

# **DOCUMENTATION**

**PROGRAMMING TECHNIQUES**

**ASSIGNMENT**



**ORDERS MANAGEMENT**

STUDENT NAME: NECHITA FLORINA-ELENA  
GROUP: 30424\_2

## CONTENTS

1. Assignment Objective .....	3
2. Problem Analysis, Modeling, Scenarios, Use Cases.....	3
3. Design .....	6
4. Further improvements .....	<b>Eroare! Marcaj în document nedefinit.</b>
5. Results.....	13
6. Conclusions.....	13
7. Bibliography .....	13

## 1. Assignment Objective

The objective of this assignment is to develop a Java application for managing clients, products, and orders, with a graphical user interface. The application should adhere to the following specifications:

- Use object-oriented programming design principles, with classes limited to a maximum of 300 lines and methods limited to a maximum of 30 lines, following Java naming conventions.
- Utilize Javadoc for documenting classes and generate corresponding Javadoc files.
- Store data in a relational database, with a minimum of three tables: Client, Product, and Order.
- Create a graphical user interface with the following features:
  - A window for client operations, including adding new clients, editing existing clients, deleting clients, and viewing all clients in a table (JTable).
  - A window for product operations, including adding new products, editing existing products, deleting products, and viewing all products in a table (JTable).
  - A window for creating product orders, allowing users to select an existing product, select an existing client, and enter the desired quantity for the product to create a valid order. If there are insufficient products in stock, an under-stock message should be displayed. After finalizing the order, the product stock should be decremented accordingly.
- Utilize reflection techniques to create a method that receives a list of objects and dynamically generates the table header by extracting the object properties through reflection. The method should also populate the table with the values of the elements from the list.
- Ensure good quality documentation that covers all the sections outlined in the documentation template.
- Implement a layered architecture with the following packages: dataAccessLayer, businessLayer, model, and presentation.
- Define an immutable Bill class using Java records within the Model package. Each Bill object will be generated for an order and stored in a Log table. No updates will be allowed for the bills; they can only be inserted and read from the Log table.
- Use reflection techniques to create a generic class that contains methods for accessing the database for all tables except the Log table. The class should support object creation, editing, deletion, and retrieval. The queries for accessing the database for a specific object corresponding to a table should be generated dynamically through reflection.

An application called Orders Management is designed to handle client orders for a warehouse. The system utilizes relational databases to store information about products, clients, and orders. To ensure a well-structured architecture, the application follows the layered architecture pattern and includes the following essential classes:

**Model Classes:** These classes represent the data models used in the application. They encapsulate the attributes and behaviors of products, clients, and orders.

**Business Logic Classes:** These classes contain the core logic and processing of the application. They handle tasks such as validating orders, managing inventory, calculating pricing, and interacting with the data access layer.

**Presentation Classes:** These classes are responsible for the graphical user interface (GUI) and user interaction. They facilitate the display of information, capture user input, and provide an intuitive interface for managing orders.

**Data Access Classes:** These classes handle the interaction with the relational databases. They encapsulate the database operations required for retrieving, inserting, updating, and deleting data related to products, clients, and orders.

By adhering to this layered architecture pattern and utilizing these classes, the Orders Management application ensures a separation of concerns, modularity, and maintainability in the system design.

