PROGRAMMING TECHNIQUES

ORDERS MANAGEMENT

STUDENT: ELEKES PÉTER
UNIVERSITY: UNIVERSITY OF TECHNICAL ENGINEERING CLUJ-NAPOCA
FACULTY: COMPUTER SCIENCE
YEAR: II.
GROUP: 30422

Table of Contents

Assignment Task

Design and implement an application for managing the 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, business logic, presentation, and data access.

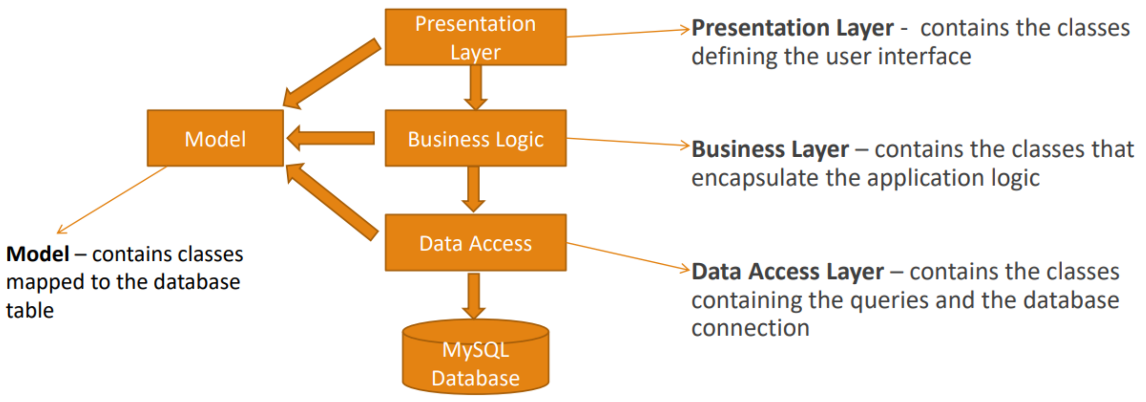
Secondary objectives are to analyze the problem and identify requirements, design the orders management application, implement the orders management application, and test the orders management application.

The project should use an object-oriented design (with encapsulation, inheritance, decomposition), should use lists instead of arrays, use foreach instead of for, have a GUI using JavaFX or Swing and every class should have less than 300 lines of code.

Requirements	**Points**
Object-oriented design	minimal requirement
Use javadoc for documenting classes and generate the corresponding JavaDoc files	minimal requirement
Use relational databases for storing the data for the application, minimum three tables: Client, Product and Order	minimal requirement
Create a graphical user interface including: o A window for client operations: add new client, edit client, delete client, view all clients in a table (JTable) o A window for product operations: add new product, edit product, delete product, view all product in a table (JTable) o A window for creating product orders - the user will be able to select an existing product, select an existing client, and insert a desired quantity for the product to create a valid order. In case there are not enough products, an under-stock message will be displayed. After the order is finalized, the product stock is decremented.	minimal requirement
Use reflection techniques to create a method that receives a list of objects	minimal requirement

and generates the header of the table by extracting through reflection the object properties and then populates the table with the values of the elements from the list.	
Layered Architecture (the application will contain at least four packages: dataAccessLayer, businessLayer, model and presentation)	2 points
Create a bill for each order as a text file or .pdf file	1 point
Use reflection techniques to create a generic class that contains the methods for accessing the DB: create object, edit object, delete object and find object. The queries for accessing the DB for a specific object that corresponds to a table will be generated dynamically through reflection.	2 points

Assignment analysis, scenario, approach, and use cases


Presentation Layer
Business Logic
Data Access
MySQL Database
Model
Presentation Layer - contains the classes defining the user interface
Business Layer – contains the classes that encapsulate the application logic
Data Access Layer – contains the classes containing the queries and the database connection
Model – contains classes mapped to the database table

I used a layered architecture to ease the workflow and to have a logical overview of the problem. Using this approach, we can simply create a Model class, which describes every table of the database, where we can map the interaction with the database in our view. The Presentation Layer is used to create the user interface, it contains and describes the way a user can interact with the application. The Business Layer contains the necessary logic of the program, and the Data Access Layer is used to forward the queries from the program to the database

.

Use Case

The application can be used to: add a new client, add a new product, or add a new order.

