MINISTRY OF EDUCATION AND RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

PROGRAMMING TECHNIQUES

ASSIGNMENT 3

ORDERS MANAGEMENT

Reckerth Daniel-Peter

30444/2

Contents

I. Assignment Objective

The fourth assignment aims to create and implement an order management application for processing client orders for a warehouse. Relational databases should be used to store the products, the clients, and the orders. The application should be designed according to the layered architecture pattern 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

II. Analysis, Modelling, Scenarios, Use Cases

1. Problem Analysis

The real-life problem which arises is how to efficiently manage orders and processing clients and products. Storing the information into a database for further use is also important. Therefore the products, orders and clients should be stored into a databas.e

2. Modelling the Problem

When we talk about modelling, we talk about high-level abstractions, devoid of the complexity and low-level details.

To model the problem, we must think of many things like products, orders and clients. Therefore, we modeled the clients minimally, i.e. they have a name, address and phone numbers. Products have a description, quantity (stock) and price. For the order we have taken a different approach meaning we have order and purchase order. The order models data of an order of a product, i.e. we account for the quantity ordered of the product and the total price computed from it. In the purchase order (invoice) we also model the client and the order which he/she made.

3. Use Case Scenarios

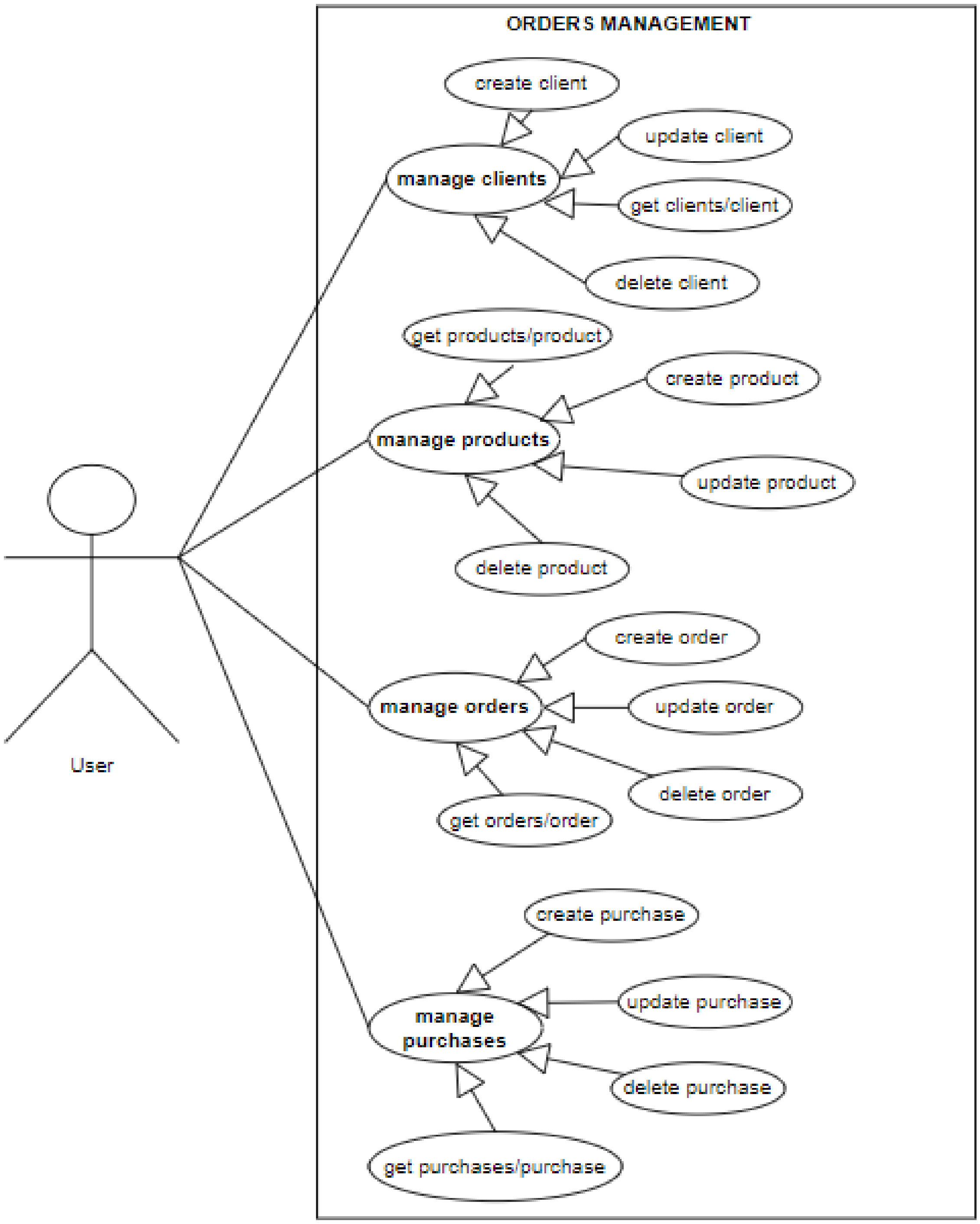
Scenarios represent different actions that can be taken by the user of the application. Together with a use case, they represent a list of actions or event steps typically defining the interactions between a role (known in UML as an actor) and the system to achieve a goal. [Wikipedia]

Therefore, a use case is an important and indispensable requirement analysis technique. Some actions the user can take which we talked before are:

- Manage clients which includes:
    - Create client

    - Update client
    - Get clients and client
    - Delete client
- Manage products which includes:
    - Create product
    - Update product
    - Get products and product
    - Delete product
- Manage orders which includes
    - Create new order which verifies if the order can be placed (i.e. if an understock can occur)
    - Update order
    - Delete order
    - Get orders and order by id
- Manage purchase orders
    - Create a purchase order (client and order)
    - Update a purchase order
    - Delete a purchase order
    - Get purchases and purchase by id

The following shows the use case diagram of the user.


ORDERS MANAGEMENT
create client
update client
manage clients
get clients/client
delete client
get products/product
create product
manage products
update product
delete product
create order
manage orders
update order
delete order
get orders/order
create purchase
update purchase
manage purchases
delete purchase
get purchases/purchase
User

III. Design

1. Design Decisions

Regarding the design decisions we had to take into account that some constraints were imposed by the assignment.

Firstly, we had to organize our assignment and project into layers, i.e. we had to use the layered architecture pattern. Moreover, we needed to use reflection techniques in order to create methods that access the database.

The presentation layer concerns with the UI. It is a desktop based UI which takes user action and sends it to the controller and at the end it shows result taken from controller to the user. We have somehow followed the MVC pattern in order to achieve this. The presentation layer contains both the view and the controller concerned classes.

Database handling layer of the application. It may contain entity definitions, ORM framework or DB connection codes having SQL sentences, according to the abstraction decision. Its role is getting data from controller, performing data operation on database and sending results again to controller (if result exists). Database independence is a very important plus for this layer, which brings flexibility.

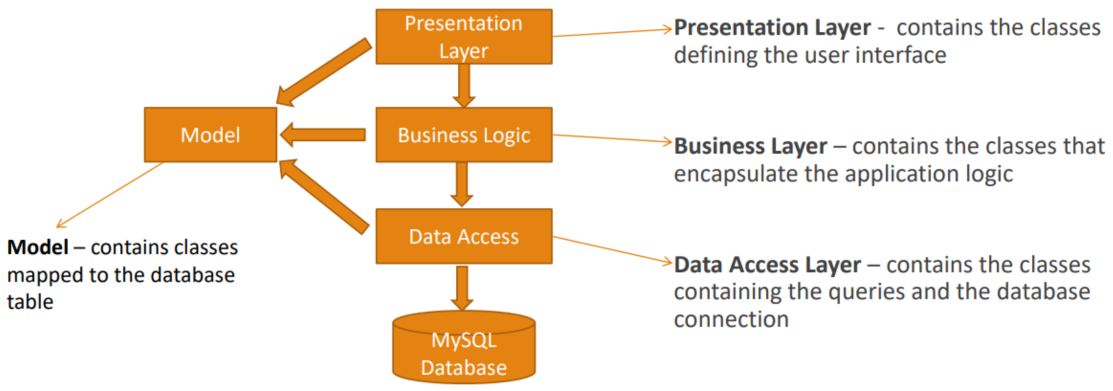
Model layer contains our model classes, namely the one which models products, orders, clients and purchases order.

