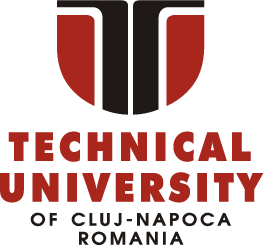
Technical University of Cluj-Napoca

Programming Techniques

Laboratory – Assignment 3

Order Management

**

Teacher: prof. Ioan Salomie

Teacher Assistant: Cristina Bianca Pop

Student: Fleser Mihai

Group: E\_30424

1. Assignment Objective

Consider an application OrderManagement 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.

Secondary Objectives:

* Analyze the problem and identify requirements ………………………...………………. Chapter 2
* Design the Order Management App …..…………………………………………………. Chapter 3
* Implement the Order Management App ……………………………………….………….. Chapter 4
* Test the Order Management App ………………………………………………..……….. Chapter 5

1. Problem analysis, modeling, scenarios, use cases
   1. *Analyzing the problem*

PROBLEM: Design an application intended to process client orders for a warehouse.

Furthermore, the application should be structured in packages using a layered 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

We must use reflection techniques to create a generic class that contains the methods for accessing the DB: create object, edit object, delete object and find object. The queries for accessing the DB for a specific object that corresponds to a table will be generated dynamically through reflection.

Also, we must use reflection techniques to create a method that receives a list of objects and generates the header of the table by extracting through reflection the object properties and then populates the table with the values of the elements from the list.

Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

Java Reflection is quite powerful and can be very useful. For instance, Java Reflection can be used to map properties in JSON files to getter / setter methods in Java objects, like [Jackson, GSON, Boon etc.](http://tutorials.jenkov.com/java-json/index.html) does. Or, Reflection can be used to map the column names of a [JDBC](http://tutorials.jenkov.com/jdbc/index.html) ResultSet to getter / setter methods in a Java object.

This tutorial will get into Java reflection in depth. It will explain the basics of Java Reflection including how to work with arrays, annotations, generics and dynamic proxies, and do dynamic class loading and reloading.

It will also show you how to do more specific Java Reflection tasks, like reading all getter methods of a class, or accessing private fields and methods of a class.

These applications are commonly used to model real world domains. Either for a warehouse or for an online shop, these applications are present in all industries all over the world. It is critical that businesses have these kinds of applications because it is very difficult to manage the orders and the stock without them.

The user must be able to insert, update and delete clients or products and must be able to make a new order using a client and a product. Also, he will be able to receive a bill after an order is successfully done.

1. *Modelling the problem*

The user will be able to select, in the beginning, on which data from the database he will want to work. He will have 3 tabs: the Client tab, the Product Tab and the Order Tab. Depending on what Tab he has opened, he will be able to perform operations only on that specific tables from the database. The tables are connected by foreign keys, especially the Order -> Client and Order -> Product. The user will be able to:

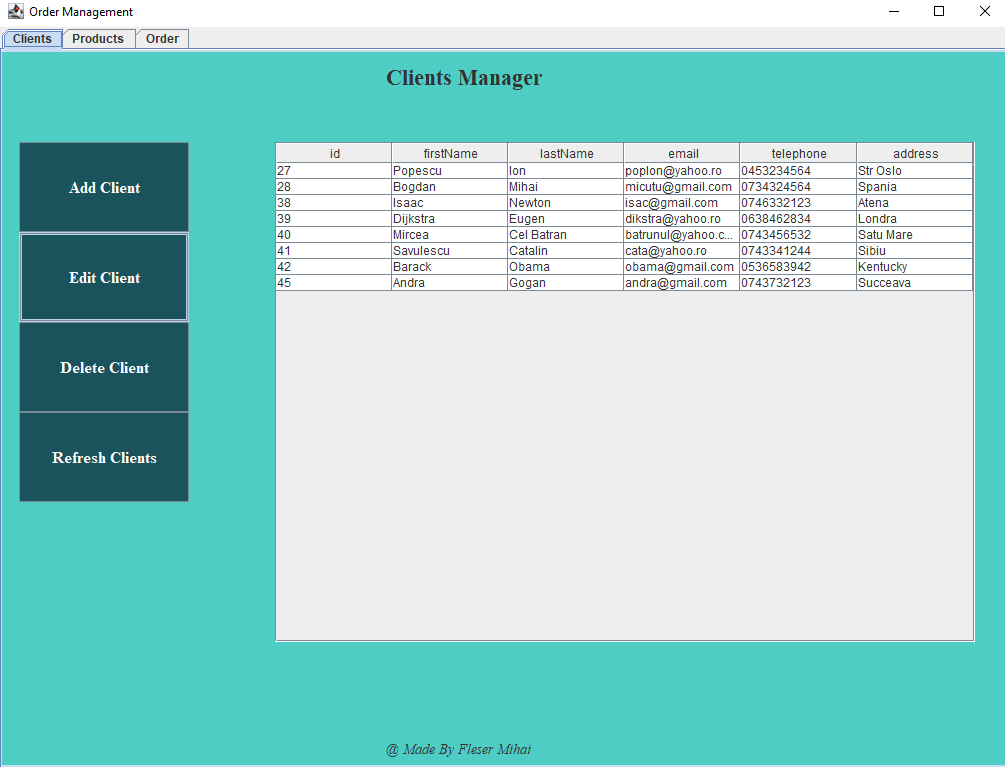
* Insert a new client
* Edit a client
* Delete a client
* Insert a new product
* Edit a product
* Delete a product
* Make a new order
* Receive a bill for that order.

