DOCUMENTATION

ASSIGNMENT *3*

ORDERS MANAGEMENT

STUDENT NAME: Vlad Darius Stefan
GROUP: 30226

CONTENTS

1. Assignment Objective

Main objective:

Design and implement an application for managing the client orders for a warehouse

Sub-objectives:

- Analyze the problem and identify requirements
- Design the orders management application
- Implement the orders management application
- Test the orders management application

2. Problem Analysis, Modeling, Scenarios, Use Cases

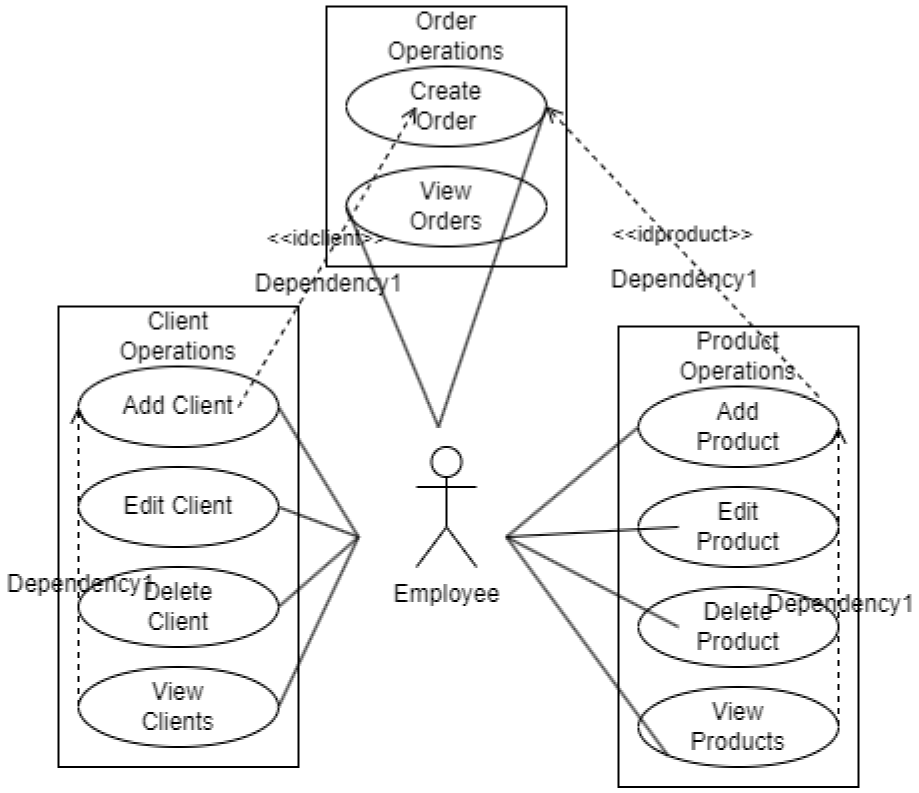
Functional requirements:

- The application should allow an employee to add a new client
- The application should allow an employee to add a new product
- The application should allow an employee to create a new order
- The application should allow an employee to delete a client
- The application should allow an employee to edit a client's details
- The application should allow to see all the clients in the database
- The application should allow an employee to edit a product’s details
- The application should allow an employee to delete a product
- The application should allow to see all the products in the database
- The application should allow to see all the orders that have been made
- The application should create a log file for every order created

Non-functional requirements:

- The application should be intuitive and easy to use by the employee
- The application should be visually appealing and easy for the user to follow
- The application should carefully treat exception cases and not interrupt the flow

Use case diagram:



***Use case:** Add Client
Primary actor: Employee
Main success scenario:

1. The employee clicks on the "ADD" button from the "Client Operations" page.
2. The application will display a form where the client details should be inserted.
3. The employee inserts the name of the client , its address , its email and its age and then presses the "Done button".
4. The application stores the client data in the database and displays an acknowledge message.

Alternative sequence:

1. The user inserts invalid values for the client data , or doesn't insert data in all the requested fields.
2. The application displays an error message.
3. The scenario returns to step 3.

***Use case:** Edit Client

Primary actor: Employee

Main success scenario:

1. The employee clicks on the "EDIT" button from the "Client Operations" page.
2. The application will display a form where the client id should be inserted.
3. The employee inserts the id of the client he wishes to edit the details for.
4. The employee clicks on the "Done" button.
5. A page with the clients details will open if the id is valid.
6. The employee changes the fields he wishes to edit.
7. The employee clicks on the "Done" button.
8. The client is edited in the database and an acknowledge message is displayed.

Alternative sequence 1:

1. The id which the employee inserts does not exist.
2. The application will display an error message.
3. The scenario returns to step 3.

Alternative sequence 2:

1. The client details the employee edits are not valid.
2. An error message will be displayed.
3. The scenario returns to step 6.

***Use case:** Delete client

Primary actor : Employee

Main success scenario:

1. The employee clicks on "DELETE" button from the "Client Operations" page.
2. The application will display a form where the client id should be inserted.
3. The employee inserts the id of the client he wishes to delete.
4. The client with the selected ID is deleted.

Alternative sequence:

1. The id which the employee inserts is not valid.
2. An error message will be displayed.
3. The scenario returns to step 3.

***Use case:** View all clients

Primary actor: Employee

Main success scenario:

1. The employee clicks on the "VIEW" button from the "Client Operations" page.

2. The application will display a table where all clients with their details are shown.

Alternative sequence:

1. The are no clients in the database.
2. The application will display a message which encourages the employee to add some clients.
3. The scenario returns to step 1.

***Use case:** Add Product

Primary actor: Employee

Main success scenario:

1. The employee clicks on the “ADD” button from the “Product Operations” page.
2. The application will display a form where the product details should be inserted.
3. The employee inserts the name of the client , its price and its available stock.
4. The employee clicks on the “Done” button.
5. The application stores the product data in the database and displays an acknowledge message.

Alternative sequence:

1. The user inserts invalid values for the product data , or doesn’t insert data in all the requested fields.
2. The application displays an error message.
3. The scenario returns to step 3.

***Use case:** Edit Product

Primary actor: Employee

Main success scenario:

1. The employee clicks on the “EDIT” button from the “Product Operations” page.
2. The application will display a form where the product id should be inserted.
3. The employee inserts the id of the product he wishes to edit the details for.
4. The employee clicks on the “Done” button.
5. A page with the product details will open if the id is valid.
6. The employee changes the fields he wishes to edit.
7. The employee clicks on the “Done” button.
8. The product is edited in the database and an acknowledge message is displayed.

Alternative sequence 1:

1. The id which the employee inserts does not exist.
2. The application will display an error message.
3. The scenario returns to step 3.

Alternative sequence 2:

1. The product details the employee edits are not valid.
2. An error message will be displayed.
3. The scenario returns to step 6.

***Use case:** Delete Product

Primary actor : Employee

Main success scenario:

1.The employee clicks on “DELETE” button from the “Product Operations” page.
2. The application will display a form where the product id should be inserted.
3. The employee inserts the id of the product he wishes to delete.
4. The product with the selected ID is deleted.

Alternative sequence:

1.The id which the employee inserts is not valid.
2. An error message will be displayed.
3. The scenario returns to step 3.

***Use case:** View all Products

Primary actor: Employee

Main success scenario:

1. The employee clicks on the “VIEW” button from the “Product Operations” page.
2. The application will display a table where all products with their details are shown.

Alternative sequence:

1. The are no products in the database.
2. The application will display a message which encourages the employee to add some products.
3. The scenario returns to step 1.

***Use case:** Create Order

Primary actor: Employee

Main success scenario:

1. The employee clicks on the “Create Order” button from the “Order Operations” page.
2. The application will display a form where the order details should be inserted.
3. The employee inserts the id of the client, the id of the product and the quantity wanted.
4. The employee clicks on the “Done” button.
5. The application stores the order data in the database and displays an acknowledge message.

Alternative sequence 1:

1. The user inserts invalid values for the order , or doesn’t insert data in all the requested fields.
2. The application displays an error message.
3. The scenario returns to step 3.

Alternative sequence 2:

1. The employee inserts a wanted quantity that is over the stock of the product.
2. The application displays an error message.
3. The scenario returns to step 3.

***Use case:** View all Orders
Primary actor: Employee
Main success scenario:

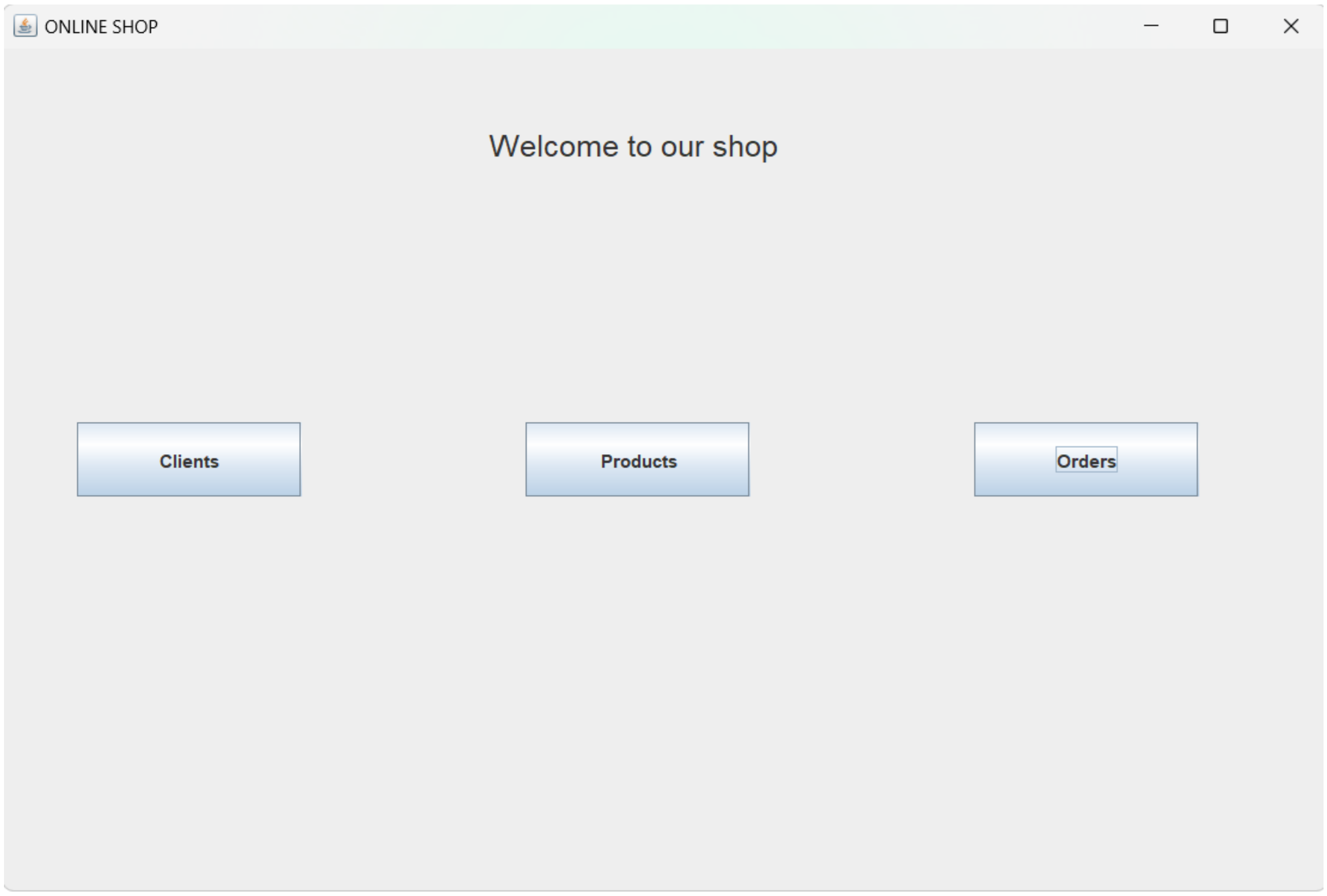
1. The employee clicks on the “VIEW” button from the “Order Operations” page.
2. The application will display a table where all orders with their details are shown.