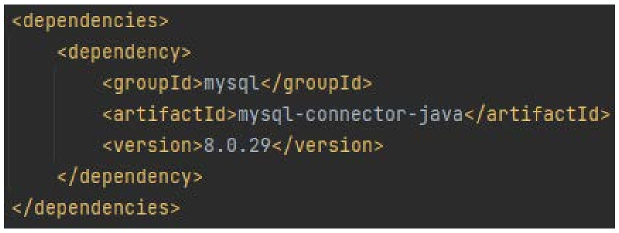
A general scenario:

1) The user selects the option to:
   a) Open Client Menu
   b) Open Product Menu
   c) Open Order Menu
2) The user inputs the necessary information to the fields
   a) If a new entry should be added, then the user should complete all the fields (e.g.: id, name, address, email in the case of a new customer) and the insert button should be pressed
   b) If an existing entry needs to be deleted, the only necessary information given is the ID
   c) If an existing entry needs to be updated, all the fields should be completed and then the update button should be pressed.
3) If everything was done correctly a success message is shown
4) If the user wants to check the database, the 'Show Table' button should be pressed

Projection

I approached the problem using a layered architecture pattern and used classes that were grouped into the following packages: BLL (Business Logic), Connection (Establishing connection to the database), DAO (Data Access), Model (Classes describing the tables of the database), Presentation (The GUI elements) and Start (The main function). This solution is an efficient approach on creating an application that is logically constructed and easy to improve.

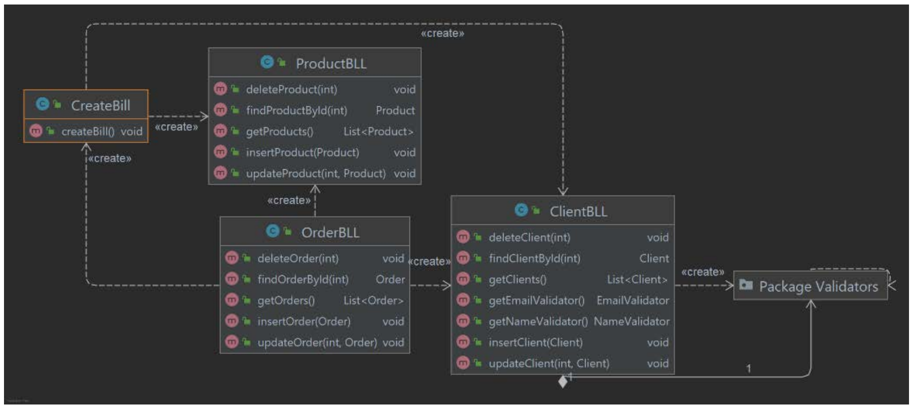
As for the database, I used MySQL, which was connected to the application using a Maven dependency:



```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.29</version>
    </dependency>
</dependencies>
```

Implementation

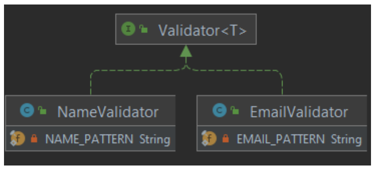
BLL (Business Logic)


«create»
ProductBLL
deleteProduct(int) void
findProductById(int) Product
getProducts() List<Product>
insertProduct(Product) void
updateProduct(int, Product) void
CreateBill
createBill() void
«create»
«create»
«create»
OrderBLL
deleteOrder(int) void
findOrderById(int) Order
getOrders() List<Order>
insertOrder(Order) void
updateOrder(int, Order) void
«create»
ClientBLL
deleteClient(int) void
findClientById(int) Client
getClients() List<Client>
getEmailValidator() EmailValidator
getNameValidator() NameValidator
insertClient(Client) void
updateClient(int, Client) void
«create»
Package Validators
1

The main classes implemented in this package are CreateBill, ProductBLL, OrderBLL, ClientBLL, a Validator interface, and two validators, namely an EmailValidator and a NameValidator.