After every operation he does, the database and the GUI will pe updated accordingly.

1. *Different scenarios and use cases*

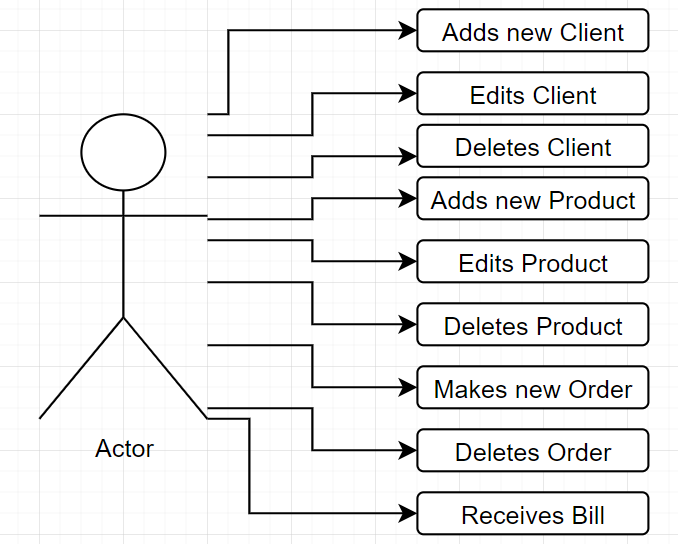
Use cases are a technique for capturing, modelling and specifying the requirements of a system. A use case corresponds to a set of behaviors that the system may perform in interaction with its actors, and which produces an observable result that contribute to its goals. Actors represent the role that human users or other systems have in the interaction.

The use cases are strongly connected with the user steps. I tried to make the user interface as friendly as I could and the result looks like this:



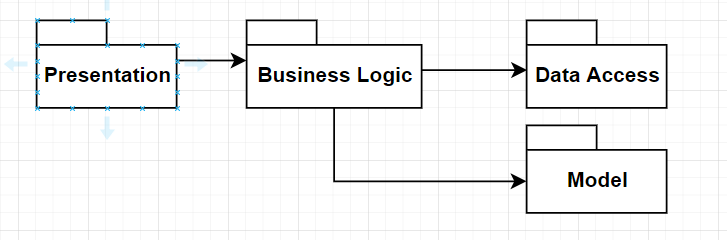
The user will be able to select a tab from the ones at the top of the application. As you see, he can perform several operations on each table. Also, the tables are loaded into the GUI automatically when the application starts. If he chooses to insert something new in the tables, a new form will appear with textfields used to insert new data. The deletion and update are made directly on the table, meaning that if the user wants to edit or delete something, he has to select a row from the table and only after that to press the corresponding buttons. If he wants to edit something, he also needs to insert the new data into the table and only after that to press the edit button.

Use case Diagram:

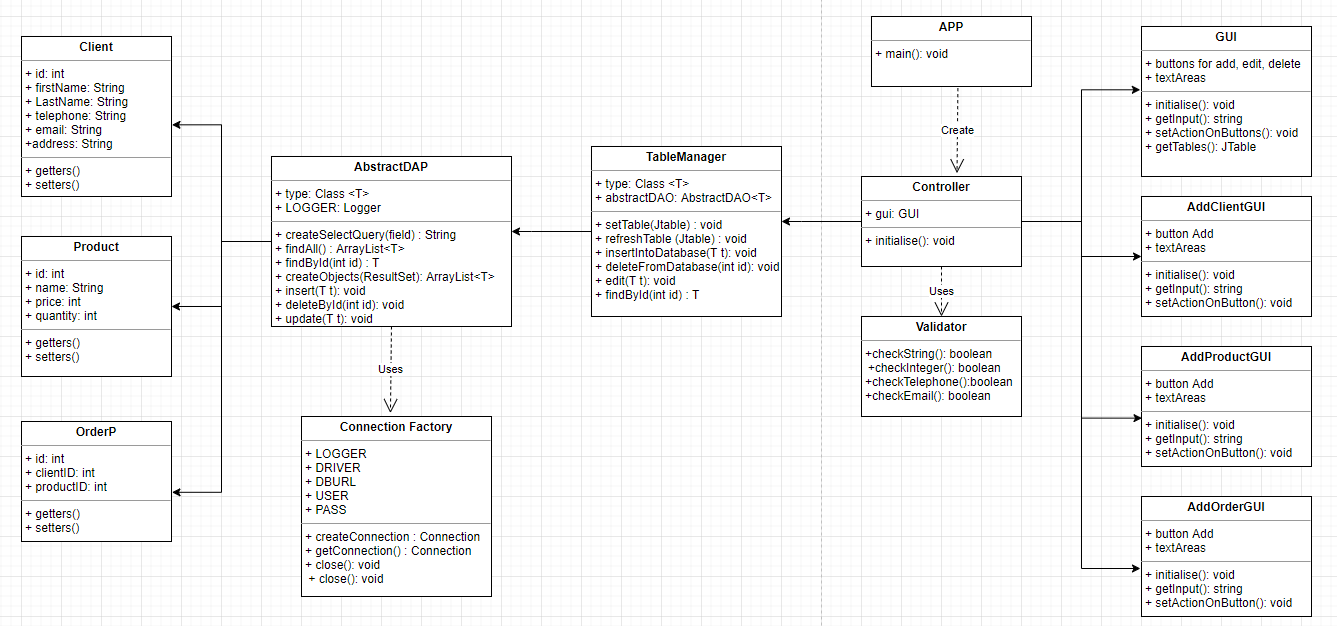


The use case presents the actor, which in our case is the user that interacts with the application. She/ He can make many operations on the tables from the database.

