# Fundamental Programming Techniques

# 2021

Assignment 3

Order management

Java application

Chereji Iulia-Adela

Group 30424-1

# CONTENTS

1. Assignment objective
2. Problem analysis
3. Design decisions
4. Implementation
5. Conclusions
6. Bibliography

# 1. ASSIGNMENT OBJECTIVE

## 1.1 Main objective

Design and implement an application for processing client orders for a warehouse. Relational databases are used to store the products, the clients and the orders.

## 1.2 Secondary objectives

### 1.2.1 Analyze the problem and identify requirements

The modeling of the problem will be performed (i.e. identify the functional and non-functional requirements, identify the scenarios, detail and analyze the use cases). – Ch. 2

### 1.2.2 Design the queues simulator

The OOP design decisions will be presented (i.e. UML diagrams, data structures, class design, relationships, packages, user interfaces). –Ch. 3

### 1.2.3 Implement the queues simulator

The implementation of the classes (i.e. fields and important methods) and of the graphical user interface will be presented. –Ch. 4

# 2. PROBLEM ANALYSIS

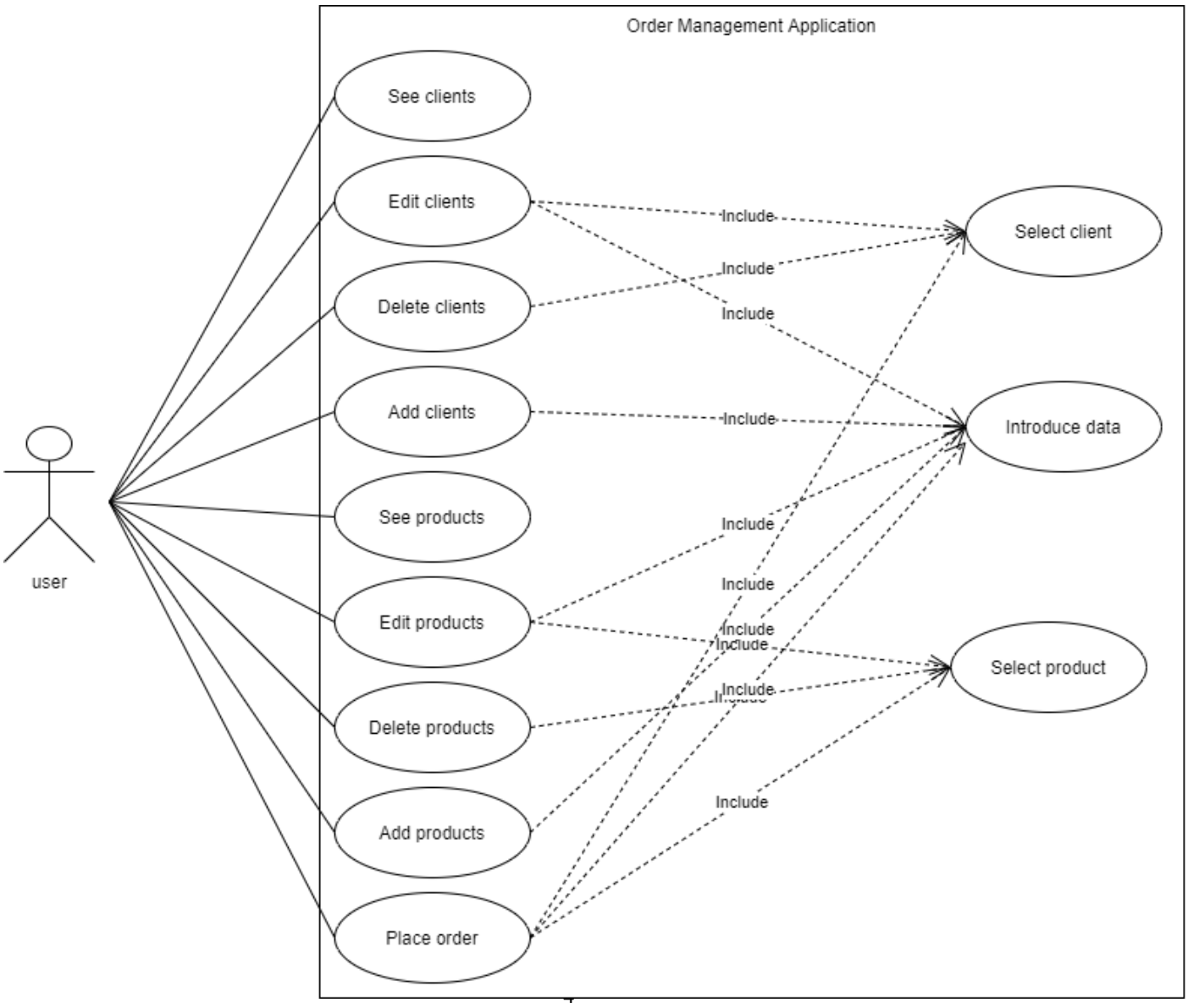
## 2.1 Functional requirements

- The application should allow users to add, delete, edit and see the clients.
- The application should allow users to add, delete, edit and see the products.
- The application should allow users to make orders by selecting the client and selecting the products and the quantities.
- The application should make the connection with the database and display its contents correctly all the time.
- The application should make the proper modifications in the database and show the results.
- The application should be documented using JavaDoc files.
- The application should have three windows for making clients operations, product operations, and orders operations.
- The application should use reflection techniques to populate the data tables and to work with the database.
- The application should use the layered architectural pattern and have at least the 4 packages: dataAccessLayer, businessLayer, model and presentation.
- The application should create a bill for each order as a .txt file.

2.2 Non-functional requirements

- The application should be intuitive and easy to use by the user.
- The application should display an error message if one of the inputs is incorrect or in the wrong format for adding or editing a client (name containing special characters, wrong email format, wrong phone format, wrong date of birth format).
- The application should display an error message if one of the inputs is incorrect or in the wrong format for adding or editing a product (wrong price or stock).
- The application should display an error message if one of the inputs is incorrect or in the wrong format for making an order (quantity not a valid integer or greater than existing stock of the product).
- The application should display an error message if the user wants to edit or delete a product without selecting it.
- The application should display an error message if the user wants to edit or delete a client without selecting it.

2.3 Use cases



2.3.1 Use Case: See clients

Primary Actor: user

Main Success Scenario:

1. The user clicks on the “CLIENTS” button in the graphical user interface and sees the clients in a table.

2.3.2 Use Case: Edit clients

Primary Actor: user

Main Success Scenario:

1. The user clicks on the “CLIENTS” button in the graphical user interface and sees the clients in a table.
2. The user clicks on a row in the table to select a client.
3. The user clicks on the “EDIT CLIENT” button.
4. The user inserts new data into the fields.
5. The user clicks on the “SAVE” button.

Alternative Sequence 1: Client not selected

1. The user clicks on the “CLIENTS” button in the graphical user interface and sees the clients in a table.
2. The user clicks on the “EDIT CLIENT” button.
3. The application displays an error message.
4. The scenario returns to step 1.

Alternative Sequence 2: Incorrect data

1. The user clicks on the “CLIENTS” button in the graphical user interface and sees the clients in a table.
2. The user clicks on a row in the table to select a client.
3. The user clicks on the “EDIT CLIENT” button.
4. The user inserts incorrect data.
5. The user clicks on the “SAVE” button.
6. The application displays an error message.
7. The scenario returns to step 1.

2.3.3 Use Case: Delete clients

Primary Actor: user

Main Success Scenario:

1. The user clicks on the “CLIENTS” button in the graphical user interface and sees the clients in a table.
2. The user clicks on a row in the table to select a client.
3. The user clicks on the “DELETE CLIENT” button.

Alternative Sequence: Client not selected

1. The user clicks on the “CLIENTS” button in the graphical user interface and sees the clients in a table.

2. The user clicks on the “DELETE CLIENT” button.
3. The application displays an error message.
4. The scenario returns to step 1.

2.3.4 Use Case: Add clients

Primary Actor: user

Main Success Scenario:

1. The user clicks on the “CLIENTS” button in the graphical user interface and sees the clients in a table.
2. The user clicks on the “ADD CLIENT” button.
3. The user inserts new data into the fields.
4. The user clicks on the “SAVE” button.

Alternative Sequence 1: Incorrect data

1. The user clicks on the “CLIENTS” button in the graphical user interface and sees the clients in a table.
2. The user clicks on the “ADD CLIENT” button.
3. The user inserts incorrect data.
4. The user clicks on the “SAVE” button.
5. The application displays an error message.
6. The scenario returns to step 1.

