Contents

1. Assignment Objective

Main objective

The main objective of this assignment is to design and implement an order management system with a dedicated graphical user interface through which the user can perform operations on the clients, products and orders.

Sub-objectives

- Analyze the problem and identify requirements.
- Design the client management system.
- Design the product management system.
- Design the order management system.
- Implement the client, product, order management systems.

2. Problem analysis, modeling, scenarios, use cases

Problem analysis

The purpose of the project is to solve the following problem: managing the clients, products and orders in a specific commercial institution, such as a bookstore since managing this kind of operations and organizations on paper would be difficult and time consuming.

Solution: As a solution, the project provides a faster way of inserting information into a virtual system, making it easier for the user to keep track of all the actions performed regarding the clients and orders.

Functional requirements:

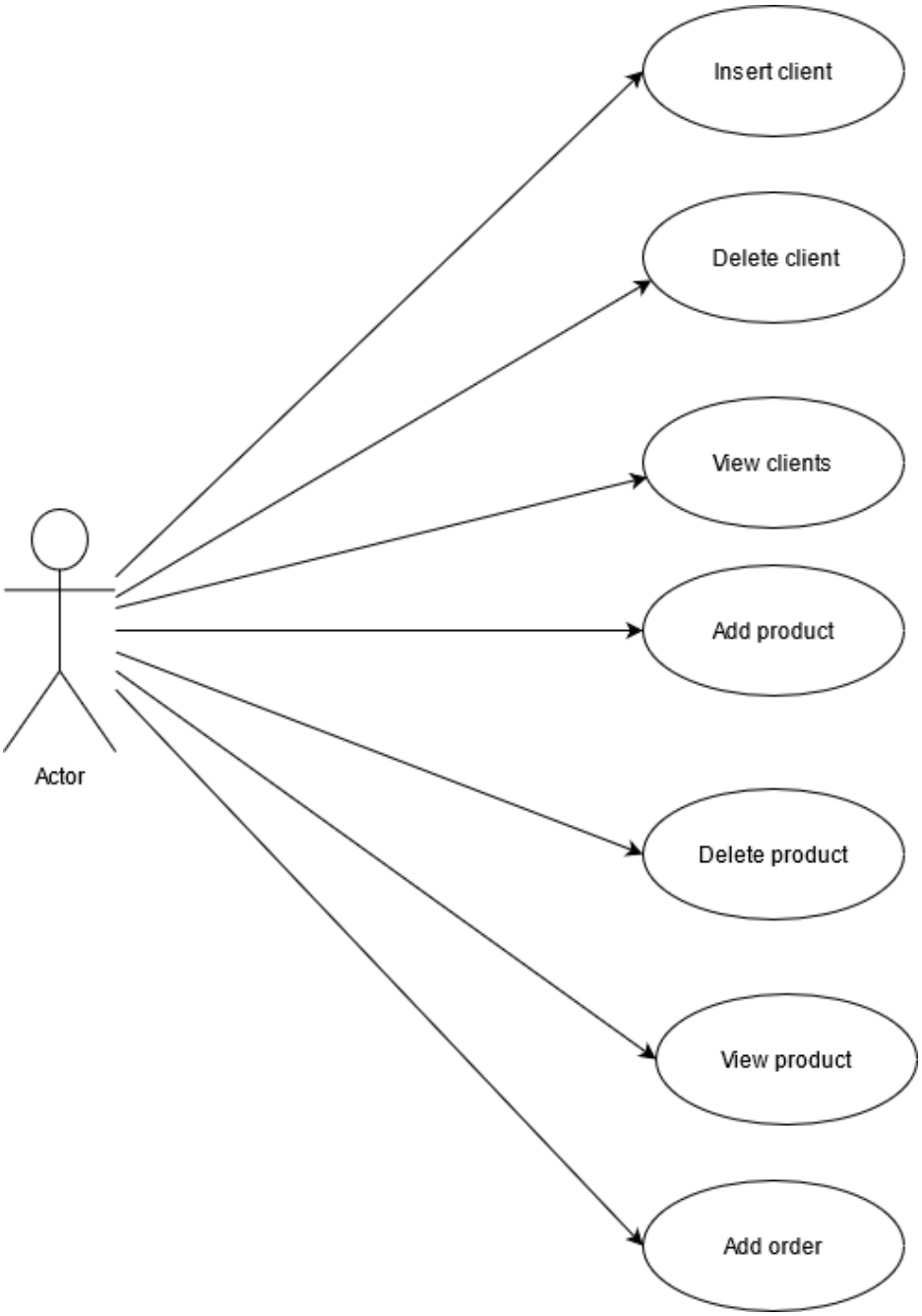
- The order management system should allow inserting clients in a database.
- The order management system should allow deleting clients from the database.
- The order management system should allow viewing all clients from the database.
- The order management system should allow inserting products in a database.
- The order management system should allow deleting products from a database.
- The order management system should allow viewing all products from a database.
- The order management system should allow placing an order.
- The order management system should print a receipt for every order performed.

