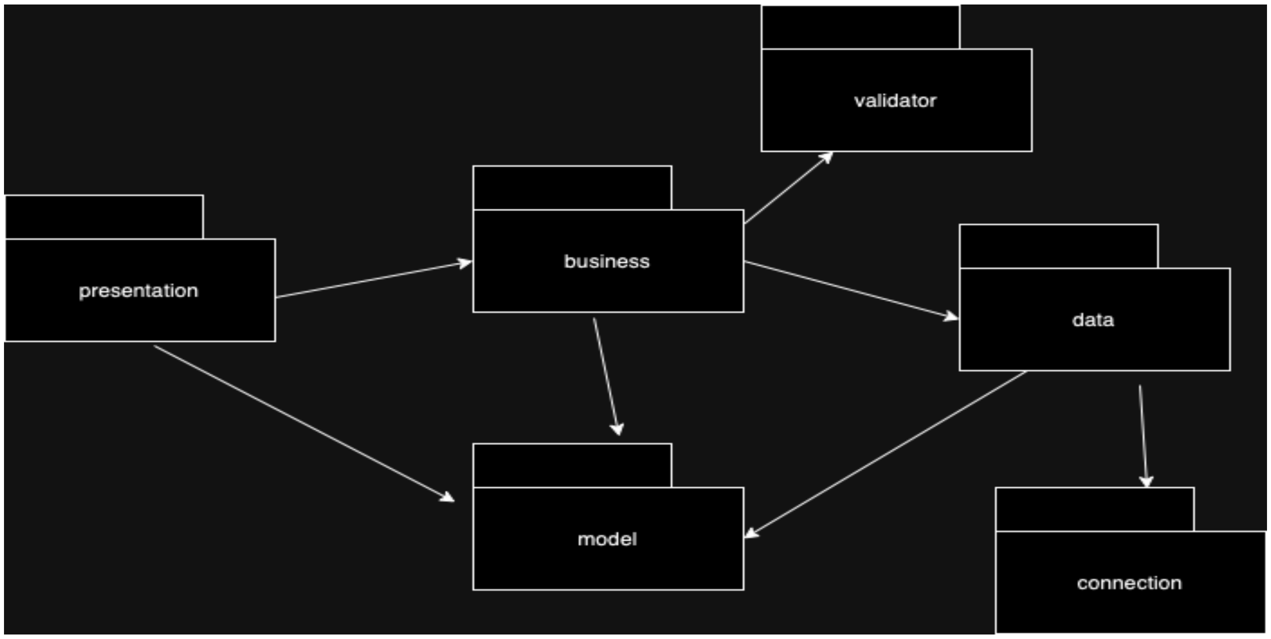
3. **Business Package**
   This package represents the operational logic of the program. It includes classes that encapsulate the business rules and logic for handling operations related to clients, products, and orders.

4. **Connection Package**
   This package handles connections and transactions, coordinating interactions between the business logic and data access layers.

