

DOCUMENTATION

ASSIGNMENT No. 3

STUDENT: Mureșan Salomeea
GROUP: 30424

CONTENTS

1.	Assignment objective.....	Error! Bookmark not defined.
2.	Problem analysis, modeling, scenarios, use cases.....	3
3.	Design	Error! Bookmark not defined.
4.	Implementation	7
5.	Results.....	9
6.	Conclusion	Error! Bookmark not defined.
7.	Bibliography	11

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 (in this case: Client, Product and Orders classes, grouped in the model package)
- Business Logic Classes – they contain the application logic (in this case: ClientBLL, ProductBLL, OrdersBLL, grouped in the businessLogic package)
- Presentation classes – GUI related classes (in this case: View, in the presentation package)
- Data access classes – classes that contain the access to the data base (in this case: AbstractDAO, ClientDAO, ProductDAO, OrderDAO, grouped in the dataAccess package; ConnectionFactory class, in connection package)

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

- a. Analyze the application domain, determine the structure and behavior of its classes and draw an UML class diagram.
- b. Implement the application classes (Model classes, Business Logic classes, Presentation classes, Data Access classes). Also use Javadoc for documenting the classes.
- c. Use reflection techniques to create a method that receives a list of objects and generates the header (column names) of the table by extracting through reflection the object properties and then populates the table with the values of the elements from the list.

2. Problem analysis, modeling, scenarios, use cases

- General overview:

The application should be able to fulfill all the requirements in order to display, modify and keep track of orders, clients and products. All the data is stored in a relational MySQL database that contains 3 tables: Client, Product and Orders. Only one user has access to this application.

For example, the user can add a new client in the Client data table by inserting in the designated text fields the name and email address of the new client. There is no need for the user to set an ID for the new client, because the program will add an automatically incremented ID (the ID of the last client in the data base, incremented with '1').

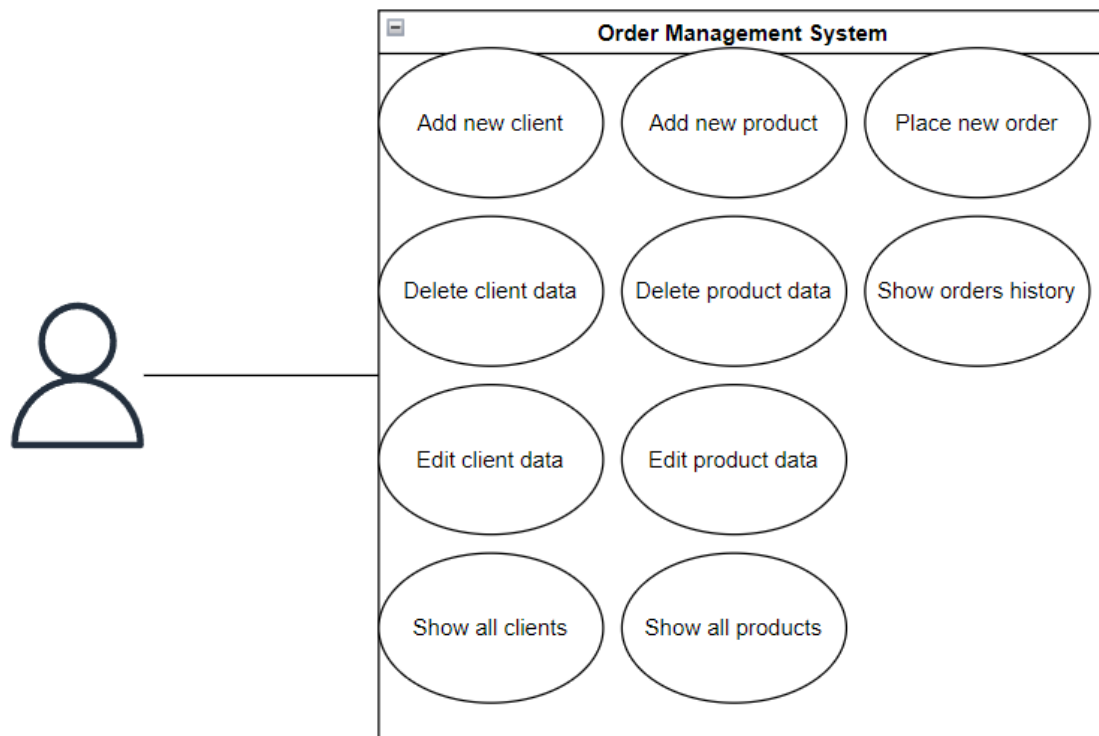
- Input and Output

When speaking about the input for the application, the one user has the choice to manage the three available tables from the data base: Client, Product and Orders. All the tables have the options defined as CRUD operations:

1. (C) create (insertion query: the user can add a new client, a new product or place a new order)
2. (R) read (select query: the user can retrieve, search or view already existing entries in a table)
3. (U) update (the user can edit or update existing entries in the data base)
4. (D) delete (the user can delete, deactivate or remove already existing entries in the data base)

The user can introduce in the specific text boxes (JTextBox) the values for new clients or new products and it can also select from 2 different combo boxes (JComboBox) the client id for the client that wants to place an order and the product id for which product the client wants to buy. All fields have to respect the standard rules of formatting, for example the new client's name has to contain only characters, the new client's name has to respect the specific pattern [name] @ [host name] . [suffix], the price for a new product has to be an integer or a float variable (eg. 10 or 93.2).

- Use case diagram



a. Add new client

The user can register a new client to the database by introducing the client details:

- Client name
- Client email address

After inserting the data in the correct text fields, the user has to click on the “ADD NEW CLIENT” button. If the name or email address do not conform to the formatting rules of the specific fields (name and email address), a message dialog will pop up, asking the user to correct the input data.

Actors: user.

b. Delete client data

The user can delete an already existing client from the database by selecting the row from a combo box representing the wanted client’s data and then clicking on the “DELETE CLIENT” button. If no row from the table is selected, a message dialog will pop up, asking the user to select a row from the clients combo box.

Actor: user.

c. Edit client data

The user can modify in the clients table the data of one or more clients. The user has to choose from a combo box the client’s data which he or she wants to update. After choosing a row from the combo box, the user has to complete one or more text fields representing the new name or new email that the client has.

Actor: user.

d. Show all clients

The user can see all the entries, meaning all clients saved in the data base with their data, by clicking on the “SHOW CLIENTS LIST” button. After doing so, the clients data table will appear in a pop up frame, showing all the entries existent in the data base.

Actor: user.

e. Add new product

The user can register a new client to the database by introducing the product details:

- Product name
- Price per unit for the new product
- Number of items in stock for the new product

After inserting the data in the correct text fields, the user has to click on the “ADD NEW PRODUCT” button. If the name, price per unit or number of items in stock do not respect the formatting necessary, a message dialog will pop up, asking the user to correct the input data.

Actors: user.

f. Delete product data

The user can delete an already existing product from the database by selecting the row representing the wanted product’s data and then clicking on the “DELETE PRODUCT” button. If no row from the table is selected, a message dialog will pop up, asking the user to select a row from the products table.

Actor: user.

g. Edit product data

The user can modify in the clients table the data of one or more products. After doing so, the user has to click on the “EDIT PRODUCT DATA” in order for the updates to be saved in the data base.

Actor: user.

h. Show all products

The user can see all the entries, meaning all products saved in the data base with their data, by clicking on the “SHOW PRODUCTS LIST” button. After doing so, the product data table will refresh and show all the existing entries.

Actor: user.

i. Place new order

The user selects from two combo boxes the client that wants to place an order and the product which the client wants to purchase. In the available text field, the user has to give the number of items in stock for the new product. After doing so, the user has to click on the “PLACE ORDER” button to save the new entry.

Actor: user.

j. Show orders history

The user can see all the entries, meaning all orders saved in the data base with their data, by clicking on the “SHOW ORDERS HISTORY” button. After doing so, the orders data table will refresh and show all the existing entries.

Actor: user.

