DOCUMENTATION

ASSIGNMENT 3

FECHETE DAVID EMANUEL
GROUP: 30423

CONTENTS

1. Assignment Objective

1.1 Main Objective

Main objective of the assignment is to design and implement an order-management application that makes it easier for a warehouse company or various other companies to manage products, clients and orders by offering software capable of storing large data about these actors as well as making a simple and effective application that replaces the hand-written registries which are difficult and time consuming.

1.2 Sub-objectives

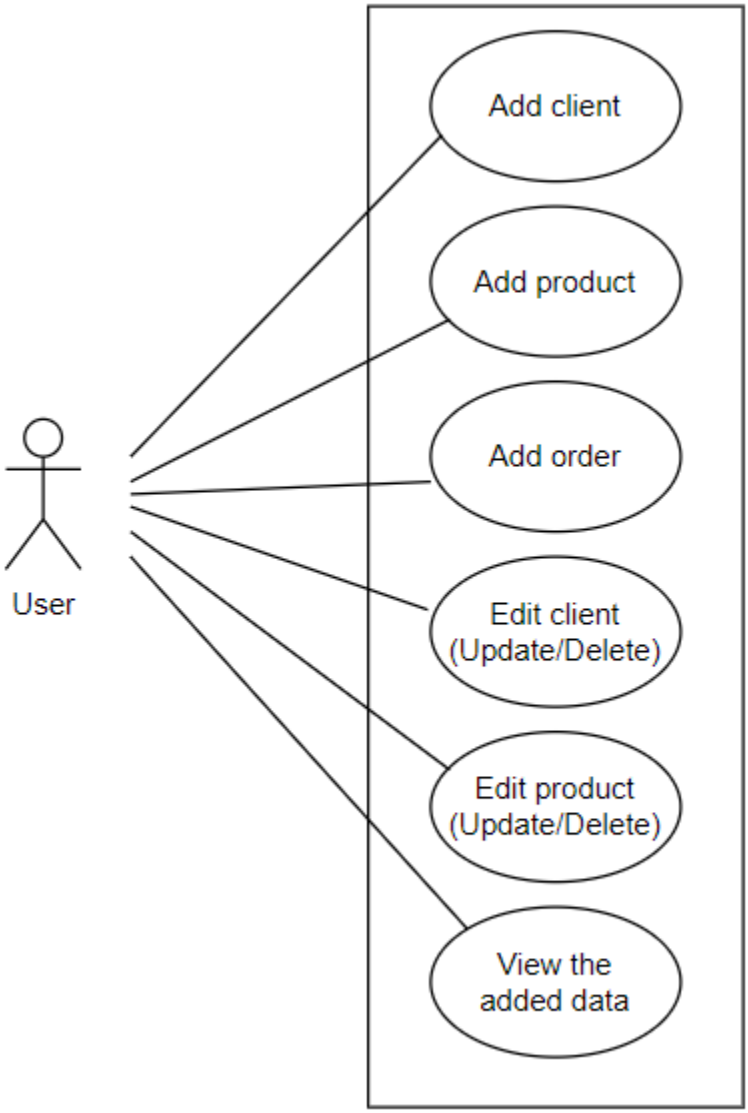
- Analyzing the problem and identify requirements (Described in section 2)
- Designing the orders management application (Described in section 3)
- Implementing the orders management application (Described in section 4)
- Testing the orders management application (Described in section 5)

2. Problem Analysis, Modeling, Scenarios, Use Cases

Use Case scenarios:

- Use Case: add or edit an object (product, client, order)
- Primary actor: employee
- Main succes scenario:
    - The employee selects the option to add a new object.
    - The application will display a form in which the object details should be inserted
    - The employee inserts the required data of the object in the text fields.
    - The employee clicks on the “Add” button
    - The application stores the object’s data in the database and displays an acknowledge message
- An alternative sequence happens whenever the employee inputs invalid values for the object’s data case in which the application displays an error message and requests the user to insert valid inputs and the scenarion resets to the steps mentioned above.
- The application should be intuitive and easy to use by the user. It should have straightforward commands and the displaying of each object should be clear and visible in order for them to be easily readable. As functional requirements the application should allow the employee to add a new client, product or order whenever they want as well as visualise any data needed by the employee at any time.

Use case diagram for our simulation application:

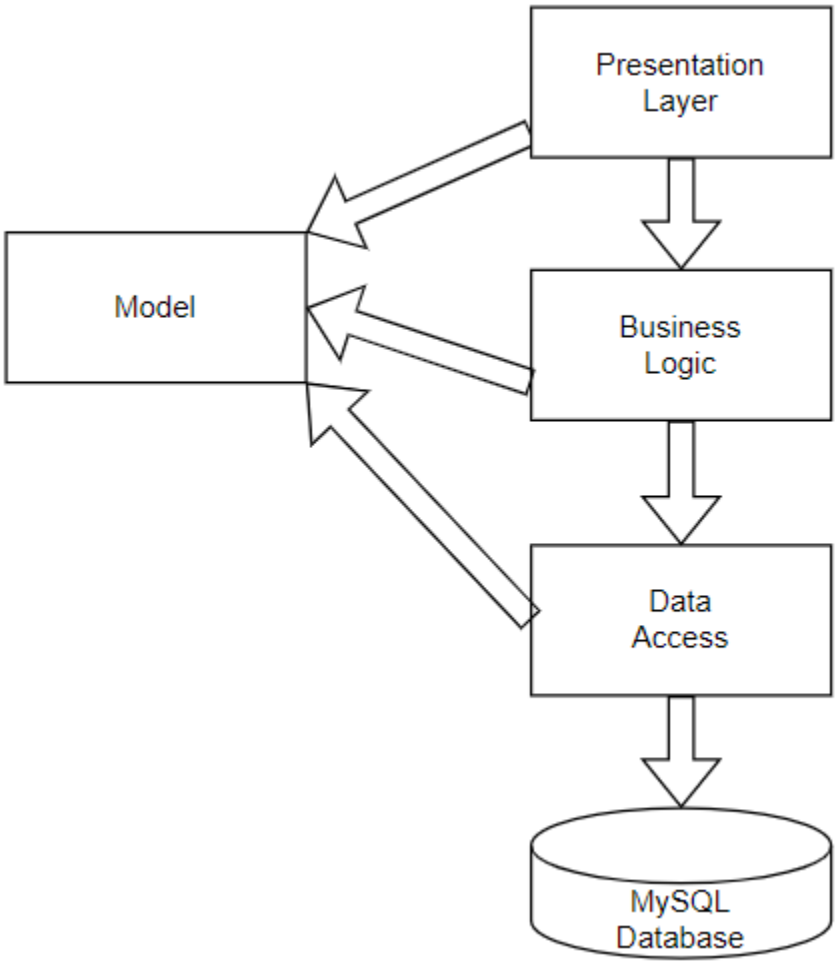


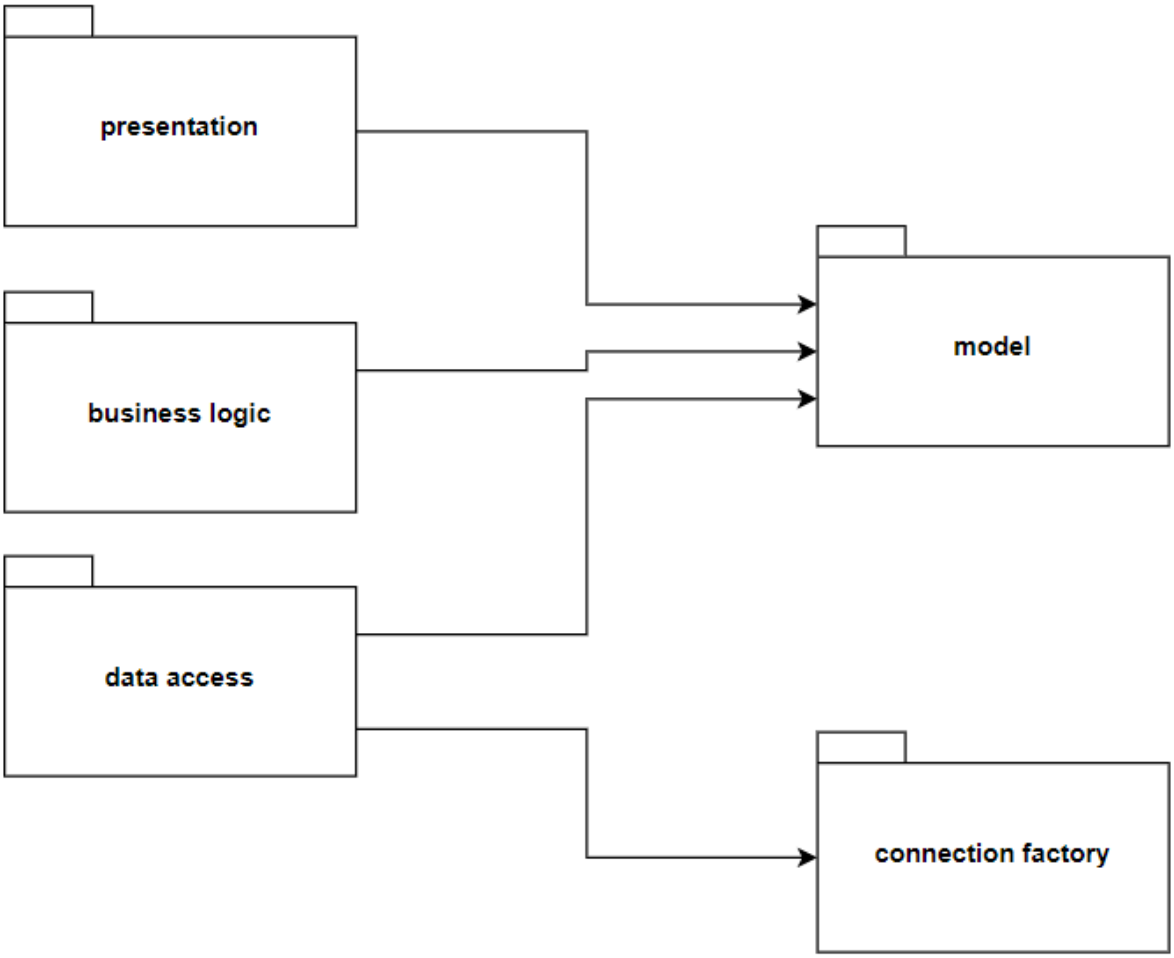
Used data structures

As data structures used the data is stored in Lists of various types and is implemented as an ArrayList which allows for easy access to any data stored in it

3. Design

Conceptual Architecture of the application:





- The presentation package contains the classes that are used to implement the GUIs and also the controller this package containing both the View and Controller packages from the MVC architecture
- The model package contains the definitions and attributes of the classes Client, Order, Product and Bill. The client holds information about its id and other useful information a client possess. The product also holds this kind of information. The Order contains the id of the order as well as the id of the product that is ordered and the id of the client that wants that product as well as the quantity of it.
- The business logic package contains the the data handling process and implements the various rules of our application, in this case it implements validators that are needed to be run through the data in order for them to be inserted into the database. This needs to be done in order for the input data to respect the various restrictions imposed.
- The data access package contains the logic behind the C-R-U-D operations and is implemented using a generic class that is very useful because it reduces the repetetive code needed for the database operations.
- The connection factory package contains the logic behind the needed connection to the database server.

Presentation (GUI and Controller)

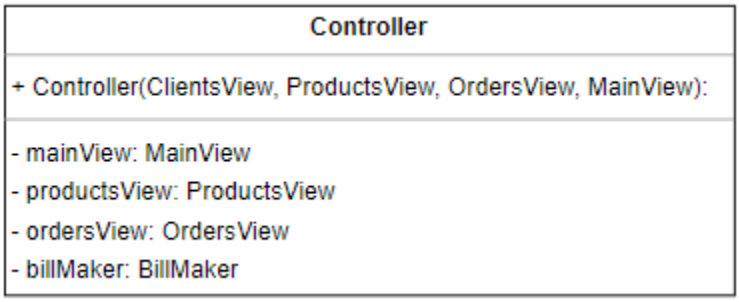
The first package contains the GUIs implementations. There are a total of 4 GUIs implemented each similar to one another. They act as canvases for the different component that are put onto them.

The classes also have listener methods used for the buttons and also methods that are used to display error messages or confirmation messages.

These are the GUI classes with every method definition. The overall class diagram will not contain the methods to offer a better visualization of the UML.

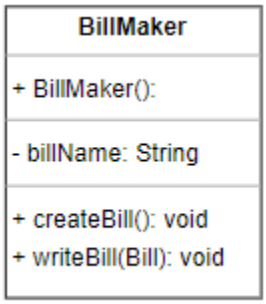
Controller

The other class in the presentation layer is the Controller which offers the main logic and implementation of the inner classes that implement the listeners for each button.



Bill Maker

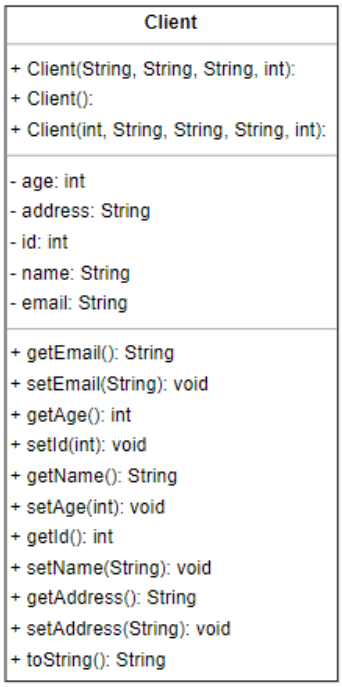
The last class from the presentation layer is the Bill Maker class which is in charge of creating the files in which each bill is stored. It contains 2 simple methods, one that creates a Bill with the time and date of the present and the other that just writes the log in the bill.



Models

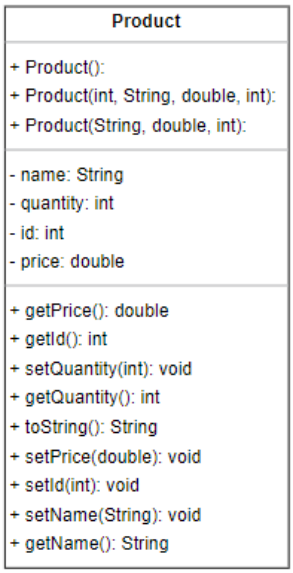
Client

The first class implemented in the Models package is the Client which is the placer of some orders. It has some basic attributes as well as a unique id which identifies it.



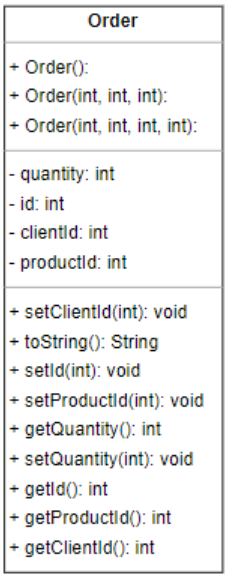
Product

This class is the other main class of the Models package which implements the sought after item. It has an unique identifier id and also other details of the product like its name, price and quantity(in stock).



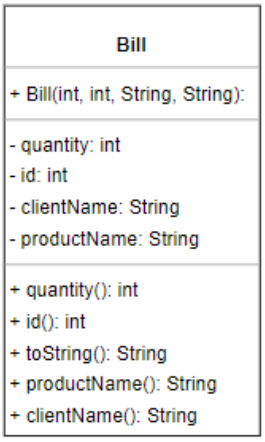
Order

The order is the most important class that has a unique identifier as well as the ids of the client that places that order and the id of the product that is ordered.



Bill

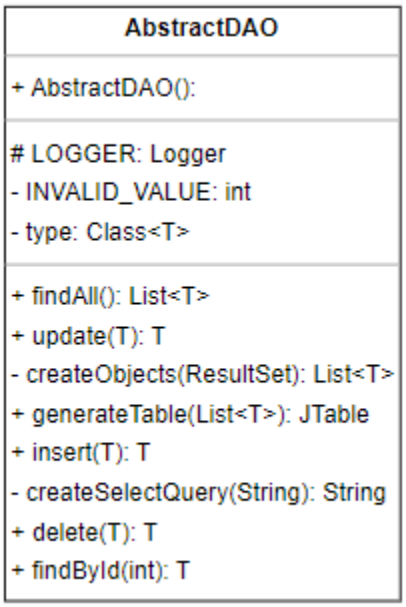
The last class from the models package is the bill which is very useful for outputting the order that is generated at each step. It is a record, meaning it's immutable and once created it can't be updated, only read. It has information about the name of the product and the client that bought it as well as the quantity of the product. It's in close relationship with the BillMaker as it has a toString() method that helps us writing to the file the data from a bill.



DAO

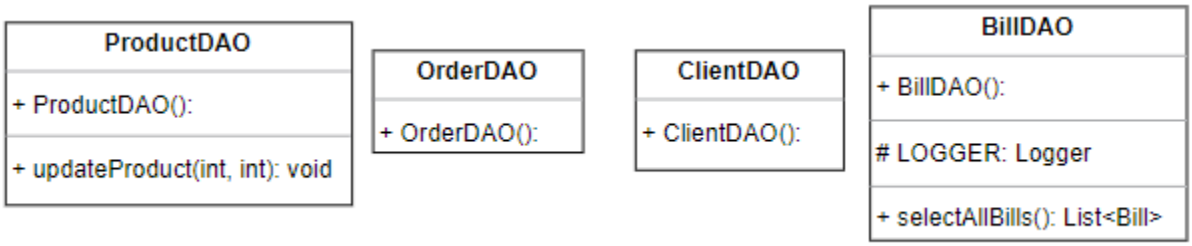
AbstractDAO

The first class present in the data access layer is the AbstractDAO class which is perhaps the most important of all classes from the application. It contains the whole logic behind the creation of the queries used to manipulate data from and to the database. It works with generics meaning that its functions work for any of the classes that are expressed in the database as well. It has methods for all 4 SQL main operations: C(insert)-R(findAll)-U(update)-D(delete).



