ORDERS MANAGEMENT APP

Horvath Ariana-Cristine

Table of Contents

1. Assignment objective

1.1 Main Objective

Consider an Order Management application for processing client orders for a warehouse. Relational databases are used to store the products, the clients and the orders. Furthermore, the application should be structured in packages using a layered architecture and should use (minimally) the following classes:

• Model classes - represent the data models of the application
• Business Logic classes - contain the application logic
• Presentation classes – GUI related classes
• Data access classes - classes that contain the access to the database

Note: Other classes and packages can be added to implement the full functionality of the application.

1.2 Sub-objectives

- **Analyze** the problem and identify requirements – how the application should work, what it should do and what are the use cases (described in chapter 2. Analysis)
- **Design** the Orders Manager – which architectural pattern should be used, how the packages, classes and methods are divided (described in chapter 3. Design)
- **Implement** the Orders Manager – the Java code written for the implementation (described in chapter 4.Implementation)
- **Test** the Orders Manager – testing the application to check if it works properly (described in chapter 5.Results)

2. Analysis

2.1 Requirements

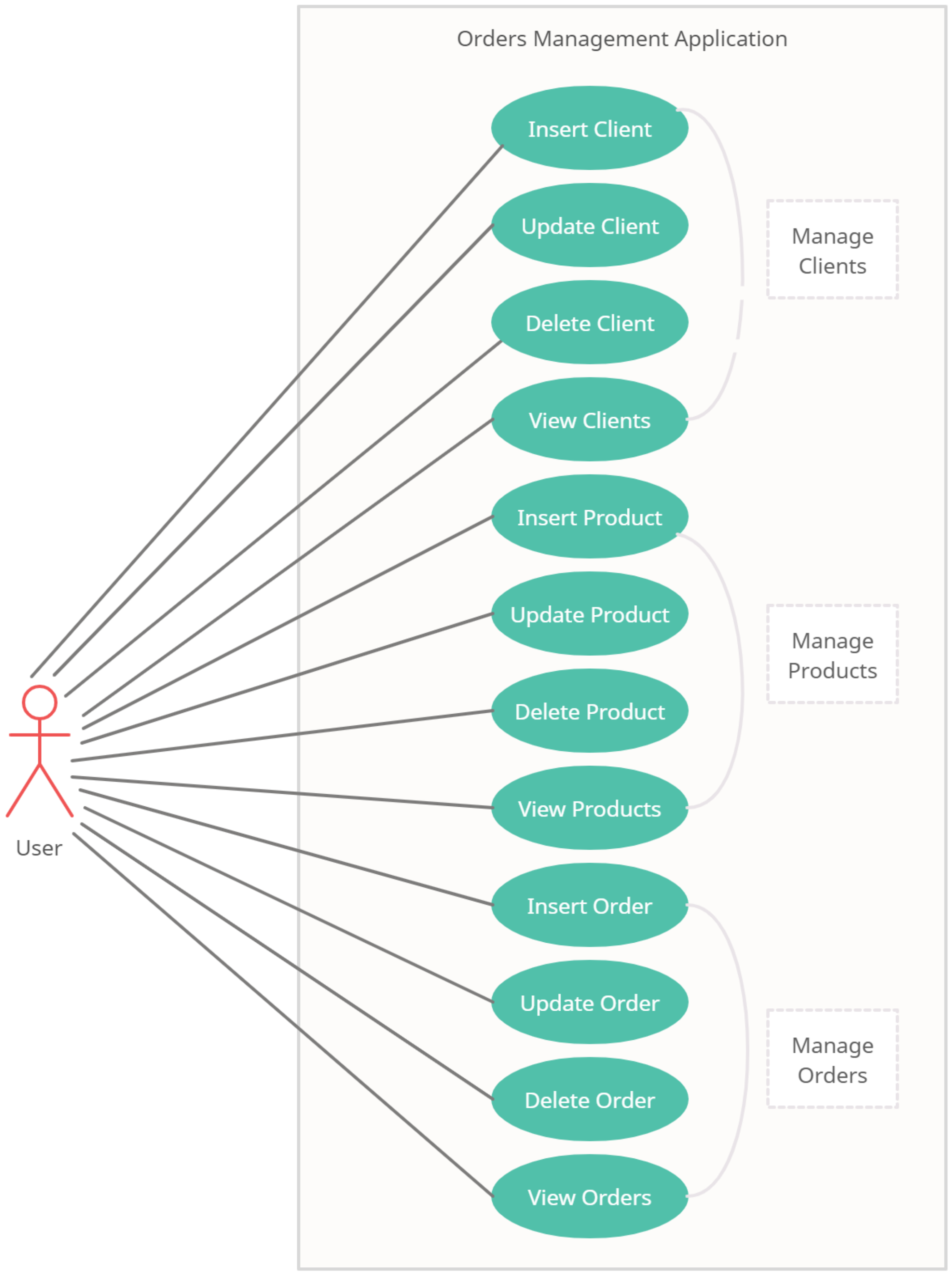
2.1.1 Functional requirements

- The orders management application should allow users to select the table/entity to be processed, i.e. Client, Product and Order.
- For each of these, the application should allow user to perform the CRUD operations (Create, Read, Update and Delete) that will be saved in the database.
- The application should validate input and make modifications in the database only if the clients, products or orders are valid.
- The application should create a bill (in a .txt file) for every order made.

2.1.2 Non-Functional requirements

- The orders management application should be intuitive and easy to use by the user.
- The orders management application should not allow the user to introduce invalid input.
- The orders management application should warn the user if the input is not valid or empty.

2.2 <u>Use-cases and scenarios</u>



[4]

- **Use Case**: Insert Client

Primary Actor: user

Main Success Scenario:

1. The user selects “Manage Clients”.
2. The user inserts the data for a client: name, email, phone number and address (id is auto-generated so it is left empty in this case).
3. The user selects the insert client operation by pressing a button.
4. The application saves the client in the database.
5. An information message is shown to let the user know that the operation was performed successfully.

Alternative Sequence: Incorrect input

- The user leaves empty one or more of the text fields (name, email, phone number or address) or it inserts invalid data (name doesn’t contain only letters, email, phone or address is not of valid pattern);
- A pop-up with an error message is displayed;
- The user presses the “Ok” button or “x”(Close);
- The scenario returns to step 1.

- **Use Case**: Update Client

Primary Actor: user

Main Success Scenario:

1. The user selects “Manage Clients”.
2. The user inserts the data for a client to be updated: id, name, email, phone number and address.
3. The user selects the update client operation by pressing a button.
4. The application updates the client in the database.
5. An information message is shown to let the user know that the operation was performed successfully.

Alternative Sequence: Incorrect input or inexistent client

- The user leaves empty one or more of the text fields (id, name, email, phone number or address) or it inserts invalid data (id is not an integer, name doesn’t contain only letters, email, phone or address is not of valid pattern);
- The user inserts an ID for a client which doesn’t exist
- A pop-up with an error message is displayed;
- The user presses the “Ok” button or “x”(Close);
- The scenario returns to step 1.

- **Use Case**: Delete Client

Primary Actor: user

Main Success Scenario:

1. The user selects “Manage Clients”.
2. The user inserts the id for the client to be deleted.
3. The user selects the delete client operation by pressing a button.
4. The application deletes the client from the database.
5. An information message is shown to let the user know that the operation was performed successfully.

Alternative Sequence: Incorrect input or inexistent client

- The user leaves empty the id field or the introduced id is not an integer or the user inserts an ID for a client which doesn’t exist;
- A pop-up with an error message is displayed;
- The user presses the “Ok” button or “x”(Close);
- The scenario returns to step 1.

- **Use Case**: View Clients

Primary Actor: user

Main Success Scenario:

6. The user selects “Manage Clients”.
7. The user selects the view clients operations by pressing a button.
8. The user sees a table with the existent clients and their information (id, name, email, phone, address) – if there are no clients the table is empty and contains only the headers.

Alternative Sequence: none

- **Use Case**: Insert Product

Primary Actor: user

Main Success Scenario:

1. The user selects “Manage Products”.
2. The user inserts the data for a product: name and stock (id is auto-generated so it is left empty in this case).
3. The user selects the insert product operation by pressing a button.
4. The application saves the product in the database.
5. An information message is shown to let the user know that the operation was performed successfully.

Alternative Sequence: Incorrect input

- The user leaves empty one or more of the text fields (name or stock) or it inserts invalid data (name doesn’t contain only letters, stock is not a positive integer);
- A pop-up with an error message is displayed;
- The user presses the “Ok” button or “x”(Close);
- The scenario returns to step 1.

- **Use Case**: Update Product

Primary Actor: user

Main Success Scenario:

1. The user selects “Manage Products”.
2. The user inserts the data for a product to be updated: id, name and stock.
3. The user selects the update product operation by pressing a button.
4. The application updates the product in the database.
5. An information message is shown to let the user know that the operation was performed successfully.

Alternative Sequence: Incorrect input or inexistent product

- The user leaves empty one or more of the text fields (id, name or stock) or it inserts invalid data (id or stock is not a positive integer, name doesn’t contain only letters);
- The user inserts an ID for a product which doesn’t exist
- A pop-up with an error message is displayed;
- The user presses the “Ok” button or “x”(Close);
- The scenario returns to step 1.

- **Use Case**: Delete Product

Primary Actor: user

Main Success Scenario:

1. The user selects “Manage Products”.
2. The user inserts the id for the product to be deleted.
3. The user selects the delete product operation by pressing a button.

4. The application deletes the product from the database.
5. An information message is shown to let the user know that the operation was performed successfully.

Alternative Sequence: Incorrect input or inexistent client

- The user leaves empty the id field or the introduced id is not a positive integer or the user inserts an ID for a product which doesn't exist;
- A pop-up with an error message is displayed;
- The user presses the "Ok" button or "x"(Close);
- The scenario returns to step 1.

- **Use Case**: View Products

Primary Actor: user

Main Success Scenario:

1. The user selects "Manage Products".
2. The user selects the view products operations by pressing a button.
3. The user sees a table with the existent products and their information (id, name and stock) – if there are no products the table is empty and contains only the headers.

Alternative Sequence: none

- **Use Case**: Insert Order

Primary Actor: user

Main Success Scenario:

1. The user selects "Manage Orders".
2. The user selects the client id and product id from a list of existing objects (order id is auto-generated).
3. The user inserts the quantity of the product he wants to order.
4. The user selects the insert order operation by pressing a button.
5. The application saves the order in the database and decrements the stock of the product.
6. An information message is shown to let the user know that the operation was performed successfully.

Alternative Sequence: Incorrect input or not enough stock

- The user leaves empty the quantity text fields or it inserts invalid data (quantity is not a positive integer);
- The user tries to order a quantity of products which are not available in the stock;
- A pop-up with an error message is displayed;
- The user presses the "Ok" button or "x"(Close);
- The scenario returns to step 1.

- **Use Case**: Update Order

Primary Actor: user

Main Success Scenario:

1. The user selects "Manage Orders".
2. The user inserts the data for an order to be updated: id, client id, product id and quantity (from the logical point of view, only the quantity should be modified for an order).
3. The user selects the update order operation by pressing a button.
4. The application updates the order in the database and modifies the product stock accordingly.
5. An information message is shown to let the user know that the operation was performed successfully.

Alternative Sequence: Incorrect input, inexistent order or not enough stock

- The user leaves empty one or more of the text fields (id or quantity) or it inserts invalid data (id or quantity is not a positive integer);
- The user inserts an ID for an order which doesn't exist;
- The user tries to order a quantity of products which are not available in the stock;

- A pop-up with an error message is displayed;
- The user presses the "Ok" button or "x"(Close);
- The scenario returns to step 1.

- **Use Case**: Delete Order

Primary Actor: user

Main Success Scenario:

1. The user selects "Manage Orders".
2. The user inserts the id for the order to be deleted.
3. The user selects the delete order operation by pressing a button.
4. The application deletes the order from the database and increments the product stock.
5. An information message is shown to let the user know that the operation was performed successfully.

Alternative Sequence: Incorrect input or inexistent order

- The user leaves empty the id field or the introduced id is not a positive integer or the user inserts an ID for an order which doesn't exist;
- A pop-up with an error message is displayed;
- The user presses the "Ok" button or "x"(Close);
- The scenario returns to step 1.

