Technical University of Cluj-Napoca
Programming Techniques
Assignment 3

Order Management

Teacher: Prof. Ioan Salomie
Teacher Assistant: Dr. Viorica Chifu
Student: Domokos Robert - Ștefan
Group: 30421

Content

1. Objective

This assignment has as main objective the development of an application that allows the user to perform operations on a database made of three tables Client, Product and Order. This application should allow the user to perform operations such as: inserting, updating or deleting a client, product or order. The data should be saved in the required live database.

This assignment has a vast list of secondary objectives as well, some of them being:

- Designing an interface which should be easy to use for any kind of user;
- Validating the email, age of a client and the order quantity;
- Using JavaDoc for making the necessary comments for the aplication;
- Creating the required database with valid orders, clients ad products;
- Creating a receipt for every inserted order;

2. Analysis

- Modeling and requirements

The modeling part for this assignment requires that the data, which is divided into 3 classes: Client, Product and Orderr, has to be modeled in a way that it can be processed in Java. So as input we have the data that is modeled in the manner explained above or as the database that we created and the output is resembled as a list of clients, products or orders that are inserted in the database and a receipt for each inserted order.
The 3 main classes that model the data are implemented as:

- Client
  id (int);
  name (String);
  address (String);
  email (String);
  age (int);

- Product
  id (int);
  name (String);
  quantity (int);

- Orderr
  id (int);
  idClient (int);
  idProduct (int);
  quantity (int);

- Scenarios and use case

The use case represents the ways that an actor or user can use the application with the purpose of processing the data that he inputs. If the input data type doesn't match the type required by the application then the program will print an error in the console.

The user can make multiple operation with the data present in the data base or the data that he inputs, such as:

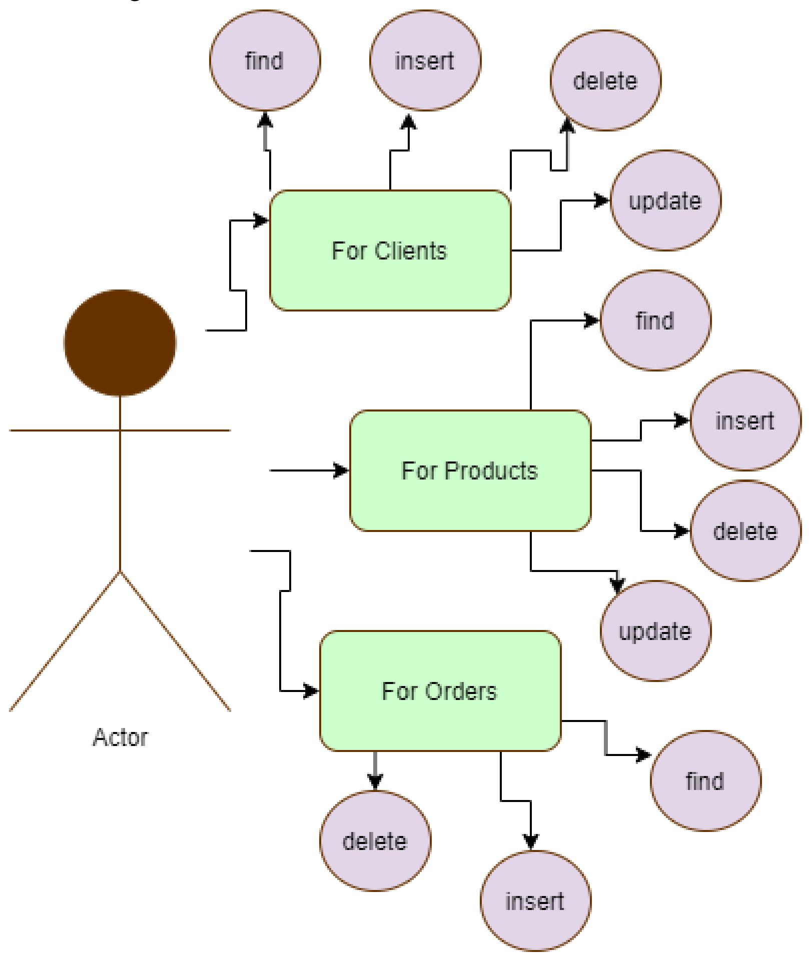
- find:
  Displays all the data from the chosen table from the database (Clients, Products, Orders);