3. Design

3.1. Database schema

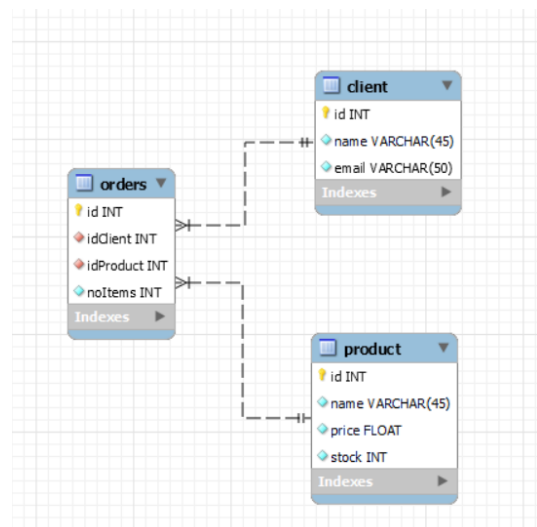
The data base used for this application consists of three tables:

Client, orders and product.

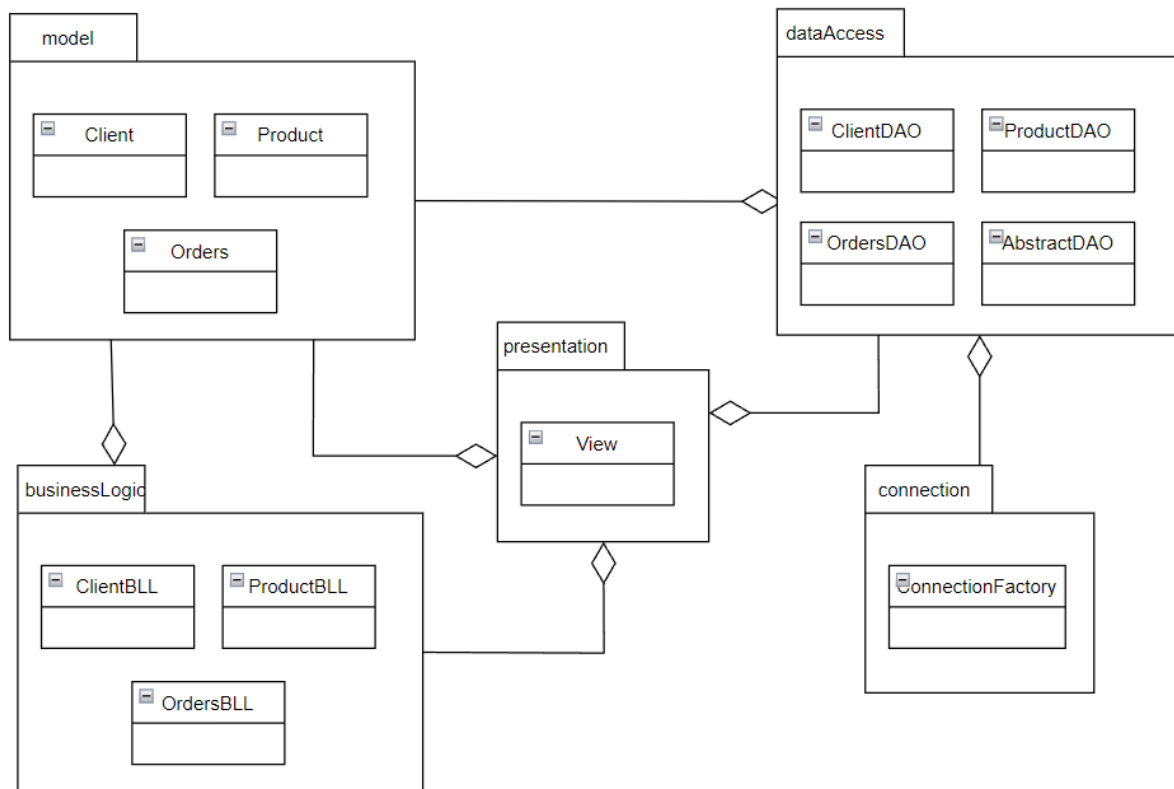
Relations between tables:

Client -> Orders: one to many (a client can place one or more orders)