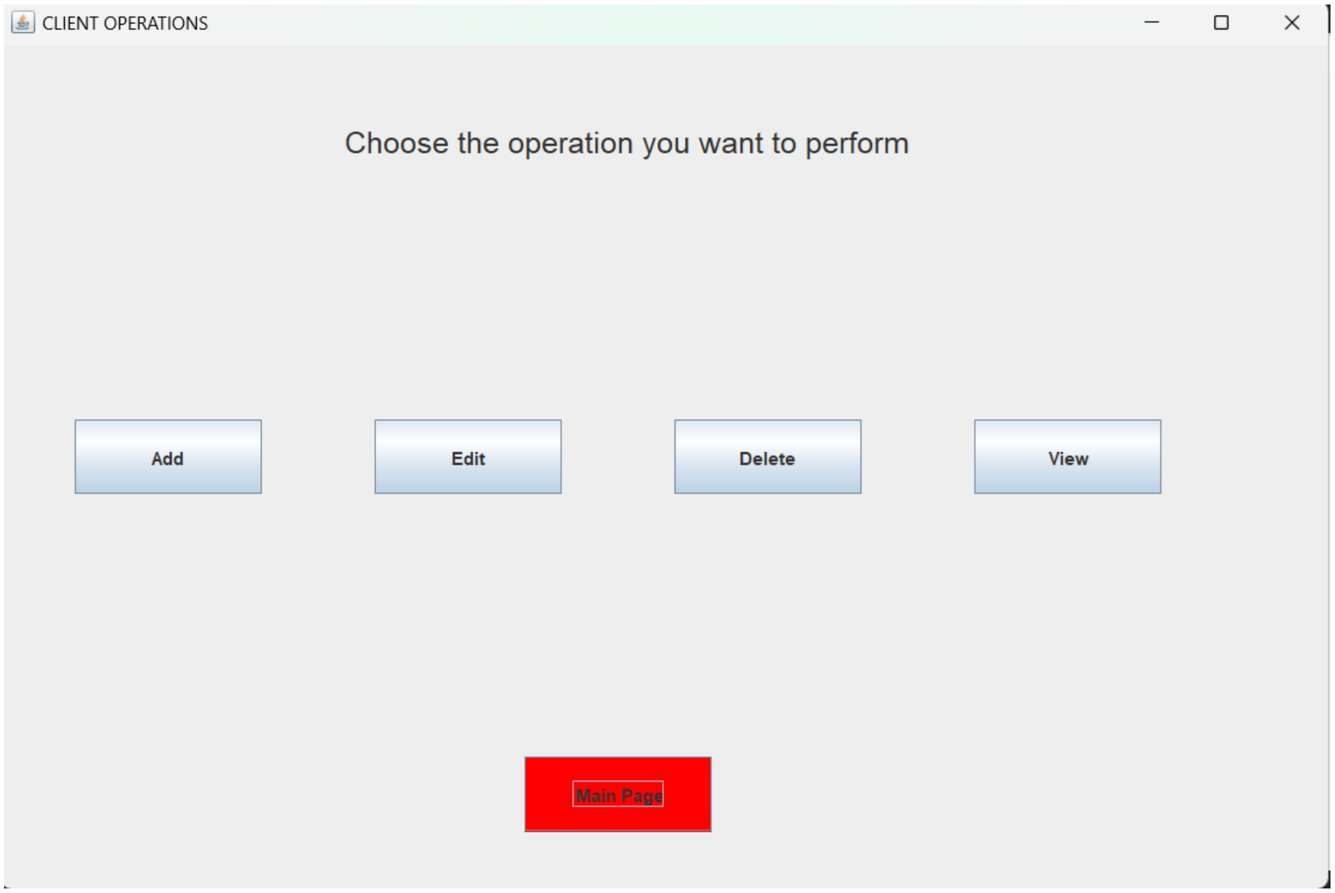
Alternative sequence:

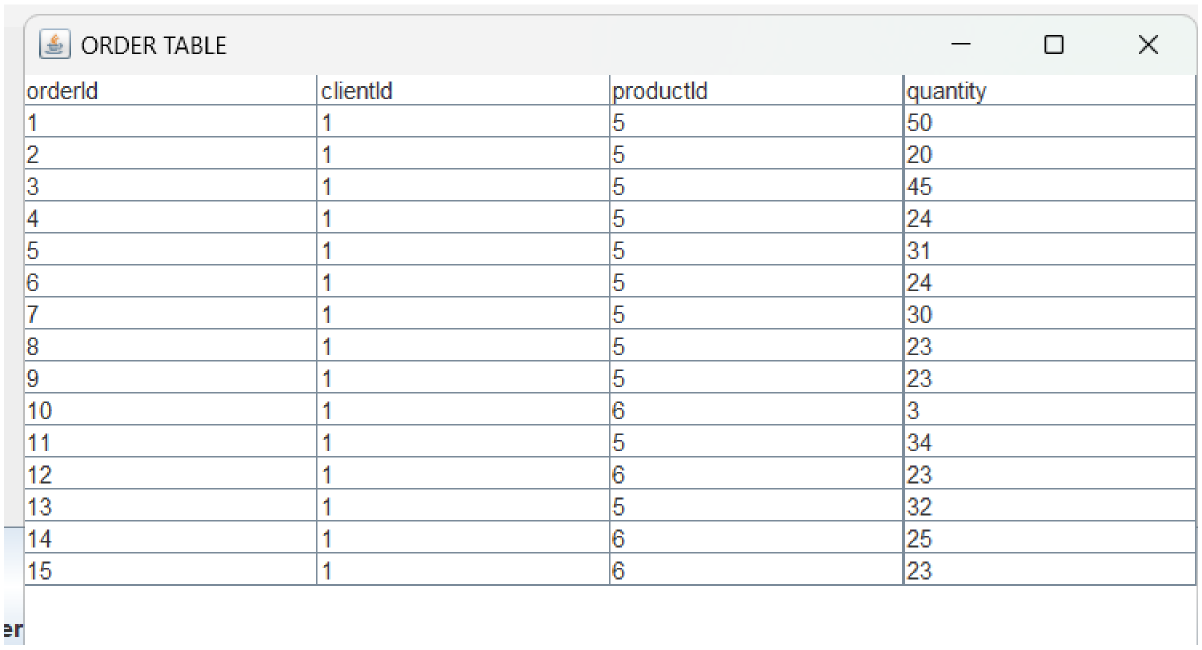
1. The are no orders in the database.
2. The scenario returns to step 1.

3. Design

Final design:

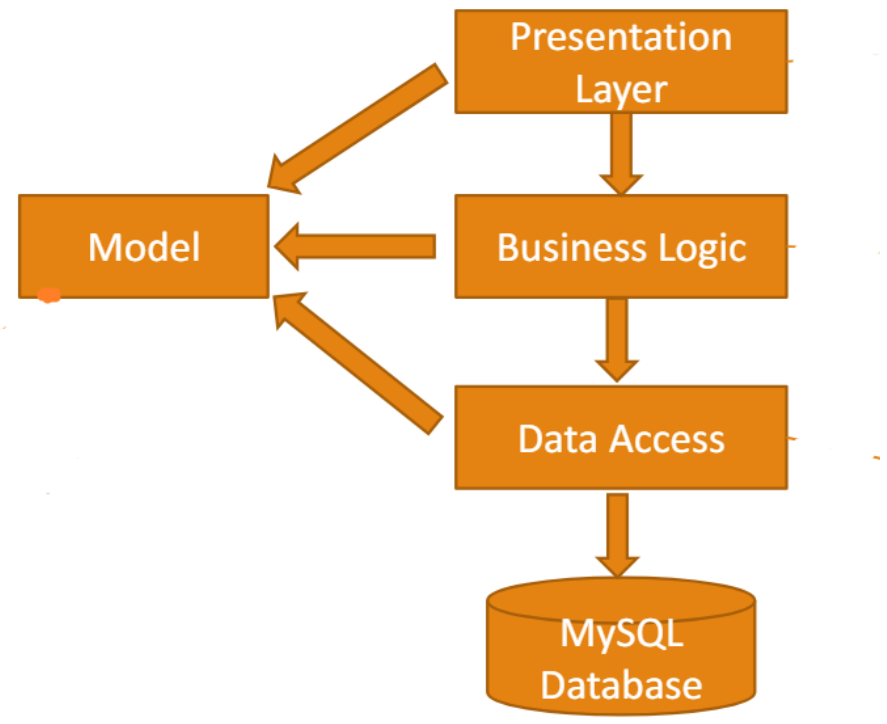




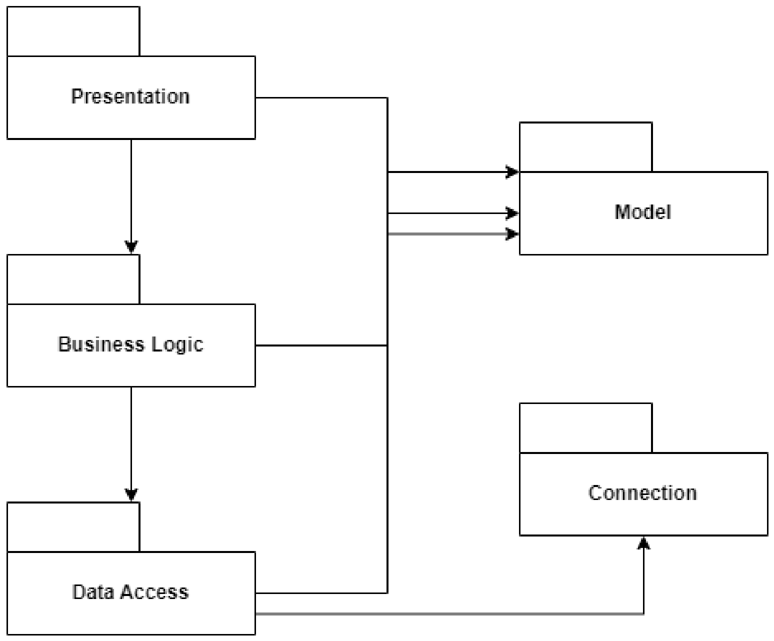

ORDER TABLE

orderId	clientId	productId	quantity
1	1	5	50
2	1	5	20
3	1	5	45
4	1	5	24
5	1	5	31
6	1	5	24
7	1	5	30
8	1	5	23
9	1	5	23
10	1	6	3
11	1	5	34
12	1	6	23
13	1	5	32
14	1	6	25
15	1	6	23

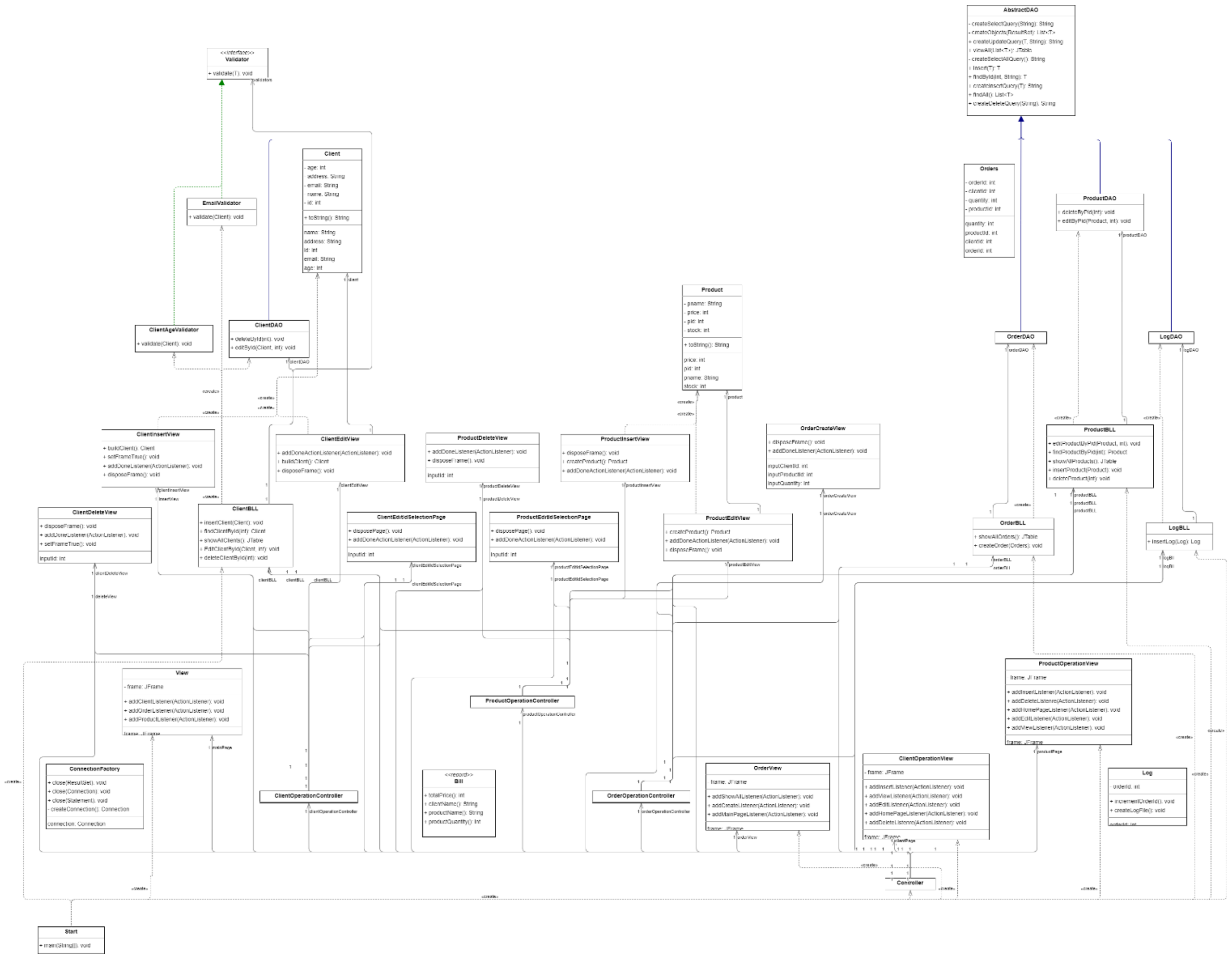
Conceptual architecture:

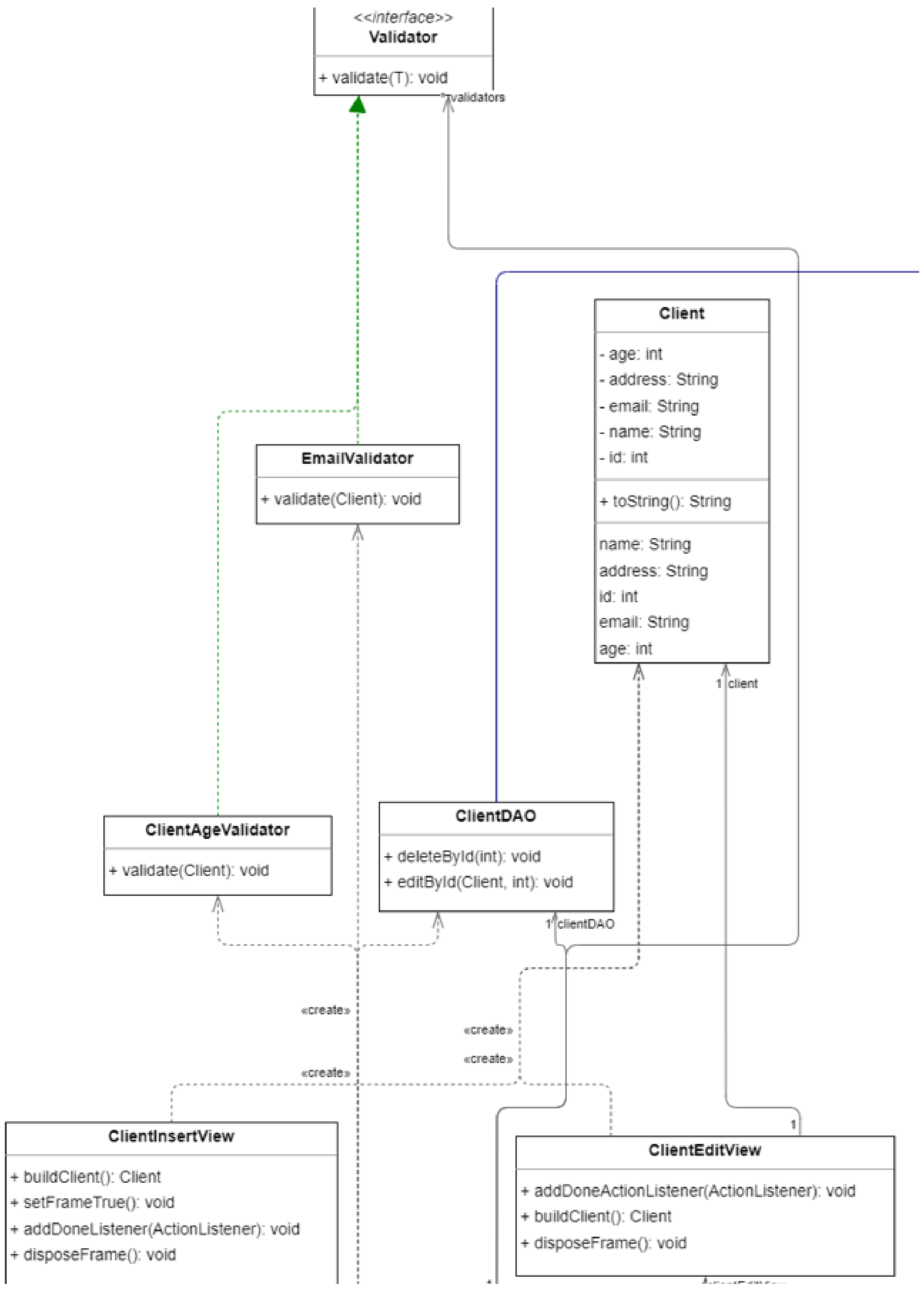


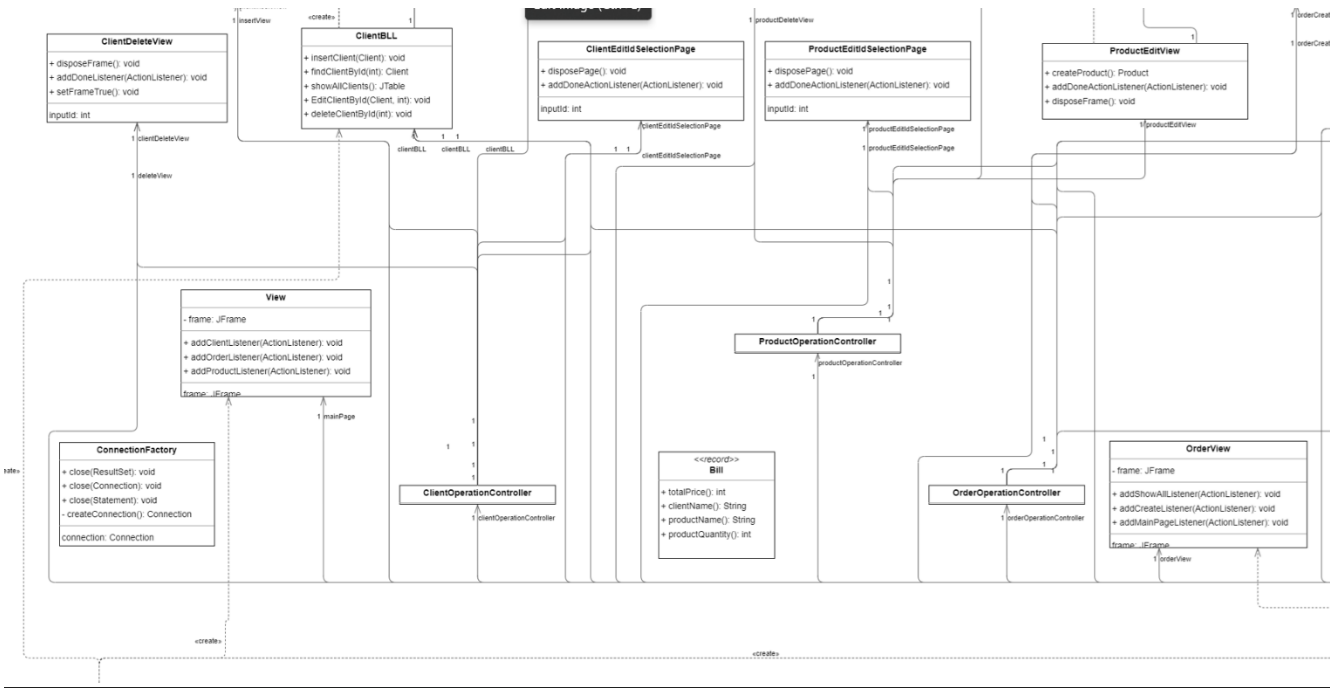
Package diagram:

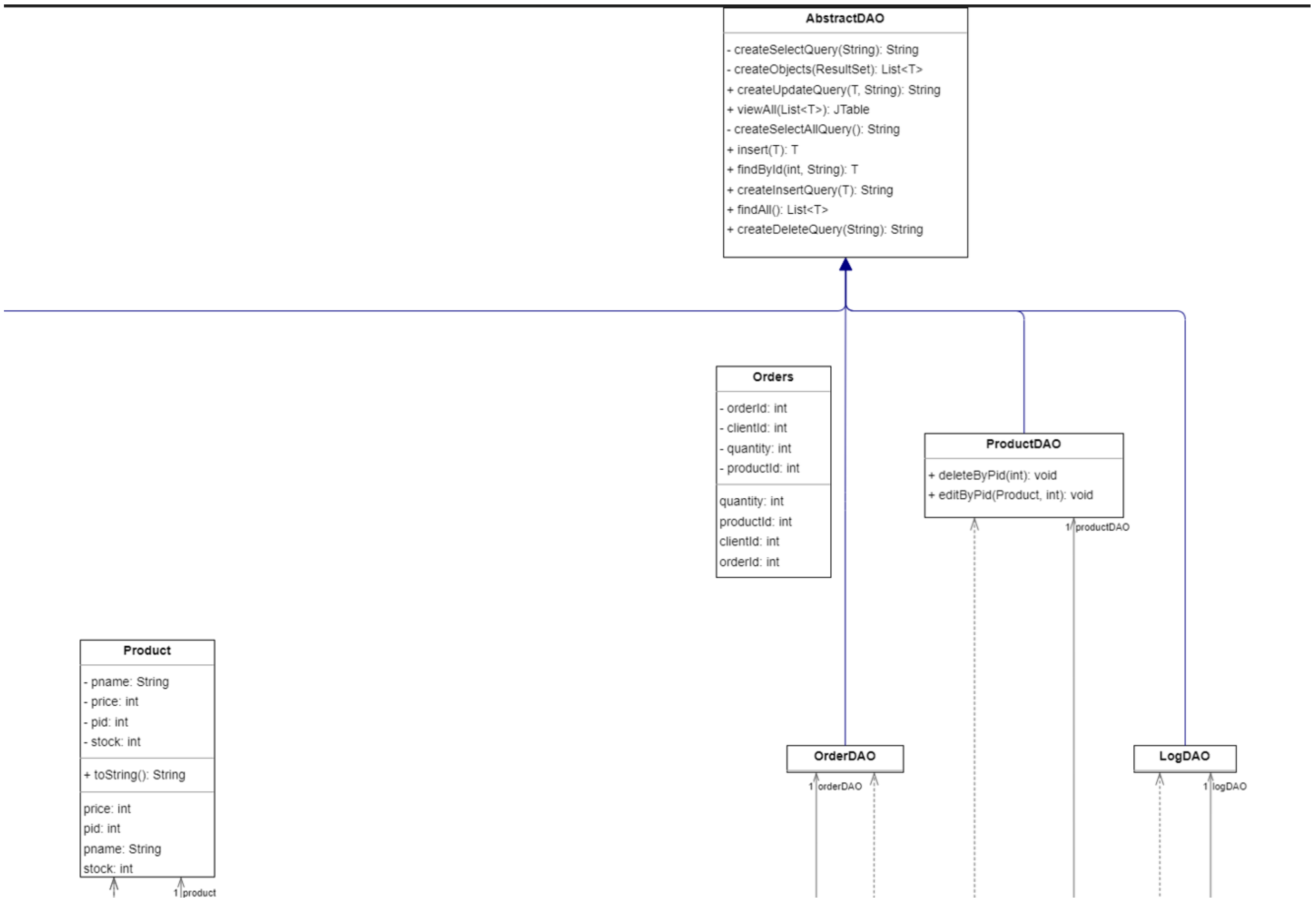


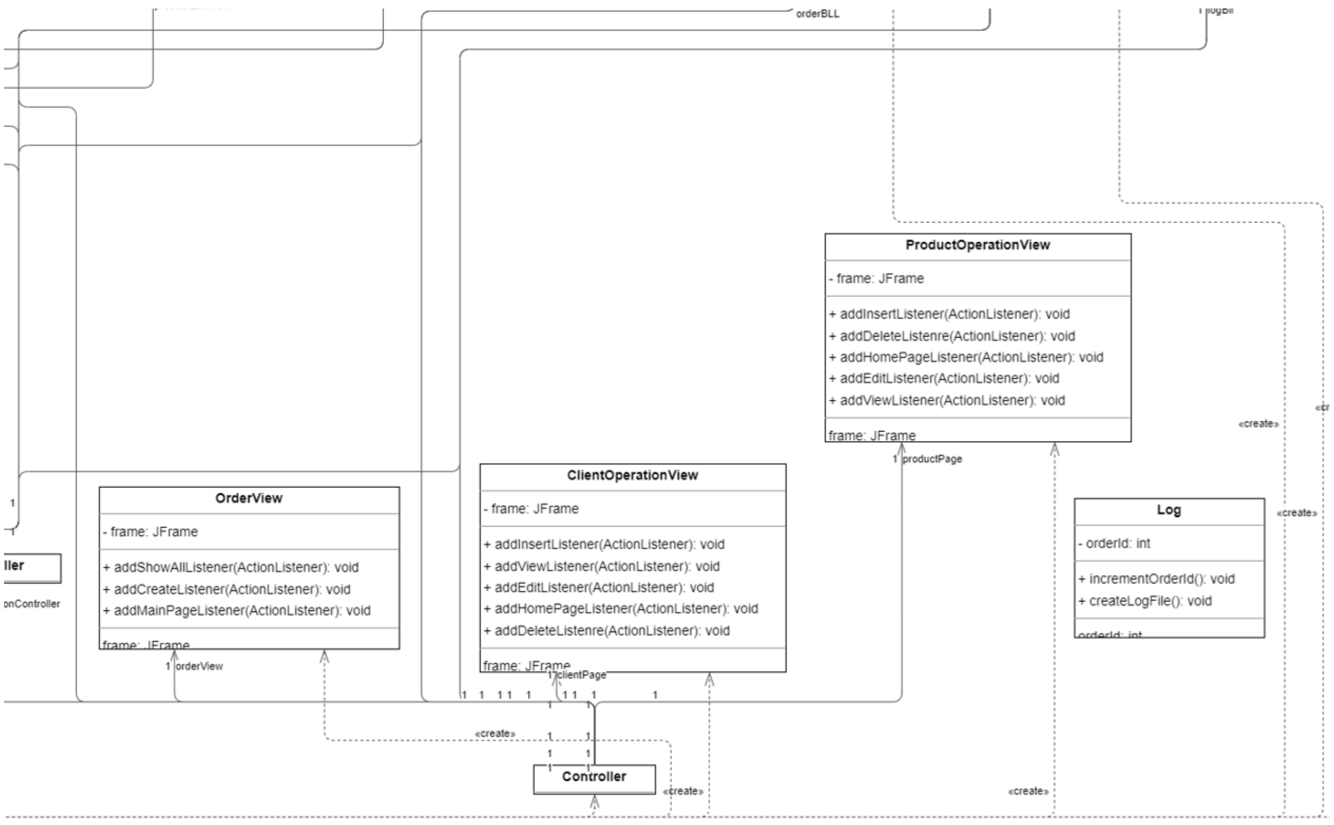
UML Class Diagram:




<<interface>>
Validator
+ validate(T): void
validators
EmailValidator
+ validate(Client): void
Client
- age: int
- address: String
- email: String
- name: String
- id: int
+ toString(): String
name: String
address: String
id: int
email: String
age: int
1 client
ClientAgeValidator
+ validate(Client): void
ClientDAO
+ deleteById(int): void
+ editById(Client, int): void
1 clientDAO
«create»
«create»
«create»
«create»
1
ClientInsertView
+ buildClient(): Client
+ setFrameTrue(): void
+ addDoneListener(ActionListener): void
+ disposeFrame(): void
ClientEditView
+ addDoneActionListener(ActionListener): void
+ buildClient(): Client
+ disposeFrame(): void


ClientDeleteView
+ disposeFrame(): void
+ addDoneListener(ActionListener): void
+ setFrameTrue(): void
inputId: int
ClientBLL
+ insertClient(Client): void
+ findClientById(int): Client
+ showAllClients(): JTable
+ EditClientById(Client, int): void
+ deleteClientById(int): void
ClientEditIdSelectionPage
+ disposePage(): void
+ addDoneActionListener(ActionListener): void
inputId: int
ProductEditIdSelectionPage
+ disposePage(): void
+ addDoneActionListener(ActionListener): void
inputId: int
ProductEditView
+ createProduct(): Product
+ addDoneActionListener(ActionListener): void
+ disposeFrame(): void
View
- frame: JFrame
+ addClientListener(ActionListener): void
+ addOrderListener(ActionListener): void
+ addProductListener(ActionListener): void
frame: JFrame
ProductOperationController
ConnectionFactory
+ close(ResultSet): void
+ close(Connection): void
+ close(Statement): void
- createConnection(): Connection
connection: Connection
ClientOperationController
<<record>>
Bill
+ totalPrice(): int
+ clientName(): String
+ productName(): String
+ productQuantity(): int
OrderOperationController
OrderView
- frame: JFrame
+ addShowAllListener(ActionListener): void
+ addCreateListener(ActionListener): void
+ addMainPageListener(ActionListener): void
frame: JFrame
«create»
insertView
clientDeleteView
deleteView
clientBLL
clientEditIdSelectionPage
productDeleteView
productEditIdSelectionPage
productEditView
orderCreat
productOperationController
mainPage
clientOperationController
orderOperationController
orderView


AbstractDAO
- createSelectQuery(String): String
- createObjects(ResultSet): List<T>
+ createUpdateQuery(T, String): String
+ viewAll(List<T>): JTable
- createSelectAllQuery(): String
+ insert(T): T
+ findById(int, String): T
+ createInsertQuery(T): String
+ findAll(): List<T>
+ createDeleteQuery(String): String
Orders
- orderId: int
- clientId: int
- quantity: int
- productId: int
quantity: int
productId: int
clientId: int
orderId: int
ProductDAO
+ deleteByPid(int): void
+ editByPid(Product, int): void
1 productDAO
Product
- pname: String
- price: int
- pid: int
- stock: int
+ toString(): String
price: int
pid: int
pname: String
stock: int
1 product
OrderDAO
1 orderDAO
LogDAO
1 logDAO


orderBLL
ProductOperationView
- frame: JFrame
+ addInsertListener(ActionListener): void
+ addDeleteListenre(ActionListener): void
+ addHomePageListener(ActionListener): void
+ addEditListener(ActionListener): void
+ addViewListener(ActionListener): void
frame: JFrame
productPage
OrderView
- frame: JFrame
+ addShowAllListener(ActionListener): void
+ addCreateListener(ActionListener): void
+ addMainPageListener(ActionListener): void
frame: JFrame
orderView
ClientOperationView
- frame: JFrame
+ addInsertListener(ActionListener): void
+ addViewListener(ActionListener): void
+ addEditListener(ActionListener): void
+ addHomePageListener(ActionListener): void
+ addDeleteListenre(ActionListener): void
frame: JFrame
clientPage
Log
- orderId: int
+ incrementOrderId(): void
+ createLogFile(): void
orderId: int
Controller
«create»

The **BLL** package contains business logic related classes. It contains 4 classes , each one containing the business logic for the class. It contains the classes : **ClientBLL** , **ProductBLL , OrderBLL** , **LogBLL**.

The **Model** package contains 5 main classes that are used for the application. We have a class called **Client** , that models the real life entity of clients and a class called **Product** , where we create a representation of a Product. We also have the **Order** class , the **Bill** class and the **Log** class.

The **DAO** package contains the **database operations** which are being performed in the application.

The **Connection** package contains a class that helps us connect to the database used for the application.

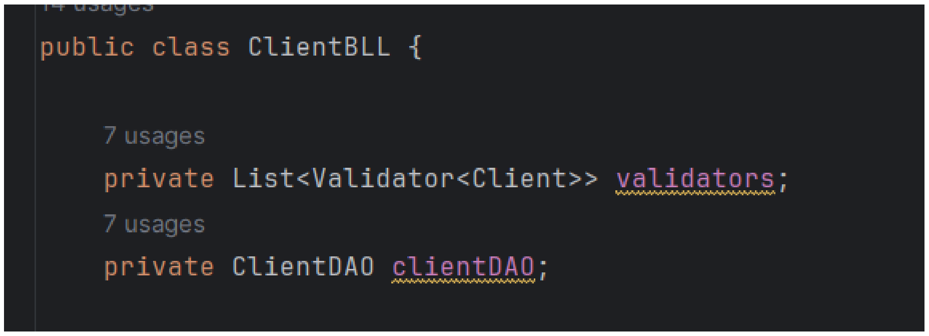
The **Presentation** package contains the design for each of the pages and the controllers for every button in the application

.

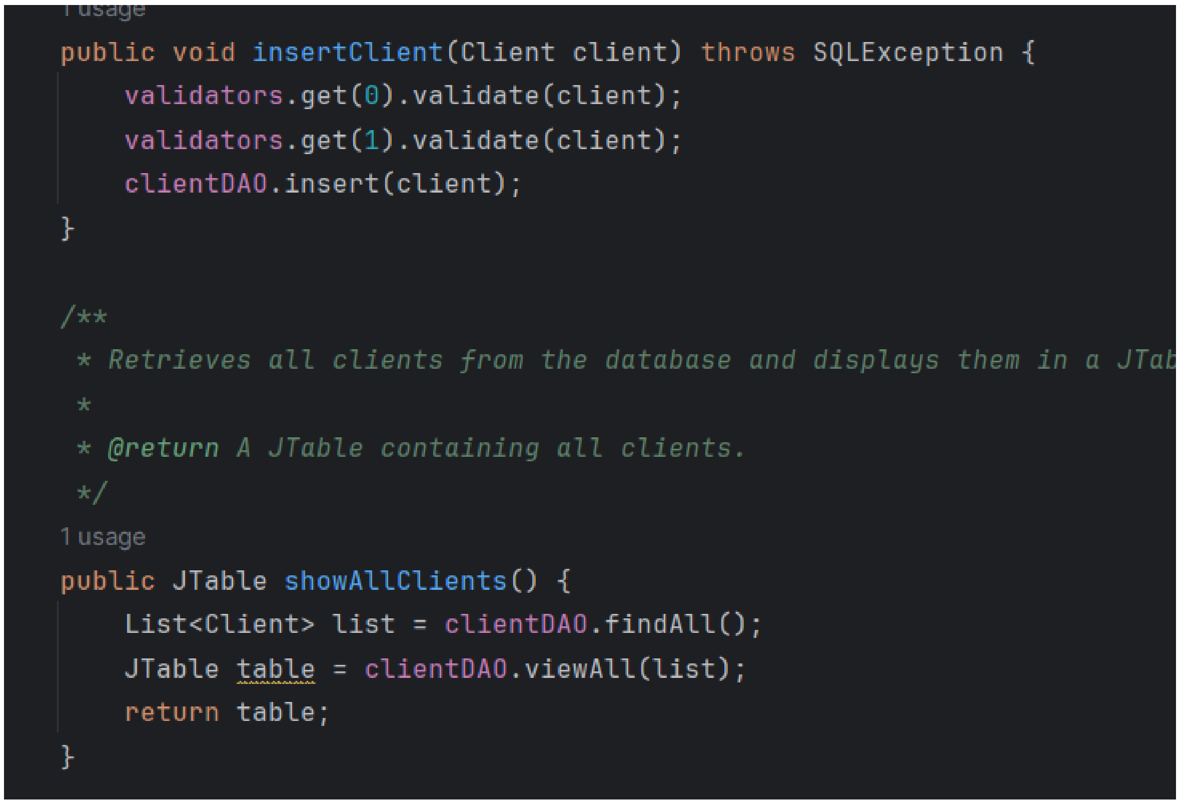
Used Data Structures:

- ArrayList

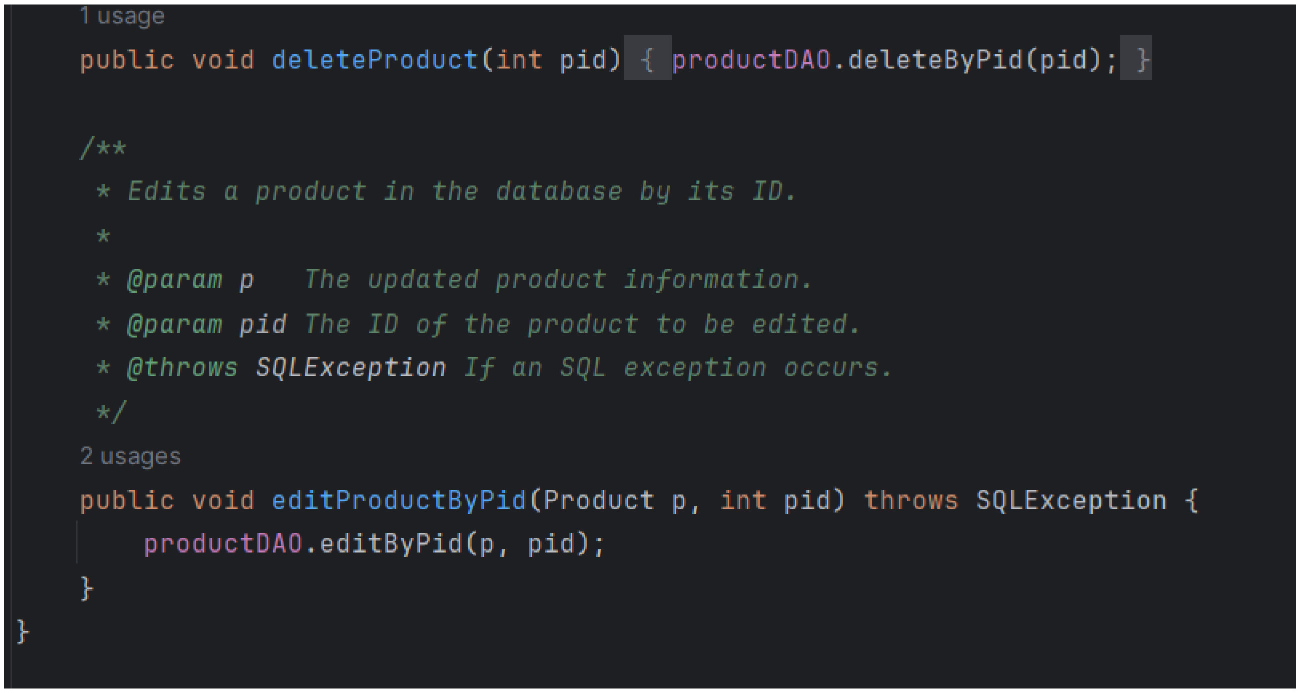
4. Implementation


```
public class ClientBLL {

    7 usages
    private List<Validator<Client>> validators;
    7 usages
    private ClientDAO clientDAO;
```

The ClientBLL class contains a list of Validators which are being used to validate different fields for Clients , like email or age , if the shop requires an age to create orders and a ClientDAO instance , to perfom database operations.


```
public void insertClient(Client client) throws SQLException {
    validators.get(0).validate(client);
    validators.get(1).validate(client);
    clientDAO.insert(client);
}

/**
 * Retrieves all clients from the database and displays them in a JTab
 *
 * @return A JTable containing all clients.
 */
1 usage
public JTable showAllClients() {
    List<Client> list = clientDAO.findAll();
    JTable table = clientDAO.viewAll(list);
    return table;
}
```