- **Use Case**: View Orders

Primary Actor: user

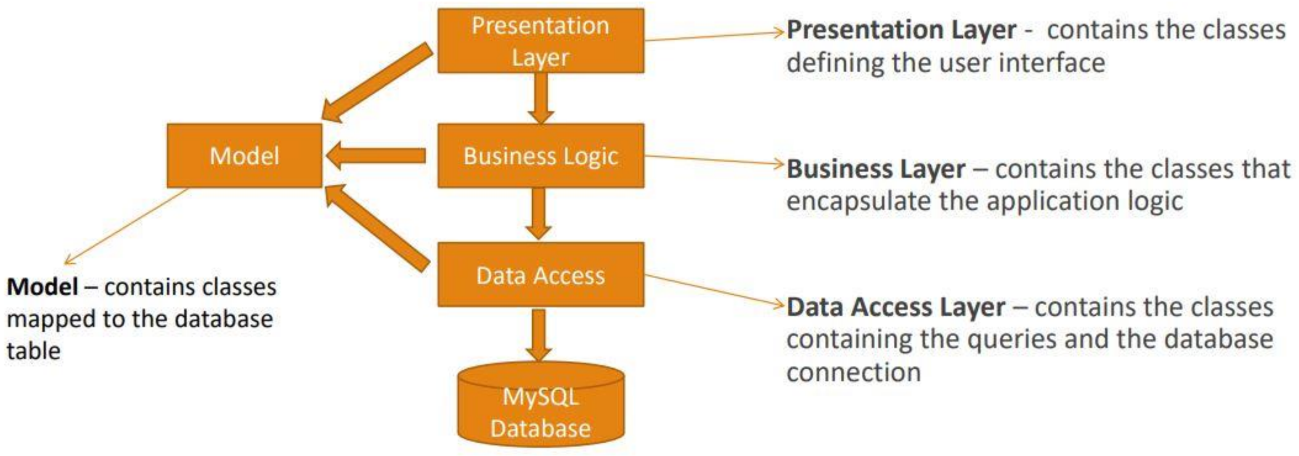
Main Success Scenario:

1. The user selects "Manage Orders".
2. The user selects the view orders operations by pressing a button.
3. The user sees a table with the existent orders and their information (order id, client, product and quantity) – if there are no orders the table is empty and contains only the headers.

Alternative Sequence: none

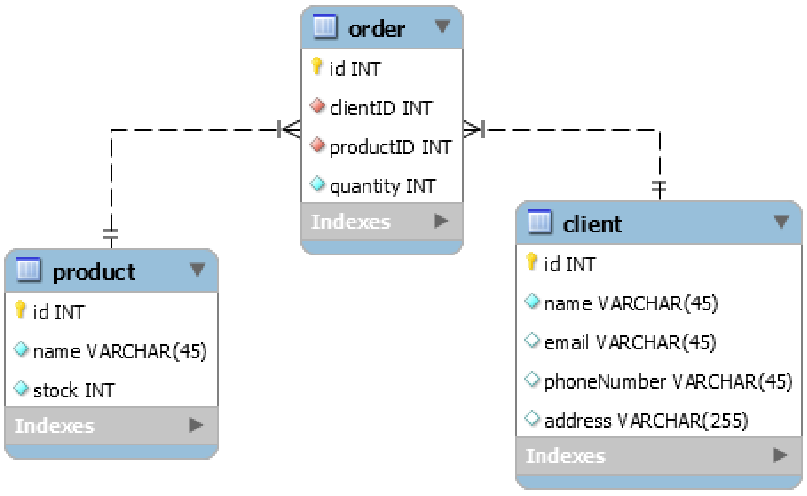
3. Design

In the design of the Orders Management Application, the layered architectural pattern was used, which splits the application in different layers and each layer has a special purpose and calls functions of the layers below it:

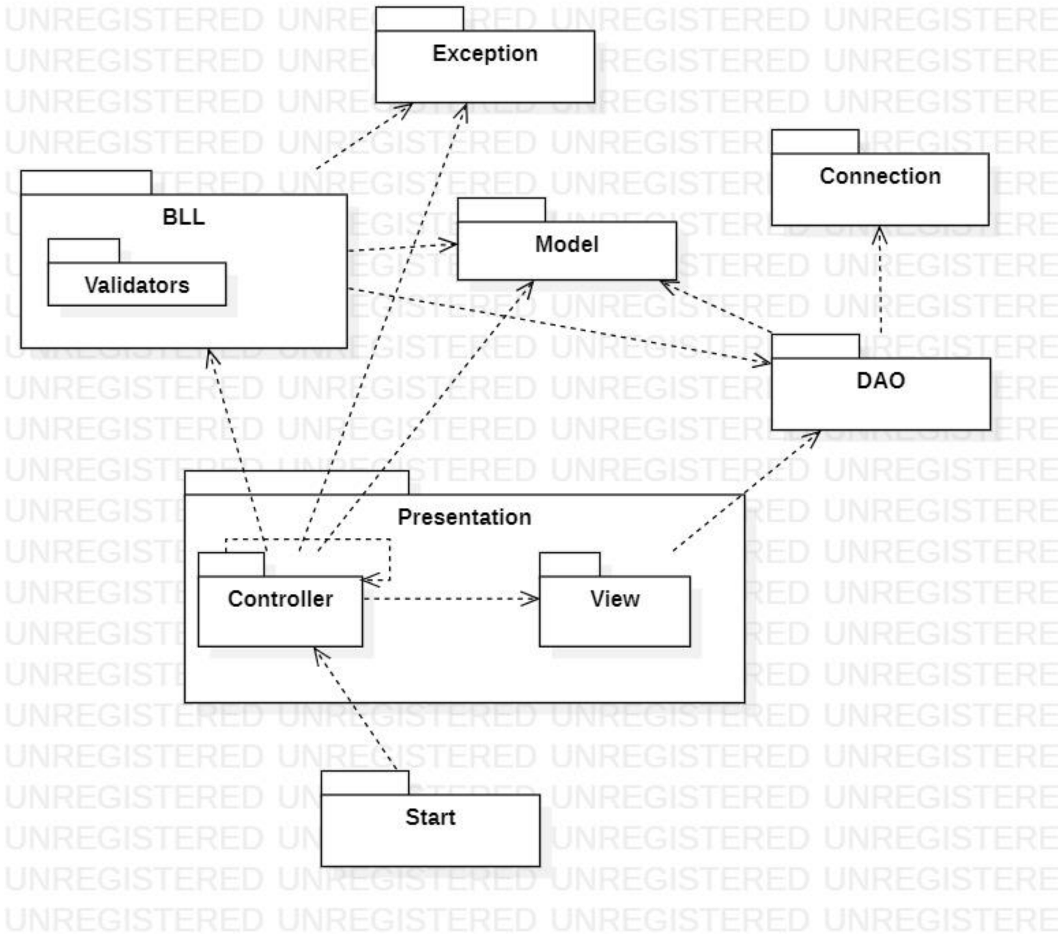

Presentation Layer
Model
Business Logic
Data Access
MySQL Database
Presentation Layer - contains the classes defining the user interface
Business Layer – contains the classes that encapsulate the application logic
Data Access Layer – contains the classes containing the queries and the database connection
Model – contains classes mapped to the database table

