

**ASSIGNMENT\_3**

**ORDERS MANAGEMENT**

DOCUMENTATION

Student name: Ciufudean Teodora Gabriela

Group: 30423

Contents

[1. Assignment Objectives 3](#_Toc135391589)

[2. Problem Analysis, Modeling, Scenarios, Use Cases 3](#_Toc135391590)

[3. Design 10](#_Toc135391591)

[4. Implementation 12](#_Toc135391592)

[5.Conclusions 25](#_Toc135391593)

[6. Bibliography 26](#_Toc135391594)

# **Assignment Objectives**

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

# **Problem Analysis, Modeling, Scenarios, Use Cases**

Analysis

Managing the products, the clients and the orders for a warehouse using handwritten registries is difficult and time consuming, so a good solution would be an order management application.

Modeling

Functional requirements:

- The application should allow an employee to add a new client

- The application should allow an employee to add a new product

- The application should allow an employee to add a new order

- The application should allow an employee to delete an existing client

- The application should allow an employee to delete an existing product

- The application should allow an employee to edit an existing client

- The application should allow an employee to edit an existing product

- The application should allow an employee to see all existing clients

- The application should allow an employee to see all existing products

- The application should allow an employee to see all existing orders

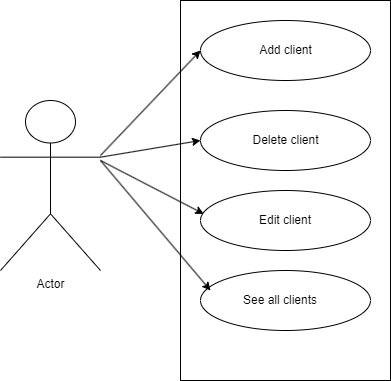
- The application should allow an employee to see all existing bills

Non-Functional requirements:

- The application should be intuitive and easy to use by the use

Scenarios and use cases

Use case diagrams:



**Use Case**: add client

Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for client GUI

2. The application will display a form in which the client details should be inserted

3. The employee inserts the id of the client, its name, age and address

4. The employee clicks on the “Add” button

5. The application stores the client data in the database and displays an acknowledge message

Alternative Sequence: Invalid values for the client’s data - The user inserts a value smaller than 10 or greater than 80 for the age of the client - The application displays an error message and requests the user to insert a valid stock - The scenario returns to step 3

**Use Case**: delete client

Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for client GUI

2. The application will display a form in which the client details should be inserted

3. The employee inserts the id of the client

4. The employee clicks on the “Delete” button

5. The application deletes client data from the database and displays an acknowledge message

Alternative Sequence: Invalid values for the client’s data – The user inserts a value for id which does not exist in the database - The scenario returns to step 3

**Use Case**: edit client

Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for client GUI

2. The application will display a form in which the client details should be inserted

3. The employee inserts the id of the client, its name, age and address

4. The employee clicks on the “Edit” button

5. The application updates the client data in the database and displays an acknowledge message

Alternative Sequence: Invalid values for the client’s data - The user inserts a value smaller than 10 or greater than 80 for the age of the client - The application displays an error message and requests the user to insert a valid stock - The scenario returns to step 3

**Use Case**: see all clients

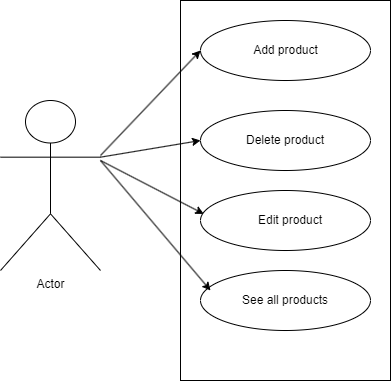
Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for client GUI

2. The employee clicks on the “View all” button

5. The application shows a table with all clients from the database

Alternative Sequence: -



**Use Case**: add product

Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for product GUI

2. The application will display a form in which the product details should be inserted

3. The employee inserts the id of the product, its name and current stock

4. The employee clicks on the “Add” button

5. The application stores the client data in the database and displays an acknowledge message

Alternative Sequence: Invalid values for the product’s data - The user inserts a value smaller than 0 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**: delete product

Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for product GUI

2. The application will display a form in which the product details should be inserted

3. The employee inserts the id of the product

4. The employee clicks on the “Delete” button

5. The application deletes client data from the database and displays an acknowledge message

Alternative Sequence: Invalid values for the product’s data – The user inserts a value for id which does not exist in the database - The scenario returns to step 3