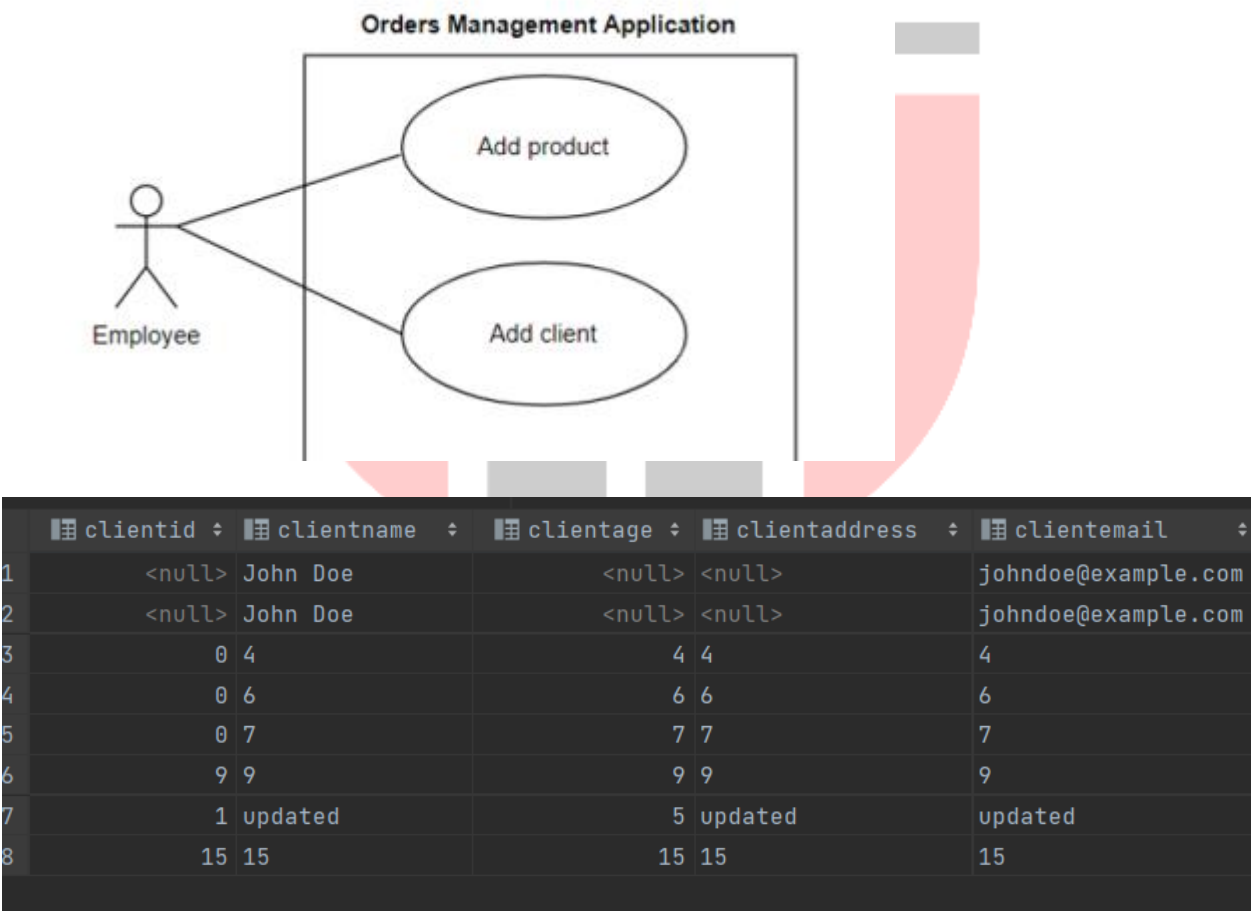
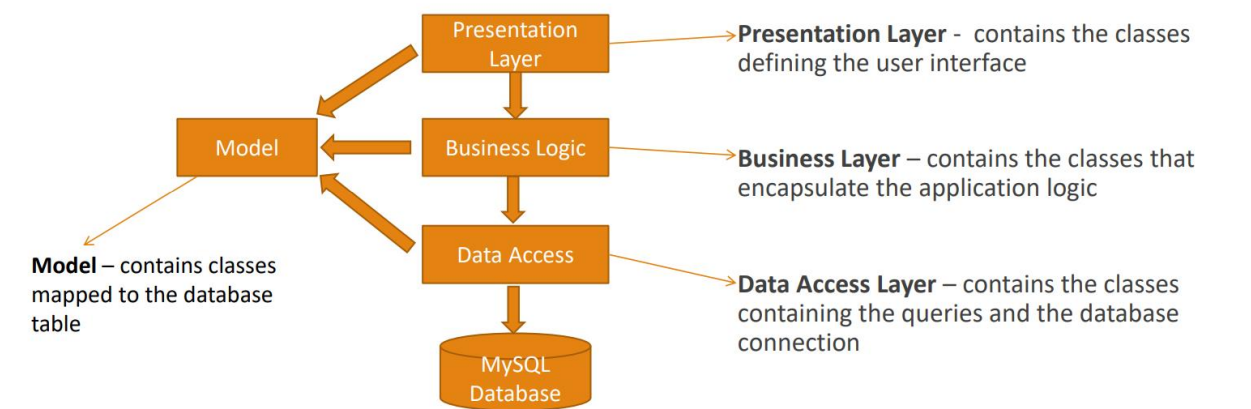
## **2. Modeling, Scenarios, Use Cases**

The goal of this project is to develop a user-friendly application that allows users to perform CRUD operations (Create, Read, Update, Delete) on orders, clients, and products. All data is stored in a relational SQLite database, which provides easy access and retrieval of information across different computers.