3.1 Diagrams

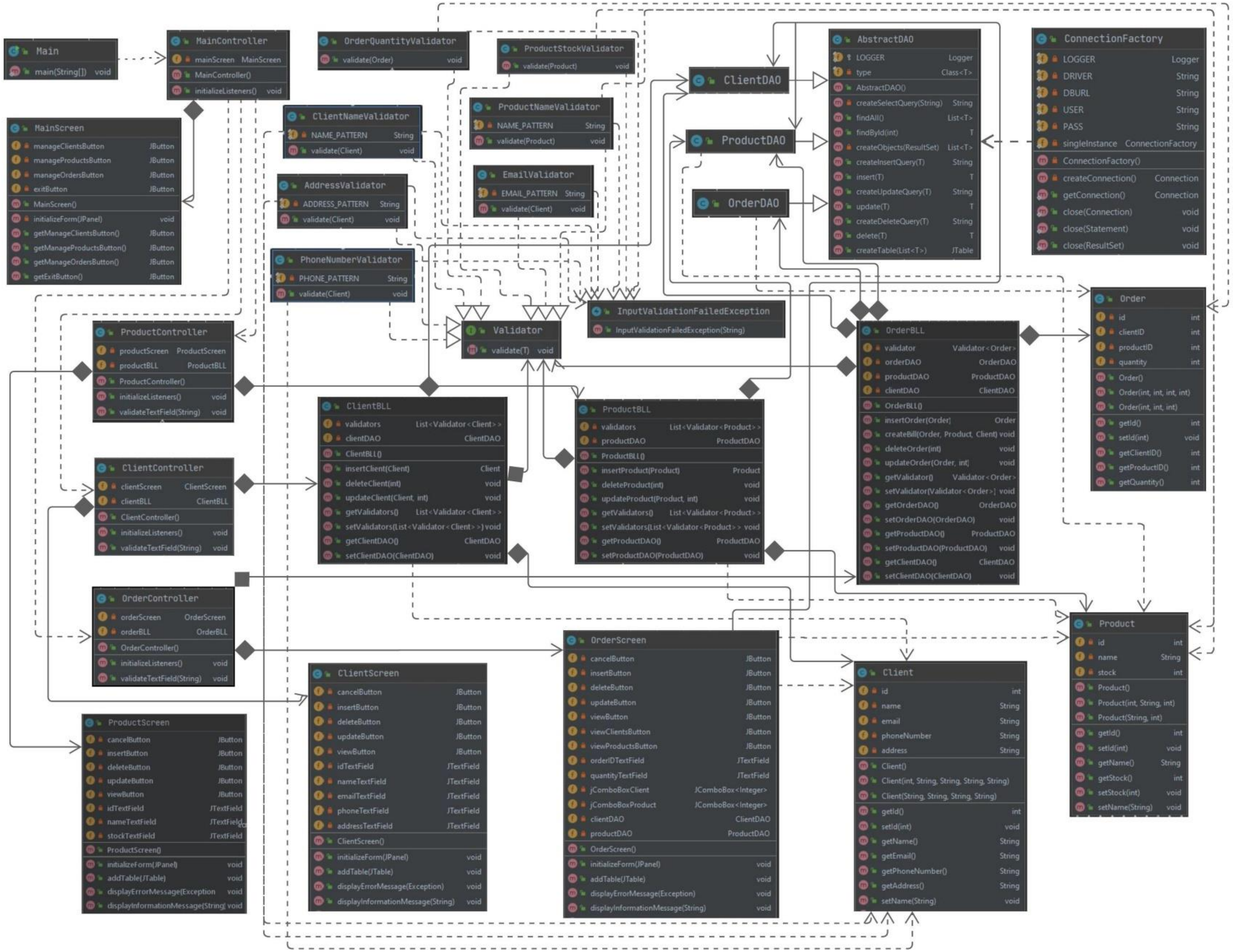
3.1.1 ERD Diagram



3.1.2 Package Diagram [2]



3.1.3 Class Diagram [3]


Main
main(String[]) void
MainController
mainScreen MainScreen
MainController()
initializeListeners() void
MainScreen
manageClientsButton JButton
manageProductsButton JButton
manageOrdersButton JButton
exitButton JButton
MainScreen()
initializeForm(JPanel) void
getManageClientsButton() JButton
getManageProductsButton() JButton
getManageOrdersButton() JButton
getExitButton() JButton
OrderQuantityValidator
validate(Order) void
ProductStockValidator
validate(Product) void
ClientNameValidator
NAME_PATTERN String
validate(Client) void
ProductNameValidator
NAME_PATTERN String
validate(Product) void
AddressValidator
ADDRESS_PATTERN String
validate(Client) void
EmailValidator
EMAIL_PATTERN String
validate(Client) void
PhoneNumberValidator
PHONE_PATTERN String
validate(Client) void
Validator
validate(T) void
InputValidationFailedException
InputValidationFailedException(String)
ClientDAO
ProductDAO
OrderDAO
AbstractDAO
LOGGER Logger
type Class<T>
AbstractDAO()
createSelectQuery(String) String
findAll() List<T>
findById(int) T
createObjects(ResultSet) List<T>
createInsertQuery(T) String
insert(T) T
createUpdateQuery(T) String
update(T) T
createDeleteQuery(T) String
delete(T) T
createTable(List<T>) JTable
ConnectionFactory
LOGGER Logger
DRIVER String
DBURL String
USER String
PASS String
singleInstance ConnectionFactory
ConnectionFactory()
createConnection() Connection
getConnection() Connection
close(Connection) void
close(Statement) void
close(ResultSet) void
ProductController
productScreen ProductScreen
productBLL ProductBLL
ProductController()
initializeListeners() void
validateTextField(String) void
ClientController
clientScreen ClientScreen
clientBLL ClientBLL
ClientController()
initializeListeners() void
validateTextField(String) void
OrderController
orderScreen OrderScreen
orderBLL OrderBLL
OrderController()
initializeListeners() void
validateTextField(String) void
ClientBLL
validators List<Validator<Client>>
clientDAO ClientDAO
ClientBLL()
insertClient(Client) Client
deleteClient(int) void
updateClient(Client, int) void
getValidators() List<Validator<Client>>
setValidators(List<Validator<Client>>) void
getClientDAO() ClientDAO
setClientDAO(ClientDAO) void
ProductBLL
validators List<Validator<Product>>
productDAO ProductDAO
ProductBLL()
insertProduct(Product) Product
deleteProduct(int) void
updateProduct(Product, int) void
getValidators() List<Validator<Product>>
setValidators(List<Validator<Product>>) void
getProductDAO() ProductDAO
setProductDAO(ProductDAO) void
OrderBLL
validator Validator<Order>
orderDAO OrderDAO
productDAO ProductDAO
clientDAO ClientDAO
OrderBLL()
insertOrder(Order) Order
createBill(Order, Product, Client) void
deleteOrder(int) void
updateOrder(Order, int) void
getValidator() Validator<Order>
setValidator(Validator<Order>) void
getOrderDAO() OrderDAO
setOrderDAO(OrderDAO) void
getProductDAO() ProductDAO
setProductDAO(ProductDAO) void
getClientDAO() ClientDAO
setClientDAO(ClientDAO) void
Order
id int
clientID int
productID int
quantity int
Order()
Order(int, int, int, int)
Order(int, int, int)
getId() int
setId(int) void
getClientID() int
getProductID() int
getQuantity() int
Product
id int
name String
stock int
Product()
Product(int, String, int)
Product(String, int)
getId() int
setId(int) void
getName() String
getStock() int
setStock(int) void
setName(String) void
Client
id int
name String
email String
phoneNumber String
address String
Client()
Client(int, String, String, String, String)
Client(String, String, String, String)
getId() int
setId(int) void
getName() String
getEmail() String
getPhoneNumber() String
getAddress() String
setName(String) void
OrderScreen
cancelButton JButton
insertButton JButton
deleteButton JButton
updateButton JButton
viewButton JButton
viewClientsButton JButton
viewProductsButton JButton
orderIDTextField JTextField
quantityTextField JTextField
jComboBoxClient JComboBox<Integer>
jComboBoxProduct JComboBox<Integer>
clientDAO ClientDAO
productDAO ProductDAO
OrderScreen()
initializeForm(JPanel) void
addTable(JTable) void
displayErrorMessage(Exception) void
displayInformationMessage(String) void
ClientScreen
cancelButton JButton
insertButton JButton
deleteButton JButton
updateButton JButton
viewButton JButton
idTextField JTextField
nameTextField JTextField
emailTextField JTextField
phoneTextField JTextField
addressTextField JTextField
ClientScreen()
initializeForm(JPanel) void
addTable(JTable) void
displayErrorMessage(Exception) void
displayInformationMessage(String) void
ProductScreen
cancelButton JButton
insertButton JButton
deleteButton JButton
updateButton JButton
viewButton JButton
idTextField JTextField
nameTextField JTextField
stockTextField JTextField
ProductScreen()
initializeForm(JPanel) void
addTable(JTable) void
displayErrorMessage(Exception void
displayInformationMessage(String) void

3.2 Packages and classes

-All the classes contain getters, setters and constructors, where needed.

- **Package start**
  - **Main** – instantiates the controller and thus it starts the application.

- **Package model**
  - **Client –** entity in the database: has id, name, email, phone number and address.
  - **Product -** entity in the database: has id, name and stock.
  - **Order** - entity in the database: has id, client id, product id and quantity.

- **Package presentation**
  - **Package view**
    - **MainScreen** - the principal GUI.
    - **ClientScreen –** the GUI for managing Clients.
    - **ProductScreen** - the GUI for managing Products.
    - **OrderScreen -** the GUI for managing Orders.

  - **Package controller**
    - **MainController** - - opens the principal frame for the application and initializes listeners for the buttons.
    - **ClientController–** - opens the frame for managing clients and initializes listeners for the buttons - validates the input from GUI to be non-empty, creates a new Client and passes it to the ClientBLL for validation before making operations in the database.
    - **ProductController** - opens the frame for managing product and initializes listeners for the buttons - validates the input from GUI to be non-empty, creates a new Product and passes it to the ProductBLL for validation before making operations in the database.
    - OrderController - opens the frame for managing orders and initializes listeners for the buttons - validates the input from GUI to be non-empty, creates a new Order and passes it to the OrderBLL for validation before making operations in the database.

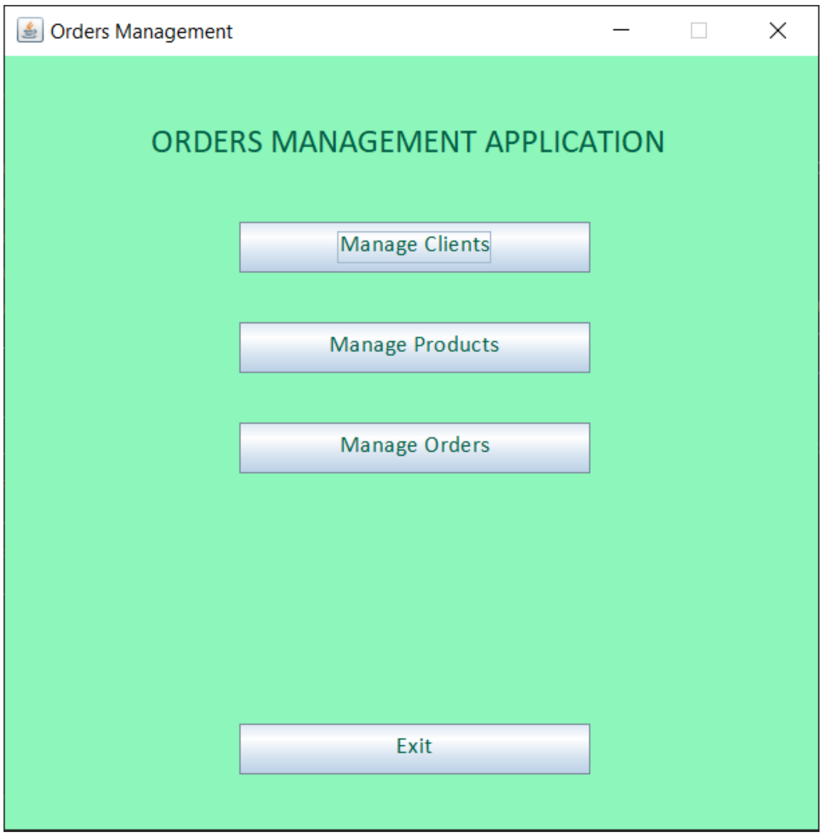
- **Package bll (business logic level)**
  - **Package validators** – contains interface *Validator* and classes which implement it for validation of clients, products and orders.
  - **ClientBLL –** validates the client to be added/updated into the database or for delete ensures that it exists.
  - **ProductBLL -** validates the order to be added/updated into the database or for delete it ensures that it exists.
  - **OrderBLL –** validates the product to be added/updated into the database or for delete ensures that it exists.

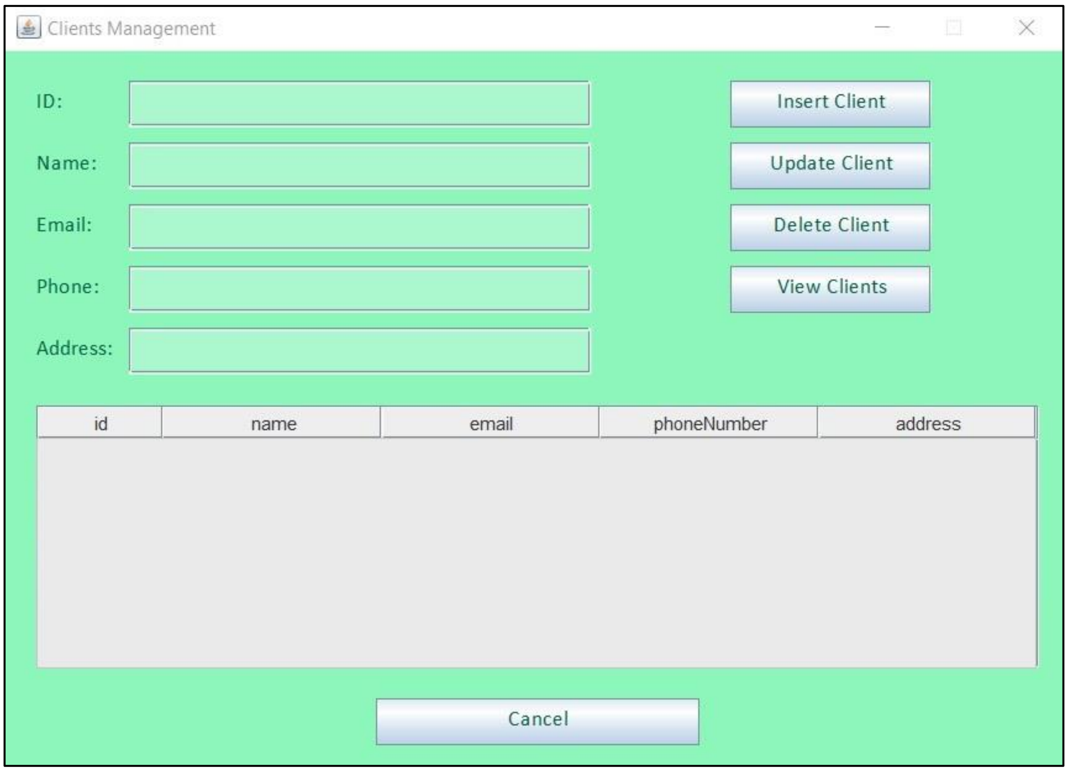
- **Package dao (data access)**
  - **AbstractDAO<T> -** defines the common operations for accessing a table: Insert, Update, Delete, FindById, FindAll; T can be Client, Product or Order.
  - **ClientDAO** - extends AbstractDAO<Client>
  - **ProductDAO** - extends AbstractDAO< Product >
  - **OrderDAO** - extends AbstractDAO< Order >

- **Package connection**
  - **ConnectionFactory** - contains the name of the driver (initialized through reflection), the database location (DBURL), and the user and the password for accessing the MySQL Server.

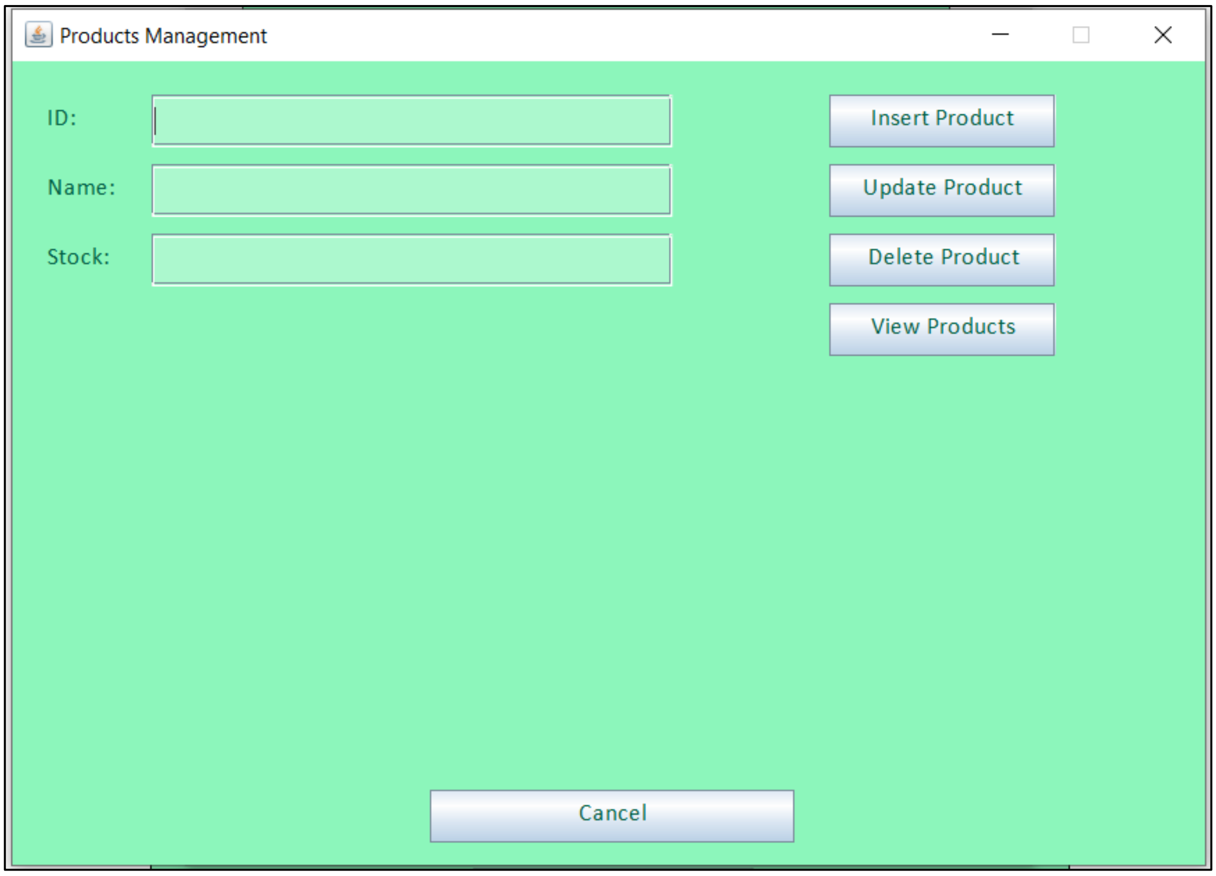
- **Package exception**
    - **InputValidationFailedException –** extends RuntimeException; is thrown if the input is not valid

