**Documentation Homework 3**

**Order Management**

First Name: Marius

Last Name: Măcean

Group: 30423/2

Laboratory Professor: Marcel Antal

**Table of contents**

1. Functional Requirements……………………………………………………………………………..……. 3
2. Objectives…………………………………………………………………………………………….….… 3
   1. Main Objective…………………………………………………………………………………..… 3
   2. Secundary Objectives………………………………………………………………………...…..... 3

.

1. Analysing the Problem………………………………………………………………………………...….… 4
2. Projection……………………………………………………………………………………………..….….. 5
   1. Data Structures…………………………………………………………………………….…..…... 5
   2. Main Class Diagrams..…...………………………………………………………………….....….. 5
   3. Algorithms……………………………………………………………………………………….… 7
3. Implementation…………………………………………………………………………………………..… 10
4. Testing……………………………………………………………………………………………….….…. 15
5. Conclusions and Future Developments……………………………………………………………….…… 16
6. Bibliography………………………………………………………………………………………………. 16
7. **Functional Requirements**

* Use an Object-oriented programming design, classes with maximum 300 lines, methods with maximum 30 lines, Java naming conventions.
* Use javadoc for documenting classes and generate the corresponding JavaDoc files.
* Use relational database for storing the data for the application, minimum three tables: Client, Product, Order.
* Create a graphical user interface including:
* A window for client operations: add new client, edit client, delete client, view all clients in a table (JTable);
* A window for product operations: add new product, edit product, delete product, view all products in a table (JTable);
* 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 cse there are not enough products, and under stock message will be displayed. After the order is finalized, the product stock is decremented.
* Use reflection techniques to create a method that recieves a list of objects 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.
* Use Layered Architecture ( the application will contain at least four packages: dataAccessLayer, businessLayer, model and presentation ).
* 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.

1. **Objectives**
   1. **Main Objective**

The main objective of the project is to come up with of solution of an implementation of a simulation application aiming to process clients orders for a warehouse. It is strongly recommended to use relation database in order to store the products, the clients and the orders. Furthermore, the application should be structured in packages using a layered architecture.

* 1. **Secundary Objectives**

|  |  |  |
| --- | --- | --- |
| Objective | Description | Chapter in the documentation |
| Development of the use cases and different scenarios. | Use of behavioral diagrams in order to describe the goals of the application, the users and other systems that interact with the developed program. This part explain the best how the application has to work. Also, here they are presented several particular cases when interacting with the program (that can ruin it if not handled properly). | 3 |
| Choosing the data structures | Use appropriate data to store date types, String types and int types. The clients, products and orders will be stored in lists of objects. | 4 |
| Class distribution | Every main object in the system will form a class (int the **model** layer). Moreover, there will be needed the main class (int the **start** layer), a UI part, with the View class and the Controller class (in the **presentation** layer), the data acces classes (one corresponding to each model, in the **dao** layer) and the business logic classes (in the bll Layer). Also, there will be a connection class (in the **connection** layer), that will make the connection between the java code and the MySQL database. | 4 |
| Development of the algorithms | Each operation (the insertion of an object – client/ product/ order in the database, the deletion of an object from the database, the update of an object in the database) consits of a certain algorithm that has to be properly developed, in order to reach efficiency and also not to get unexpected errors (bugs). | 4 |
| Solution Implementation | In order to implement the solution, it is required to declare the classes, the attributes for each class, the data structures. The next step is to implement the methods in every class. Using the methods, the algorithms will be implemented, translated from the pseudocode written when developing them. | 5 |
| Testing | It is required to test every single case in order not to omit any particular one. This is the part where the bugs are fixed. For some logs of events (varying the data input), it is good to test at least 3 different scenarios and see the result. When a result is not the correct one or an error occurs, the code will be debugged on that exact scenario (reviewing it from the class where the result is displayed on the GUI to the class where the main methods are implemented). | 6 |

1. **Analysing the Problem**

When analysing the problem, use cases for every operation on the queues will be provided. For example, an analysis of the operation of updating an object (a product – for example) is described below.