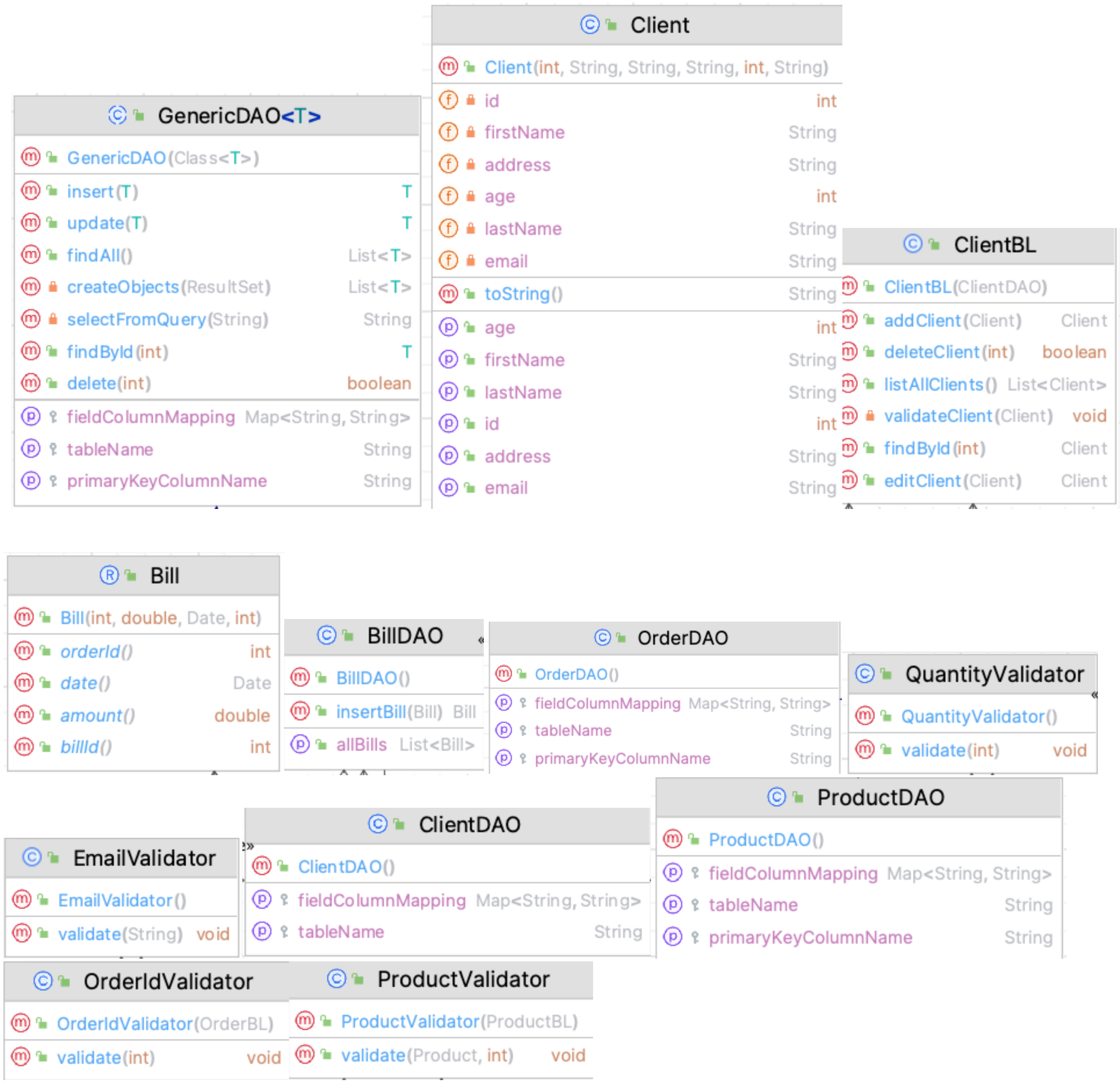
5. **Validator Package**
   This package provides various validation utilities to ensure data integrity and correctness.