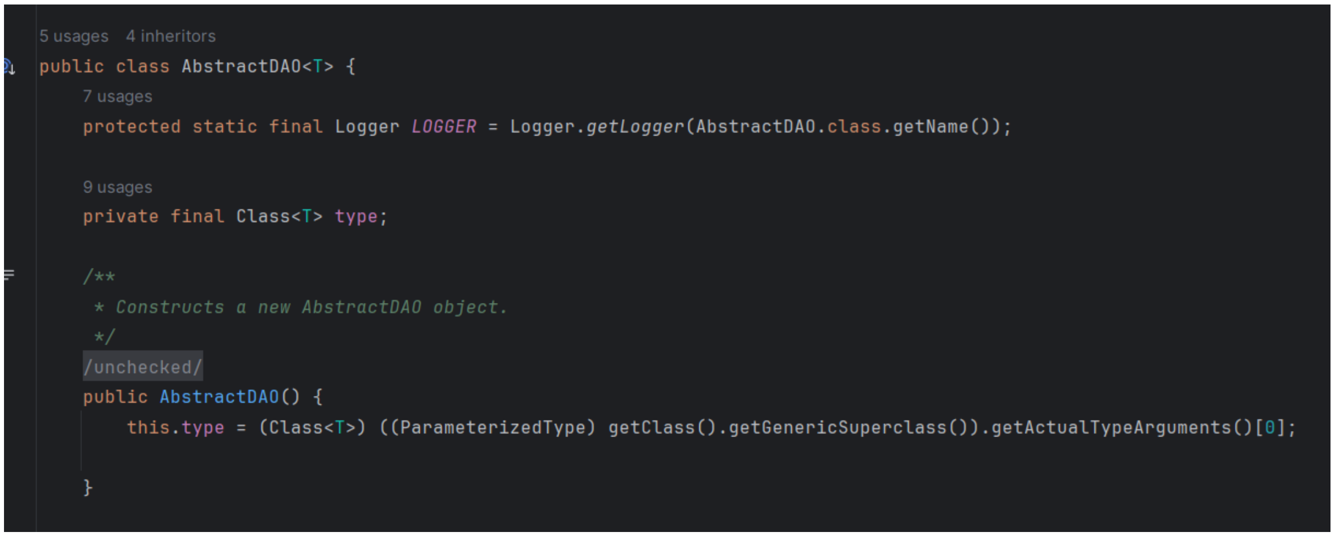
For inserting a client , the ClientBLL class only initializes the validators and uses the insert method from the ClientDAO.



```
    1 usage
    public void deleteProduct(int pid) { productDAO.deleteByPid(pid); }

    /**
     * Edits a product in the database by its ID.
     *
     * @param p   The updated product information.
     * @param pid The ID of the product to be edited.
     * @throws SQLException If an SQL exception occurs.
     */
    2 usages
    public void editProductByPid(Product p, int pid) throws SQLException {
        productDAO.editByPid(p, pid);
    }
}
```

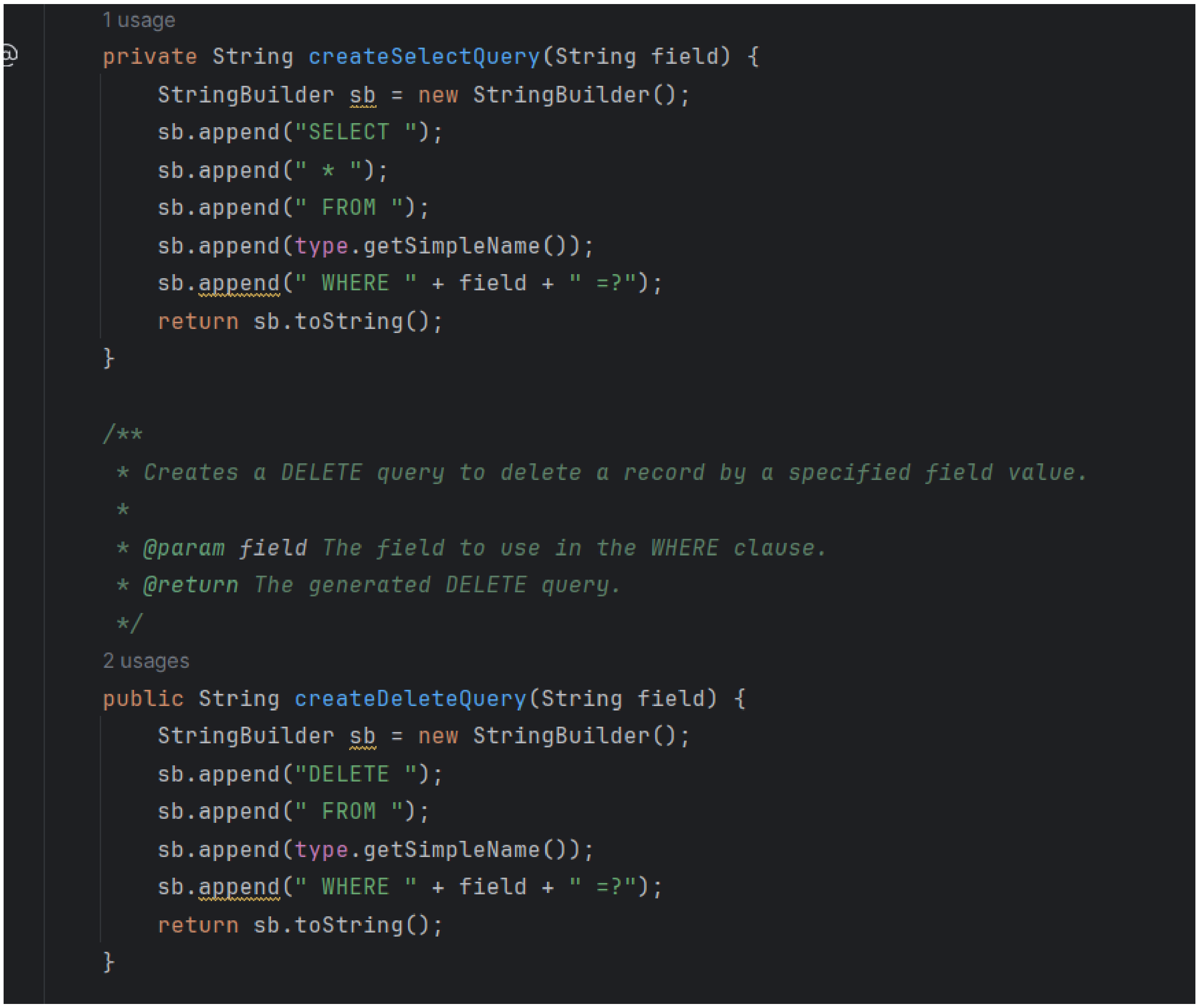
The ProductBLL is very similar to ClientBLL. For deleting an instance of a product from the database , it just uses the productDAO method deleteByPID. We want to delete by ID because this is the primary key of the Product table , so we don't unintentionally delete products what we want to keep.

The editProduct method requires a Product , which will be the „new" Product inserted into the table , at the same id.



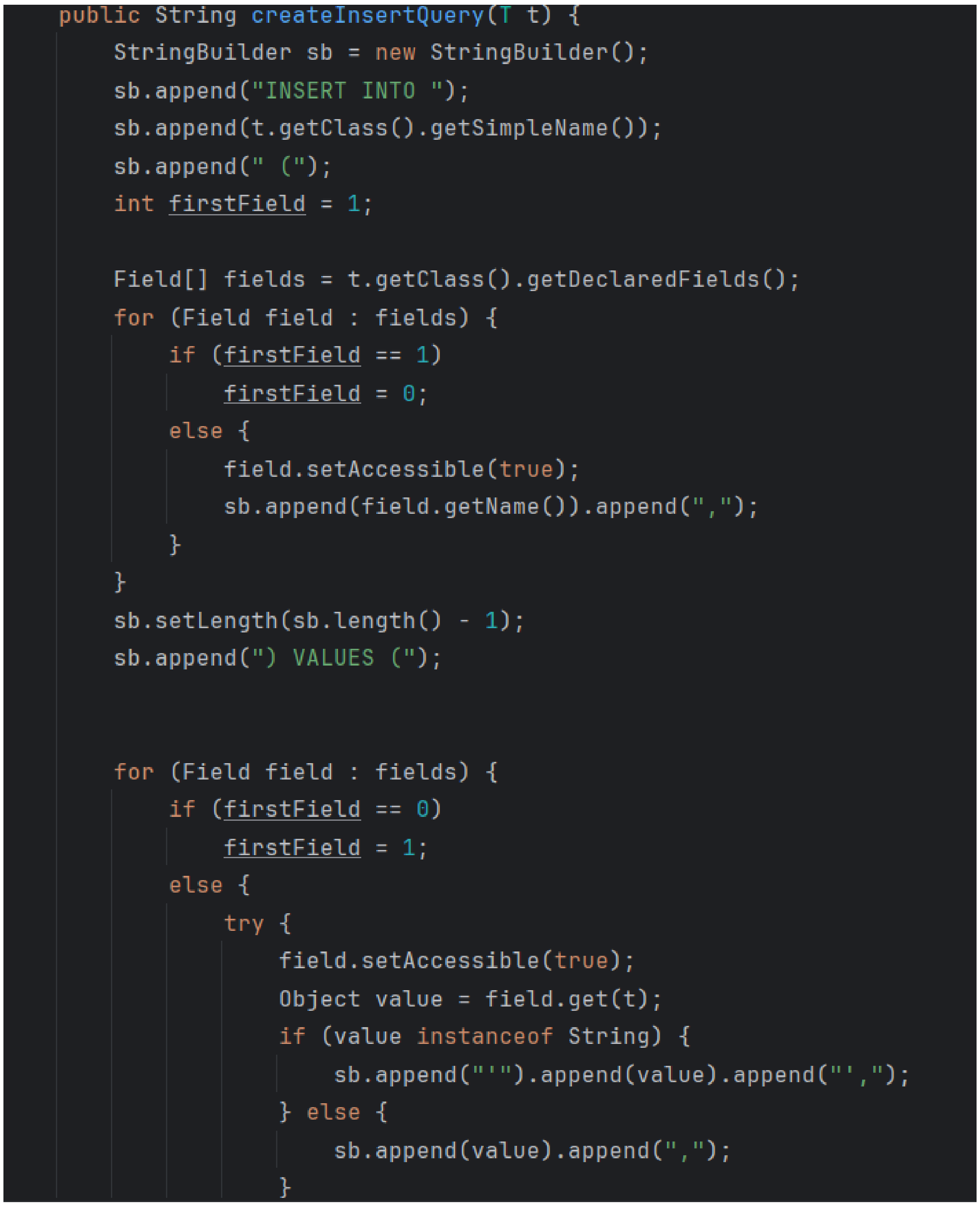
```
5 usages   4 inheritors
public class AbstractDAO<T> {
    7 usages
    protected static final Logger LOGGER = Logger.getLogger(AbstractDAO.class.getName());

    9 usages
    private final Class<T> type;

    /**
     * Constructs a new AbstractDAO object.
     */
    /unchecked/
    public AbstractDAO() {
        this.type = (Class<T>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[0];

    }
```

The abstractDAO class uses reflection techniques to create queries that work for Clients , Products and Orders , making the code easier to write and with less classes. It gets the type by using getClass.getName and then gets the field for the object given.



