DOCUMENTATION

ASSIGNMENT *3*

STUDENT NAME: Butas Rafael-Dorian
GROUP: 30424

CONTENTS

1. Assignment Objective

Main Objective
The main objective of the assignment is to develop a comprehensive and efficient order management system that integrates client, product, order, and billing functionalities using Object-Oriented Programming (OOP) principles in Java.

Sub-objective	**Description**	**Section**
Define Requirements	Identify and document the functional and non-functional requirements of the system, including the primary use cases.	2. Problem Analysis, Modeling, Scenarios, Use Cases
Develop Use Cases	Create detailed use cases for each system functionality, including use case diagrams and descriptions.	2. Problem Analysis, Modeling, Scenarios, Use Cases
Design System Architecture	Design the overall system architecture using UML diagrams, ensuring clear separation of concerns and adherence to OOP principles.	3. Design
Design Data Model	Define the data model, including the data structures, relationships between entities, and database schema.	3. Design
Implement Core Functionality	Develop the core functionality for managing clients, products, orders, and bills, including CRUD operations and business logic.	4. Implementation
Implement Data Access Layer	Create the Data Access Object (DAO) layer for database interactions, ensuring efficient and secure data management.	4. Implementation
Implement Business Logic Layer	Develop the Business Logic Layer (BLL) to handle the core operations and validation logic of the application.	4. Implementation
Implement Validation	Create validation logic to ensure data integrity and enforce business rules across the system.	4. Implementation
Test the System	Conduct comprehensive testing to validate the functionality, performance, and security of the system, including unit tests, integration tests, and user acceptance tests.	Throughout the project
Document the System	Create detailed documentation covering the system design, implementation, and usage, ensuring clarity and completeness for future maintenance and development.	Throughout the project
Reflect and Conclude	Summarize the project outcomes, lessons learned, and potential future improvements, providing a holistic view of the development process.	5. Conclusions
Test the System	Conduct comprehensive testing to validate the functionality, performance, and security of the system, including unit tests, integration tests, and	Throughout the project