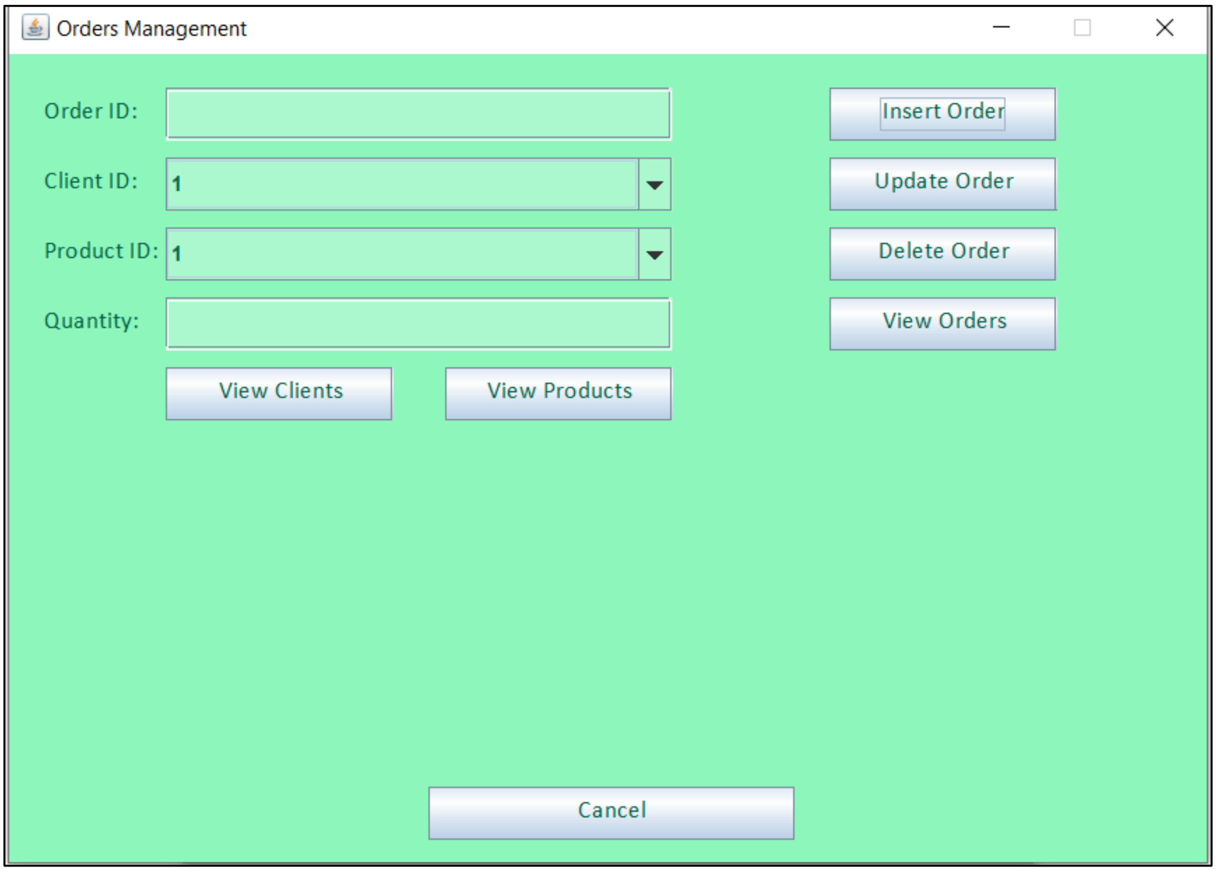
3.3 GUI Design





id	name	email	phoneNumber	address


Products Management
ID:
Name:
Stock:
Insert Product
Update Product
Delete Product
View Products
Cancel


Orders Management
Order ID:
Client ID:
1
Product ID:
1
Quantity:
View Clients
View Products
Insert Order
Update Order
Delete Order
View Orders
Cancel

3.4 Data Structures

- **ArrayList** – to create the header of the JTables dynamically through reflection;
  - to store the clients, orders and products resulted from the query for find all.

4. Implementation

- Data Access: *AbstractDAO<T>*

-example to create a query: insert query

```
public String createInsertQuery(T t) {
    StringBuilder sb = new StringBuilder();
    sb.append("INSERT INTO orders_db.");
    sb.append(type.getSimpleName());
    sb.append("(");
    String prefix = "";
    for (Field field : type.getDeclaredFields()) {
        String fieldName = field.getName();
        sb.append(prefix);
        prefix = ",";
        sb.append(fieldName);
    }
    sb.append(") VALUES(");
    prefix = "";
    for (Field field : type.getDeclaredFields()) {
        Object value;
        field.setAccessible(true);
        try {
            value = field.get(t);
            sb.append(prefix);
            prefix = ",";
            if(value instanceof String)
                sb.append("'");
            sb.append(value.toString());
            if(value instanceof String)
                sb.append("'");
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
    sb.append(");");
    return sb.toString();
}
```

-example to execute a query: insert

```
public T insert(T t) {
    String query = createInsertQuery(t);
    Connection connection = null;
    PreparedStatement statement = null;
    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.executeUpdate();

    } catch (SQLException e) {
        LOGGER.log(Level.WARNING, type.getName() + "DAO:insert " + e.getMessage());
    } finally {
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }
    return t;
}
```

-example to create objects from ResultSet returned by query

```
private List<T> createObjects(ResultSet resultSet) {
    List<T> list = new ArrayList<T>();
    try {
        while (resultSet.next()) {
            T instance = type.getDeclaredConstructor().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 (Exception e) {
        e.printStackTrace();
    }
    return list;
}
```

-example to create a JTable with headers the fields of a model object

```
public JTable createTable(List<T> list) {
    ArrayList<String> columns = new ArrayList<>();
    for(Field field : type.getDeclaredFields()) {
        field.setAccessible(true);
        columns.add(field.getName());
    }
    DefaultTableModel model = new DefaultTableModel();
    model.setColumnIdentifiers(columns.toArray());
    for(Object obj : list) {
        ArrayList<Object> objects = new ArrayList<>();
        for(Field field : type.getDeclaredFields()) {
            field.setAccessible(true);
            try {
                objects.add(field.get(obj));
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
        }
        model.addRow(objects.toArray());
    }
    JTable table = new JTable(model);
    table.getColumnModel().getColumn(0).setPreferredWidth(15);
    return table;
}
```

- Business Logic: *OrderBLL*

-example to insert an order by validating it

```
public Order insertOrder(Order order) {
    validator.validate(order);
    Product product = productDAO.findById(order.getProductID());
    Client client = clientDAO.findById(order.getClientID());
    if (product.getStock() < order.getQuantity()) {
        throw new InputValidationFailedException("Stock for product "+product.getName()+" is not sufficient.");
    }
    product.setStock(product.getStock() - order.getQuantity());
    productDAO.update(product);
    ArrayList<Order> orders = (ArrayList<Order>) orderDAO.findAll();
    Order lastOrder = orders.get(orders.size()-1);
    order.setId(lastOrder.getId()+1);
    createBill(order, product, client);
    return orderDAO.insert(order);
}
```

-example to delete an order

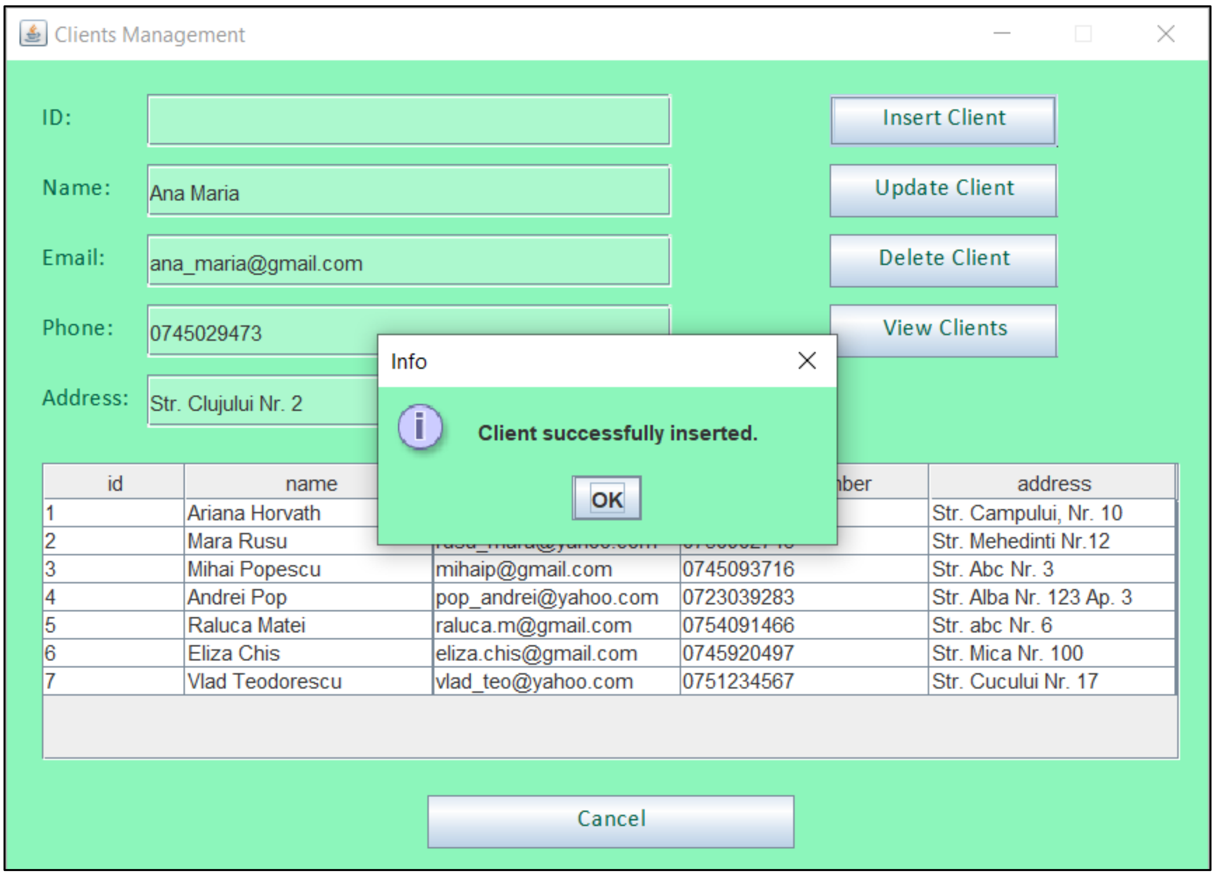
```
public void deleteOrder(int id) {
    Order order = orderDAO.findById(id);
    if (order == null)
        throw new InputValidationFailedException("Order with id "+id+" not existent.");
    Product product = productDAO.findById(order.getProductID());
    product.setStock(product.getStock() + order.getQuantity());
    productDAO.update(product);
    orderDAO.delete(order);
}
```

-example to create a bill for each order inserted

```
public void createBill(Order order, Product product, Client client) {
    FileWriter writer;
    try {
        writer = new FileWriter("bill" + order.getId()+".txt");
        writer.append("Bill \nOrder ID: " + order.getId() + "\nProduct: " + product.getName()+", quantity: "+
order.getQuantity()
                + "\nClient: " + client.getName()+ ", email: "+client.getEmail()+", phone number:
"+client.getPhoneNumber()+
                "\nDelivery address: " + client.getAddress());
        writer.close();
    } catch(IOException e) {
        e.printStackTrace();
    }
}
```