2.3.5 Use Case: See products

Primary Actor: user

Main Success Scenario:

1. The user clicks on the “PRODUCTS” button in the graphical user interface and sees the products in a table

2.3.6 Use Case: Edit products

Primary Actor: user

Main Success Scenario:

1. The user clicks on the “PRODUCTS” button in the graphical user interface and sees the products in a table.
2. The user clicks on a row in the table to select a product.
3. The user clicks on the “EDIT PRODUCT” button.
4. The user inserts new data into the fields.
5. The user clicks on the “SAVE” button.

Alternative Sequence 1: Product not selected

1. The user clicks on the “PRODUCTS” button in the graphical user interface and sees the products in a table.
2. The user clicks on the “EDIT PRODUCT” button.
3. The application displays an error message.
4. The scenario returns to step 1.

Alternative Sequence 2: Incorrect data

1. The user clicks on the “PRODUCTS” button in the graphical user interface and sees the products in a table.
2. The user clicks on a row in the table to select a product.
3. The user clicks on the “EDIT PRODUCT” button.
4. The user inserts incorrect data.
5. The user clicks on the “SAVE” button.
6. The application displays an error message.
7. The scenario returns to step 1.

2.3.7 Use Case: Delete products

Primary Actor: user

Main Success Scenario:

1. The user clicks on the “PRODUCTS” button in the graphical user interface and sees the products in a table.
2. The user clicks on a row in the table to select a product.
3. The user clicks on the “DELETE PRODUCT” button.

Alternative Sequence: Product not selected

1. The user clicks on the “PRODUCTS” button in the graphical user interface and sees the products in a table.
2. The user clicks on the “DELETE PRODUCT” button.
3. The application displays an error message.
4. The scenario returns to step 1.

2.3.8 Use Case: Add products

Primary Actor: user

Main Success Scenario:

5. The user clicks on the “PRODUCTS” button in the graphical user interface and sees the products in a table.
6. The user clicks on the “ADD PRODUCT” button.
7. The user inserts new data into the fields.
8. The user clicks on the “SAVE” button.

Alternative Sequence: Incorrect data

7. The user clicks on the “PRODUCTS” button in the graphical user interface and sees the products in a table.
8. The user clicks on the “ADD PRODUCT” button.
9. The user inserts incorrect data.
10. The user clicks on the “SAVE” button.
11. The application displays an error message.
12. The scenario returns to step 1.

2.3.9 Use Case: Place order

Primary Actor: user

Main Success Scenario:

1. The user clicks on the “ORDERS” button in the graphical user interface.
2. The user clicks on the combo box to select a client.
3. The user clicks on the combo box to select a product.
4. The user inserts a quantity.
5. The user clicks on the “ADD ITEM” button.
6. The user clicks on the “PLACE ORDER” button.

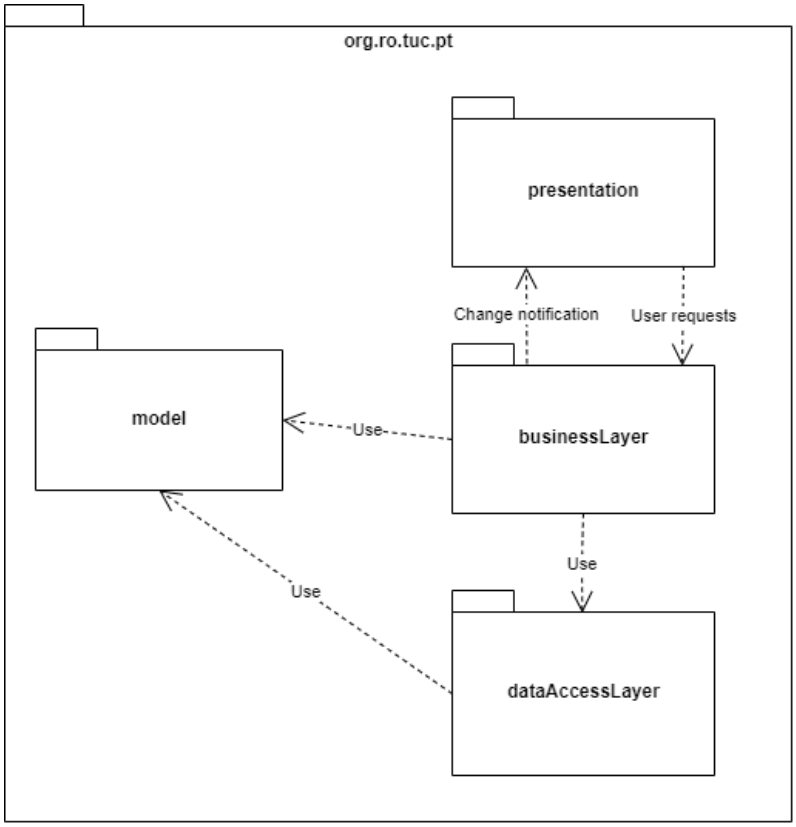
Alternative Sequence: Incorrect quantity format or quantity larger than stock

1. The user clicks on the “ORDERS” button in the graphical user interface.
2. The user clicks on the combo box to select a client.
3. The user clicks on the combo box to select a product.
4. The user inserts a wrong quantity.
5. The user clicks on the “ADD ITEM” button.
6. The application displays an error message.
7. The scenario returns to step 1.

# 3.DESIGN DECISIONS

## 3.1 Division into packages

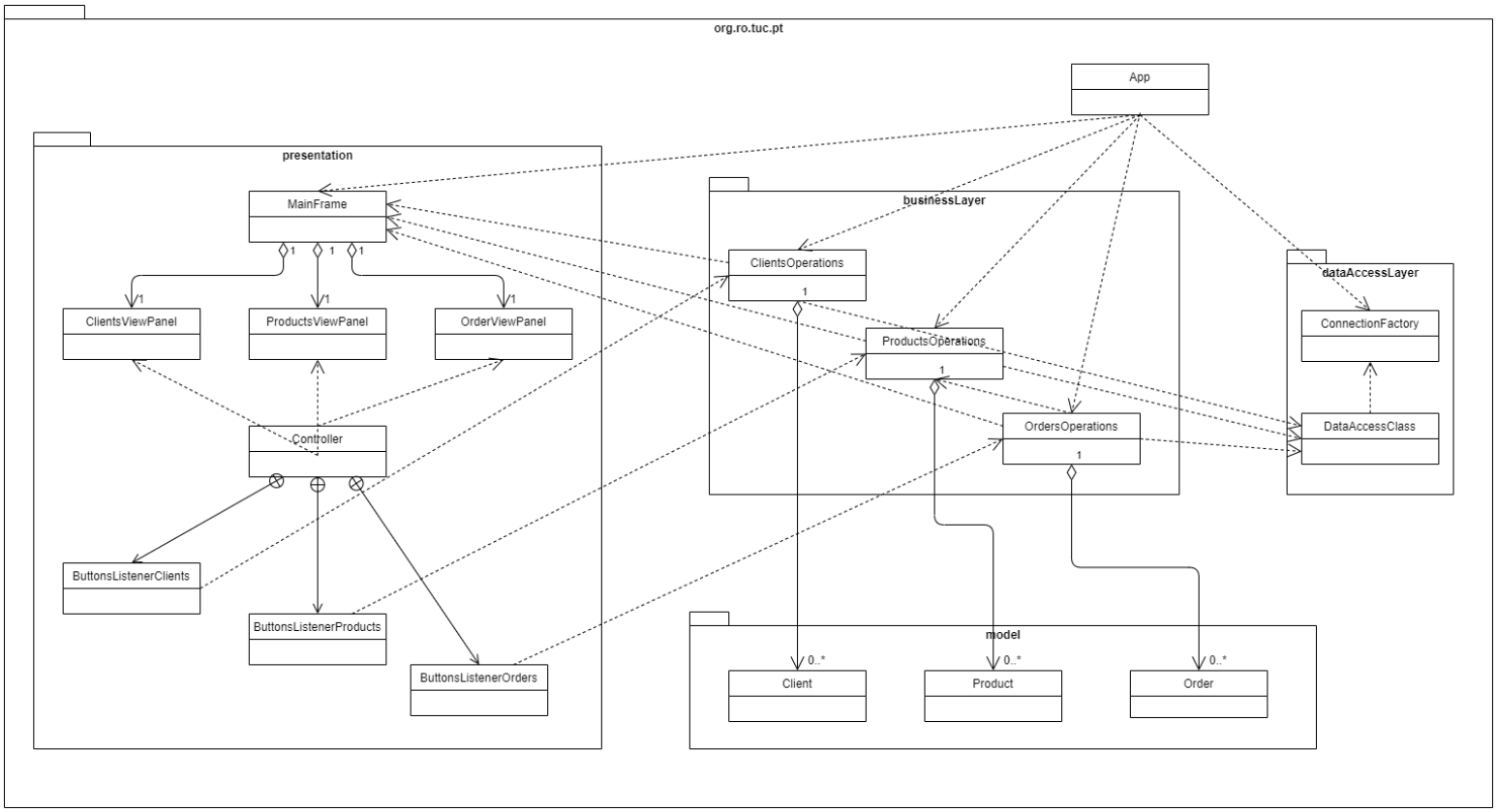
The Layered Architectural Pattern was used.



