DOCUMENTATION

ASSIGNMENT 3

Orders Management System

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA ROMANIA

STUDENT NAME: Seserman Victor

GROUP: 30422

CONTENTS

1. Assignment Objective

Consider an application Orders Management for processing client orders for a warehouse. Relational databases should be used to store the products, the clients, and the orders. The application should be designed according to the layered architecture pattern and should use (minimally) the following classes:

• Model classes - represent the data models of the application

• Business Logic classes - contain the application logic

• Presentation classes – GUI related classes

• Data access classes - classes that contain the access to the database

Note: Other classes and packages can be added to implement the full functionality of the application.

2. Problem Analysis, Modeling, Scenarios, Use Cases

a) Analyzing the problem

In order to solve the problem, there are many scenarios that have to be taken into consideration, the most important one being the structure of the problem, how the classes communicate with each other and how the classes get divided into packages.

We have 3 important classes that need to be defined:

-Client, which has an id, a name and an address.

-Product, which has an id, a name, a price and a stock to keep track of how many products there are left;

-Order, which has an id, an id for the client, an id for the product and the amount of products that are ordered.

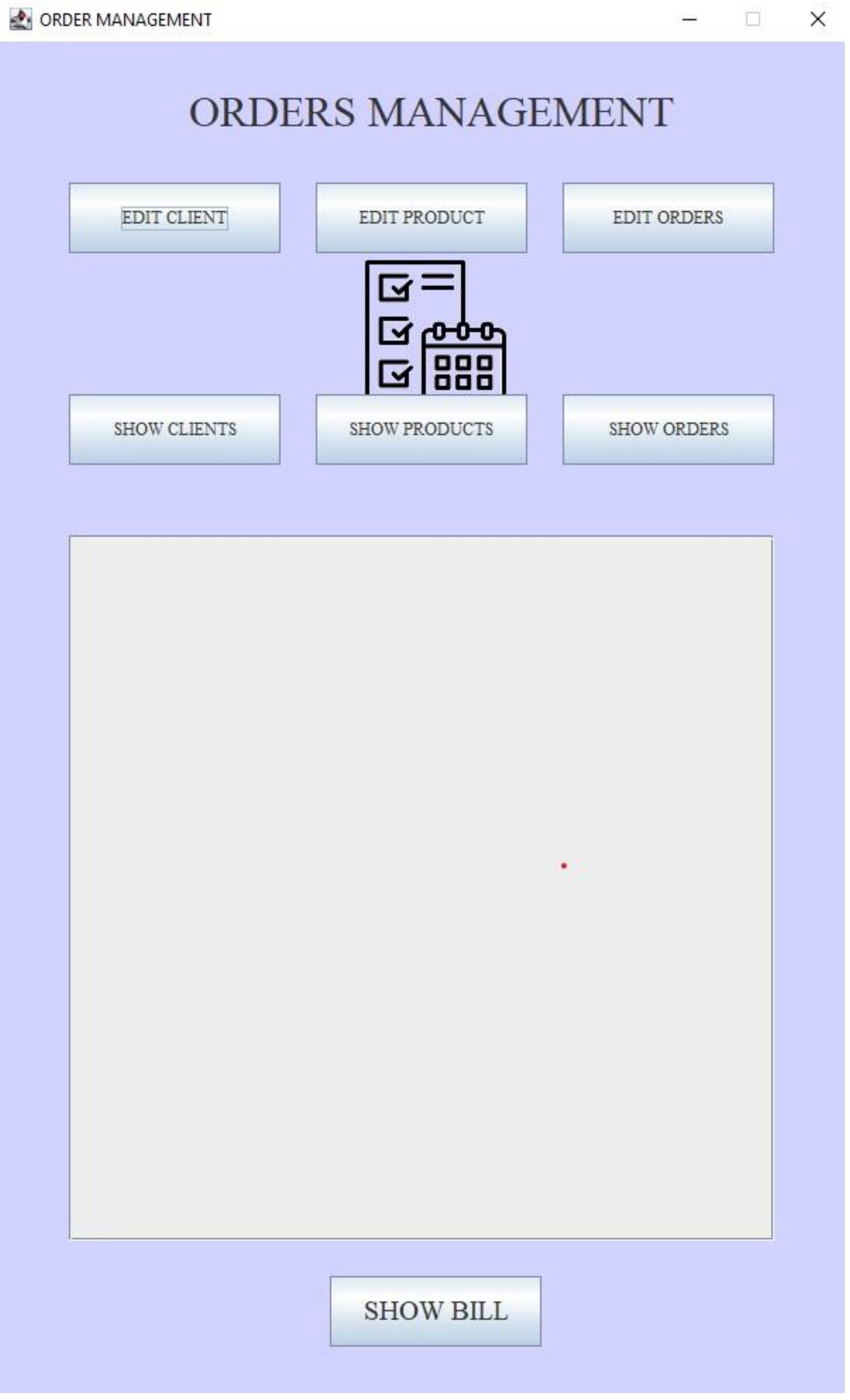
b) Modeling the problem

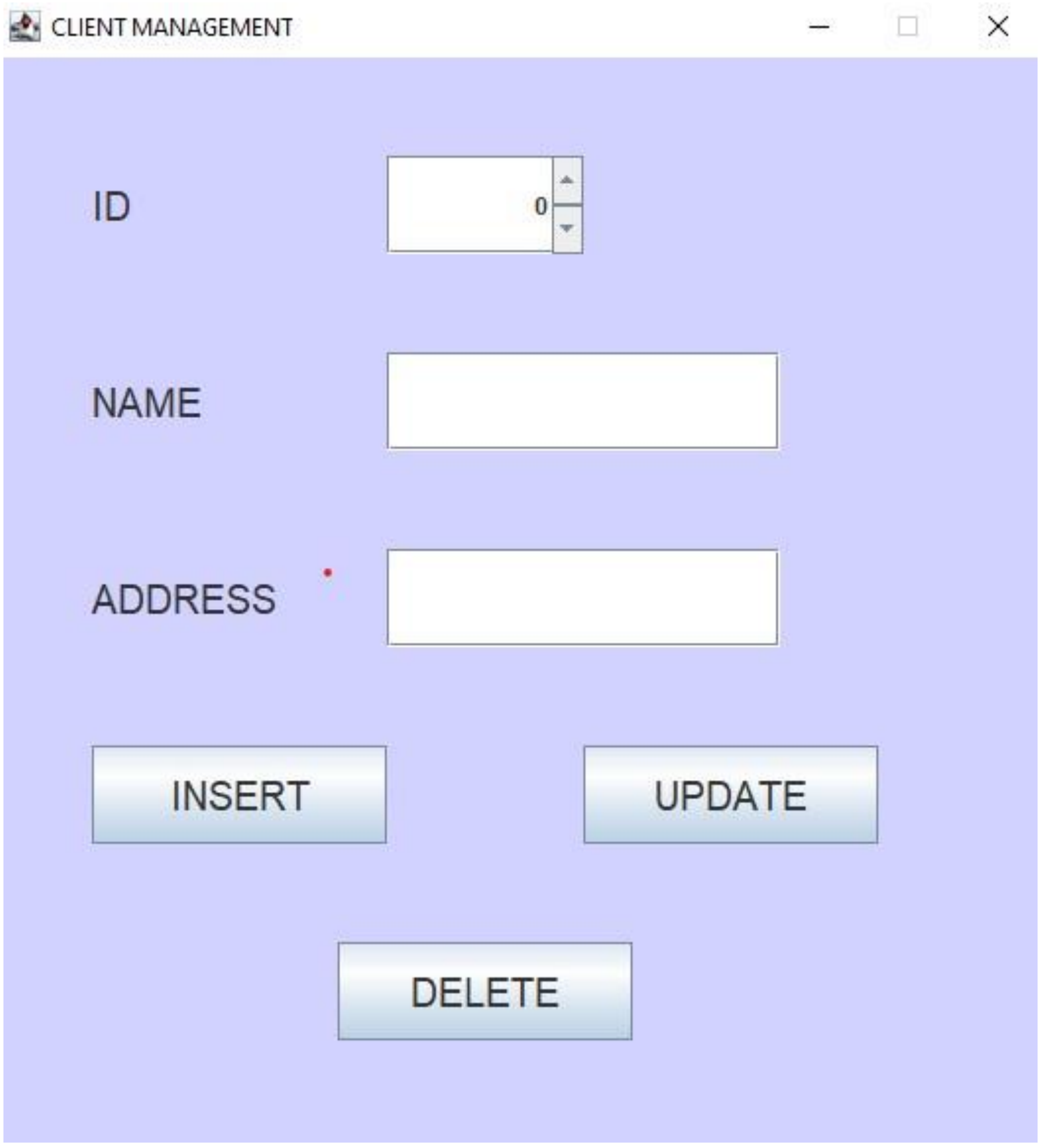
The user will be able to choose what they want to see, clients, products or orders, and then, they can choose to edit the database. The user can change information about these three parts, delete a product, client or order, or they can add a new one. Which means that the operations used on the database will be SELECT, INSERT, UPDATE or DELETE.