Sub-objective	Description	Section

2. Problem Analysis, Modeling, Scenarios, Use Cases

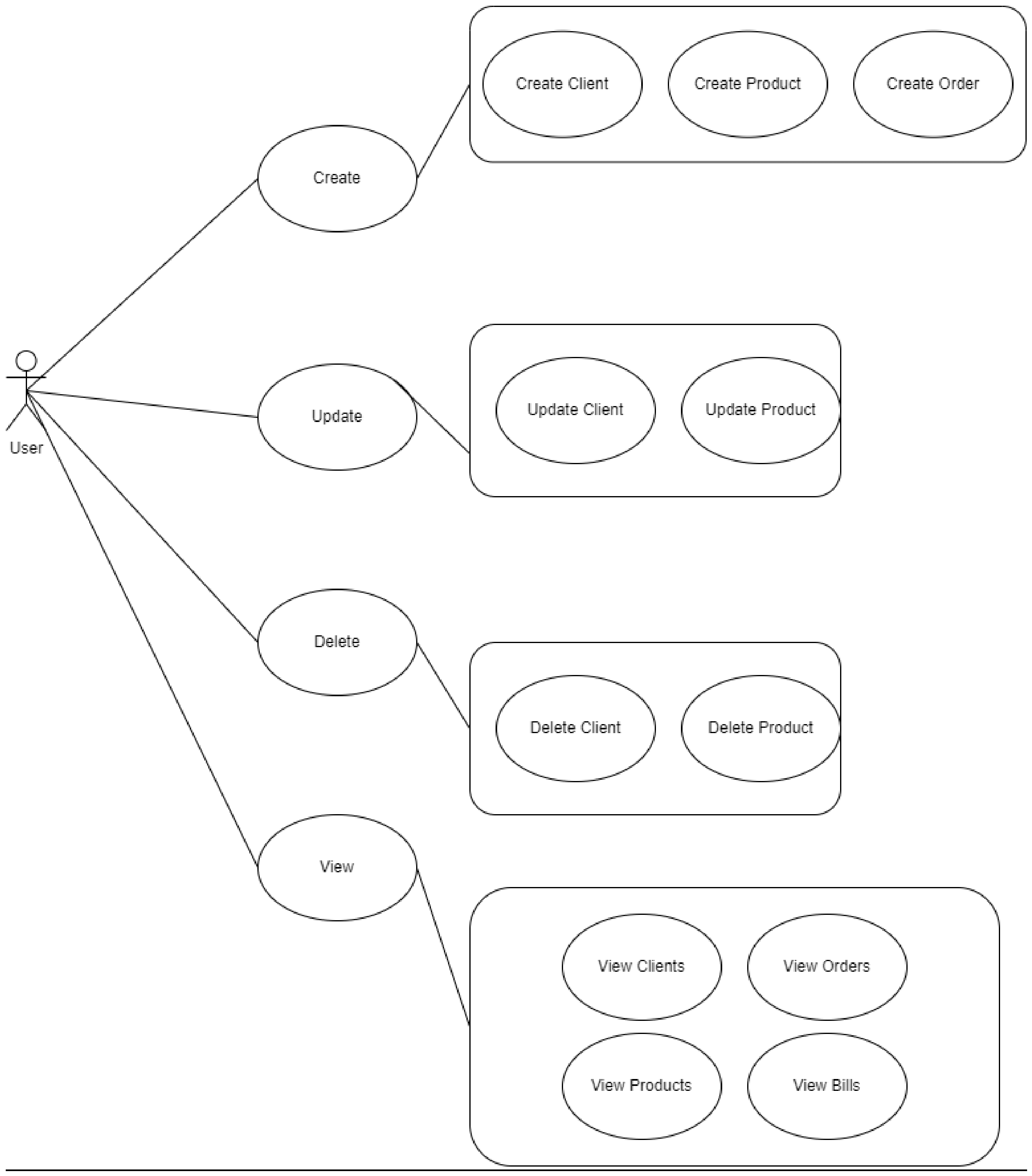
Functional Requirements:

- *Client Management:*
  - *Add Client:*
    - *The system shall allow the user to update existing client details.*
    - *The system shall validate the new input data and update the database accordingly.*
  - *Delete Client:*
    - *The system shall allow the user to delete a client from the database.*
    - *The system shall prompt the user for confirmation before deleting the client.*
  - *List Clients:*
    - *The system shall provide a list of all clients stored in the database.*
    - *The system shall allow the user to search and filter clients based on specific criteria.*
- *Product Management:*
  - *Add Product:*
    - *The system shall allow the user to add new products by providing necessary details such as name, quantity, and price.*
    - *The system shall validate the input data to ensure all required fields are filled and that the data is in the correct format.*
  - *Update Product:*
    - *The system shall allow the user to update existing product details.*
    - *The system shall validate the new input data and update the database accordingly.*
  - *Delete Product:*
    - *The system shall allow the user to delete a product from the database.*
    - *The system shall prompt the user for confirmation before deleting the product.*
  - *List Products:*
    - *The system shall provide a list of all products stored in the database.*
    - *The system shall allow the user to search and filter products based on specific criteria.*
- *Order Management:*

- *Create Order:*
    - *The system shall allow the user to create a new order by selecting a client and products.*
    - *The system shall calculate the total price of the order based on the selected products and their quantities.*
    - *The system shall validate the input data and ensure the selected products are available in the required quantities.*
- *List Orders:*
    - *The system shall provide a list of all orders stored in the database.*
    - *The system shall allow the user to search and filter orders based on specific criteria.*

- *Billing:*
    - *Generate Bill:*
        - *The system shall allow the user to generate a bill for a selected order.*
        - *The system shall include order details, client information, and the total amount in the bill.*
        - *The system shall allow the user to print or save the generated bill.*

<u>Use Case Diagram:</u>


User
Create
Update
Delete
View
Create Client
Create Product
Create Order
Update Client
Update Product
Delete Client
Delete Product
View Clients
View Orders
View Products
View Bills

Use Case Descriptions:

- *Use Case: Manage Clients*
  - *Actors: User*
  - *Description: This use case describes the steps involved in managing clients within the system, including adding, updating, deleting, and listing clients.*
  - *Add Client:*
    - *Preconditions: User is authenticated and has access to the client management interface.*
    - *Main Flow:*
      - *User selects the option to add a new client.*
      - *System displays a form for entering client details.*
      - *User fills in the client details and submits the form.*
      - *System validates the input data.*
      - *System adds the new client to the database.*
      - *System confirms the addition to the user.*
      - *Postconditions: New client is added to the database.*
    - *Alternate Flow:*
      - *If validation fails, system displays error messages and prompts user to correct the input data.*
- *Update Client:*
  - *Preconditions: User is authenticated and has access to the client management interface.*
  - *Main Flow:*
    - *User selects a client to update from the client list.*
    - *System displays the client details in a form.*
    - *User updates the client details and submits the form.*
    - *System validates the new input data.*
    - *System updates the client details in the database.*
    - *System confirms the update to the user.*
    - *Postconditions: Client details are updated in the database.*
- *Alternate Flow:*
  - *If validation fails, system displays error messages and prompts user to correct the input data.*
- *Delete Client:*
- *Preconditions: User is authenticated and has access to the client management interface.*
- *Main Flow:*
  - *User selects a client to delete from the client list.*
  - *System prompts the user for confirmation.*
  - *User confirms the deletion.*
  - *System removes the client from the database.*
  - *System confirms the deletion to the user.*
  - *Postconditions: Client is removed from the database.*
- *Alternate Flow:*
  - *If user cancels the confirmation, client is not deleted.*
- *List Clients:*

- *Preconditions: User is authenticated and has access to the client management interface.*
- *Main Flow:*
    - *User selects the option to view the list of clients.*
    - *System retrieves and displays the list of clients.*
    - *Postconditions: User views the list of clients.*
- *Use Case: Manage Products*
    - *Actors: User*
    - *Description: This use case describes the steps involved in managing products within the system, including adding, updating, deleting, and listing products.*
    - *Add Product:*
        - *Preconditions: User is authenticated and has access to the product management interface.*
        - *Main Flow:*
            - *User selects the option to add a new product.*
            - *System displays a form for entering product details.*
            - *User fills in the product details and submits the form.*
            - *System validates the input data.*
            - *System adds the new product to the database.*
            - *System confirms the addition to the user.*
            - *Postconditions: New product is added to the database.*
    - *Alternate Flow:*
        - *If validation fails, system displays error messages and prompts user to correct the input data.*
- *Update Product:*
    - *Preconditions: User is authenticated and has access to the client management interface.*
    - *Main Flow:*
        - *User selects a product to update from the product list.*
        - *System displays the product details in a form.*
        - *User updates the product details and submits the form.*
        - *System validates the new input data.*
        - *System updates the product details in the database.*
        - *System confirms the update to the user.*
        - *Postconditions: product details are updated in the database.*
- *Alternate Flow:*
    - *If validation fails, system displays error messages and prompts user to correct the input data.*
- *Delete Product:*
- *Preconditions: User is authenticated and has access to the product management interface.*
- *Main Flow:*
    - *User selects a product to delete from the product list.*
    - *System prompts the user for confirmation.*
    - *User confirms the deletion.*
    - *System removes the product from the database.*
    - *System confirms the deletion to the user.*

    - *Postconditions: product is removed from the database.*
- *Alternate Flow:*
    - *If user cancels the confirmation, product is not deleted.*
- *List Products:*
- *Preconditions: User is authenticated and has access to the product management interface.*
- *Main Flow:*
    - *User selects the option to view the list of products.*
    - *System retrieves and displays the list of products.*
    - *Postconditions: User views the list of products.*
- *Use Case: Manage Orders*
- *Actors: User*

- *Description: This use case describes the steps involved in managing orders within the system, including creating, updating, deleting, and listing orders.*