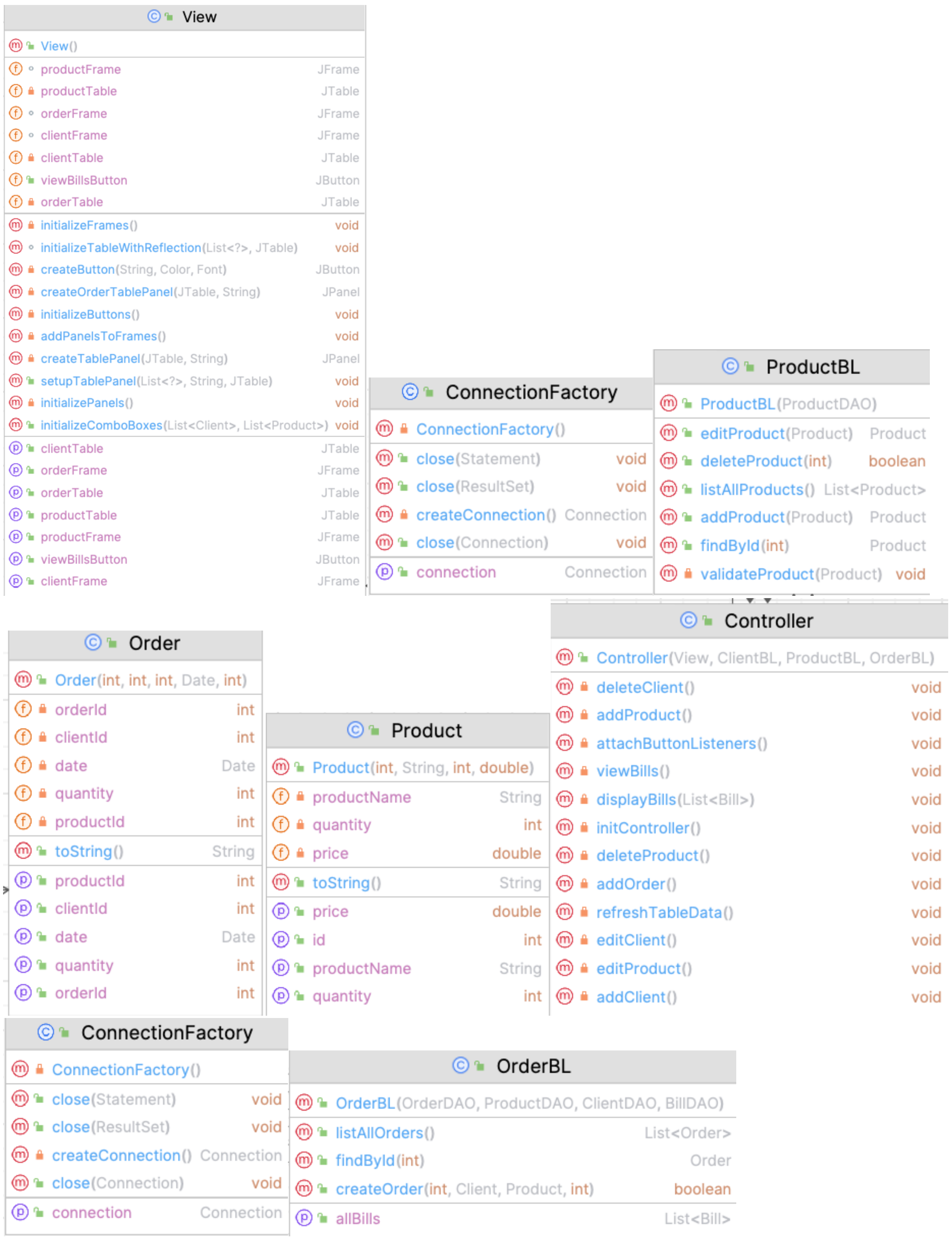
6. **Presentation Package**
   This package is dedicated to the user interface, managing how users interact with the system.

The Presentation package depends on the Business package. The Business package depends on the Data and Model packages, as well as the Validator package. The Connection package manages interactions between the Business and Data packages. The Validator package depends on the Bussiness package.



3.3. Class diagram





View	
ⓜ View()	
ⓕ productFrame	JFrame
ⓕ productTable	JTable
ⓕ orderFrame	JFrame
ⓕ clientFrame	JFrame
ⓕ clientTable	JTable
ⓕ viewBillsButton	JButton
ⓕ orderTable	JTable
ⓜ initializeFrames()	void
ⓜ initializeTableWithReflection(List<?>, JTable)	void
ⓜ createButton(String, Color, Font)	JButton
ⓜ createOrderTablePanel(JTable, String)	JPanel
ⓜ initializeButtons()	void
ⓜ addPanelsToFrames()	void
ⓜ createTablePanel(JTable, String)	JPanel
ⓜ setupTablePanel(List<?>, String, JTable)	void
ⓜ initializePanels()	void
ⓜ initializeComboBoxes(List<Client>, List<Product>)	void
ⓟ clientTable	JTable
ⓟ orderFrame	JFrame
ⓟ orderTable	JTable
ⓟ productTable	JTable
ⓟ productFrame	JFrame
ⓟ viewBillsButton	JButton
ⓟ clientFrame	JFrame

ConnectionFactory	
ⓜ ConnectionFactory()	
ⓜ close(Statement)	void
ⓜ close(ResultSet)	void
ⓜ createConnection()	Connection
ⓜ close(Connection)	void
ⓟ connection	Connection

ProductBL	
ⓜ ProductBL(ProductDAO)	
ⓜ editProduct(Product)	Product
ⓜ deleteProduct(int)	boolean
ⓜ listAllProducts()	List<Product>
ⓜ addProduct(Product)	Product
ⓜ findById(int)	Product
ⓜ validateProduct(Product)	void

Controller	
ⓜ Controller(View, ClientBL, ProductBL, OrderBL)	
ⓜ deleteClient()	void
ⓜ addProduct()	void
ⓜ attachButtonListeners()	void
ⓜ viewBills()	void
ⓜ displayBills(List<Bill>)	void
ⓜ initController()	void
ⓜ deleteProduct()	void
ⓜ addOrder()	void
ⓜ refreshTableData()	void
ⓜ editClient()	void
ⓜ editProduct()	void
ⓜ addClient()	void