5. Results

- Insert Client



Clients Management

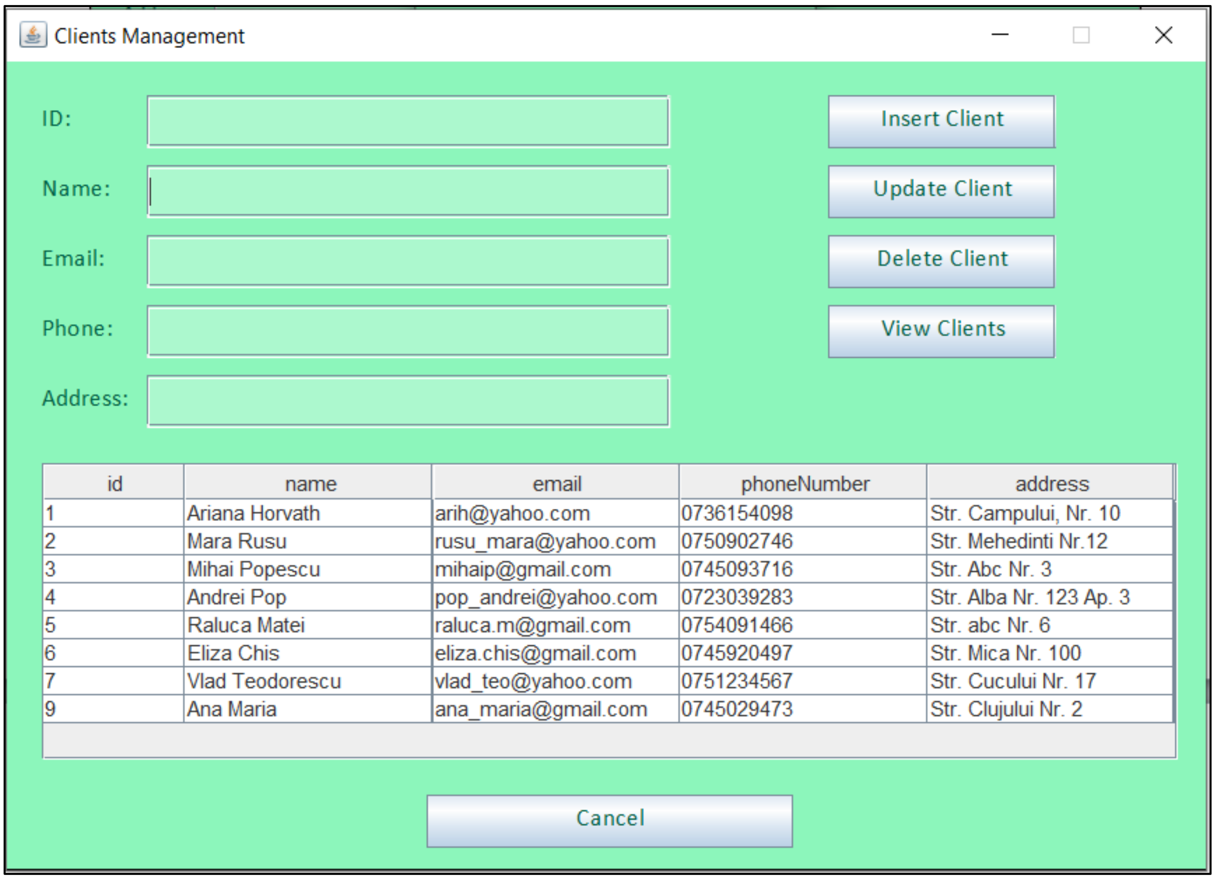
ID:
Name: Ana Maria
Email: ana_maria@gmail.com
Phone: 0745029473
Address: Str. Clujului Nr. 2

Info

Client successfully inserted.

id	name		ber	address
1	Ariana Horvath			Str. Campului, Nr. 10
2	Mara Rusu	[illegible]	[illegible]	Str. Mehedinti Nr.12
3	Mihai Popescu	mihaip@gmail.com	0745093716	Str. Abc Nr. 3
4	Andrei Pop	pop_andrei@yahoo.com	0723039283	Str. Alba Nr. 123 Ap. 3
5	Raluca Matei	raluca.m@gmail.com	0754091466	Str. abc Nr. 6
6	Eliza Chis	eliza.chis@gmail.com	0745920497	Str. Mica Nr. 100
7	Vlad Teodorescu	vlad_teo@yahoo.com	0751234567	Str. Cucului Nr. 17

- View Clients



ID:

Name:

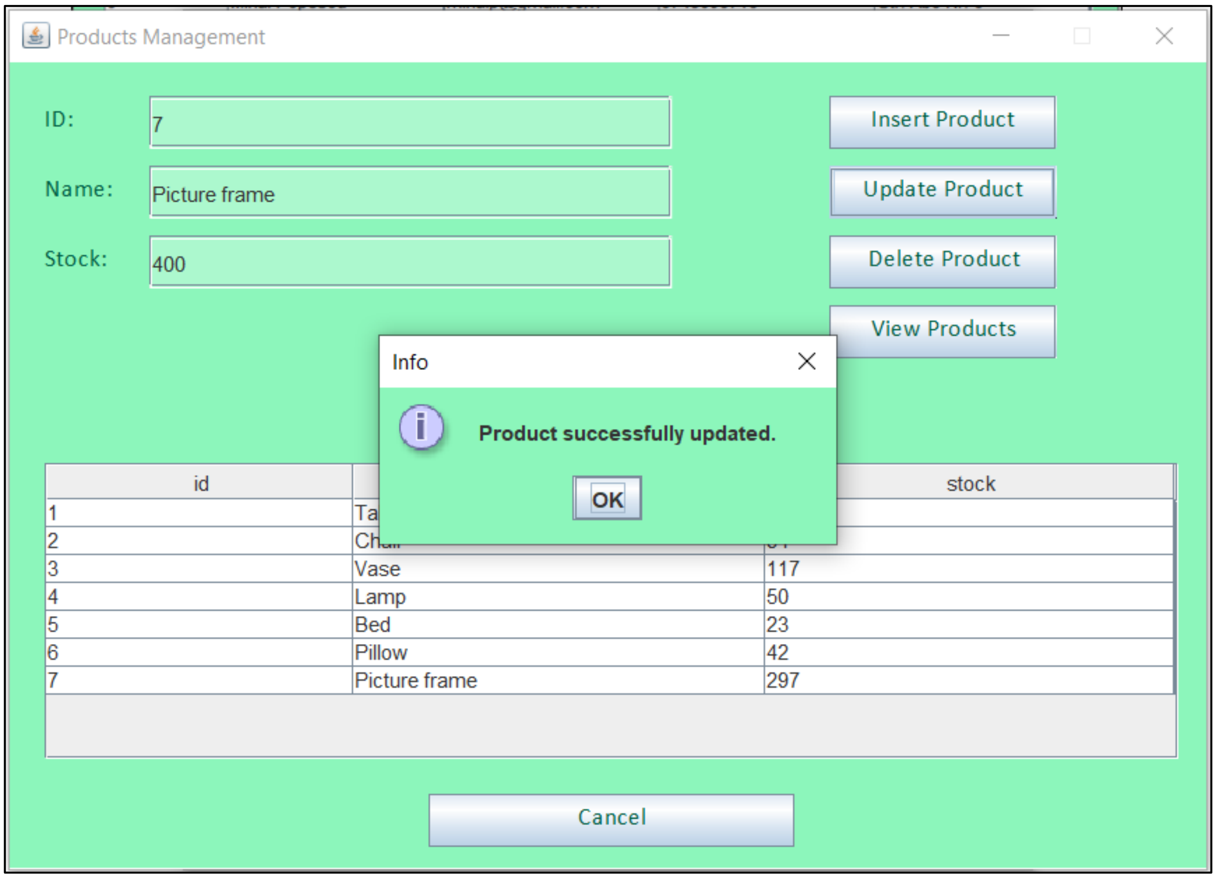
Email:

Phone:

Address:

id	name	email	phoneNumber	address
1	Ariana Horvath	arih@yahoo.com	0736154098	Str. Campului, Nr. 10
2	Mara Rusu	rusu_mara@yahoo.com	0750902746	Str. Mehedinti Nr.12
3	Mihai Popescu	mihaip@gmail.com	0745093716	Str. Abc Nr. 3
4	Andrei Pop	pop_andrei@yahoo.com	0723039283	Str. Alba Nr. 123 Ap. 3
5	Raluca Matei	raluca.m@gmail.com	0754091466	Str. abc Nr. 6
6	Eliza Chis	eliza.chis@gmail.com	0745920497	Str. Mica Nr. 100
7	Vlad Teodorescu	vlad_teo@yahoo.com	0751234567	Str. Cucului Nr. 17
9	Ana Maria	ana_maria@gmail.com	0745029473	Str. Clujului Nr. 2

- Update Products



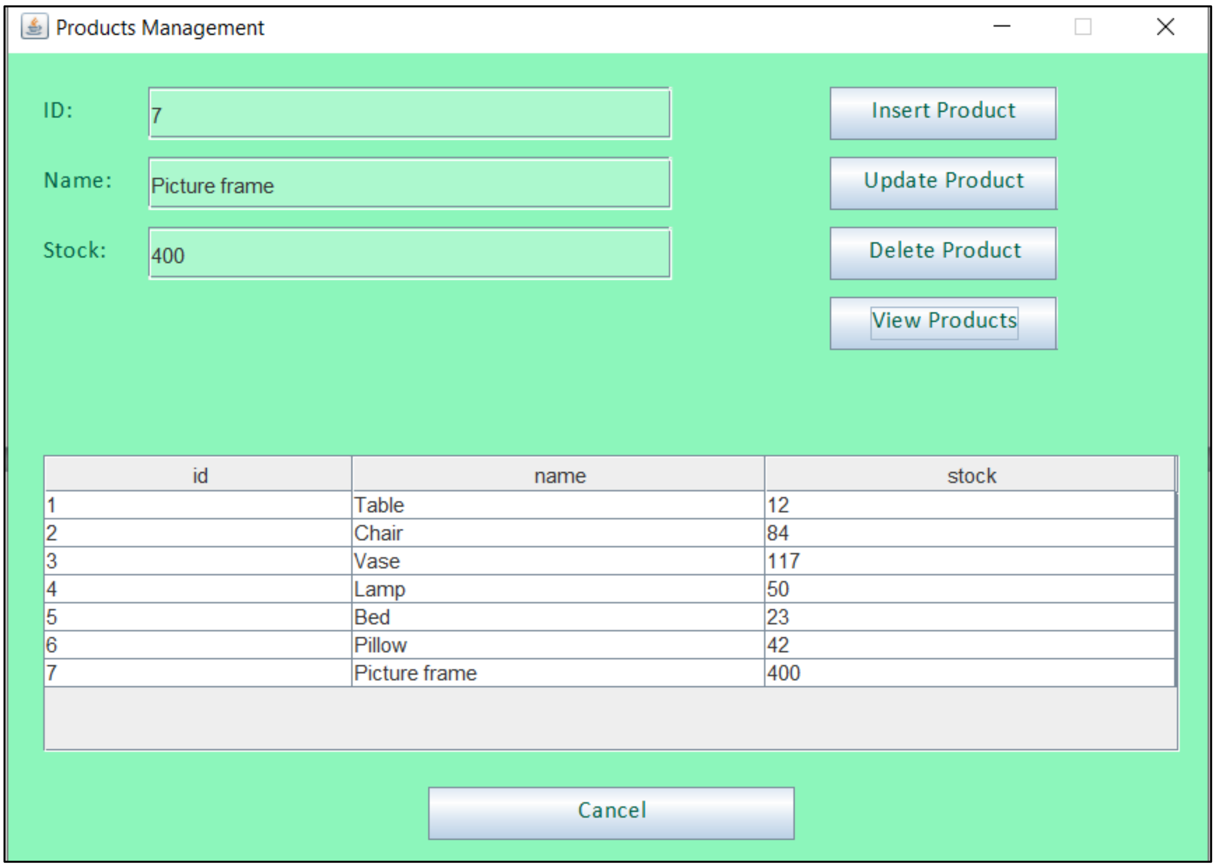
ID: 7

Name: Picture frame

Stock: 400

Product successfully updated.

id		stock
1	Ta	
2	Ch	
3	Vase	117
4	Lamp	50
5	Bed	23
6	Pillow	42
7	Picture frame	297