- model package contains classes mapped to the database tables.
- presentation package contains the classes defining the user interface.
- businessLayer package contains the classes that encapsulate the application logic.
- dataAccessLayer package contains the classes containing the queries and the database connection.

## 3.2 Division into classes

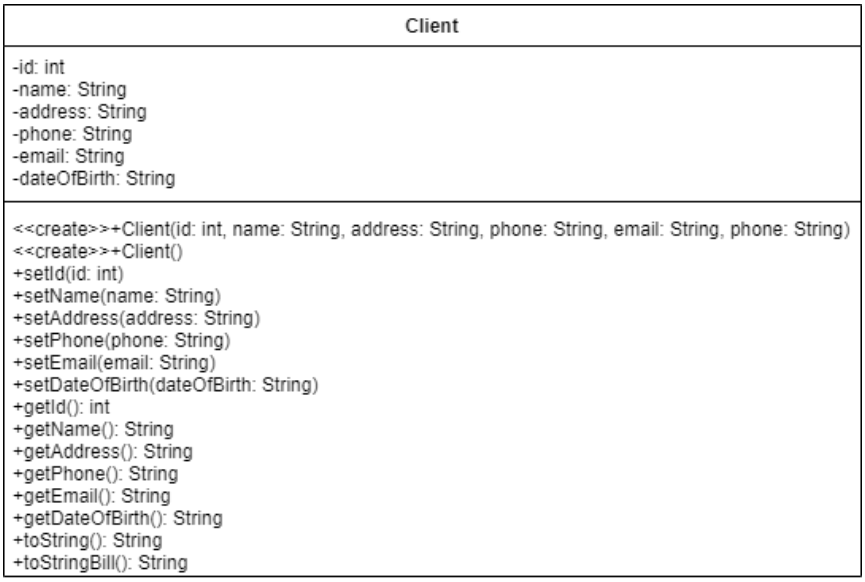
### 3.2.1 Class diagram



# 4. IMPLEMENTATION

## 4.1 Class descriptions

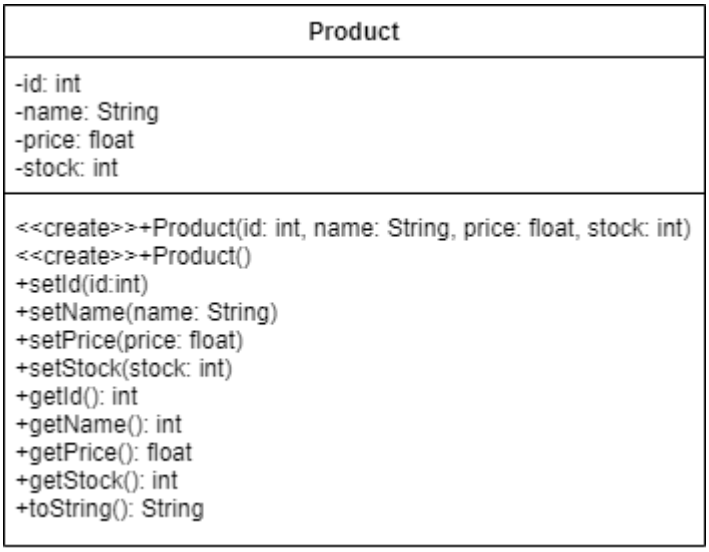
### 4.1.1 Class Client




Class Client is used to hold client's data and it is mapped to a dataBase table. The toString method is used for viewing a client in the drop box when making an order, and the toStringBill method is used when printing the order information to the bill text file.

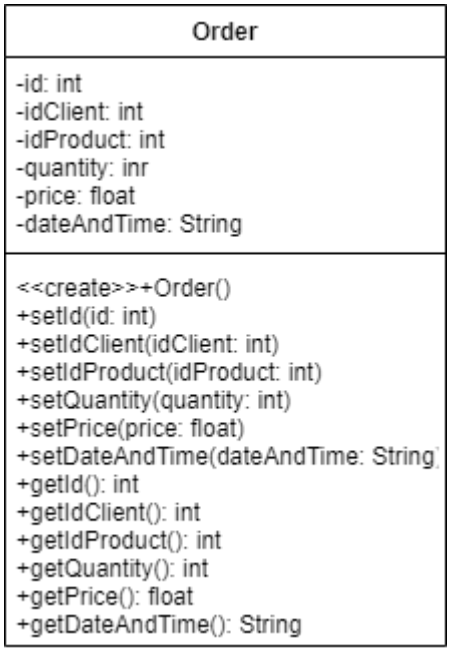
### 4.1.2 Class Product

Class Product is used to hold product's data and it is mapped to a dataBase table.



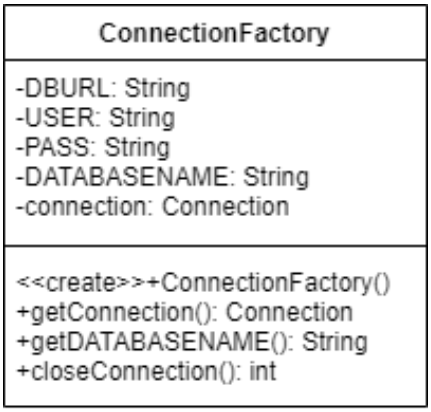
### 4.1.3 Class Order

Class Order is used to hold order's data and it is mapped to a dataBase table.




### 4.1.4 Class ConnectionFactory

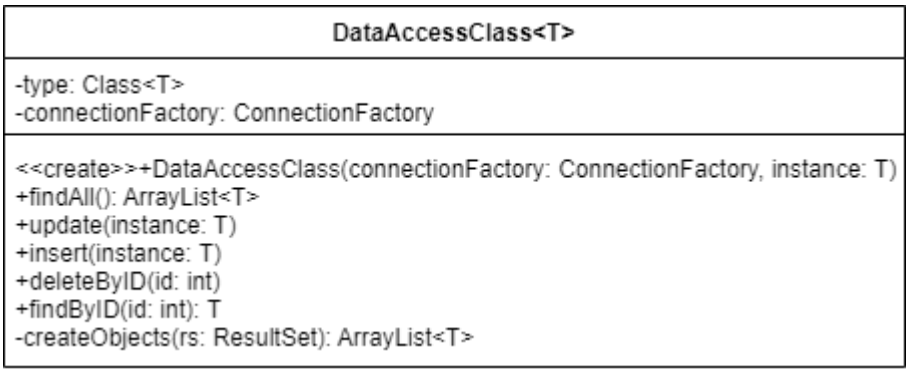
Class ConnectionFactory is used for creating the connection with the database.




```
public ConnectionFactory()
{
    try{
        connection= DriverManager.getConnection(DBURL,USER,PASS);
    }
    catch (SQLException exception) {

        System.out.println("An error occurred while connecting SQL
database");
        exception.printStackTrace();
    }
}

public int closeConnection()
{
    try{
        if(connection!=null && !connection.isClosed())
        {
            connection.close();
            return 0;
        }
        return -1;
    }
    catch(SQLException e)
    {
```

```
            System.out.println("Error closing connection\n");
            return -1;
        }
    }
```

### 4.1.5 Class DataAccessClass

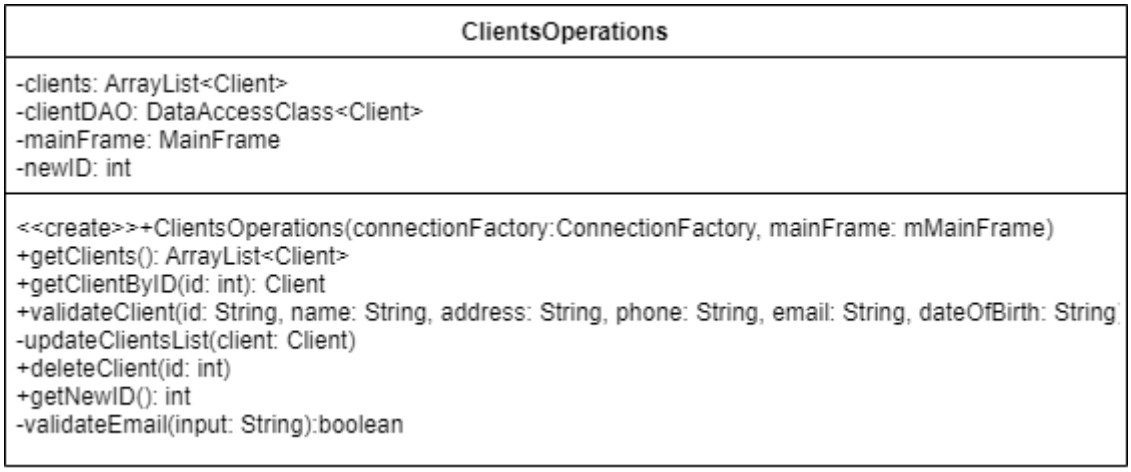



The class DataAccessClass is a generic class that uses reflection to create SQL queries for accessing the database.