To facilitate understanding, let's clarify the concept of CRUD. CRUD represents the four essential functions needed to implement a persistent storage application: create, read, update, and delete. Persistent storage refers to storage devices that retain data even when power is lost, such as hard disks or solid-state drives. On the other hand, volatile memory, like RAM, loses its data when power is lost.

Instead of connecting to a separate database server, this project utilizes SQLite, an embedded SQL database engine. Unlike other SQL databases, SQLite directly reads and writes data to disk files. The entire SQL database, including tables, indices, triggers, and views, is contained in a single file. SQLite's file format is cross-platform, allowing easy database transfer between different systems. SQLite is widely used as an application file format and is recommended by the US Library of Congress. It serves as a replacement for file I/O operations (like `fopen()`) rather than a full-scale database server like Oracle.

In the application, the user is presented with three branches corresponding to the three tables: clients, products, and orders. The user can choose a branch and perform CRUD operations within that table. The graphical user interface (GUI) provides a straightforward and intuitive experience for users. It is important to note that this project deviates from the traditional Model-View-Controller (MVC) architecture and instead follows a Layered Architecture approach to design and implementation.



	productid	productname	productprice	productstock
1	1	1	1	1
2	2	2	2	2
3	4	4	4	4
4	6	6	6	6
5	3	5	5	5

	orderid	quantity	clientid	productid
1	2	2	2	1
2	1	1	1	1
3	5	5	5	5
4	2	6	2	2
5	3	7	3	3

### 3. Design

*The following should be presented: OOP design of the application, UML package and class diagrams, used data structures, defined interfaces and used algorithms (if it is the case).*

Upon launching the application, users are presented with a main window that offers different options to proceed. Each button corresponds to a specific functionality and opens a separate window. To return to the main menu, users need to close all open windows except the main one.

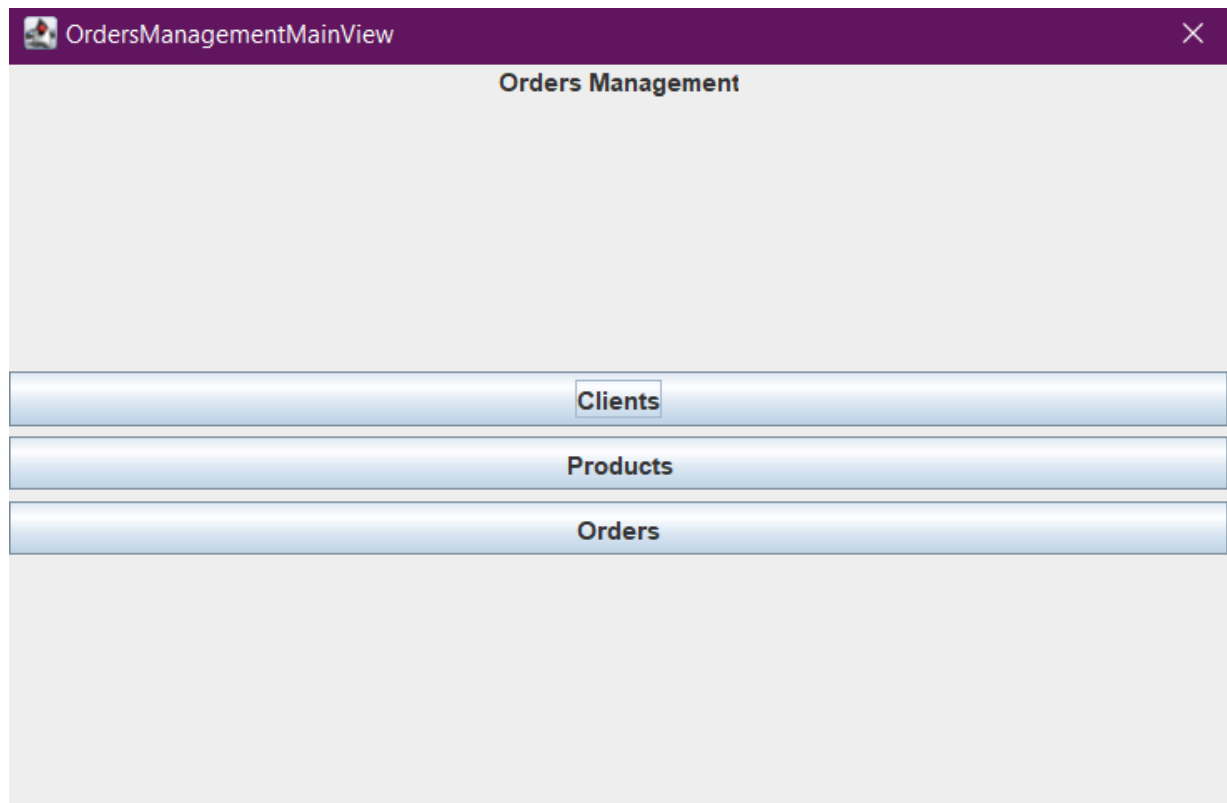
The branches for Clients and Products share a similar structure and allow users to perform CRUD operations for managing client and product data respectively.