We also have the business logic layer in which our services reside which use the classes from the data access layer.

2. UML Diagrams

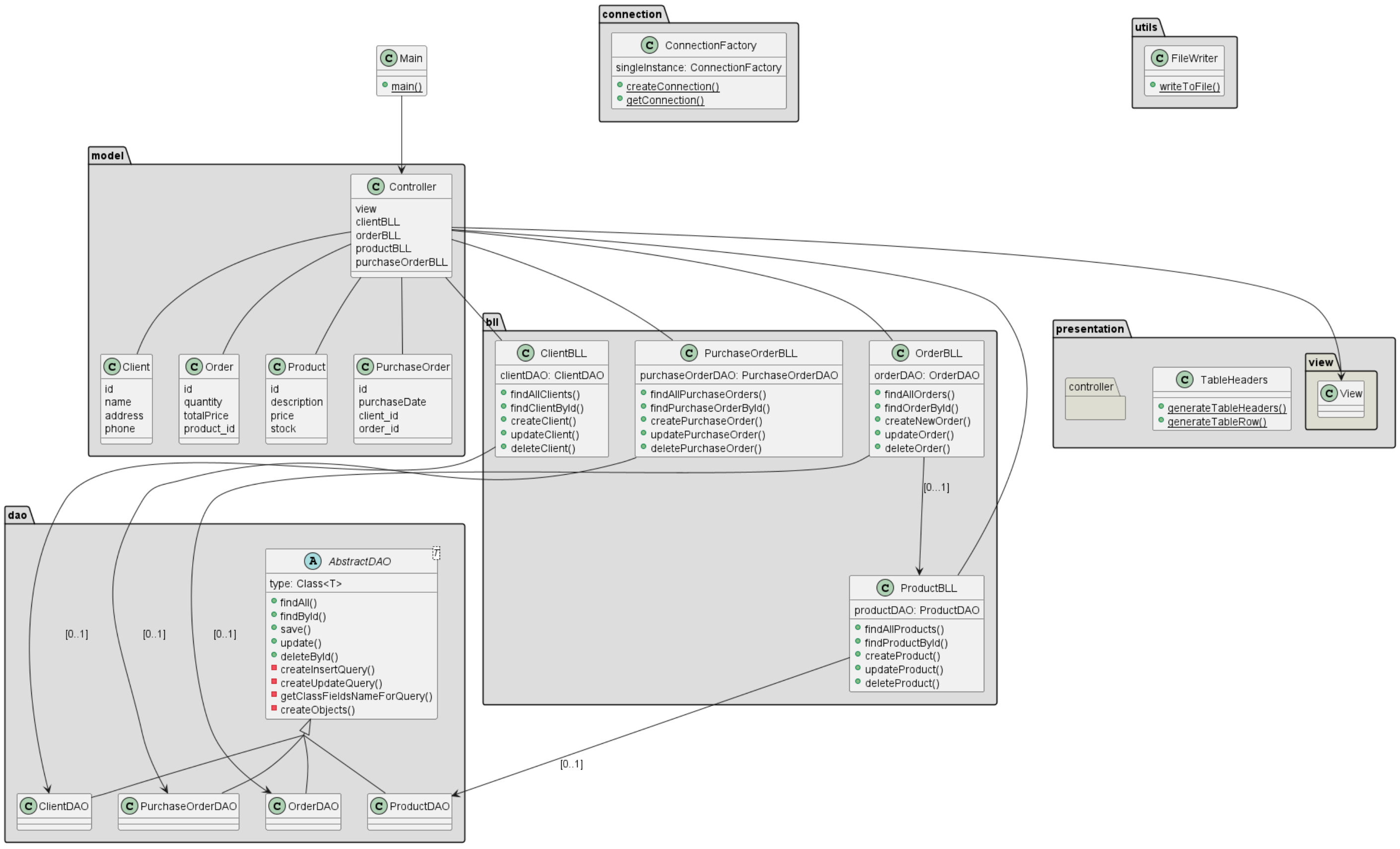
2.1. Class Diagram

We present next the class diagram of the system. To be noted that we also showcase here the packages they are into. Also, we have been provided with an initial class diagram as a starting design of the system. The initial diagram showcases three layers: presentation, data access and business logic. We will showcase now the initial conceptual architecture.


Presentation Layer
Business Logic
Data Access
MySQL Database
Model
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

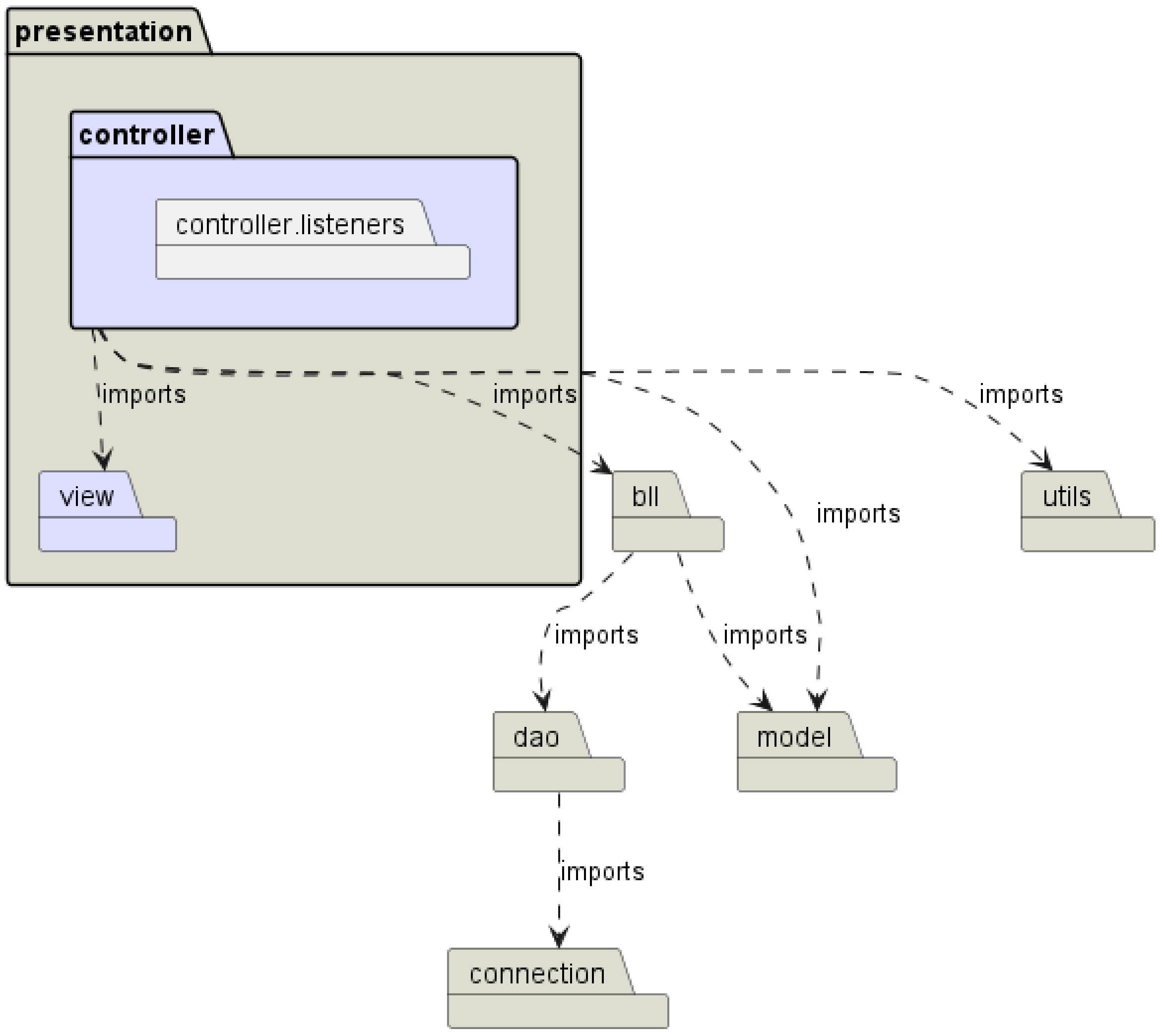
The complete diagram of our class after having the previously as starting point is presented below.

As it can be seen it generally follows the initial diagram however additional classes are included like the ones used for user concerns (user modeled entity, user service) and the controllers, whose role is to bind the view and the business data for the user's actions to take effect.