Non-functional requirements:

- The order management system should be intuitive and easy to use by the user.
- The order management system should have a pleasant interface.

Use Cases

Use case diagram



Use case : add client;

Primary actor: user;

Main success scenario:

1. The user selects the client operation field.
2. The user selects add client operation.
3. The user completes the necessary fields for inserting a client.
4. The user finishes the action and presses the button for insertion.
5. The application displays a message for successfully inserting a client.

Alternative sequence:

- The user inserts invalid input (e.g. Invalid phone number)
- The scenario returns to first step 1.
- The connection to the database fails.

Use case : delete client;

Primary actor: user;

Main success scenario:

1. The users selects the client operations field.
2. The user chooses the delete client operation.
3. The user chooses what entry to be deleted.
4. The user presses the delete button and deletes the entry.
5. The application displays a message for successfully deleting a client.

Alternative sequence:

- The scenario returns to first step 1.

Use case : edit client;

Primary actor: user;

Main success scenario:

1. The users selects the client operations field.
2. The user chooses the edit client operation.
3. The user chooses what entry to be edited.
4. The user inserts the fields to be edited.
5. The application displays a message for successfully deleting a client.

Alternative sequence:

- The user inserts an invalid input.
- The scenario returns to the first step.

Use case : view clients

Primary actor: user;

Main success scenario:

1. The users selects the client operations field.
2. The user chooses the view clients operation.
3. The user chooses what entry to be deleted.
4. The application shows the table with all the entries from the field.

Alternative sequence:

- The scenario returns to the first step.

Use case : add product;

Primary actor: user;

Main success scenario:

1. The user selects the product operation field.
2. The user selects add product operation.
3. The user completes the necessary fields for inserting a product.
4. The user finishes the action and presses the button for insertion.
5. The application displays a message for successfully inserting a product.

Alternative sequence:

- The user inserts invalid input (e.g. Invalid phone number)
- The scenario returns to first step 1

Use case : delete product;

Primary actor: user;

Main success scenario:

1. The users selects the product operations field.
2. The user chooses the delete product operation.
3. The user chooses what entry to be deleted.
4. The user presses the delete button and deletes the entry.
5. The application displays a message for successfully deleting a product.

Alternative sequence:

- The scenario returns to the first step.

Use case : edit product;

Primary actor: user;

Main success scenario:

1. The users selects the product operations field.
2. The user chooses the edit product operation.
3. The user chooses what entry to be edited.
4. The user inserts the fields to be edited.
5. The application displays a message for successfully deleting a product.

Alternative sequence:

- The user inserts an invalid input.

Use case : view products

Primary actor: user;

Main success scenario:

1. The users selects the product operations field.
2. The user chooses the view product operation.
3. The user chooses what entry to be deleted.
4. The application shows the table with all the entries from the field.

Alternative sequence:

- The scenario returns to the first step.

