**Food delivery**

**Documentation**

**Csatlos-Koncz Andrei**

**Group 30422**

**1.Assignment objective**

The main objective for this assignment is to design and implement a food delivery management system for a catering company. The client can order products from the company’s menu. The system should have three types of users that log in using a username and a password: administrator, regular employee, and client

The sub-objectives are:

* Analyze the problem and identify requirements
* Design the food delivery
* Implement the food delivery
* Test the food delivery
* Classes with maximum 300 lines (except the UI classes)
* Java naming conventions
* Object-oriented programming design
* Lists instead of arrays
* Foreach instead of simple for
* Graphical User Interface
* JUnit for testing

**2. Problem analysis, modeling, scenarios, use cases**

The modeling of the problem was largely done according to the example presented. Each table is assigned a class that has the table columns as attributes. The modeling was done in layers, as requested, so that there are 4 levels of execution, each being represented by a package:

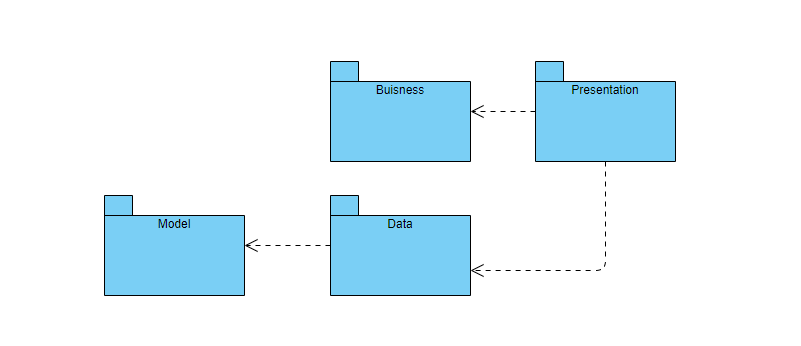
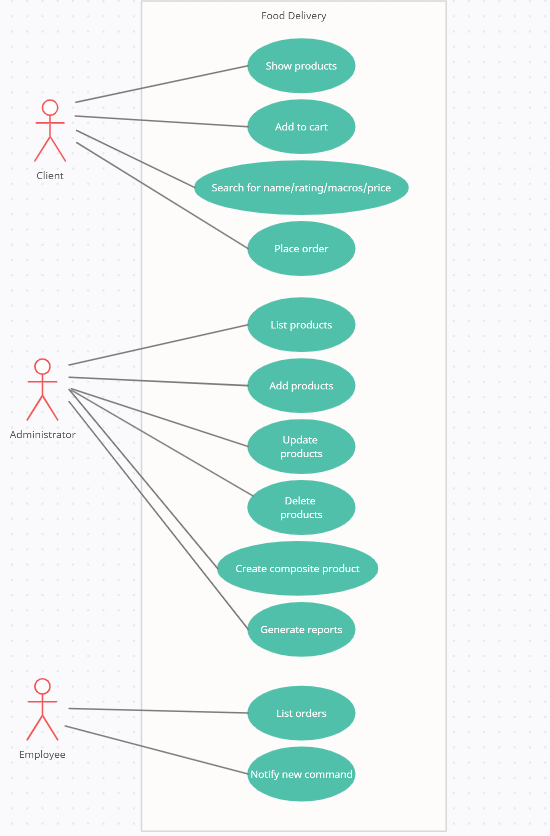
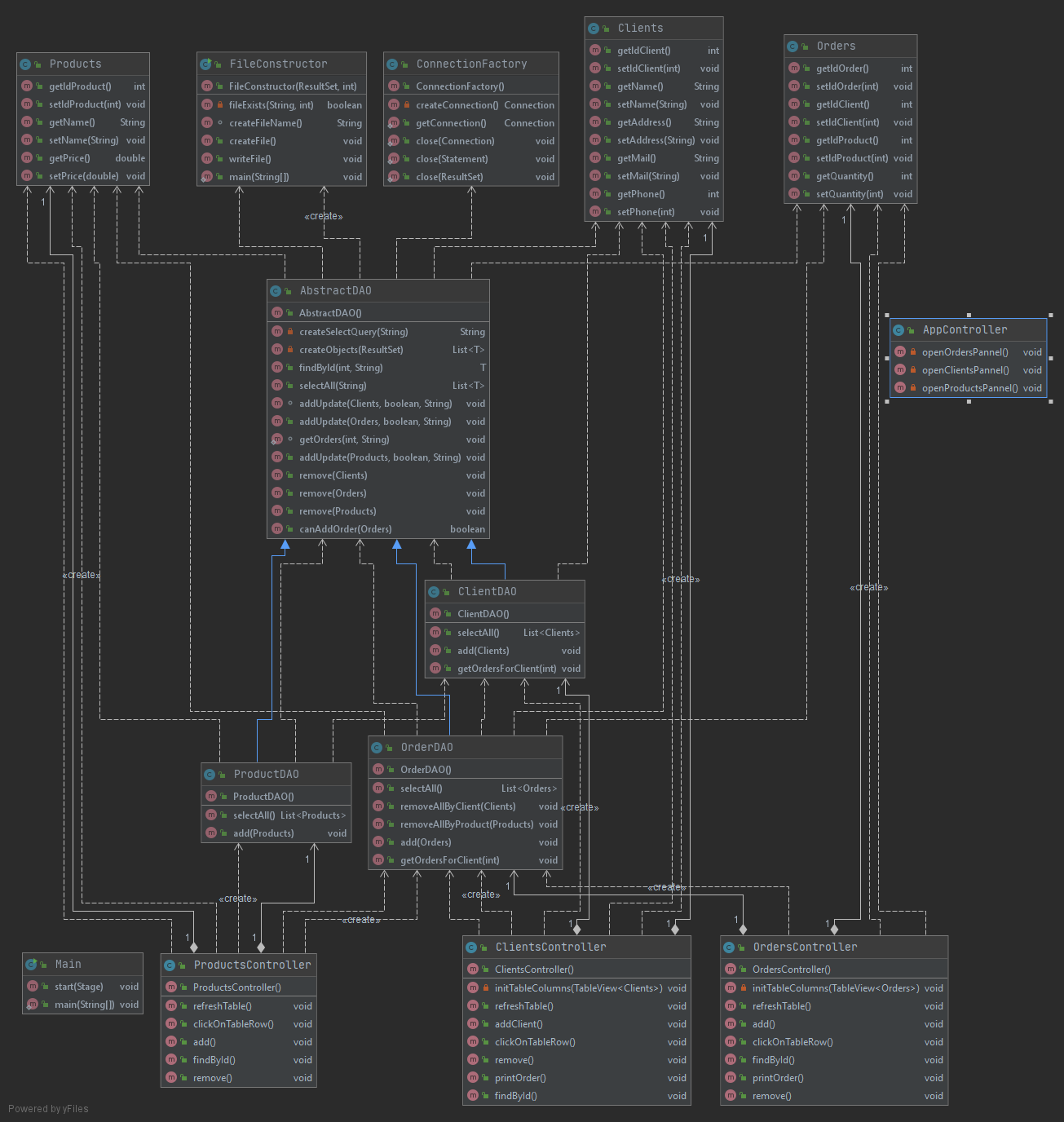
1. Business (contains the classes: Clients, Products, Order), which is responsible for the concrete definition of the actions for each class.
2. Data (contains the DatabaseConnection class), where we make the connection with a local database in which we store users
3. Presentation, which contains the Controller classes (which deals with parsing the input file and extracting the input data and uses the methods from the classes related to the desired action in BLL) and app (which is the main project, in which the constructor is called for Controller , which then receives the name of the input file from which to extract the data).