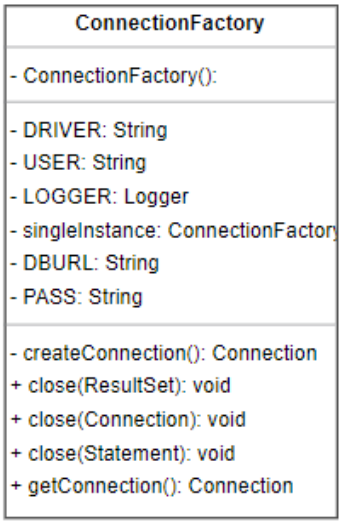
ClientDAO/BillDAO/ProductDAO/OrderDAO

There are 4 other classes in this package, all of them extending the main AbstractDAO. Only BillDAO and ProductDAO have some specific methods. BillDAO has a specific select method that works only on records while ProductDAO implements a method used for updating the products quantity in stock after an order has been placed.



Connection

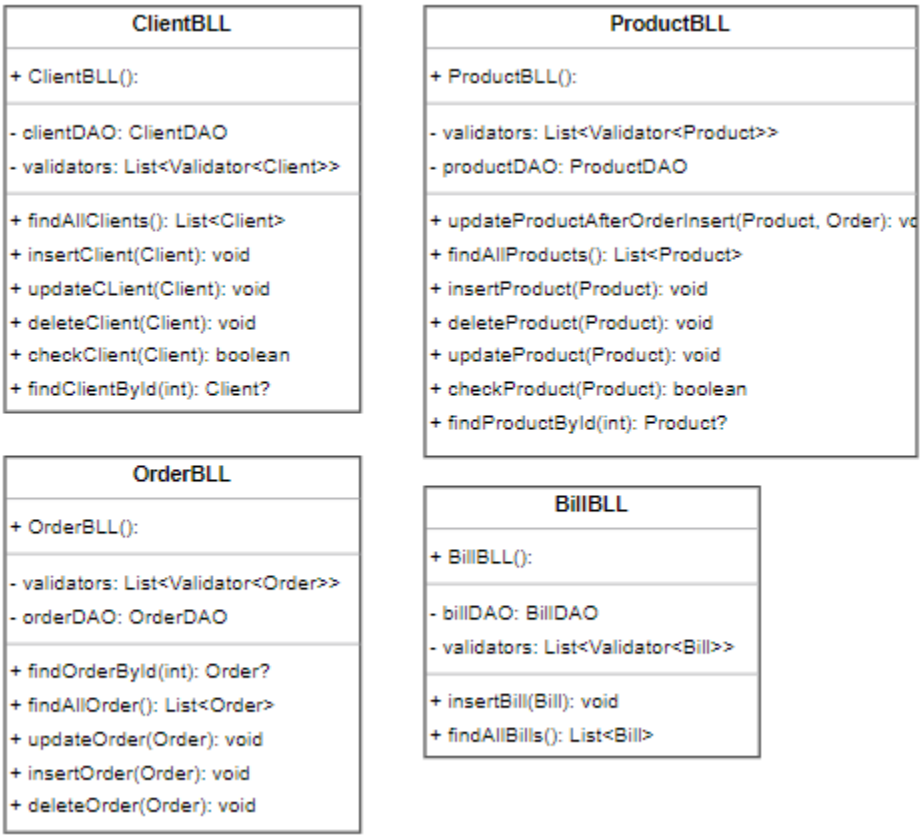
This package contains a single class: Connection Factory class which establishes the connection to the database.



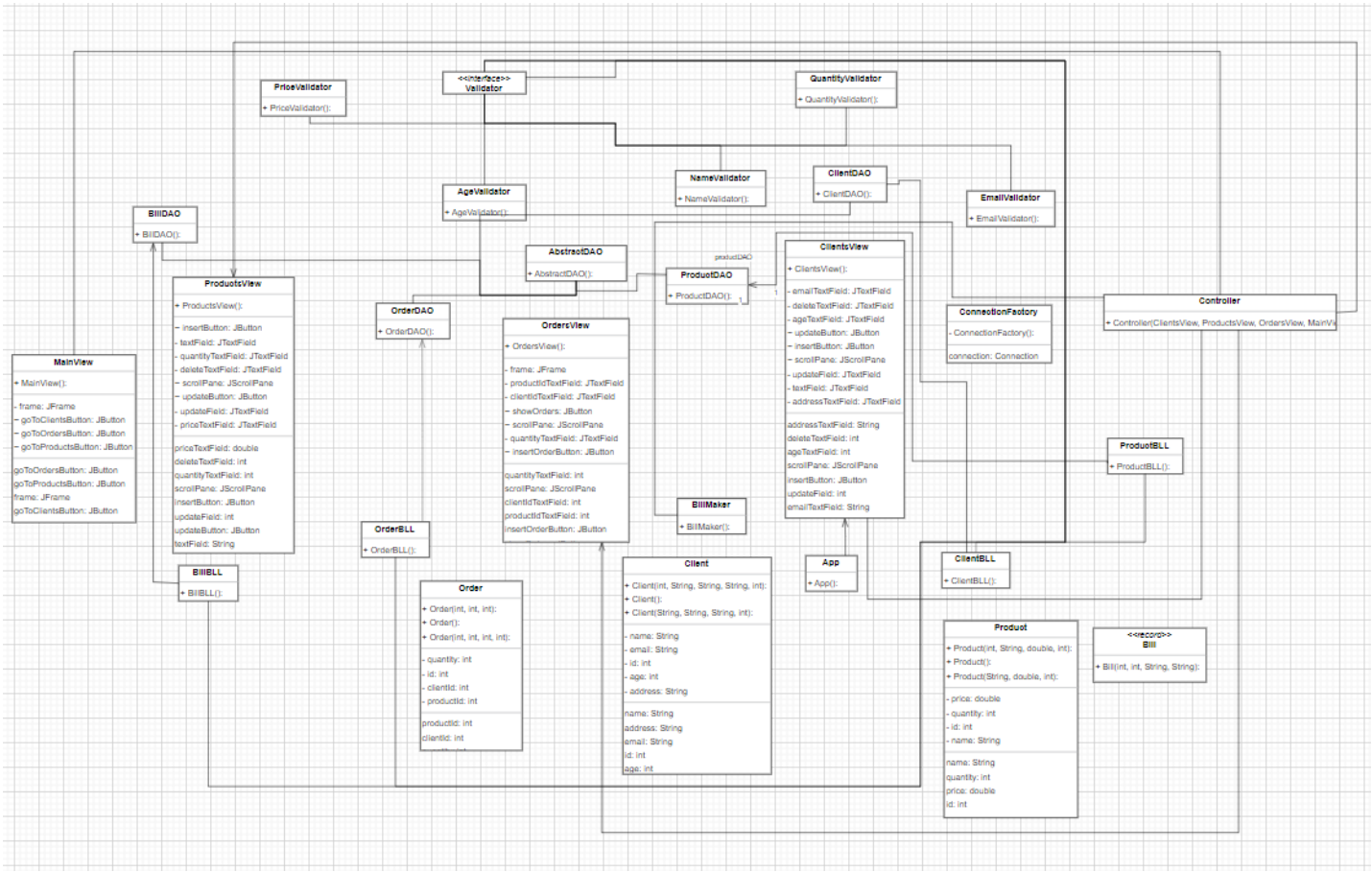
BLL

ClientBLL/ProductBLL/OrderBLL/BillBLL

The business logic level (layer) contains 4 classes and a package containing 5 more classes and an interface. This layer represents an extra check before the data from the input (GUI) is finally sent to the database. It contains validators that need to be applied to the data sent as parameters and checked whether that data corresponds to the rules imposed by the validators. These are: age must be over 18, name must not contain numbers, email must be of proper pattern, price and quantity must be over 0 in order for it to be put in the database.