Use case: Placing an order

Primary actor: user;

Main success scenario:

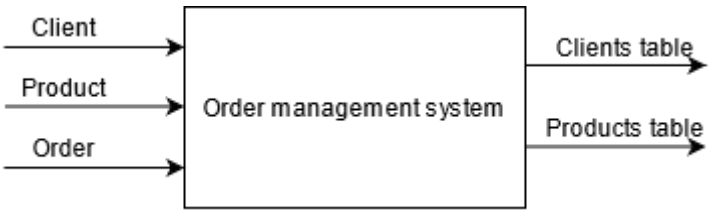
1. The user selects the order operations field
2. The user chooses one client.
3. The user chooses one product.
4. The user places the order.
5. The app generates a bill for the specific order.

Alternative sequence:

- The scenario returns to the first step.

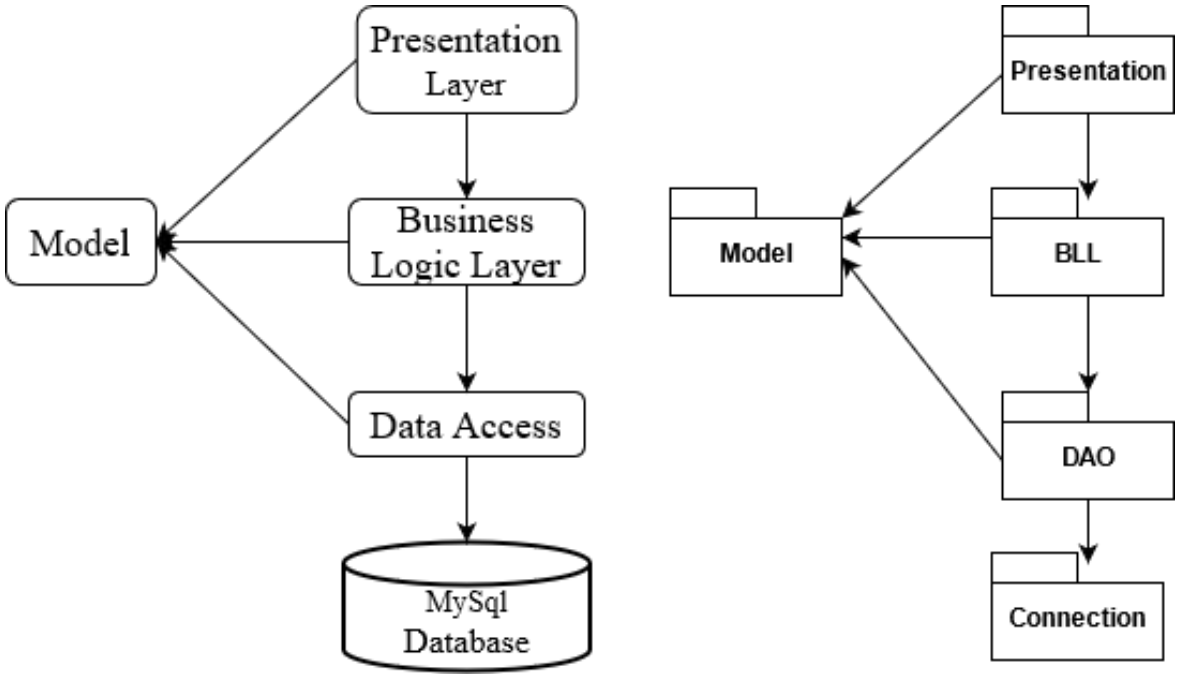
3. Design

Level 1 : Overall system design



Level 2: Division into packages

The project follows a layered architecture. It is divided in six subsystems/packages. Layered architecture abstracts the view of the system as whole while providing enough detail to understand the roles of individual layers and the relationships in between them. In a layered architecture, each modular layer depends on the layers beneath it and is completely independent of the layers on top of it. In this way, these layers can be used strictly, where the layer only knows of the layer beneath it, or in a flexible manner, and access all layers beneath it.



There are several advantages to using layered architecture:

- Layers are autonomous: A group of changes in one layer does not affect the others. This is good because we can increase the functionality of a layer, for example, making an application that works only on PCs to work on phones and tablets, without having to rewrite the whole application.
- Layers allow better system customization.

There are also a few key disadvantages:

- Layers make an application more difficult to maintain. Each change requires analysis.
- Layers may affect application performance because they create overhead in execution: each layer in the upper levels must connect to those in the lower levels for each operation in the system.