**Use Case**: edit client

Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for client GUI

2. The application will display a form in which the client details should be inserted

3. The employee inserts the id of the product, its name and current stock

4. The employee clicks on the “Edit” button

5. The application updates the product data in the database and displays an acknowledge message

Alternative Sequence: Invalid values for the product’s data - The user inserts a value smaller than 0 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**: see all products

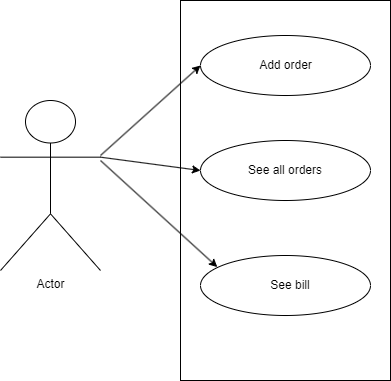
Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for product GUI

2. The employee clicks on the “View all” button

5. The application shows a table with all products from the database

Alternative Sequence: -



**Use Case**: add order

Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for order GUI

2. The application will display a form in which the order details should be inserted

3. The employee inserts the id of the order, client id, product id and the quantity

4. The employee clicks on the “Add” button

5. The application stores the order data in the database and displays an acknowledge message

Alternative Sequence: Invalid values for the product’s data - The user inserts a value smaller than 0 for the quantity of the order - The application displays an error message and requests the user to insert a valid stock - The scenario returns to step 3

**Use Case**: see all orders

Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for order GUI

2. The employee clicks on the “View all” button

5. The application shows a table with all orders from the database

Alternative Sequence: -

**Use Case**: see bill

Primary Actor: employee

Main Success Scenario: 1. The employee selects the option for order GUI

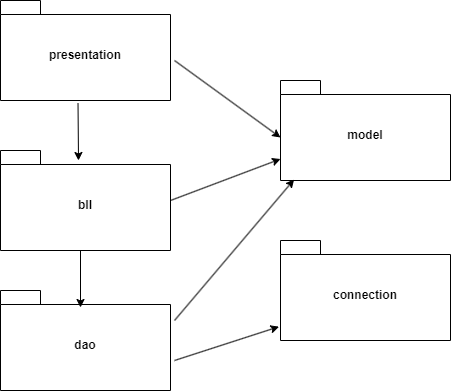
2. The employee clicks on the “Bill” button