Products Management

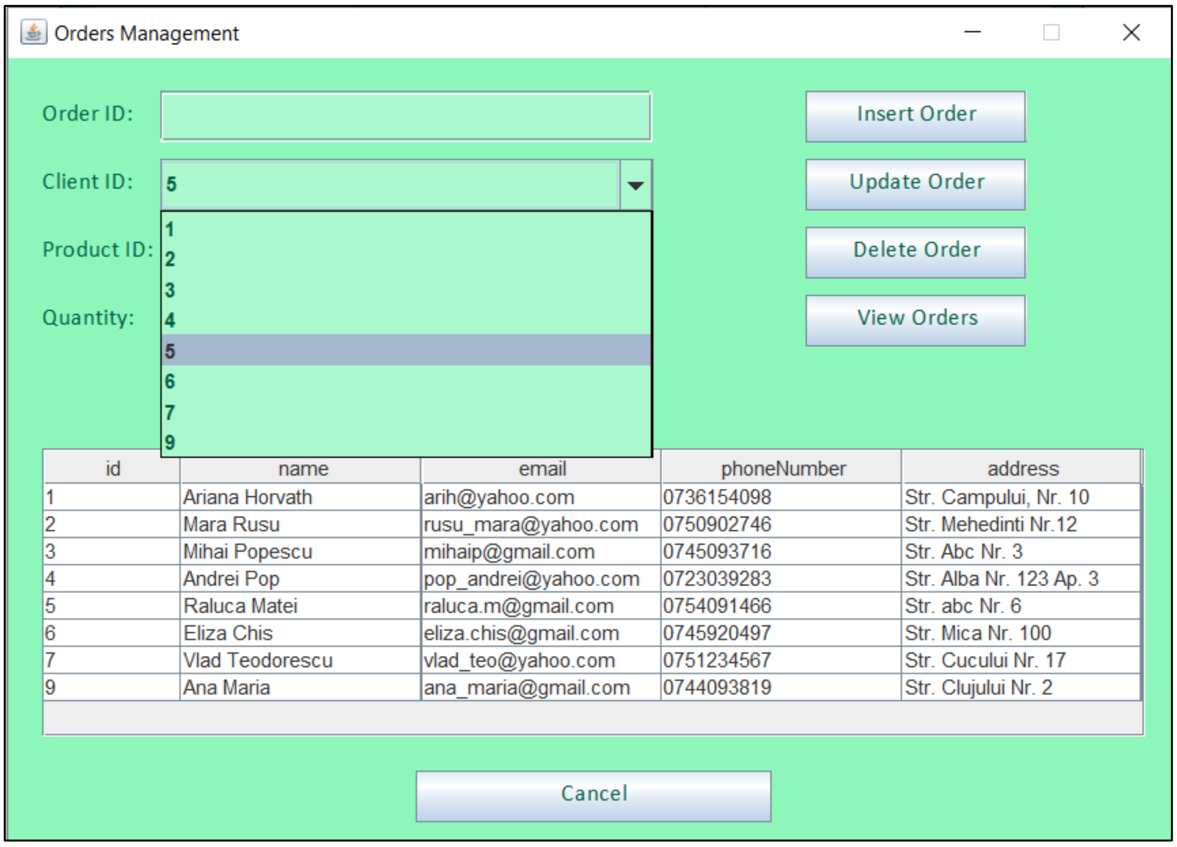
ID: 7

Name: Picture frame

Stock: 400

id	name	stock
1	Table	12
2	Chair	84
3	Vase	117
4	Lamp	50
5	Bed	23
6	Pillow	42
7	Picture frame	400

- Insert Order
  -select client from existent ones, which can be seen in the table (press "View Clients")


Orders Management

Order ID:

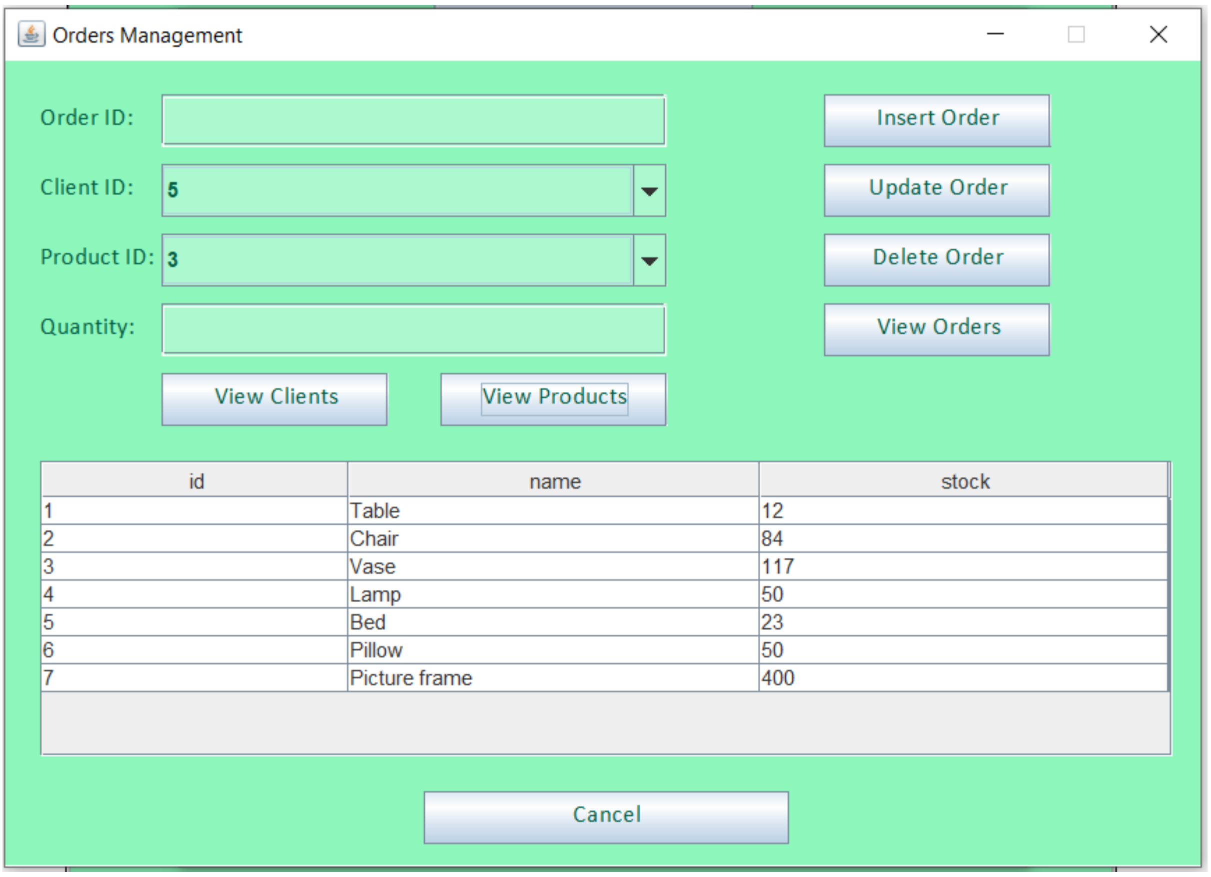
Client ID: 5

Product ID:

Quantity:

id	name	email	phoneNumber	address
1	Ariana Horvath	arih@yahoo.com	0736154098	Str. Campului, Nr. 10
2	Mara Rusu	rusu_mara@yahoo.com	0750902746	Str. Mehedinti Nr.12
3	Mihai Popescu	mihaip@gmail.com	0745093716	Str. Abc Nr. 3
4	Andrei Pop	pop_andrei@yahoo.com	0723039283	Str. Alba Nr. 123 Ap. 3
5	Raluca Matei	raluca.m@gmail.com	0754091466	Str. abc Nr. 6
6	Eliza Chis	eliza.chis@gmail.com	0745920497	Str. Mica Nr. 100
7	Vlad Teodorescu	vlad_teo@yahoo.com	0751234567	Str. Cucului Nr. 17
9	Ana Maria	ana_maria@gmail.com	0744093819	Str. Clujului Nr. 2

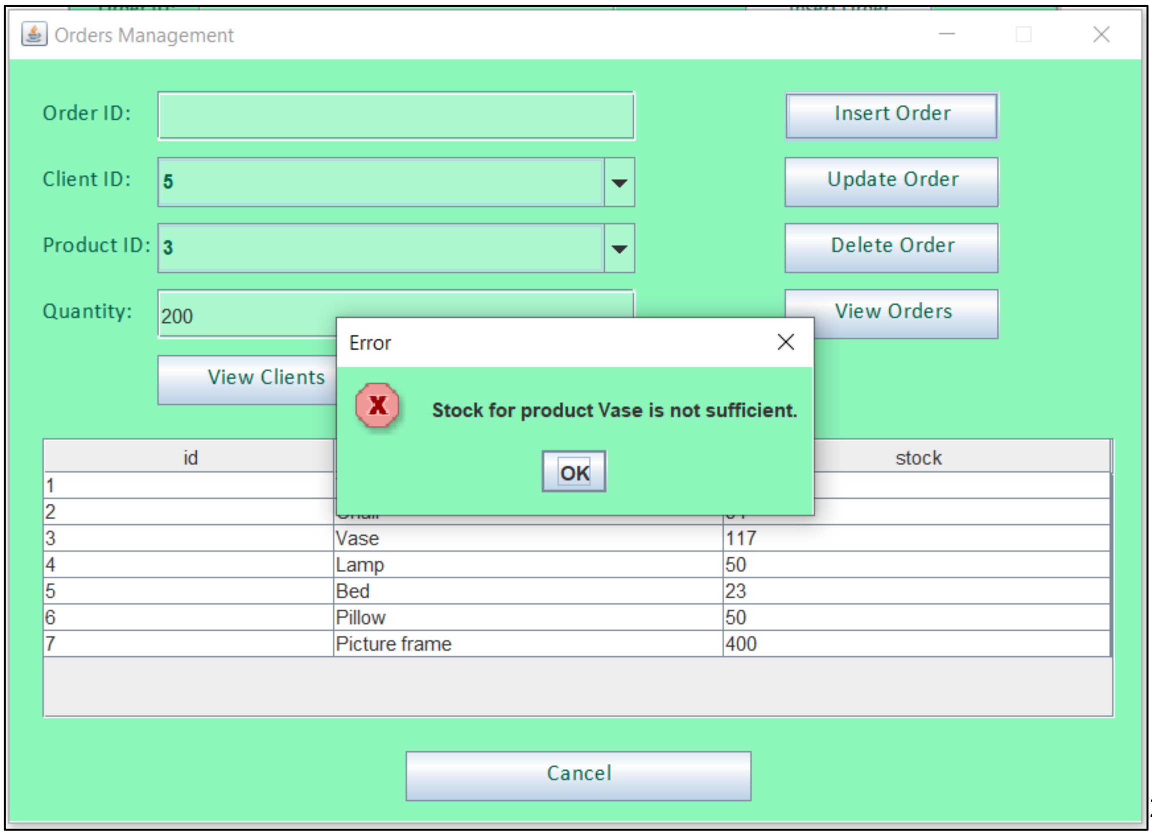
- select product from existent ones, which can be seen in the table (press “View Products”)



Client ID: 5
Product ID: 3

id	name	stock
1	Table	12
2	Chair	84
3	Vase	117
4	Lamp	50
5	Bed	23
6	Pillow	50
7	Picture frame	400

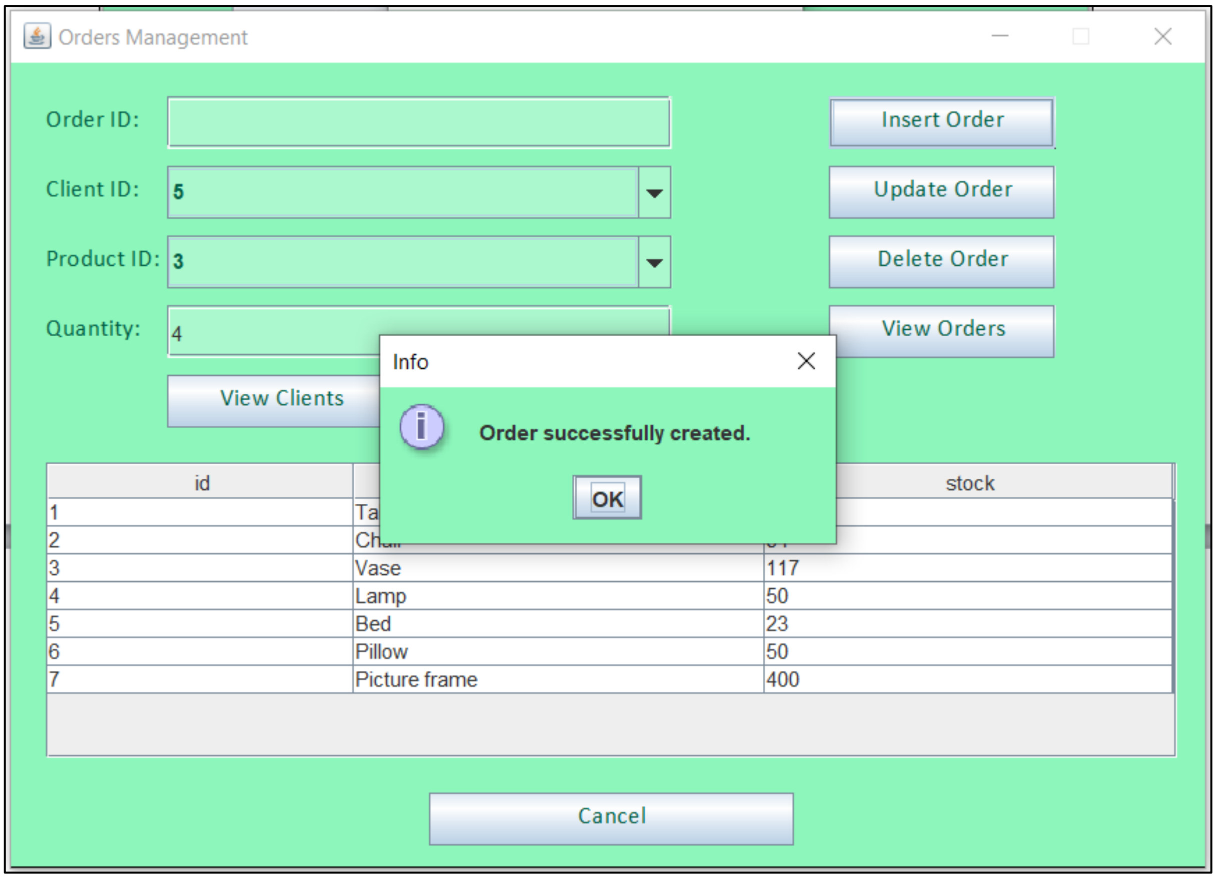
-insert quantity
-stock is not sufficient, message is displayed, insert an available stock