- insert:
  Inserts a new member in the chosen table from the database;

- delete
  Deletes a chosen member from a chosen table;

- update
  Modifies the data from an already existing member from the chosen table;

  - Use case diagram



3. Design

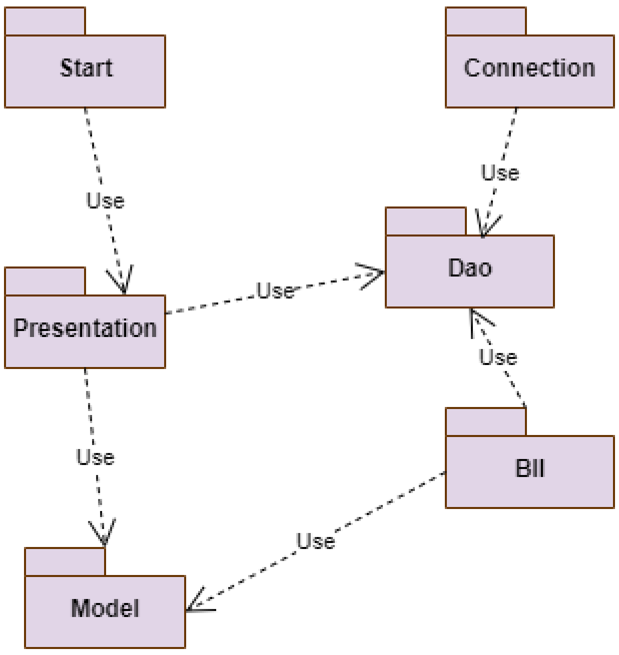
The design part of the project/assignment consists of the design decisions, UML diagrams, relationships, data structures, class design, packages and user interface;

- Design decision

As already mentioned in the requirements part for this assignment, a Layered Architecture is used in the code to separate the created classes into 6 main packages:

- model (where the data is modeled)
- presentation (where the GUI is implemented)
- start (where the main class exists)
- bll (where the validators and operations are attributed to each class)
- connection (where the connection with the database is established)
- dao (where the operations are made)