5. The application shows a table with all bills from the database

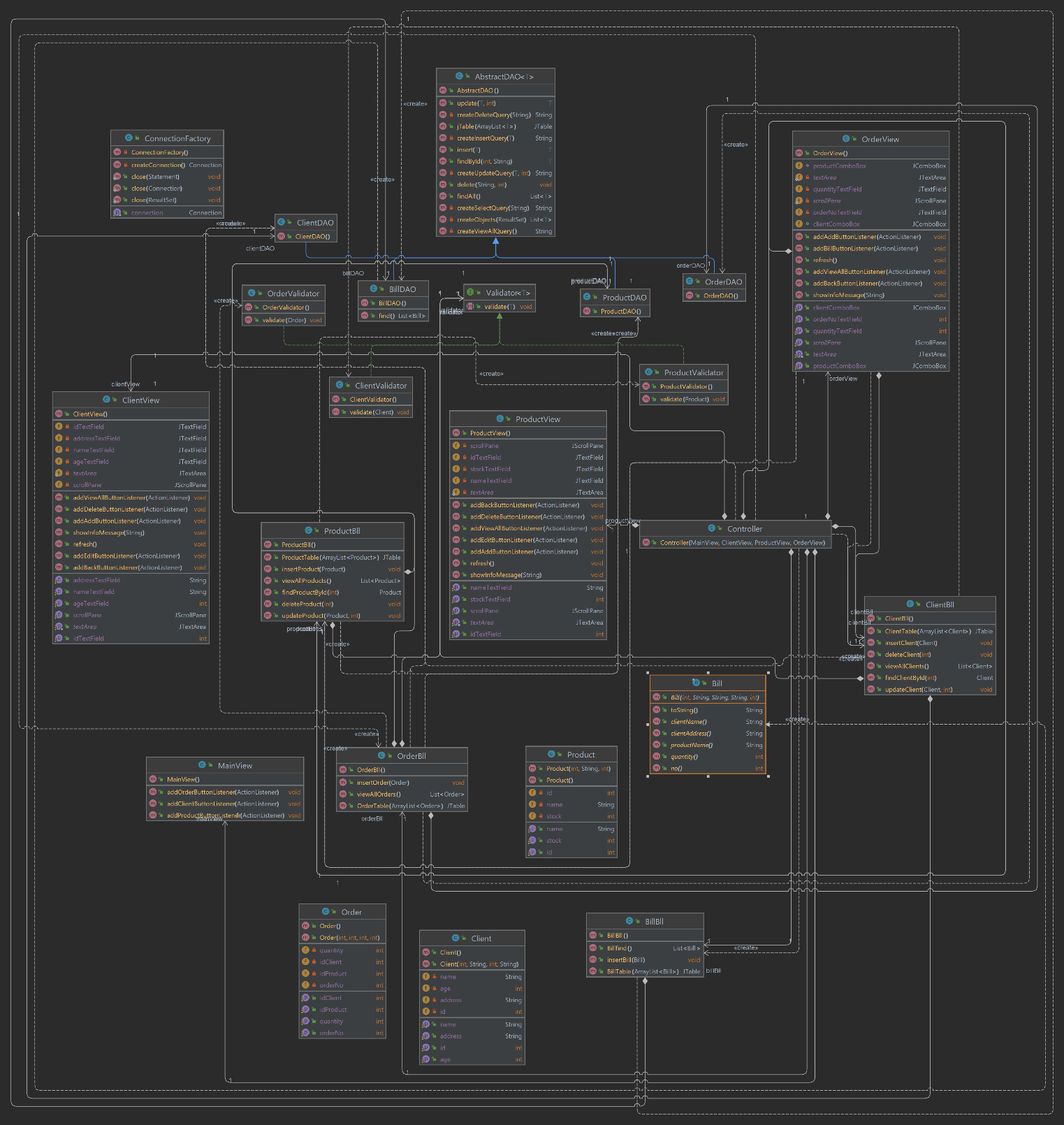
Alternative Sequence: -

# **Design**

Package diagram:



Class diagram:



# **Implementation**

**Client** class - class represents a client and has the following components:

Fields:

* id (int): Represents the ID of the client.
* name (String): Represents the name of the client.
* age (int): Represents the age of the client.
* address (String): Represents the address of the client.

Constructors:

* Client(): Default constructor with no parameters.
* Client(int id, String name, int age, String address): Constructor that takes in the ID, name, age, and address of the client to initialize the object.

Methods:

* getId(): Returns the ID of the client.
* setId(int id): Sets the ID of the client.
* getName(): Returns the name of the client.
* setName(String name): Sets the name of the client.
* getAge(): Returns the age of the client.
* setAge(int age): Sets the age of the client.
* getAddress(): Returns the address of the client.
* setAddress(String address): Sets the address of the client.

These methods are used to access and modify the attributes of the Client objects. The getId, getName, getAge, and getAddress methods are getters, while setId, setName, setAge, and setAddress methods are setters for the respective fields**.**

**Product** class - represents a product and has the following components:

Fields:

* id (int): Represents the ID of the product.
* name (String): Represents the name of the product.
* stock (int): Represents the stock quantity of the product.

Constructors:

* Product(): Default constructor with no parameters.
* Product(int id, String name, int stock): Constructor that takes in the ID, name, and stock quantity of the product to initialize the object.

Methods:

* getId(): Returns the ID of the product.
* setId(int id): Sets the ID of the product.
* getName(): Returns the name of the product.
* setName(String name): Sets the name of the product.
* getStock(): Returns the stock quantity of the product.
* setStock(int stock): Sets the stock quantity of the product.

These methods are used to access and modify the attributes of the Product objects. The getId, getName, and getStock methods are getters, while setId, setName, and setStock methods are setters for the respective fields.

**Order** class - represents an order and has the following components:

Fields:

* orderNo (int): Represents the order number.
* idClient (int): Represents the ID of the client who placed the order.
* idProduct (int): Represents the ID of the product being ordered.
* quantity (int): Represents the quantity of the product ordered.

Constructors:

* Order(): Default constructor with no parameters.
* Order(int orderNo, int idClient, int idProduct, int quantity): Constructor that takes in the order number, client ID, product ID, and quantity to initialize the object.