2.2. Package Diagram

Below is presented the package diagram. As it can be seen the package organization follows the required prerequisites. We have the presentation packages which contains the view and controller sub-packages. As it can be seen, it imports other packages which use it as the business logic one, the model and the utils package in which we have a class for writing a txt file. Also the business logic layer imports other packages like the data access one and the model layer.



3. Data Structures

Regarding data structures, we do not have special data structures. We have used the normal ones, meaning List for storing orders, products or clients when retrieved from the database. Besides this we have not use some special data structures. We have worked with 2D arrays for generating the rows data for the JTable using reflection.

4. Designing of Classes

Designing of classes follows the requirements. We have designed the model classes accordingly, containing information about the product, order, purchase order or client. In the data access layer we have design a class called AbstractDAO which uses a generic type and performs common CRUD database operation. Most of these operation use reflection techniques in order to access and retrieve the fields or methods of the received object, in order to achieve this abstractisation. We then defined DAO classes for each model entity – client, product, order, purchase, order – which extend this AbstractDAO class with the corresponding type.

In the presentation, we have designed our view as tabbed JPanel and then register it in the controller. We have JTables for each of the model class and in the controller, we also registered all the needed action listeners and mouse listeners. In the controller we also registered all of our services which we use for certain actions as presented in the use cases.

The business logic layer contains the service classes, each using the corresponding DAO class. We perform here additional logic, like throwing exceptions when entities are not found or some certain operations are not accordingly.

5. Algorithms

No special algorithms have been developed in this assignment, but there are some interesting implementations.

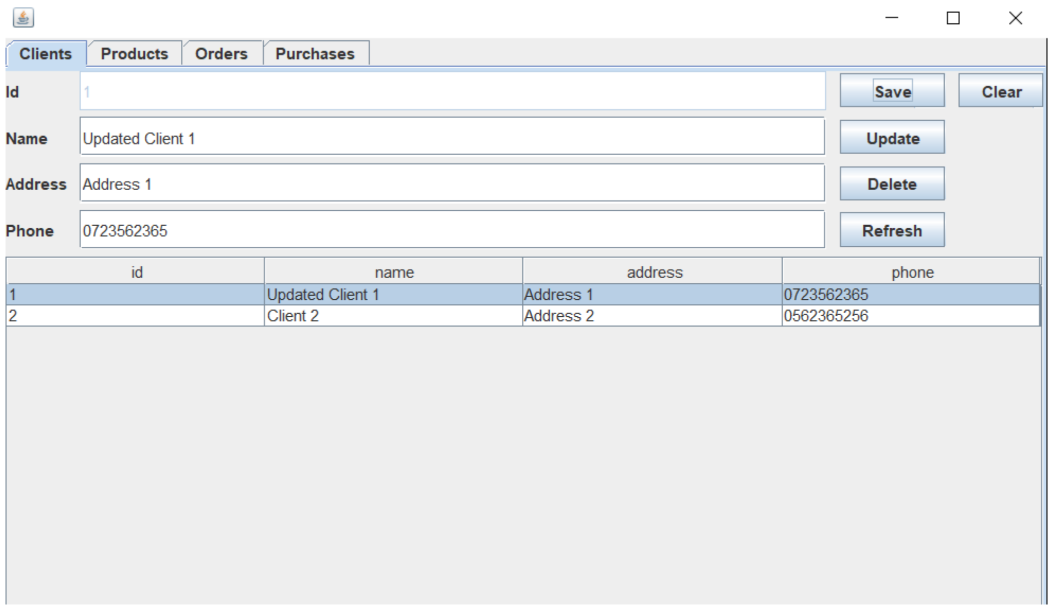
Most of the methods are used following the reflection techniques especially in the DAO classes. We build the query by retrieving the declared fields and methods of the receiving object and we keep it as general as possible. We try not to conditionally create queries based on the model classes we have but keep it open-for-extension if some additional model entities are made in the application.

In the business logic layer we simply call this methods, so nothing special. However when placing a new order we had to take into account some special conditions like the ordered quantity not to be greater than the currently stock and that we automatically compute the total price of the order.

6. Interface

In this chapter we will present some snippets of the user interface. As said earlies we use JTabbedPane in order to have all the panes corresponding to clients, products, order and purchase order.

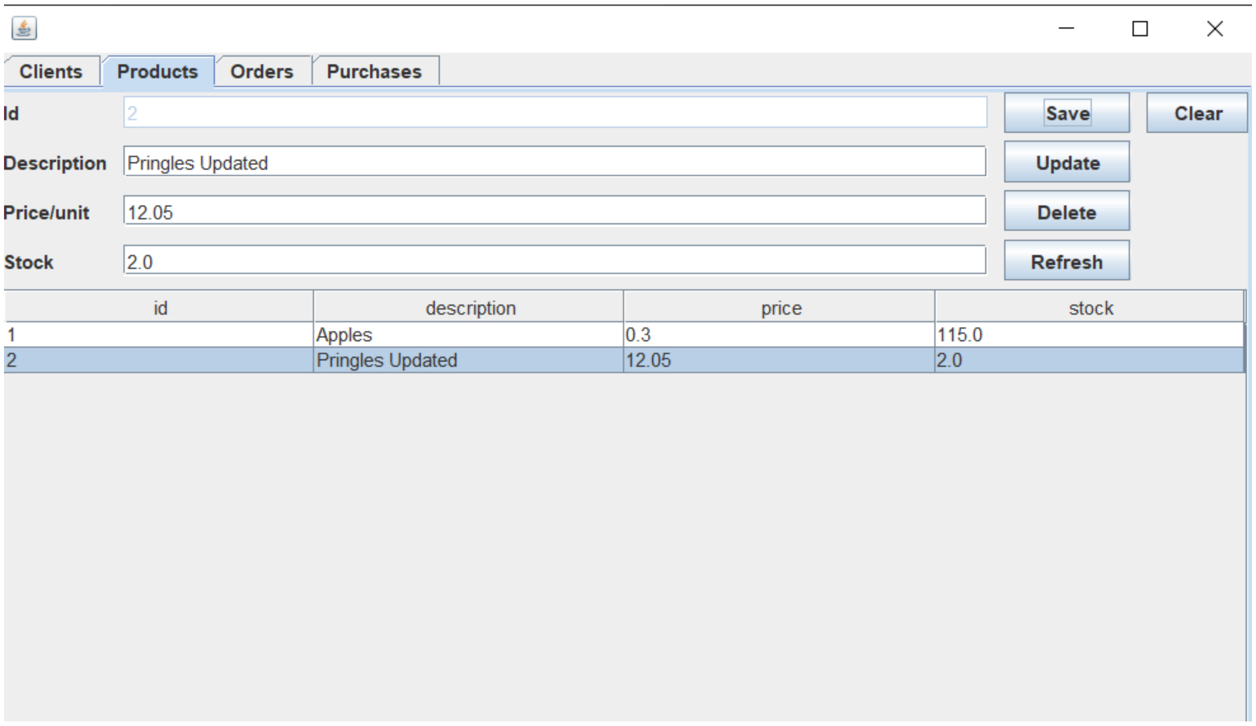
Present here is the view of the clients:



Id	1
Name	Updated Client 1
Address	Address 1
Phone	0723562365

id	name	address	phone
1	Updated Client 1	Address 1	0723562365
2	Client 2	Address 2	0562365256

