DOCUMENTATION

ASSIGNMENT *3*

STUDENT NAME: Duică Sebastian
GROUP: 30424

CONTENTS

1. Assignment Objective

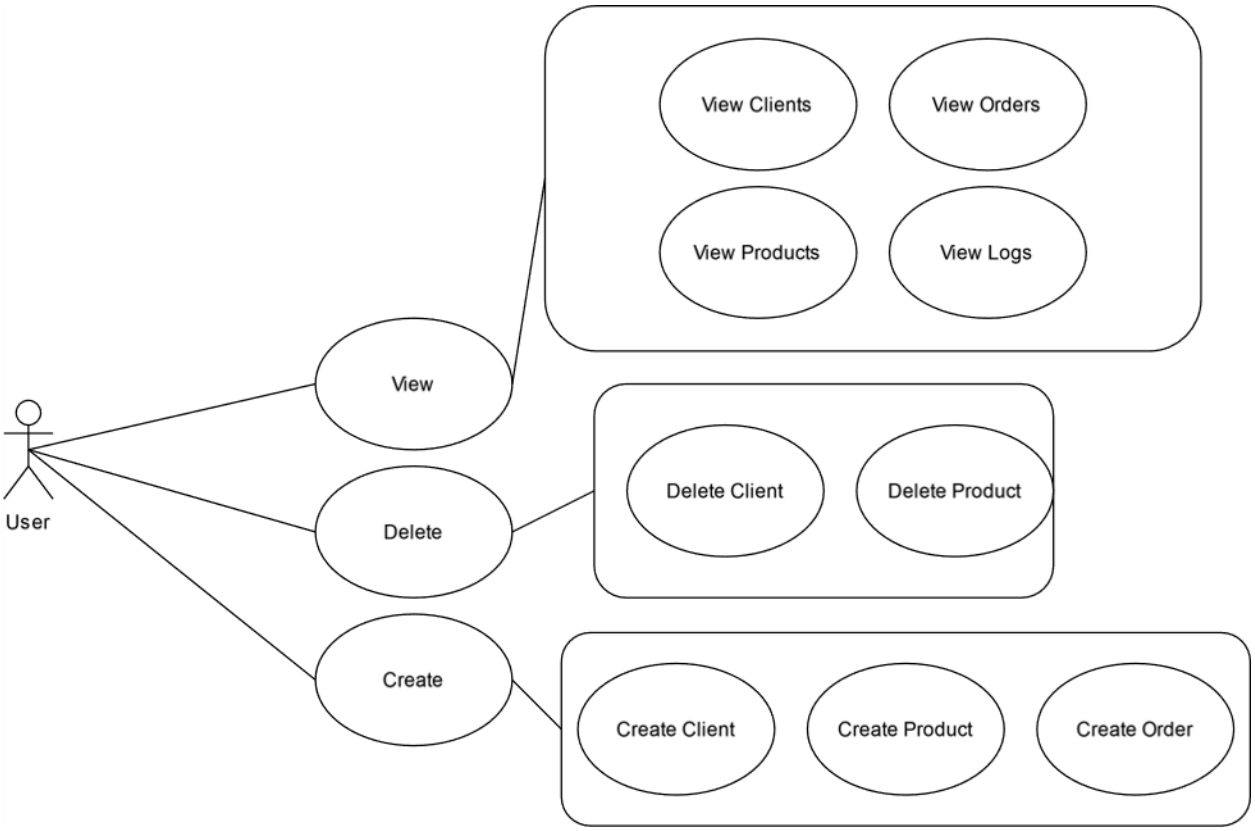
To develop a CRUD application for managing clients, products, orders, and logs in a Java environment.

Sub-objectives:

Sub-objective	Description	Section
Implement Client class	Define a class to represent clients with attributes like ID, name, email, and address.	Model
Implement Product class	Define a class to represent products with attributes like ID, name, price, and stock.	Model
Implement Orders class	Define a class to represent orders with attributes like ID, client ID, product ID, and quantity.	Model
Implement Log class	Define a class to represent logs with attributes like ID and bill.	Model
Implement DAO classes for CRUD operations	Create Data Access Object (DAO) classes for each model class to perform CRUD operations in the database.	DAO
Implement Logic classes for business logic	Develop Logic classes to handle business logic and operations for each model class.	Logic
Create database schema	Design and implement the database schema to store clients, products, orders, and logs.	Database Setup
Connect application to the database	Establish connection between the Java application and the database.	Database Setup
Implement create, read, update, and delete operations	Implement methods in Logic classes to perform CRUD operations for clients, products, orders, and logs.	Logic
Develop user interface	Create a user interface (UI) for interacting with the application.	UI

Each sub-objective represents a step towards achieving the main objective of developing the CRUD application. These steps are divided into sections such as Model, DAO, Logic, Database Setup, and UI, where each section focuses on a specific aspect of the application development process.

