

**Programming Techniques**

**Order Management**

*Course Teacher*: prof. Ioan Salomie

*Laboratory Assistant*: ing. Cristian Stan

*Student*: Ivan Alexandru-Ionut

Contents:

1. Assignment objective
2. Problem analysis, scenario, modeling, use-cases
3. Design
4. Implementation
5. Results
6. Conclusions
7. Bibliography
8. Assignment objective

The objective of this assignment was to develop an Order Management application that simply manages the orders created by the user and the creation, deletion and editing of users or products from/into the database. As secondary objectives, there are:

* To create an UI interface that the user can interact with, consisting of three windows: one for client, one for product and another one for order management.
* To use Javadoc for documenting classes and generate corresponding Javadoc files.
* To use relational databases for storing the data for the app.
* To use reflection techniques to create a method that receives a list of objects and correctly generates the header of a table by extracting through reflection the properties of the objects.
* To clearly make use of good OOP concepts

1. Problem analysis, modeling, scenario and use-cases
   1. Problem analysis

We are required to manage the orders of clients, orders that are created and placed by our user for every client. Our user has the ability to create users and manipulate them. Orders are defined by a user, a product and a quantity of the specific product. The products are also managed through our application, by the user, and every product has a quantity in the database.

We are supposed to manage all the requests our user sends and to send them further to the database, which will then make the required changes to the tables. We also had to handle all other exceptions and database-problems related for our application.

For the deletion of a client/product we also had to be sure that it will not make any problems for the understanding of orders: e.g., when a client is deleted, all its orders are deleted, as well and this thing should also happen for the products.

* 1. Modeling

We will use a Layered Architecture to make the project more readable and because it was a requirement.

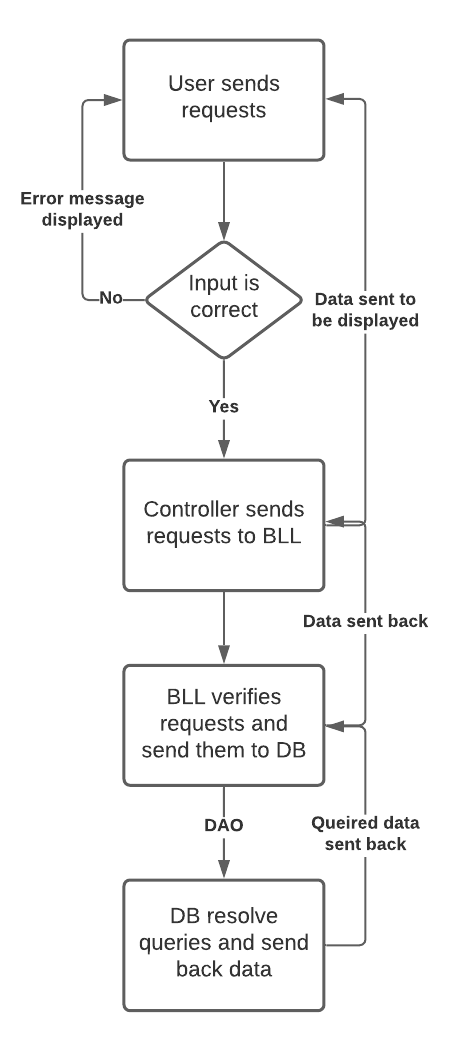
We will model the classes from the model package to suit 1:1 to the tables from the relational databases, and so we will have the Client, Product and Order class, that will all have an id as primary key. The client will also have a Name and Email, the Product will have a Name and a quantity, while the Order will contain the information for the client and product – their IDs, in order to easily fetch their information.

For the data access layer, we will follow the example from the presentation: we’ll make use of the BLLs and create an AbstractDAO to easily create the queries for every model through reflection, using generic classes. Moreover, we will create a Singleton class for the Connection to the database in order to be sure that we do not waste memory with the connection.

For the business layer, we will create the required controllers for the presentation and also the reflection classes that will generate the JTables through reflection. In this way, any change to our models will propagate with reflection and we do not have to change any other line of code.

The last layer is the presentation/view layer, where we simply display and get/create requests from the user, this was done using JavaFX.

* 1. Scenario and use-cases

The following flow-chart resumes all the scenarios the user can go through.