Client ID: 5
Product ID: 3
Quantity: 200

Stock for product Vase is not sufficient.

id		stock
1		
2		
3	Vase	117
4	Lamp	50
5	Bed	23
6	Pillow	50
7	Picture frame	400


Orders Management

Order ID:

Client ID: 5

Product ID: 3

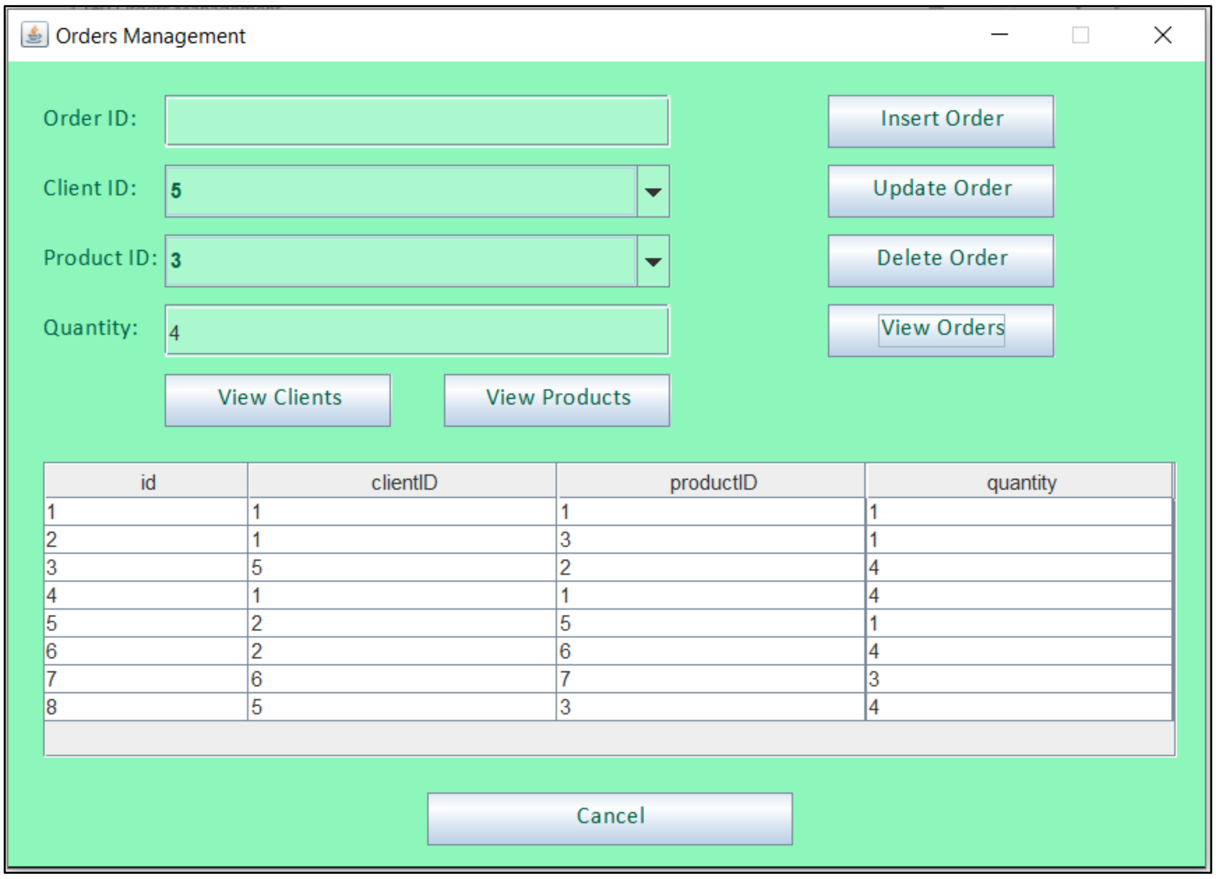
Quantity: 4

Info

Order successfully created.

id			stock
1	Ta		
2	Ch		
3	Vase		117
4	Lamp		50
5	Bed		23
6	Pillow		50
7	Picture frame		400

-view orders


Orders Management

Order ID:

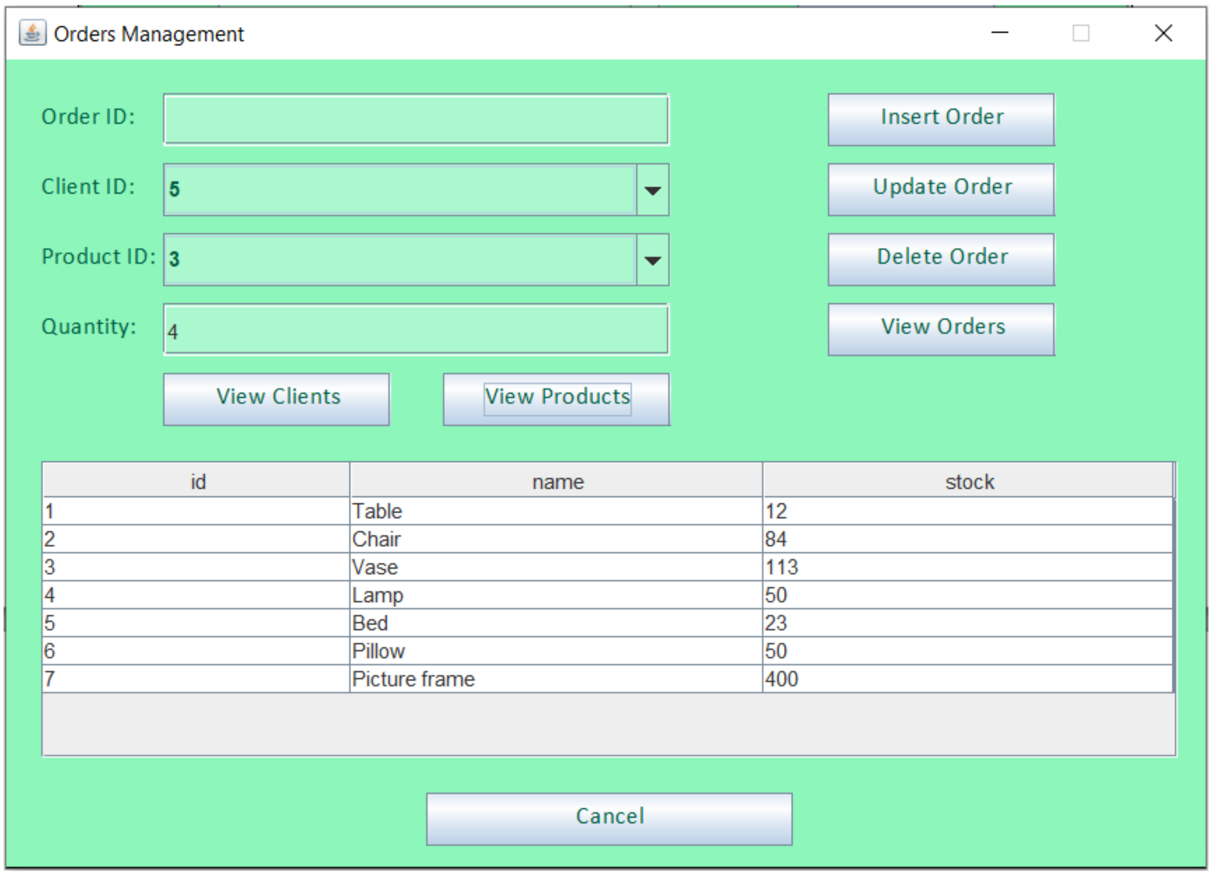
Client ID: 5

Product ID: 3

Quantity: 4

id	clientID	productID	quantity
1	1	1	1
2	1	3	1
3	5	2	4
4	1	1	4
5	2	5	1
6	2	6	4
7	6	7	3
8	5	3	4

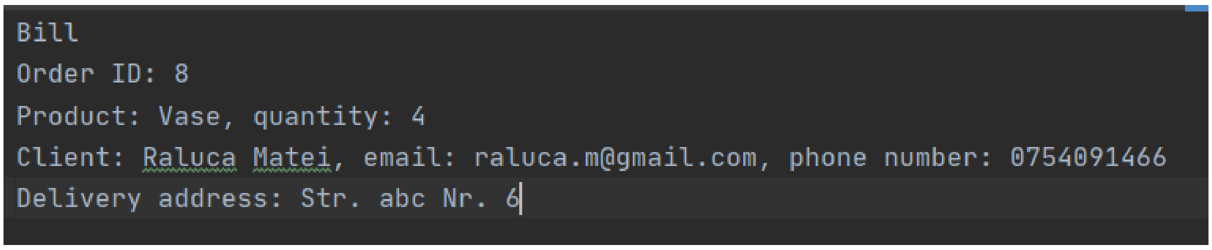
- view products: notice that the stock is decremented



Client ID: 5
Product ID: 3
Quantity: 4

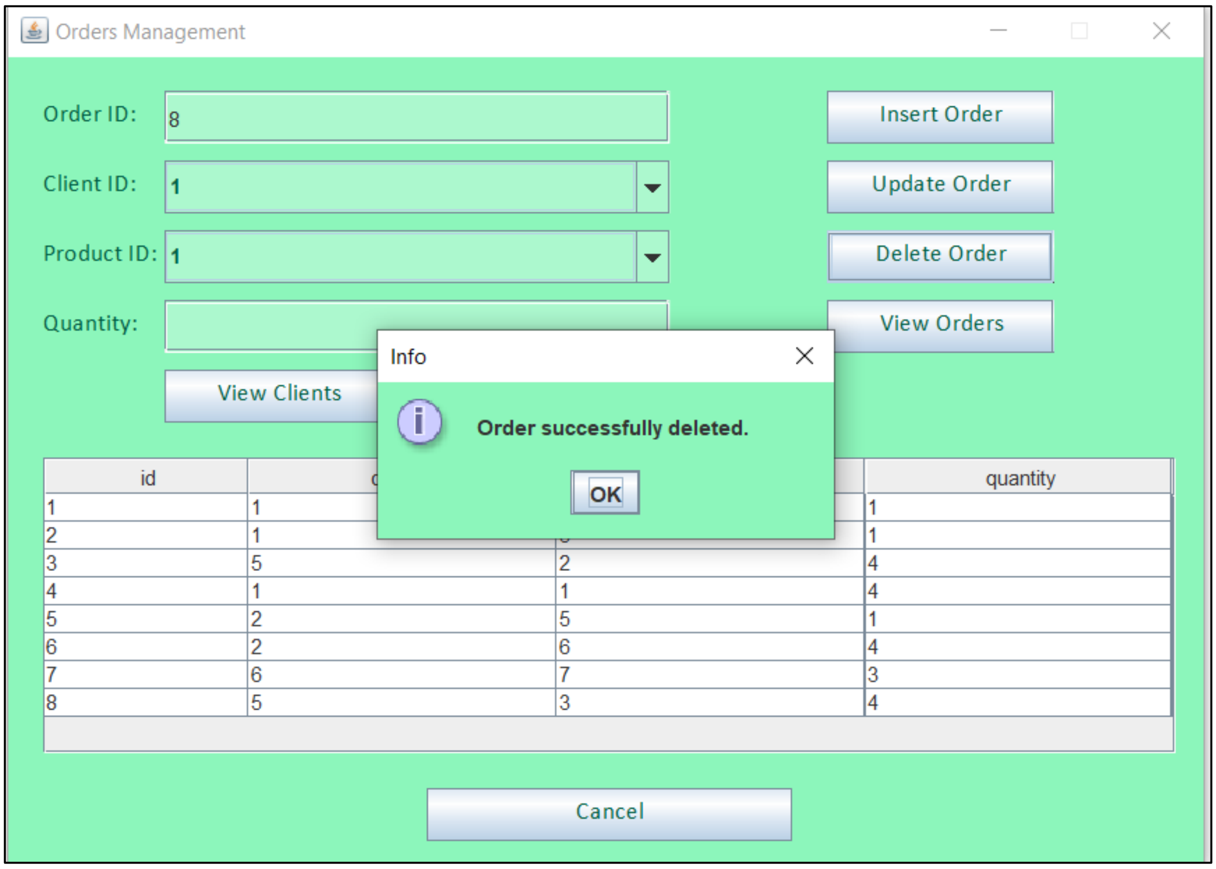
id	name	stock
1	Table	12
2	Chair	84
3	Vase	113
4	Lamp	50
5	Bed	23
6	Pillow	50
7	Picture frame	400