2. Problem Analysis, Modeling, Scenarios, Use Cases



Functional Requirements:

1. **Manage Clients:**
    - Add a new client.
    - View existing clients.
    - Edit client information.
    - Delete a client.
2. **Manage Products:**
    - Add a new product.
    - View existing products.
    - Edit product information.
    - Delete a product.
3. **Manage Orders:**
    - Create a new order.
    - View existing orders.
    - View order details.
    - Edit order details.
    - Delete an order.
4. **Manage Logs:**
    - View system logs.

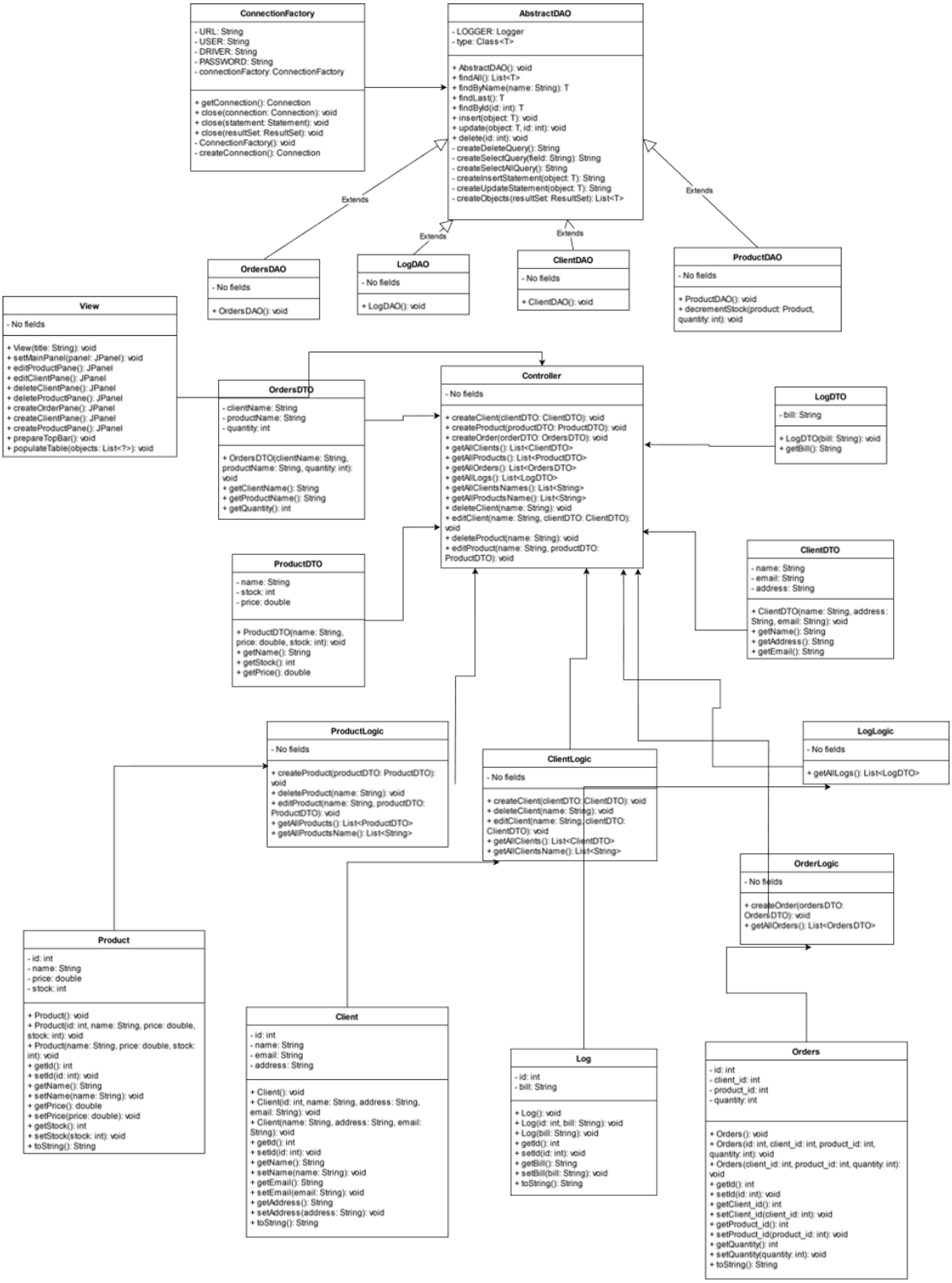
Use Case Descriptions:

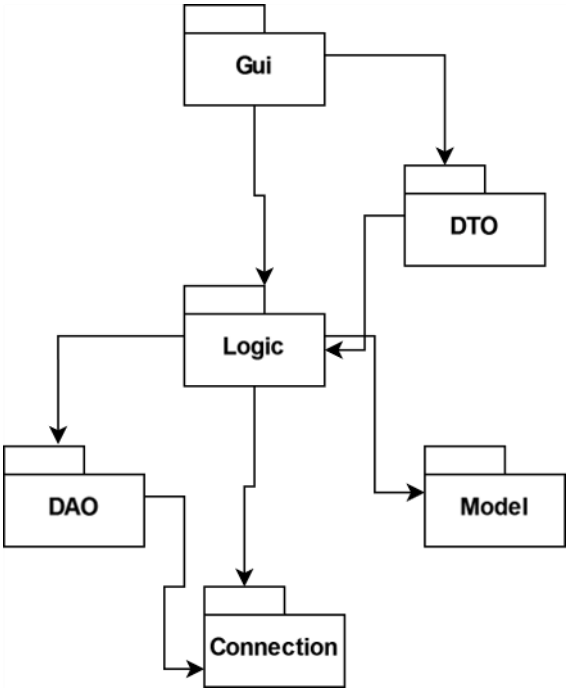
1. **Manage Clients:**
    - **Add a New Client:**
        - User inputs client details (name, email, address).
        - System validates inputs.
        - System adds the new client to the database.
    - **View Existing Clients:**
        - System retrieves all clients from the database.
        - System displays a list of clients to the user.
    - **Edit Client Information:**
        - User selects a client to edit.
        - User modifies client details.
        - System updates the client information in the database.
    - **Delete a Client:**
        - User selects a client to delete.
        - System confirms deletion.
        - System removes the client from the database.
2. **Manage Products:**
    - **Add a New Product:**
        - User inputs product details (name, price, stock).
        - System validates inputs.
        - System adds the new product to the database.
    - **View Existing Products:**
        - System retrieves all products from the database.
        - System displays a list of products to the user.
    - **Edit Product Information:**
        - User selects a product to edit.
        - User modifies product details.
        - System updates the product information in the database.
    - **Delete a Product:**
        - User selects a product to delete.
        - System confirms deletion.
        - System removes the product from the database.
3. **Manage Orders:**
    - **Create a New Order:**
        - User selects a client.
        - User selects a product and specifies quantity.
        - System validates the order.
        - System creates the order and updates product stock.
    - **View Existing Orders:**
        - System retrieves all orders from the database.
        - System displays a list of orders to the user.

- **View Order Details:**
  - User selects an order to view.
  - System retrieves order details.
  - System displays order details to the user.
- **Edit Order Details:**
  - User selects an order to edit.
  - User modifies order details.
  - System updates the order information in the database.
- **Delete an Order:**
  - User selects an order to delete.
  - System confirms deletion.
  - System removes the order from the database.

4. **Manage Logs:**
   - **View System Logs:**
     - System retrieves all logs from the database.
     - System displays system logs to the user.