- Relationship Packages



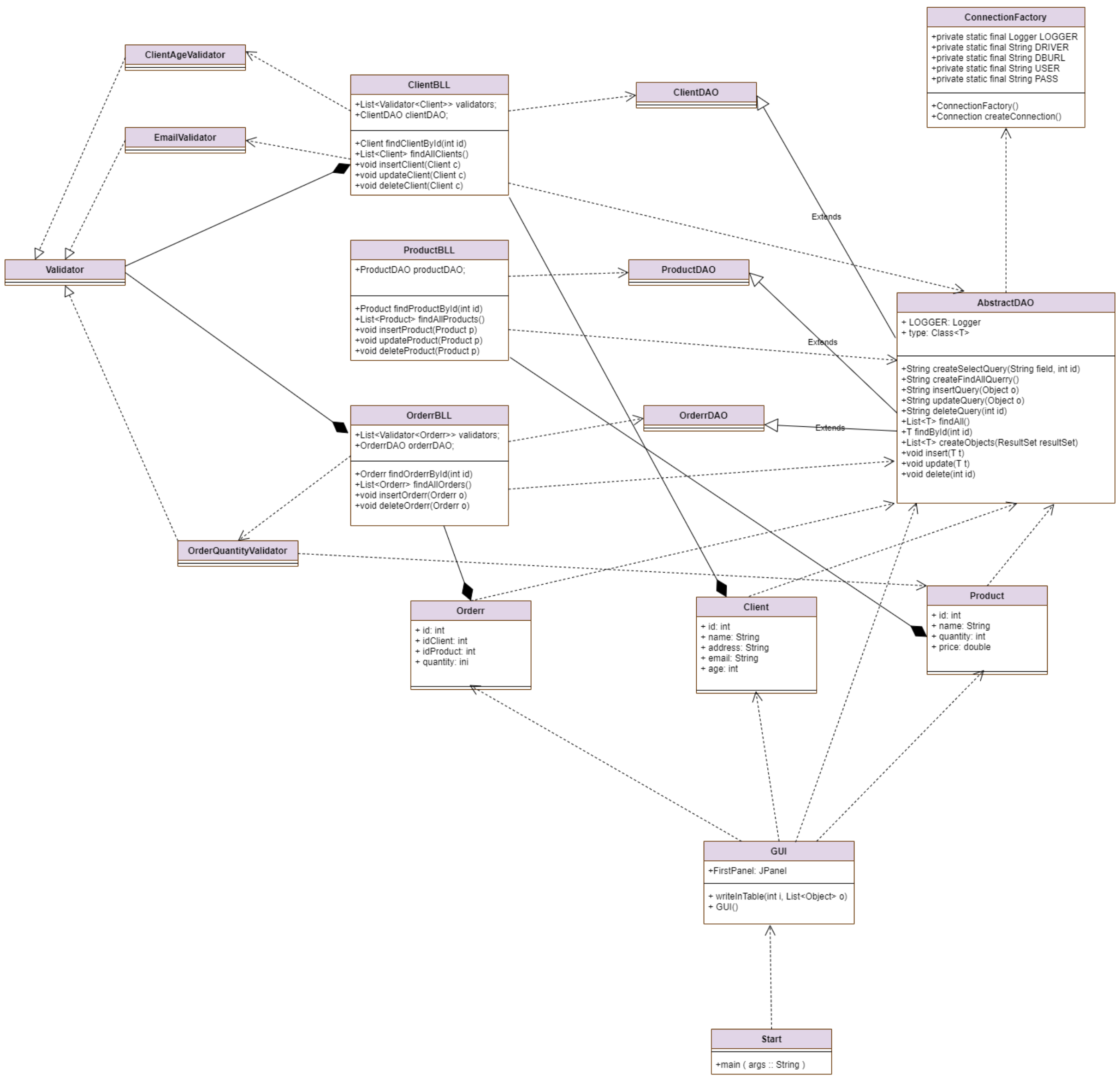
The UML diagram shows how the classes are divided in these 6 main packages already mentioned above and shows an easier to understand representation of the relationships, most of them being the relationship of dependency.

- Start: contains only the main class that runs the program;
- Dao: contains the classes that implement the operations on the data from the database using queries;
- Bll: contains the Validator package and a class for each class of the modeling package to associate the required operations;
- Model: contains the 3 main classes from the modeling part: Client, Product, Orderr;
- Presentation: contains the GUI and its class that works as a View and as a Controller;
- Connection: contains the class ConnectionFactory that resolves the connection of the database with our application;

- Data structures

As for the data structures used for this application, there aren't many unusual or special data structures used, but some interesting ones that could be mentioned here are those that help the establishment of the connection, such as Connection, Logger, Statement and ResultSet.