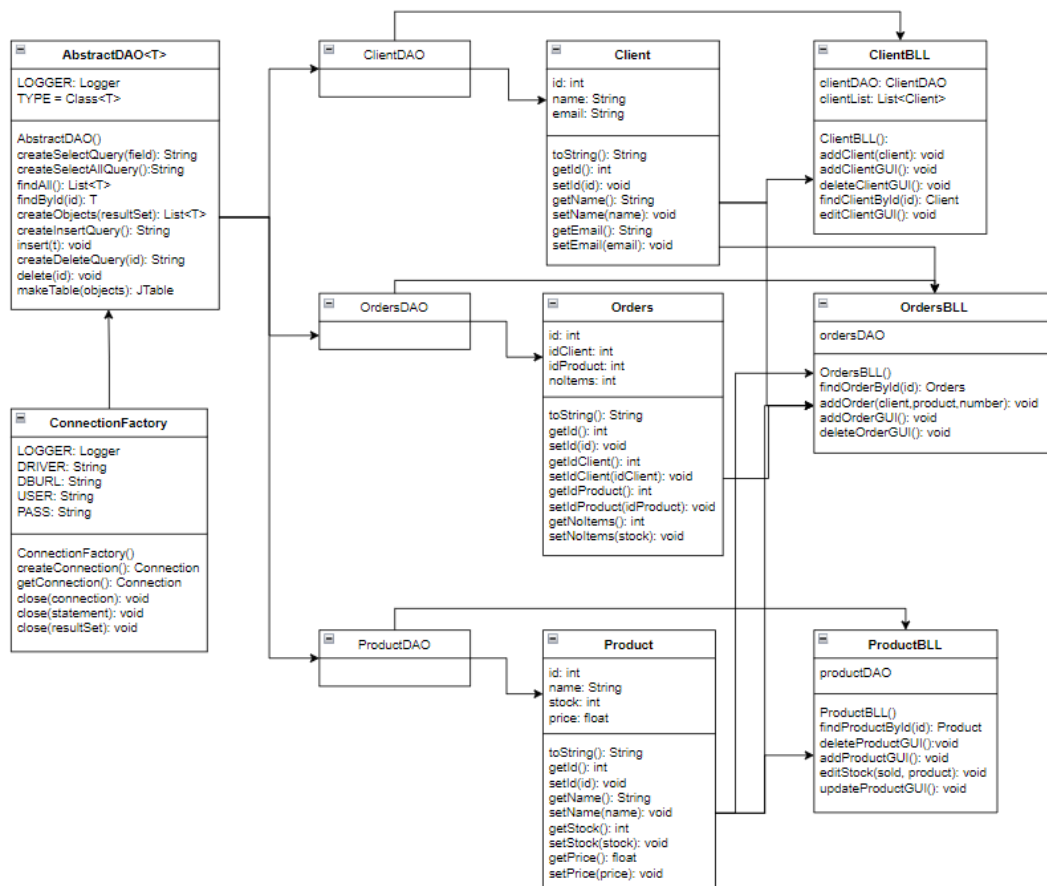
Product -> Orders: one to many (more orders can contain the same product).



3.2. Package diagram



3.3. Class diagram



4. Implementation

In this part of the documentation, each class will be discussed and have its functionalities presented.

4.1. View

View is responsible for arranging the window/s that will be seen by the user, essentially designing the layout of the application. In order to create a user friendly interface, I used Swing UI Designer.

UI Designer in IntelliJ IDEA enables the user to create graphical user interfaces (GUI) for his or her applications using Swing library components. The tool helps speeding up the most frequent tasks: creating dialogs and groups of controls to be used in a top-level container such as a JFrame. When designing a form with the GUI Designer, the user creates a pane rather than a frame. Dialogs and groups of control created with the GUI Designer can coexist in the application with the components that are created directly with Java code.

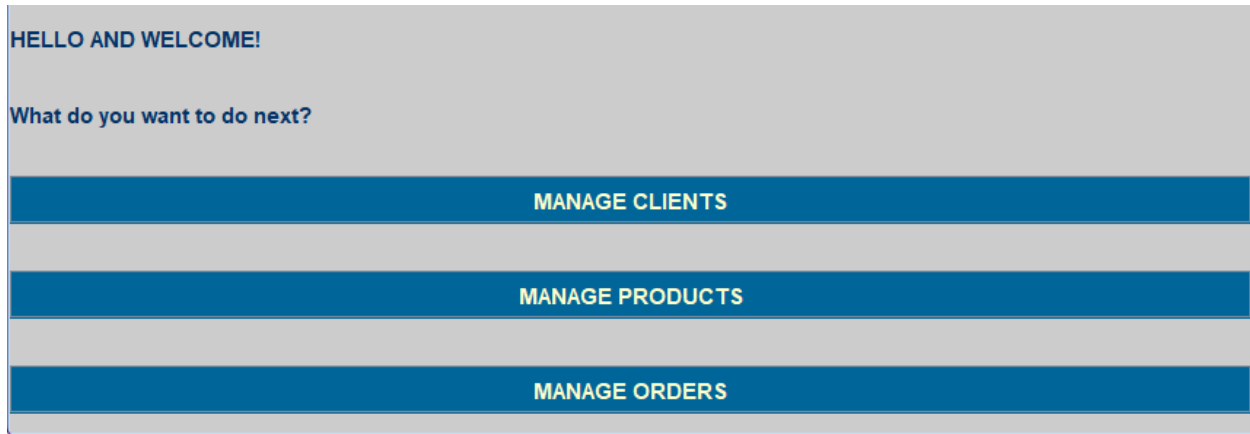
Using the GUI Designer is subject to the following limitations:

- GUI Designer does not support modeling of non-Swing components.
- The GUI Designer does not create a main frame for an application, nor does it create menus.

By default, IntelliJ IDEA automatically creates a Java class at the same time it creates a new GUI form. The new form automatically binds to the new class via the bind to class property. When components are added to the design

form, a field for each component is automatically inserted into the source file of the Form's class. The field name appears in the relevant component's field name property, thereby binding the component to code in the class. It only remains to augment the generated code with whatever additional code is needed to implement behaviors and functionality to the form.

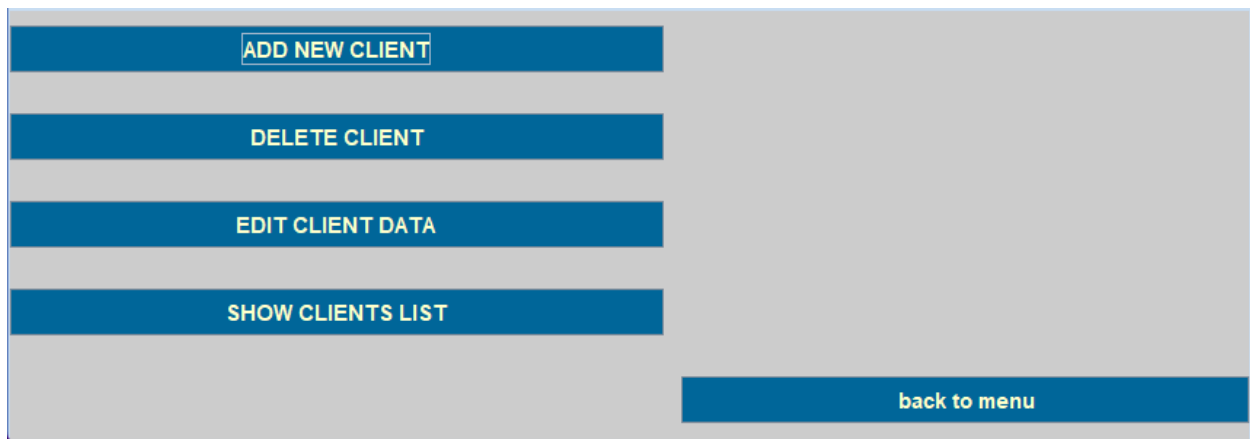
When first running the application, the user will see a menu:



A screenshot of a Java Swing window titled "HELLO AND WELCOME!". Below the title bar, the text "What do you want to do next?" is displayed. The window contains three blue buttons with white text, stacked vertically: "MANAGE CLIENTS", "MANAGE PRODUCTS", and "MANAGE ORDERS". The buttons are separated by light gray horizontal bars.

The user has three options: he or she can perform operations on the clients data table, on the products data table or on the orders data table. After pressing any of those three buttons, the user will be redirected to another window where operations can be performed.

When choosing to operate on the clients data table, the user has the following options:



A screenshot of a Java Swing window titled "ADD NEW CLIENT". The window contains four blue buttons with white text, stacked vertically: "DELETE CLIENT", "EDIT CLIENT DATA", and "SHOW CLIENTS LIST". The buttons are separated by light gray horizontal bars. At the bottom right of the window, there is a blue button with white text labeled "back to menu".

All operations are done using JOptionPanes. Whenever a button is pressed (except the ones from the menu), a new message dialog will appear, making the whole navigation through the application much easier. Below you can see some examples for these JOptionPanes used for the implementation of my order management program.