Methods:

* getOrderNo(): Returns the order number.
* setOrderNo(int orderNo): Sets the order number.
* getIdClient(): Returns the ID of the client who placed the order.
* setIdClient(int idClient): Sets the ID of the client.
* getIdProduct(): Returns the ID of the product being ordered.
* setIdProduct(int idProduct): Sets the ID of the product.
* getQuantity(): Returns the quantity of the product ordered.
* setQuantity(int quantity): Sets the quantity of the product ordered.

These methods are used to access and modify the attributes of the Order objects. The getOrderNo, getIdClient, getIdProduct, and getQuantity methods are getters, while setOrderNo, setIdClient, setIdProduct, and setQuantity methods are setters for the respective fields.

**Bill** class - represents a bill and is defined as a record in Java. A record is an immutable class introduced in Java 14 that provides a concise way to declare classes for immutable data.

Fields:

* no (int): Represents the bill number.
* clientName (String): Represents the name of the client.
* clientAddress (String): Represents the address of the client.
* productName (String): Represents the name of the product.
* quantity (int): Represents the quantity of the product.

Constructor:

* Bill(int no, String clientName, String clientAddress, String productName, int quantity): Constructor that takes in the bill number, client name, client address, product name, and quantity to initialize the object. This constructor is automatically generated for record classes in Java.

Methods:

* toString(): Overrides the toString() method to provide a custom string representation of the Bill object. It returns a formatted string containing the bill number, client name, client address, product name, and quantity.

This class is designed to be immutable, meaning that once the object is created, its state cannot be changed. The fields are final and can only be set through the constructor. The toString() method provides a formatted string representation of the Bill object.

**AbstractDao** class - provides generic data access operations for entities in a database. It is a generic class that takes a type parameter **T** representing the entity type.

Fields:

* LOGGER: A Logger object used for logging messages.

Constructor:

* AbstractDAO(): Constructs an AbstractDAO object and retrieves the type of the entity using reflection.

Methods:

* createSelectQuery(String field): Creates a SELECT query to retrieve a specific record based on the provided field value.
* createViewAllQuery(): Creates a query to retrieve all records of the specified type.
* createDeleteQuery(String field): Creates a DELETE query to delete a specific record based on the provided field value.
* createUpdateQuery(T t, int id): Creates an UPDATE query to update the specified record with the given values.
* createInsertQuery(T t): Creates an INSERT query to insert a new record with the given values.
* findAll(): Retrieves all entities from the database and returns a list of entities.
* findById(int id, String field): Retrieves an entity by its ID from the database and returns the found entity or null if not found.
* createObjects(ResultSet resultSet): Creates a list of objects of type T from the provided ResultSet.
* insert(T t): Inserts an entity into the database and returns the inserted entity.
* update(T t, int id): Updates an entity in the database and returns the updated entity.
* delete(String field, int id): Deletes an entity from the database based on the provided field and ID.
* jTable(ArrayList<T> data): Creates a JTable representation of the provided data.

The AbstractDAO class utilizes reflection to dynamically create SQL queries and maps the database records to Java objects using the ResultSet and Constructor classes. It provides generic CRUD (Create, Read, Update, Delete) operations for entities and allows for retrieving and manipulating data from the database.

**ClientDAO** class - responsible for handling operations related to the **Client** model using a Data Access Object (DAO) design pattern. The DAO pattern separates the data access logic from the business logic.

**ProductDAO** class - responsible for handling operations related to the **Product** model using a Data Access Object (DAO) design pattern. The DAO pattern separates the data access logic from the business logic.

**OrderDAO** class - responsible for handling operations related to the **Order** model using a Data Access Object (DAO) design pattern. The DAO pattern separates the data access logic from the business logic.

**BillDAO** class - responsible for handling operations related to the **Bill** model using a Data Access Object (DAO) design pattern. The DAO pattern separates the data access logic from the business logic.

**ClientValidator** class - responsible for validating the age of a **Client** object. It implements the **Validator** interface with a type parameter of **Client**

Fields:

MIN\_AGE: A constant representing the minimum age allowed for a client. The value is set to 10.

MAX\_AGE: A constant representing the maximum age allowed for a client. The value is set to 90.

Method:

validate(Client client): This method validates the age of a Client object. It takes a Client object as a parameter and checks if the client's age is within the allowed range (between MIN\_AGE and MAX\_AGE). If the age is outside the allowed range, an IllegalArgumentException is thrown with an appropriate error message.

It's important to note that the validate method does not return any value but throws an exception when the validation fails.