Level 3: Division into classes

a) The model package – contains the main classes: Client, Product and Order.

Each class corresponds to an entry in the table and stores information provided by the fields from the database. The client class corresponds to the client table and stores important information about clients such as name, address, email and phone number. The product class stores information about the products from the database, such as name, price, description and the quantity left in stock. The order class keeps information from the order table, storing information such as the total price, client's id, product's id and quantity of the product. Each class has an unique identifier, an id which assures the uniqueness in the table such that the information won't be mixed up.

b) The presentation package – contains the classes which implement the graphical user interface.

Each class is represented by a JFrame and plays a specific role for the application. The main frame, named MainFrame is formed from a multiple buttons which will eventually lead to the other frames: ClientFrame, ProductFrame and OrderFrame. Each frame has specific operations: the ClientFrame and ProductFrame will contain the following operaions: add, edit, view, delete. The OrderFrame class contains only two JComboBox objects and a button which plays the role in proceeding the order selected.

c) The Business Logic Layer package – contains classes that will concretely implement the neede operations on objects.

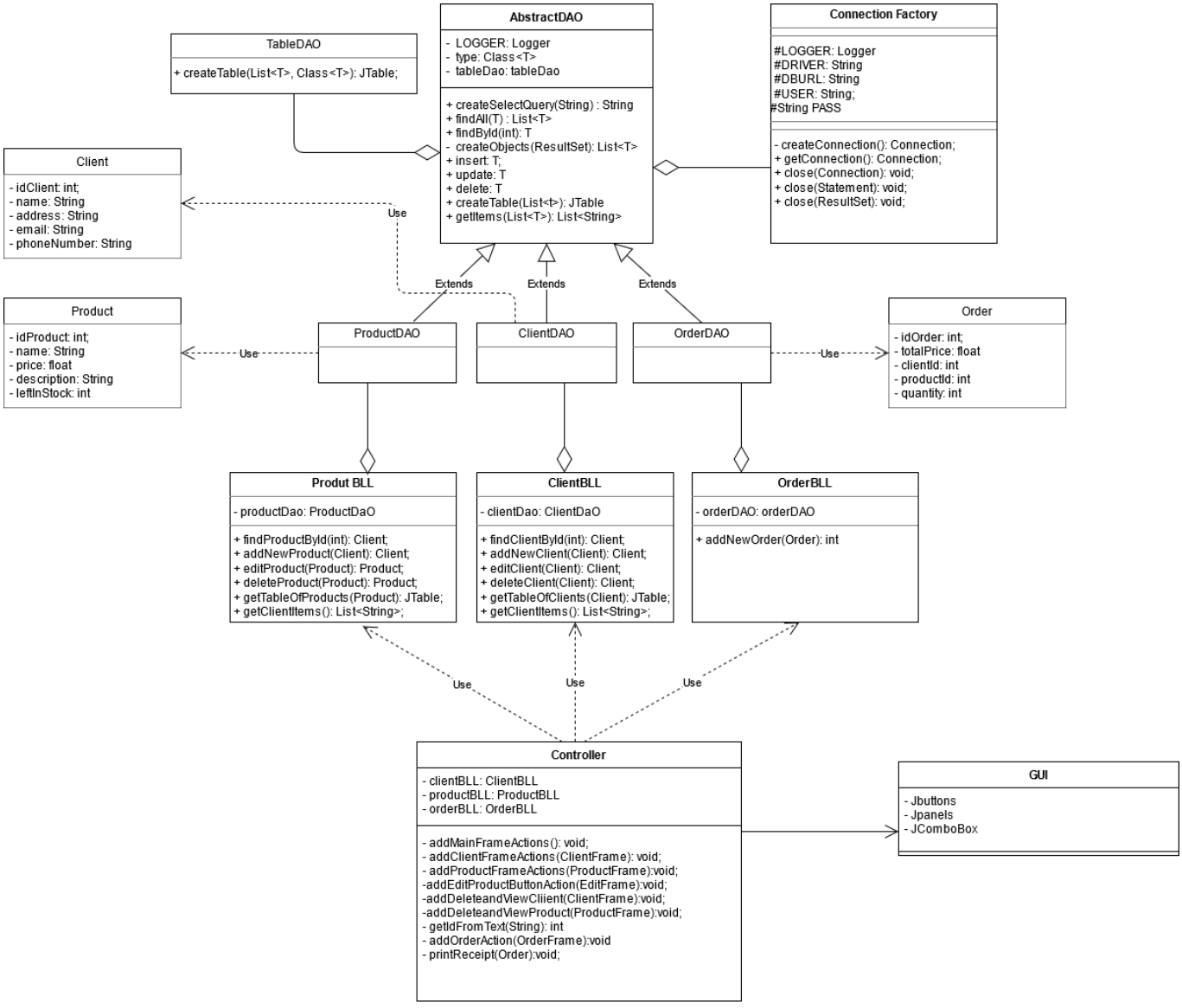
ClientBLL class uses an object of type ClientDAO for returning the needed types from the generic type. ProductBLL class is using a ProductDAO object in order to declare the needed methods for performing operations on the products. The OrderBLL uses an OrderDAO object in order to declare methods for processing the information regarding the orders from the database.