This is the overall class diagram of the application simulation:

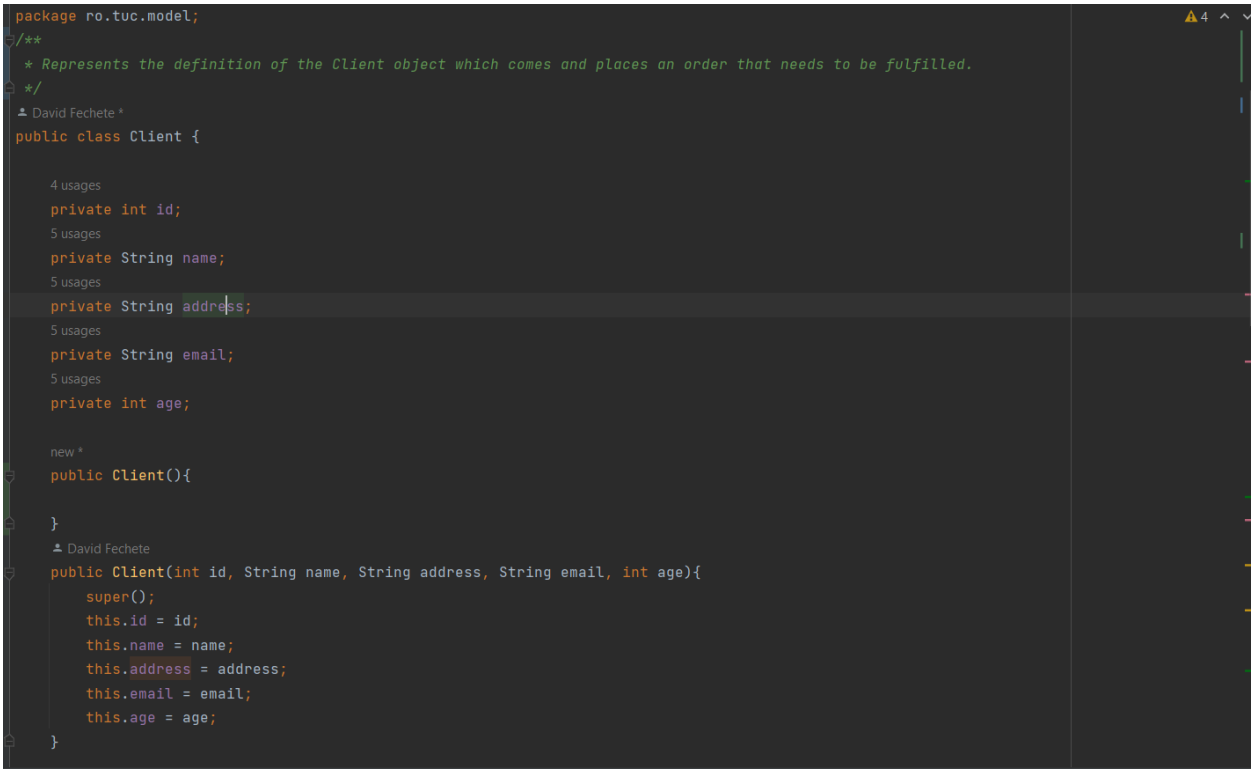


Methods have been ignored in this display of the UML Diagram for better visualization.

4. Implementation

Client

The first implemented class was the Client class that represents the main actor of the application. The client is the instance of the buyer that comes in and places and order for a product and also how much of that product he/she wants. In this way, the attributes needed for the Client object are: the id (that uniquely identifies the Client), the name, address, email and age of the client. The object has no specific methods.



```java
package ro.tuc.model;
/**
 * Represents the definition of the Client object which comes and places an order that needs to be fulfilled.
 */
David Fechete *
public class Client {

    4 usages
    private int id;
    5 usages
    private String name;
    5 usages
    private String address;
    5 usages
    private String email;
    5 usages
    private int age;

    new *
    public Client(){

    }
    David Fechete
    public Client(int id, String name, String address, String email, int age){
        super();
        this.id = id;
        this.name = name;
        this.address = address;
        this.email = email;
        this.age = age;
    }
```

Product

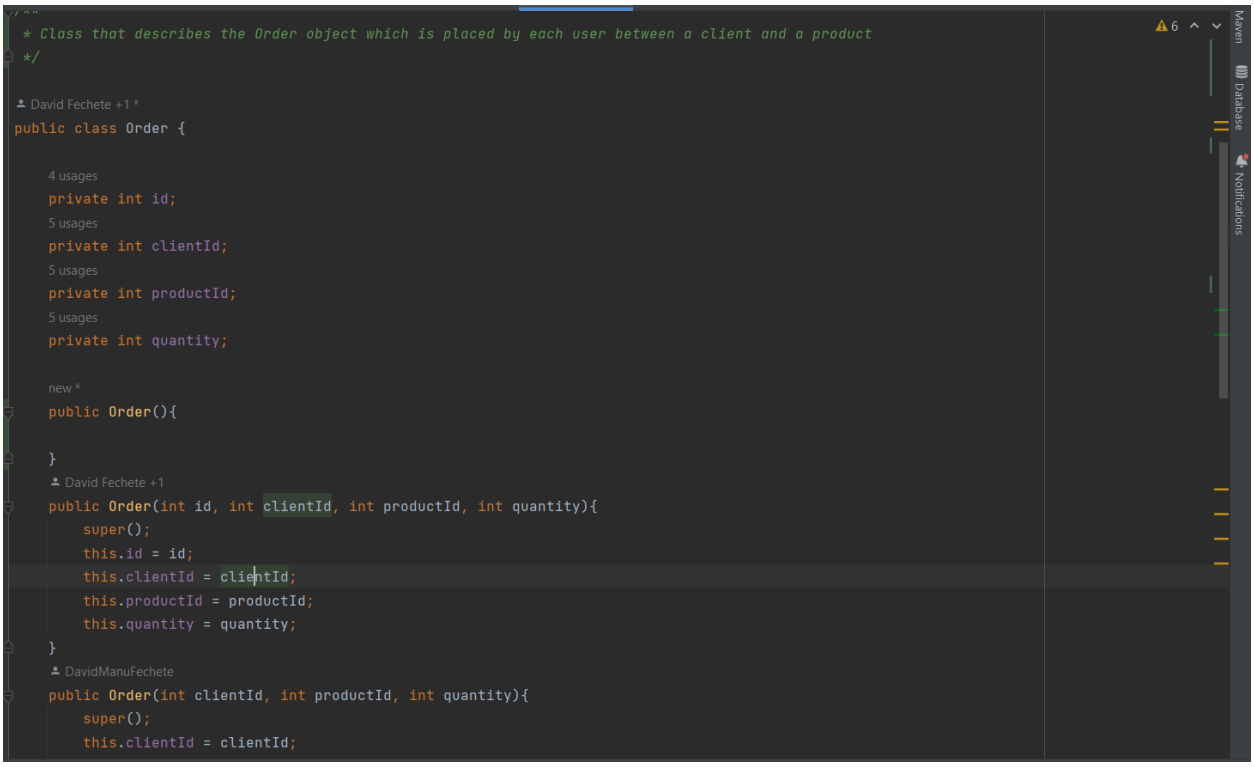
The second implemented class is the product class that contains the definition and attributes of the product that is bought by a client. It also has a unique id that identifies the product, a name, a price and the quantity of that product (the stock from the warehouse). It has no specific methods as well.

```java
package ro.tuc.model;
/**
 * Class that describes the Product object used in the application that represents the object that can be ordered by a client
 */
David Fechete +1 *
public class Product {

    4 usages
    private int id;
    5 usages
    private String name;
    4 usages
    private double price;
    5 usages
    private int quantity;

    new *
    public Product(){

    }
    David Fechete +1
    public Product(int id, String name, double price, int quantity){
        super();
        this.id = id;
        this.price = price;
        this.name = name;
        this.quantity = quantity;
    }
    DavidManuFechete
    public Product (String name, double price, int quantity){
        super();
        this.name = name;
```

```java
    DavidManuFechete
    public Product (String name, double price, int quantity){
        super();
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }

    DavidManuFechete
    public double getPrice() { return price; }

    DavidManuFechete
    public void setPrice(double price) { this.price = price; }

    David Fechete
    public int getId() { return id; }

    David Fechete
    public void setId(int id) { this.id = id; }

    David Fechete +1
    public String getName() { return name; }

    David Fechete +1
    public void setName(String name) { this.name = name; }

    David Fechete
    public int getQuantity() { return quantity; }

    David Fechete
    public void setQuantity(int quantity) { this.quantity = quantity; }
```