* 1. Design
     + 1. *UML package diagram*



* + - 1. *UML class Diagram*



* + - 1. *Data Structures*

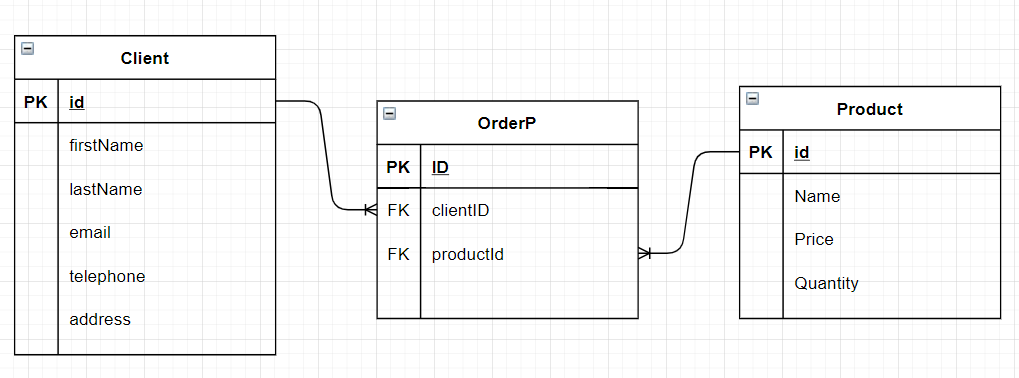
For my project, I have used the following data Structures: ArrayList and String.

I decided to use ArrayList instead of classic arrays because ArrayList does not have a fixed length as normal arrays do, thus making them more memory efficient. Also, I do not have to worry about exceeding the length of the array.

I used String to store the some of input from the user in the model Classes.

* + - 1. *Database Design*

My database consists primary of 3 tables: Client table, Product table and OrderP Table.



* + - 1. *Design Decisions and Packages*

I have done my project using a Layered Arhitecture.

In a standard enterprise application which has a database and graphical UI (web or desktop), there are some typical layers which constitute the application:

* Presentation: Every enterprise application has a UI, in fact graphical UI. This UI can be web or desktop based, which doesn't matter. The rule is simple. UI takes user action and sends it to the controller. And at the end it shows result taken from controller to the user. UI can be implemented according to MVC, MVP, MVVM or another approach.
* BusinessLogic: Handles business logic of the application. Takes info from user and sends it to DB layer (DAO or ORM framework) and vice versa. Abstraction type of controller may vary (a separate control and business layers for example)  according to the application parameters or development patterns (MVC, MVP, MVVM, ...) but the main idea remains the same.
* Data Access: 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:
* Model Layer: Contains classes that model the tables within the database. They usually have only getters and setters.
  + - 1. *Class Design*

As I said earlier, my program consists of 4 packages, following the Layered Architecture design

The Model Package:

* Client Class – models the client from the database, it has only getters and setters
* Product Class – models the product from the database, it has only getters and setters
* OrderP Class – models the orderP from the database, it has only getters and setters

The DataAcess Package:

* AbstractDAO Class – performs operations on the tables of the database using reflections techniques. Such operations include inserting, deleting, finding or updating the tables.
* ConnectionFactory Class – ensures the connection with the mySQL database

The BusinessLogic Package:

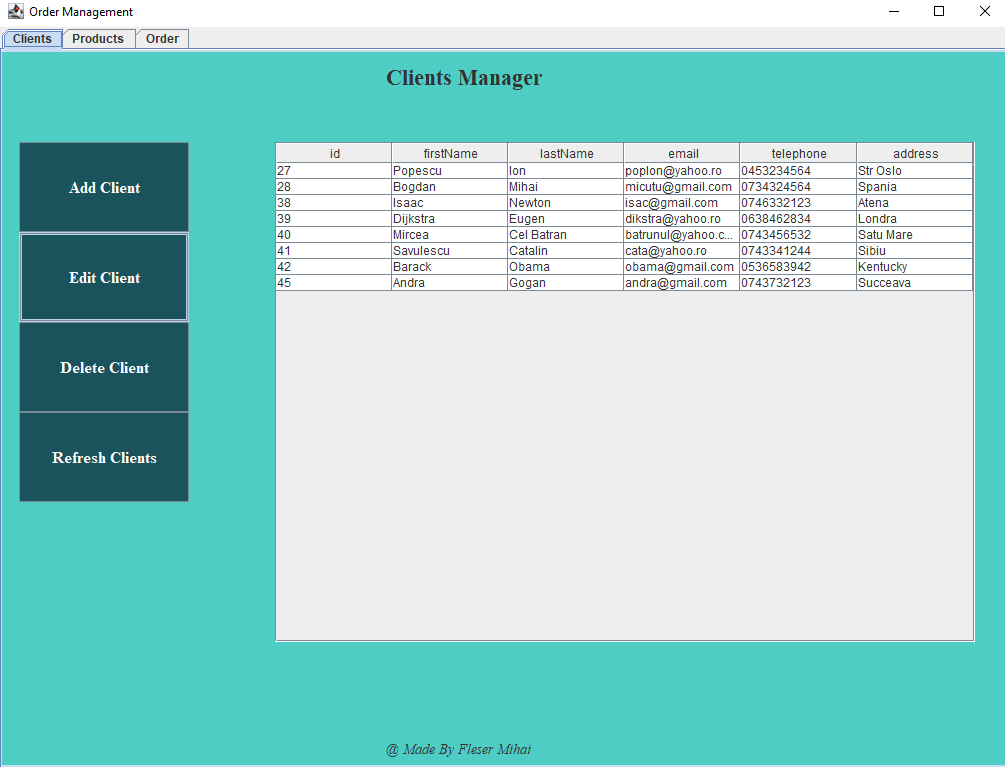
* Validator Class – checks if the input from the user is or is not valid. It can perform checks on the email, on the telephone, on some input Strings or on some input Integers.
* TableManager Class – takes a Jtable and builds its header through reflection and then introduces the data from the database in this Jtable