- *Create Order:*
- *Preconditions: User is authenticated and has access to the order management interface.*
- *Main Flow:*
    - *User selects the option to create a new order.*
    - *System displays a form for entering order details, including selecting a client and products.*
    - *User fills in the order details and submits the form.*
    - *System validates the input data and ensures the selected products are available in the required quantities.*
    - *System calculates the total price of the order.*
    - *System creates the new order in the database.*
    - *System confirms the creation to the user.*
    - *Postconditions: New order is created in the database.*
- *Alternate Flow:*
    - *If validation fails, system displays error messages and prompts user to correct the input data.*
    - *List Orders:*
- *Preconditions: User is authenticated and has access to the order management interface.*
- *Main Flow:*
    - *User selects the option to view the list of orders.*
    - *System retrieves and displays the list of orders.*
    - *Postconditions: User views the list of orders.*
- *Use Case: Generate Bill*
- *Actors: User*
- *Description: This use case describes the steps involved in generating a bill for a specific order.*
- *List Bills:*
- *Preconditions: User is authenticated and has access to the billing interface.*
- *Main Flow:*
- *User selects the option to view the list of bills.*

- *System retrieves and displays the list of bills.*
- *Postconditions: User views the list of bills.*

These detailed functional requirements and use cases provide a comprehensive overview of the system's functionalities and how users interact with them. This information will guide the design and implementation phases of the project.