**ProductValidator** class - responsible for validating a **Product** object. It implements the **Validator** interface with a type parameter of **Product**

Method:

* + validate(Product product): This method validates the stock of a Product object. It takes a Product object as a parameter and checks if the product's stock is less than or equal to 0. If the stock is less than or equal to 0, an IllegalArgumentException is thrown with an appropriate error message.

It's important to note that the validate method does not return any value but throws an exception when the validation fails.

**OrderValidator** class - responsible for validating an **Order** object. It implements the **Validator** interface with a type parameter of **Order**

Method:

* + validate(Order order): This method validates the quantity of an Order object. It takes an Order object as a parameter and checks if the order's quantity is less than 1. If the quantity is less than 1, an IllegalArgumentException is thrown with an appropriate error message.

It's important to note that the validate method does not return any value but throws an exception when the validation fails.

**Validator** interface - defines the contract for validation classes and provides a generic method for validating objects of type **T**. The **<T>** is a type parameter that represents the type of object to be validated.

Method:

* + validate(T t): This method is responsible for validating an object of type T. It takes an object of type T as a parameter and performs the necessary validation logic. The specific implementation of the validation logic will be defined in classes that implement this interface.

It's important to note that the validate method does not return any value but may throw exceptions when the validation fails.

**ClientBll** class - the business logic layer for managing **Client** objects. This class interacts with the **ClientDAO** and provides methods to perform various operations such as finding, inserting, deleting, updating, and retrieving clients.

1. Fields:
   * validator: An instance of the Validator<Client> interface, specifically the ClientValidator class implementation. It is responsible for validating Client objects before performing database operations.
   * clientDAO: An instance of the ClientDAO class, which handles database operations related to the Client model.
2. Constructor:
   * ClientBll(): Initializes the validator and clientDAO objects. It creates a new instance of ClientValidator and ClientDAO.
3. Methods:
   * findClientById(int id): Finds a Client object by its ID using the ClientDAO. It calls the findById method of ClientDAO and returns the found Client object. If the client is not found, it throws a NoSuchElementException.
   * insertClient(Client client): Inserts a Client object into the data store using the ClientDAO. It first validates the client object using the validator and then calls the insert method of ClientDAO.
   * deleteClient(int id): Deletes a Client object from the data store using the ClientDAO. It calls the delete method of ClientDAO and specifies the ID of the client to be deleted.
   * updateClient(Client client, int id): Updates a Client object in the data store using the ClientDAO. It calls the update method of ClientDAO and provides the updated client object and the ID of the client to be updated.
   * viewAllClients(): Retrieves a list of all Client objects from the data store using the ClientDAO. It calls the findAll method of ClientDAO and returns the list of Client objects.
   * ClientTable(ArrayList<Client> clients): Generates a JTable representation of the given list of Client objects using the ClientDAO. It calls the jTable method of ClientDAO and provides the list of Client objects to be represented as a JTable.

**ProductBll** class - the business logic layer for managing **Product** objects. This class interacts with the **ProductDAO** and provides methods to perform various operations such as finding, inserting, deleting, updating, and retrieving products.

1. Fields:
   * validator: An instance of the Validator<Product> interface, specifically the ProductValidator class implementation. It is responsible for validating Product objects before performing database operations.
   * productDAO: An instance of the ProductDAO class, which handles database operations related to the Product model.
2. Constructor:
   * ProductBll(): Initializes the validator and productDAO objects. It creates a new instance of ProductValidator and ProductDAO.
3. Methods:
   * findProductById(int id): Finds a Product object by its ID using the ProductDAO. It calls the findById method of ProductDAO and returns the found Product object. If the product is not found, it throws a NoSuchElementException.
   * insertProduct(Product product): Inserts a Product object into the data store using the ProductDAO. It first validates the product object using the validator and then calls the insert method of ProductDAO.
   * deleteProduct(int id): Deletes a Product object from the data store using the ProductDAO. It calls the delete method of ProductDAO and specifies the ID of the product to be deleted.
   * updateProduct(Product product, int id): Updates a Product object in the data store using the ProductDAO. It calls the update method of ProductDAO and provides the updated product object and the ID of the product to be updated.
   * viewAllProducts(): Retrieves a list of all Product objects from the data store using the ProductDAO. It calls the findAll method of ProductDAO and returns the list of Product objects.
   * ProductTable(ArrayList<Product> products): Generates a JTable representation of the given list of Product objects using the ProductDAO. It calls the jTable method of ProductDAO and provides the list of Product objects to be represented as a JTable.