When adding a new entry, there are specific fields to be filled. For clients, the required information includes the ID, name, home address, email, and age. On the other hand, for products, users need to provide the ID, name, price, and stock.

During the editing process for clients, products, or orders, the application first verifies the existence of an entity with the given ID. If no matching entity is found, an error message is displayed. Similarly, when deleting an entity, users must specify the corresponding ID.

To display all rows of data in the database tables for clients, products, or orders, the application utilizes reflection techniques to create a JTable. This table allows users to manipulate the display by rearranging columns and rows according to their preferences.

The JTable class, a component of the Java Swing Package, is specifically designed for presenting and editing two-dimensional data in a tabular form. It resembles a spreadsheet, providing a structured layout for organized data representation.



ClientView

×

Clients

Add Client

Delete Client

Edit Client

Show all Clients

Back

AddClient

×

Add Client

Client ID

Name

Age


Address

E-mail

Add Client

Back




 DeleteClient ✕

Delete Client

Client ID

Delete Client

Back

 EditClient ✕

Edit Client

Client ID

Name

Age

Address

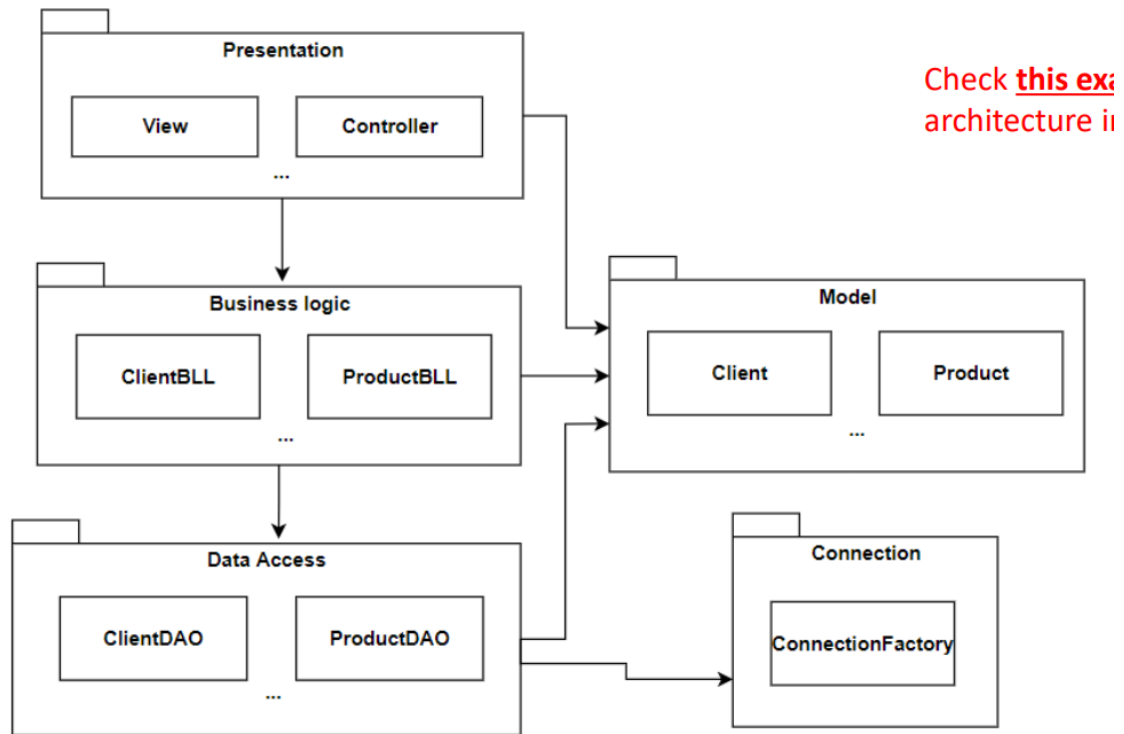
E-mail

Edit Client

Back

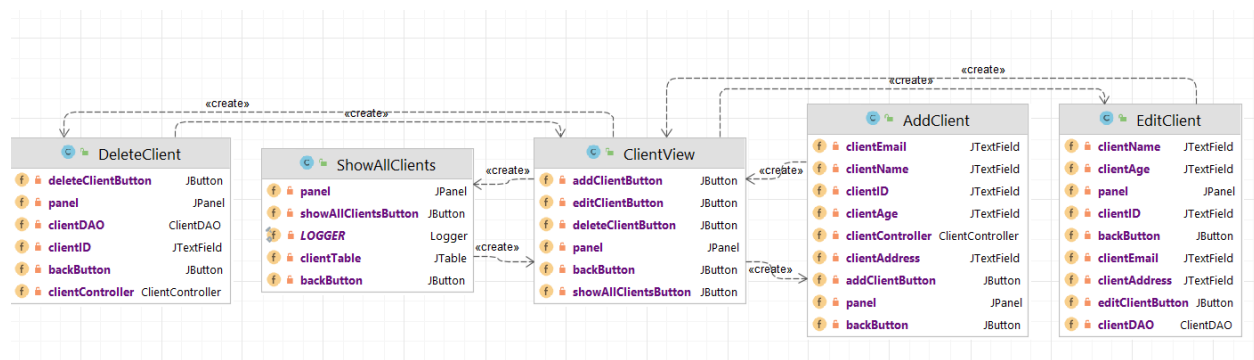
ShowAllClients				
All Clients				
Show all clients				
0	John Doe	0		johndoe@example.c...
0	John Doe	0		johndoe@example.c...
0	4	4	4	4
0	6	6	6	6
0	7	7	7	7
9	9	9	9	9
1	updated	5	updated	updated
15	15	15	15	15
Back				

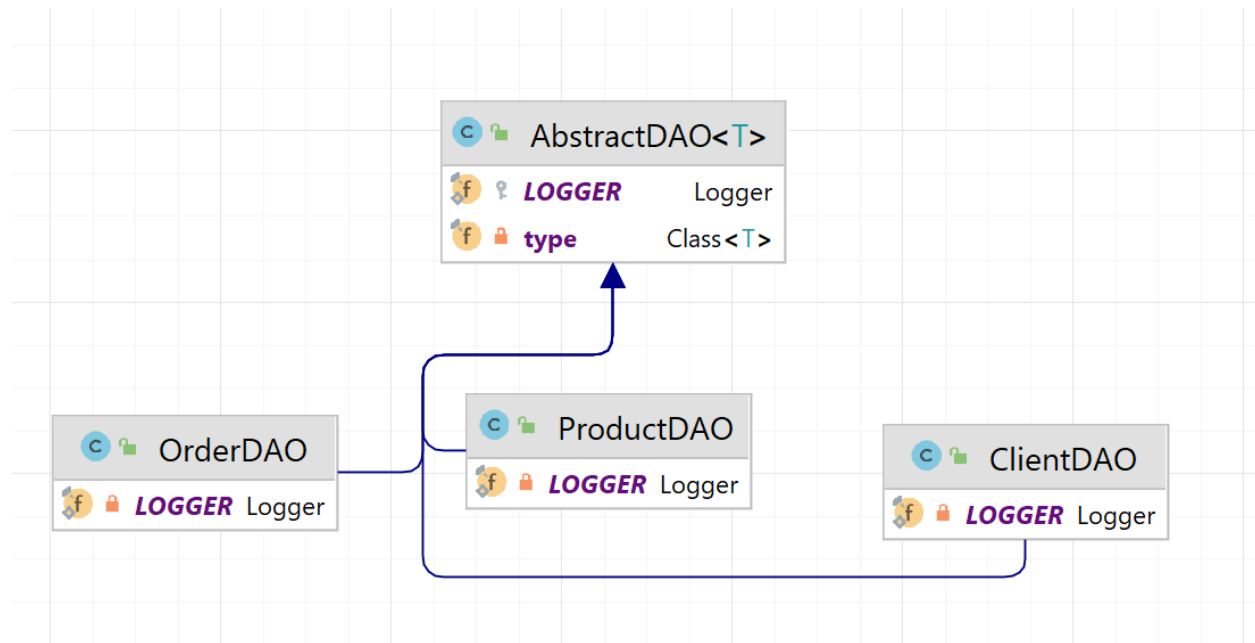
And the same for orders and products.