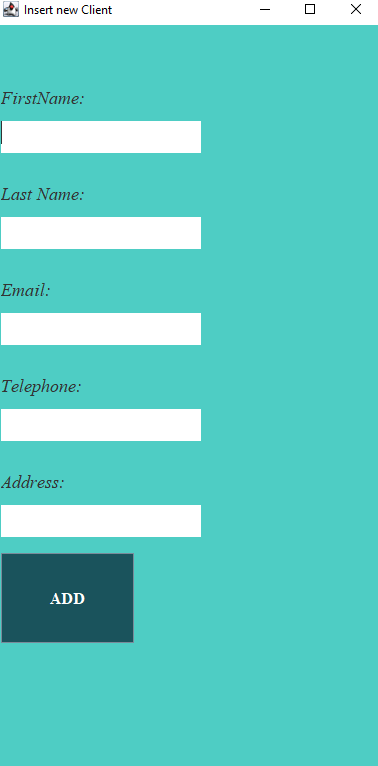
The Presentation Package:

* Controller Class – This class communicates between the GUIs and Bussiness Logic. It handles the events that happens on the GUI. It takes the data introduced by the user and transfers it further to the BussinessLogic to manage it.
* GUI class - The main GUI of the application. It is used to control all its functions and perform all the application's operations. The user has 3 tabs: The client tab, the product tab and the order tab. He can insert, delete, edit and view the clients and the products from the database and he can make a new order and see all orders that have been made.
* AddClientGUI – it’s a GUI used to insert a new Client in the Database.
* AddProductGUI – it’s a GUI used to insert a new Product in the Database.
* AddOrderGUI – it’s a GUI used to insert a new Order in the Database.

App class – the entry point of the application, just creates a new Controller.

* + - 1. *User Interface*





* 1. Implementation

public class TableManager<T> {

private final Class<T> type;

private AbstractDAO<T> abstractDAO;

public TableManager(Class<T> type) {

this.type = type;

this.abstractDAO = new AbstractDAO<T>(type);

}

/\*\*

\* Receives from the database an arraylist of objects and sets up a table.

\* It generates the header of the table by extracting through reflection the object properties and

\* then populates the table with the values of the elements from the list.

\* @param table - the table to be set up

\*/

public void setTable(JTable table)

{

DefaultTableModel model = (DefaultTableModel) table.getModel();

ArrayList<T> elements = abstractDAO.findAll();

for (Field field : type.getDeclaredFields())

{

model.addColumn(field.getName());

}

for(T element: elements)

{

LinkedList<Object> data = new LinkedList<>();

for(Field field: type.getDeclaredFields())

{

field.setAccessible(true);

try {

Object value = field.get(element);

data.add(value.toString());

} catch (IllegalAccessException e) {

e.printStackTrace();

}

}

model.addRow(data.toArray());

}

}

/\*\*

\* Refreshes an already generated table with the elements from the database.

\* @param table

\*/

public void refreshTable(JTable table)

{

DefaultTableModel model = (DefaultTableModel) table.getModel();

ArrayList<T> elements = abstractDAO.findAll();

model.setRowCount(0);

for(T element: elements)

{

LinkedList<Object> data = new LinkedList<>();

for(Field field: type.getDeclaredFields())

{

field.setAccessible(true);

try {

Object value = field.get(element);

data.add(value.toString());

} catch (IllegalAccessException e) {

e.printStackTrace();

}

}

model.addRow(data.toArray());

}

}

/\*\*

\* Inserts into the database a new element by calling the AbstractDAO method for inserting.

\* @param t

\*/

public void insertIntoDatabase(T t)

{

abstractDAO.insert(t);

}

/\*\*

\* Deletes from the databases using the id.

\* @param id

\*/

public void deleteFromDatabase(int id)

{

abstractDAO.deleteById(id);

}

/\*\*

\* Edits an element from the database.

\* @param t

\*/

public void edit(T t)

{

abstractDAO.update(t);

}

/\*\*

\* Finds by id an element from a table which is in the database.

\* @param id

\* @return the element searched for.

\*/

public T findById(int id)

{

return abstractDAO.findById(id);

}

}

public class Validator {

/\*\*

\* Checks if an input String is correct.

\* @param s

\* @return

\*/

public static boolean checkString(String s)

{

if(s.isEmpty())

return false;

return true;

}

/\*\*

\* Checks if an input Integer is correct.

\* @param input

\* @return

\*/

public static boolean checkInteger(int input)

{

if(input < 0)

return false;

return true;

}

/\*\*

\* Checks if the phone number is correct.