**3.Design (design decisions, UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interfaces)**

Unified Modeling Language or UML for short is a standard language for describing models and software specifications. UML was basically developed to represent the complexity of object-oriented programs, whose foundation is the structuring of programs by classes, and their instances (also called objects). However, due to its efficiency and clarity in representing abstract elements, UML is used beyond the IT domain. This is how there are UML applications for project management, business process design, etc.

To carry out this project we used 15 classes:

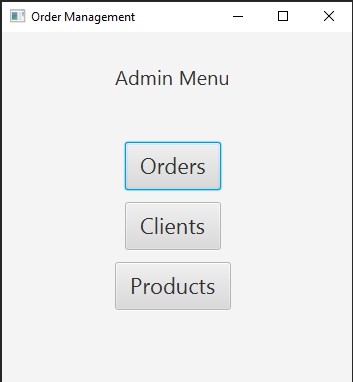
* Clients, Products and Orders which are associated with the tables from the database.
* AbstractDAO is the class given in the homework support presentation. This class defines the common operations for accessing a table: Insert, Update, Delete, FindById, FindAll. Define the operations on the specified generic type <T>
* ClientDAO, OrderDAO, ProductDAO are used to make the link between tables and code, queries, this classes have specific queries for each table they represent.
* Connection Factory makes the connection to the database thus we can manipulate it.
* File Constructor is the class with who we creates the files and print in them the orders using filewriters.
* AppController is the main controller for the app.fxml file. This is the main windows in which the program opens and from which we can select the next windows: Clients window, Orders windows, Products window.
* ClientController, OrdersController, ProductsController are the controller for each correspondent fxml file. In those we control how the application reacts to our user input.
* UML Diagram:



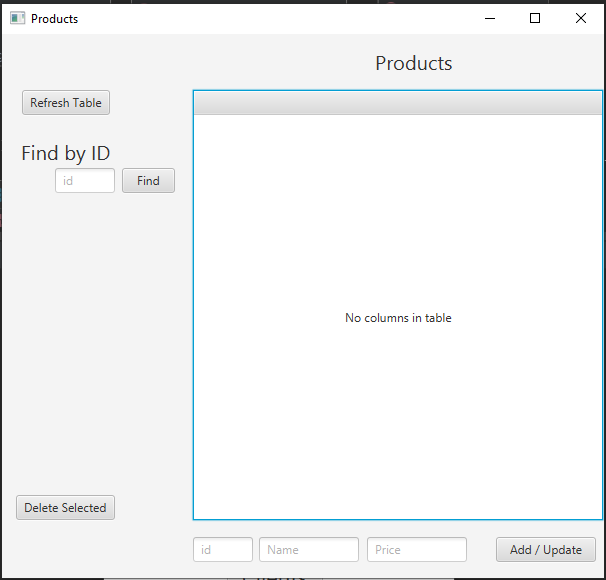
Graphical User Interface:

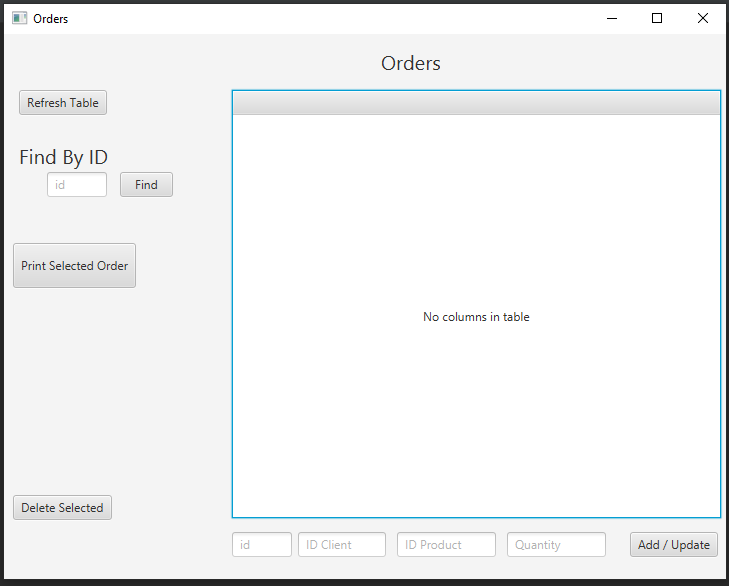
For the graphical user interface we have several windows:

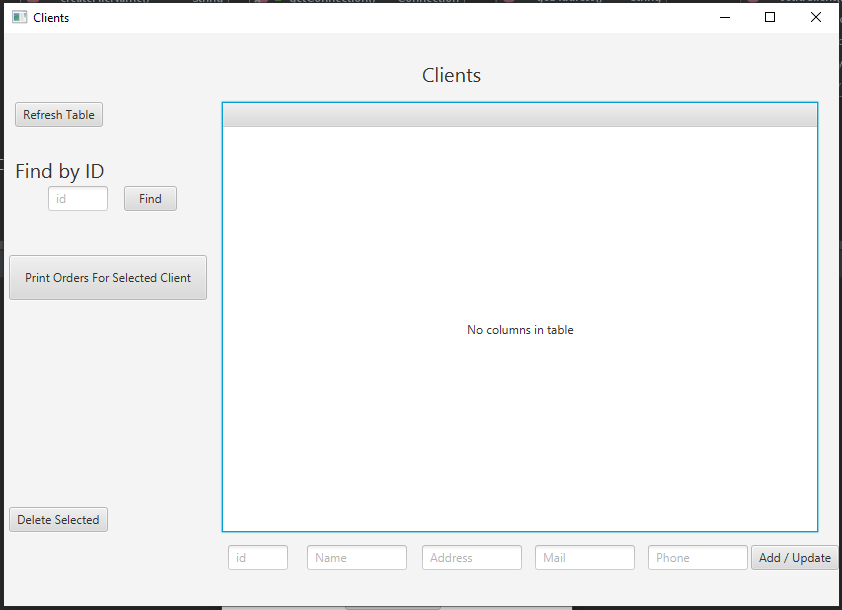
1. The first window is the main one and lets us choose which table we want to manipulate



1. The second one is the Products window which contains a Table View, several buttons and text fields.



1. The third one is the Orders window which contains a Table View, several buttons and text fields. This windows also lets us print the selected order. 
2. The second one is the Clients window which contains a Table View, several buttons and text fields. This window also lets us print all the orders of the selected client



**4. Implementation**

Packages:

1. Presentation Layer (presentation)
   1. AppController Class: This class has 3 buttons. These buttons each creates another panel: a panel for Clients, one for Orders and one for Products

public class AppController *{* public Button ordersButton;  
 public Button clientsButton;  
 public Button productsButton;  
  
 */\*\*  
 \* Opens a new fxml file  
 \** ***@throws*** *IOException  
 \*/* @FXML  
 private void openOrdersPannel*()* throws IOException *{* Parent root = FXMLLoader.*load(*getClass*()*.getResource*(*"../presentation/orders.fxml"*))*;  
 Stage orderStage = new Stage*()*;  
 orderStage.setTitle*(*"Orders"*)*;  
 orderStage.setScene*(*new Scene*(*root, 722, 545*))*;  
 orderStage.show*()*;  
 System.*out*.println*(*"Orders Scene loaded"*)*;  
 *}  
 /\*\*  
 \* Opens a new fxml file  
 \** ***@throws*** *IOException  
 \*/* @FXML  
 private void openClientsPannel*()* throws IOException *{* Parent root = FXMLLoader.*load(*getClass*()*.getResource*(*"../presentation/clients.fxml"*))*;  
 Stage orderStage = new Stage*()*;  
 orderStage.setTitle*(*"Clients"*)*;  
 orderStage.setScene*(*new Scene*(*root, 835, 573*))*;  
 orderStage.show*()*;  
 System.*out*.println*(*"Clients Scene loaded"*)*;  
 *}  
 /\*\*  
 \* Opens a new fxml file  
 \** ***@throws*** *IOException  
 \*/* @FXML  
 private void openProductsPannel*()* throws IOException *{* Parent root = FXMLLoader.*load(*getClass*()*.getResource*(*"../presentation/products.fxml"*))*;  
 Stage orderStage = new Stage*()*;  
 orderStage.setTitle*(*"Products"*)*;  
 orderStage.setScene*(*new Scene*(*root, 602, 544*))*;  
 orderStage.show*()*;  
 System.*out*.println*(*"Products Scene loaded"*)*;  
 *}  
}*