3. Design


ConnectionFactory
- URL: String
- USER: String
- DRIVER: String
- PASSWORD: String
- connectionFactory: ConnectionFactory
+ getConnection(): Connection
+ close(connection: Connection): void
+ close(statement: Statement): void
+ close(resultSet: ResultSet): void
- ConnectionFactory(): void
- createConnection(): Connection
AbstractDAO
- LOGGER: Logger
- type: Class<T>
+ AbstractDAO(): void
+ findAll(): List<T>
+ findByName(name: String): T
+ findLast(): T
+ findById(id: int): T
+ insert(object: T): void
+ update(object: T, id: int): void
+ delete(id: int): void
- createDeleteQuery(): String
- createSelectQuery(field: String): String
- createSelectAllQuery(): String
- createInsertStatement(object: T): String
- createUpdateStatement(object: T): String
- createObjects(resultSet: ResultSet): List<T>
Extends
Extends
Extends
Extends
OrdersDAO
- No fields
+ OrdersDAO(): void
LogDAO
- No fields
+ LogDAO(): void
ClientDAO
- No fields
+ ClientDAO(): void
ProductDAO
- No fields
+ ProductDAO(): void
+ decrementStock(product: Product, quantity: int): void
View
- No fields
+ View(title: String): void
+ setMainPanel(panel: JPanel): void
+ editProductPane(): JPanel
+ editClientPane(): JPanel
+ deleteClientPane(): JPanel
+ deleteProductPane(): JPanel
+ createOrderPane(): JPanel
+ createClientPane(): JPanel
+ createProductPane(): JPanel
+ prepareTopBar(): void
+ populateTable(objects: List<?>): void
OrdersDTO
- clientName: String
- productName: String
- quantity: int
+ OrdersDTO(clientName: String, productName: String, quantity: int): void
+ getClientName(): String
+ getProductName(): String
+ getQuantity(): int
Controller
- No fields
+ createClient(clientDTO: ClientDTO): void
+ createProduct(productDTO: ProductDTO): void
+ createOrder(orderDTO: OrdersDTO): void
+ getAllClients(): List<ClientDTO>
+ getAllProducts(): List<ProductDTO>
+ getAllOrders(): List<OrdersDTO>
+ getAllLogs(): List<LogDTO>
+ getAllClientsNames(): List<String>
+ getAllProductsName(): List<String>
+ deleteClient(name: String): void
+ editClient(name: String, clientDTO: ClientDTO): void
+ deleteProduct(name: String): void
+ editProduct(name: String, productDTO: ProductDTO): void
LogDTO
- bill: String
+ LogDTO(bill: String): void
+ getBill(): String
ProductDTO
- name: String
- stock: int
- price: double
+ ProductDTO(name: String, price: double, stock: int): void
+ getName(): String
+ getStock(): int
+ getPrice(): double
ClientDTO
- name: String
- email: String
- address: String
+ ClientDTO(name: String, address: String, email: String): void
+ getName(): String
+ getAddress(): String
+ getEmail(): String
ProductLogic
- No fields
+ createProduct(productDTO: ProductDTO): void
+ deleteProduct(name: String): void
+ editProduct(name: String, productDTO: ProductDTO): void
+ getAllProducts(): List<ProductDTO>
+ getAllProductsName(): List<String>
ClientLogic
- No fields
+ createClient(clientDTO: ClientDTO): void
+ deleteClient(name: String): void
+ editClient(name: String, clientDTO: ClientDTO): void
+ getAllClients(): List<ClientDTO>
+ getAllClientsName(): List<String>
LogLogic
- No fields
+ getAllLogs(): List<LogDTO>
OrderLogic
- No fields
+ createOrder(ordersDTO: OrdersDTO): void
+ getAllOrders(): List<OrdersDTO>
Product
- id: int
- name: String
- price: double
- stock: int
+ Product(): void
+ Product(id: int, name: String, price: double, stock: int): void
+ Product(name: String, price: double, stock: int): void
+ getId(): int
+ setId(id: int): void
+ getName(): String
+ setName(name: String): void
+ getPrice(): double
+ setPrice(price: double): void
+ getStock(): int
+ setStock(stock: int): void
+ toString(): String
Client
- id: int
- name: String
- email: String
- address: String
+ Client(): void
+ Client(id: int, name: String, address: String, email: String): void
+ Client(name: String, address: String, email: String): void
+ getId(): int
+ setId(id: int): void
+ getName(): String
+ setName(name: String): void
+ getEmail(): String
+ setEmail(email: String): void
+ getAddress(): String
+ setAddress(address: String): void
+ toString(): String
Log
- id: int
- bill: String
+ Log(): void
+ Log(id: int, bill: String): void
+ Log(bill: String): void
+ getId(): int
+ setId(id: int): void
+ getBill(): String
+ setBill(bill: String): void
+ toString(): String
Orders
- id: int
- client_id: int
- product_id: int
- quantity: int
+ Orders(): void
+ Orders(id: int, client_id: int, product_id: int, quantity: int): void
+ Orders(client_id: int, product_id: int, quantity: int): void
+ getId(): int
+ setId(id: int): void
+ getClient_id(): int
+ setClient_id(client_id: int): void
+ getProduct_id(): int
+ setProduct_id(product_id: int): void
+ getQuantity(): int
+ setQuantity(quantity: int): void
+ toString(): String



4. Implementation

Class Descriptions:

1. **Client:**
   - **Fields:**
     - id: int
     - name: String
     - email: String
     - address: String
   - **Important Methods:**
     - Constructor with parameters: Initializes a client object with provided data.
     - Getters and setters: Access and modify client data.
2. **Log:**
   - **Fields:**
     - id: int
     - bill: String
   - **Important Methods:**
     - Constructor with parameters: Initializes a log object with provided data.
     - Getters and setters: Access and modify log data.
3. **Orders:**
   - **Fields:**
     - id: int
     - client_id: int
     - product_id: int

        - quantity: int
    - **Important Methods:**
        - Constructor with parameters: Initializes an order object with provided data.
        - Getters and setters: Access and modify order data.
4. **Product:**
    - **Fields:**
        - id: int
        - name: String
        - price: double
        - stock: int
    - **Important Methods:**
        - Constructor with parameters: Initializes a product object with provided data.
        - Getters and setters: Access and modify product data.

Graphical User Interface (GUI) Implementation:

1. **Client Management Interface:**
    - Form to add a new client with input fields for name, email, and address.
    - Table to display existing clients with options to edit or delete each client.
2. **Product Management Interface:**
    - Form to add a new product with input fields for name, price, and stock.
    - Table to display existing products with options to edit or delete each product.
3. **Order Management Interface:**
    - Form to create a new order with dropdowns to select a client and a product, and an input field for quantity.
    - Table to display existing orders with options to view, edit, or delete each order.
4. **Log Management Interface:**
    - Table to display system logs with columns for log id and bill description.

Implementation Details:

1. **GUI Framework:**
    - Utilize a Java GUI framework such as JavaFX or Swing for building the user interface components.
2. **Event Handling:**
    - Implement event handlers for user interactions such as button clicks and menu selections.
3. **Data Binding:**
    - Bind GUI components to corresponding data models (e.g., Client, Product) to ensure synchronization between user inputs and underlying data.
4. **Database Interaction:**
    - Implement DAO (Data Access Object) classes to handle database operations such as CRUD (Create, Read, Update, Delete) operations for each entity (Client, Product, Order, Log).

5. **Error Handling:**
    - Implement error handling mechanisms to provide feedback to users in case of invalid inputs or database errors.
6. **Layout and Design:**
    - Design visually appealing and intuitive user interfaces with appropriate layout and styling to enhance user experience.
7. **Testing and Debugging:**
    - Perform thorough testing of the GUI application to ensure proper functionality and identify and fix any bugs or issues.

5. Conclusions

In conclusion, the assignment has provided valuable insights into developing a comprehensive inventory management system using Java. Through the implementation of various classes and a graphical user interface, I have gained practical experience in designing, coding, and testing software solutions for real-world scenarios.

What I Have Learned:

1. **Object-Oriented Programming (OOP):** I have deepened my understanding of OOP principles such as encapsulation, inheritance, and polymorphism through the design and implementation of classes like Client, Product, Orders, and Log.
2. **Database Interaction:** The assignment allowed me to enhance my skills in database interaction by implementing DAO (Data Access Object) classes to perform CRUD operations and manage data persistence.
3. **Graphical User Interface (GUI) Development:** I have gained hands-on experience in developing user-friendly GUIs using Java GUI frameworks like JavaFX or Swing, including event handling, data binding, and layout design.
4. **Error Handling and Testing:** I have learned the importance of implementing robust error handling mechanisms and conducting thorough testing to ensure the reliability and stability of software applications.

Future Developments:

1. **Enhanced Functionality:** The inventory management system can be further enhanced with additional features such as reporting and analytics functionalities to provide valuable insights into sales trends and inventory optimization.
2. **Integration with External Systems:** Integrating the system with external APIs or services for tasks such as automated inventory updates or order processing can streamline operations and improve efficiency.
3. **User Authentication and Security:** Implementing user authentication mechanisms and ensuring data security through encryption and access control measures will be crucial for protecting sensitive information and preventing unauthorized access.
4. **Scalability and Performance Optimization:** As the system grows, optimizing performance and scalability will become increasingly important, requiring strategies such as database optimization and code refactoring.

Overall, the assignment has been a valuable learning experience, equipping me with practical skills and knowledge that can be applied to future software development projects.

6. Bibliography

https://dsrl.eu/courses/pt/materials/PT2024_A3_S2.pdf
https://dsrl.eu/courses/pt/materials/PT2024_A3_S1.pdf
https://dsrl.eu/courses/pt/materials/PT2024_A3.pdf
https://gitlab.com/utcn_dsrl/pt-reflection-example
https://dsrl.eu/courses/pt/