Order

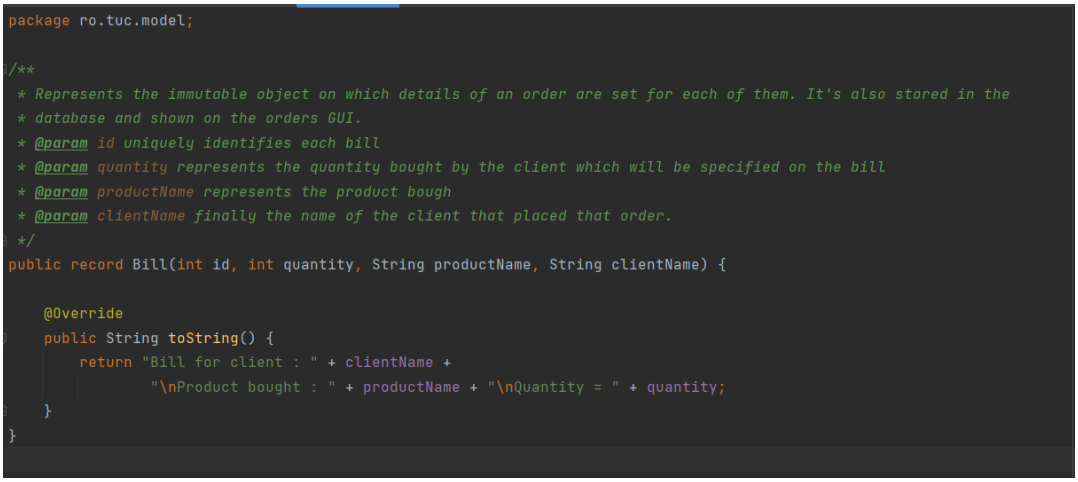
The next class from the model package is the Order class that represents the action that a client has to place whenever he/she wants to buy a product. It has as attributes a unique identifier id but also the id of the client and product that need to be placed as well as the quantity.



```java
/**
 * Class that describes the Order object which is placed by each user between a client and a product
 */

David Fechete +1 *
public class Order {

    4 usages
    private int id;
    5 usages
    private int clientId;
    5 usages
    private int productId;
    5 usages
    private int quantity;

    new *
    public Order(){

    }
    David Fechete +1
    public Order(int id, int clientId, int productId, int quantity){
        super();
        this.id = id;
        this.clientId = clientId;
        this.productId = productId;
        this.quantity = quantity;
    }
    DavidManuFechete
    public Order(int clientId, int productId, int quantity){
        super();
        this.clientId = clientId;
```

Bill

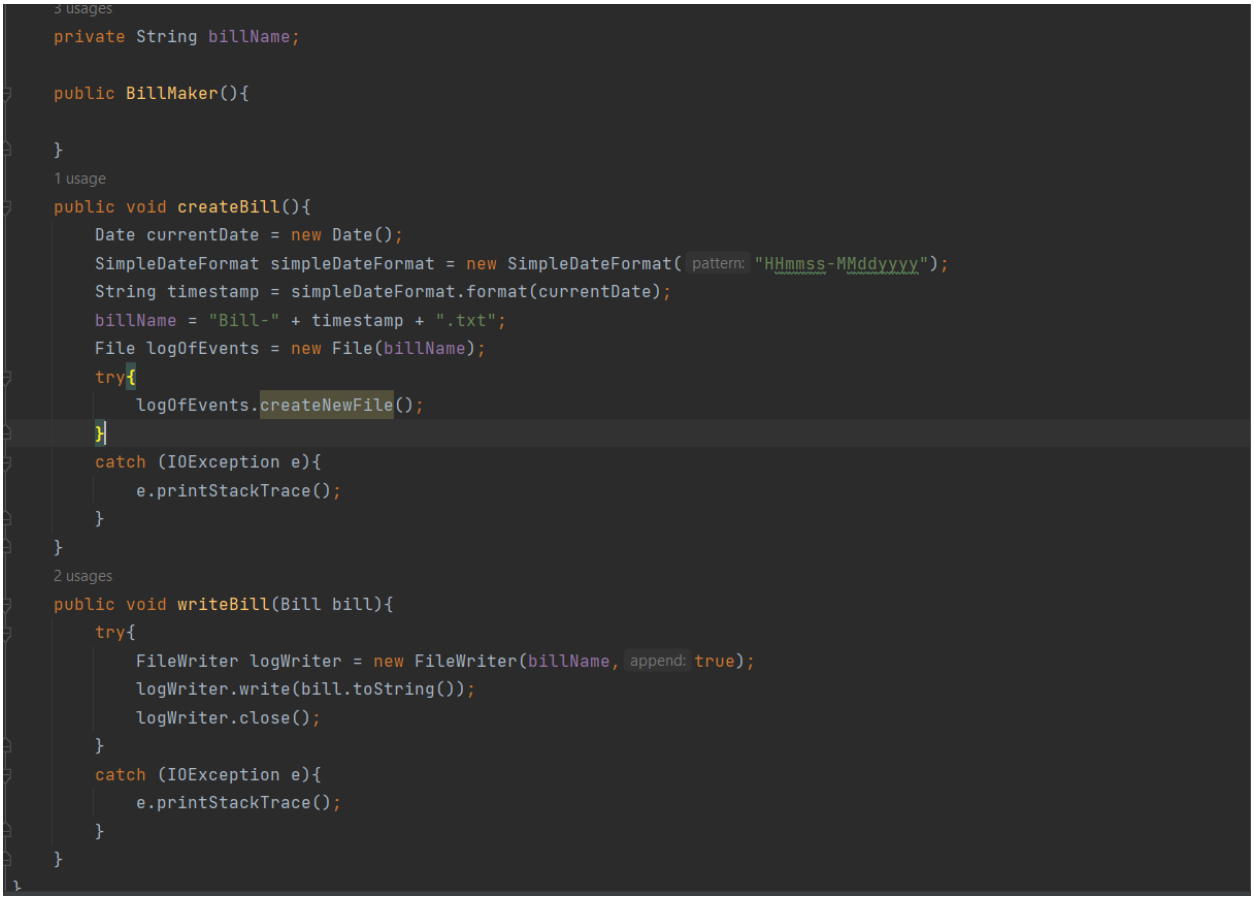
The last class from the model package is the Bill class that represents the text of the placed order. It stores data about the placed order. It's a record type class, meaning it's immutable and once created it can only be inserted into the database and read from it, no updates are allowed. It stores information about the clients name, the product and the quantity that the product is bought of.



```java
package ro.tuc.model;

/**
 * Represents the immutable object on which details of an order are set for each of them. It's also stored in the
 * database and shown on the orders GUI.
 * @param id uniquely identifies each bill
 * @param quantity represents the quantity bought by the client which will be specified on the bill
 * @param productName represents the product bough
 * @param clientName finally the name of the client that placed that order.
 */
public record Bill(int id, int quantity, String productName, String clientName) {

    @Override
    public String toString() {
        return "Bill for client : " + clientName +
                "\nProduct bought : " + productName + "\nQuantity = " + quantity;
    }
}
```

BillMaker

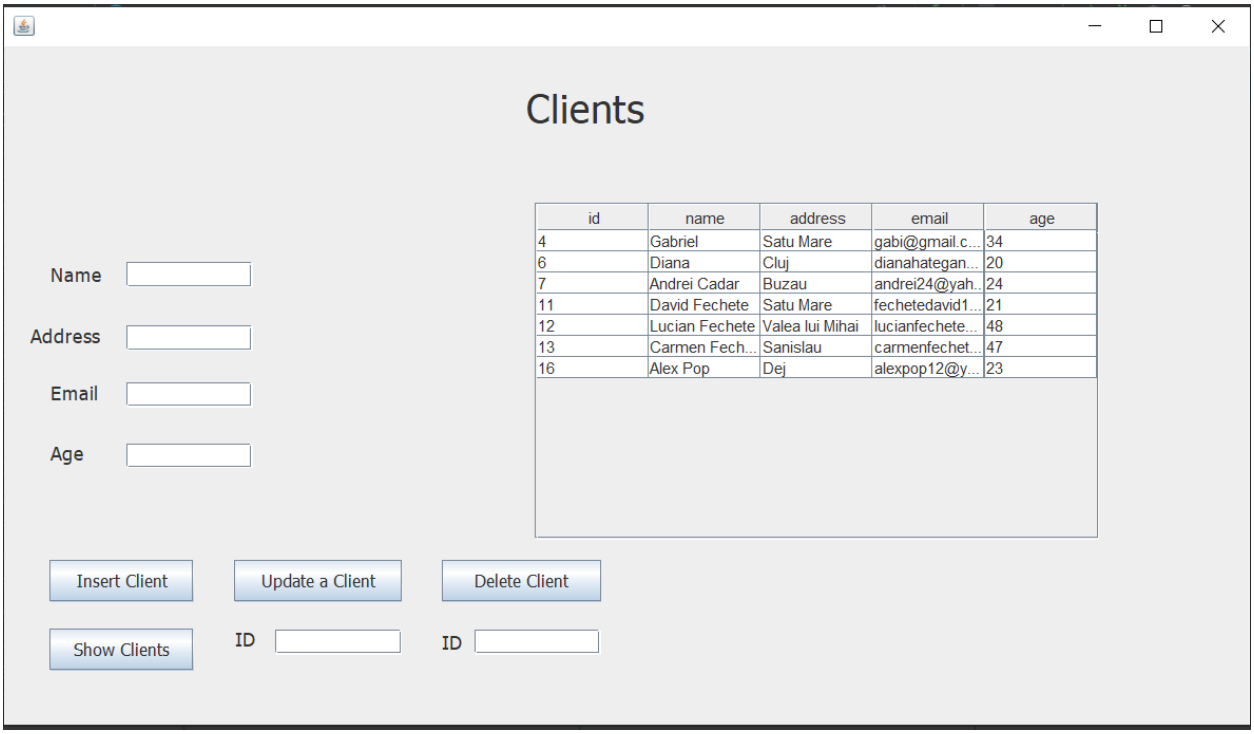
The next class is from the presentation package and it's the BillMaker class. It's a class that has a name of the bill and has 2 methods that create a separate bill at each call that containn the information about a bill and have a unique name because their title contains the exact date of the bill.



```
3 usages
private String billName;

public BillMaker(){

}
1 usage
public void createBill(){
    Date currentDate = new Date();
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat( pattern: "HHmmss-MMddyyyy");
    String timestamp = simpleDateFormat.format(currentDate);
    billName = "Bill-" + timestamp + ".txt";
    File logOfEvents = new File(billName);
    try{
        logOfEvents.createNewFile();
    }
    catch (IOException e){
        e.printStackTrace();
    }
}
2 usages
public void writeBill(Bill bill){
    try{
        FileWriter logWriter = new FileWriter(billName, append: true);
        logWriter.write(bill.toString());
        logWriter.close();
    }
    catch (IOException e){
        e.printStackTrace();
    }
}
}
```