Order	
ⓜ Order(int, int, int, Date, int)	
ⓕ orderId	int
ⓕ clientId	int
ⓕ date	Date
ⓕ quantity	int
ⓕ productId	int
ⓜ toString()	String
ⓟ productId	int
ⓟ clientId	int
ⓟ date	Date
ⓟ quantity	int
ⓟ orderId	int

Product	
ⓜ Product(int, String, int, double)	
ⓕ productName	String
ⓕ quantity	int
ⓕ price	double
ⓜ toString()	String
ⓟ price	double
ⓟ id	int
ⓟ productName	String
ⓟ quantity	int

ConnectionFactory	
ⓜ ConnectionFactory()	
ⓜ close(Statement)	void
ⓜ close(ResultSet)	void
ⓜ createConnection()	Connection
ⓜ close(Connection)	void
ⓟ connection	Connection

OrderBL	
ⓜ OrderBL(OrderDAO, ProductDAO, ClientDAO, BillDAO)	
ⓜ listAllOrders()	List<Order>
ⓜ findById(int)	Order
ⓜ createOrder(int, Client, Product, int)	boolean
ⓟ allBills	List<Bill>

3.4. Data Structures

- List<Client>, List<Product>, List<Order>, List<Bill>: These are used to store collections of clients, products, orders, and bills, respectively. They provide ordered collections that allow for efficient iteration and manipulation of data.
- Map<String, String>: This is used to map entity fields to database columns, facilitating dynamic SQL query construction and data mapping.

4. Implementation

In this part I am going to give details about each class: the structure and the functionality, giving an attentive eye around the fields(attributes), important methods.

4.1. GenericDAO Class

1. Package: 'data'

2. **Fields:**

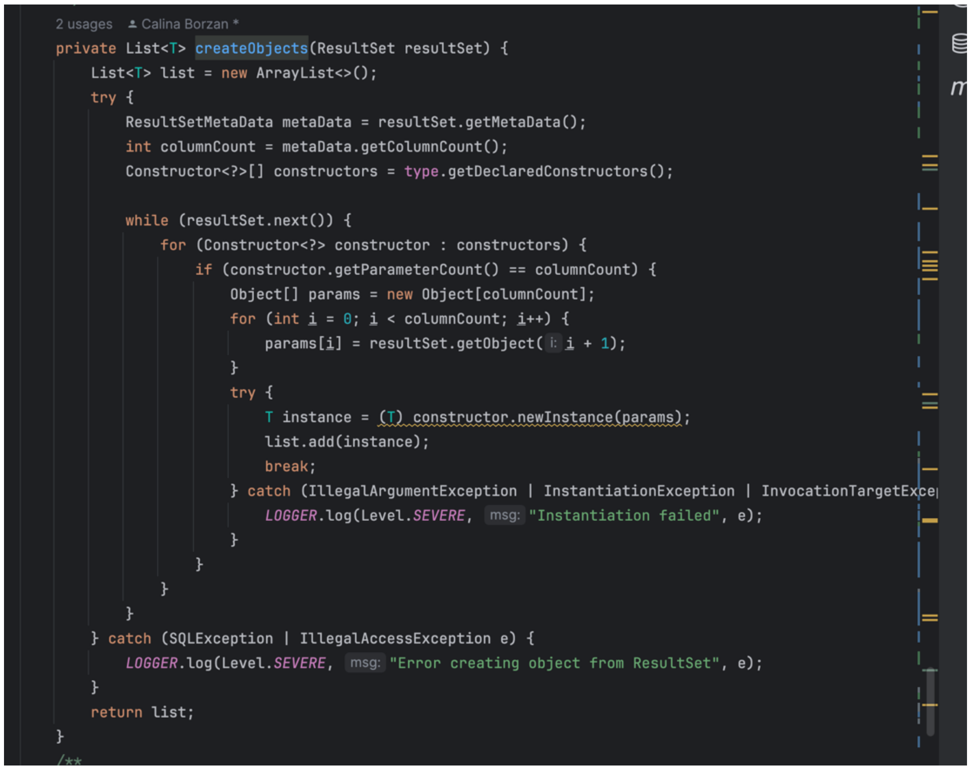
- private final Class<T> type: Represents the class of the entity that this DAO handles. It enables the class to perform operations on different entity types.
- private static final Logger LOGGER: A logger instance for logging various activities and errors that occur within the DAO class.