\* @param s

\* @return

\*/

public static boolean checkTelephoneNumber(String s)

{

if(s.length() != 10)

return false;

if(!s.matches("[0-9]+"))

return false;

return true;

}

/\*\*

\* Checks if the email is correct.

\* @param s

\* @return

\*/

public static boolean checkEmail(String s)

{

String EMAIL\_PATTERN = "(?:(?:\\r\\n)?[ \\t])\*(?:(?:(?:[^()<>@,;:\\\\\".\\[\\] \\000-\\031]+(?:(?:(?:\\r\\n)?[

if(!s.matches(EMAIL\_PATTERN))

return false;

return true;

}

}

package org.example.DataAccess;

import java.beans.IntrospectionException;

import java.beans.PropertyDescriptor;

import java.lang.reflect.Field;

import java.lang.reflect.InvocationTargetException;

import java.lang.reflect.Method;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.ArrayList;

import java.util.logging.Level;

import java.util.logging.Logger;

/\*\*

\* @author Mihai Fleser

\* This class uses reflection tehniques to generate specific queries for accessing the DB.

\* @param <T>

\*/

public class AbstractDAO<T> {

protected static final Logger LOGGER = Logger.getLogger(AbstractDAO.class.getName());

private final Class<T> type;

@SuppressWarnings("unchecked")

public AbstractDAO(Class<T> type) {

this.type = type;

}

/\*\*

\* Creates a basic select query depending on the field given.

\* @param field - the parameter used in the Where clause

\* @return - returns the query generated

\*/

private String createSelectQuery(String field) {

StringBuilder sb = new StringBuilder();

sb.append("SELECT ");

sb.append(" \* ");

sb.append(" FROM ");

sb.append(type.getSimpleName());

sb.append(" WHERE " + field + " =?");

return sb.toString();

}

/\*\*

\* Selects all the rows from a table.

\* @return - the arrayList of objects created from the rows returned by the query

\*/

public ArrayList<T> findAll() {

Connection connection = null;

PreparedStatement statement = null;

ResultSet resultSet = null;

String query = "Select \* from " + type.getSimpleName();

try {

connection = ConnectionFactory.getConnection();

statement = connection.prepareStatement(query);

resultSet = statement.executeQuery();

return createObjects(resultSet);

} catch (SQLException e) {

LOGGER.log(Level.WARNING, type.getName() + "DAO:findAll " + e.getMessage());

} finally {

ConnectionFactory.close(resultSet);

ConnectionFactory.close(statement);

ConnectionFactory.close(connection);

}

return null;

}

/\*\*

\* Finds by id a row in a table from the database.

\* @param id

\* @return the object made with that returned row

\*/

public T findById(int id) {

Connection connection = null;

PreparedStatement statement = null;

ResultSet resultSet = null;

String query = createSelectQuery("id");

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;

}

/\*\*

\* Makes an arrayList of objects based on a resultSet from a query.

\* @param resultSet

\* @return the ArrayList generated

\*/

private ArrayList<T> createObjects(ResultSet resultSet) {

ArrayList<T> list = new ArrayList<T>();

try {

while (resultSet.next()) {

T instance = type.newInstance();

for (Field field : type.getDeclaredFields()) {

Object value = resultSet.getObject(field.getName());

PropertyDescriptor propertyDescriptor = new PropertyDescriptor(field.getName(), type);

Method method = propertyDescriptor.getWriteMethod();

method.invoke(instance, value);

}

list.add(instance);

}

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (SecurityException e) {

e.printStackTrace();

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (InvocationTargetException e) {

e.printStackTrace();

} catch (SQLException e) {

e.printStackTrace();

} catch (IntrospectionException e) {

e.printStackTrace();

}

if(list.size() == 0)

list.add(null);

return list;

}

/\*\*

\* Inserts into a table from the database a new object.

\* @param t

\*/

public void insert(T t) {

Connection connection = null;

PreparedStatement statement = null;

ResultSet resultSet = null;

String query = "Insert into " + type.getSimpleName() + "(";

for(Field field: t.getClass().getDeclaredFields())

{

field.setAccessible(true);

try {

Object value = field.get(t);

if(value != null)

{

query = query + field.getName() + ", ";

}

} catch (IllegalAccessException e) {

e.printStackTrace();

}

}

query = query.substring(0,query.length() - 2);

query = query + ") VALUES (";

for (Field field : t.getClass().getDeclaredFields())

{

field.setAccessible(true);

try {

Object value = field.get(t);

if(value != null)

query = query + "'" + value + "', ";

} catch (IllegalAccessException e) {

e.printStackTrace();

}

}

query = query.substring(0,query.length() - 2);

query = query + ")";

System.out.println(query);

try {

connection = ConnectionFactory.getConnection();

statement = connection.prepareStatement(query);

statement.execute();

} catch (SQLException e) {

LOGGER.log(Level.WARNING, type.getName() + "DAO:insert " + e.getMessage());

} finally {

ConnectionFactory.close(resultSet);

ConnectionFactory.close(statement);

ConnectionFactory.close(connection);

}

}

/\*\*

\* Deletes an element from a table depending on its id.