Use case name: Update the product based on the input from the user.

Actors:

* <Finding the product> system (based on the input “id” , added by the user in the GUI, the program will search the product with that exact id);
* <Changing some fields> system (the part of the application that will change some fields of the found product, but only those received from the user, the others will be replaced with the old ones);
* <Building and updating the product> system (after the product will be changed, the database will be updated with the new values of the fields of this product);

Triggers:

* The button “edit product” is clicked.

Pre-conditions:

* Some fields of the product has to be introduced into the GUI, especially the .

Post-condition:

* The user will be notified if the product was successfully updated.

Normal Flow:

* The system will read the input from the GUI.
* The < Finding the product > system will search (using a simple algorithm) the product that has to be changed from the list of products from the database, based on its id.
* Then, it will check what fields were given new values and what fileds has to keep their old values.
* The <Changing some fields> system will change the fields that have to be changed with the new introduced values.
* The <Building and updating the product> system will update the product in the database with the new values using a query.
* The system will display a message of confirmation of update in case of successful operation.

Alternate Flow:

From 5: The system will find out that there are some newly introduced values of fields that are not valid, therefore it will not update it and will return to step 1.

From 2: The system will not found any product in the database in the that certain id. It will return to step 1 and continue.

1. **Projection**
   1. **Data Structures**

There will be used some Data Structures in order to retrieve information from the database. All these structures are of lists of objects or Class Type:

Class<T> type,

List<T> list

* 1. **Main Class Diagrams**

|  |
| --- |
| Client |
| -age : int  -email : String  -address : String  -name : String  -id : int |
| + toString() : String  Getters, Setters  + Client(name : String, address : String, email : String, age : int)  + Client(id : int, name : String, address : String, email : String, age : int)  + Client() |

|  |
| --- |
| Product |
| -stock : int  -material : String  -price : int  -name : String  -id : int |
| + toString() : String  Getters, Setters  + Product(name : String, price : int, material : String, stock : int)  + Product (id : int, name : String, price : int, material : String, stock : int)  + Product () |

|  |
| --- |
| Order |
| -productID : int  -clientID : int  -price : int  -date : String  -id : int |
| + toString() : String  Getters, Setters  + Order(date : String, price : int, clientID : int, productID : int)  + Order (id : int, date : String, price : int, clientID : int, productID : int)  + Order () |

|  |
| --- |
| ConnectionFactory |
| -singleInstance : ConnectionFactory  -PASS : String (readOnly)  -USER : String (readOnly)  -DBURL : String (readOnly)  -DRIVER : String (readOnly)  -LOGGER : Logger (readOnly) |
| + close(resultSet : ResultSet) : void  + close(statement : Statement) : void  + close(connection : Connection) : void  + getConnection() : Connection  -createConnection() : Connection  -ConnectionFactory() |

|  |
| --- |
| Controller |
| -panel : View |
| -retrieveProperties (object : Object) : String  + actionPerformed(e : ActionEvent) : void  + Controller(panel : View) |

|  |
| --- |
| AbstractDAO |
| -type : Class<T> (readOnly)  # LOGGER : Logger (readOnly) |
| + delete(id : int) : T  + update (t : T) : T  + retrieveProperties (object : Object) : void  -createObjects (resultSet : ResultSet) : List<T>  + findById (id : int) : T  + findAll() : List<T>  -createSelectAllQuery() : String  -createDeleteQuery(field : String) : String  -createUpdateQuery(t : T) : String  -createInsertQuery(t : T) : String  -createSelectAllQuery(field : String) : String  + AbstractDAO() |

ClientDAO

Class Relationships:

ClientBLL

Start

= extends

OrderBLL

ProductBLL

*Validation*

*Classes*

*Model*

*Classes*

ProductDAO

OrderDAO

ConnectionFactory

AbstractDAO

Controller

View

* 1. **Algorithms**

The most complex algorithm of the project are those applied when handling the input from the user and diplaying the result on the GUI. All of them basically consist of a for loop that accesses the database and then applying one of the operations (add, update, delete, show all) on the retrieved object / objects.