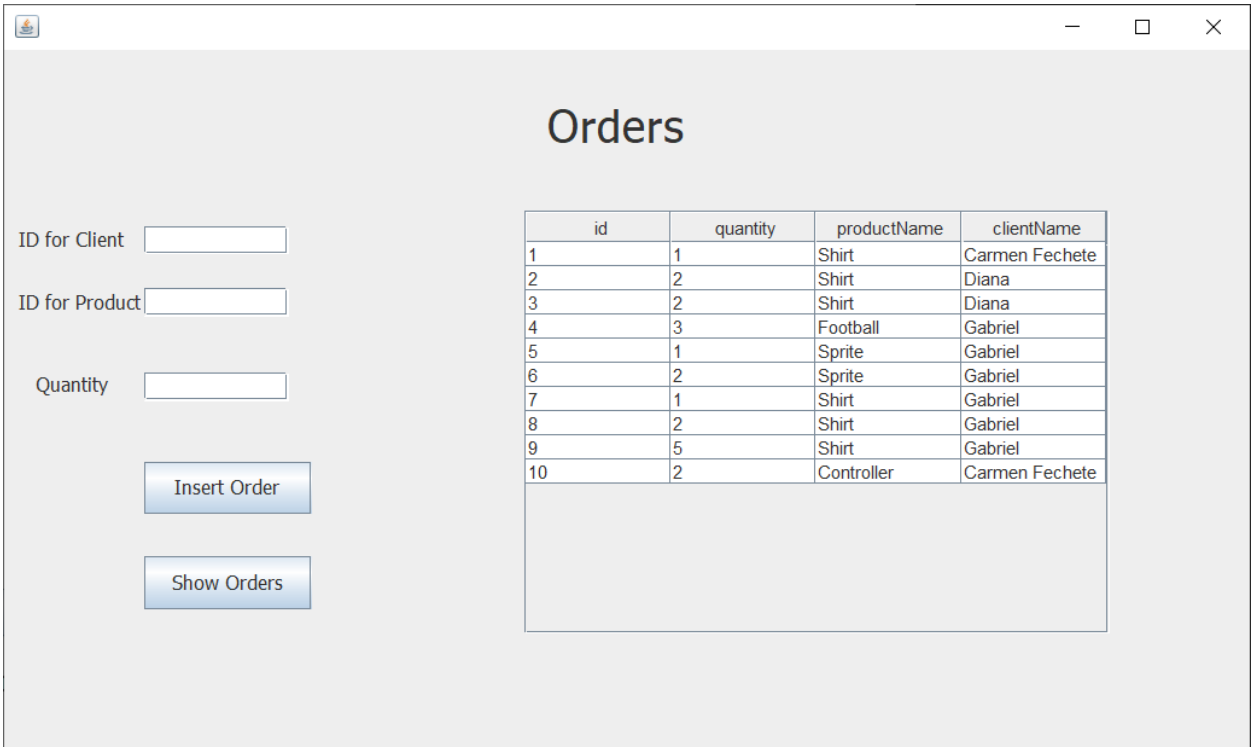
ClientsView

ClientsView class is the first implemented GUI class that extends JFrame and contains the implementation of the GUI. It's like a canvas that has components added onto it like buttons, text fields and labels and also a scrollpane onto which the table populated by clients is added.



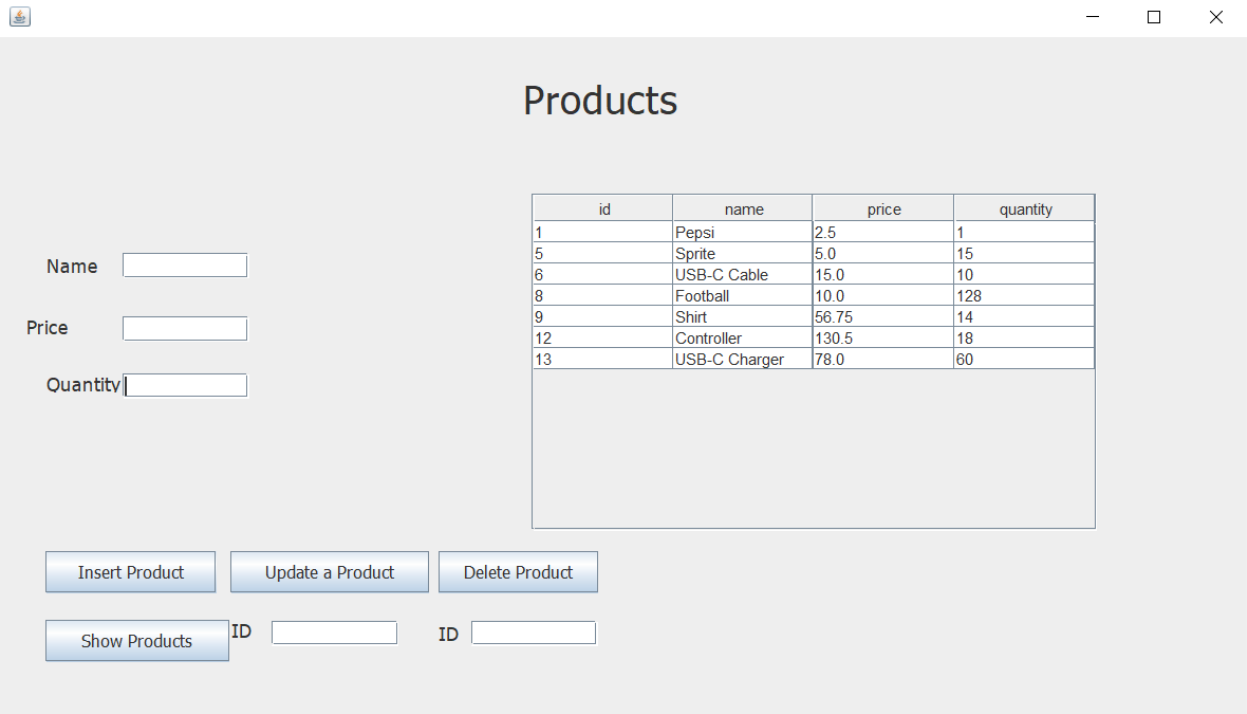
OrdersView

OrdersView class is very similar to the ClientsView. It also is a JFrame extension and like a canvas onto which less buttons are added. The table is present again and this time it will contain the bills of the orders.



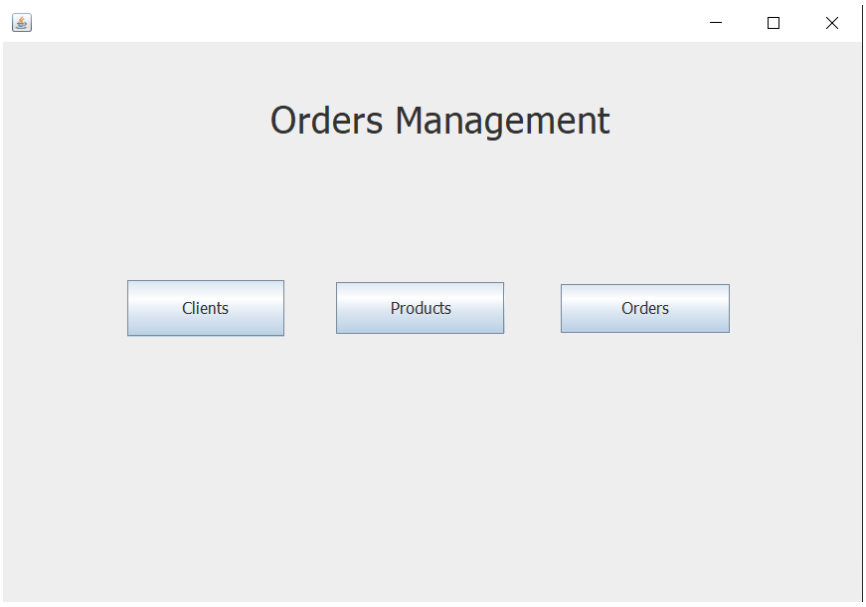
ProductsView

ProductsView is the last GUI class that extends JFrame and is very similar to the other 2 GUI classes.