- Class design


ConnectionFactory
+private static final Logger LOGGER
+private static final String DRIVER
+private static final String DBURL
+private static final String USER
+private static final String PASS
+ConnectionFactory()
+Connection createConnection()
ClientAgeValidator
EmailValidator
Validator
OrderQuantityValidator
ClientBLL
+List<Validator<Client>> validators;
+ClientDAO clientDAO;
+Client findClientById(int id)
+List<Client> findAllClients()
+void insertClient(Client c)
+void updateClient(Client c)
+void deleteClient(Client c)
ClientDAO
ProductBLL
+ProductDAO productDAO;
+Product findProductById(int id)
+List<Product> findAllProducts()
+void insertProduct(Product p)
+void updateProduct(Product p)
+void deleteProduct(Product p)
ProductDAO
OrderrBLL
+List<Validator<Orderr>> validators;
+OrderrDAO orderrDAO;
+Orderr findOrderrById(int id)
+List<Orderr> findAllOrders()
+void insertOrderr(Orderr o)
+void deleteOrderr(Orderr o)
OrderrDAO
Extends
Extends
Extends
AbstractDAO
+ LOGGER: Logger
+ type: Class<T>
+String createSelectQuery(String field, int id)
+String createFindAllQuerry()
+String insertQuery(Object o)
+String updateQuery(Object o)
+String deleteQuery(int id)
+List<T> findAll()
+T findById(int id)
+List<T> createObjects(ResultSet resultSet)
+void insert(T t)
+void update(T t)
+void delete(int id)
Orderr
+ id: int
+ idClient: int
+ idProduct: int
+ quantity: ini
Client
+ id: int
+ name: String
+ address: String
+ email: String
+ age: int
Product
+ id: int
+ name: String
+ quantity: int
+ price: double
GUI
+FirstPanel: JPanel
+ writeInTable(int i, List<Object> o)
+ GUI()
Start
+main (args :: String)

The classes that compose this project are presented above in the Class Diagram. The main classes required for the application can be presented such:

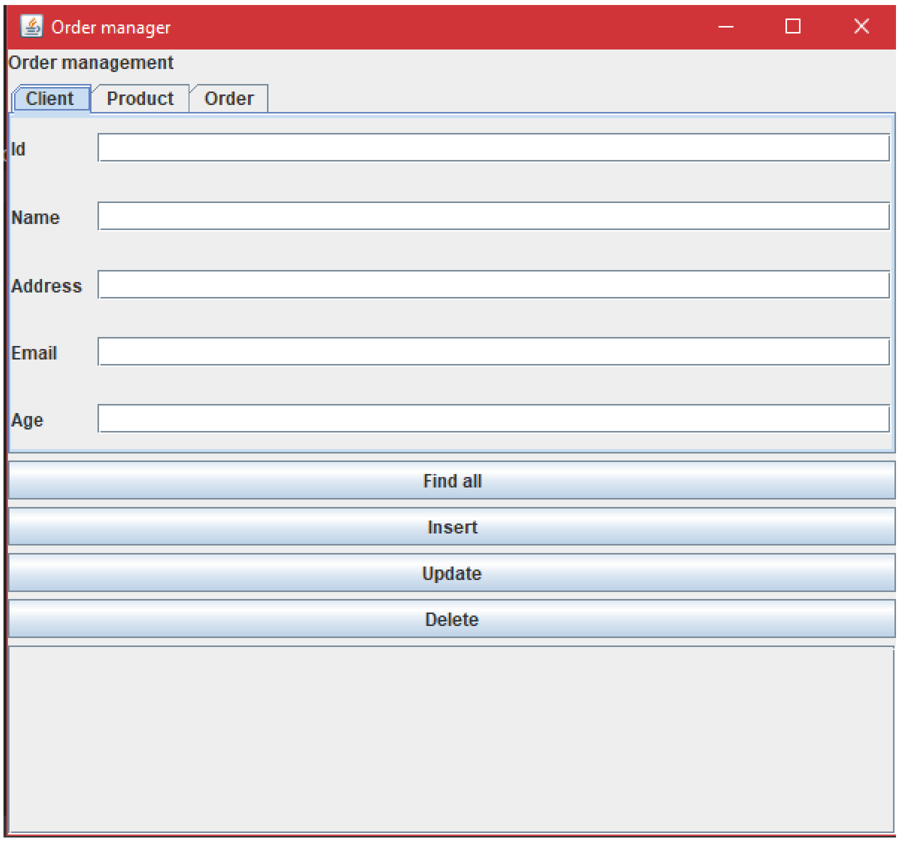
- Start: contains only the main function that runs the execution of the application;
- GUI: contains the buttons, text fields and every other object that form the interface;
- Client: models the clients;
- Product: models the products;
- Orderr: models the orders;
- ConnectionFactory: establish the connection between the application and the database from MySQL;
- AbstractDAO, ClientDAO, ProductDAO, OrderrDAO are classes that model the data from database and use queries to implement the operations;
- ClientBLL, OrderrBLL, ProductBLL: add to each one of the classes Client, Orderr, Product, the validators and the operations implemented in AbstractDAO;
- Validator: is a interface that is implemented by the other 3 classes that validates;