**OrderBll** class - the business logic layer for managing **Order** objects. This class interacts with the **OrderDAO** and **ProductDAO** and provides methods to perform operations such as inserting orders, retrieving orders, and generating a **JTable** representation of orders.

1. Fields:
   * validator: An instance of the Validator<Order> interface, specifically the OrderValidator class implementation. It is responsible for validating Order objects before inserting them into the data store.
   * orderDAO: An instance of the OrderDAO class, which handles database operations related to the Order model.
   * productDAO: An instance of the ProductDAO class, which handles database operations related to the Product model.
2. Constructor:
   * OrderBll(): Initializes the validator, orderDAO, and productDAO objects. It creates new instances of OrderValidator, OrderDAO, and ProductDAO.
3. Methods:
   * insertOrder(Order order): Inserts an Order object into the data store using the OrderDAO. It first validates the order object using the validator. It retrieves the associated Client and Product objects using their respective DAOs. If the client or product is not found, it throws an IllegalArgumentException. It then updates the stock of the associated product by subtracting the quantity from the current stock. Finally, it calls the insert method of OrderDAO to insert the order and the update method of ProductDAO to update the product's stock.
   * viewAllOrders(): Retrieves a list of all Order objects from the data store using the OrderDAO. It calls the findAll method of OrderDAO and returns the list of Order objects.
   * OrderTable(ArrayList<Order> orders): Generates a JTable representation of the given list of Order objects using the OrderDAO. It calls the jTable method of OrderDAO and provides the list of Order objects to be represented as a JTable

**BillBll** class - the business logic layer for managing **Bill** objects. This class interacts with the **BillDAO** and provides methods to perform operations such as inserting bills, retrieving bills, and generating a **JTable** representation of bills.

1. Fields:
   * billDAO: An instance of the BillDAO class, which handles database operations related to the Bill model.
2. Constructor:
   * BillBll(): Initializes the billDAO object. It creates a new instance of the BillDAO class.
3. Methods:
   * insertBill(Bill bill): Inserts a Bill object into the data store using the BillDAO. It calls the insert method of the BillDAO and provides the Bill object to be inserted.
   * Billfind(): Retrieves a list of all Bill objects from the data store using the BillDAO. It calls the find method of the BillDAO and returns the list of Bill objects.
   * BillTable(ArrayList<Bill> bills): Generates a JTable representation of the given list of Bill objects using the BillDAO. It calls the jTable method of the BillDAO and provides the list of Bill objects to be represented as a JTable.

**ConnectionFactory** class - provides methods for establishing and managing database connections. It uses JDBC (Java Database Connectivity) to connect to a MySQL database.

**App** class - serves as the entry point of the application and contains the main method that is executed when the application starts. The purpose of the **App** class is to provide a starting point for the application and instantiate the necessary views and controllers to manage the user interface and application logic.

**Controller** class - manages the interaction between the views and the business logic of the application.

1. Fields:
   * mainView, clientView, productView, orderView: Instances of the views used in the application.
   * clientBll, productBll, orderBll, billBll: Instances of the business logic classes (Bll) responsible for interacting with the corresponding model classes (Client, Product, Order, Bill).
2. Constructor:
   * The constructor takes instances of the main view, client view, product view, and order view as arguments and initializes the corresponding fields.
   * Event listeners are registered for various buttons in the views, linking them to the appropriate action listeners defined in the Controller class.
3. Event listener classes:
   * The Controller class contains several nested classes that implement action listeners for different buttons in the views.
   * These listener classes define the actions to be performed when the corresponding buttons are clicked.
   * The listener classes handle tasks such as switching between views, adding, deleting, and updating entities, retrieving data from the database, and generating bills.