MainView

MainView is the main GUI that opens up when the application is started and it's a very simple interface that contains only 3 buttons that send the user to the other 3 GUI in which the user can communicate with the program.



Controller

The next class, and last of the presentation package, is the Controller class. In the controller class we have the listeners for all the buttons present across the application. This class has as parameters all the Views and is very useful to make sure that each button is configured properly and that each pressing of the button sends the information from the view to the appropriate operation from the DAO and BLL classes.

```
DavidManuFechete *
public Controller (ClientsView clientsView, ProductsView productsView, OrdersView ordersView, MainView mainView) {

    this.billMaker = new BillMaker();
    this.billMaker.createBill();
    this.clientsView = clientsView;
    this.productsView = productsView;
    this.ordersView = ordersView;
    this.mainView = mainView;

    this.mainView.goToClientsListener(new goToClients());
    this.mainView.goToProductsListener(new goToProducts());
    this.mainView.goToOrdersListener(new goToOrders());

    this.clientsView.addClientListener(new InsertClient());
    this.clientsView.deleteClientListener(new DeleteClient());
    this.clientsView.showClientsListener(new ShowClients());
    this.clientsView.updateClientsListener(new UpdateClient());

    this.productsView.addProductListener(new InsertProduct());
    this.productsView.deleteProductListener(new DeleteProduct());
    this.productsView.showProductListener(new ShowProducts());
    this.productsView.updateProductListener(new UpdateProduct());

    this.ordersView.insertOrderListener(new InsertOrder());
    this.ordersView.showOrdersListener(new ShowOrders());
}
```

```
1 usage  DavidManuFechete *
class InsertProduct implements ActionListener{
    DavidManuFechete *
    @Override
    public void actionPerformed(ActionEvent e) {

        Product product = new Product(productsView.getTextField(), productsView.getPriceTextField(), productsView.getQuantityTextField());
        if(ProductBLL.checkProduct(product)){
            ProductBLL.insertProduct(product);
            productsView.showInsertMessage( msg: "Product inserted successfully !");
        }
        else{
            productsView.errorInsertMsg("Wrong input !");
        }
    }
}
1 usage  DavidManuFechete *
class ShowClients implements ActionListener{
    DavidManuFechete *
    @Override
    public void actionPerformed(ActionEvent e) {
        ClientDAO clientDAO = new ClientDAO();
        List<Client> clientList = ClientBLL.findAllClients();
        List<Client> clients = clientDAO.findAll();
        JTable table = clientDAO.generateTable(clients);
        clientsView.getScrollPane().setViewportView(table);
    }
}
```

Example of inner classes present in the Controller that are used as listeners for the buttons to show clients and insert products.

AbstractDAO

The dao package (data access layer) is perhaps the most important package and AbstractDAO the most important class. It's a class that's implemented with generics <T> in order for it to be able to work with any of the types that are present in the database. It uses reflection tehniques for generating the objects or to extract information from an object. It's a class that has methods used to create the necessary queries for the C-R-U-D operations for the database. It has 1 attribute: the type of the class T so it knows what class the object that needs a query is. It has methods for: finding all objects of that type and creating a list that contains all the objects from the database named findAll()

```
David Fechete *
public List<T> findAll() {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.append("SELECT ").append(" * ").append(" FROM orders_management.").append(type.getSimpleName());
    String query = stringBuilder.toString();
    try{
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        resultSet = statement.executeQuery();
        return createObjects(resultSet);
    }
    catch (SQLException e){
        LOGGER.log(Level.WARNING,  msg: type.getName() + "DAO:findAll " + e.getMessage());
    }
    finally{
        ConnectionFactory.close(resultSet);
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }
    return null;
}
```

It also has a findById function that is useful for finding a certain object given by its id

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

The createObjects() method creates the specific object given by a resultSet and returning the list.

```
private List<T> createObjects(ResultSet resultSet) {
    List<T> list = new ArrayList<T>();
    Constructor[] ctors = type.getDeclaredConstructors();
    Constructor ctor = null;
    for (int i = 0; i < ctors.length; i++) {
        ctor = ctors[i];
        if (ctor.getGenericParameterTypes().length == 0)
            break;
    }
    try {
        while (resultSet.next()) {
            ctor.setAccessible(true);
            T instance = (T)ctor.newInstance();
            for (Field field : type.getDeclaredFields()) {
                String fieldName = field.getName();
                Object value = resultSet.getObject(fieldName);
                PropertyDescriptor propertyDescriptor = new PropertyDescriptor(fieldName, 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) {
```

Next method is the generateTable(List <T> list) method which is used to generate the specific table model needed for each class. As each object is of a different type they need certain models that need to be generated, and this method does this by taking the fields of the first element from the list and setting the name of the columns to be the name of the fields. Then it traverses through the list and at each step takes the element from the list and goes through the table and puts at each cell from the table the corresponding information of the object. Then it returns that table.

```
3 usages   DavidManuFechete
public JTable generateTable(List<T> list){

    DefaultTableModel model = new DefaultTableModel();
    Field[] fields = list.get(0).getClass().getDeclaredFields();
    ArrayList<String> columns = new ArrayList<~>();
    for (Field field : fields){
        field.setAccessible(true);
        columns.add(field.getName());
    }
    for (String columnName : columns){
        model.addColumn(columnName);
    }
    for (T element : list){
        ArrayList<Object> objects = new ArrayList<>();
        Field[] elementField = element.getClass().getDeclaredFields();
        for (Field field : elementField){
            field.setAccessible(true);
            try{
                objects.add(field.get(element));
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
        model.addRow(objects.toArray());
    }
    JTable table = new JTable(model);
    return table;
}
```

The next method present is the insert (T t) object that creates an INSERT SQL statement specific for the t object that it's given as a parameter. It uses reflection to extract the fields of the object and it then creates a string queryBuilder that will contain the needed query for any object given as parameter

```
public T insert(T t) {

    Connection connection = null;
    PreparedStatement statement = null;
    StringBuilder queryBuilder = new StringBuilder();
    queryBuilder.append("INSERT INTO orders_management.").append(type.getSimpleName().toLowerCase());
    queryBuilder.append(" VALUES(");
    for (Field field : t.getClass().getDeclaredFields()){
        field.setAccessible(true);
        Object value;
        try{
            value = field.get(t);
            if(value.getClass().getSimpleName().equals("String")){
                queryBuilder.append("'").append(value).append("',");
            }
            else{
                queryBuilder.append(value).append(",");
            }
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
    if (queryBuilder.charAt(queryBuilder.length() - 1) == ','){
        queryBuilder.deleteCharAt( index: queryBuilder.length() - 1);
        queryBuilder.append(")");
    }
    String query = queryBuilder.toString();
```

Next method is the delete method that creates the DELETE statement for any object. It gets as a parameter the object that needs to be deleted and it takes the id field from it using reflection and constructs the query for the delete query

```java
public T delete(T t){

    Connection connection = null;
    PreparedStatement statement = null;
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.append("DELETE FROM orders_management.").append(type.getSimpleName().toLowerCase()).append(" WHERE id = ");
    try{
        connection = ConnectionFactory.getConnection();
        Field[] fields = t.getClass().getDeclaredFields();
        fields[0].setAccessible(true);
        stringBuilder.append(fields[0].get(t));
        String query = stringBuilder.toString();
        statement = connection.prepareStatement(query);
        statement.execute(query);
    }
    catch(Exception e){
        e.printStackTrace();
    }
    ConnectionFactory.close(statement);
    ConnectionFactory.close(connection);
    return t;
}
```

Last method from this class is the update method which creates the update statement for the SQL. It takes as parameter an object that shares the id with the object that needs to be updated but this parameter has the updated fields so basically it creates a statement that takes the fields from the given object and updates them in place of the object with the same id from the database.

```java
public T update(T t) {

    Connection connection = null;
    PreparedStatement statement = null;
    StringBuilder queryBuilder = new StringBuilder();
    Object value = null;
    boolean first = true;
    int id = INVALID_VALUE;
    queryBuilder.append("UPDATE orders_management.").append(type.getSimpleName().toLowerCase());
    // UPDATE orders_management.client
    try{
        Field[] fields = t.getClass().getDeclaredFields();
        fields[0].setAccessible(true);
        value = fields[0].get(t);
    }
    catch(Exception e){
        e.printStackTrace();
    }
    queryBuilder.append(" SET ");
    for (Field field : t.getClass().getDeclaredFields()) {
        field.setAccessible(true);
        try {
            if (first) {
                first = false;
                id = (int) field.get(t);
            } else {
                queryBuilder.append(field.getName().toLowerCase()).append(" = ");
                if (field.getType().getSimpleName().equals("String")) {
                    queryBuilder.append("'").append(field.get(t)).append("',");
```

```
                if (field.getType().getSimpleName().equals("String")) {
                    queryBuilder.append("'").append(field.get(t)).append("',");
                } else {
                    queryBuilder.append(field.get(t)).append(",");
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        if (id == INVALID_VALUE) {
            System.out.println("Invalid");
            return null;
        }
    }
    queryBuilder.deleteCharAt( index: queryBuilder.length() - 1);
    queryBuilder.append(" WHERE id = ").append(id);
    String query = queryBuilder.toString();
    try{
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.execute(query);
        connection.close();
        statement.close();
    }
    catch(Exception e){
        e.printStackTrace();
    }
```