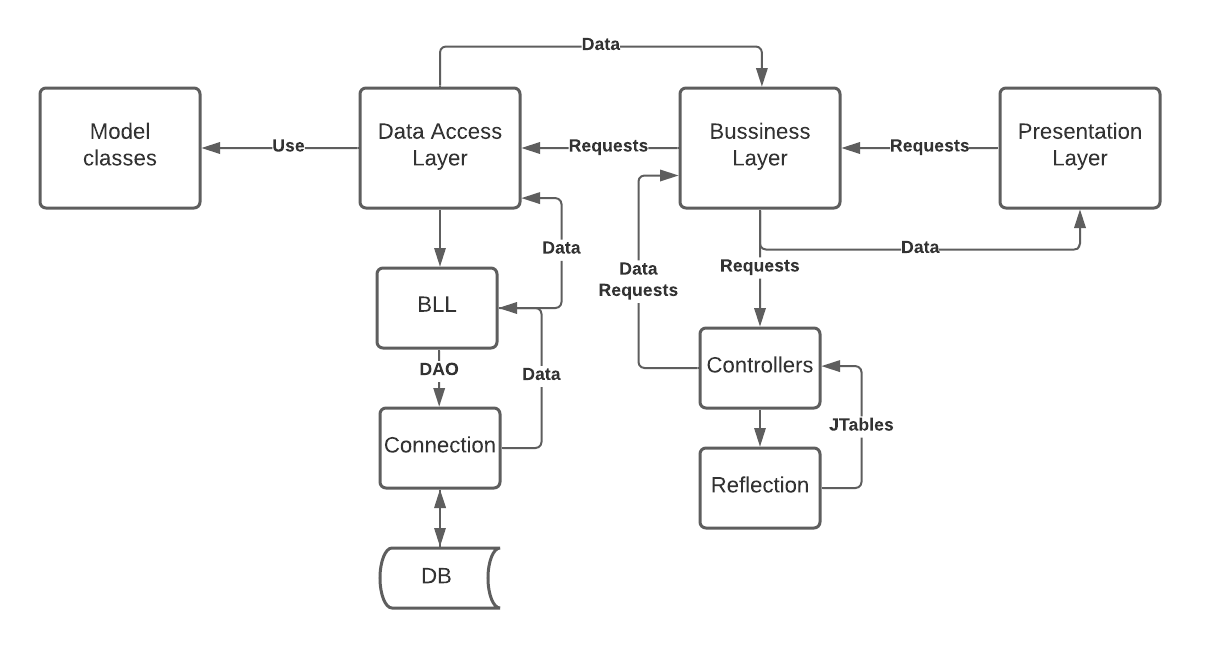
The application is done in such a manner that if the input is wrong an error message will be displayed and the user has to introduce the correct input.

After the input is processed, the controller manages the requests and sends them further to the BLL, which verifies them and then sends them to the database through the DAOs, which will make the needed changes and them send the data back to all the components. After that, the data is displayed on the View when the user refreshes the JTables.

1. Design
   1. OOP design

Regarding the OOP design, we have used Reflection, Generic classes and also Layered architecture, so the project follows the imposed OOP design. I have still used Lombok to generate the Getters, Setters and the Constructors. The models classes only have their fields, which are private, so good OOP design was used here as well. Inheritance was used when working with the generic AbstractDAO and AbstractReflection, while I have used the interface for the validators.

* 1. Class diagram / Application diagram



Above is a simplified application diagram that shows all the relation between classes and how the application works. We can now easily see the Layered Architecture that we were required to make, as we have in the left part the Model and the DataAccess Layer, and on the right the Business Layer and the Presentation one., which also reflect the MVC pattern.

* 1. Data Structures

Besides the simple data structures data I have used – integers, Booleans, floats or other primitives, I had to use SQL data structures in order to create the Connection and execute the queries, like Connection, Statement, PreparedStatement, ResultSet. For the Reflection to work, I had to use PropertyDescriptor and Method classes, also.

* 1. Class design

In this part I will briefly explain the class design regarding the division in the packages.

* **model**
  + - Client
      * This is simply the Client Model class
    - Strategy
      * The Product Model class.
    - OrderT
      * This is the Order model class, called OrderT due to SQL-query reasons.
* **dataAccessLayer**
  + - bll
      * ClientBLL, ProductBLL, OrderBLL
        + These classes manage the requests send to the DB and received from the business Layer.
      * validators
        + Email Validator