d) DAO package – contains the generic classes that use reflection in order to provide a dynamic implementation of the fields

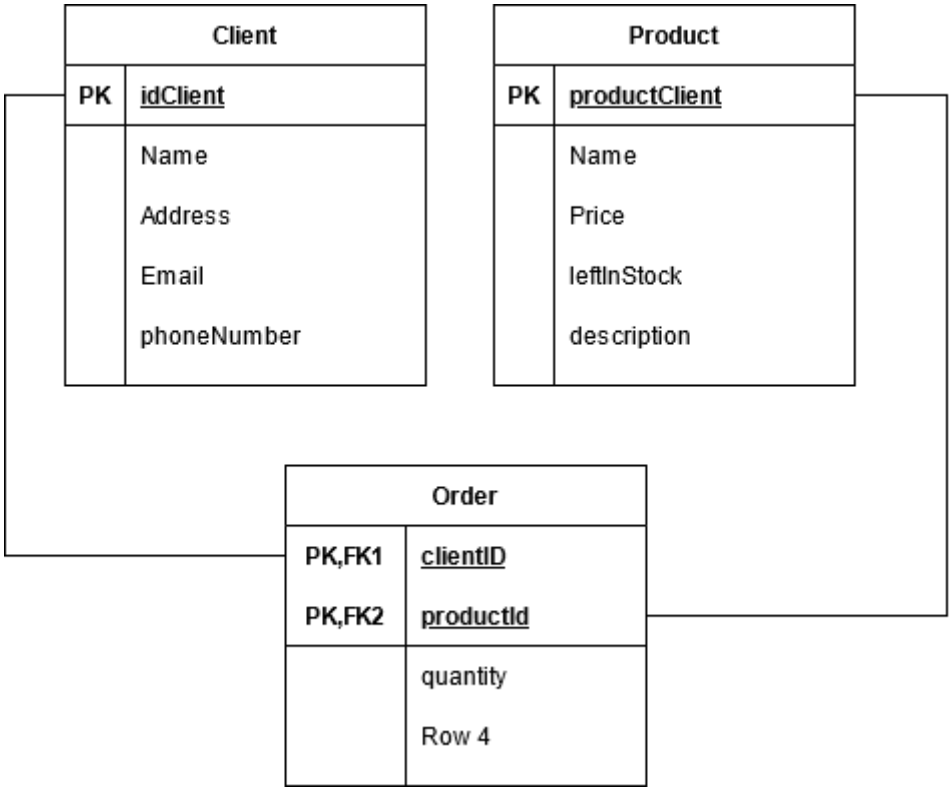
The main class that declares the methods is the AbstractDao class which emphasizes the use of reflection techniques. The ClientDAO, OrderDAO and ProductDAO classes are just extensions of this class and the TableDAO class is providing a generic implementation for creating a JTable with dynamically allocated elements.

e) Connection – provides a single class ConnectionFactory which is meant to provide the connection with the database.
f) mainController package – contains the Controller class which provides the connection logic between the gui and all the classes mentioned above.