3. **Important Methods:**

- protected String getTableName(): Returns the table name based on the entity type by using the simple name of the class as the table name. Ensures that SQL queries reference the correct table corresponding to the entity class.
- private String selectFromQuery(String field): Constructs an SQL query string for selecting entities based on a specific field value. Facilitates the generation of dynamic SQL queries for retrieving entities by specific fields, typically used for finding entities by their primary key.
- protected String getPrimaryKeyColumnName(): Retrieves the primary key column name of the table associated with the entity. It can be overridden by subclasses if the primary

key column is not named 'id'. Standardizes the identification of the primary key column across different entities, essential for CRUD operations.

- public List<T> findAll(): Executes a SELECT query to retrieve all entities of type T from the database, returning a list of all entities found in the corresponding table. Provides a way to fetch all records from a specific table, useful for listing all entries of an entity type.
- public T insert(T t): Dynamically constructs an SQL INSERT statement to add a new record to the database using the entity's fields. Updates the entity's ID to reflect the generated key from the database. Adds a new record to the database, ensuring that the entity's ID is correctly set after insertion.
- public T findById(int id): Constructs an SQL SELECT query dynamically to find a single entity by its ID. Retrieves a specific record from the database using its primary key, enabling precise data access.
- public T update(T t): Constructs an SQL update query dynamically based on the entity's fields to update an existing entity in the database. Modifies an existing record with new data, ensuring that changes are persisted in the database.
- public boolean delete(int id): Executes an SQL DELETE statement to remove a record from the database by its primary key ID. Removes a record from the database, facilitating the deletion of entities.
- private List<T> createObjects(ResultSet resultSet): Utilizes reflection to instantiate objects based on the ResultSet metadata, creating a list of entities from a ResultSet. Converts database query results into entity objects, enabling the transformation of raw data into structured entities.
- protected abstract Map<String, String> getFieldColumnMapping(): Must be implemented by subclasses to provide a mapping from entity fields to database column names. Ensures correct field-to-column mapping in SQL operations, essential for dynamically generating SQL statements.



```
2 usages  Calina Borzan *
private List<T> createObjects(ResultSet resultSet) {
    List<T> list = new ArrayList<>();
    try {
        ResultSetMetaData metaData = resultSet.getMetaData();
        int columnCount = metaData.getColumnCount();
        Constructor<?>[] constructors = type.getDeclaredConstructors();

        while (resultSet.next()) {
            for (Constructor<?> constructor : constructors) {
                if (constructor.getParameterCount() == columnCount) {
                    Object[] params = new Object[columnCount];
                    for (int i = 0; i < columnCount; i++) {
                        params[i] = resultSet.getObject( i: i + 1);
                    }
                    try {
                        T instance = (T) constructor.newInstance(params);
                        list.add(instance);
                        break;
                    } catch (IllegalArgumentException | InstantiationException | InvocationTargetExce
                        LOGGER.log(Level.SEVERE,  msg: "Instantiation failed", e);
                    }
                }
            }
        }
    } catch (SQLException | IllegalAccessException e) {
        LOGGER.log(Level.SEVERE,  msg: "Error creating object from ResultSet", e);
    }
    return list;
}
/**
```