\* @param id

\*/

public void deleteById(int id)

{

Connection connection = null;

PreparedStatement statement = null;

ResultSet resultSet = null;

String query = "DELETE FROM " + type.getSimpleName() + " Where id = '" + id + "'";

System.out.println(query);

try {

connection = ConnectionFactory.getConnection();

statement = connection.prepareStatement(query);

statement.execute();

} catch (SQLException e) {

LOGGER.log(Level.WARNING, type.getName() + "DAO:deleteById " + e.getMessage());

} finally {

ConnectionFactory.close(resultSet);

ConnectionFactory.close(statement);

ConnectionFactory.close(connection);

}

}

/\*\*

\* Updates an element from a table in the database.

\* @param t

\*/

public void update(T t) {

Connection connection = null;

PreparedStatement statement = null;

ResultSet resultSet = null;

String query = "UPDATE " + type.getSimpleName() + " SET ";

Object id = null;

boolean first = false;

for(Field field: t.getClass().getDeclaredFields())

{

field.setAccessible(true);

try {

Object value = field.get(t);

if(!first)

{

first = true;

id = value;

}

query = query + field.getName() + " ='" + value + "', ";

} catch (IllegalAccessException e) {

e.printStackTrace();

}

}

query = query.substring(0,query.length() - 2);

query = query + " WHERE id = " + id;

System.out.println(query);

try {

connection = ConnectionFactory.getConnection();

statement = connection.prepareStatement(query);

statement.execute();

} catch (SQLException e) {

LOGGER.log(Level.WARNING, type.getName() + "DAO:update " + e.getMessage());

} finally {

ConnectionFactory.close(resultSet);

ConnectionFactory.close(statement);

ConnectionFactory.close(connection);

}

}

}

package org.example.DataAccess;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.logging.Level;

import java.util.logging.Logger;

/\*\* @author Mihai Fleser

\* Connects to the database from the MySql server.

\*/

public class ConnectionFactory {

private static final Logger LOGGER = Logger.getLogger(ConnectionFactory.class.getName());

private static final String DRIVER = "com.mysql.cj.jdbc.Driver";

private static final String DBURL = "jdbc:mysql://localhost:3306/ordermanagement";

private static final String USER = "root";

private static final String PASS = "root";

private static ConnectionFactory singleInstance = new ConnectionFactory();

private ConnectionFactory() {

try {

Class.forName(DRIVER);

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

private Connection createConnection() {

Connection connection = null;

try {

connection = DriverManager.getConnection(DBURL, USER, PASS);

} catch (SQLException e) {

LOGGER.log(Level.WARNING, "An error occured while trying to connect to the database");

e.printStackTrace();

}

return connection;

}

public static Connection getConnection() {

return singleInstance.createConnection();

}

public static void close(Connection connection) {

if (connection != null) {

try {

connection.close();

} catch (SQLException e) {

LOGGER.log(Level.WARNING, "An error occured while trying to close the connection");

}

}

}

public static void close(Statement statement) {

if (statement != null) {

try {

statement.close();

} catch (SQLException e) {

LOGGER.log(Level.WARNING, "An error occured while trying to close the statement");

}

}

}

public static void close(ResultSet resultSet) {

if (resultSet != null) {

try {

resultSet.close();

} catch (SQLException e) {

LOGGER.log(Level.WARNING, "An error occured while trying to close the ResultSet");

}

}

}

}

public class Client {

private Integer id;

private String firstName;

private String lastName;

private String email;

private String telephone;

private String address;

public Client()

{

}

public Client(Integer id, String firstName, String lastName, String email, String telephone, String address)

{

this.id = id;

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

this.telephone = telephone;

this.address = address;

}

public Client(String firstName, String lastName, String email, String telephone, String address)

{

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

this.telephone = telephone;

this.address = address;

}

public class OrderP {

private Integer id;

private Integer clientId;

private Integer productId;

private Integer quantity;

public OrderP()

{

}

public OrderP(Integer id, Integer clientId, Integer productId, Integer quantity)

{

this.id = id;

this.clientId = clientId;

this.productId = productId;

this.quantity = quantity;

}

public OrderP(Integer clientId, Integer productId, Integer quantity)

{

this.clientId = clientId;

this.productId = productId;

this.quantity = quantity;

}

public class Product {

private Integer id;

private String name;

private Integer price;

private Integer quantity;

public Product()

{

}

public Product(Integer id, String name, Integer price, Integer quantity)

{

this.id = id;

this.name = name;

this.price = price;

this.quantity = quantity;

}

public Product(String name, Integer price, Integer quantity)

{

this.name = name;

this.price = price;

this.quantity = quantity;

}

public class AddClientGUI {

private JFrame frame = new JFrame("Insert new Client");

private JTextArea firstName = new JTextArea();

private JTextArea lastName = new JTextArea();

private JTextArea email = new JTextArea();

private JTextArea telephone = new JTextArea();

private JTextArea address = new JTextArea();

private JButton add = new JButton("ADD");

public void initialise()

{}

public class AddOrderGUI {

private JFrame frame = new JFrame("Add new Order");

private JTextArea clientId = new JTextArea();

private JTextArea productId = new JTextArea();

private JTextArea quantity = new JTextArea();

private JButton add = new JButton("ADD");

public AddOrderGUI()

{

}

public void initialise()

{}

public class AddProductGUI {

private JFrame frame = new JFrame("Insert new Product");

private JTextArea name = new JTextArea();

private JTextArea price = new JTextArea();

private JTextArea quantity = new JTextArea();

private JButton add = new JButton("ADD");

public AddProductGUI()

{

}

public void initialise()

{}

public class Controller {

private GUI gui;

private AddClientGUI addClientGUI;

private AddProductGUI addProductGUI;

private AddOrderGUI addOrderGUI;

private TableManager<Client> clientTableManager = new TableManager(Client.class);

private TableManager<Product> productTableManager = new TableManager(Product.class);

private TableManager<OrderP> orderTableManager = new TableManager(OrderP.class);

public Controller()

{

this.gui = new GUI();

this.addClientGUI = new AddClientGUI();

this.addProductGUI = new AddProductGUI();

this.addOrderGUI = new AddOrderGUI();

}

/\*\*

\* Takes the input from the addClientGUI and commands the clientManager to insert the new created client in the database.