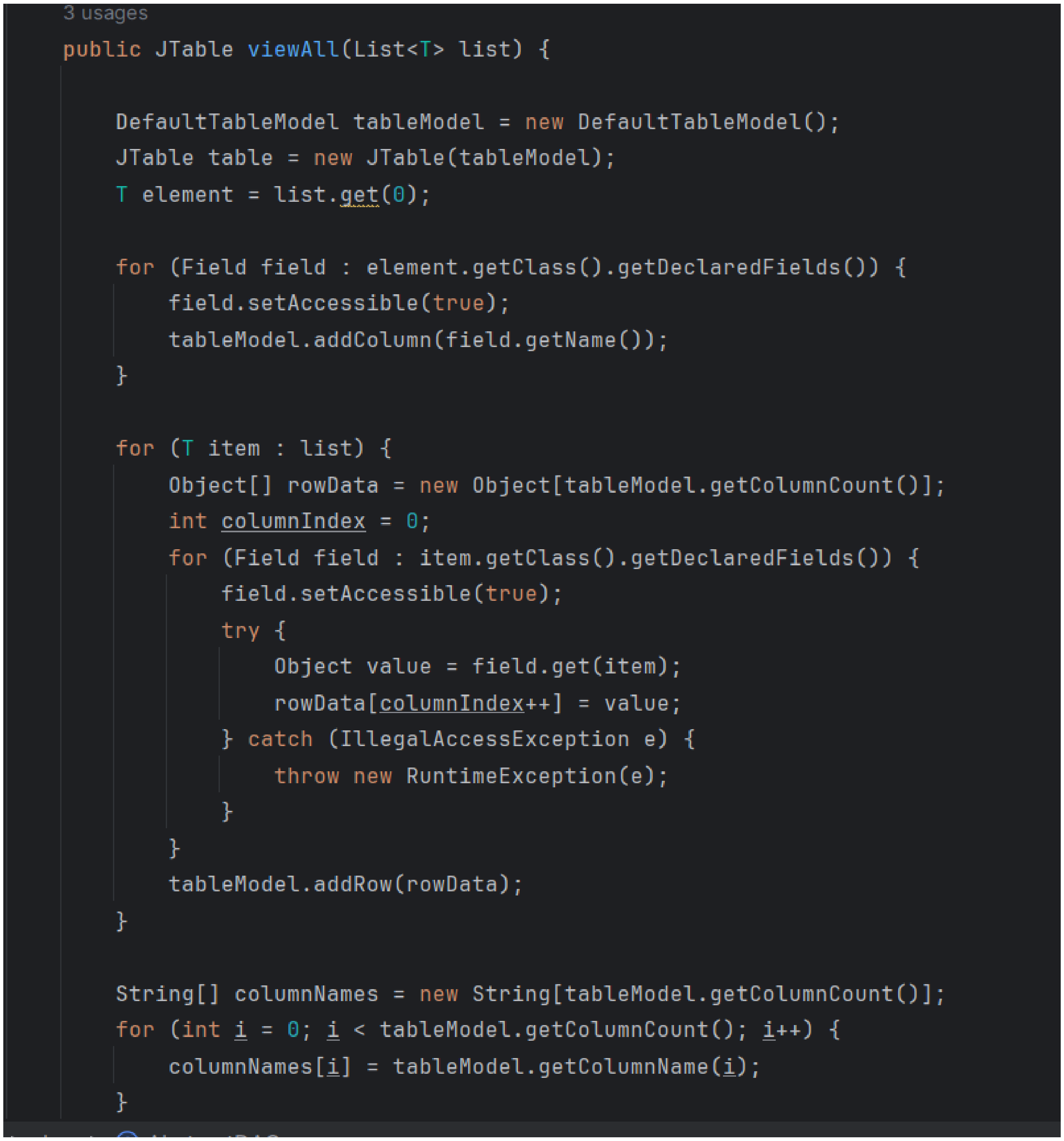
```
1 usage
private String createSelectQuery(String field) {
    StringBuilder sb = new StringBuilder();
    sb.append("SELECT ");
    sb.append(" * ");
    sb.append(" FROM ");
    sb.append(type.getSimpleName());
    sb.append(" WHERE " + field + " =?");
    return sb.toString();
}

/**
 * Creates a DELETE query to delete a record by a specified field value.
 *
 * @param field The field to use in the WHERE clause.
 * @return The generated DELETE query.
 */
2 usages
public String createDeleteQuery(String field) {
    StringBuilder sb = new StringBuilder();
    sb.append("DELETE ");
    sb.append(" FROM ");
    sb.append(type.getSimpleName());
    sb.append(" WHERE " + field + " =?");
    return sb.toString();
}
```

Here we create different querries that will be used to perform database operations.



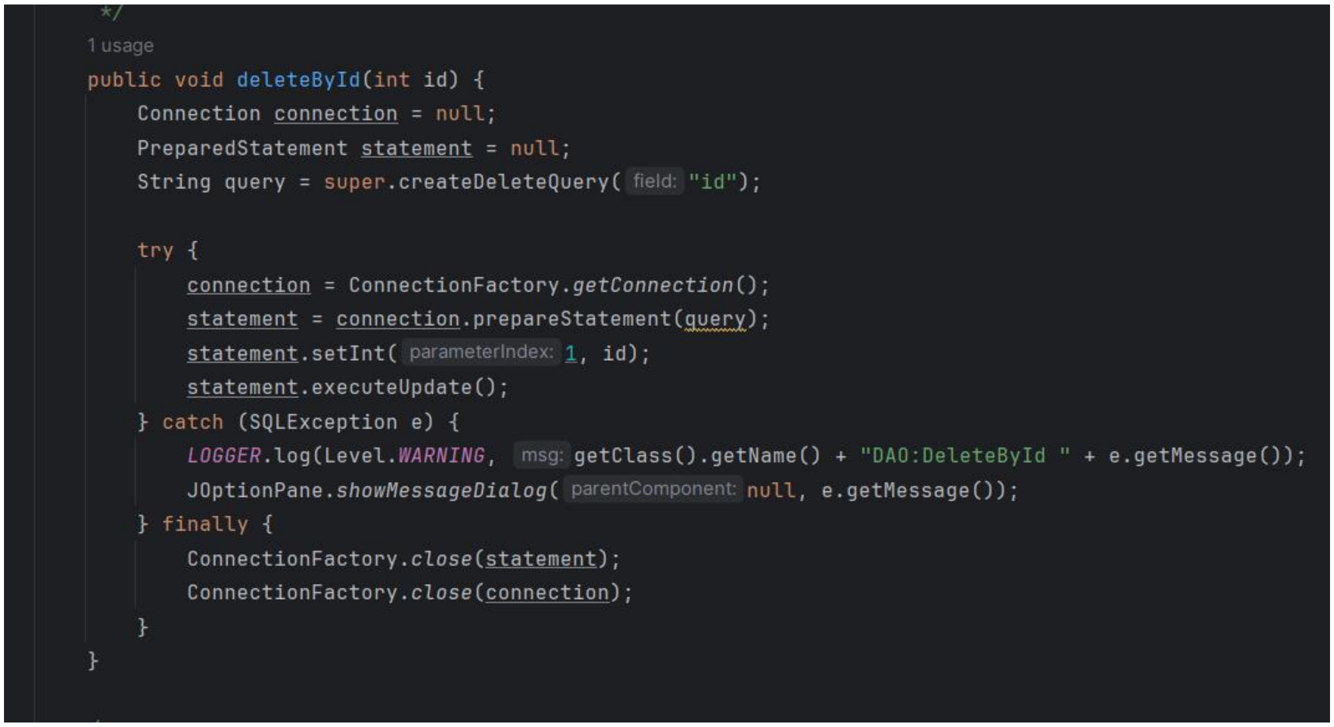
```java
    public String createInsertQuery(T t) {
        StringBuilder sb = new StringBuilder();
        sb.append("INSERT INTO ");
        sb.append(t.getClass().getSimpleName());
        sb.append(" (");
        int firstField = 1;

        Field[] fields = t.getClass().getDeclaredFields();
        for (Field field : fields) {
            if (firstField == 1)
                firstField = 0;
            else {
                field.setAccessible(true);
                sb.append(field.getName()).append(",");
            }
        }
        sb.setLength(sb.length() - 1);
        sb.append(") VALUES (");


        for (Field field : fields) {
            if (firstField == 0)
                firstField = 1;
            else {
                try {
                    field.setAccessible(true);
                    Object value = field.get(t);
                    if (value instanceof String) {
                        sb.append("'").append(value).append("',");
                    } else {
                        sb.append(value).append(",");
                    }
```

The createInsertQuerry uses reflection techniques to build an insert querry , by getting the class name and the fields.



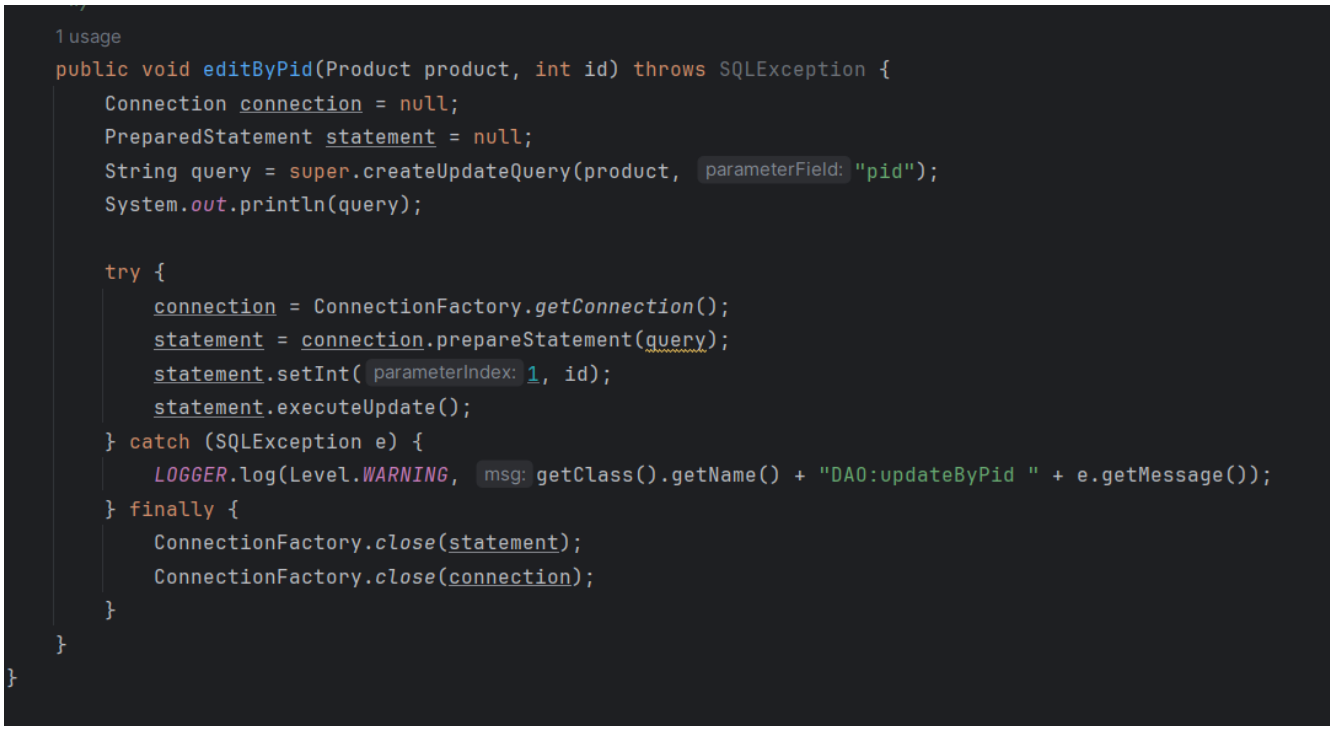
```
3 usages
public JTable viewAll(List<T> list) {

    DefaultTableModel tableModel = new DefaultTableModel();
    JTable table = new JTable(tableModel);
    T element = list.get(0);

    for (Field field : element.getClass().getDeclaredFields()) {
        field.setAccessible(true);
        tableModel.addColumn(field.getName());
    }

    for (T item : list) {
        Object[] rowData = new Object[tableModel.getColumnCount()];
        int columnIndex = 0;
        for (Field field : item.getClass().getDeclaredFields()) {
            field.setAccessible(true);
            try {
                Object value = field.get(item);
                rowData[columnIndex++] = value;
            } catch (IllegalAccessException e) {
                throw new RuntimeException(e);
            }
        }
        tableModel.addRow(rowData);
    }

    String[] columnNames = new String[tableModel.getColumnCount()];
    for (int i = 0; i < tableModel.getColumnCount(); i++) {
        columnNames[i] = tableModel.getColumnName(i);
    }
```

The viewAll method uses reflection techniques to create a generic class that contains the methods for accessing the DB (all tables except Log): 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.



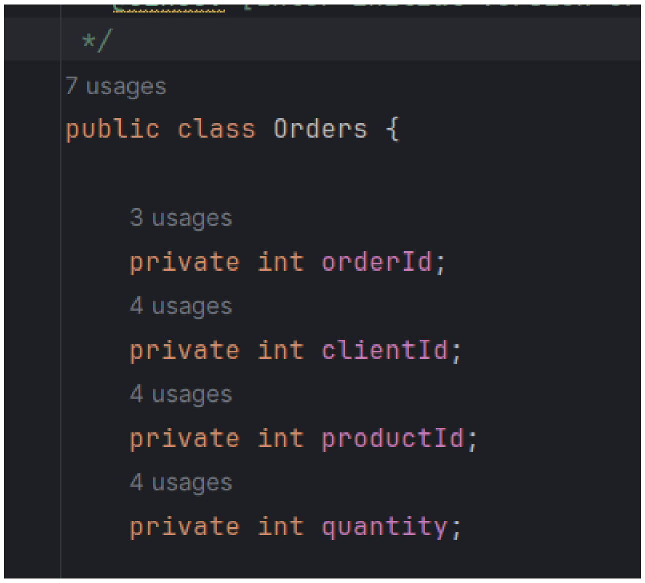
```
    */
    1 usage
    public void deleteById(int id) {
        Connection connection = null;
        PreparedStatement statement = null;
        String query = super.createDeleteQuery( field: "id");

        try {
            connection = ConnectionFactory.getConnection();
            statement = connection.prepareStatement(query);
            statement.setInt( parameterIndex: 1, id);
            statement.executeUpdate();
        } catch (SQLException e) {
            LOGGER.log(Level.WARNING,  msg: getClass().getName() + "DAO:DeleteById " + e.getMessage());
            JOptionPane.showMessageDialog( parentComponent: null, e.getMessage());
        } finally {
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }
    }
```

The ClientDAO contains specific queries for the Client class. Here is the deleteById method where the specific field is substituted with the field id so the delete works on the id field.



```
    1 usage
    public void editByPid(Product product, int id) throws SQLException {
        Connection connection = null;
        PreparedStatement statement = null;
        String query = super.createUpdateQuery(product,  parameterField: "pid");
        System.out.println(query);

        try {
            connection = ConnectionFactory.getConnection();
            statement = connection.prepareStatement(query);
            statement.setInt( parameterIndex: 1, id);
            statement.executeUpdate();
        } catch (SQLException e) {
            LOGGER.log(Level.WARNING,  msg: getClass().getName() + "DAO:updateByPid " + e.getMessage());
        } finally {
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }
    }
}
```