UML diagram

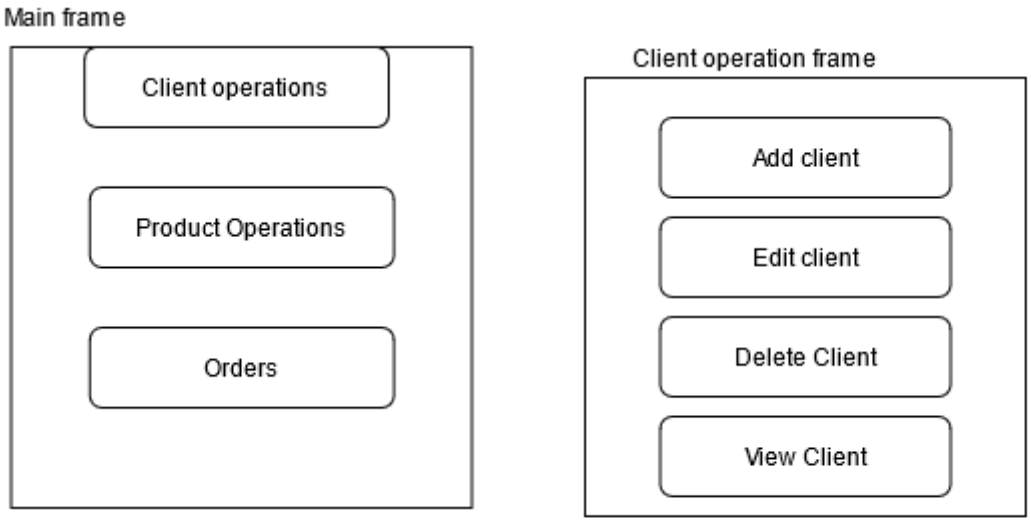


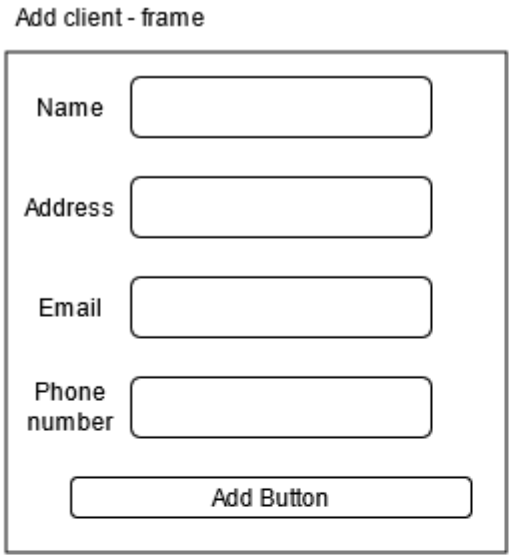
Database Relationships

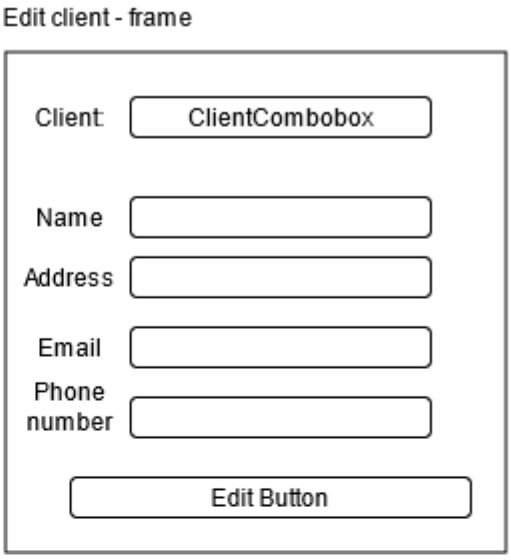


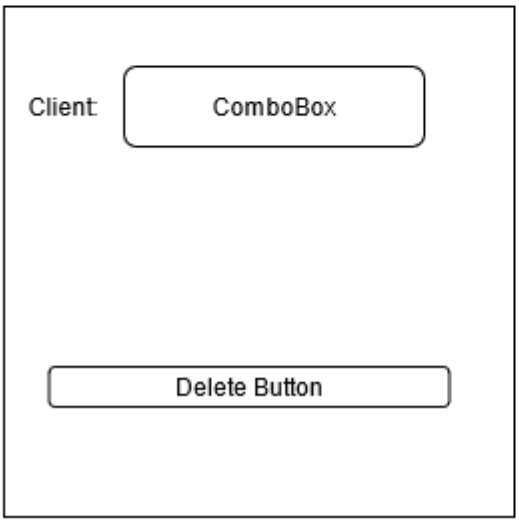
GUI

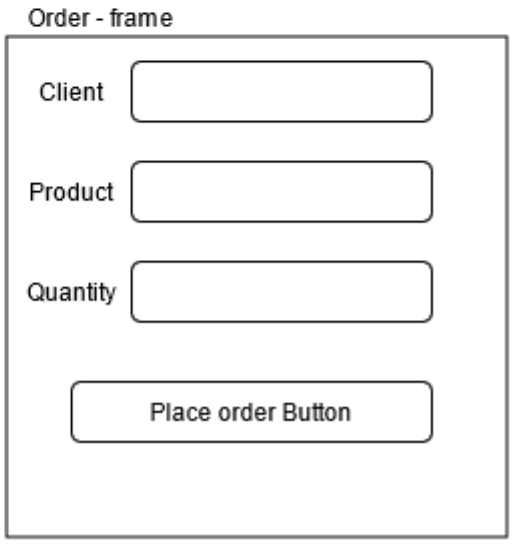
MainFrame GUI











Every button has an action listener that triggers the operation that is needed to be performed. When the one of the buttons Add, Delete, Edit, Place Order is pressed the operation will be performed on the database as it is required. The entries in the database will be modified or deleted regarding the operation that was selected. The GUI for the product is similar with the GUI for the clients.

4. Implementation

The implementation is based on the reflection feature that is offered by JAVA. Reflection allows to a Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its members and display them. This feature allows programmers to reuse code for implementing similar behaviors in methods. This will help in identifying the fields from each class and use them for creating the needed queries.

For creating and executing queries on a generic object, the following algorithms were used:

```
Insert( GenericObject genericObject) {
        dataBaseConnection = initializeDataBaseConnection ( )
        query = "INSERT INTO " + tableName
        for each field of genericObject{
                query.appent( field + "," );
        }
        query.append(lastField_of_genericObject + ") VALUES (");
        for each field of genericObject{
                query.append(field.value + ",")
        }
        query.append(lastField_of_genericObject.value + ")")
        query.execute();