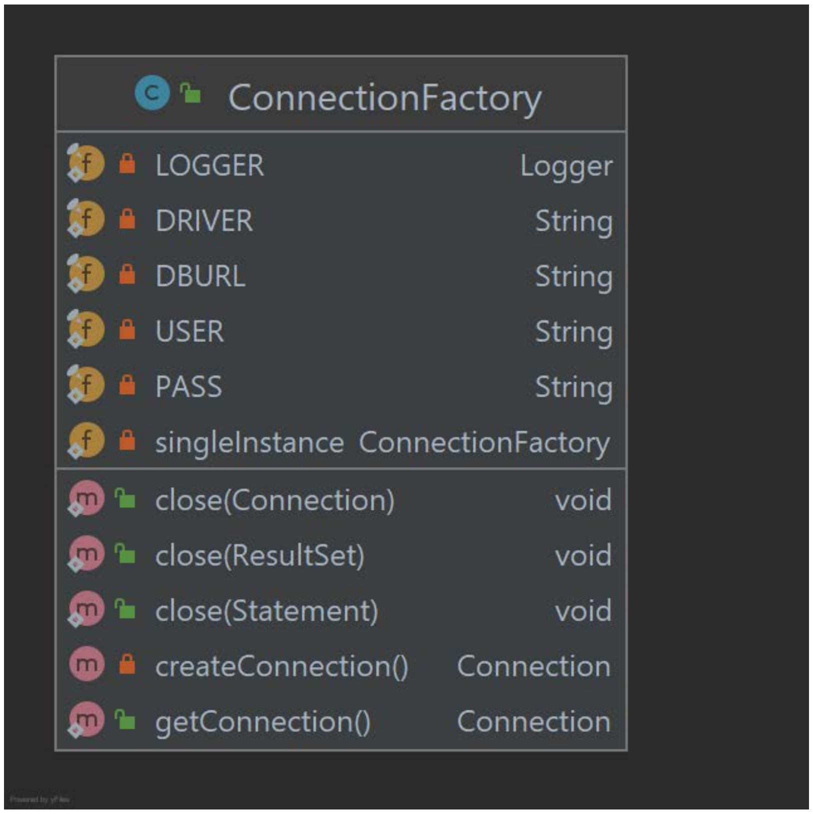
The CreateBill class has an Order attribute, and its constructor requires an Order as a parameter. It contains a single method, having the name createBill, which creates a new .txt file in the bills/ folder. Information written in this file is: the bill id (Order id); customer name, address, email; product name, price, quantity ordered; and a total price.

The ClientBLL, ProductBLL, and OrderBLL classes all include and implement the necessary methods used by the GUI, like adding a new client, finding a client by id, etc…



The Validator sub package includes a Validator Interface, and two Validator classes that implement the interface. Both of these classes use RegEx to validate a given objects' attribute. If the pattern doesn't match they throw an IllegalArgumentException.

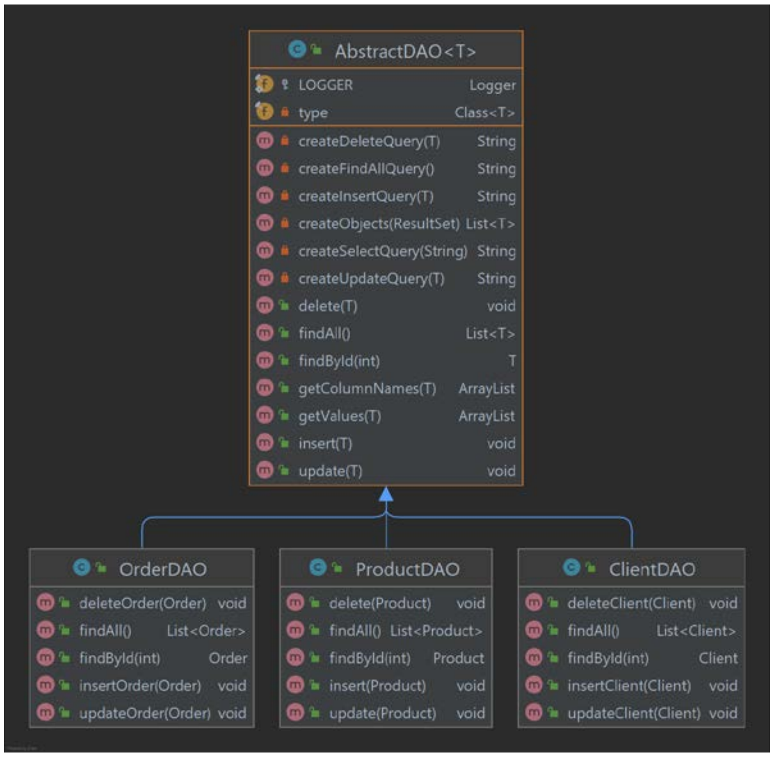
Connection



There is a single class in this package, the ConnectionFactory class. This class was provided to us by the problem statement, but in essence, this class is used to establish a connection to the given database. It has as its arguments a LOGGER, DRIVER, DATABASE URL, USER, AND PASSWORD, all of these needed by the program to establish a connection to the database.