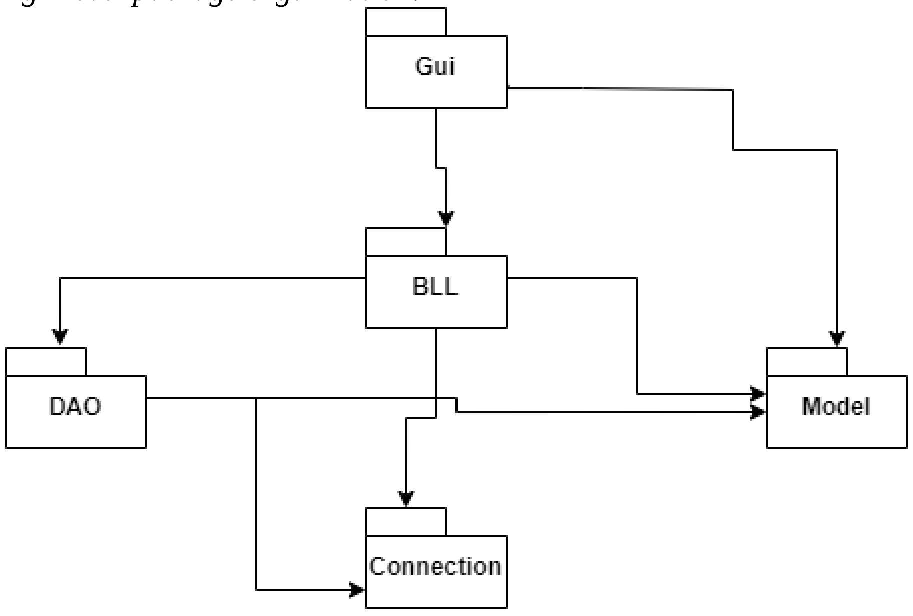
3. Design

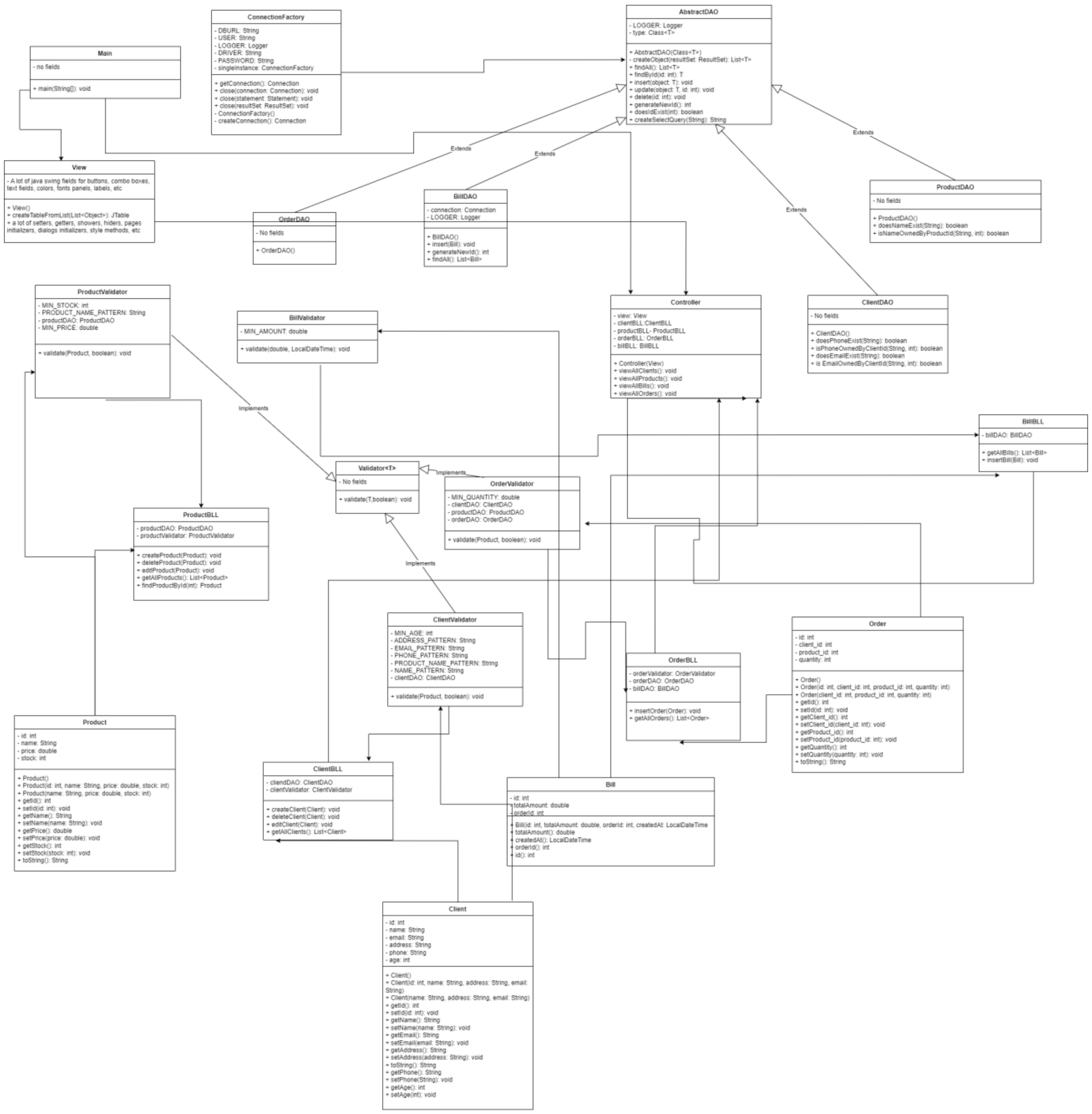
OOP Design

The system is designed following Object-Oriented Programming (OOP) principles to ensure modularity, reusability, and maintainability. The main components of the system include the data access layer (DAO), business logic layer (BLL), and the presentation layer (GUI). Each layer interacts with others through well-defined interfaces, encapsulating functionality and data management.

UML Package Diagram

The system is organized into several packages to separate concerns and improve maintainability. Below is the high-level package organization:




Main
- no fields
+ main(String[]): void
ConnectionFactory
- DBURL: String
- USER: String
- LOGGER: Logger
- DRIVER: String
- PASSWORD: String
- singleInstance: ConnectionFactory
+ getConnection(): Connection
+ close(connection: Connection): void
+ close(statement: Statement): void
+ close(resultSet: ResultSet): void
- ConnectionFactory()
- createConnection(): Connection
AbstractDAO
- LOGGER: Logger
- type: Class<T>
+ AbstractDAO(Class<T>)
- createObject(resultSet: ResultSet): List<T>
+ findAll(): List<T>
+ findById(id: int): T
+ insert(object: T): void
+ update(object: T, id: int): void
+ delete(id: int): void
+ generateNewId(): int
+ doesIdExist(int): boolean
+ createSelectQuery(String): String
Extends
Extends
Extends
Extends
View
- A lot of java swing fields for buttons, combo boxes, text fields, colors, fonts panels, labels, etc
+ View()
+ createTableFromList(List<Object>): JTable
+ a lot of setters, getters, showers, hiders, pages initializers, dialogs initializers, style methods, etc
OrderDAO
- No fields
+ OrderDAO()
BillDAO
- connection: Connection
- LOGGER: Logger
+ BillDAO()
+ insert(Bill): void
+ generateNewId(): int
+ findAll(): List<Bill>
ProductDAO
- No fields
+ ProductDAO()
+ doesNameExist(String): boolean
+ isNameOwnedByProductId(String, int): boolean
ProductValidator
- MIN_STOCK: int
- PRODUCT_NAME_PATTERN: String
- productDAO: ProductDAO
- MIN_PRICE: double
+ validate(Product, boolean): void
BillValidator
- MIN_AMOUNT: double
+ validate(double, LocalDateTime): void
Controller
- view: View
- clientBLL:ClientBLL
- productBLL- ProductBLL
- orderBLL: OrderBLL
- billBLL: BillBLL
+ Controller(View)
+ viewAllClients(): void
+ viewAllProducts(): void
+ viewAllBills(): void
+ viewAllOrders(): void
ClientDAO
- No fields
+ ClientDAO()
+ doesPhoneExist(String): boolean
+ isPhoneOwnedByClientId(String, int): boolean
+ doesEmailExist(String): boolean
+ is EmailOwnedByClientId(String, int): boolean
Implements
BillBLL
- billDAO: BillDAO
+ getAllBills(): List<Bill>
+ insertBill(Bill): void
Validator<T>
- No fields
+ validate(T,boolean): void
Implements
OrderValidator
- MIN_QUANTITY: double
- clientDAO: ClientDAO
- productDAO: ProductDAO
- orderDAO: OrderDAO
+ validate(Product, boolean): void
ProductBLL
- productDAO: ProductDAO
- productValidator: ProductValidator
+ createProduct(Product): void
+ deleteProduct(Product): void
+ editProduct(Product): void
+ getAllProducts(): List<Product>
+ findProductById(int): Product
Implements
ClientValidator
- MIN_AGE: int
- ADDRESS_PATTERN: String
- EMAIL_PATTERN: String
- PHONE_PATTERN: String
- PRODUCT_NAME_PATTERN: String
- NAME_PATTERN: String
- clientDAO: ClientDAO
+ validate(Product, boolean): void
Order
- id: int
- client_id: int
- product_id: int
- quantity: int
+ Order()
+ Order(id: int, client_id: int, product_id: int, quantity: int)
+ Order(client_id: int, product_id: int, quantity: int)
+ 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
OrderBLL
- orderValidator: OrderValidator
- orderDAO: OrderDAO
- billDAO: BillDAO
+ insertOrder(Order): void
+ getAllOrders(): List<Order>
Product
- id: int
- name: String
- price: double
- stock: int
+ Product()
+ Product(id: int, name: String, price: double, stock: int)
+ Product(name: String, price: double, stock: int)
+ 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
ClientBLL
- cliendDAO: ClientDAO
- clientValidator: ClientValidator
+ createClient(Client): void
+ deleteClient(Client): void
+ editClient(Client): void
+ getAllClients(): List<Client>
Bill
- id: int
- totalAmount: double
- orderId: int
+ Bill(id: int, totalAmount: double, orderId: int, createdAt: LocalDateTime
+ totalAmount(): double
+ createdAt(): LocalDateTime
+ orderId(): int
+ id(): int
Client
- id: int
- name: String
- email: String
- address: String
- phone: String
- age: int
+ Client()
+ Client(id: int, name: String, address: String, email: String)
+ Client(name: String, address: String, email: String)
+ 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
+ getPhone(): String
+ setPhone(String): void
+ getAge(): int
+ setAge(int): void

4. Implementation

This section provides detailed descriptions of each class, including fields and important methods. The implementation of the graphical user interface (GUI) is also described.