1. **Implementation**

Start.java class. Its main method will call the method from the panel, that will run the application (from the GUI).

Client.java, Product.java, Order.java classes. These are the classes that represent the main objects of the project. Also, each of them corresponds to a table from the database.

Important method:

@Override  
public String toString() {  
 return "Client [id=" + id + ", date=" + date + ", price=" + price + ", clientID=" + clientID + ", productID=" + productID  
 + "]";  
}

AbstractDAO.java class.This class hold the most of the main methods of the project. It is a generic class, using reflection techniques to access the database.

Attributes:

* Logger LOGGER: it is used to catch exceptuions thrown by the MySQL database;
* Class<T> type: holds the generic class of the main objects.

Important methods:

* createInsertQuery: uses StringBuilder in order to build the string that will represent the MySQL query. It also uses reflection techniques to receive data of the object *t* from the database.

private String createInsertQuery(T t) {  
 StringBuilder sb = new StringBuilder();  
 sb.append("INSERT INTO ");  
 sb.append(type.getSimpleName());  
 sb.append("(");  
 boolean firstField = true;  
 int nrOfFields = 0;  
 for (Field field : t.getClass().getDeclaredFields()) {  
 nrOfFields++;  
 field.setAccessible(true);  
 try {  
 if (!firstField) {  
 sb.append(", ");  
 } else {  
 firstField = false;  
 }  
 sb.append(field.getName());  
  
 } catch (IllegalArgumentException e) {  
 e.printStackTrace();  
 }  
 }  
 sb.append(") " + "values (NULL");  
 for (int i = 1; i < nrOfFields; i++) {  
 sb.append(", ?");  
 }  
 sb.append(")");  
 return sb.toString();  
}

* createSelectQuery, createUpdateQuery, createDeleteQuery, createSelectAllQuery: are methods similar to the one above, but for different queries;
* findAll: return the list off all objects of a certain type from the database;

public List<T> findAll() {  
 Connection connection = null;  
 PreparedStatement statement = null;  
 ResultSet resultSet = null;  
 String query = createSelectAllQuery();  
 try {  
 connection = ConnectionFactory.getConnection();  
 statement = connection.prepareStatement(query);  
 resultSet = statement.executeQuery();  
  
 return createObjects(resultSet);  
 } catch (SQLException e) {  
 LOGGER.log(Level.WARNING, type.getName() + "DAO:findAll " + e.getMessage());  
 } finally {  
 ConnectionFactory.close(resultSet);  
 ConnectionFactory.close(statement);  
 ConnectionFactory.close(connection);  
 }  
 return null;  
}

* findById: return the object from the database having the id given as argument;

public T findById(int id) {  
 Connection connection = null;  
 PreparedStatement statement = null;  
 ResultSet resultSet = null;  
 String query = createSelectQuery("id");  
 try {  
 connection = ConnectionFactory.getConnection();  
 statement = connection.prepareStatement(query);  
 statement.setInt(1, id);  
 resultSet = statement.executeQuery();  
 System.out.println("id: " + id);  
 return createObjects(resultSet).get(0);  
 } catch (SQLException e) {  
 LOGGER.log(Level.WARNING, type.getName() + "DAO:findById " + e.getMessage());  
 } finally {  
 ConnectionFactory.close(resultSet);  
 ConnectionFactory.close(statement);  
 ConnectionFactory.close(connection);  
 }  
 return null;  
}