The findAll method returns an arraylist with all the entries in the table.

```
public ArrayList<T> findAll()
{
    ResultSet rs = null;
    Statement st=null;
    try {
        st = connectionFactory.getConnection().createStatement();
        rs = st.executeQuery("SELECT * from " +
connectionFactory.getDATABASENAME() + "." + type.getSimpleName());
        return createObjects(rs);
    }
    catch (SQLException e)
    {
        System.out.println("ERROR trying to findAll\n");
        e.printStackTrace();
        return null;
    }
    finally {
       try{ if(rs!=null) rs.close(); if(st!=null) st.close();}
       catch(SQLException e){ System.out.println("ERROR closing\n");
e.printStackTrace();}
    }
}
```

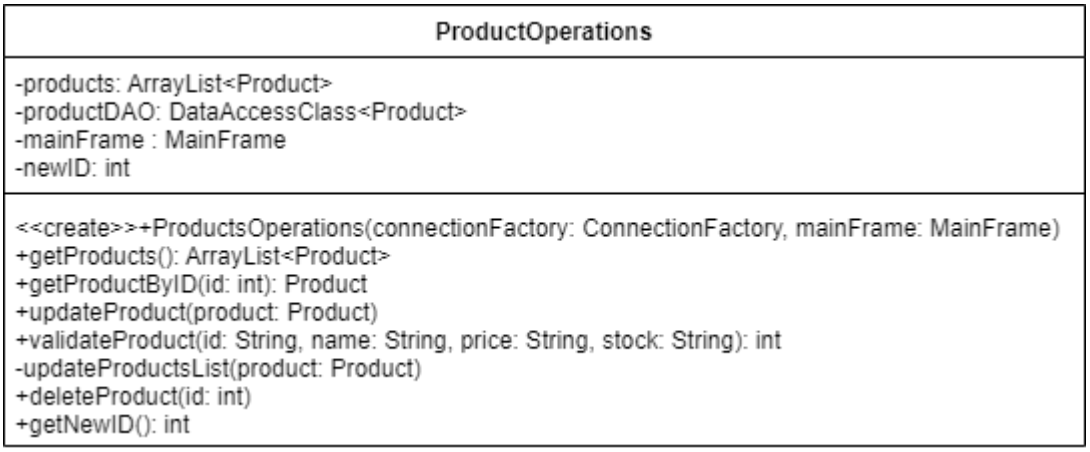
### 4.1.6 Class ClientsOperations



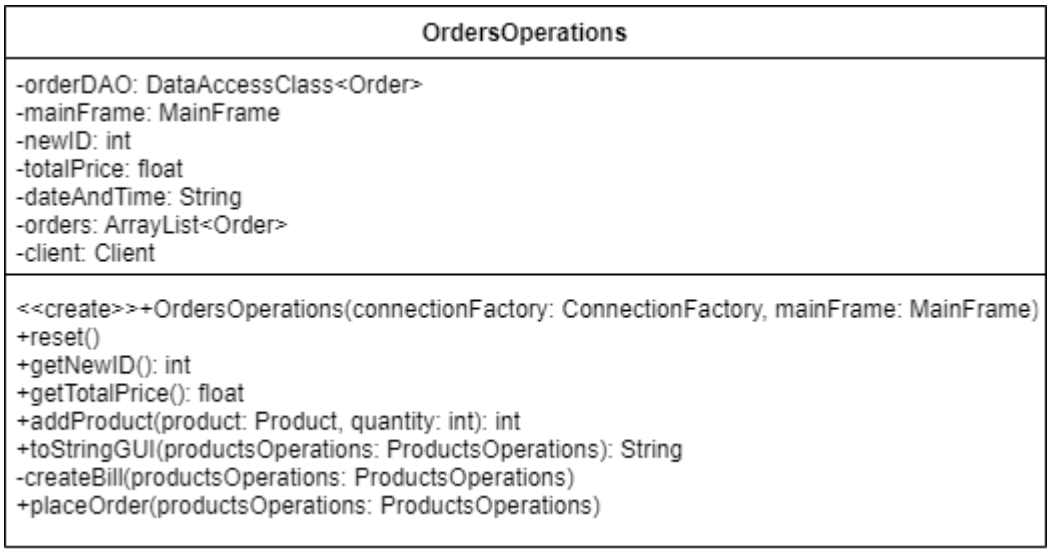

Class ClientsOperations performes the operations on clients. It holds the clients in the clients field and uses clientDAO to communicate with the database. Field newID is the id that will be assigned to the next client that will be added.

When changes are made, the clients table, database, and clients combo box are updated.

### 4.1.7 Class ProductsOperations



### 4.1.8 Class OrdersOperations

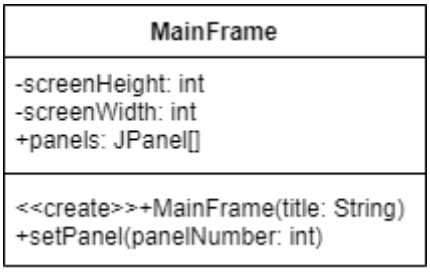



```
public int addProduct(Product product, int quantity)
{
    int i=0;
    while(i<orders.size() &&
orders.get(i).getIdProduct()!=product.getId())
        i++;

    int q;
    if(i==orders.size()) //this item is not here
    {
        if(product.getStock()<quantity) return -1;
        Order order= new Order();
        order.setIdProduct(product.getId());
        order.setQuantity(quantity);
        order.setPrice(product.getPrice()*quantity);
        orders.add(order);
        totalPrice+=product.getPrice()*quantity;
        return 0; //ok
    }
    if((q=orders.get(i).getQuantity()+quantity)>product.getStock())
return -1;
    //we can add it
    totalPrice=totalPrice- orders.get(i).getPrice();
    orders.get(i).setQuantity(q);
    orders.get(i).setPrice(product.getPrice()*q);
    totalPrice=totalPrice+ orders.get(i).getPrice();
    return 0; //ok
}

public void placeOrder(ProductsOperations productsOperations)
{
    long millis= System.currentTimeMillis();
    java.sql.Date date = new java.sql.Date(millis);
    java.sql.Time time = new java.sql.Time(millis);
    dateAndTime= date + " " + time;
    client =
(Client)(((OrderViewPanel)mainFrame.panels[2]).clientsComboBox.getSelec
tedItem());
    int idClient = client.getId();
    for(int i=0;i< orders.size();i++)
    {
        Order order= orders.get(i);
        order.setDateAndTime(dateAndTime);
```

```
            order.setId(newID);
            order.setIdClient(idClient);
            orderDAO.insert(order);
            Product product =
productsOperations.getProductByID(order.getIdProduct());
            product.setStock(product.getStock()-order.getQuantity());
            productsOperations.updateProduct(product);
            createBill(productsOperations);
        }
        reset();
        newID++;
    }

    private void createBill(ProductsOperations productsOperations)
    {
        File billFile = new File("Bill_"+newID+".txt");
        try {
            FileWriter fw = new FileWriter(billFile);
            PrintWriter pw = new PrintWriter(fw);
            String str= "Order with id: "+newID +"\n\nClient:
"+client.toStringBill()+"\n\nDate and time: "+dateAndTime+"\n\nTotal
price: "+totalPrice+"\n\nProducts:\n\n";
            for(int i=0;i< orders.size();i++) {
                Product product =
productsOperations.getProductByID(orders.get(i).getIdProduct());
                str=str+product.toString() +", quantity:
"+orders.get(i).getQuantity()+", total price:
"+orders.get(i).getPrice()+"\n";
            }

            pw.append(str);
            pw.close();
        } catch (IOException e) {System.out.println("ERROR writing to
bill\n"); e.printStackTrace();}
    }
```