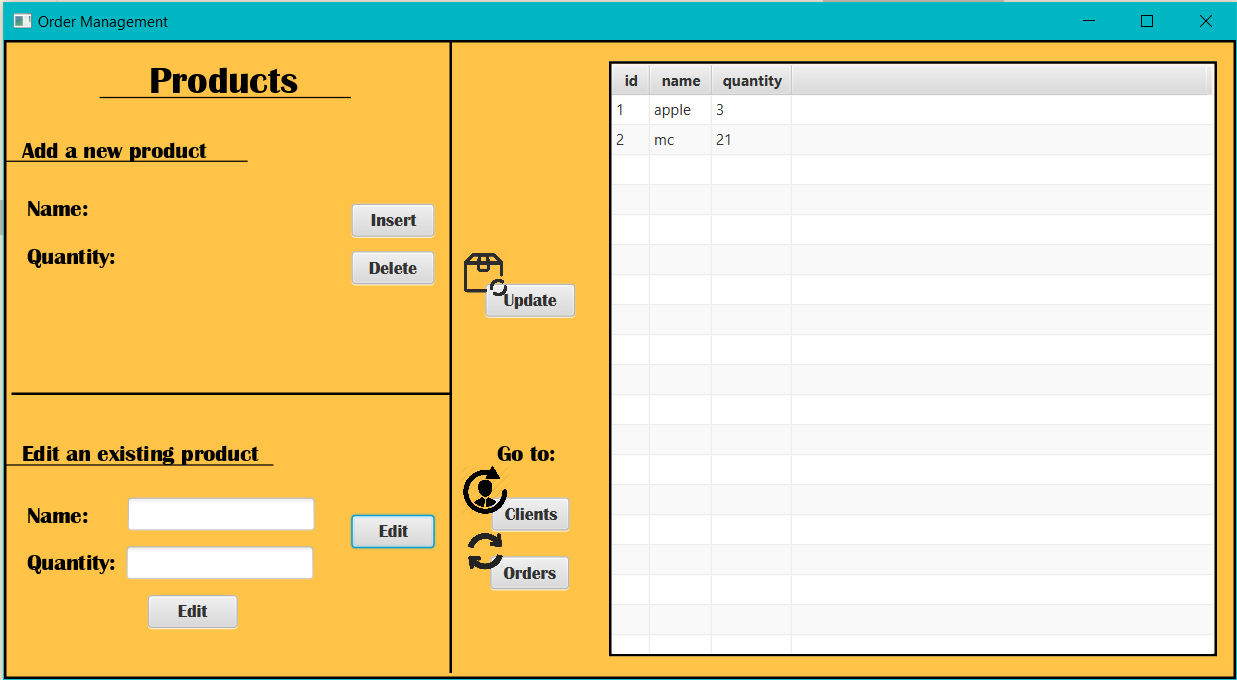
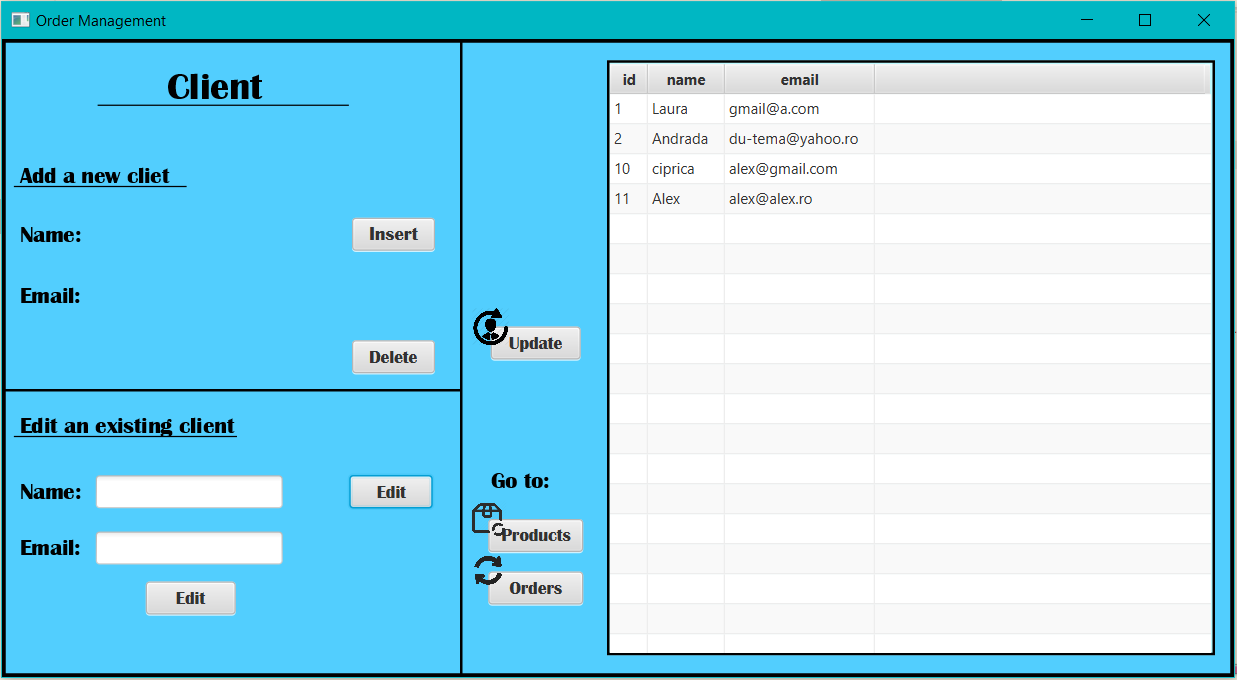
The only validator class, it validates the emails of the Clients.