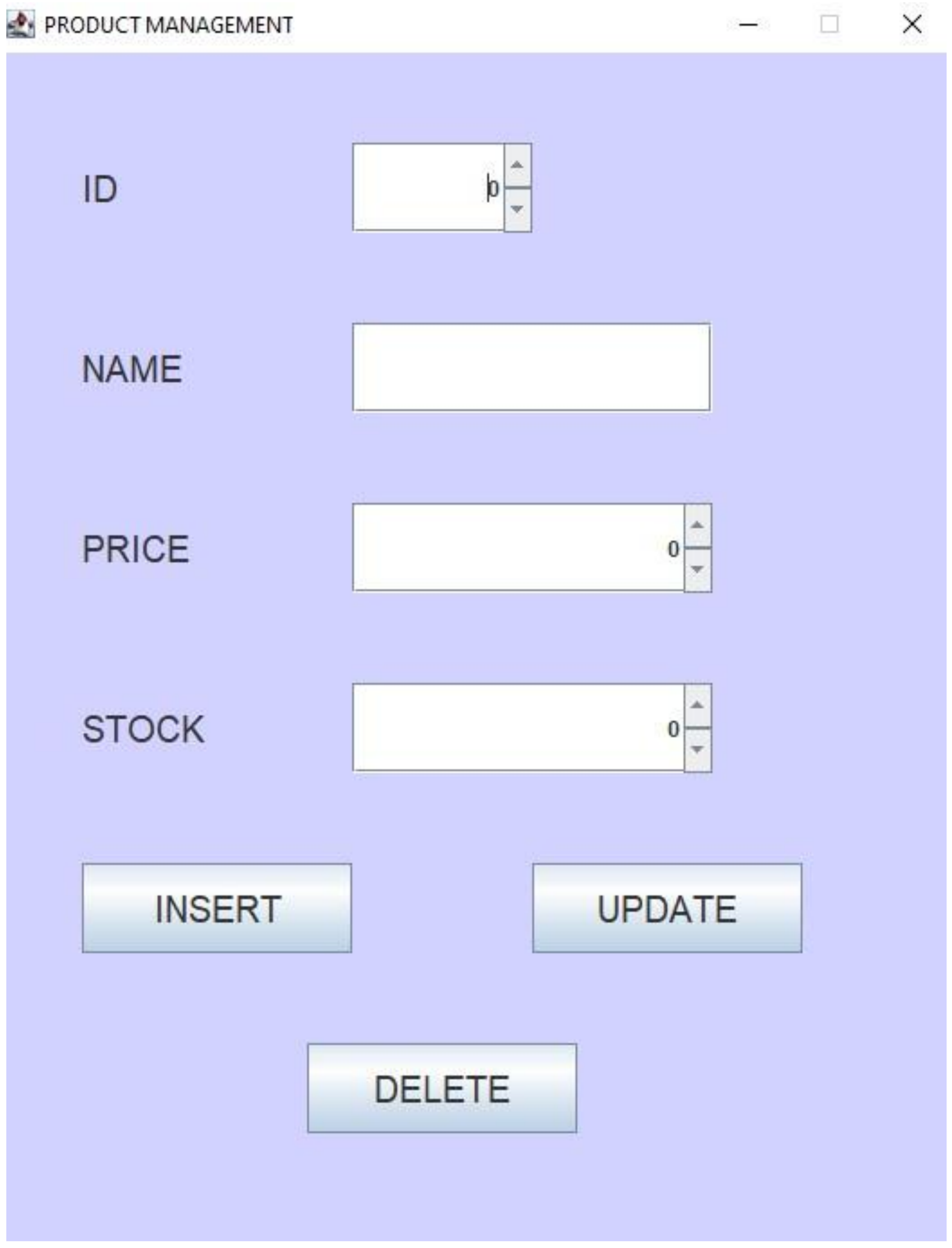
c) Scenarios and use cases

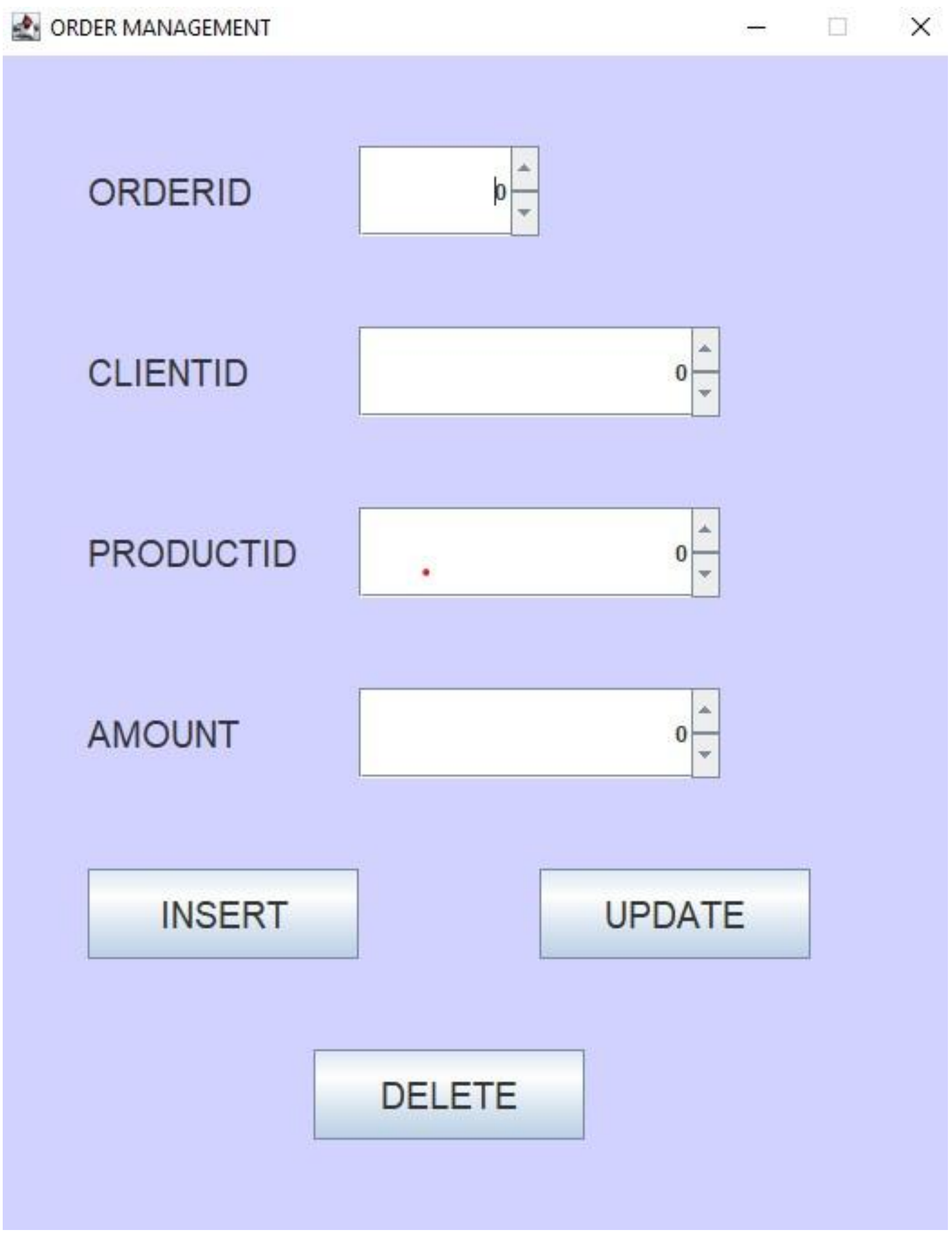
A use case is a methodology used in system analysis to identify, clarify and organize system requirements. The use case is made up of a set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal.