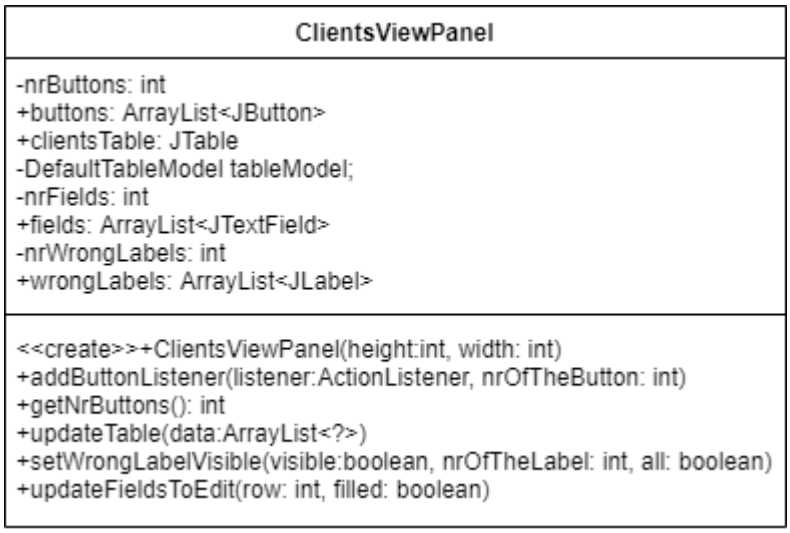
### 4.1.9 Class MainFrame

Class MainFrame extends JFrame. It Is the main window of the app and its contents change.
It creates its 3 panels: ClientsViewPanel, ProductsViewPanel and OrderViewPanel.



### 4.1.10 Class ClientsViewPanel

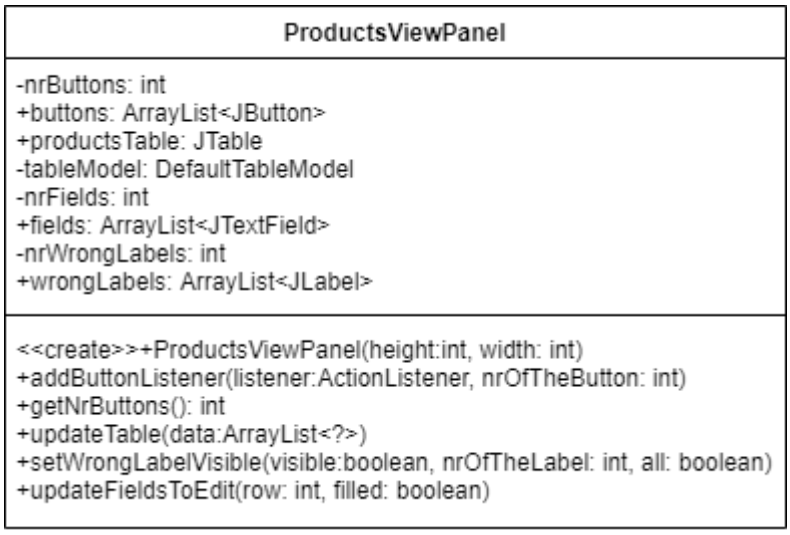
Class extends JPanel.




```
public void updateTable(ArrayList<?> data)
{
    ArrayList<String> columns=new ArrayList<>();
    Class cls = data.get(0).getClass();
    ArrayList<Method> readMethods = new ArrayList<>();

    for(Field field: cls.getDeclaredFields()) {
        columns.add(field.getName());
        try {
            PropertyDescriptor propertyDescriptor = new
PropertyDescriptor(field.getName(),cls);
            readMethods.add(propertyDescriptor.getReadMethod());
        } catch (Exception e) { System.out.println("ERROR updating
Table Clients\n"); e.printStackTrace();}
    }
    tableModel.setColumnCount(0);
    for(int i=0;i<columns.size();i++)
        tableModel.addColumn(columns.get(i));
    int nr= tableModel.getRowCount();
    for(int i=nr-1;i>=0;i--)
        tableModel.removeRow(i);
    Object[] values= new Object[columns.size()];
    try{
        for(int i=0;i< data.size();i++) {
            for(int j=0;j<readMethods.size();j++) {
                values[j] = (readMethods.get(j)).invoke(data.get(i));
            }
            tableModel.addRow(values);
        }
    } catch (Exception e) { System.out.println("ERROR updating Table
Clients\n"); e.printStackTrace(); }
    clientsTable.setModel(tableModel);
}
```

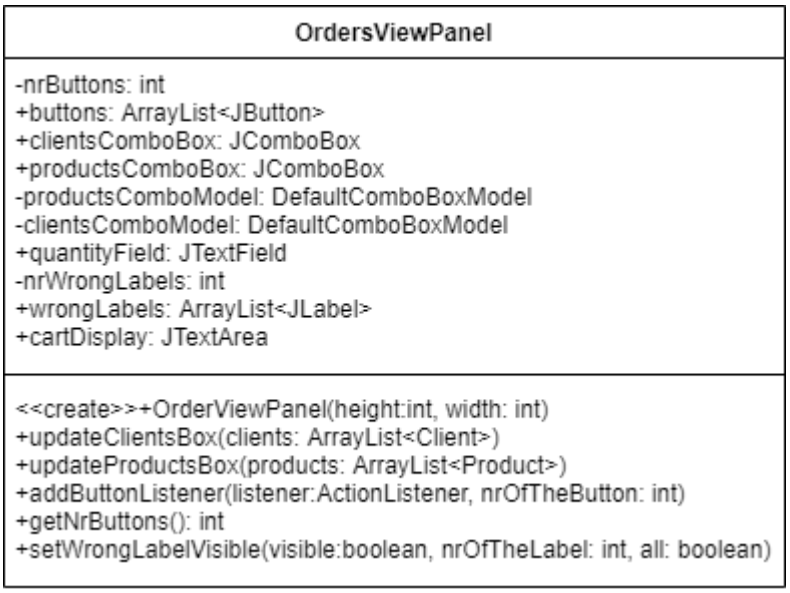
### 4.1.11 Class ProductsViewPanel

Class extends JPanel.