* createObjects: it returns a list of the objects received from the database in the form of a List<Object>;

private List<T> createObjects(ResultSet resultSet) {  
 List<T> list = new ArrayList<T>();  
  
 try {  
 while (resultSet.next()) {  
 T instance = type.newInstance();  
 for (Field field : type.getDeclaredFields()) {  
 Object value = resultSet.getObject(field.getName());  
 PropertyDescriptor propertyDescriptor = new PropertyDescriptor(field.getName(), type);  
 Method method = propertyDescriptor.getWriteMethod();  
 method.invoke(instance, value);  
 }  
 list.add(instance);  
 }  
 } catch (InstantiationException e) {  
 e.printStackTrace();  
 } catch (IllegalAccessException e) {  
 e.printStackTrace();  
 } catch (SecurityException e) {  
 e.printStackTrace();  
 } catch (IllegalArgumentException e) {  
 e.printStackTrace();  
 } catch (InvocationTargetException e) {  
 e.printStackTrace();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 } catch (IntrospectionException e) {  
 e.printStackTrace();  
 }  
 return list;  
}

* insert: it inserts an object to the table of objects from the database;

public T insert(T t) {  
 Connection connection = null;  
 PreparedStatement statement = null;  
 String query = createInsertQuery(t);  
 System.out.println(query);  
 try {  
 connection = ConnectionFactory.getConnection();  
 statement = connection.prepareStatement(query);  
 int nrOfFields = 0;  
 for (Field field : t.getClass().getDeclaredFields()) {  
 field.setAccessible(true);  
 Object value;  
 try {  
 value = field.get(t);  
 nrOfFields++;  
 if (nrOfFields != 1) {  
 statement.setObject(nrOfFields - 1, value);  
 }  
  
 } catch (IllegalArgumentException e) {  
 e.printStackTrace();  
 } catch (IllegalAccessException e) {  
 e.printStackTrace();  
 }  
 }  
 int resultSet = statement.executeUpdate();  
  
 if (resultSet == 1) return t;  
 } catch (SQLException e) {  
 LOGGER.log(Level.WARNING, type.getName() + "DAO:insert " + e.getMessage());  
 } finally {  
 ConnectionFactory.close(statement);  
 ConnectionFactory.close(connection);  
 }  
 return null;  
}

* update: it updates an object from the table of objects from the database;

public T update(T t) {  
 Connection connection = null;  
 PreparedStatement statement = null;  
 String query = createUpdateQuery(t);  
 try {  
 connection = ConnectionFactory.getConnection();  
 statement = connection.prepareStatement(query);  
 int nrOfFields = 0;  
 boolean firstField = true;  
 Object id = null;  
 for (Field field : t.getClass().getDeclaredFields()) {  
 field.setAccessible(true);  
 Object value;  
 try {  
 if (firstField) {  
 id = field.get(t);  
 firstField = false;  
 }  
 value = field.get(t);  
 nrOfFields++;  
 statement.setObject(nrOfFields, value);  
  
 } catch (IllegalArgumentException e) {  
 e.printStackTrace();  
 } catch (IllegalAccessException e) {  
 e.printStackTrace();  
 }  
 }  
 nrOfFields++;  
 statement.setObject(nrOfFields, id);  
 int resultSet = statement.executeUpdate();  
  
 if (resultSet == 1) return t;  
 } catch (SQLException e) {  
 LOGGER.log(Level.WARNING, type.getName() + "DAO:update " + e.getMessage());  
 } finally {  
 ConnectionFactory.close(statement);  
 ConnectionFactory.close(connection);  
 }  
 return null;  
}

* delete: it deletes an object from the table of objects from the database;

ClientDAO.java, ProductDAO.java, OrderDAO. java classes. Each one of them extends the AbstractDAO.java class, therefore, they use its methods.

ClientBLL.java, ProductBLL.java, OrderBLL.java and the classes from validators package. These are the classes that validates the input given by the user (check if the input fot the fields of each object is correctly introduces – ame type, not null, not negative etc.).