* 1. Clients Controller, OrdersController, ProductsController Classes: are the controller for each correspondent fxml file. In those we control how the application reacts to our user input. I made these classes using several text fields, buttons and a table view.

public class ClientsController {

public ClientsController()

{

tableView = new TableView<>();

}

/\*\*

\* Initializes table columns by Client table

\* @param tableView

\*/

private void initTableColumns(TableView<Clients> tableView)

{

columnId = new TableColumn<>("id");

columnId.setMinWidth(60);

columnId.setCellValueFactory(new PropertyValueFactory<>("idClient"));

columnName = new TableColumn<>("Name");

columnName.setMinWidth(100);

columnName.setCellValueFactory(new PropertyValueFactory<>("name"));

columnAddress = new TableColumn<>("Address");

columnAddress.setMinWidth(100);

columnAddress.setCellValueFactory(new PropertyValueFactory<>("address"));

columnMail = new TableColumn<>("Mail");

columnMail.setMinWidth(100);

columnMail.setCellValueFactory(new PropertyValueFactory<>("mail"));

columnPhone = new TableColumn<>("Phone");

columnPhone.setMinWidth(100);

columnPhone.setCellValueFactory(new PropertyValueFactory<>("phone"));

tableView.getColumns().addAll(columnId,columnName,columnAddress,columnMail,columnPhone);

}

/\*\*

\* Refreshes table

\*/

public void refreshTable()

{

tableView.getItems().clear();

tableView.getColumns().clear();

tableView.setEditable(true);

initTableColumns(tableView);

List<Clients> clientsList = clientDAO.selectAll();

final ObservableList<Clients> data = FXCollections.observableArrayList(clientsList);

tableView.setItems(data);

}

/\*\*

\* Gets input from text fields and adds client in SQL table and refreshes the tableview

\*/

public void addClient()

{

int id = Integer.parseInt(textFieldId.getText());

String name = textFieldName.getText();

String address = textFieldAddress.getText();

String mail = textFieldMail.getText();

int phone = Integer.parseInt(textFieldPhone.getText());

Clients client = new Clients();

client.setIdClient(id);

client.setName(name);

client.setAddress(address);

client.setMail(mail);

client.setPhone(phone);

clientDAO.add(client);

refreshTable();

}

/\*\*

\* Refreshes text fields with selected table view row

\*/

public void clickOnTableRow()

{

Clients client = tableView.getSelectionModel().getSelectedItem();

textFieldId.setText(Integer.toString(client.getIdClient()));

textFieldName.setText(client.getName());

textFieldAddress.setText(client.getAddress());

textFieldMail.setText(client.getMail());

textFieldPhone.setText(Integer.toString(client.getPhone()));

}

/\*\*

\* Removes selected table row from sql table

\*/

public void remove() {

Clients client = tableView.getSelectionModel().getSelectedItem();

OrderDAO orderDAO = new OrderDAO();

orderDAO.removeAllByClient(client);

clientDAO.remove(client);

refreshTable();

}

public void printOrder()

{

Clients client = tableView.getSelectionModel().getSelectedItem();

clientDAO.getOrdersForClient(client.getIdClient());