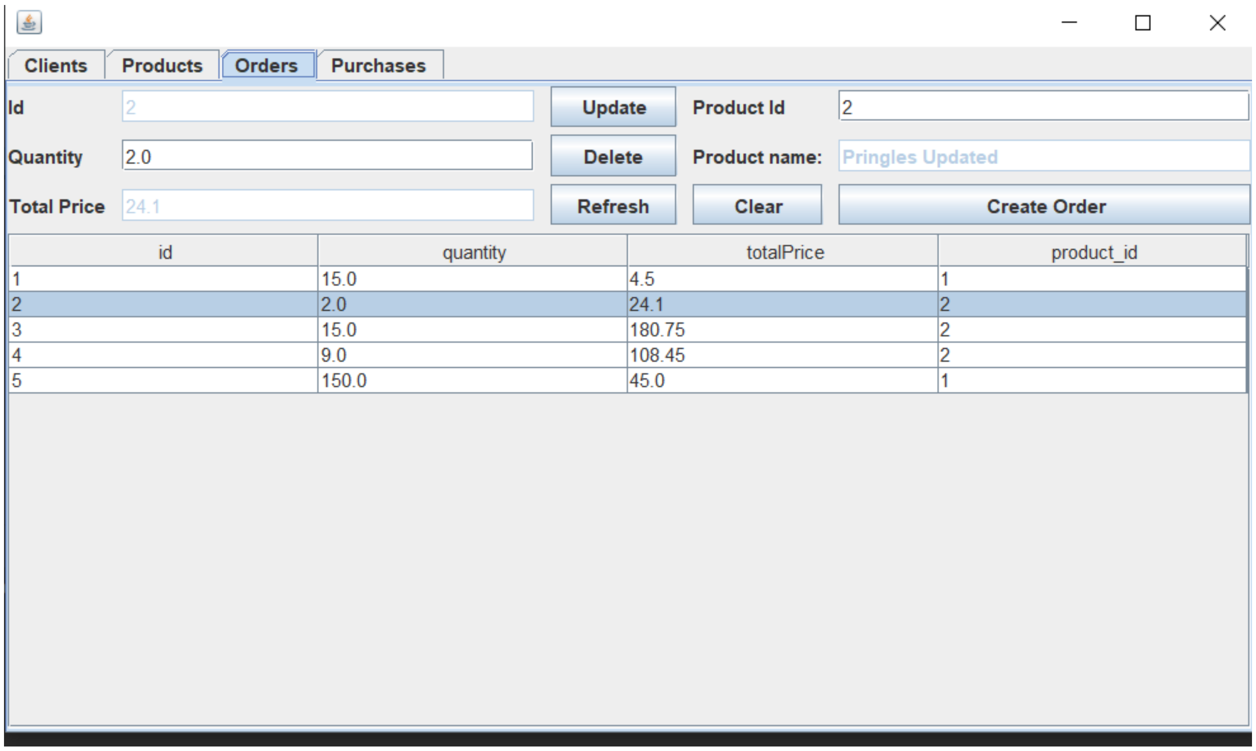
Next one is the view of the products:



Id	2
Description	Pringles Updated
Price/unit	12.05
Stock	2.0

id	description	price	stock
1	Apples	0.3	115.0
2	Pringles Updated	12.05	2.0

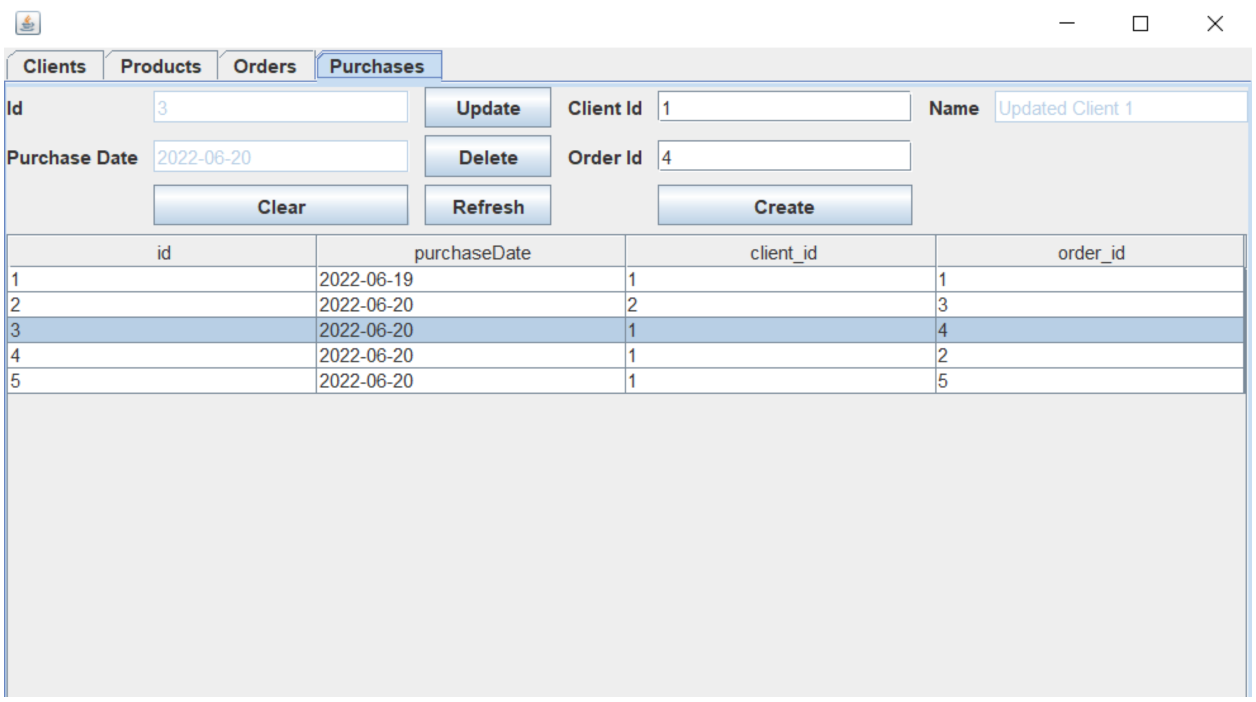
Next we have the orders view:



Id	2
Quantity	2.0
Total Price	24.1
Product Id	2
Product name:	Pringles Updated

id	quantity	totalPrice	product_id
1	15.0	4.5	1
2	2.0	24.1	2
3	15.0	180.75	2
4	9.0	108.45	2
5	150.0	45.0	1

And finally the purchase orders view:



Id	3	Client Id	1	Name	Updated Client 1
Purchase Date	2022-06-20	Order Id	4		

id	purchaseDate	client_id	order_id
1	2022-06-19	1	1
2	2022-06-20	2	3
3	2022-06-20	1	4
4	2022-06-20	1	2
5	2022-06-20	1	5

From each panel the user can perform all the aforementioned use cases.

IV. Implementation

1. Business Logic

In the business logic we have implemented the services which uses the DAO classes. Some are presented below

1.1. ClientBLL

```
package bll;

import dao.ClientDAO;
import model.Client;

import java.util.List;
import java.util.NoSuchElementException;

/**
 * Client service (Business Logic Layer)
 *
 * @author Daniel Reckerth
 */
public class ClientBLL {
  private final ClientDAO clientDAO;
```

```
public ClientBLL(ClientDAO clientDAO) {
  this.clientDAO = clientDAO;
}

/**
 * Find all clients
 *
 * @return List of clients
 */
public List<Client> findAllClients() {
  final List<Client> clients = clientDAO.findAll();
  if (clients.isEmpty()) {
    throw new NoSuchElementException("No clients found");
  }
  return clients;
}

/**
 * Find client by id
 *
 * @param id Client id
 * @return Client with specified id
 */
public Client findClientById(int id) {
  final Client client = clientDAO.findById(id);
  if (client == null) {
    throw new NoSuchElementException("Client with id " + id + " does not exist");
  }
  return client;
}

/**
 * Create new client
 *
 * @param client Client to create
 * @return True if client was created successfully, false otherwise
 */
public boolean createClient(Client client) {
  return clientDAO.save(client);
}

/**
 * Update client
```

```
     *
     * @param id     Client id to update
     * @param client Updating client data
     * @return True if client was updated successfully, false otherwise
     */
    public boolean updateClient(int id, Client client) {
        return clientDAO.update(id, client);
    }

    /**
     * Delete client
     *
     * @param id Client id to delete
     * @return True if client was deleted successfully, false otherwise
     */
    public boolean deleteClient(int id) {
        return clientDAO.deleteById(id);
    }
}
```