-Bill8.txt



```
Bill
Order ID: 8
Product: Vase, quantity: 4
Client: Raluca Matei, email: raluca.m@gmail.com, phone number: 0754091466
Delivery address: Str. abc Nr. 6
```

- Delete Order



Order ID: 8

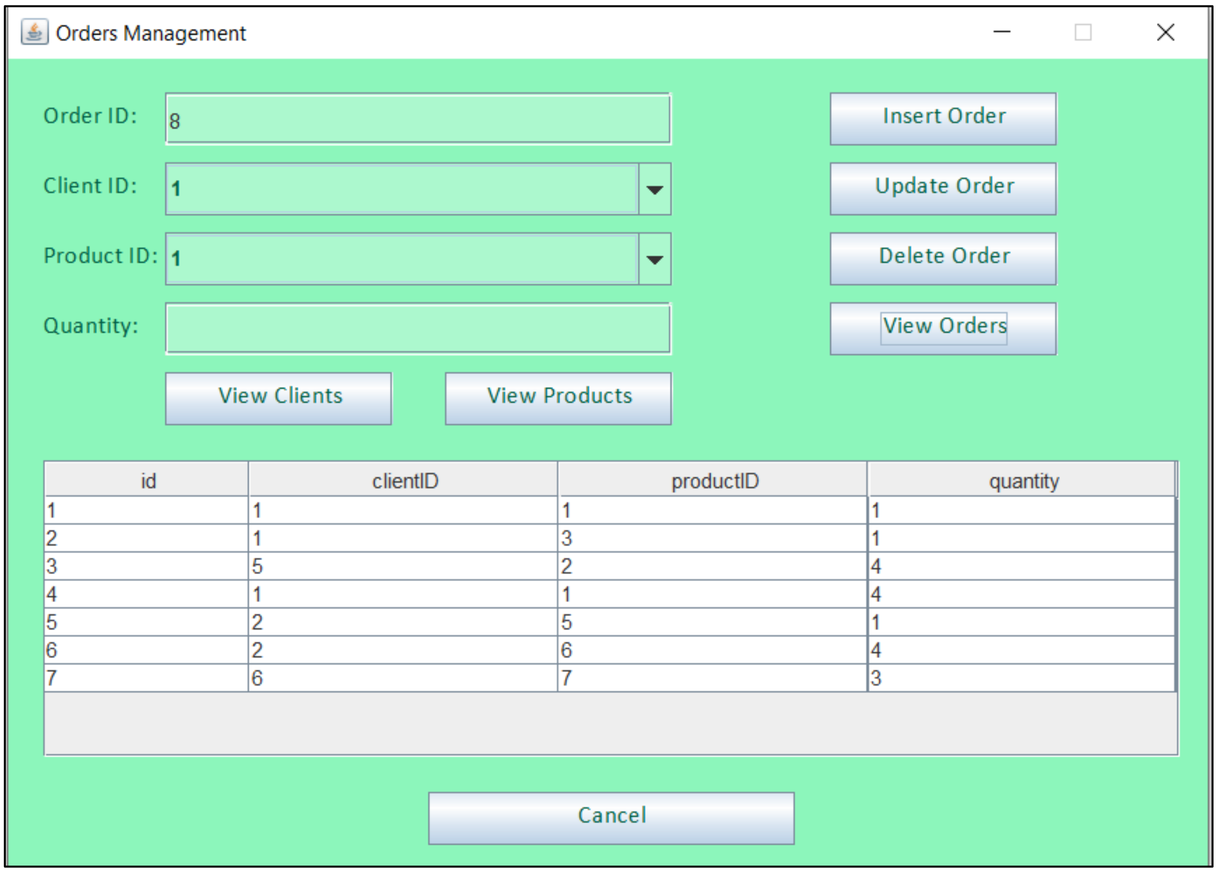
Client ID: 1

Product ID: 1

Quantity:

Order successfully deleted.

id	c		quantity
1	1		1
2	1		1
3	5	2	4
4	1	1	4
5	2	5	1
6	2	6	4
7	6	7	3
8	5	3	4



Order ID: 8

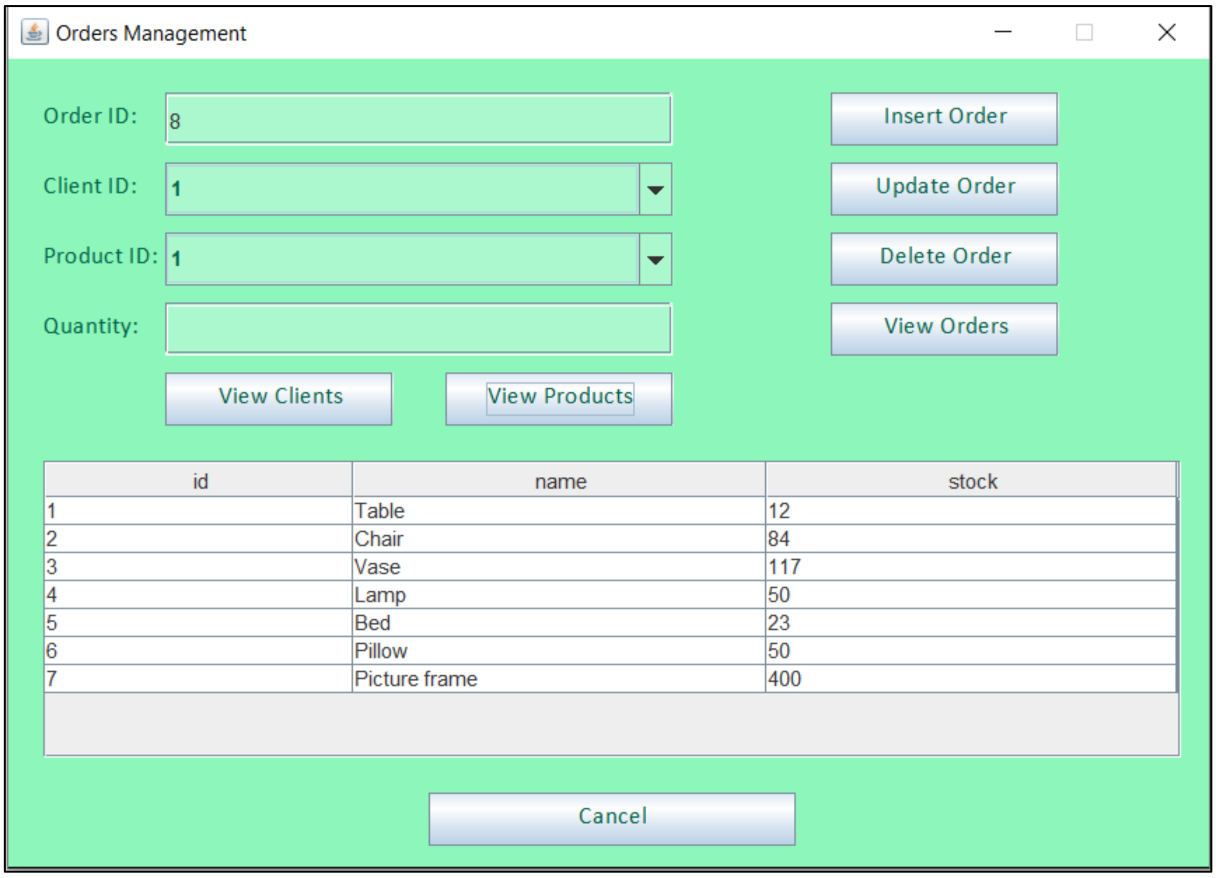
Client ID: 1

Product ID: 1

Quantity:

id	clientID	productID	quantity
1	1	1	1
2	1	3	1
3	5	2	4
4	1	1	4
5	2	5	1
6	2	6	4
7	6	7	3

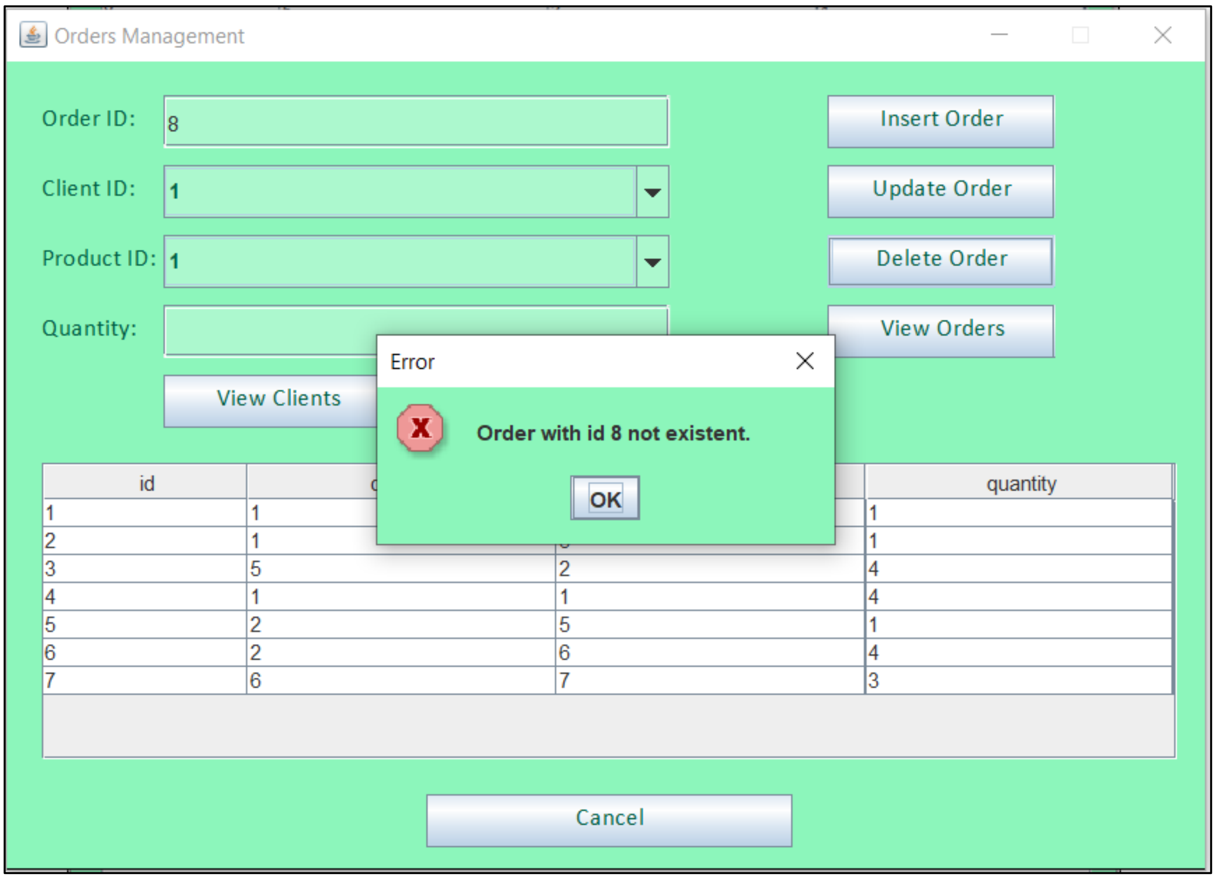
- notice that the stock for the product Vase is incremented


Orders Management

Order ID:	8
Client ID:	1
Product ID:	1
Quantity:	

id	name	stock
1	Table	12
2	Chair	84
3	Vase	117
4	Lamp	50
5	Bed	23
6	Pillow	50
7	Picture frame	400

- Try to delete inexistent client


Orders Management

Order ID:	8
Client ID:	1
Product ID:	1
Quantity:	

Error

Order with id 8 not existent.

id	c		quantity
1	1		1
2	1		1
3	5	2	4
4	1	1	4
5	2	5	1
6	2	6	4
7	6	7	3

6. Conclusions

I consider that the user interface of the Orders Management Application is intuitive and easy to use and that the application does everything that is required and it can be useful for managing orders.

The implementation had a lot of notions to consider, mostly relating to reflection techniques and access to the MySQL database.

What I learned from this assignment was a deepening of the OOP concepts learned the last semester and working with GUIs, Maven etc. but also learning new things such as implementing reflection, managing an SQL database from the Java application and working with JTables in GUI.

Future improvements of the application could be: a bigger GUI so that the longer data from the tables can be seen better and the possibility to order different kinds of products in the same order.

7. Bibliography

[1] Fundamental Programming Techniques – Lecture Slides, Assignment_3_Support_Presentation

[2] https://staruml.io/

[3] https://www.lucidchart.com/pages/

[4] https://app.creately.com/diagram/

[5] https://stackoverflow.com/questions/953972/java-jtable-setting-column-width

[6] https://regex101.com/

[7] https://alvinalexander.com/java/edu/pj/jdbc/jdbc0002/