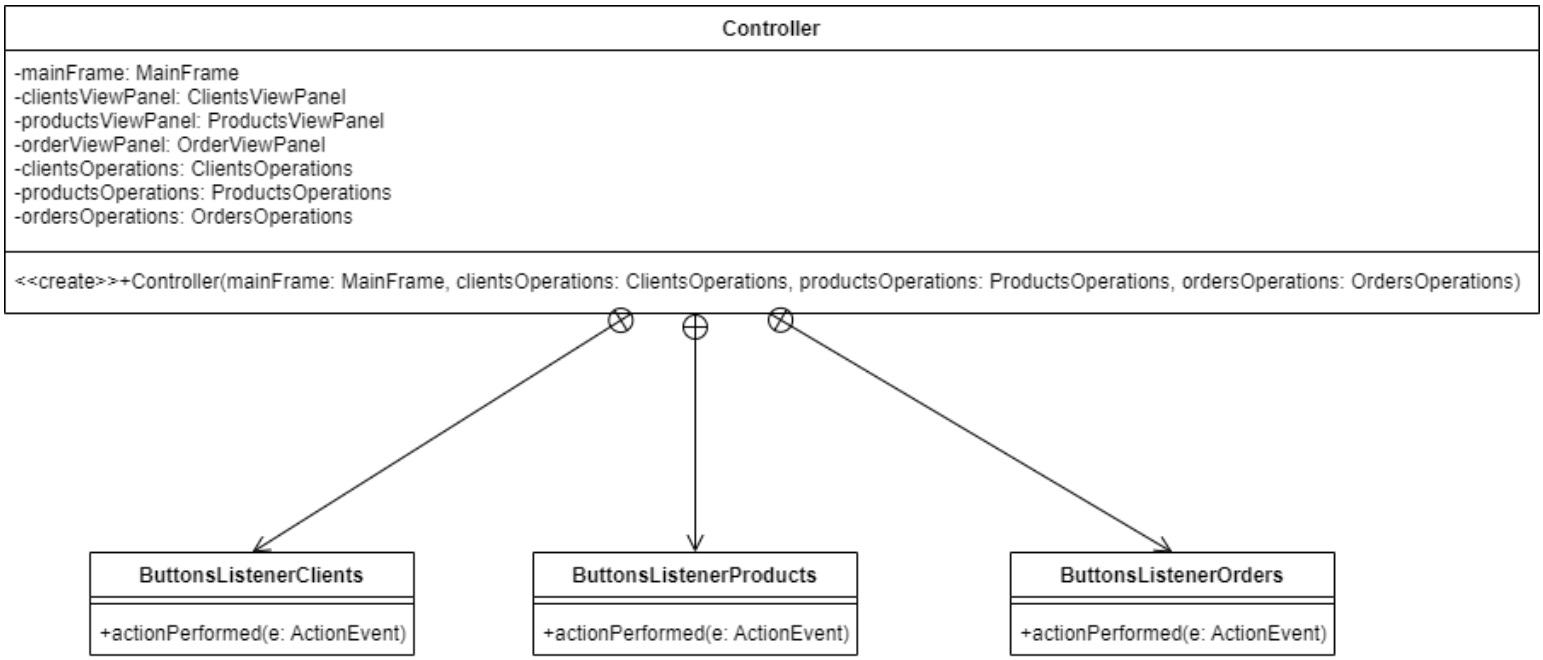



### 4.1.12 Class OrderViewPanel

Class extends JPanel. The field cartDisplay is the place where the items that were already selected will apprear.



### 4.1.13 Class Controller




### 4.1.14 Class ButtonsListenerClients

Class implements ActionListener.

```
@Override
public void actionPerformed(ActionEvent e) {
    Object event=e.getSource(); int row;
clientsViewPanel.setWrongLabelVisible(false, 0,true); int result;
    if(event==clientsViewPanel.buttons.get(0)) //clients
    {   clientsViewPanel.updateFieldsToEdit(0,false);
mainFrame.setPanel(0);}
    else if(event==clientsViewPanel.buttons.get(1)) //products
    { productsViewPanel.updateFieldsToEdit(0,false);
mainFrame.setPanel(1);}
    else if(event==clientsViewPanel.buttons.get(2)) //orders
    {   ordersOperations.reset();
orderViewPanel.cartDisplay.setText(ordersOperations.toStringGUI(produ
ctsOperations)); mainFrame.setPanel(2);}
    else if(event==clientsViewPanel.buttons.get(3)) //edit client
        if((row=clientsViewPanel.clientsTable.getSelectedRow())>=0)
            clientsViewPanel.updateFieldsToEdit(row, true);
        else clientsViewPanel.wrongLabels.get(4).setVisible(true);
    else if(event==clientsViewPanel.buttons.get(4)) //delete client
        if((row=clientsViewPanel.clientsTable.getSelectedRow())>=0)
        {
clientsOperations.deleteClient(Integer.valueOf(String.valueOf(clients
ViewPanel.clientsTable.getValueAt(row,0))));}
        else clientsViewPanel.wrongLabels.get(4).setVisible(true);
    else if(event==clientsViewPanel.buttons.get(5)) //add client

clientsViewPanel.fields.get(5).setText(Integer.toString(clientsOperat
ions.getNewID()));
    else if(event==clientsViewPanel.buttons.get(6)) //save
            if((result=
```

```
clientsOperations.validateClient(clientsViewPanel.fields.get(5).getTe
xt(), clientsViewPanel.fields.get(0).getText(),
clientsViewPanel.fields.get(1).getText(),
clientsViewPanel.fields.get(2).getText(),
clientsViewPanel.fields.get(3).getText(),
clientsViewPanel.fields.get(4).getText()))!=-1)
                clientsViewPanel.setWrongLabelVisible(true, result,
false);
            else { clientsViewPanel.setWrongLabelVisible(false, 0,
true); clientsViewPanel.updateFieldsToEdit(-1,false);}
}
```

### 4.1.15 Class ButtonsListenerProducts

Class implements ActionListener.

```
@Override
public void actionPerformed(ActionEvent e) {
    Object event=e.getSource(); int row;
productsViewPanel.setWrongLabelVisible(false, 0, true); int result;
    if(event == productsViewPanel.buttons.get(0)) //clients
    {   clientsViewPanel.updateFieldsToEdit(0,false);
mainFrame.setPanel(0);}
    else if(event == productsViewPanel.buttons.get(1)) //products
    { productsViewPanel.updateFieldsToEdit(0,false);
mainFrame.setPanel(1);}
    else if(event == productsViewPanel.buttons.get(2)) //orders
    {   ordersOperations.reset();
orderViewPanel.cartDisplay.setText(ordersOperations.toStringGUI(produ
ctsOperations)); mainFrame.setPanel(2);}
    else if(event==productsViewPanel.buttons.get(3)) //edit product
        if((row=productsViewPanel.productsTable.getSelectedRow())>=0)
            productsViewPanel.updateFieldsToEdit(row, true);
        else productsViewPanel.wrongLabels.get(3).setVisible(true);
    else if(event==productsViewPanel.buttons.get(4)) //delete product
        if((row=productsViewPanel.productsTable.getSelectedRow())>=0)
        {
productsOperations.deleteProduct(Integer.valueOf(String.valueOf(produ
ctsViewPanel.productsTable.getValueAt(row,0))));
            }
        else productsViewPanel.wrongLabels.get(3).setVisible(true);
    else if(event==productsViewPanel.buttons.get(5)) //add product

productsViewPanel.fields.get(3).setText(Integer.toString(productsOper
ations.getNewID()));
    else if(event==productsViewPanel.buttons.get(6)) //save
        if((result=
productsOperations.validateProduct(productsViewPanel.fields.get(3).ge
tText(), productsViewPanel.fields.get(0).getText(),
productsViewPanel.fields.get(1).getText(),
productsViewPanel.fields.get(2).getText()))!=-1)
            productsViewPanel.setWrongLabelVisible(true, result,
false);
        else { productsViewPanel.setWrongLabelVisible(false, 0,
true); productsViewPanel.updateFieldsToEdit(-1,false);}
}
```

### 4.1.16 Class ButtonsListenerOrders

Class implements ActionListener.

```
@Override
public void actionPerformed(ActionEvent e) {
    Object event=e.getSource();
    if(event == orderViewPanel.buttons.get(0)) //clients
    {   clientsViewPanel.updateFieldsToEdit(0,false);
mainFrame.setPanel(0);}
    else if(event == orderViewPanel.buttons.get(1)) //products
    { productsViewPanel.updateFieldsToEdit(0,false);
mainFrame.setPanel(1);}
    else if(event == orderViewPanel.buttons.get(2)) //orders
    {   ordersOperations.reset();
orderViewPanel.cartDisplay.setText(ordersOperations.toStringGUI(produ
ctsOperations)); mainFrame.setPanel(2);}
    else if(event == orderViewPanel.buttons.get(3)) //add item
    {
        int q;
        try {
            q =
Integer.parseInt(orderViewPanel.quantityField.getText());

if(ordersOperations.addProduct((Product)(orderViewPanel.productsCombo
Box.getSelectedItem()), q)!=0)
                orderViewPanel.setWrongLabelVisible(true,1, false);
            else
            {orderViewPanel.quantityField.setText("");
orderViewPanel.setWrongLabelVisible(false,0,true);
orderViewPanel.cartDisplay.setText(ordersOperations.toStringGUI(produ
ctsOperations));}
        }
        catch (NumberFormatException ex) {
orderViewPanel.setWrongLabelVisible(true,0, false); }
    }
    else if(event == orderViewPanel.buttons.get(4)) //place order
    {
        if(ordersOperations.getTotalPrice()!=0) {
            ordersOperations.placeOrder(productsOperations);

orderViewPanel.cartDisplay.setText(ordersOperations.toStringGUI(produ
ctsOperations));
        }
    }
}
```