The use cases are strongly connected with the user steps. That is the reason why the design of the interface is very friendly and below is the result.


ORDER MANAGEMENT
ORDERS MANAGEMENT
EDIT CLIENT
EDIT PRODUCT
EDIT ORDERS
SHOW CLIENTS
SHOW PRODUCTS
SHOW ORDERS
SHOW BILL


CLIENT MANAGEMENT
ID
0
NAME
ADDRESS
INSERT
UPDATE
DELETE

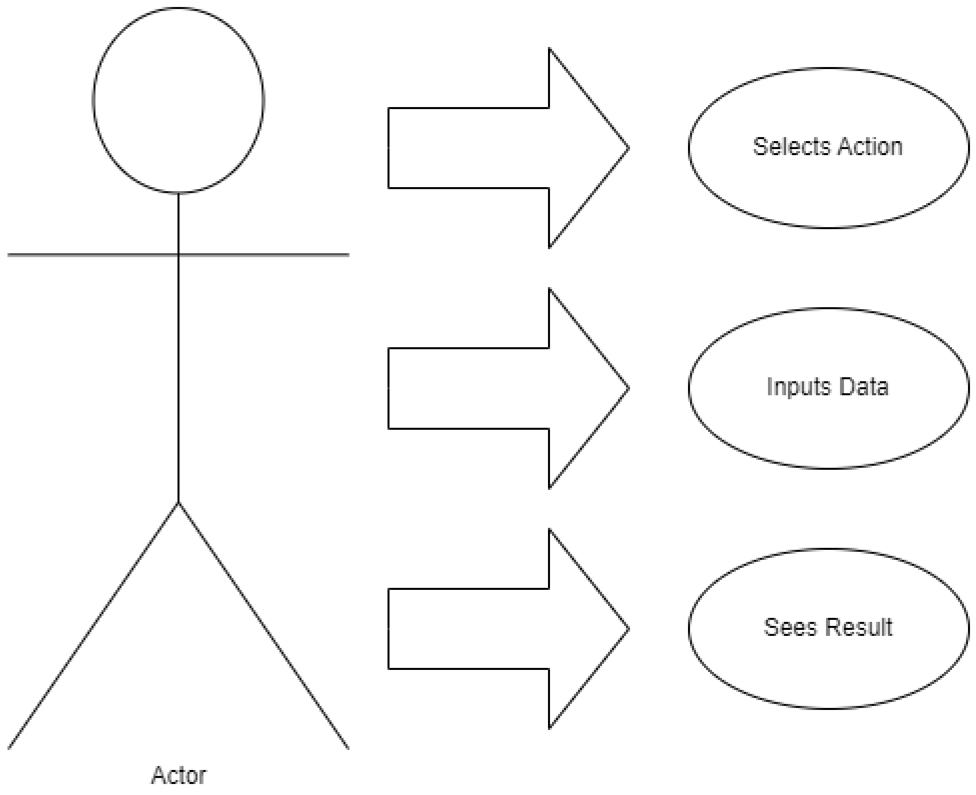