4.2. OrderBL Class

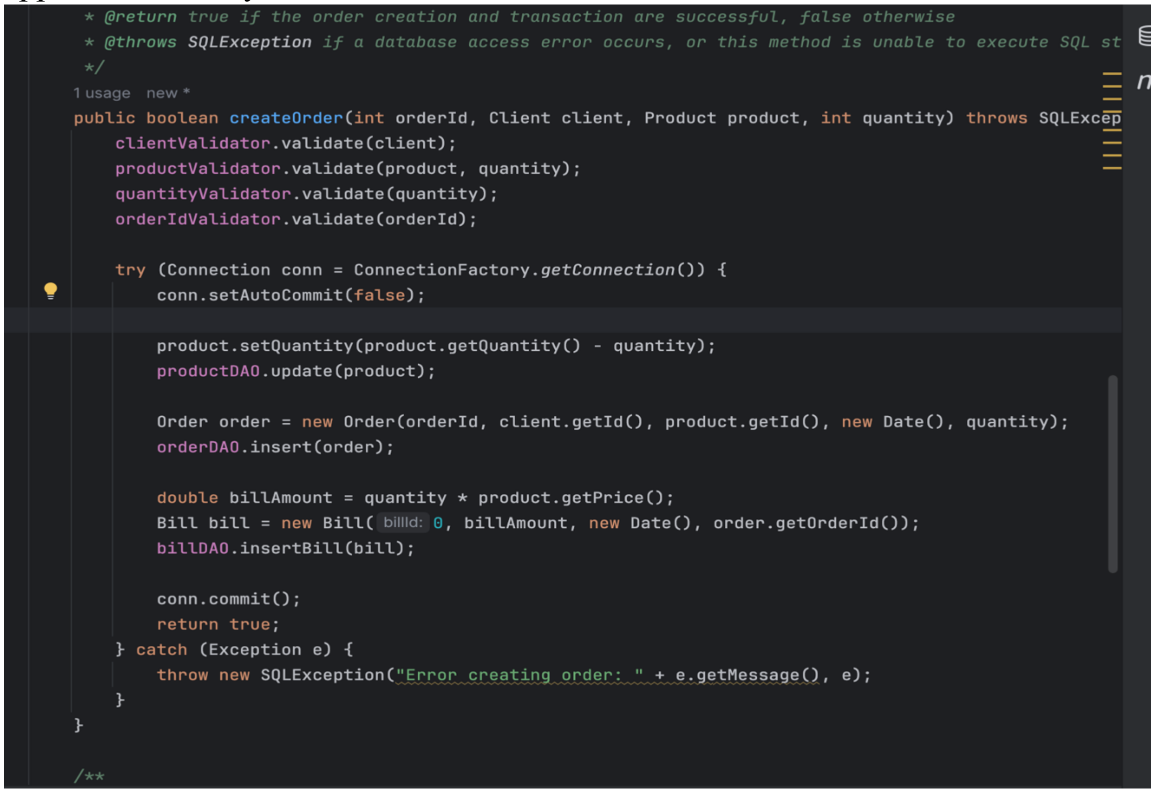
1. **Package:** ‘business’

2. **Fields:**
    - orderDAO: An instance of OrderDAO used to perform database operations related to orders.
    - productDAO: An instance of ProductDAO used to manage product data.
    - clientDAO: An instance of ClientDAO used to handle client data.
    - billDAO: An instance of BillDAO used to manage billing information.
    - clientValidator: An instance of ClientValidator used to validate client data.
    - productValidator: An instance of ProductValidator used to validate product data.
    - quantityValidator: An instance of QuantityValidator used to validate order quantities.

- orderIdValidator: An instance of OrderIdValidator used to validate order IDs.

3. Important Methods:

- createOrder(int orderId, Client client, Product product, int quantity) : This method first validates the client, product, quantity, and order ID to ensure they meet the required business rules. It then starts a database transaction to ensure atomicity of the order creation process. The product quantity is updated to reflect the new inventory level. An Order object is created and inserted into the database. A Bill object is created based on the order details and inserted into the database. The transaction is committed if all operations are successful, ensuring that all changes are applied atomically.



```
 * @return true if the order creation and transaction are successful, false otherwise
 * @throws SQLException if a database access error occurs, or this method is unable to execute SQL st
 */
1 usage  new *
public boolean createOrder(int orderId, Client client, Product product, int quantity) throws SQLExcep
    clientValidator.validate(client);
    productValidator.validate(product, quantity);
    quantityValidator.validate(quantity);
    orderIdValidator.validate(orderId);

    try (Connection conn = ConnectionFactory.getConnection()) {
        conn.setAutoCommit(false);

        product.setQuantity(product.getQuantity() - quantity);
        productDAO.update(product);

        Order order = new Order(orderId, client.getId(), product.getId(), new Date(), quantity);
        orderDAO.insert(order);

        double billAmount = quantity * product.getPrice();
        Bill bill = new Bill( billId: 0, billAmount, new Date(), order.getOrderId());
        billDAO.insertBill(bill);

        conn.commit();
        return true;
    } catch (Exception e) {
        throw new SQLException("Error creating order: " + e.getMessage(), e);
    }
}

/**
```