This class includes a createConnection method which calls the DriverManager.getConnection() method with the database url, user, and password parameters. This method is private in order to establish a secure connection, and preserve the encapsulation principle of Object Oriented Programming. If no such connection could be made then an exception is thrown, otherwise the method returns the connection. Similarly, the createConnection method, which is public, returns the created connection. There is an overloaded close method, having 3 different variants with different parameters. All of these parameters specify an object that has to be closed in relation to thedatabase.

DAO (Data Access)



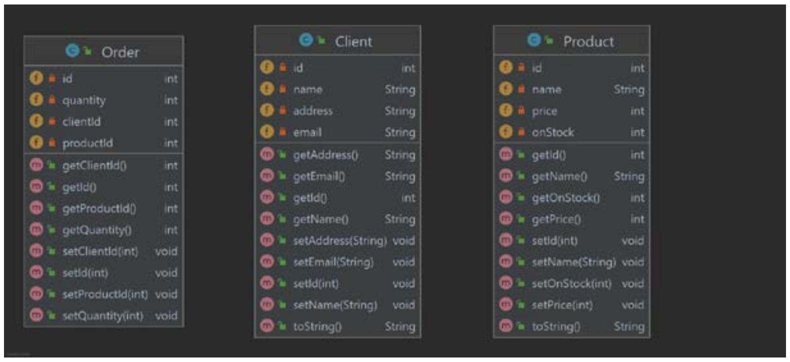
This package is the bread and butter of the whole application. A reflection technique is used to construct and Abstract class that is a general description of how data should be read from the database.

The AbstractDAO class has many useful methods to interact with the database, using the SQL Language, and queries. An interesting roadblock that I hit during production is that when accessing any table of the database I had to specify the name of the database too. I think this error occurred because the name of my database is 'order_management' and I have a table called 'order'. Whenever I wanted to access the Order table it threw a SQL Syntax error, stating that no such table exists. I overcame the problem by making every query implicitly describe the name of the database, e.g.: INSERT INTO \`order_management\`.\`order\`.

Each of the classes extending the AbstractDAO are calling its' methods, such as insertClient, which calls the super.insert method.

This reflection technique works this way: the generic AbstractDAO class contains the methods for accessing the database (creates the object, edits the object, deletes the object, or finds the obejct). All of the queries for accessing a specific object that corresponds to a table are generated dynamically through reflection.

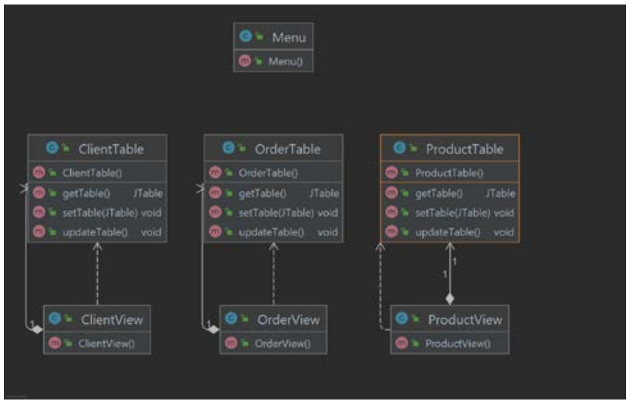
Model



In the Model package there are 3 classes, which correspond to the 3 tables in the database. All of the attributes of this class match perfectly the attributes of the corresponding table. This representation is necessary in order to establish a relationship between the program and the database.

All of these classes have empty, default constructors and constructors which takes as parameters all the attributes of the class. Setter and getter methods are also implemented in order to preserve the Object Oriented approach of programming, and encapsulation.