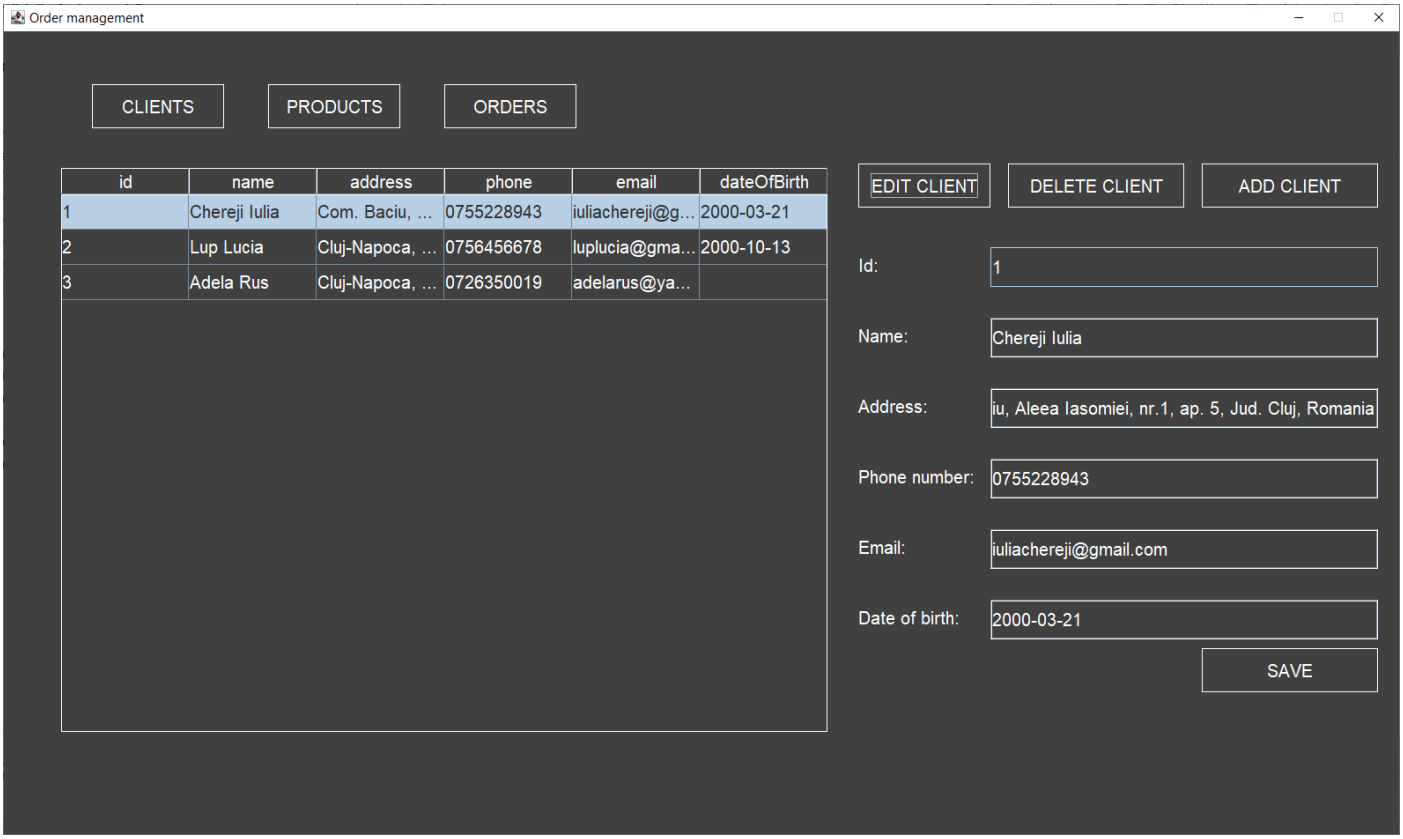
### 4.1.17 Class App

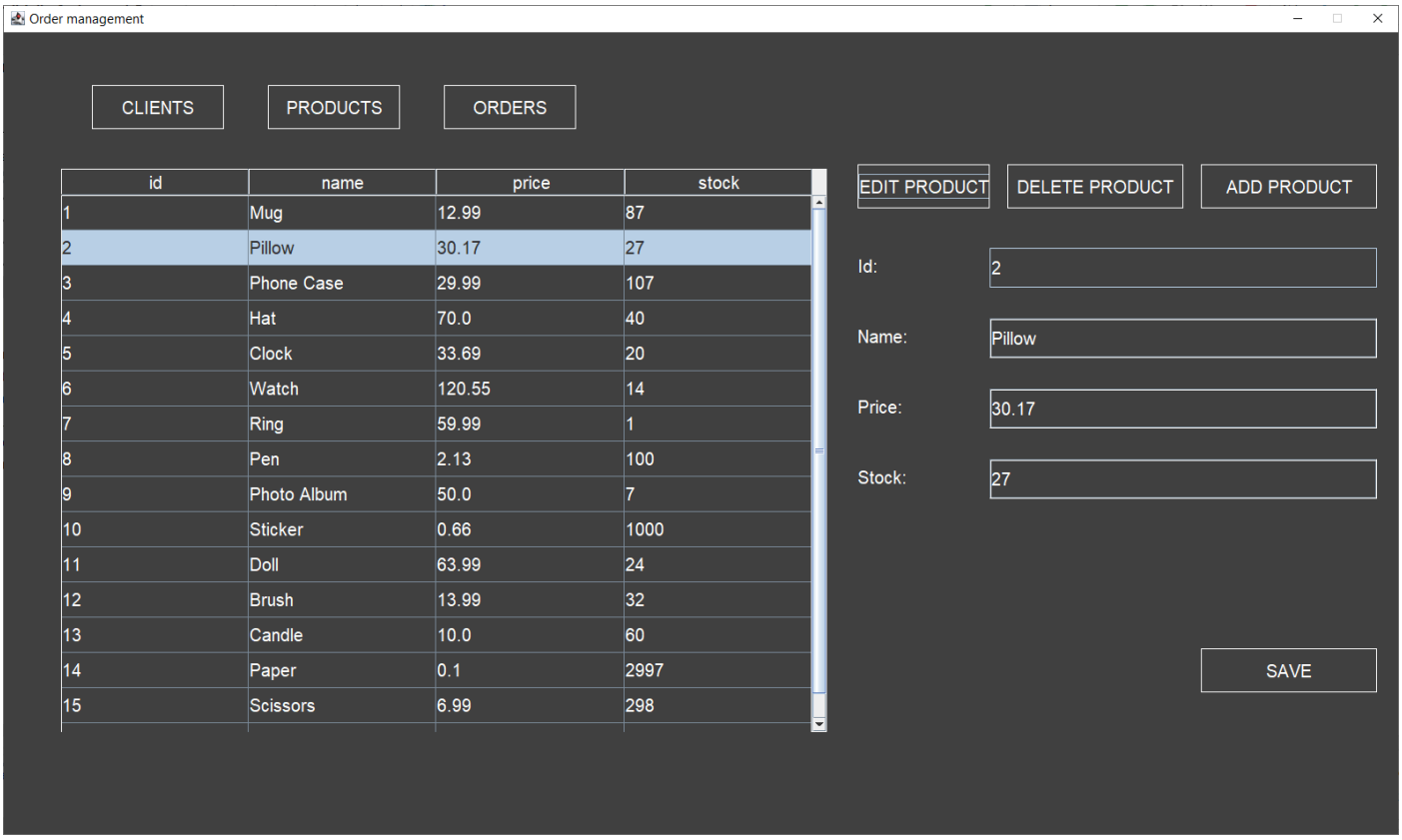

```
public static void main( String[] args )
{
    ConnectionFactory myConnection=new ConnectionFactory();
    MainFrame mainFrame= new MainFrame("Order management");
    ClientsOperations clientsOperations = new
ClientsOperations(myConnection, mainFrame);
    ProductsOperations productsOperations = new
ProductsOperations(myConnection, mainFrame);
    OrdersOperations ordersOperations = new
OrdersOperations(myConnection, mainFrame);
```