4.3. View Class

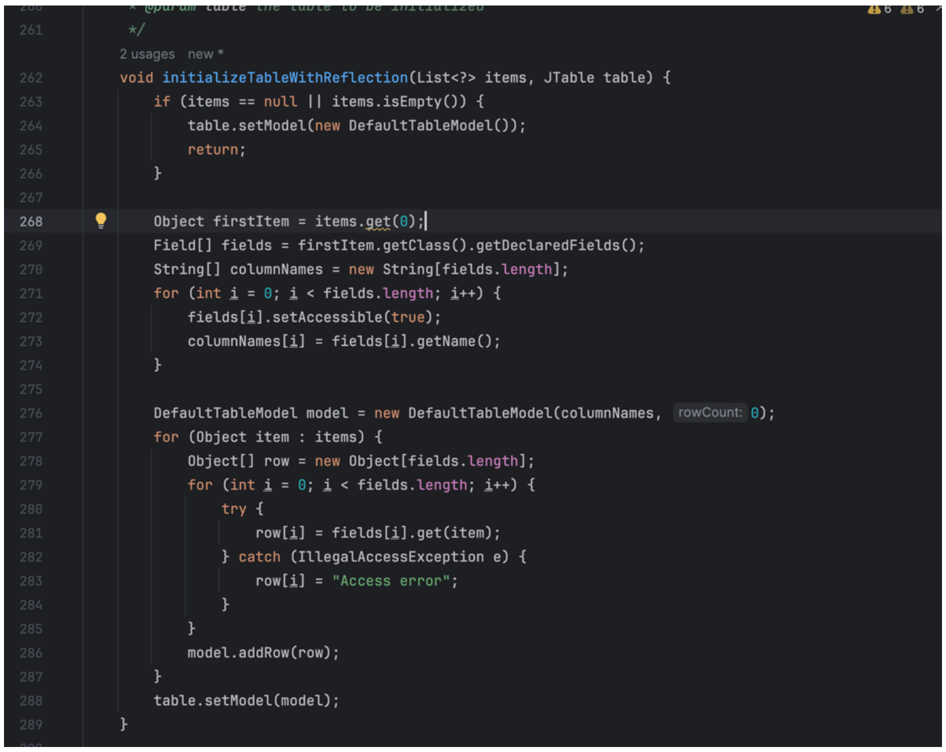
1. **Package:** ‘presentation’

2. **Fields:**

- JFrame clientFrame: The main frame for managing clients.
- JFrame productFrame: The main frame for managing products.
- JFrame orderFrame: The main frame for managing orders.
- JTable clientTable: The table for displaying client information.
- JTable productTable: The table for displaying product information.
- JTable orderTable: The table for displaying order information.
- JPanel clientPanel: The panel containing the client table and buttons.
- JPanel productPanel: The panel containing the product table and buttons.
- JPanel orderPanel: The panel containing the order table and buttons.
- JButton addClientButton: Button to add a new client.
- JButton editClientButton: Button to edit an existing client.
- JButton deleteClientButton: Button to delete a client.
- JButton addProductButton: Button to add a new product.
- JButton editProductButton: Button to edit an existing product.
- JButton deleteProductButton: Button to delete a product.
- JButton viewBillsButton: Button to view bills.
- JButton addOrderButton: Button to add a new order.

3. **Important Methods:**

   - void initializeTableWithReflection(List<?> items, Jtable table): method is designed to dynamically set up a table model for a JTable by using Java reflection to determine the structure of the table. This method allows for the table to be populated with data from a list of objects without hardcoding the column names and field values, making it highly flexible and reusable for different types of objects. The method uses reflection to analyze the first item in the list to determine the field names. These field names will be used as the column headers for the table. It retrieves the declared fields of the first item using getDeclaredFields method. This allows access to all fields, including private ones. Each field's name is then extracted and stored in a String array named columnNames. The fields are made accessible using setAccessible(true) to ensure that their values can be accessed later.



```
 * @param table the table to be initialized
 */
2 usages  new *
void initializeTableWithReflection(List<?> items, JTable table) {
    if (items == null || items.isEmpty()) {
        table.setModel(new DefaultTableModel());
        return;
    }

    Object firstItem = items.get(0);
    Field[] fields = firstItem.getClass().getDeclaredFields();
    String[] columnNames = new String[fields.length];
    for (int i = 0; i < fields.length; i++) {
        fields[i].setAccessible(true);
        columnNames[i] = fields[i].getName();
    }

    DefaultTableModel model = new DefaultTableModel(columnNames, rowCount: 0);
    for (Object item : items) {
        Object[] row = new Object[fields.length];
        for (int i = 0; i < fields.length; i++) {
            try {
                row[i] = fields[i].get(item);
            } catch (IllegalAccessException e) {
                row[i] = "Access error";
            }
        }
        model.addRow(row);
    }
    table.setModel(model);
}
```

5. Results

- The application successfully performed Create, Read, Update, and Delete operations on clients, products, orders, and bills.