PRODUCT MANAGEMENT
ID
0
NAME
PRICE
0
STOCK
0
INSERT
UPDATE
DELETE



3. Design

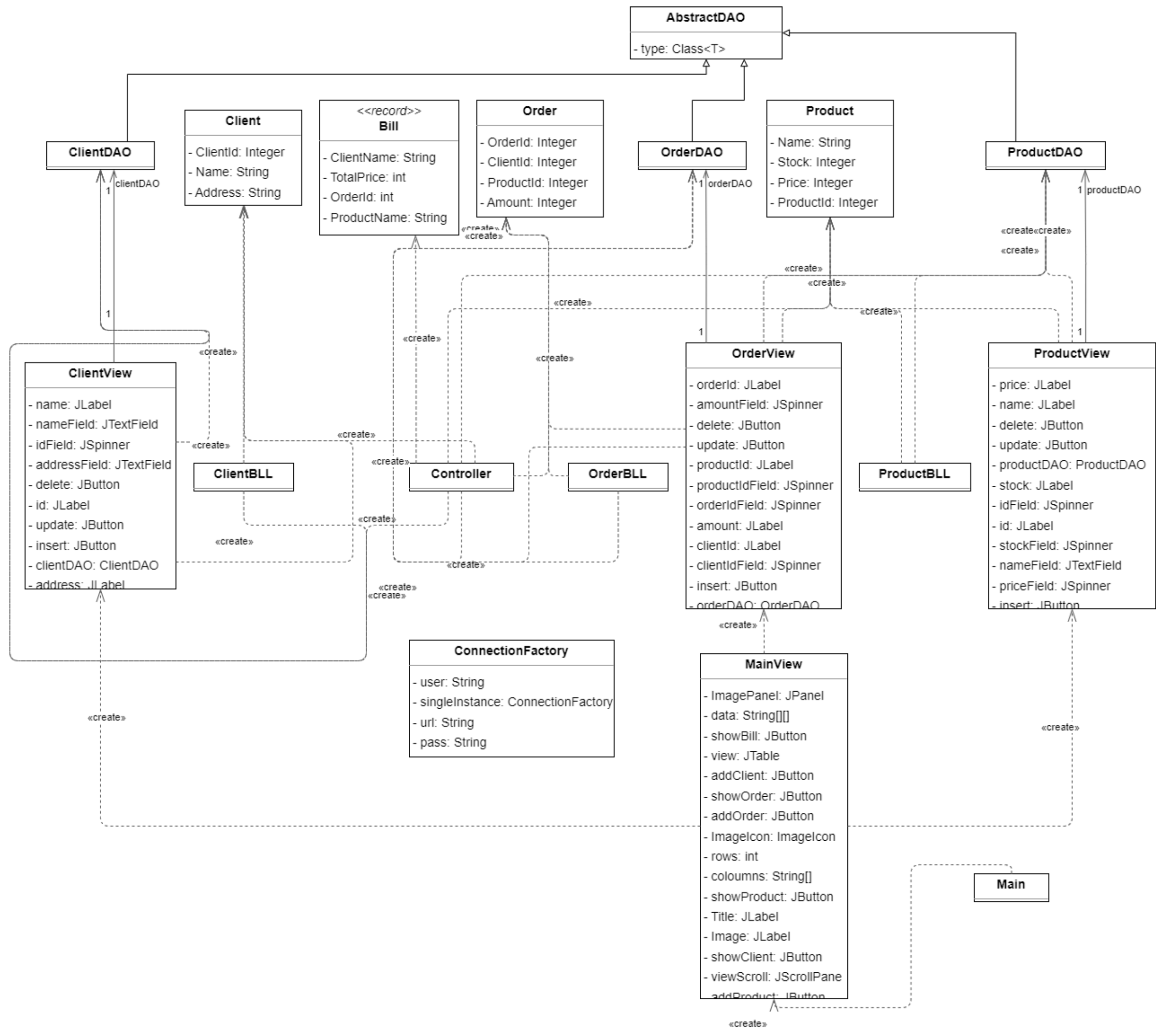
a) Diagrams

Use cases diagram:



The use cases present the user which can perform different actions mentioned above. They can perform several actions on the application and they can also interact with the application in a multitude of ways.