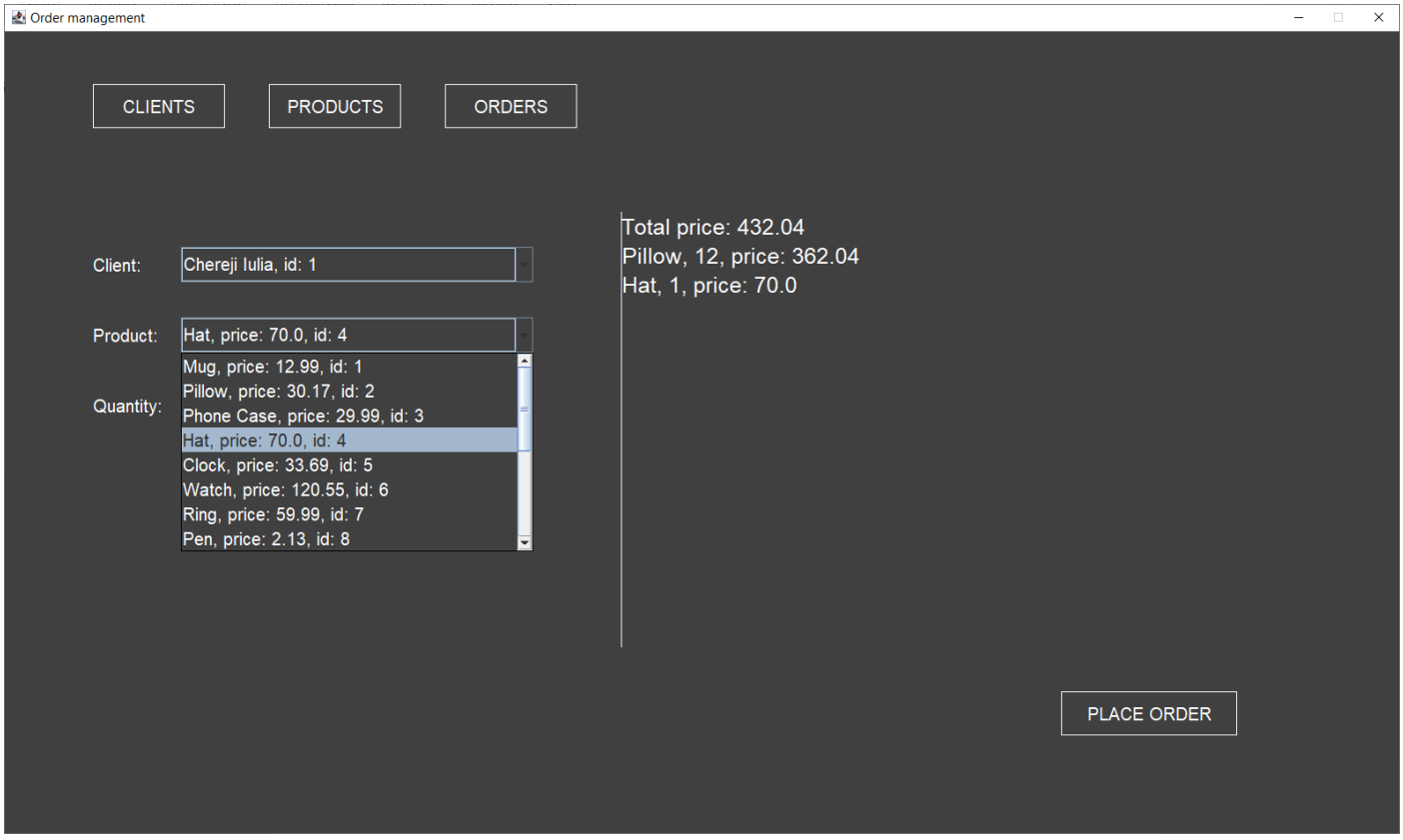
```
    Controller myController = new Controller(mainFrame,
clientsOperations, productsOperations, ordersOperations);

((ClientsViewPanel)mainFrame.panels[0]).updateTable(clientsOperations
.getClients());

((ProductsViewPanel)mainFrame.panels[1]).updateTable(productsOperatio
ns.getProducts());

((OrderViewPanel)mainFrame.panels[2]).updateClientsBox(clientsOperati
ons.getClients());

((OrderViewPanel)mainFrame.panels[2]).updateProductsBox(productsOpera
tions.getProducts());

}
```

## 4.2 GUI description

The application has 3 windows that allow for clients operations, products operations and orders operations.



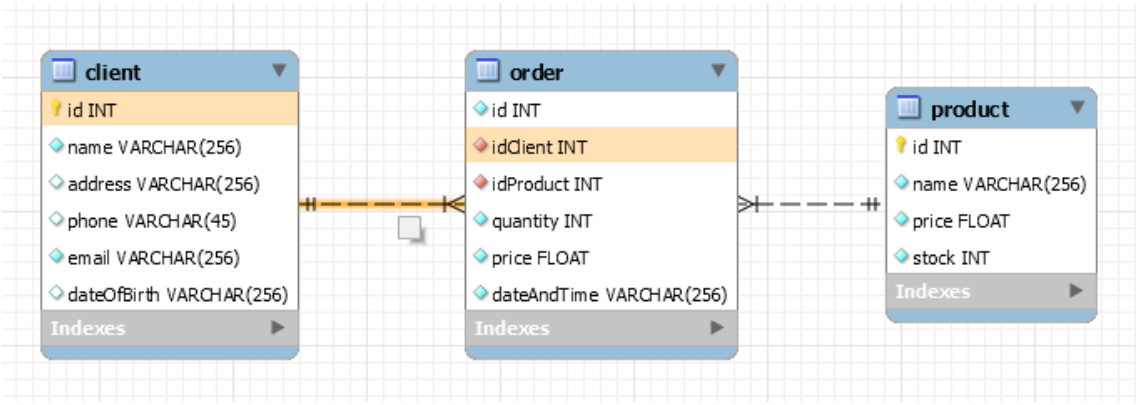

Order management
CLIENTS
PRODUCTS
ORDERS
id
name
price
stock
1
Mug
12.99
87
2
Pillow
30.17
27
3
Phone Case
29.99
107
4
Hat
70.0
40
5
Clock
33.69
20
6
Watch
120.55
14
7
Ring
59.99
1
8
Pen
2.13
100
9
Photo Album
50.0
7
10
Sticker
0.66
1000
11
Doll
63.99
24
12
Brush
13.99
32
13
Candle
10.0
60
14
Paper
0.1
2997
15
Scissors
6.99
298
EDIT PRODUCT
DELETE PRODUCT
ADD PRODUCT
Id:
2
Name:
Pillow
Price:
30.17
Stock:
27
SAVE


Order management
CLIENTS
PRODUCTS
ORDERS
Client:
Chereji Iulia, id: 1
Product:
Hat, price: 70.0, id: 4
Quantity:
Mug, price: 12.99, id: 1
Pillow, price: 30.17, id: 2
Phone Case, price: 29.99, id: 3
Hat, price: 70.0, id: 4
Clock, price: 33.69, id: 5
Watch, price: 120.55, id: 6
Ring, price: 59.99, id: 7
Pen, price: 2.13, id: 8
Total price: 432.04
Pillow, 12, price: 362.04
Hat, 1, price: 70.0
PLACE ORDER

When an order is placed, a bill text file is generated.

```
Order with id: 9

Client: name: Lup Lucia; id: 2; address: Cluj-Napoca, Str. Paltinis, nr. 10, Jud. Cluj, Romania; phone: 0756456678; email: luplucia@gmail.com

Date and time: 2021-04-17 11:19:23

Total price: 28.27

Products:

Paper, price: 0.1, id: 14, quantity: 3, total price: 0.3
Brush, price: 13.99, id: 12, quantity: 1, total price: 13.99
Scissors, price: 6.99, id: 15, quantity: 2, total price: 13.98
```

The MySQL database has the following schema, its tables matching the Client, Product and Order classes of the project.




# 5. CONCLUSIONS

## 5.1 What I have learned

- How to use maven dependency to make the connection to a database.
- How to use the Layered Architectural pattern.
- How to write a generic class.
- How to use reflection.
-

## 5.2 Future development

- The application could have a more friendly user interface with images or icons.
- The option to delete items from cart could be implemented.

- The users could have passwords required when performing an action involving him.

# 6. BIBLIOGRAPHY


- https://stackoverflow.com/questions/11067512/java-lang-class-cannot-be-cast-to-java-lang-reflect-parameterizedtype
- https://stackoverflow.com/questions/16252399/how-do-i-add-a-final-variable-to-class-diagram?fbclid=IwAR1ZDDsAQPfwjlI_3MIi0_JfGhYLCPgJPhsOkUoNlxdp28QGmek20BzqfiM
- https://stackoverflow.com/questions/4837190/java-generics-get-class/17767068?fbclid=IwAR0M11fRFw_SlhFViqzIhXUYFqx_BmrUfiWEdEIniIkKKcSI0eRKitjpqbg
- https://docs.oracle.com/javase/tutorial/java/generics/types.html?fbclid=IwAR17Zf1pwVzDBHuGWI4D_SWUd3cW01F-2hmuBGoMADQylwlWqXEiGDCbtBg
- https://stackoverflow.com/questions/27696238/sql-select-name-by-id/27696377?fbclid=IwAR1ZDDsAQPfwjlI_3MIi0_JfGhYLCPgJPhsOkUoNlxdp28QGmek20BzqfiM
- https://stackoverflow.com/questions/33900418/casting-java-arraylistcustom-class-to-arraylistobject?fbclid=IwAR2lqOGrqk-uynuFxrWlcre77umVP_CjkYVNr_-MDuLNOzYJBvcPLWpqZQ0