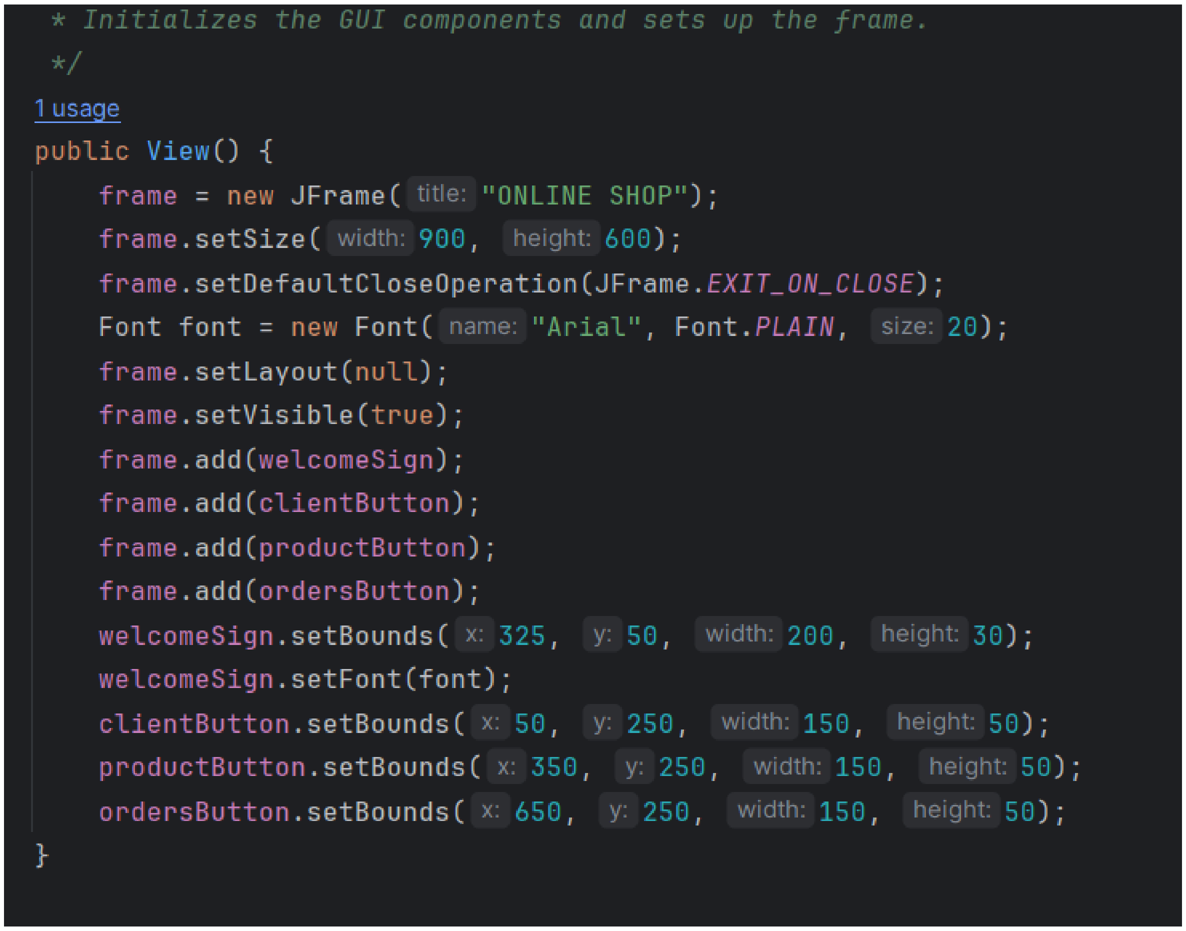
The productDAO class is similar to the clientDAO class. Here we can see the specific editByPid method that edits with a given Product and at the given Pid.



```
 */
7 usages
public class Orders {

    3 usages
    private int orderId;
    4 usages
    private int clientId;
    4 usages
    private int productId;
    4 usages
    private int quantity;
```

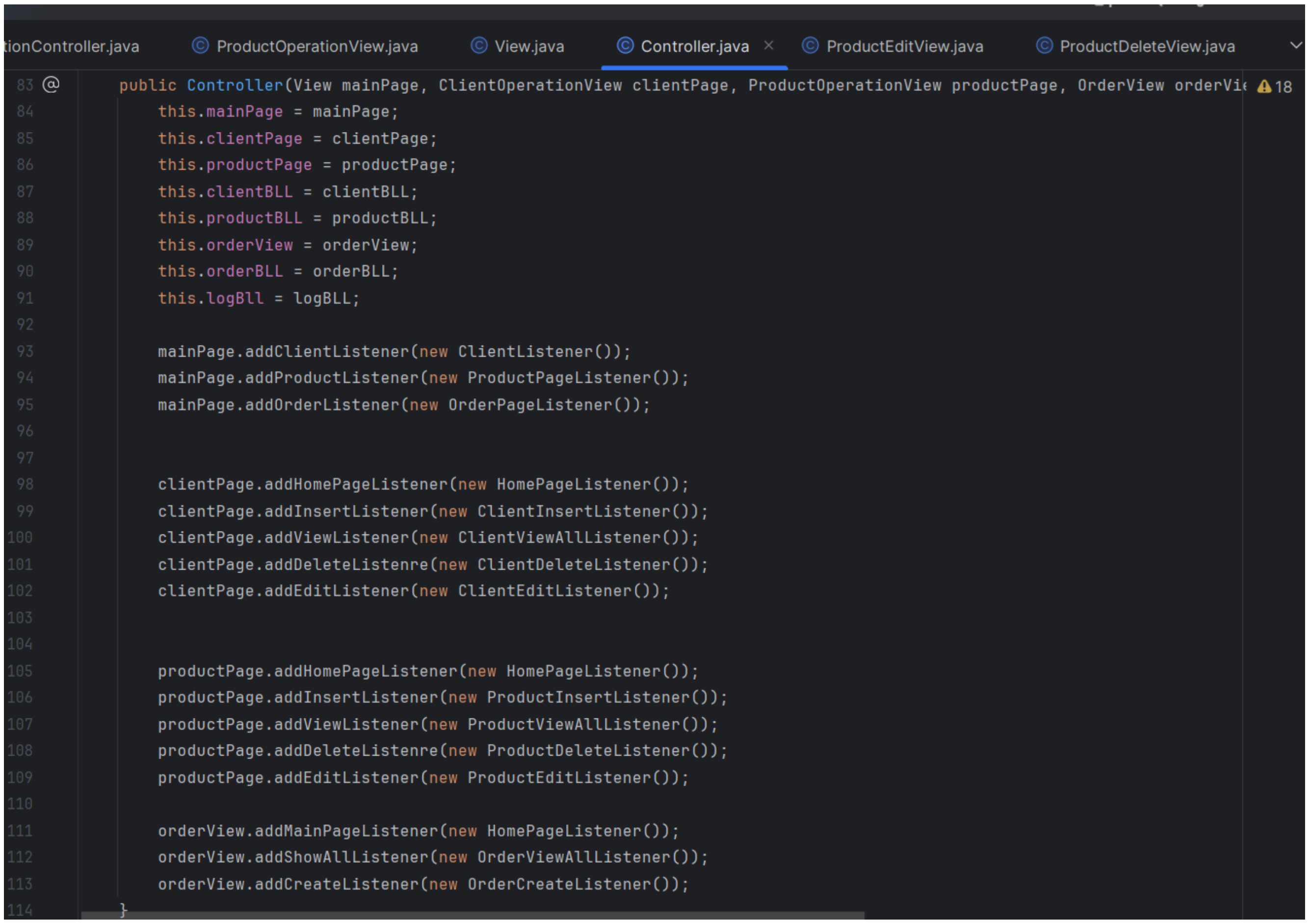
The Orders class contains the orderId field , which is the primary key in the Orders field , the clientId , the productId and the quantity wanted.
The methods used in this class are only constructors , getters and setters.

Client , Product are simillar , the only difference being their specific fields.



```java
 * Initializes the GUI components and sets up the frame.
 */
1 usage
public View() {
    frame = new JFrame( title: "ONLINE SHOP");
    frame.setSize( width: 900,  height: 600);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Font font = new Font( name: "Arial", Font.PLAIN,  size: 20);
    frame.setLayout(null);
    frame.setVisible(true);
    frame.add(welcomeSign);
    frame.add(clientButton);
    frame.add(productButton);
    frame.add(ordersButton);
    welcomeSign.setBounds( x: 325,  y: 50,  width: 200,  height: 30);
    welcomeSign.setFont(font);
    clientButton.setBounds( x: 50,  y: 250,  width: 150,  height: 50);
    productButton.setBounds( x: 350,  y: 250,  width: 150,  height: 50);
    ordersButton.setBounds( x: 650,  y: 250,  width: 150,  height: 50);
}
```

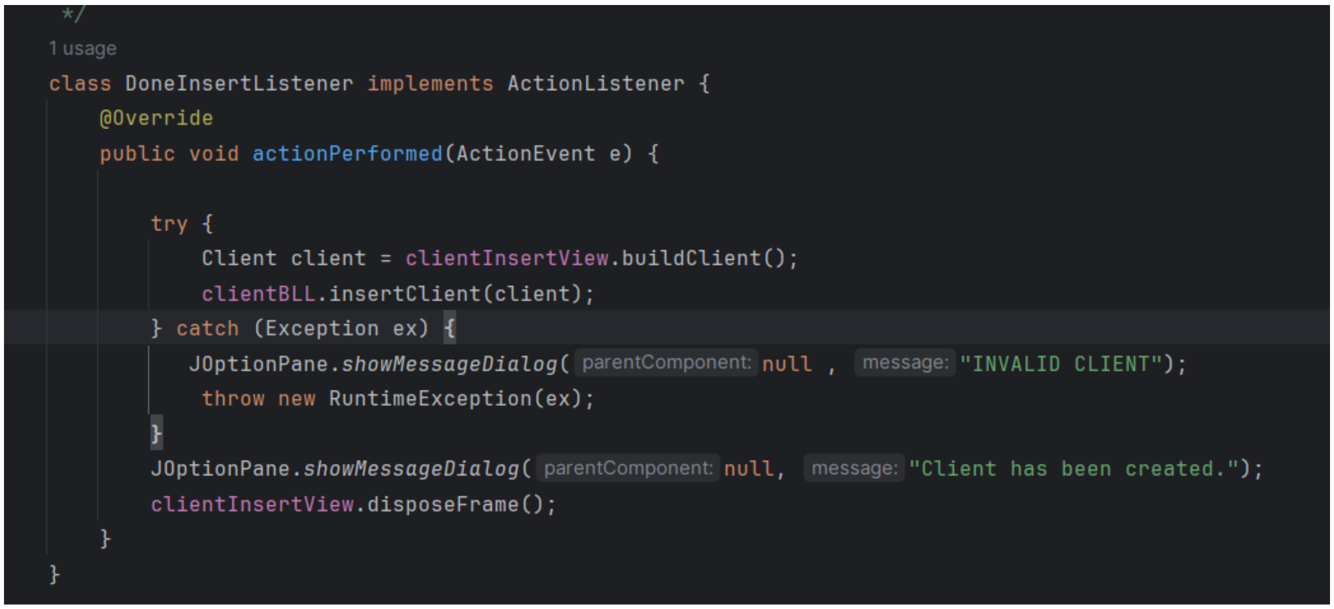
The View class contains the design for the HomePage. It contains different JLabels and JButtons.



ionController.java | ProductOperationView.java | View.java | Controller.java | ProductEditView.java | ProductDeleteView.java

```java
    public Controller(View mainPage, ClientOperationView clientPage, ProductOperationView productPage, OrderView orderVi
        this.mainPage = mainPage;
        this.clientPage = clientPage;
        this.productPage = productPage;
        this.clientBLL = clientBLL;
        this.productBLL = productBLL;
        this.orderView = orderView;
        this.orderBLL = orderBLL;
        this.logBll = logBLL;

        mainPage.addClientListener(new ClientListener());
        mainPage.addProductListener(new ProductPageListener());
        mainPage.addOrderListener(new OrderPageListener());


        clientPage.addHomePageListener(new HomePageListener());
        clientPage.addInsertListener(new ClientInsertListener());
        clientPage.addViewListener(new ClientViewAllListener());
        clientPage.addDeleteListenre(new ClientDeleteListener());
        clientPage.addEditListener(new ClientEditListener());


        productPage.addHomePageListener(new HomePageListener());
        productPage.addInsertListener(new ProductInsertListener());
        productPage.addViewListener(new ProductViewAllListener());
        productPage.addDeleteListenre(new ProductDeleteListener());
        productPage.addEditListener(new ProductEditListener());

        orderView.addMainPageListener(new HomePageListener());
        orderView.addShowAllListener(new OrderViewAllListener());
        orderView.addCreateListener(new OrderCreateListener());
    }
```

The Controller class contains ActionListeners for several buttons used in the application , most of them being the main interface buttons (not the ones on the forms that pop up).