Class Diagram:



b) Data Structures

The data structures used in this projects are of two types: primitive, such as Integer, Double and String, and more complex, the only complex data structure used is the List Interface.

c) Packages

The packages used in order to solve the problem are organized using a layered architecture:

-Model, this represents the logical structure of the project and the high-level class associated with it;

-BusinessLogic, this represents the algorithms used for calculating various things;

-Connection, here we can find the connection to the database,

-DAO, here we can find the relationship between our java application and the database;

-Presentation, this represents everything the user sees and interacts with;   -
App, this is where our application begins.

4. Implementation

Classes:

Client class:

- Represents a model for a client entity.
- Stores information about a client's ID, name, and address.
- Provides constructors to initialize the instance variables.
- Offers getter and setter methods to access and modify the values of the instance variables.

Order class:

- Represents a model for an order entity.
- Stores information about an order's ID, client ID, product ID, and amount.
- Provides constructors to initialize the instance variables.
- Offers getter and setter methods to access and modify the values of the instance variables.

Product class:

- Represents a model for a product entity.
- Stores information about a product's ID, name, price, and stock.
- Provides constructors to initialize the instance variables.

- Offers getter and setter methods to access and modify the values of the instance variables.

Bill record:

- Represents a record for a bill, containing information such as the order ID, client name, product name, and total price.
- The record class is introduced in Java 14 and is similar to a class, but with some additional features for concise declaration and immutability.
- The `Bill` record is declared using the `record` keyword, followed by the record name and a list of its components.
- The components of the `Bill` record are defined as `int OrderId`, `String ClientName`, `String ProductName`, and `int TotalPrice`.
- The record automatically generates the constructor, getter methods, `equals()`, `hashCode()`, and `toString()` methods for its components.

ConnectionFactory class:

- The class has private instance variables `url`, `user`, and `pass` representing the database connection URL, username, and password, respectively.
- The `ConnectionFactory` class follows the Singleton design pattern, as it has a private static instance `singleInstance` and a private constructor, ensuring that only one instance of the class is created.
- The `createConnection()` method is a private method that establishes a connection to the database using the provided URL, username, and password.

- The `getConnection()` method is a public static method that returns an instance of the connection by invoking the `createConnection()` method on the singleton instance.
- The `close(Connection connection)`, `close(Statement statement)`, and

`close(ResultSet resultSet)` methods are static methods used to close the database resources. They accept the corresponding resource objects and close them if they are not null, handling any potential SQLExceptions.

Controller class:

- This class is responsible for handling any algorithms
- The methods it contains are calculateRowsClients, showClients, calculateProductsClients, showProducts, calculateRowsOrders, showOrders, showBills, createInsertBill, createDeleteBill, createBill, updateBill; **AbstractDAO class:**
- This is the class responsible for accessing the database;
- Here is where most of the project is solved;
- This class is the most complicated one, but everything that is contained is self-explanatory;
- The Select, Insert, Delete and Update queries for each table are created and the methods are implemented using reflection techniques.

ClientDAO, ProductDAO, OrderDAO classes:

- All three classes extend AbstractDAO, they exist so that we can give the AbstractDAO its generic type.

ClientView, ProductView, OrderView classes:

- All three are responsible for inserting, deleting and updating objects in the database.

MainView class:

- This is responsible for viewing the clients, products, orders and the user can choose to see a bill;
- Here the user can choose what they want to edit, client, product or order.

Main class:

- This is where the application begins.

5. Conclusions

Personally, this assignment taught me a bunch of new ideas for structuring projects, it was a fun, but annoying experience. This project was probably the one where actual coding took the longest, there were a small number of errors that happened along the way. I liked the fact that the problem was concrete, but at the same time it felt like we could really put our own trademark into it. Solving the assignment with my colleagues was also a fun experience because it seemed like every project was different, yet we did basically the same thing.

6. Bibliography

- **PT courses**; - **JetBrains DataGrip**;
- **JavaDoc**.