Connection Factory

This class is part of a package connection and it's the sole class in the package. It's used for establishing the connection with the database by placing the connection in a singleton object. The class has a driver name that is initialized using reflection, the database location and the user data (username and password) for accessing the MySQL server. It has methods that are used to close the connection, resultSet and statement.

```
4 usages
private static final Logger LOGGER = Logger.getLogger(ConnectionFactory.class.getName());
1 usage
private static final String DRIVER = "com.mysql.cj.jdbc.Driver";
1 usage
private static final String DBURL = "jdbc:mysql://localhost:3306/orders_management";
1 usage
private static final String USER = "root";
1 usage
private static final String PASS = "";

1 usage
private static ConnectionFactory singleInstance = new ConnectionFactory();

1 usage  David Fechete
private ConnectionFactory() {
    try {
        Class.forName(DRIVER);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}

1 usage  David Fechete
private Connection createConnection() {
    Connection connection = null;
    try {
        connection = DriverManager.getConnection(DBURL, USER, PASS);
    } catch (SQLException e) {
```

Business logic layer

This package contains classes for each object. They implement an extra layer of validation for before letting the data be operated on the database. It has validator classes and a Validator interface that implement various rules for the input data to be checked before letting it be inserted into the database. The classes are very similar to each other, and they all have methods that make calls for the DAO methods after checking if the input corresponds to the rules imposed.

```
public ProductBLL(){
    validators = new ArrayList<Validator<Product>>();
    validators.add(new PriceValidator());
    validators.add(new QuantityValidator());
    productDAO = new ProductDAO();
}

2 usages
public static Product findProductById(int id){
    Product product = productDAO.findById(id);
    if (product == null){
        return null;
    }
    return product;
}
1 usage
public static boolean checkProduct(Product product){
    PriceValidator priceValidator = new PriceValidator();
    QuantityValidator quantityValidator = new QuantityValidator();
    try{
        priceValidator.validate(product);
        quantityValidator.validate(product);
```

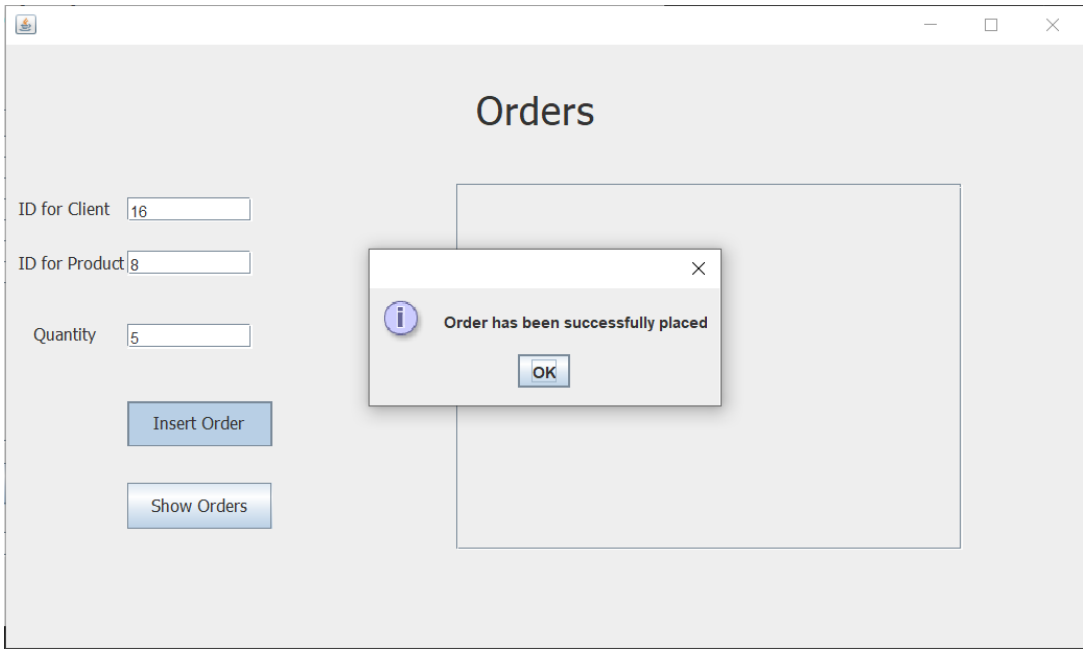
Each BLL class has a list of validators and also a checkObject method that makes sure that the object corresponds to the rules.

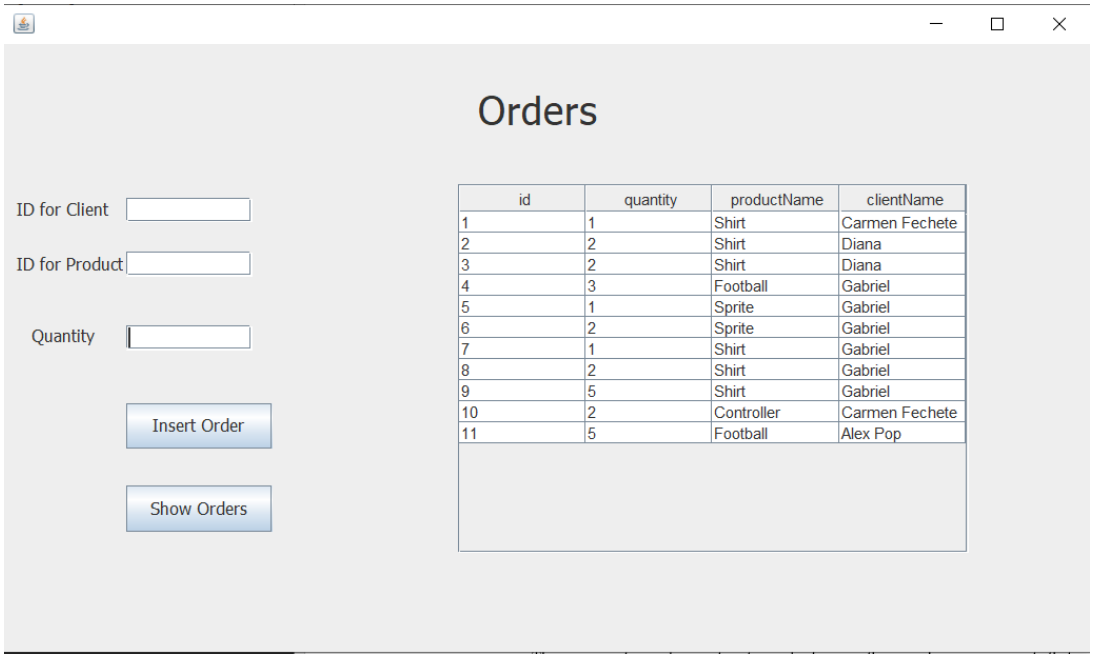
```
package ro.tuc.bll.validators;

import ro.tuc.model.Client;
/**
 * Validator that implements the validate method, in this case the validation of the age, the age of a client must be over 18
 * in order for he/she to be able to place any orders.
 */
David Fechete
public class AgeValidator implements Validator<Client> {

    1 usage
    private static final int MIN_AGE = 18;

    David Fechete
    public void validate(Client t) {

        if (t.getAge() < MIN_AGE) {
            throw new IllegalArgumentException("Age limit not respected ! Must be over 18");
        }
    }
}
```

Example of validator. All clients must be over 18 in order to place an order.

5. Results

Testing can be done by checking whether an order is correctly place when selecting the ids of the product and client as well as the quantity generates the right bill.





6. Conclusions

The orders management system is fully functional and meets the basic needs of a warehouse but it can be greatly improved by adding a better GUI, both visually and by functional means as well as adding more systems like a pricing system or a better visualization method. Overall I believe that the application greatly improved my skills on creating apps that interact with a database. The application has been documented using the help of Javadoc. The SQL dump file is present on github.

7. Bibliography

1. *https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html*
2. *https://www.baeldung.com/javadoc*
3. *https://jenkov.com/tutorials/java-reflection/index.html*
4. *https://dzone.com/articles/layers-standard-enterprise*