1.2. OrderBLL

```
package bll;

import dao.OrderDAO;
import model.Order;
import model.Product;

import java.util.List;
import java.util.NoSuchElementException;

/**
 * Order service (business logic layer)
 *
 * @author Daniel Reckerth
 */
public class OrderBLL {
    private final OrderDAO orderDAO;
    private final ProductBLL productBLL;

    public OrderBLL(OrderDAO orderDAO, ProductBLL productBLL) {
        this.orderDAO = orderDAO;
        this.productBLL = productBLL;
```

```
  }

  /**
   * Get all orders
   *
   * @return List of orders
   */
  public List<Order> findAllOrders() {
    final List<Order> orders = orderDAO.findAll();
    if (orders == null || orders.isEmpty()) {
      throw new NoSuchElementException("No orders found");
    }
    return orders;
  }

  /**
   * Get order by id
   *
   * @param id Order id
   * @return Order
   */
  public Order findOrderById(int id) {
    final Order order = orderDAO.findById(id);
    if (order == null) {
      throw new NoSuchElementException("Order with id " + id + " does not exist");
    }
    return order;
  }

  /**
   * Create new order
   *
   * @param order Order to create
   * @return True if order was created, false otherwise
   */
  public boolean createNewOrder(Order order) {
    try {
      Product product = productBLL.findProductById(order.getProduct_id());
      if (order.getQuantity() > product.getStock()) {
        throw new IllegalArgumentException("Not enough stock of product with id " +
order.getProduct_id());
      }
      product.setStock(product.getStock() - order.getQuantity());
      productBLL.updateProduct(product.getId(), product);
      order.setTotalPrice(product.getPrice() * order.getQuantity());
      return orderDAO.save(order);
    } catch (NoSuchElementException e) {
      throw new NoSuchElementException("Product with id " + order.getProduct_id() + " does not
exist");
    }

  }

  /**
   * Update order
   *
   * @param id    Order id
   * @param order Order to update
   * @return True if order was updated, false otherwise
   */
  public boolean updateOrder(int id, Order order) {
    try {
      Product product = productBLL.findProductById(order.getProduct_id());
      if (order.getQuantity() > product.getStock()) {
        throw new IllegalArgumentException("Not enough stock of product with id " +
order.getProduct_id());
      }
      product.setStock(product.getStock() - order.getQuantity());
      productBLL.updateProduct(product.getId(), product);
      order.setTotalPrice(product.getPrice() * order.getQuantity());
      return orderDAO.update(id, order);
    } catch (NoSuchElementException e) {
      throw new NoSuchElementException("Product with id " + order.getProduct_id() + " does not
exist");
    }
  }

  /**
   * Delete order
   *
   * @param id Order id
   * @return True if order was deleted, false otherwise
   */
  public boolean deleteOrder(int id) {
    return orderDAO.deleteById(id);
  }
}
```

1.3. ProductBLL

```
package bll;

import dao.ProductDAO;
import model.Product;

import java.util.List;
import java.util.NoSuchElementException;

/**
 * Product service (business logic layer)
 *
 * @author Daniel Reckerth
 */
public class ProductBLL {
  private final ProductDAO productDAO;

  public ProductBLL(ProductDAO productDAO) {
    this.productDAO = productDAO;
  }

  /**
   * Get all products
   *
   * @return List of products
   */
  public List<Product> findAllProducts() {
    final List<Product> clients = productDAO.findAll();
    if (clients.isEmpty()) {
      throw new NoSuchElementException("No products found");
    }
    return clients;
  }

  /**
   * Get product by id
   *
   * @param id Product id
   * @return Product
   */
  public Product findProductById(int id) {
    final Product product = productDAO.findById(id);
    if (product == null) {
      throw new NoSuchElementException("Product with id " + id + " does not exist");
    }
    return product;
  }

  /**
   * Create new product
   *
   * @param product Product to create
   * @return True if product was created, false otherwise
   */
  public boolean createProduct(Product product) {
    return productDAO.save(product);
  }

  /**
   * Update product
   *
   * @param id      Product id
   * @param product Product to update
   * @return True if product was updated, false otherwise
   */
  public boolean updateProduct(int id, Product product) {
    return productDAO.update(id, product);
  }

  /**
   * Delete product
   *
   * @param id Product id
   * @return True if product was deleted, false otherwise
   */
  public boolean deleteProduct(int id) {
    return productDAO.deleteById(id);
  }
}
```

1.4. PurchaseOrderBLL

```
package bll;

import dao.PurchaseOrderDAO;
import model.PurchaseOrder;
```

```java
import java.util.List;
import java.util.NoSuchElementException;

/**
 * Purchase service (business logic layer)
 *
 * @author Daniel Reckerth
 */
public class PurchaseOrderBLL {
  private final PurchaseOrderDAO purchaseOrderDAO;

  public PurchaseOrderBLL(PurchaseOrderDAO purchaseOrderDAO) {
    this.purchaseOrderDAO = purchaseOrderDAO;
  }

  /**
   * Get all purchase orders
   *
   * @return List of purchase orders
   */
  public List<PurchaseOrder> findAllPurchaseOrders() {
    final List<PurchaseOrder> purchaseOrders = purchaseOrderDAO.findAll();
    if (purchaseOrders.isEmpty()) {
      throw new NoSuchElementException("No purchase orders found");
    }
    return purchaseOrders;
  }

  /**
   * Get purchase order by id
   *
   * @param id Id of purchase order
   * @return Purchase order found
   */
  public PurchaseOrder findPurchaseOrderById(int id) {
    final PurchaseOrder purchaseOrder = purchaseOrderDAO.findById(id);
    if (purchaseOrder == null) {
      throw new NoSuchElementException("Purchase order with id " + id + " does not exist");
    }
    return purchaseOrder;
  }
  /**
   * Create purchase order
   *
   * @param purchaseOrder Purchase order to create
   * @return True if purchase order was created, false otherwise
   */
  public boolean createPurchaseOrder(PurchaseOrder purchaseOrder) {
    return purchaseOrderDAO.save(purchaseOrder);
  }

  /**
   * Update purchase order
   *
   * @param id            Id of purchase order to update
   * @param purchaseOrder Purchase order to update
   * @return True if purchase order was updated, false otherwise
   */
  public boolean updatePurchaseOrder(int id, PurchaseOrder purchaseOrder) {
    return purchaseOrderDAO.update(id, purchaseOrder);
  }

  /**
   * Delete purchase order
   *
   * @param id Id of purchase order to delete
   * @return True if purchase order was deleted, false otherwise
   */
  public boolean deletePurchaseOrder(int id) {
    return purchaseOrderDAO.deleteById(id);
  }
}
```

2. Connection with ConnectionFactory

```
package connection;

import java.sql.*;
import java.util.logging.Logger;

/**
 * Factory for class responsible for database connection.
 *
 * @author Daniel Reckerth
 */
public class ConnectionFactory {
  private static final Logger LOGGER = Logger.getLogger(ConnectionFactory.class.getName());
  private static final String DRIVER = "com.mysql.jdbc.Driver";
  private static final String URL = "jdbc:mysql://localhost:3306/order_management";
  private static final String USER = "root";
  private static final String PASSWORD = "admin";

  private static final ConnectionFactory singleInstance = new ConnectionFactory();

  private ConnectionFactory() {
    try {
      Class.forName(DRIVER);
    } catch (ClassNotFoundException e) {
      LOGGER.severe("Driver not found");
    }
  }

  /**
   * Get single instance of connection factory.
   *
   * @return Single instance of connection factory
   */
  private Connection createConnection() {
    Connection connection = null;
    try {
      connection = DriverManager.getConnection(URL, USER, PASSWORD);
    } catch (SQLException e) {
      LOGGER.warning("Connection failed");
      e.printStackTrace();
    }
    return connection;
  }
```

```
/**
 * Get single instance of connection factory.
 *
 * @return Single instance of connection factory
 */
public static Connection getConnection() {
  return singleInstance.createConnection();
}

/**
 * Close connection.
 *
 * @param connection Connection to close
 */
public static void close(Connection connection) {
  if (connection != null) {
    try {
      connection.close();
    } catch (SQLException e) {
      LOGGER.warning("Connection close failed");
      e.printStackTrace();
    }
  }
}

/**
 * Close statement.
 *
 * @param statement Statement to close
 */
public static void close(Statement statement) {
  if (statement != null) {
    try {
      statement.close();
    } catch (SQLException e) {
      LOGGER.warning("Statement close failed");
      e.printStackTrace();
    }
  }
}

/**
```

```
   * Close result set.
   *
   * @param resultSet Result set to close
   */
  public static void close(ResultSet resultSet) {
    if (resultSet != null) {
      try {
        resultSet.close();
      } catch (SQLException e) {
        LOGGER.warning("ResultSet close failed");
        e.printStackTrace();
      }
    }
  }
}
```

3. DAO

3.1. AbstractDAO

```
package dao;

import java.beans.IntrospectionException;
import java.beans.PropertyDescriptor;
import java.lang.reflect.*;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.logging.Logger;

import static connection.ConnectionFactory.close;
import static connection.ConnectionFactory.getConnection;

/**
 * Abstract Data Access Object (DAO)
 *
 * @param <T> Type of entity
 * @author Daniel Reckerth
 */
public class AbstractDAO<T> {
  protected static final Logger LOGGER = Logger.getLogger(AbstractDAO.class.getName());
  private final Class<T> type;

  @SuppressWarnings("unchecked")
```