\*/

private void addClient()

{

try {

String firstName = addClientGUI.getFirstName();

String lastName = addClientGUI.getLastName();

String email = addClientGUI.getEmail();

String telephone = addClientGUI.getTelephone();

String address = addClientGUI.getAdrress();

if(!Validator.checkString(firstName) || !Validator.checkString(lastName) || !Validator.checkString(email) || !Validator.checkString(telephone) || !Validator.checkString(address))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: You must complete all fields!");

return;

}

if(!Validator.checkTelephoneNumber(telephone))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong telephone!");

return;

}

if(!Validator.checkEmail(email))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong email!");

return;

}

Client client = new Client(firstName, lastName, email, telephone, address);

clientTableManager.insertIntoDatabase(client);

}catch (Exception exception)

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong Input");

}

}

/\*\*

\* Generates a new Bill with the client and the product he ordered.

\* @param client

\* @param product

\* @param quantity

\*/

private void generateBill(Client client, Product product, int quantity)

{

DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");

LocalDateTime now = LocalDateTime.now();

String dateAndTime = dtf.format(now);

dateAndTime = dateAndTime.replace(' ', '\_');

dateAndTime = dateAndTime.replace(':', '/');

dateAndTime = dateAndTime.replace('/', '\_');

String pdfName = "BILL\_" + dateAndTime +".pdf";

Document document = new Document();

try {

PdfWriter.getInstance(document, new FileOutputStream(pdfName));

} catch (DocumentException e) {

e.printStackTrace();

} catch (FileNotFoundException e) {

e.printStackTrace();

}

document.open();

Paragraph paragraph = new Paragraph("Client " + client.getFirstName() + " " + client.getLastName() + " has ordered:\n " +

product.getName() + " in quantity of " + quantity + " for a total: \n" +

product.getPrice() + " x " + quantity +" = "+product.getPrice() \* quantity);

try {

document.add(paragraph);

} catch (DocumentException e) {

e.printStackTrace();

}

document.close();

}

/\*\*

\* Takes the input from the addOrderGUI and commands the orderManager to insert the new created order in the database.

\*/

private void addOrder()

{

try {

int clientId = Integer.valueOf(addOrderGUI.getClientId());

int productId = Integer.valueOf(addOrderGUI.getProductId());

int quantity = Integer.valueOf(addOrderGUI.getQuantity());

Client client = clientTableManager.findById(clientId);

if (client == null) {

JOptionPane.showMessageDialog(new JFrame(), "Can't find this client in the database.");

return;

}

if(!Validator.checkInteger(quantity))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong Quantity");

return;

}

Product product = productTableManager.findById(productId);

if (product == null) {

JOptionPane.showMessageDialog(new JFrame(), "Can't find this product in the database.");

return;

}

if (product.getQuantity() < quantity) {

JOptionPane.showMessageDialog(new JFrame(), "Not Enough Products in stock.");

return;

}

orderTableManager.insertIntoDatabase(new OrderP(clientId, productId, quantity));

product.setQuantity(product.getQuantity() - quantity);

productTableManager.edit(product);

generateBill(client,product,quantity);

}catch (Exception exception)

{

exception.printStackTrace();

}

}

/\*\*

\* Deletes the selected order from the database.

\*/

private void deleteOrder()

{

try {

int row = gui.getOrdersTable().getSelectedRow();

int col = 0;

String value = gui.getOrdersTable().getModel().getValueAt(row, col).toString();

int id = Integer.valueOf(value);

orderTableManager.deleteFromDatabase(id);

}catch (Exception exception)

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: You Must select something to delete");

}

}

/\*\*

\* Deletes the selected client from the database.

\*/

private void deleteClient()

{

try {

int row = gui.getClientsTable().getSelectedRow();

int col = 0;

String value = gui.getClientsTable().getModel().getValueAt(row, col).toString();

int id = Integer.valueOf(value);

clientTableManager.deleteFromDatabase(id);

}catch (Exception exception)

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: You Must select something to delete");

}

}

/\*\*

\* Deletes the selected product from the database.

\*/

private void deleteProduct()

{

try {

int row = gui.getProductsTable().getSelectedRow();

int col = 0;

String value = gui.getProductsTable().getModel().getValueAt(row, col).toString();

int id = Integer.valueOf(value);

productTableManager.deleteFromDatabase(id);

}catch (Exception exception)

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: You Must select something to delete");

}

}

/\*\*

\* Edits the selected client with the new values inserted by the user.