For the better understanding of the diagram visit the Implementation part of the documentation where every class is explained together with every method that these classes possess.

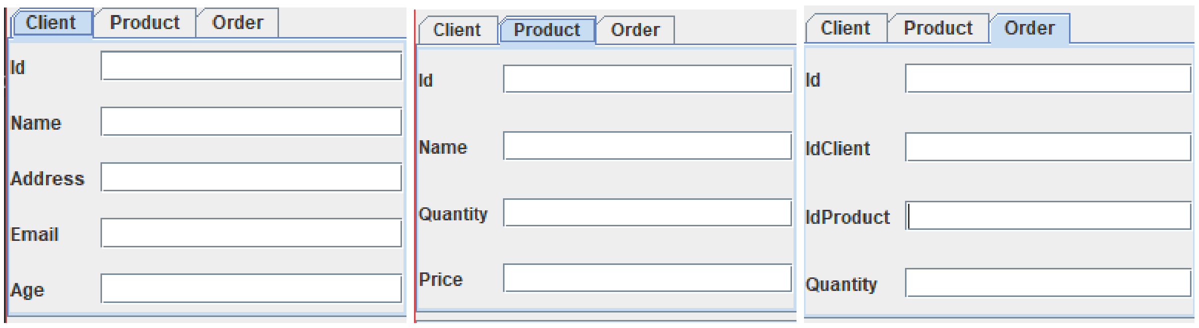
- User Interface

The user interface is composed of multiple text fields as JTextFields that make possible for the user to input the data he wants, four buttons Find all, Insert, Delete, Update each representing the operation that can be made on the data from the database and a JTable where the data from a specific class is presented. The text fields are divided in a JTabbedPane that allows the user to choose in which class he wants to operate.

The interface looks as follows:



And this is how the division of the text fields in the JTabbedPane looks like:



4. Implementation

For this part I will take every class and explain all the important methods that are implemented:

- Start.java

This class contains the main function that runs the code for the application and starts the interface.

- GUI.java
  Forms the GUI together with the buttons and the necessary text fields to process the data.

```
public void writeInTable(int i,List<Object> o)
  {
     DefaultTableModel tab = (DefaultTableModel) table1.getModel();
     tab.setColumnCount(0);
     tab.setRowCount(0);
     Class c;
     if(i == 0)
        c = Client.class;
     else if(i == 1)
        c = Product.class;
     else
        c = Orderr.class;
     int j=0;
     String[] arr = new String[c.getDeclaredFields().length];
     for(Field f: c.getDeclaredFields()){
        f.setAccessible(true);
        arr[j++]=f.getName();
     }
     tab.setColumnIdentifiers(arr);
     for(Object ob: o){
        List<String> st = new ArrayList<>();
        for(Field f: c.getDeclaredFields()){
           f.setAccessible(true);
            try {
               st.add(f.get(ob).toString());
            } catch (IllegalAccessException e) {
               e.printStackTrace();
            }
        }
        tab.addRow(st.toArray());
     }

  }
```

This method helps me use the JTable to print the data from the database tables on my interface by taking the selected pane form the JTabedPane with the “int i” parameter.

- Client.java
  This class models the clients that have as fields the id, name, address, email and the age.