4.2. Client

In order to extract elements from the data base table, a special class named entity must be created. This class must have the fields exactly the same type as the columns from the corresponding table. The class must have also constructors, getters and setters.

Column Name	Datatype
id	INT
name	VARCHAR(45)
email	VARCHAR(50)

```
public class Client {
    private int id;
    private String name;
    private String email;
    ...
}
```

There are no other methods implemented in this class.

4.3. Product

As the Client class, the Product class has the exact same fields as the product data table in the designed data base, and the only methods implemented are the constructor, the setters and the getters.

Column Name	Datatype
id	INT
name	VARCHAR(45)
price	FLOAT
stock	INT

```
public class Product {
    private int id;
    private String name;
    private int stock;
    private float price;
    ...
}
```

4.4. Orders

As the Client class, the Orders class has the exact same fields as the product data table in the designed data base, and the only methods implemented are the constructor, the setters and the getters.

Column Name	Datatype
id	INT
idClient	INT
idProduct	INT
noItems	INT

```
public class Orders {  
    private int id;  
    private int idClient;  
    private int idProduct;  
    private int noItems;  
    ...  
}
```

4.5. AbstractDAO

AbstractDAO is a class that defines the common operations for accessing a table: Insert, Update, Delete, FindById, FindAll. Define the operations on the specified generic type (! T can be any Java Model Class that is mapped to the Database, and has the same name as the table and the same instance variables and data types as the table fields).

```
public class AbstractDAO<T> {  
  
    protected static final Logger LOGGER =  
        Logger.getLogger(AbstractDAO.class.getName());  
    private Class<T> type;  
  
    public AbstractDAO() {  
        this.type = (Class<T>) ((ParameterizedType)  
            getClass().getGenericSuperclass()).getActualTypeArguments()[0];  
    }  
}
```

In this class I implemented all the queries necessary for the well execution of the program:

- Insert

```
private String createInsertQuery() {  
    StringBuilder sb = new StringBuilder();  
    sb.append("INSERT INTO ");  
    sb.append(type.getSimpleName().toLowerCase());  
    sb.append(" VALUES ( ");  
    return sb.toString();  
}
```

- Update

The update query is implemented by saving the wanted product or client in an auxiliary object, setting it's fields to the new values for name, email, price etc., deleting the old product from the data base and inserting it with the same id as before (this is important, so the id doesn't change if for example there is a sale and a product will be cheaper).

- Delete

```
public String createDeleteQuery(int id) {  
    StringBuilder sb = new StringBuilder();  
    sb.append("DELETE FROM ");  
    sb.append(type.getSimpleName().toLowerCase());  
    sb.append(" WHERE id = ");  
    sb.append(id);  
}
```

```
        return sb.toString();  
    }
```

The deletion of a row of data from a table is made by giving as parameter to the method the id of the object that has to be removed. The client will select the data to be deleted from a combo box, so the user won't be able to try to delete a non-existing client or product.

- Select

For the select query, I implemented a select all query and a select with options query.

```
private @NotNull String createSelectAllQuery() {  
    StringBuilder sb = new StringBuilder();  
    sb.append("SELECT ");  
    sb.append(" * ");  
    sb.append("FROM ");  
    sb.append(type.getSimpleName().toLowerCase());  
    return sb.toString();  
}
```

The select all query is used for populating the tables for when the user wants to check the warehouse records in this program.

4.6. ClientDAO

This class extends AbstractDAO <Client>.

4.7. ProductDAO

This class extends AbstractDAO <Product>.

4.8. OrdersDAO

This class extends AbstractDAO <Orders>.

4.9. ClientBLL

In this class I implemented methods that will be called in the View class.

4.10. ProductBLL

4.11. OrdersBLL

4.12. ConnectionFactory

5. Bibliography

- https://gitlab.com/utcn_dsrl/pt-layered-architecture
- https://gitlab.com/utcn_dsrl/pt-reflection-example
- <https://www.jetbrains.com/help/idea/designing-gui-major-steps.html>
- <https://dsrl.eu/courses/pt/>
- <https://www.baeldung.com/javadoc>
- <https://stackoverflow.com/>
- <https://www.w3schools.com/>