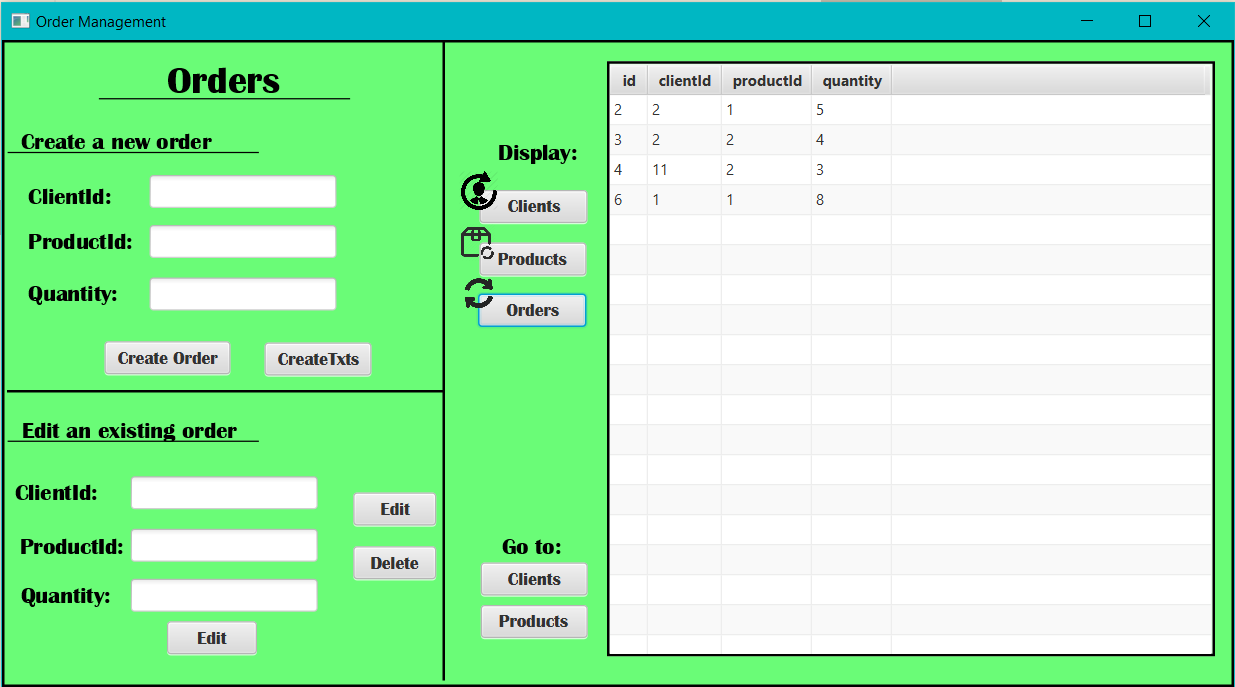
* + - * + Validator

This is the interface for the Validators.

* + - Connection
      * ConnectionFactory
        + This is the singleton class for the Connection to the Database.
    - Dao
      * AbstractDAO
        + The generic class for the DAO, being abstract as it uses Reflection.
      * ClientDAO, ProductDAO, OrderDAO
        + The DAOs for every model class that inherit the AbstractDAO class.
* bussinessLayer
  + - controller
      * ClientController, ProductController, OrderController
        + The classes that manipulate the Presentation requests and sends data to be displayed.
    - reflection
      * AbstractReflection
        + The generic class for the generation of JTables.
      * ClientReflection, ProductReflection, OrderReflection
        + The classes that generate JTables specific to the Model classes, inheriting the AbstractReflection class.
* **Presentation**
  + - **ViewsController**
      * The class that has functions that all the Controllers use.
    - ClientView, ProductView, OrderView
      * The Views for every Model classes, that are controller by their specific Controllers.

There also exists a **App** class that contains the main function and it has the responsibility of starting the application, which can also be started by using ***gradlew run.***

* 1. UI part



Above we have examples for the 3 UI windows, the blue one being the one for the Client, the orange for Product and the green one for Orders. It is easy to use and read, as the user has to input data into textFields in order to create requests and manage the tables.