}
```

```
Update( GenericObject genericObject) {
        dataBaseConnection = initializeDataBaseConnection ( )
        query = "UPDATE " + tableName
        for each field of GenericObject {
                query.appent( field + " = " )
                query.append(field.value + ",")
        }
        query.append( lastField + " = " )
        query.append(lastField.value + ",")
        query.execute();
}
```

```
Delete( GenericObject genericObject) {
        dataBaseConnection = initializeDataBaseConnection ( )
        query = “DELTE FROM “ + tableName + “WHERE”
        query.append( GenericObject.firstField + “= “ genericObject.FirstFieldValue);
        query.execute();
}
```

For retrieving the type of the Generic Object there were used reflection techniques:

```
this.type = (Class<T>) ((ParameterizedType)
getClass().getGenericSuperclass()).getActualTypeArguments()[0];
```

ParameterizedType represents a parameterized type such as Collection<String>.A parameterized type is created the first time it is needed by a reflective method, as specified in this package. When a parameterized type p is created, the generic type declaration that p instantiates is resolved, and all type arguments of p are created recursively.

.getGenericSuperclass() returns the Type representing the direct superclass of the entity (class, interface, primitive type or void) represented by this Class.If the superclass is a parameterized type, the Type object returned must accurately reflect the actual type arguments used in the source code.

getActualTypeArguments returns an array of Type objects representing the actual type arguments to this type.