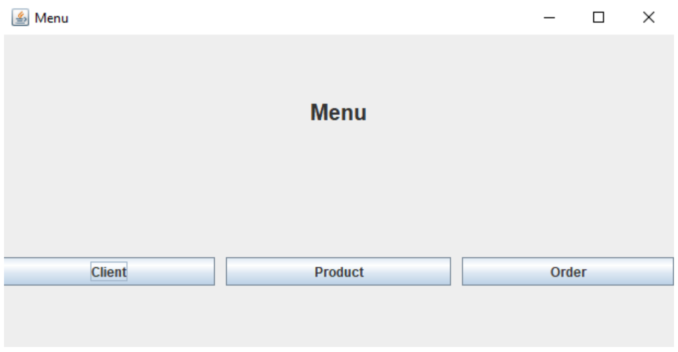
Presentation (GUI)

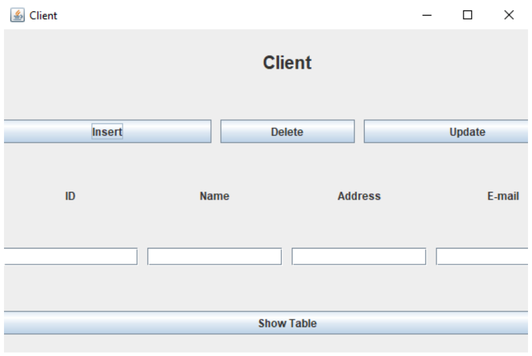


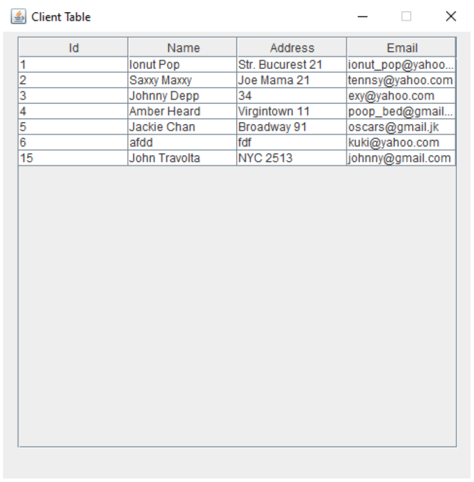
One of the most necessary packages, the Presentation packages includes all the classes which implement the GUI.

The Menu class is a simple view, presenting the user with 3 buttons: Client, Product, and Order.

If any of these buttons are pressed a new window pops up, where the user can provide necessary information to interact with the database.






Client Table

Id	Name	Address	Email
1	Ionut Pop	Str. Bucurest 21	ionut_pop@yahoo...
2	Saxxy Maxxy	Joe Mama 21	tennsy@yahoo.com
3	Johnny Depp	34	exy@yahoo.com
4	Amber Heard	Virgintown 11	poop_bed@gmail...
5	Jackie Chan	Broadway 91	oscars@gmail.jk
6	afdd	fdf	kuki@yahoo.com
15	John Travolta	NYC 2513	johnny@gmail.com

The use case was already explained beforehand, but the usage of the program is intuitive. Any action performed by the user has validation messages, ensuring that anything that's happening in the program is transparent.

The Show Table button opens a new window, with a JTable in it, showing all the data in the database, from the table. This table automatically updates should any action be performed.

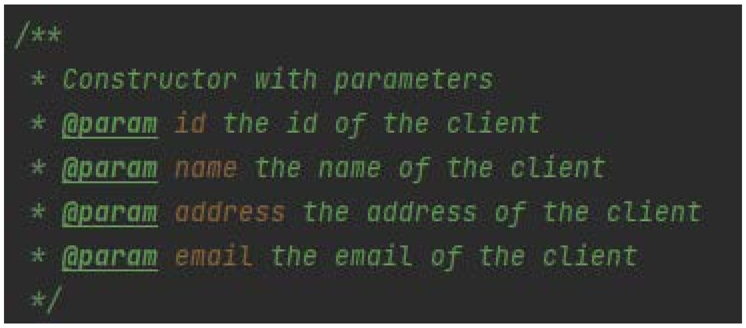
The implementation of these views consists of a class for each view: there are Menu, ClientView, ProductView, and OrderView classes. Every table has its class aswell, ensuring that the Client, Product, and Order views all have their corresponding tables.

In the View classes all the ActionListeners are implemented, and whenever a button is pressed all the textfields are validated, not allowing any wrong information to be provided.