- Verified that the database accurately reflects these operations, ensuring data integrity and consistency.

- Implemented dynamic table population using reflection, allowing for flexible and adaptable data displays without hardcoding column names.

- Ensured that tables in the GUI correctly display current data from the database, providing real-time updates and accurate information to users.

- Integrated comprehensive validation mechanisms for client, product, order, and quantity data.

- Prevented invalid data entries and ensured that business rules and constraints are enforced before database operations are performed.
- Implemented transaction management to ensure atomicity and consistency during order creation.

- Demonstrated that the system correctly handles rollbacks in case of errors, maintaining database integrity.
- Developed a user-friendly graphical user interface that allows users to easily manage clients, products, orders, and view bills.

- Provided clear feedback to users for actions such as successful operations or error messages, enhancing user experience.

6. Conclusions

Embarking on the journey to develop an order management application has been both an enriching and challenging experience, broadening my understanding of software development principles and data modeling. Here are the key takeaways, learnings, and potential avenues for future enhancements.

What I Have Learned

- Encapsulation: I learned the importance of bundling data and methods within classes to protect the internal state and provide controlled access through methods.
- Polymorphism: I experienced how to design flexible and reusable code using interfaces and abstract classes. This was particularly useful in creating generic data access objects (DAOs) that can handle various entity types.
- Inheritance and Composition: I understood the distinction and appropriate usage of inheritance for extending classes and composition for including instances of other classes as member variables to build complex objects.

- Separation of Concerns: Implementing the layered architecture helped me understand how to organize code into distinct layers (Model, Business Logic, Data Access, Presentation), each with specific responsibilities, enhancing modularity and maintainability.
- Dependency Management: I learned to manage dependencies between layers effectively, ensuring that changes in one layer have minimal impact on others.

- CRUD Operations: I gained hands-on experience in implementing create, read, update, and delete operations using JDBC, ensuring efficient and secure database interactions.
- Transaction Management: I learned the significance of transactions to ensure data integrity, particularly in scenarios involving multiple related operations, like order creation.

- Dynamic Table Population: Using reflection to dynamically determine and populate table columns based on object fields was a valuable lesson in building flexible and adaptive user interfaces.
- Runtime Class Manipulation: I learned how to inspect and manipulate classes, fields, and methods at runtime, enhancing my understanding of Java's reflective capabilities.

- Swing Framework: Designing a user-friendly interface using Swing taught me the intricacies of GUI development, including event handling, layout management, and component customization.
- User Feedback and Interaction: I understood the importance of providing clear feedback to users for actions performed, which improves user experience and satisfaction.

Future Developments

- Enhanced Validation: Implement more comprehensive validation rules and user feedback mechanisms to cover a wider range of possible user inputs and scenarios.
- Advanced Reporting: Develop advanced reporting features to provide detailed insights and analytics on orders, client behavior, and product sales. This could include graphical representations like charts and graphs.
- User Authentication and Authorization: Implement a user authentication and authorization system to ensure that only authorized users can perform certain operations, enhancing security.
- Integration with External Systems: Integrate the application with external systems such as inventory management, payment gateways, and CRM systems to provide a seamless experience and automate more business processes.

- Scalability Improvements: Optimize the application for better performance with larger datasets and concurrent users. This could involve database indexing, query optimization, and implementing caching mechanisms.
- Cloud Deployment: Explore deploying the application on cloud platforms to take advantage of scalability, availability, and managed services for databases and authentication.
- Mobile Application: Develop a mobile version of the application to provide users with the flexibility to manage clients, products, and orders on the go.
- Enhanced User Interface: Improve the user interface with modern design principles, possibly using frameworks like JavaFX or transitioning to web-based technologies such as React or Angular.

Overall, developing a order management application has been a rewarding experience, providing insights into complex system dynamics and the intricacies of simulation modeling.

7. Bibliography

The references I while implementing this program:

1. https://dzone.com/articles/layers-standard-enterprise

2. https://jenkov.com/tutorials/java-reflection/index.html

3. https://www.baeldung.com/java-pdf-creation