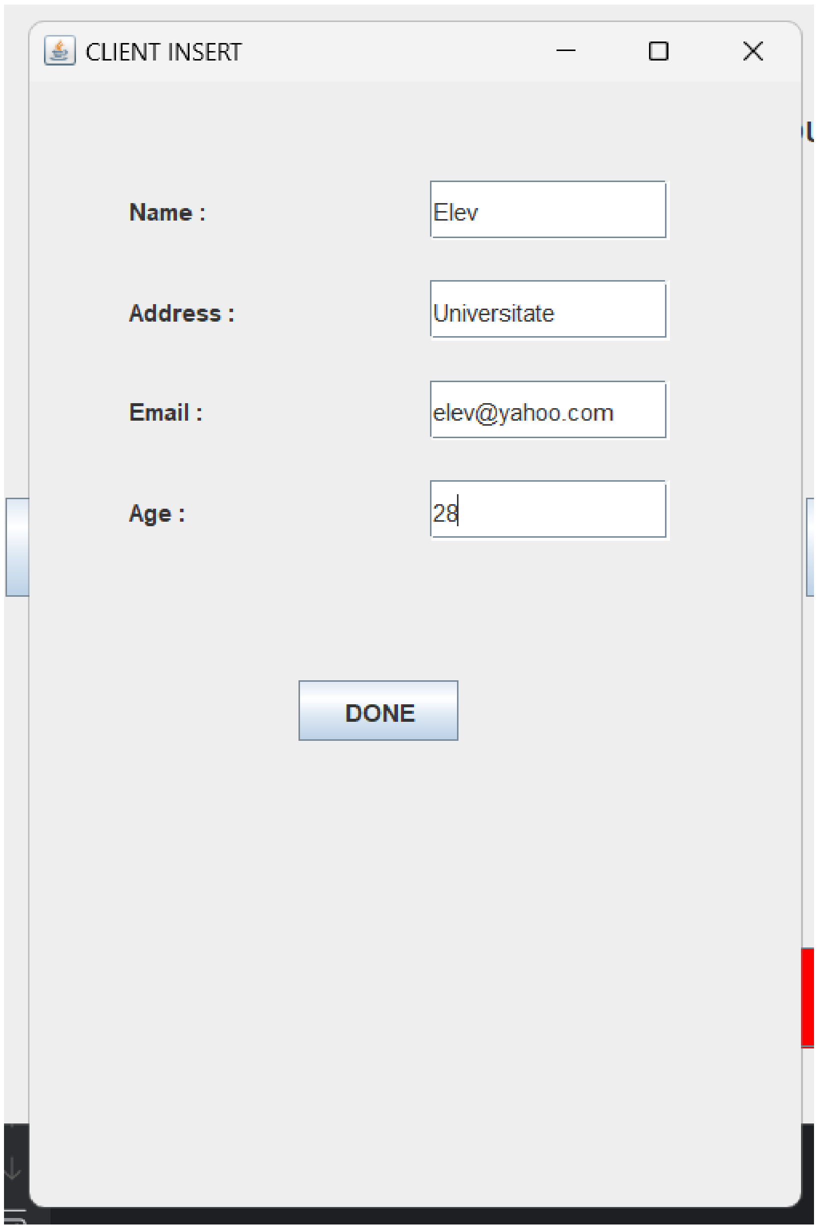
```java
 */
1 usage
class DoneInsertListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {

        try {
            Client client = clientInsertView.buildClient();
            clientBLL.insertClient(client);
        } catch (Exception ex) {
           JOptionPane.showMessageDialog( parentComponent: null ,  message: "INVALID CLIENT");
            throw new RuntimeException(ex);
        }
        JOptionPane.showMessageDialog( parentComponent: null,  message: "Client has been created.");
        clientInsertView.disposeFrame();
    }
}
```

An example of an ActionListener from a form. It uses the ClientBLL method and eventually treats different exception cases.

5. Results

The application works correctly for all the operations.

1.Insert



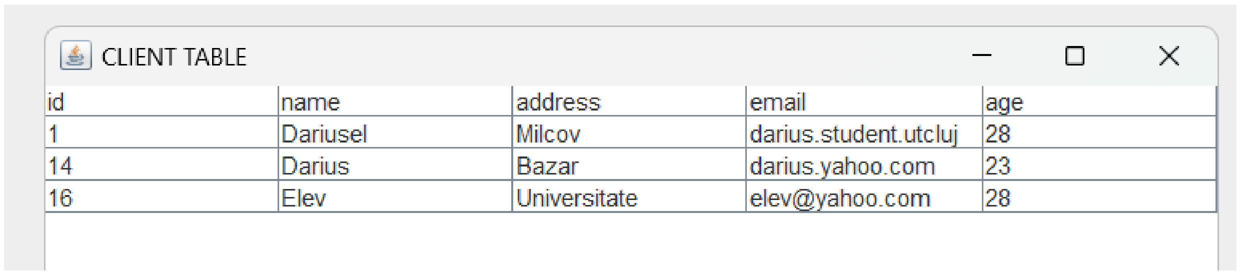
Name : Elev

Address : Universitate

Email : elev@yahoo.com

Age : 28

Here we try to insert a client with the fields in the photo.

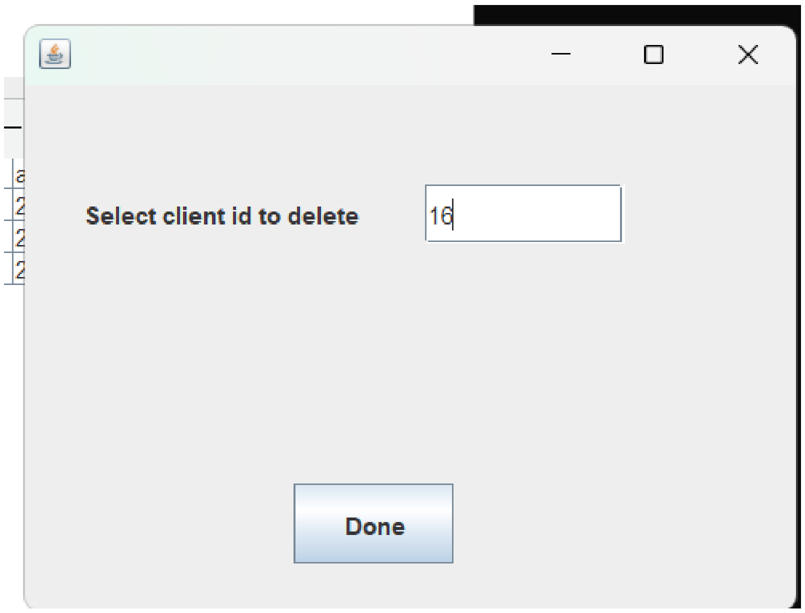


CLIENT TABLE

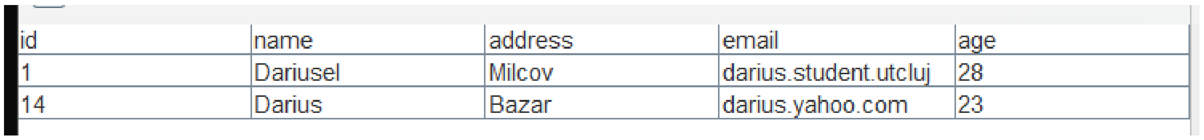
id	name	address	email	age
1	Dariusel	Milcov	darius.student.utcluj	28
14	Darius	Bazar	darius.yahoo.com	23
16	Elev	Universitate	elev@yahoo.com	28

We can see that the client is correctly inserted and given the ID 16.

2.Delete



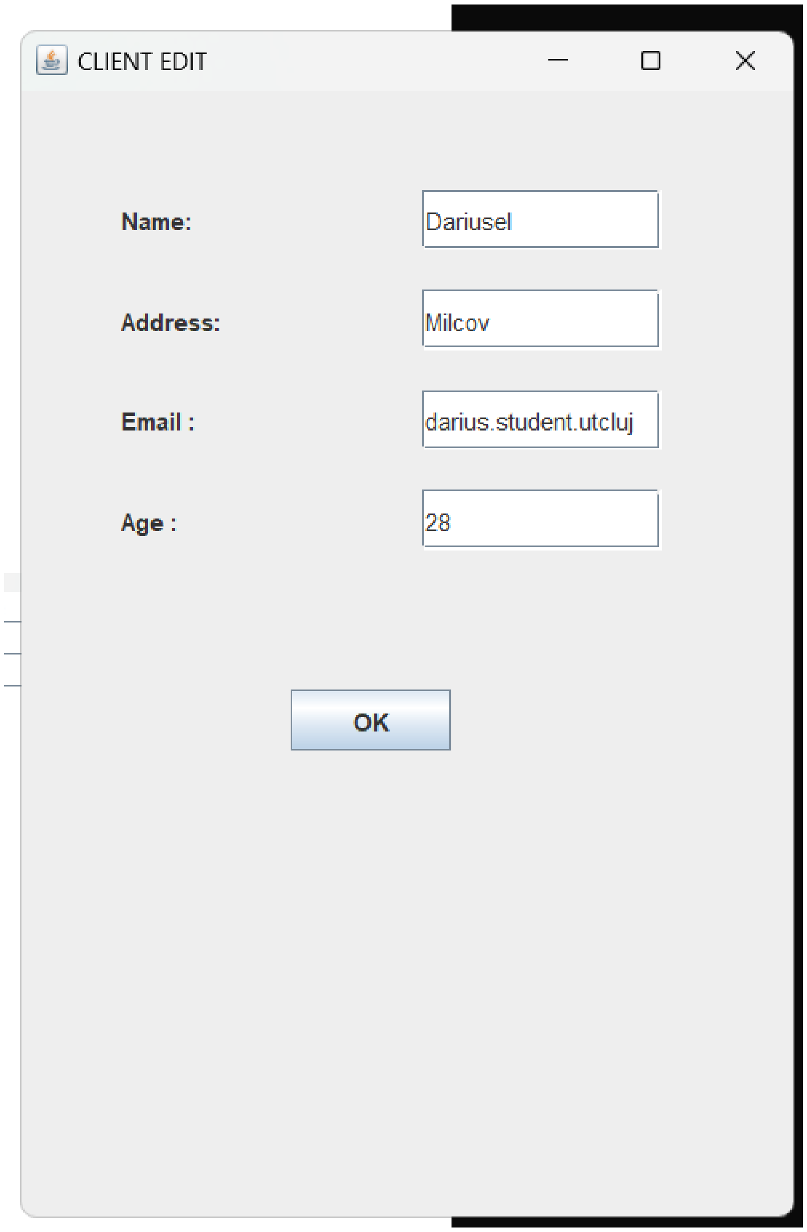
Here we try to delete the client we just added.



id	name	address	email	age
1	Dariusel	Milcov	darius.student.utcluj	28
14	Darius	Bazar	darius.yahoo.com	23

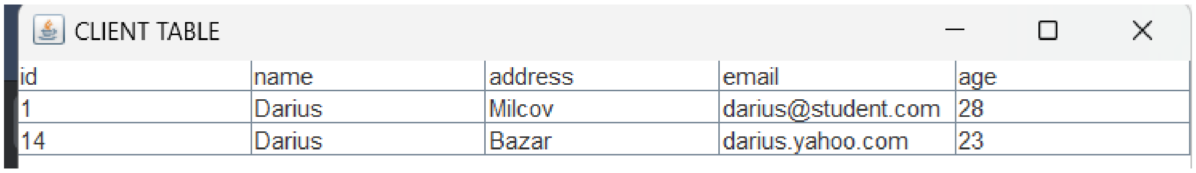
We can see that the client no longer exists in the database.

3.EDIT


CLIENT EDIT

Name:	Dariusel
Address:	Milcov
Email :	darius.student.utcluj
Age :	28

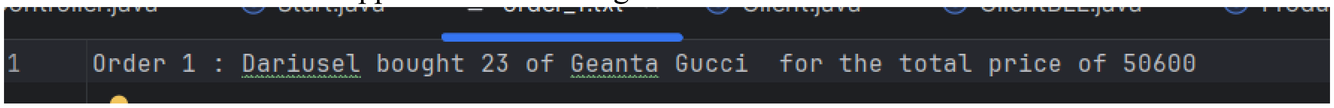
Here we try to edit the client with ID 1.


CLIENT TABLE

id	name	address	email	age
1	Darius	Milcov	darius@student.com	28
14	Darius	Bazar	darius.yahoo.com	23

We can see that the changes have been done.

Also another result of this application are the log files created after each order.



```
Order 1 : Dariusel bought 23 of Geanta Gucci  for the total price of 50600
```

6. Conclusions

The conclusion of this assignment is the creation of a fully functional **orders management application.**

From this homework assignment I have learned how to work with JDBC , how to create queries in java that work in our database. The hardest part for me in this homework was using reflection techniques to build different methods , but with a little research I have got over it very easily.

For the **future developments** , the orders management system could have an even easier to use interface , create sorting methods to sort the clients or the products. Another future development would be a graphical interface with images of the product and maybe a log in page.

7. Bibliography

1. https://dsrl.eu/courses/pt/
2. https://users.utcluj.ro/~igiosan/teaching_poo.html
3. https://www.programiz.com/c-programming/c-file-input-output#google_vignette
4. https://ioflood.com/blog/java-record/