During the project development, various diagrams were created to aid in the design process. One of these diagrams is the UML (Unified Modeling Language) diagram, which is used to visualize, specify, and document software systems. The class diagram, in particular, represents the Java classes utilized in the project implementation, along with their dependencies and relationships.

The class diagram provides an overview of the project's structure, showcasing how different classes interact and collaborate with each other. It helps in understanding the organization of the codebase and facilitates communication between team members working on the project. By visualizing the classes and their relationships, the class diagram serves as a blueprint for the software system's design and implementation.





In software development, a Data Access Object (DAO) is a design pattern that provides an abstract interface for accessing and manipulating data in a database or any other persistence mechanism. The purpose of the DAO pattern is to encapsulate the details of data access and provide a standardized way to perform common data operations without exposing the underlying database implementation.

In this project, each model class (such as clients, products, and orders) has its own corresponding DAO class. These DAO classes are responsible for handling data operations specific to their respective models. They encapsulate the CRUD methods (Create, Read, Update, Delete) that interact with the database using SQL queries. While the DAO classes share similarities in structure and functionality, they differ in the specific SQL queries and variables they work with.

The DAO pattern promotes the principle of single responsibility by separating the data access needs of the application (expressed through the public interface of the DAO) from the implementation details related to the specific database management system (DBMS) and database schema. This isolation allows for flexibility and ease of maintenance, as changes to the underlying data storage can be handled within the DAO implementation without impacting the rest of the application.

## 4. Further improvements

*Each class will be described (fields, important methods). Also, the implementation of the graphical user interface will be described.*

In addition to the project requirements, there are some areas that could be further developed to enhance the application:

**Abstract/Generic DAO:** It would be beneficial to create an abstract or generic DAO class that can be used as a foundation for the model-specific DAOs. This abstract DAO can provide common functionality and reduce the risk of query errors by centralizing the database access logic. By reflecting this abstract class, it becomes easier to create consistent and reliable DAOs for each model.

**Improved GUI Design:** Instead of having separate windows for each button or action, it would be more user-friendly to have a single window that dynamically updates based on user interactions. This approach reduces window clutter and provides a smoother user experience. Additionally, considering a shift from Swing to JavaFX can give the application a more professional and visually appealing appearance.

**MVC Architecture:** To further enhance the application's design and maintainability, implementing an MVC (Model-View-Controller) architecture on top of the existing layered architecture would be advantageous. The MVC pattern helps separate concerns by defining clear roles for the model, view, and controller components. This separation promotes code organization, modularity, and testability.

By incorporating these enhancements, the project can deliver a more robust and user-friendly application, with improved code structure and maintainability.

## 5. Results

Results can be seen in the DataGrip tables.

## 6. Conclusions

. This project served as an excellent opportunity to revisit and reinforce the object-oriented programming (OOP) concepts learned in the first semester, while also expanding my knowledge by exploring new concepts. It was both challenging and rewarding to delve into topics such as reflection and database connectivity in Java, as they presented fresh learning experiences.

One particular aspect that proved to be quite challenging was working with the `ResultSet` to create queries. Initially, understanding the `ResultSet` and its appropriate methods took considerable effort, and I encountered difficulties in grasping its functionality. However, through perseverance and dedicated research, I gradually gained a thorough understanding of how to utilize `ResultSet` effectively and interpret the data it provides.

Engaging with and resolving coding problems independently offers significant benefits. It not only fosters a deeper comprehension of new concepts but also encourages the utilization of existing knowledge. During the course of this project, I encountered various obstacles and utilized research and problem-solving skills to overcome them. Ultimately, this approach facilitated my growth as a programmer and expanded my proficiency in handling complex code challenges.

## 7. Bibliography

<https://dsrl.eu/courses/pt/>

<https://www.youtube.com/>

<https://www.geeksforgeeks.org/>

<https://chat.openai.com/>