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

/**
 * Get all entities
 *
 * @return List of entities
 */
public List<T> findAll() {
  Connection connection = null;
  PreparedStatement statement = null;
  ResultSet resultSet = null;
  String query = "SELECT * FROM order_management.%s".formatted(type.getSimpleName());
  try {
    connection = getConnection();
    statement = connection.prepareStatement(query);
    resultSet = statement.executeQuery();
    return createObjects(resultSet);
  } catch (SQLException throwables) {
    LOGGER.warning(type.getName() + " DAO:findAll " + throwables.getMessage());
    throwables.printStackTrace();
  } finally {
    close(resultSet);
    close(statement);
    close(connection);
  }
  return null;
}

/**
 * Get entity by id
 *
 * @param id Id of entity
 * @return Entity found
 */
public T findById(int id) {
  Connection connection = null;
  PreparedStatement statement = null;
  ResultSet resultSet = null;
  String query = "SELECT * FROM order_management.%s WHERE id =
?".formatted(type.getSimpleName().toLowerCase());
    try {
      connection = getConnection();
      statement = connection.prepareStatement(query);
      statement.setInt(1, id);
      resultSet = statement.executeQuery();

      final List<T> objects = createObjects(resultSet);
      if (objects.size() > 0) {
        return objects.get(0);
      } else {
        return null;
      }
    } catch (SQLException e) {
      LOGGER.warning(type.getName() + " DAO:findById " + e.getMessage());
      e.printStackTrace();
    } finally {
      close(resultSet);
      close(statement);
      close(connection);
    }
    return null;
  }

  /**
   * Create entity
   *
   * @param object Object to create
   * @return True if created, false otherwise
   */
  public boolean save(T object) {
    Connection connection = null;
    PreparedStatement statement = null;
    String query = createInsertQuery(type.getSimpleName().toLowerCase(),
getClassFieldsNameForQuery(object), type.getDeclaredFields().length);
    try {
      connection = getConnection();
      statement = connection.prepareStatement(query);
      int position = 1;
      for (Field field : type.getDeclaredFields()) {
        field.setAccessible(true);
        statement.setObject(position, field.get(object));
        position++;
      }
      statement.executeUpdate();
      return true;
    } catch (SQLException | IllegalAccessException throwables) {
      LOGGER.warning(type.getName() + " DAO:save " + throwables.getMessage());
      throwables.printStackTrace();
    } finally {
      close(statement);
      close(connection);
    }
    return false;
  }

  /**
   * Update entity
   *
   * @param id     Id of entity to update
   * @param object Updating object
   * @return True if updated, false otherwise
   */
  public boolean update(int id, T object) {
    Connection connection = null;
    PreparedStatement statement = null;
    String updateQuery = createUpdateQuery(type.getSimpleName().toLowerCase(),
type.getDeclaredFields());
    try {
      connection = getConnection();
      statement = connection.prepareStatement(updateQuery);
      int position = 1;
      for (Field field : type.getDeclaredFields()) {
        field.setAccessible(true);
        statement.setObject(position, field.get(object));
        position++;
      }
      statement.setInt(position, id);
      statement.executeUpdate();
      return true;
    } catch (SQLException | IllegalAccessException throwables) {
      LOGGER.warning(type.getName() + " DAO:update " + throwables.getMessage());
      throwables.printStackTrace();
    } finally {
      close(statement);
      close(connection);
    }

    return false;
  }

  /**
   * Delete entity
   *
   * @param id Id of entity to delete
   * @return True if deleted, false otherwise
   */
  public boolean deleteById(int id) {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    String query = "DELETE FROM order_management.%s WHERE id =
?".formatted(type.getSimpleName().toLowerCase());
    try {
      connection = getConnection();
      statement = connection.prepareStatement(query);
      statement.setInt(1, id);
      statement.executeUpdate();
      return true;
    } catch (SQLException e) {
      LOGGER.warning(type.getName() + " DAO:deleteById " + e.getMessage());
      e.printStackTrace();
    } finally {
      close(statement);
      close(connection);
    }
    return false;
  }

  /**
   * Create insert query
   *
   * @param className      Table name
   * @param valuesDeclared Fields of the table as string
   * @param numberOfFields Number of fields
   */
  private String createInsertQuery(String className, String valuesDeclared, int numberOfFields)
{
    StringBuilder query = new StringBuilder("INSERT INTO order_management." + className
+ "(" + valuesDeclared + ") VALUES(");
    for (int i = 0; i < numberOfFields; i++) {
      query.append("?");
      if (i != numberOfFields - 1) {
        query.append(", ");
      }
    }
    query.append(")");
    return query.toString();
  }

  /**
   * Create update query
   *
   * @param className Table name
   * @param fields    Fields of the table
   */
  private String createUpdateQuery(String className, Field[] fields) {
    StringBuilder query = new StringBuilder("UPDATE order_management." + className + "
SET ");
    for (int i = 0; i < fields.length; i++) {
      query.append(fields[i].getName()).append(" = ?");
      if (i != fields.length - 1) {
        query.append(", ");
      }
    }
    query.append(" WHERE id = ?");
    return query.toString();
  }

  /**
   * Get fields name of the table as string
   *
   * @param object Object to get fields name
   * @return Fields name of the table as string
   */
  private String getClassFieldsNameForQuery(T object) {
    Class<?> objectClass = object.getClass();
    Field[] classFields = objectClass.getDeclaredFields();
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < classFields.length; i++) {
      stringBuilder.append(classFields[i].getName());
      if (i != classFields.length - 1) {
        stringBuilder.append(", ");
      }
    }
    return stringBuilder.toString();
  }

  /**
   * Create objects from result set
   *
   * @param resultSet Result set
   * @return List of objects
   */
  private List<T> createObjects(ResultSet resultSet) {
    List<T> list = new ArrayList<>();
    Constructor[] constructors = type.getDeclaredConstructors();
    Constructor constructor = null;

    for (Constructor item : constructors) {
      constructor = item;
      if (constructor.getGenericParameterTypes().length == 0) {
        break;
      }
    }

    try {
      while (resultSet.next()) {
        Objects.requireNonNull(constructor).setAccessible(true);
        T instance = (T) constructor.newInstance();
        for (Field field : type.getDeclaredFields()) {
          String fieldName = field.getName();
          Object value = resultSet.getObject(fieldName);
          if (value instanceof java.sql.Date) {
            value = ((Date) value).toLocalDate();
          }
          PropertyDescriptor propertyDescriptor = new PropertyDescriptor(fieldName, type);
          Method method = propertyDescriptor.getWriteMethod();
          method.invoke(instance, value);
        }
        list.add(instance);
      }
    } catch (SQLException | IllegalAccessException | InstantiationException |
InvocationTargetException |
             IntrospectionException throwables) {
```

```
      throwables.printStackTrace();
    }
    return list;
  }

}
```

4. Model

4.1. Client

```
package model;

/**
 * Client model class
 *
 * @author Daniel Reckerth
 */
public class Client {
  private int id;
  private String name;
  private String address;
  private String phone;

  public Client() {
  }

  public Client(int id, String name, String address, String phone) {
    this.id = id;
    this.name = name;
    this.address = address;
    this.phone = phone;
  }
  ….
}
```

4.2. Order

```
package model;

/**
 * Order model class
 *
 * @author Daniel Reckerth
 */
```

```
public class Order {
  private int id;
  private double quantity;
  private double totalPrice;

  private int product_id;

  public Order() {
  }

  public Order(int id, double quantity, double totalPrice, int product_id) {
    this.id = id;
    this.quantity = quantity;
    this.totalPrice = totalPrice;
    this.product_id = product_id;
  }
  ….
 }
```

5. Presentation

5.1. Controller