- Main.java
  - Description: The entry point of the application. It initializes the controller and starts the GUI.
  - Fields: None
  - Important Methods:
    - main(String[] args): Initializes the Controller and starts the application.

- Controller.java
    - Description: Manages the interaction between the GUI and the BLL. Handles user actions and updates the view accordingly.
    - Fields:
        - View view: The GUI component.
        - ClientBLL clientBLL: Business logic for managing clients.
        - ProductBLL productBLL: Business logic for managing products.
        - OrderBLL orderBLL: Business logic for managing orders.
        - BillBLL billBLL: Business logic for managing bills.
- Important Methods:
    - Controller(View view): Constructor that initializes the view and BLL components.
    - initialize(): Sets up action listeners for GUI components.
- Client.java
- Description: Represents a client in the system.
- Fields:
    - int id: Unique identifier for the client.
    - String name: Name of the client.
    - String address: Address of the client.
    - String email: Email address of the client.
    - String phone: Phone number of the client.
- Methods:
    - Getters and setters for each field.
    - Constructors for initializing client objects.
- Product.java
- Description: Represents a product in the system.
- Fields:
    - int id: Unique identifier for the product.
    - String name: Name of the product.
    - int quantity: Available quantity of the product.
    - double price: Price of the product.
- Methods:
    - Getters and setters for each field.
    - Constructors for initializing product objects.
- Order.java
- Description: Represents an order in the system.
- Fields:
    - int id: Unique identifier for the order.
    - int clientId: Identifier of the client who placed the order.
    - int productId: Identifier of the product in the order.
    - int quantity: Quantity of the product ordered.
    - double totalPrice: Total price of the order.
- Methods:
    - Getters and setters for each field.
    - Constructors for initializing order objects.
- Bill.java

- Description: Represents a bill in the system.
- Fields:
    - int id: Unique identifier for the bill.
    - int orderId: Identifier of the order associated with the bill.
    - Date date: Date when the bill was generated.
    - double totalAmount: Total amount of the bill.
- DAO Package:
    - AbstractDAO: methods for creating queries, methods that use the queries to add/update/delete objects.
    - BillDAO, ClientDAO, OrderDAO, ProductDAO: these classes inherit the methods from the AbstractDAO and implement some other specific functionalities.
- BLL Package:
    - BillBLL, ClientBLL, OrderBLL, ProductBLL: used to facilitate the communication between the GUI and database, allowing to filter things by logic operations.
- Connection Package:
    - ConnectionFactory: utility class for managing connections to the database.Validators
- Validator.java
    - Description: Interface for validators.
    - Methods:
        - validate(Object object): Validates the given object.
- ClientValidator.java
- Description: Validator for Client entities.
- Methods:
        - validate(Client client): Validates client details such as name, email, and phone.
- ProductValidator.java
    - Description: Validator for Product entities.
    - Methods:
        - validate(Product product): Validates product details such as name, quantity, and price.
- OrderValidator.java
    - Description: Validator for Order entities.
    - Methods:
        - validate(Order order): Validates order details such as product availability and quantity.
- BillValidator.java
    - Description: Validator for Bill entities.
    - Methods:
        - validate(Bill bill): Validates bill details such as total amount and order reference.
- View.java
    - Description: Implements the graphical user interface using Java Swing. Provides forms and tables for managing clients, products, orders, and bills.

        - <u>Action Listeners:</u>
          The Controller class sets up action listeners for various GUI components, such as buttons. These listeners handle user actions, such as clicking a button to add a new client, by invoking the corresponding methods in the BLL.
        - The implementation section provides detailed descriptions of each class involved in the order management system. This includes the fields and important methods for each class, as well as an overview of the GUI implementation using Java Swing. This detailed documentation ensures that each component of the system is well understood, facilitating maintenance, future development, and overall system comprehension.

5. Conclusions

The development of the order management system successfully achieved the main objective of integrating client, product, order, and billing functionalities. The system adheres to Object-Oriented Programming (OOP) principles, ensuring modularity, reusability, and maintainability. The project was divided into distinct layers: the data access layer (DAO), the business logic layer (BLL), and the presentation layer (GUI), each encapsulating specific responsibilities and interacting through well-defined interfaces.