\*/

private void editClient()

{

try {

int row = gui.getClientsTable().getSelectedRow();

int col = 0;

String value = gui.getClientsTable().getModel().getValueAt(row, col).toString();

int id = Integer.valueOf(value);

value = gui.getClientsTable().getModel().getValueAt(row, col + 1).toString();

String firstName = value;

value = gui.getClientsTable().getModel().getValueAt(row, col + 2).toString();

String lastName = value;

value = gui.getClientsTable().getModel().getValueAt(row, col + 3).toString();

String email = value;

value = gui.getClientsTable().getModel().getValueAt(row, col + 4).toString();

String telephone = value;

value = gui.getClientsTable().getModel().getValueAt(row, col + 5).toString();

String address = value;

if(!Validator.checkString(firstName) || !Validator.checkString(lastName) || !Validator.checkString(email) || !Validator.checkString(telephone) || !Validator.checkString(address))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: You must complete all fields!");

return;

}

if(!Validator.checkTelephoneNumber(telephone))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong telephone!");

return;

}

if(!Validator.checkEmail(email))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong email!");

return;

}

Client c = new Client(id, firstName, lastName, email, telephone, address);

clientTableManager.edit(c);

}catch (Exception exception)

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: You Must select something to edit and you must insert valid data");

}

}

/\*\*

\* Edits the selected product with the new values inserted by the user.

\*/

private void editProduct()

{

try {

int row = gui.getProductsTable().getSelectedRow();

int col = 0;

String value = gui.getProductsTable().getModel().getValueAt(row, col).toString();

int id = Integer.valueOf(value);

value = gui.getProductsTable().getModel().getValueAt(row, col + 1).toString();

String firstName = value;

value = gui.getProductsTable().getModel().getValueAt(row, col + 2).toString();

int price = Integer.valueOf(value);

value = gui.getProductsTable().getModel().getValueAt(row, col + 3).toString();

int quantity = Integer.valueOf(value);

if(!Validator.checkString(firstName))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: You must give a name to the new product");

return;

}

if(!Validator.checkInteger(quantity))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong Quantity");

return;

}

if(!Validator.checkInteger(price))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong Price");

return;

}

Product p = new Product(id, firstName, price, quantity);

productTableManager.edit(p);

}catch (Exception exception)

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: You Must select something to edit and you must insert valid data");

}

}

/\*\*

\* Takes the input from the addProductGUI and commands the productManager to insert the new created product in the database.

\*/

private void addProduct()

{

try {

if(!Validator.checkString(addProductGUI.getName()))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: You must give a name to the new product");

return;

}

if(!Validator.checkInteger(Integer.valueOf(addProductGUI.getQuantity())))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong Quantity");

return;

}

if(!Validator.checkInteger(Integer.valueOf(addProductGUI.getPrice())))

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong Price");

return;

}

Product product = new Product(addProductGUI.getName(), Integer.valueOf(addProductGUI.getPrice()), Integer.valueOf(addProductGUI.getQuantity()));

productTableManager.insertIntoDatabase(product);

}catch (Exception exception)

{

JOptionPane.showMessageDialog(new JFrame(), "ERROR: Wrong Input");

}

}

/\*\*

\* Sets up the actionListeners on the GUIs.

\*/

public void run()

public class GUI {

private JFrame frame = new JFrame("Car Parking");

private JLayeredPane lpane = new JLayeredPane();

private JButton refreshClients=new JButton("Refresh Clients");

private JButton addClient=new JButton("Add Client");

private JButton editClient=new JButton("Edit Client");

private JButton deleteClient=new JButton("Delete Client");

private JButton refreshProducts=new JButton("Refresh Products");

private JButton addProduct=new JButton("Add Product");

private JButton editProduct=new JButton("Edit Product");

private JButton deleteProduct=new JButton("Delete Product");

private JButton deleteOrder=new JButton("Delete Order");

private JButton addOrder=new JButton("Add Order");

private JTable clientsTable = new JTable(new DefaultTableModel(){@Override

public boolean isCellEditable(int row, int col) {

return col != 0;

}});

private JTable productsTable = new JTable(new DefaultTableModel(){@Override

public boolean isCellEditable(int row, int col) {

return col != 0;

}});

public class App

{

public static void main( String[] args )

{

Controller controller = new Controller();

controller.run();

}

}

* 1. Results

I tested this project by introducing, editing or deleting clients, products or orders. If the input is incorrect, the user will receive a message informing what is the problem and the database will not pe updated. I have included my database in the project in order to see how the application performed in my case.

* 1. Conclusions

I liked this Project because it helped me to understand better how to work with generics and reflection and further develop my OOP knowledge. I have learned, again, that time management is quite important to not be stressed out and do the project in time. I also liked the fact that I have learned how to manage a database and perform all kinds of operations on it.

Future improvements:

* One idea will be to make more tables, as in a warehouse there are a lot of different products on different categories.
* Another one will be to make the GUI more user friendly in order to be more easily to work on it.
  1. Bibliography
* Object-Oriented Programming - Lecture Slides of prof. Marius JOLDOS
* Programming Techniques – Lectures of prof. Ioan SALOMIE
* <https://docs.oracle.com/javase/tutorial/uiswing/>
* <https://docs.oracle.com/javafx/2/get_started/jfxpub-get_started.htm>
* <https://www.baeldung.com/javafx>
* <https://www.vogella.com/tutorials/JUnit/article.html>
* <https://stackoverflow.com/>
* <http://tutorials.jenkov.com/java-concurrency/>
* <https://dzone.com/articles/layers-standard-enterprise>
* <http://tutorials.jenkov.com/java-reflection/index.html>
* <https://www.baeldung.com/java-pdf-creation>
* <https://www.baeldung.com/javadoc>
* <http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>