1. Implementation

* **Client, Product, OrderT**
  + Nothing much to be said about the model classes, as they were created using Lombok, they just have the required fields
* **AbstractDAO**
  + The AbstractDAO implements in an abstract mode some example methods for the model classes: findById, findAll, edit, insert, delete, all of which are done using Reflection, so that the other DAO classes can easily implement the same methods by only calling the super methods.
  + It also has a createObjects method that creates a List of objects from a ResultSet using the same Reflection Technique
  + ClientDAO, ProductDAO and OrderDAO just extend this Abstract Class and call the super method, showing the power of Reflection.
* **ConnectionFactory**
  + The Singleton class used to create and handle the Connection to the database. It has simple methods, like **createConnection, getConnection** and the **close** method for Connection, Statements or ResultSets.
* **ClientBLL, ProductBLL, OrderBLL**
  + This BLL classes make use of the Client, Product and Order DAOs, through which will make requests that will be send to the DB. Every class of BLL will have the same methods that are also in the AbstractDAOs and will return what they return.The BLLs also validate the inputs that are to be inserted or updated into the DB.
* **AbstractReflection**
  + This class generated the JTable for the T type that is given using the Reflection techniques. It iterates through all the fields of the Class type and generates the column based on them. After that, it creates the Columns and populates the table with the List of Objects that is given as input. The class is abstract in order to be extended by the other classes from this package, that are specific for Client, Product or Order.
* **ClientReflection, ProductReflection, OrderReflection**
  + These classes extend the AbstractReflection in order to be able to generate the tables specific for their Model Classes and they only implement the method of generating the table by calling the super method.
* **ClientController, ProductController, OrderController**
  + These classes manage the requests from the Views and sends them further to the BLLs and the Databases, at the same time sending data back to the Views to be displayed. The Controllers also make sure the input data is correct and there is no bad request sent to the BLL, so the verification is done in this step. The controllers also manage the edit, insert, findAll, detele or generating the JTable of the current class type that will be ready to be displayed in the View class.
* **ViewsController**
  + This class implements some of the methods that are required for all the Views/Windows, such as **showAlert** or **changeSceneTo**, which shows an alert with the given parameters or respectively changes the scene to the given window.
* **ClientView, ProductView, OrderView**
  + The Views Classes implement the FXML fields and receives the first requests from the Users, which will then be transmitted to their specific Controllers and the data will be received and ready to be displayed. The Views have certain methods that are identical to the ones in the AbstractDAO: insert, display(findAll), edit or delete.

1. Results

It was not a requirement to test our methods and it was truly not necessary to test them, as we worked with Reflection. The results of the application can also be seen in the database.

1. Conclusions and further implementation

The conclusion of the project is that we make use of strong OOP concepts and clearly divide it into small sub-problems, the project becomes simpler in no time. Another conclusion is that, when you start documenting your project, you realize that you might have not understood some ideas or your might have not been so clear in your implementation, so it is always better to start your project documenting and stating your ideas and your plan.

I have also learnt new things, like using Reflection and a layered Architecture and methods, as well as using Lombok to generate methods that I need without wasting time, especially if my models have a lot of fields. I have also made my third project using the MCV pattern, so I have consolidated a more in-depth understanding of the pattern. I have learnt how to create a good documentation and good code documentation comments, while IntelliJ helped me with the Collections and Arrays functions. The other important thing I have got better at is working with git, git bash and GitLab, which is very important for the future. Also, I have learnt how to use streams, which seem to be very important and useful.

This is also my first project with Gradle and I find it very useful, just as useful as Maven is, but simpler and easier to follow **build.gradle** than **pom.xml**. So I might as well try it again in further projects using Spring Boot, probably.

The other thing that I have learnt and I think it is very important is JavaDoc and using it gave me the opportunity to generate .html files of java documentation of my project.

A further implementation for the project might be to extend the database and use more complex queries to manipulate in a complex way the relational database.

1. Bibliography

* [**https://www.baeldung.com/java-jdbc**](https://www.baeldung.com/java-jdbc)
* [**https://app.diagrams.net/**](https://app.diagrams.net/)
* **https://lucid.app/**
* [**https://dzone.com/articles/layers-standard-enterprise**](https://dzone.com/articles/layers-standard-enterprise)
* [**https://www.atlassian.com/git/tutorials/setting-up-a-repository**](https://www.atlassian.com/git/tutorials/setting-up-a-repository)
* **https://www.baeldung.com/javadoc**