ConnectionFactory.java class. It connects the project classes presented above with the database from the server where all the tables corresponding to the models are created.

Attributes:

* Logger LOGGER: gets the name of the connection, in order to catch exceptions thrown by the database;
* String DRIVER: the name of the driver for the database;
* String DBURL: the datapath of the database
* String USER: the username of the server;
* String PASS: the password of the server;

Important methods:

* createConnection: connect to the database using its information;

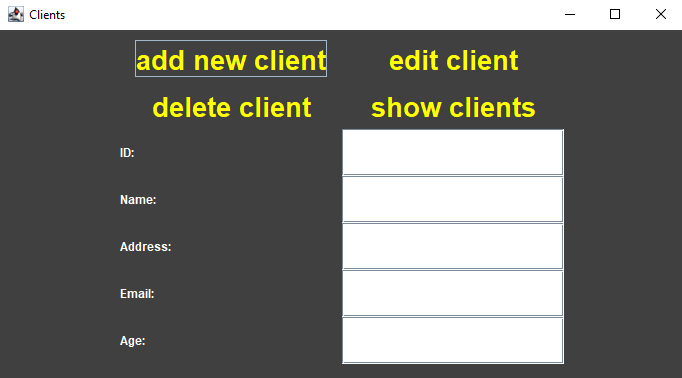
private Connection createConnection() {  
 Connection connection = null;  
 try {  
 connection = DriverManager.getConnection(DBURL, USER, PASS);  
 } catch (SQLException e) {  
 LOGGER.log(Level.WARNING, "An error occured while trying to connect to the database");  
 e.printStackTrace();  
 }  
 return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

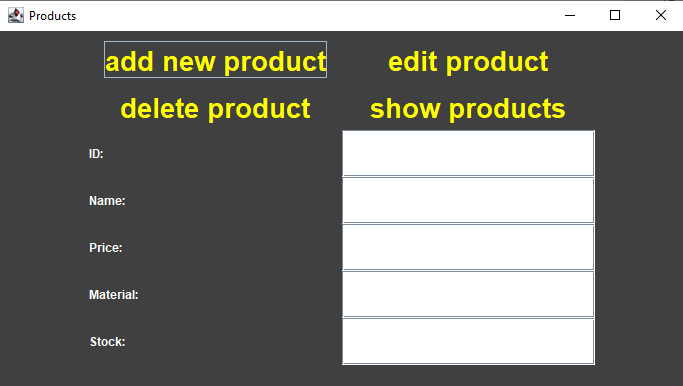
* getConnection: getter;
* close(Connection connection), close(Statement statement), close(ResultSet resultSet): close the connection to the database, as well as the statement and resulted data received from the database.

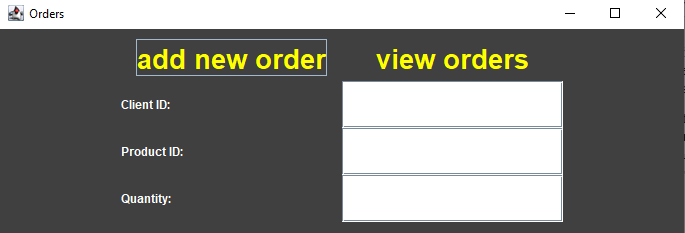
Controller.java, View.java classes. They represent the GUI classes, the classes that create what the users see and interact with. The magic of the screen of the program happens here.