```
package presentation.controller;

import bll.ClientBLL;
import bll.OrderBLL;
import bll.ProductBLL;
import bll.PurchaseOrderBLL;
import model.Client;
import model.Order;
import model.Product;
import model.PurchaseOrder;
import presentation.controller.listeners.ClientsTableMouseListener;
import presentation.controller.listeners.OrdersTableMouseListener;
import presentation.controller.listeners.ProductsTableMouseListener;
import presentation.controller.listeners.PurchaseTableMouseListener;
import presentation.view.View;
import utils.FileWriter;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.time.LocalDate;

import static javax.swing.JOptionPane.*;
```

```
/**
 * Controller class
 *
 * @author Daniel Reckerth
 */
public class Controller {
  private final View view;
  private final ClientBLL clientBLL;
  private final ProductBLL productBLL;
  private final OrderBLL orderBLL;
  private final PurchaseOrderBLL purchaseOrderBLL;

  /**
   * Constructor. Also registers listeners for the view.
   *
   * @param view             View
   * @param clientBLL        Client service
   * @param productBLL       Product service
   * @param orderBLL         Order service
   * @param purchaseOrderBLL Purchase service
   */
  public Controller(View view, ClientBLL clientBLL, ProductBLL productBLL, OrderBLL
orderBLL, PurchaseOrderBLL purchaseOrderBLL) {
    this.view = view;
    this.clientBLL = clientBLL;
    this.productBLL = productBLL;
    this.orderBLL = orderBLL;
    this.purchaseOrderBLL = purchaseOrderBLL;
    view.addSaveClientButtonActionListener(new AddClientButtonActionListener());
    view.addUpdateClientButtonActionListener(new UpdateClientButtonActionListener());
    view.addDeleteClientButtonActionListener(new DeleteClientButtonActionListener());
    view.addClearClientButtonActionListener(new ClearClientButtonActionListener());
    view.addClientsTableMousedActionListener(new ClientsTableMouseListener(view,
clientBLL));
    view.addRefreshClientButtonActionListener(new RefreshClientButtonActionListener());
    view.addSaveProductButtonActionListener(new AddProductButtonActionListener());
    view.addUpdateProductButtonActionListener(new UpdateProductButtonActionListener());
    view.addDeleteProductButtonActionListener(new DeleteProductButtonActionListener());
    view.addClearProductButtonActionListener(new ClearProductButtonActionListener());
    view.addRefreshProductButtonActionListener(new RefreshProductButtonActionListener());
    view.addProductsTableMousedActionListener(new ProductsTableMouseListener(view,
productBLL));
```

```
    view.addCreateOrderButtonActionListener(new AddOrderButtonActionListener());
    view.addUpdateOrderButtonActionListener(new UpdateOrderButtonActionListener());
    view.addDeleteOrderButtonActionListener(new DeleteOrderButtonActionListener());
    view.addClearOrderButtonActionListener(new ClearOrderButtonActionListener());
    view.addRefreshOrderButtonActionListener(new RefreshOrderButtonActionListener());
    view.addOrdersTableMousedActionListener(new OrdersTableMouseListener(view,
productBLL));
    view.addCreatePurchaseButtonActionListener(new AddPurchaseButtonActionListener());
    view.addUpdatePurchaseButtonActionListener(new UpdatePurchaseButtonActionListener());
    view.addDeletePurchaseButtonActionListener(new DeletePurchaseButtonActionListener());
    view.addClearPurchaseButtonActionListener(new ClearPurchaseButtonActionListener());
    view.addRefreshPurchaseButtonActionListener(new
RefreshPurchaseButtonActionListener());
    view.addPurchasesTableMousedActionListener(new PurchaseTableMouseListener(view,
clientBLL));
  }

  private class AddClientButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      if (view.isClientDataEmpty()) {
        view.showMessage("Please fill all fields!", WARNING_MESSAGE);
      } else {
        int id = Integer.parseInt(view.getIdClient());
        String name = view.getNameClient();
        String address = view.getAddressClient();
        String phone = view.getPhoneClient();
        Client client = new Client(id, name, address, phone);
        if (clientBLL.createClient(client)) {
          view.showMessage("Client added successfully!", INFORMATION_MESSAGE);
        } else {
          view.showMessage("Client not added!", ERROR_MESSAGE);
        }
      }
    }
  }

  private class UpdateClientButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      int id = Integer.parseInt(view.getIdClient());
      String name = view.getNameClient();
      String address = view.getAddressClient();
```

```
        String phone = view.getPhoneClient();
        if (view.isClientDataEmpty()) {
            view.showMessage("Please fill all fields!", WARNING_MESSAGE);
        } else {
            Client client = new Client(id, name, address, phone);
            if (clientBLL.updateClient(id, client)) {
                view.showMessage("Client updated successfully!", INFORMATION_MESSAGE);
            } else {
                view.showMessage("Client not updated!", ERROR_MESSAGE);
            }
        }
    }
}

private class DeleteClientButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        String id = view.getIdClient();
        if (id == null || id.isEmpty()) {
            view.showMessage("Select a client!", WARNING_MESSAGE);
        } else {
            if (clientBLL.deleteClient(Integer.parseInt(id))) {
                view.showMessage("Client deleted successfully!", INFORMATION_MESSAGE);
            } else {
                view.showMessage("Client not deleted!", ERROR_MESSAGE);
            }
        }
    }
}

private class ClearClientButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        view.clearClientData();
    }
}

private class RefreshClientButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        view.setClientsTable(clientBLL.findAllClients());
    }
}
```

```java
private class AddProductButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    if (view.isProductDataEmpty()) {
      view.showMessage("Please fill all fields!", WARNING_MESSAGE);
    } else {
      int id = Integer.parseInt(view.getIdProduct());
      String description = view.getDescriptionProduct();
      double price = Double.parseDouble(view.getPriceProduct());
      int stock = Integer.parseInt(view.getStockProduct());
      Product product = new Product(id, description, price, stock);
      if (productBLL.createProduct(product)) {
        view.showMessage("Product added successfully!", INFORMATION_MESSAGE);
      } else {
        view.showMessage("Product not added!", ERROR_MESSAGE);
      }
    }
  }
}

private class UpdateProductButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    int id = Integer.parseInt(view.getIdProduct());
    String description = view.getDescriptionProduct();
    double price = Double.parseDouble(view.getPriceProduct());
    double stock = Double.parseDouble(view.getStockProduct());
    if (view.isProductDataEmpty()) {
      view.showMessage("Please fill all fields!", WARNING_MESSAGE);
    } else {
      Product product = new Product(id, description, price, stock);
      if (productBLL.updateProduct(id, product)) {
        view.showMessage("Product updated successfully!", INFORMATION_MESSAGE);
      } else {
        view.showMessage("Product not updated!", ERROR_MESSAGE);
      }
    }
  }
}

private class DeleteProductButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    String id = view.getIdProduct();
    if (id == null || id.isEmpty()) {
      view.showMessage("Select a product!", WARNING_MESSAGE);
    } else {
      if (productBLL.deleteProduct(Integer.parseInt(id))) {
        view.showMessage("Product deleted successfully!", INFORMATION_MESSAGE);
      } else {
        view.showMessage("Product not deleted!", ERROR_MESSAGE);
      }
    }
  }
}

private class ClearProductButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    view.clearProductData();
  }
}

private class RefreshProductButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    view.setProductsTable(productBLL.findAllProducts());
  }
}

private class AddOrderButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    if (view.isOrderDataEmpty()) {
      view.showMessage("Please fill all fields!", WARNING_MESSAGE);
    } else {
      int id = Integer.parseInt(view.getIdOrder());
      int productId = Integer.parseInt(view.getProductOrderId());
      int quantity = Integer.parseInt(view.getQuantity());
      Order order = new Order(id, quantity, 0, productId);
      try {
        boolean status = orderBLL.createNewOrder(order);
        if (status) {
          view.showMessage("Order added successfully!", INFORMATION_MESSAGE);
        } else {
          view.showMessage("Order not added!", ERROR_MESSAGE);
        }
      } catch (IllegalArgumentException ex) {
        view.showMessage(ex.getMessage(), WARNING_MESSAGE);
      }
    }
  }
}

private class UpdateOrderButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    int id = Integer.parseInt(view.getIdOrder());
    int productId = Integer.parseInt(view.getIdProduct());
    int quantity = Integer.parseInt(view.getQuantity());
    Order order = new Order(id, quantity, 0, productId);
    if (view.isOrderDataEmpty()) {
      view.showMessage("Please fill all fields!", WARNING_MESSAGE);
    } else {
      if (orderBLL.updateOrder(id, order)) {
        view.showMessage("Order updated successfully!", INFORMATION_MESSAGE);
      } else {
        view.showMessage("Order not updated!", ERROR_MESSAGE);
      }
    }
  }
}

private class DeleteOrderButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    String id = view.getIdOrder();
    if (id == null || id.isEmpty()) {
      view.showMessage("Select an order!", WARNING_MESSAGE);
    } else {
      if (orderBLL.deleteOrder(Integer.parseInt(id))) {
        view.showMessage("Order deleted successfully!", INFORMATION_MESSAGE);
      } else {
        view.showMessage("Order not deleted!", ERROR_MESSAGE);
      }
    }
  }
}
```