The Java code implementing the algorithm is as it follows:

Update method:

```
public T update(T t) {
    Connection connection = null;
    PreparedStatement updateStatement = null;
    String tableName = t.getClass().getSimpleName().toLowerCase();
    StringBuilder updateStatementString = new StringBuilder("UPDATE
order_management_system." + tableName + " SET ");
    int i;
    try {
         //  Establishing connection with the database

        Object fieldValue;
        connection = ConnectionFactory.getConnection();
        Field fields[] = t.getClass().getDeclaredFields();

         // Processing the fields names and values and appending them to the
query

        for (i = 1; i < fields.length - 1; i++) {
```

```
            fields[i].setAccessible(true);
            fieldValue = fields[i].get(t);
            updateStatementString.append(fields[i].getName() + "=" + "'" +
fieldValue + "' ");
            updateStatementString.append(",");
        }

         // Processing the last element and using the first element into the
condition statement
         // The value of the object will be updated with respect to its id

        fields[i].setAccessible(true);
        fields[0].setAccessible(true);
        fieldValue = fields[i].get(t);
        updateStatementString.append(fields[i].getName() + "=" + "'" +
fieldValue + "' ");
        fieldValue = fields[0].get(t);
        updateStatementString.append("WHERE " + fields[0].getName() + " = " +
fieldValue);
        updateStatement =
connection.prepareStatement(updateStatementString.toString(),
Statement.RETURN_GENERATED_KEYS);
        updateStatement.executeUpdate();

    } catch (Exception e) {
        LOGGER.log(Level.WARNING, type.getName() + "DAO: update " +
e.getMessage());
    } finally {
         // Closing the connections

        ConnectionFactory.close(updateStatement);
        ConnectionFactory.close(connection);
    }
    return t;
}
```

Delete method:

```
public T delete(T t) {
    Connection connection = null;
    PreparedStatement deleteStatement = null;
    String tableName = t.getClass().getSimpleName().toLowerCase();
    StringBuilder deleteStatementString = new StringBuilder("DELETE FROM
order_management_system." + tableName + " WHERE ");
    int i;
    try {
        // Establishing connection with the database
        Object fieldValue;
        connection = ConnectionFactory.getConnection();
        Field fields[] = t.getClass().getDeclaredFields();
        fields[0].setAccessible(true);
        fieldValue = fields[0].get(t);
        deleteStatementString.append( fields[0].getName() + " = " + fieldValue);
        deleteStatement =
connection.prepareStatement(deleteStatementString.toString(),
Statement.RETURN_GENERATED_KEYS);
        deleteStatement.executeUpdate();
```

```
    } catch (Exception e) {
        LOGGER.log(Level.WARNING, type.getName() + "DAO: delete " +
e.getMessage());
    } finally {

        ConnectionFactory.close(deleteStatement);
        ConnectionFactory.close(connection);
    }
    return t;
}
```

Method for finding by id an object:

```
public T findById(int id) {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;

    String query = createSelectQuery(type.getDeclaredFields()[0].getName());
    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.setInt(1, id);
        resultSet = statement.executeQuery();

        return createObjects(resultSet).get(0);
    } catch (SQLException e) {
        LOGGER.log(Level.WARNING, type.getName() + "DAO:findById " +
e.getMessage());
    } finally {
        ConnectionFactory.close(resultSet);
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }
    return null;
}
```

5. Results

The application implements the minimal function of an order management system. It can insert, edit and delete operation in the databases and also work with objects in a reflexive and dynamic manner,

6. Conclusions

The application provides a good example for using reflection features offered by Java. Working with a database can involve using the same queries for different type of tables and using reflection techniques the code can be reused making it easier for the programmer.

7. Bibliography

- FUNDAMENTAL PROGRAMMING TECHNIQUES – ASSIGNMENT3 – SUPPORT PRESENTATION
- https://javapapers.com/oops/association-aggregation-composition-abstraction-generalization-realization-dependency/

- https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html
- https://www.oracle.com/technical-resources/articles/java/javareflection.html