1. **Testing**

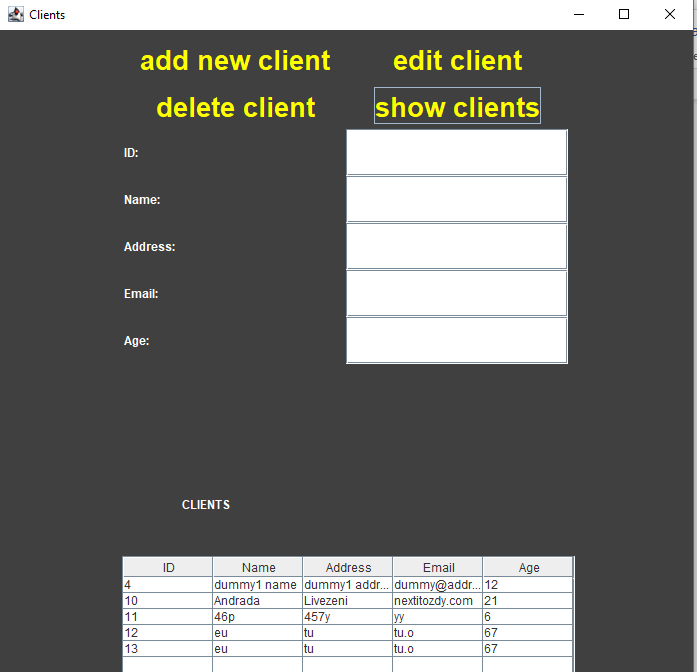
The windows before introducing any data look like this:



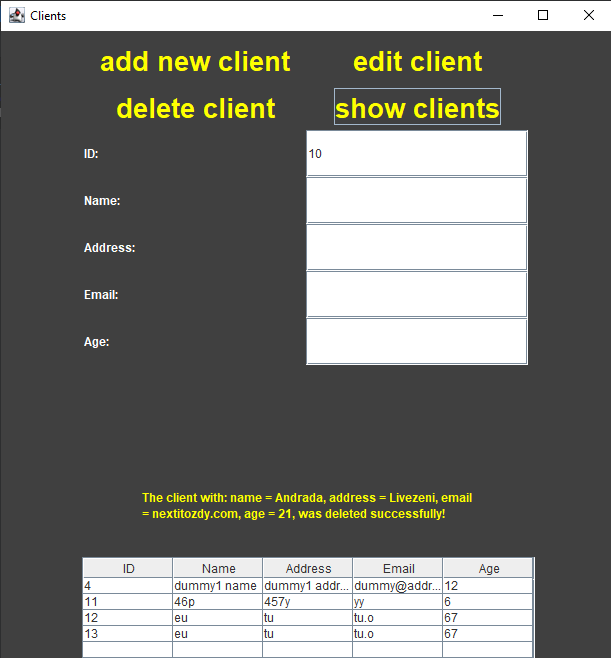




Test 1 - show client table content:

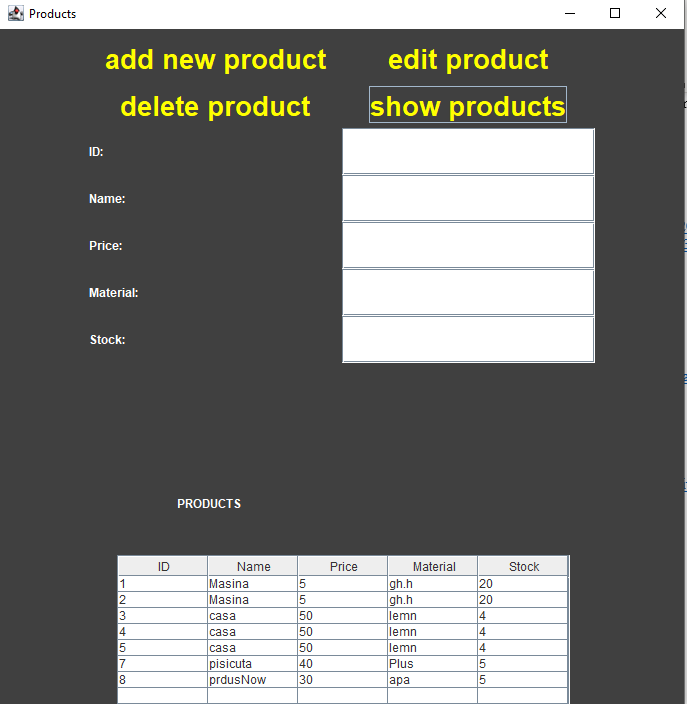


Now let’s delete client 10, for examle, by typing its id (10) in the ID texbox. Then see the result:

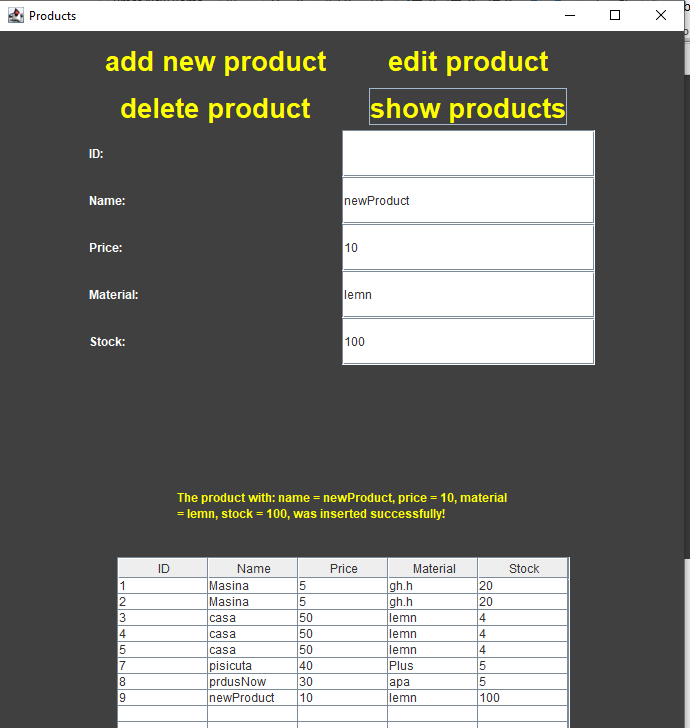


As we can see,the client was deleted and a message of confirmation appeared on the screen.

Test 2 – add product to the product table:

Initial table:

Ler’s add a product with the information : Name = newProduct, Price = 10, Material = lemn, Stock = 100 (we do not have to type in the id, because it is autogenerated):



The product was succesfully added.

Test 3 – edit this new product, but ler’s just edit its material, from lemn, to platina.



It worked, as well.

Test 4 – try to make a new order wanting the quantity that is not in stock, for example: 150 of the last product introduced (newProduct):

**Conclusions and Future Developments**

The algorithms used in the program code are quite simple to implement. The more compicated/ advanced part was working with the MySQL database and getting the connection right. The obtaining of the data from the database using reflection techniques were a also little hard to understand. Nevertheless, building the GUI using the reflection technique and the connection to the database included all of the concepts learnt in this project. While testing and debugging (on particular cases), there were a lot of minor flaws that had to be fixed. Overall, I loved the fact that I had to work on an application that has huge possible implementation in the real life. I really feel that I gained a lot of knowledge about the coding in java using MySQL database after finishing this project.

As future developments, the simulator might also be able to create an order only by clicking on the desired product and on the client making that order, rather that typing their ids in the textboxes. Moreover, the application might also support a page for autentificaton and every client should have their own account and should be able too see only their information and their orders. Also, a more advanced GUI could be implemented.

1. **Bibliography**

* <https://didatec.sharepoint.com/sites/PT_2_Eng_Spring_2021/Class%20Materials/Laboratory/Assignment%203/ASSIGNMENT_3_SUPPORT_PRESENTATION.pdf?CT=1622395402932&OR=ItemsView>
* <https://bitbucket.org/utcn_dsrl/pt-reflection-example/src/master/>
* <https://www.baeldung.com/javadoc>
* <https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html>
* <http://tutorials.jenkov.com/java-reflection/index.html>
* <https://www.baeldung.com/java-jdbc>
* <http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>
* https://www.javatpoint.com/java-date-to-string