```
  private class ClearOrderButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      view.clearOrderData();
    }
  }

  private class RefreshOrderButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      view.setOrdersTable(orderBLL.findAllOrders());
    }
  }

  private class AddPurchaseButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      if (view.isPurchaseDataEmpty()) {
        view.showMessage("Please fill all fields!", WARNING_MESSAGE);
      } else {
        int id = Integer.parseInt(view.getIdPurchase());
        int clientId = Integer.parseInt(view.getClientIdPurchase());
        int orderId = Integer.parseInt(view.getOrderIdPurchase());
        PurchaseOrder purchase = new PurchaseOrder(id, LocalDate.now(), clientId, orderId);

        if (purchaseOrderBLL.createPurchaseOrder(purchase)) {
          view.showMessage("Purchase added successfully!", INFORMATION_MESSAGE);
          PurchaseOrder retrieved = purchaseOrderBLL.findPurchaseOrderById(id);
          Client client = clientBLL.findClientById(retrieved.getClient_id());
          Order order = orderBLL.findOrderById(retrieved.getOrder_id());
          Product product = productBLL.findProductById(order.getProduct_id());
          String string = "BILL NO " + id + "\n" + purchase + "\n" + client + "\n" + order + "\n" +
product + "\n\n";
          FileWriter.writeToFile("BILL-" + id + ".txt", string);
        } else {
          view.showMessage("Purchase not added!", ERROR_MESSAGE);
        }

      }
    }
  }
```

```
private class UpdatePurchaseButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    int id = Integer.parseInt(view.getIdPurchase());
    int clientId = Integer.parseInt(view.getClientIdPurchase());
    int orderId = Integer.parseInt(view.getOrderIdPurchase());
    LocalDate date = LocalDate.parse(view.getPurchaseDate());
    PurchaseOrder purchase = new PurchaseOrder(id, date, clientId, orderId);
    if (view.isPurchaseDataEmpty()) {
      view.showMessage("Please fill all fields!", WARNING_MESSAGE);
    } else {
      if (purchaseOrderBLL.updatePurchaseOrder(id, purchase)) {
        view.showMessage("Purchase updated successfully!", INFORMATION_MESSAGE);
      } else {
        view.showMessage("Purchase not updated!", ERROR_MESSAGE);
      }
    }
  }
}

private class DeletePurchaseButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    String id = view.getIdPurchase();
    if (id == null || id.isEmpty()) {
      view.showMessage("Select a purchase!", WARNING_MESSAGE);
    } else {
      if (purchaseOrderBLL.deletePurchaseOrder(Integer.parseInt(id))) {
        view.showMessage("Purchase deleted successfully!", INFORMATION_MESSAGE);
      } else {
        view.showMessage("Purchase not deleted!", ERROR_MESSAGE);
      }
    }
  }
}

private class ClearPurchaseButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    view.clearPurchaseData();
  }
}
```

```
  private class RefreshPurchaseButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      view.setPurchasesTable(purchaseOrderBLL.findAllPurchaseOrders());
    }
  }
}
```

V. Results

Some results are shown here. For example, the bill generated for each order.

5.1. Order 4 BILL

```
BILL NO 4
PurchaseOrder{id=4, purchaseDate=2022-06-20, client_id=1, order_id=2}
Client{id=1, name=Updated Client 1, address=Address 1, phone=0723562365}
Order{id=2, quantity=2.0, totalPrice=24.1, product_id=2}
Product{id=2, description='Pringles Updated', price=12.05, stock=2.0}
```

5.2. Order 5 BILL

```
BILL NO 5
PurchaseOrder{id=5, purchaseDate=2022-06-20, client_id=1, order_id=5}
Client{id=1, name=Updated Client 1, address=Address 1, phone=0723562365}
Order{id=5, quantity=150.0, totalPrice=45.0, product_id=1}
Product{id=1, description='Apples', price=0.3, stock=115.0}
```

5.3. Dump SQL File

```
CREATE DATABASE  IF NOT EXISTS `order_management` /*!40100 DEFAULT
CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci */ /*!80016 DEFAULT
ENCRYPTION='N' */;
USE `order_management`;
-- MySQL dump 10.13  Distrib 8.0.29, for Win64 (x86_64)
--
-- Host: localhost    Database: order_management
-- ------------------------------------------------------
-- Server version	8.0.29

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS
*/;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!50503 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
```

```
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0
*/;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE,
SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `client`
--

DROP TABLE IF EXISTS `client`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `client` (
  `id` int NOT NULL,
  `name` varchar(45) NOT NULL,
  `address` varchar(45) NOT NULL,
  `phone` varchar(45) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `client`
--

LOCK TABLES `client` WRITE;
/*!40000 ALTER TABLE `client` DISABLE KEYS */;
INSERT INTO `client` VALUES (1,'Updated Client 1','Address 1','0723562365'),(2,'Client
2','Address 2','0562365256');
/*!40000 ALTER TABLE `client` ENABLE KEYS */;
UNLOCK TABLES;

--
-- Table structure for table `order`
--

DROP TABLE IF EXISTS `order`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `order` (
  `id` int NOT NULL,
  `quantity` double NOT NULL,
  `totalPrice` double NOT NULL,
  `product_id` int NOT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_product_idx` (`product_id`),
  CONSTRAINT `fk_product` FOREIGN KEY (`product_id`) REFERENCES `product` (`id`)
ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `order`
--

LOCK TABLES `order` WRITE;
/*!40000 ALTER TABLE `order` DISABLE KEYS */;
INSERT INTO `order` VALUES
(1,15,4.5,1),(2,2,24.1,2),(3,15,180.75,2),(4,9,108.45,2),(5,150,45,1);
/*!40000 ALTER TABLE `order` ENABLE KEYS */;
UNLOCK TABLES;

--
-- Table structure for table `product`
--

DROP TABLE IF EXISTS `product`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `product` (
  `id` int NOT NULL,
  `description` varchar(45) NOT NULL,
  `price` double NOT NULL,
  `stock` double NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `product`
--

LOCK TABLES `product` WRITE;
```

```
/*!40000 ALTER TABLE `product` DISABLE KEYS */;
INSERT INTO `product` VALUES (1,'Apples',0.3,115),(2,'Pringles Updated',12.05,2);
/*!40000 ALTER TABLE `product` ENABLE KEYS */;
UNLOCK TABLES;

--
-- Table structure for table `purchaseorder`
--

DROP TABLE IF EXISTS `purchaseorder`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `purchaseorder` (
  `id` int NOT NULL,
  `purchaseDate` date NOT NULL,
  `client_id` int NOT NULL,
  `order_id` int NOT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_client_idx` (`client_id`),
  KEY `fk_order_idx` (`order_id`),
  CONSTRAINT `fk_client` FOREIGN KEY (`client_id`) REFERENCES `client` (`id`) ON
DELETE CASCADE,
  CONSTRAINT `fk_order` FOREIGN KEY (`order_id`) REFERENCES `order` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `purchaseorder`
--

LOCK TABLES `purchaseorder` WRITE;
/*!40000 ALTER TABLE `purchaseorder` DISABLE KEYS */;
INSERT INTO `purchaseorder` VALUES (1,'2022-06-19',1,1),(2,'2022-06-20',2,3),(3,'2022-06-
20',1,4),(4,'2022-06-20',1,2),(5,'2022-06-20',1,5);
/*!40000 ALTER TABLE `purchaseorder` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
```

```
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2022-06-20  2:25:05
```

VI. Conclusions

This assignment was very welcomed to understand how to work with relational database, with reflection techniques and with Javadoc comments. Also it introduced the layered architectural pattern.

Further improvements can be made such as refactoring and better validation of data, although this wasn't the focus of the assignment.

VII. Bibliography

2022. Programming Techniques - Assignment 3 Support Presentation, Cluj-Napoca: UTCN, AC.

Programming Techniques Course, Cluj-Napoca: UTCN