}

1. Bussiness Logic Layer (bll)
   1. Clients, Orders and Products classes. These classes are associated with the tables from the database. They don’t have just setters and getters. \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*
2. Data Acces Layer
   1. AbstractDAO: In this class we make the queries which modify our tables. We construct the queries by using the reflection method.
      1. Here we have methods like:

private String createSelectQuery(String field)

{

StringBuilder sb =new StringBuilder();

sb.append("SELECT \* FROM ");

sb.append(type.getSimpleName());

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

return sb.toString();

}

Which Creates selesct query for the field

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

private List<T> createObjects(ResultSet resultSet)

{

List<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();

//System.out.println(method.getName() + " " + field.getName());

method.invoke(instance,value);

}

list.add(instance);

}

} catch (SQLException | IllegalAccessException | InstantiationException | InvocationTargetException | IntrospectionException throwables) {

throwables.printStackTrace();

}

return list;

}

Which Creates objects from the result set

public T findById(int id, String whoseId)

{

Connection connection;

PreparedStatement statement;

ResultSet resultSet;

String query = createSelectQuery("id" + whoseId);

try{

connection = ConnectionFactory.getConnection();

statement = connection.prepareStatement(query);

statement.setInt(1,id);

resultSet = statement.executeQuery();

List<T> ts = createObjects(resultSet);

if(ts.size() > 0)

return ts.get(0);

} catch (SQLException throwables) {

throwables.printStackTrace();

}

return null;

}

-Get object from table by id

public List<T> selectAll(String table)

{

Connection connection;

PreparedStatement statement;

ResultSet resultSet;

String query = "SELECT \* FROM " + table;

try{

connection = ConnectionFactory.getConnection();

statement = connection.prepareStatement(query);

System.out.println(statement);

resultSet = statement.executeQuery();

return createObjects(resultSet);

} catch (SQLException throwables) {

throwables.printStackTrace();

}

return null;

}

Selects all from given table

void addUpdate(@NotNull Clients client, boolean ok, String query) {

Connection connection;

PreparedStatement statement;

try{

connection = ConnectionFactory.getConnection();

statement = connection.prepareStatement(query);

statement.setInt(1,client.getIdClient());

statement.setString(2,client.getName());

statement.setString(3,client.getAddress());

statement.setString(4,client.getMail());

statement.setInt(5,client.getPhone());

if(ok)

statement.setInt(6,client.getIdClient());

System.out.println(statement);

statement.executeUpdate();

} catch (SQLException throwables) {

throwables.printStackTrace();

}

}

Which Updates table or add new entry

static void getOrders(int id, String query) {

Connection connection;

PreparedStatement statement;

try{

connection = ConnectionFactory.getConnection();

statement = connection.prepareStatement(query);

statement.setInt(1, id);

System.out.println(statement);

ResultSet rs = statement.executeQuery();

FileConstructor fC = new FileConstructor(rs,id);

fC.createFile();

fC.writeFile();

//System.out.println("-------------" + fC.currentDir + "----------------");

} catch (SQLException | IOException throwables) {

throwables.printStackTrace();

}

}

Prints Orders in new file by orderID / clientID

public void remove(@NotNull Clients client)

{

Connection connection;

PreparedStatement statement;

String query = "DELETE FROM clients WHERE (idClient = ?);";

try{

connection = ConnectionFactory.getConnection();

statement = connection.prepareStatement(query);

statement.setInt(1,client.getIdClient());

System.out.println(statement);

statement.executeUpdate();

} catch (SQLException throwables) {

throwables.printStackTrace();

}

}

Removes client from table

public boolean canAddOrder(Orders order)

{

Connection connection;

PreparedStatement statement;

String query = "SELECT idProduct FROM products WHERE idProduct = " + order.getIdProduct();

try{

connection = ConnectionFactory.getConnection();

statement = connection.prepareStatement(query);

System.out.println(statement);

ResultSet rs = statement.executeQuery();

if(rs.next())

{

query = "SELECT idClient FROM clients WHERE idClient = " + order.getIdClient();

statement = connection.prepareStatement(query);

System.out.println(statement);

rs = statement.executeQuery();

if(rs.next())

return true;

}

} catch (SQLException throwables) {

throwables.printStackTrace();

}

return false;

}Check if order can be added (Client and Product exists)

* 1. ClientDAO, OrderDAO, ProductDAO. These classes inherit the methods from AbstractDAO.