- Product.java
  This class models the products that have as fields the id, name and the quantity.

- Orderr.java
  This class models the orders that have as fields the id, idClient, idProduct and quantity.

- AbstractDAO.java
  This class uses queries to make operations with the data from the database. This operations are divided in: Find all, Insert, Update and Delete.

Some methods that return the string with the query that executes the operations are:

- for the FindAll operation:

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

- for the Insert operation:

```
public String insertQuery(Object o){
                StringBuilder sb= new StringBuilder();
                sb.append("INSERT INTO ");
                sb.append(type.getSimpleName());
                sb.append(" VALUES(");
                for(Field f: o.getClass().getDeclaredFields()){
                        f.setAccessible(true);
                        try {
                                sb.append("'" + f.get(o) + "'");
                                sb.append(",");
                        } catch (IllegalAccessException e) {
                                e.printStackTrace();
                        }
                }
                sb.deleteCharAt(sb.length()-1);
                sb.append(")");
                return sb.toString();
        }
```

- for the Update operation:

```
public String updateQuery(Object o){
                StringBuilder sb= new StringBuilder();
                sb.append("UPDATE ");
                sb.append(type.getSimpleName());
                sb.append(" SET ");
                for(Field f: o.getClass().getDeclaredFields()){
                        f.setAccessible(true);
                        try {
                                sb.append(f.getName() + "=" + "'" + f.get(o) + "'");
                                sb.append(",");
                        } catch (IllegalAccessException e) {
                                e.printStackTrace();
                        }
                }
                sb.deleteCharAt(sb.length()-1);
                Field f2 = null;
                try {
                        f2 = o.getClass().getDeclaredField("id");
                        f2.setAccessible(true);
                        sb.append(" WHERE id=" + f2.get(o));
                } catch (NoSuchFieldException | IllegalAccessException e) {
                        e.printStackTrace();
                }
                return sb.toString();
        }
```

- for the Delete operation:

```
public String deleteQuery(int id){
                StringBuilder sb= new StringBuilder();
                sb.append("DELETE FROM ");
                sb.append(type.getSimpleName());
                sb.append(" WHERE id=" + id);
                return sb.toString();
        }
```

- ClientDAO.java
  Extends the AbstractDAO.java class.

- ProductDAO.java
  Extends the AbstractDAO.java class.

- OrderrDAO.java
  Extends the AbstractDAO.java class.

- ConnectionFactory.java
  This class connects the code from the application to the database from MySQL by creating a new connection with the method:

```
private Connection createConnection() {
                Connection connection = null;
                try {
                        connection = DriverManager.getConnection(DBURL, USER, PASS);
                } catch (SQLException e) {
                        LOGGER.log(Level.WARNING, "An error occured while trying to connect to the
database");
                        e.printStackTrace();
                }
                return connection;
        }
```

- ClientBLL.java
  This class is used to validate the insertion of the clients and it adds the operation for the Client class.

- ProductBLL.java
  This class is used to add the operation for the Product class.

- OrderrBLL.java
  This class is used to validate the insertion of the orders and it adds the operation for the Orderr class. In this class also it is updated the product quantity after we checked if the order can be processed and this is made in the insert method:

```
public void insertOrderr(Orderr o){
    for(Validator<Orderr> v:validators)
       v.validate(o);
```

```
        ProductBLL bll = new ProductBLL();
        int id = o.getIdProduct();
        Product p = bll.findProductById(id);
        p.setQuantity(p.getQuantity()-o.getQuantity());
        bll.updateProduct(p);
        orderrDAO.insert(o);
        String s = writeTxt(o);
        try {
            bill.write(s);
            bill.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

- Validator.java
  Is an interface that is used to generate the required or the needed number of Validators for the data.

- OrderQuantityValidator.java
  This class implements the Validator.java interface and verifies if the order can be made.