Start

This package is quite unremarkable, it simply has a Start class implementing the main function, which initializes a new Menu window.

JavaDocs



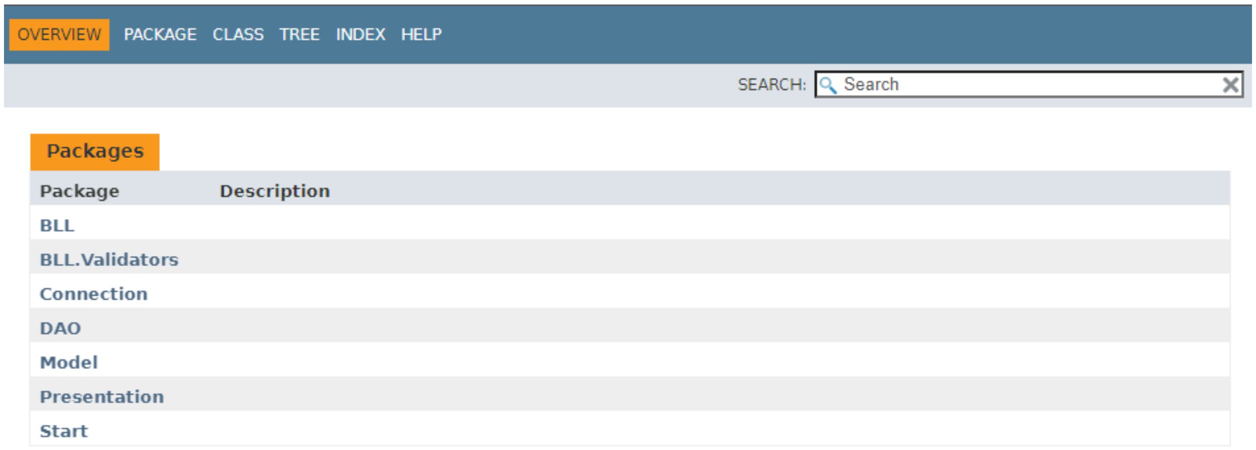
```
/**
 * Constructor with parameters
 * @param id the id of the client
 * @param name the name of the client
 * @param address the address of the client
 * @param email the email of the client
 */
```

Another requirement was that we should create JavaDoc files, which were a discovery for me. This is a documentation generator for generating API documentation in HTML format straight from the Java source code. HTML is used for easy navigation between the documents, using hyperlink.

I created my project in IntelliJ IDEA, which can automatically generate a JavaDoc HTML.

A JavaDoc comment differs from a simple comment, having the opening tag /** and ending it with */. The first paragraph should be a general description of whereas the following paragraphs can include descriptive tags such as @param, @return, @throws, or @see.

After generating the JavaDoc in IntelliJ a new folder is created, containing many HTML and script files, and if we open the index.html file, the general overview of the program is presented.



Packages

Package	Description
BLL	
BLL.Validators	
Connection	
DAO	
Model	
Presentation	
Start	

Possible further development

I think a new approach to the GUI would improve on the program. This version is intuitive but does not look really great. Also, in the View package a single Table class could be created, which displays all the 3 tables, therefore erasing the need for 3 different classes. A new validator could also be added, checking if the address provided is legitimate, though I did not think this was necessary.

Conclusion

I had a lot of fun realizing this project, working with a database and connecting it with Java. I completed a similar project in the first semester also, but I used SQLite then, and MySQL proved to be a lot more pleasant experience to work with. The Maven project simplified the connection by only including the connector dependency, not needing to have a jar file in the project directory.

Javadoc files were also a new experience to me, and I was amazed at how easy it is to create a state-of-the-art documentation with the help of JavaDocs. A good program already includes well commented code and rendering it in a good-looking way is always a plus.

I learned a new approach to program design with the layered architecture style, which I quite like, and will most likely use in the future.

The reflection technique was a bit difficult for me to grasp, but once I got it, I realized that I should have been using this approach beforehand. The simplicity of creating an abstract class, and then extending it by specific needs is elegant and effective.

I think this project taught me the ‘newest’ things so far. it was a difficult one, but it was worth to complete it.

Bibliography

Course slides

Assignment documentation

Support presentation

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

https://www.geeksforgeeks.org/

https://www.youtube.com/

https://stackoverflow.com/