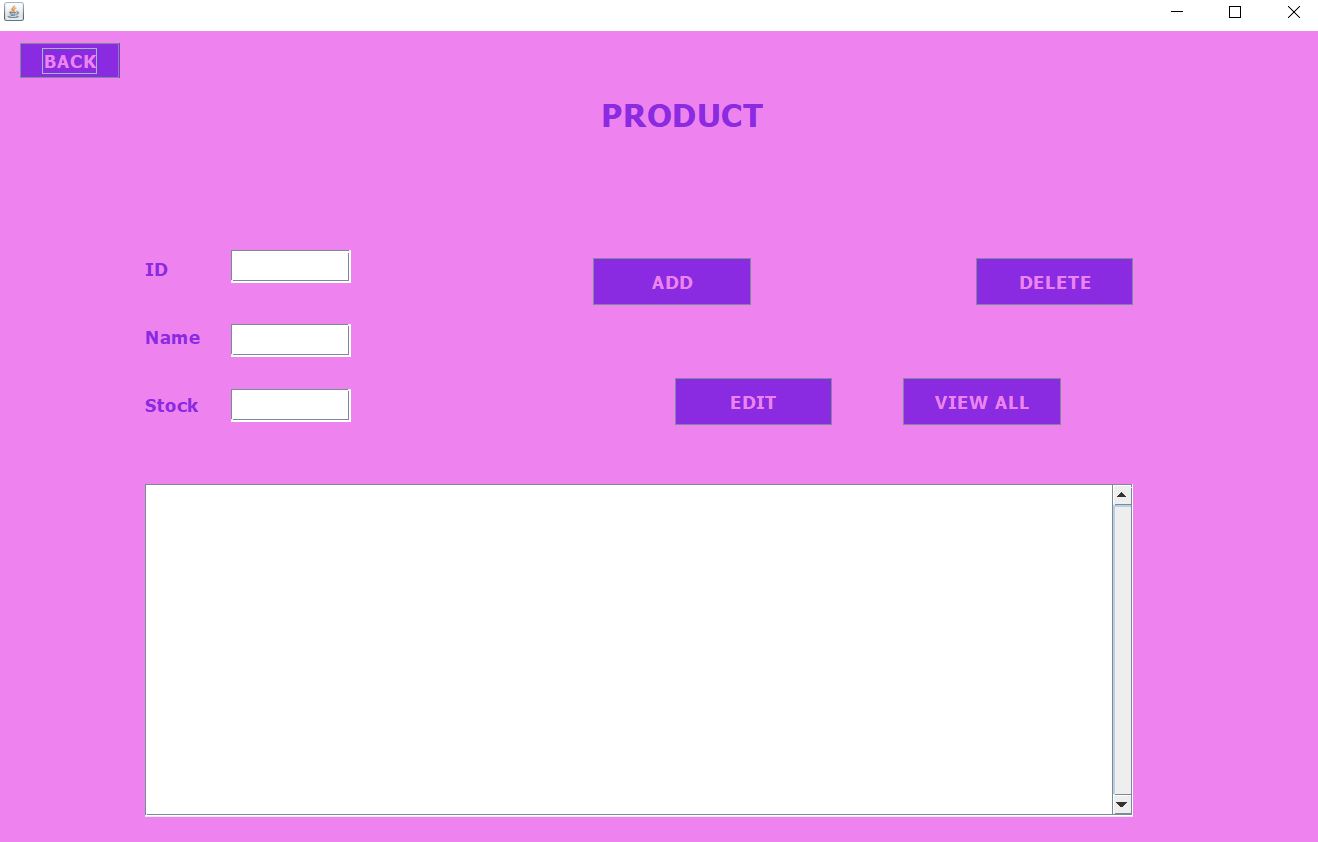
**MainView** class - epresents the main view of the Orders Management application. It extends the **JFrame** class and provides a graphical user interface (GUI) for the main menu of the application.