```
public void validate(Orderr orderr) {
        int id = orderr.getIdProduct();
        ProductBLL productBLL= new ProductBLL();
        Product p=productBLL.findProductById(id);
        if(p.getQuantity()< orderr.getQuantity()){
            throw new IllegalArgumentException("The quantity is too big for this order!");
        }
    }
```

- EmailValidator.java
  This class implements the Validator.java interface and verifies if the email of a client is valid using regex.

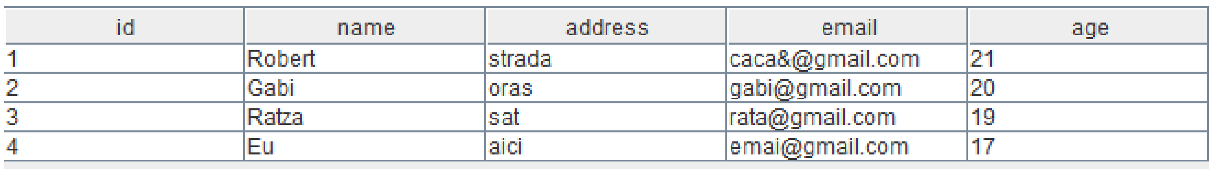
- ClientAgeValidator.java
  This class implements the Validator.java interface and verifies if the age of a client is in the interval [7, 30].

5. Results

The results are composed of the data printed in the bottom part of the GUI in the JTable and is the exact data that is present live on the database.

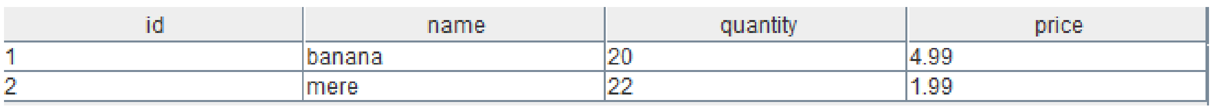
We can see in the example below how the application will show the data printed in all of the 3 main classes with their given fields.

Client:



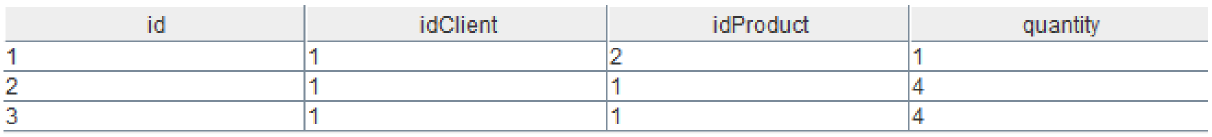
id	name	address	email	age
1	Robert	strada	caca&@gmail.com	21
2	Gabi	oras	gabi@gmail.com	20
3	Ratza	sat	rata@gmail.com	19
4	Eu	aici	emai@gmail.com	17

Product



id	name	quantity	price
1	banana	20	4.99
2	mere	22	1.99

Orderr:



id	idClient	idProduct	quantity
1	1	2	1
2	1	1	4
3	1	1	4

6. Conclusions

In conclusion, this assignment was a challenge for me, but it also improved my skills in working at the same time with a database and an application that runs while interacting with it. While working on this assignment I learned the importance of the reflection that allows us to shorten our code while also respecting the requirements.

A significant part from this application was the fact that we worked on a real world example which made it easier to understand the functionality of the connections between our code and the database and the Layered Architecture used, but also helped me learn about them in a quicker manner.

Some future improvements to the application that I think would make it better are:

- Making a direct way to edit the data from the tables;
- Making a much simpler and friendly interface;
- Making the application usable for the real world companies that require such a system;
- Making a system that notifies the user if the data was changed successfully;
- Modeling more data that would extend the number of classes for a larger reflexion base;

7. Bibliography

https://www.geeksforgeeks.org/what-is-javadoc-tool-and-how-to-use-it/
http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/
https://dzone.com/articles/layers-standard-enterprise
http://tutorials.jenkov.com/java-reflection/index.html
https://www.baeldung.com/java-pdf-creation
https://www.baeldung.com/javadoc