- Key Achievements
    - Comprehensive Functionalities:
    - The system supports all the essential operations: managing clients, products, orders, and generating bills.
    - CRUD operations are implemented efficiently for each entity, ensuring data integrity and consistency.
- Robust Architecture:
    - The application is designed using OOP principles, ensuring clear separation of concerns and high cohesion within each component.
    - The use of DAO and BLL layers ensures that the business logic and data access are modular and can be maintained or extended easily.
- User-Friendly Interface:
    - The graphical user interface, developed using Java Swing, provides a user-friendly environment for interacting with the system.
    - The GUI includes forms and tables for managing data and ensures a smooth user experience with minimal training.
- Validation and Error Handling:
    - Comprehensive validation is implemented across the system to ensure data accuracy and reliability.
    - Error handling mechanisms are in place to guide the user in case of invalid inputs or system errors.
- <u>Lessons Learned:</u>
    - Importance of Clear Requirements:
    - Defining clear and detailed functional requirements at the beginning of the project is crucial for guiding the development process and ensuring that the final product meets user expectations.

        - Effective Use of OOP Principles:
            - Applying OOP principles such as encapsulation, inheritance, and polymorphism enhances code readability, maintainability, and scalability.
            - Proper design of class hierarchies and interfaces simplifies future enhancements and maintenance.
        - Layered Architecture Benefits:
            - A layered architecture, separating the GUI, business logic, and data access, improves modularity and allows independent development and testing of each layer.
            - It also facilitates easier debugging and troubleshooting by isolating issues within specific layers.
        - Validation and Testing:
            - Implementing validation and testing from the early stages of development helps in identifying and rectifying issues promptly, ensuring a robust and error-free system.
        - User-Centric Design:
            - Designing the system with the end-user in mind, focusing on usability and ease of navigation, significantly enhances the user experience.
            - Gathering feedback from potential users during the design phase can provide valuable insights for improving the interface and functionality.

    - <u>*Future Developments*</u>
        - User Authentication and Authorization:
        - Implementing user authentication and role-based authorization to enhance security and ensure that only authorized users can access specific functionalities.
        - Advanced Reporting Features:
            - Adding reporting features to generate detailed reports and analytics on clients, products, orders, and bills, providing valuable insights for decision-making.
        - Improved GUI:
            - Enhancing the graphical user interface to be more intuitive and responsive, possibly using modern frameworks such as JavaFX.
            - Adding more interactive elements, such as drag-and-drop functionality and dynamic search filters.
        - Integration with External Systems:
            - Integrating the system with external payment gateways for processing payments securely.
            - Implementing APIs for interoperability with other systems, such as inventory management or accounting software.
        - Performance Optimization:
            - Optimizing database queries and operations to improve system performance, especially for large datasets.
            - Implementing caching mechanisms to reduce the load on the database and speed up data retrieval.
        - Mobile Application:

        - Developing a mobile version of the application to provide users with the flexibility to manage orders and clients on the go.
    - Enhanced Validation and Error Handling:
        - Further improving validation logic to cover more edge cases and enhance data integrity.
        - Implementing more user-friendly error messages and providing suggestions for correcting errors.

The order management system project has been a significant learning experience, providing valuable insights into the application of OOP principles, the importance of a robust architecture, and the need for comprehensive validation and testing. The project successfully met its objectives, delivering a functional and user-friendly system that can be easily extended and maintained. The lessons learned and the potential future developments outlined in this document provide a solid foundation for further enhancing the system and applying these insights to future projects.

6. Bibliography

1. https://dsrl.eu/courses/pt/materials/PT2024_A3_S2.pdf
2. https://dsrl.eu/courses/pt/materials/PT2024_A3_S1.pdf
   https://dsrl.eu/courses/pt/materials/PT2024_A3.pdf
3. https://gitlab.com/utcn_dsrl/pt-reflection-example
4. https://www.baeldung.com/javadoc
5. https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html