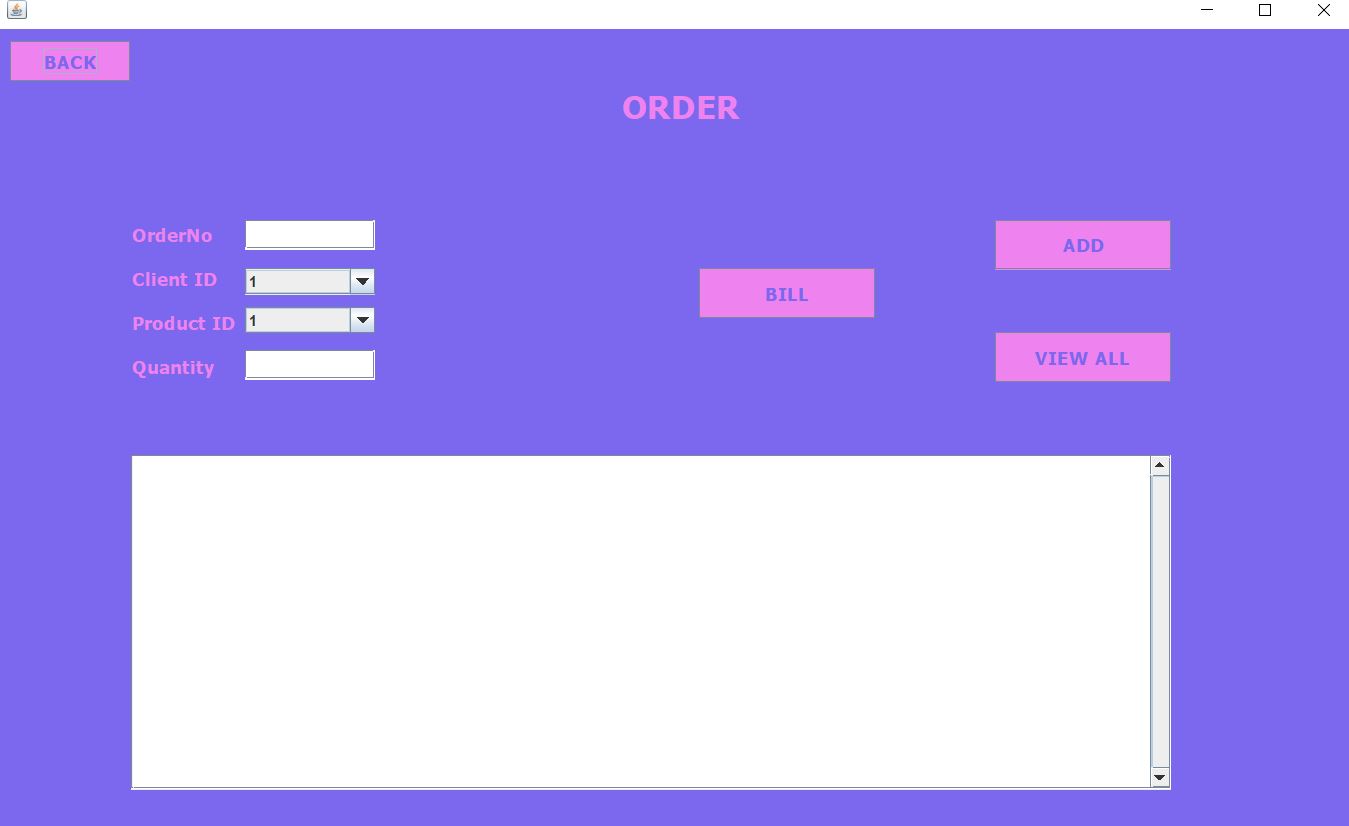
**ClientView** class - represents the view for managing clients in the Orders Management application. It extends the **JFrame** class and provides a graphical user interface (GUI) for adding, deleting, editing, and viewing client information.



**ProductView** class - represents the view for managing products in the Orders Management application. It extends the **JFrame** class and provides a graphical user interface (GUI) for adding, deleting, editing, and viewing client information.



**OrderView** class - represents the view for managing corders in the Orders Management application. It extends the **JFrame** class and provides a graphical user interface (GUI) for adding, deleting, editing, and viewing client information.



# **5.Conclusions**

This project served as an excellent exercise in both recalling and expanding upon the object-oriented programming (OOP) concepts learned during the first semester. It proved to be a valuable and challenging experience, particularly in the initial stages. Throughout the process, I gained several key insights and lessons that I would like to share:

1. Time management: I cannot emphasize enough the importance of effective time management. Having a well-organized schedule enables a gradual approach to problem-solving, which greatly contributes to overall productivity and efficiency.

2. Proper problem modeling: One of the crucial steps in software development is accurately modeling the problem domain from the outset. Investing time and effort into designing a solid foundation for the solution streamlines the subsequent implementation phase and leads to faster development.

3. Independent problem-solving and research: When encountering challenges or issues within the code, attempting to troubleshoot and resolve them independently is highly beneficial. By embracing the opportunity to investigate and explore potential solutions through research, you can expand your knowledge base and enhance your understanding of both new and familiar concepts.

Overall, by managing time effectively, focusing on robust problem modeling, and embracing independent problem-solving, I discovered the rewards of continuous learning and a deeper comprehension of programming principles. These lessons not only proved valuable during this project but are also applicable to future endeavors in software development.

# **6. Bibliography**

<https://www.w3schools.com/sql/sql_select.asp>

<https://docs.oracle.com/javase/tutorial/uiswing/components/table.html>

<https://www.jetbrains.com/help/idea/working-with-code-documentation.html>

<https://www.w3schools.com/java/java_files_create.asp>

Programming Techniques – Lectures of prof. Cristina Pop