1. Files

Here we have the class FileConstructor which helps in creating the invoice for each client or the invoice for each order.

public class FileConstructor {

private ResultSet rs;

private File file;

private int clientId;

public String path = System.getProperty("user.dir") + "\\reports\\";

private String fileName;

public FileConstructor(ResultSet rs, int clientId)

{

this.rs = rs;

this.clientId = clientId;

}

/\*\*

\* Check if file already exists

\* @param fileName

\* @param fileNo

\* @return

\*/

private boolean fileExists(String fileName, int fileNo)

{

File tmpDir = new File(path + fileName + Integer.toString(fileNo) + ".txt");

return tmpDir.exists();

}

/\*\*

\* Creates file name: clientId + 0 + (next file with same name) .txt

\* @return

\*/

String createFileName()

{

String fileName;

fileName = clientId + "0";

int fileNo = 1;

while(fileExists(fileName, fileNo))

{

fileNo++;

}

fileName += fileNo + ".txt";

return fileName;

}

/\*\*

\* Creates file

\*/

public void createFile() throws IOException {

fileName = createFileName();

path += fileName;

file = new File(path);

file.createNewFile();

System.out.println(fileName);

System.out.println(path);

}

/\*\*

\* Writes in file the Result Set obtained from SQL query

\* @throws IOException

\* @throws SQLException

\*/

public void writeFile() throws IOException, SQLException {

FileWriter fw = new FileWriter(file);

BufferedWriter out = new BufferedWriter(fw);

String tab = " ";

out.write("\n");

out.write(tab + "Order no." + fileName.substring(0,fileName.length()-4) + "\n\n");

rs.next();

out.write("Order ID: " + rs.getString(1) + "\n");

out.write("Customer Name: " + rs.getString(2) + "\n");

out.write("Address: " + rs.getString(3) + "\n");

out.write("Mail: " + rs.getString(4) + "\n");

out.write("Phone: " + rs.getString(5) + "\n");

out.write("\n");

do{

out.write(tab + "Product ID: " + rs.getString(6) + "\n");

out.write(tab + "Product Name: " + rs.getString(7) + "\n");

out.write(tab + "Product Price: " + rs.getString(8) + "\n");

out.write(tab + "Quantity: " + rs.getString(9) + "\n");

out.write(tab + tab + tab + tab + "Total: " + rs.getInt(8) \* rs.getInt(9) + "\n");

out.write("\n");

}while(rs.next());

// out.write("" + rs.getString());

// out.write("" + rs.getString());

// out.write("" + rs.getString());

out.flush();

out.close();

//fw.write();

}

public static void main(String[] args) throws IOException, SQLException {

FileConstructor fC = new FileConstructor(null, 5);

fC.createFile();

fC.writeFile();

}

}

**5. Results**

I didn’t use any testing methods but the one by the hand. I entered every type of data I could to make the program crash but as my expectations, it didn’t. I consider that the program will run smoothly and correctly at every input (of course, input of the form mentioned above).

**6. Conclusions**

It was an interesting project. I learned to make invoices, each in new file. The database part wasn’t so hard as I already knew what we were supposed to learn. I liked the reflection part as I didn’t knew any of those things and will definitely help on in the future.

**7. Bibliography**

* <https://www.jetbrains.com/help/idea/javafx.html>
* <https://www.jetbrains.com/help/idea/class-diagram.html>
* <https://docs.oracle.com/javafx/scenebuilder/1/get_started/jsbpub-get_started.htm>
* <https://stackoverflow.com/>
* <https://openjfx.io/openjfx-docs/>
* <https://www.baeldung.com/>