

DOCUMENTATION

PROJECT 3

NAME OF STUDENT: Jakab-Gyik Sarolta
GROUP: 30423

TABLE OF CONTENTS

1.	Objectives	3
2.	Problem analysis, modelling, scenarios, cases of utilization	3
3.	Description	Error! Bookmark not defined.
4.	Implementation	8
5.	Result	17
6.	Conclusion.....	19
7.	Bibliography.....	20

1. Objectives

Main objective:

Design and implement an application for managing the client orders for a warehouse

Sub-objectives:

- Analyze the problem and identify requirements
- Design the orders management application
- Implement the orders management application
- Test the orders management application

2. Problem analysis, modelling, scenarios, cases of utility

Functional requirements:

- The application should allow an employee to add a new client
- The application should allow an employee to add a new product
- The application should allow an employee to delete a client
- The application should allow an employee to delete a product
- The application should allow an employee to update a client
- The application should allow an employee to update a product
- The application should allow a client to add a new order
- The application should allow a client to delete an order
- The application should allow a client to update an order

Use Case: add product

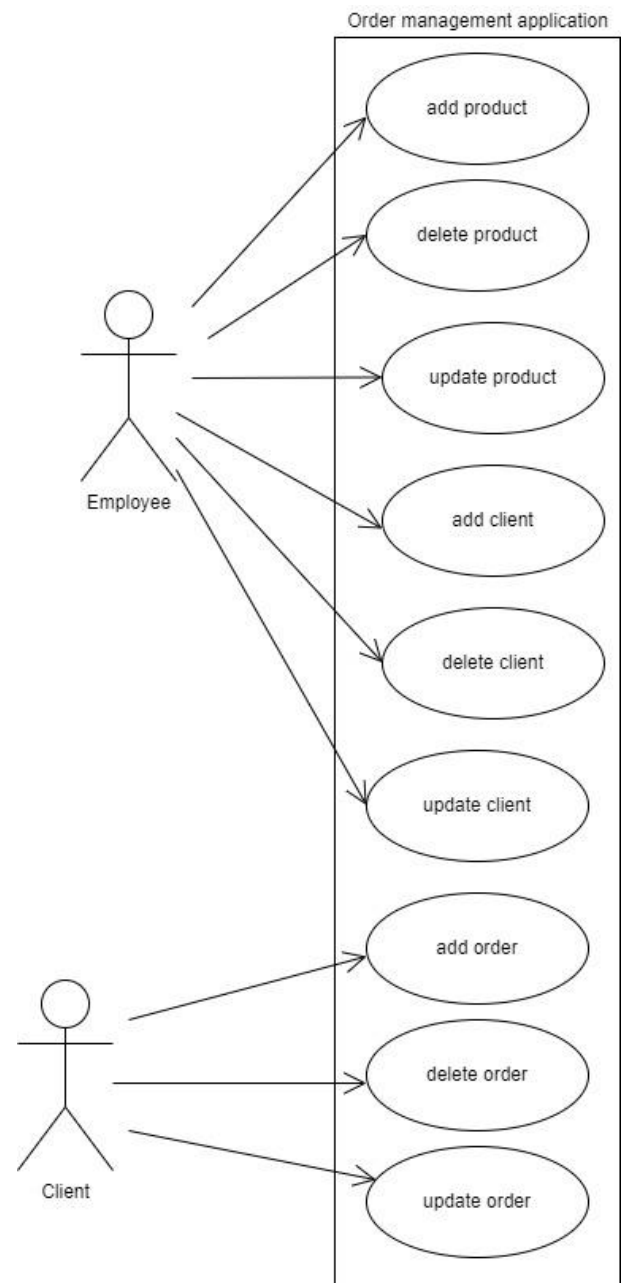
Primary Actor: employee

Main Success Scenario:

1. The application will display a form in which the product details should be inserted
2. The employee selects the option to add a new product
3. The employee inserts the name of the product, its price and current stock
4. The employee clicks on the “Add” button
5. The application stores the product data in the database and displays the updated table

Alternative Sequence: Invalid values for the product’s data

- The user inserts a negative value for the stock of the product
- The application displays an error message and requests the user to insert a valid stock
- The scenario returns to step 3



Use Case: delete product

Primary Actor: employee

Main Success Scenario:

1. The application will display a form in which the product details should be inserted
2. The employee selects the option to delete a product
3. The employee inserts the id, the name of the product, its price and current stock
4. The employee clicks on the “Delete” button
5. The application deletes the product data in the database and displays the updated table

Alternative Sequence: Invalid values for the product’s data

- The user inserts a product that is not in the database
- The application displays an error message and requests the user to insert an available product
- The scenario returns to step 3

Use Case: update product

Primary Actor: employee

Main Success Scenario:

1. The application will display a form in which the product details should be inserted
2. The employee selects the option to update a product
3. The employee inserts the id, the name of the product, its price and current stock
4. The employee clicks on the “Update” button
5. The application updates the product data in the database and displays the updated table

Alternative Sequence: Invalid values for the product’s data

- The user inserts a product that is not in the database
- The application displays an error message and requests the user to insert an available product
- The scenario returns to step 3

Use Case: add client

Primary Actor: employee

Main Success Scenario:

6. The application will display a form in which the product details should be inserted
7. The employee selects the option to add a new client
8. The employee inserts the name of the client, his address and email
9. The employee clicks on the “Add” button
6. The application stores the client’s data in the database and displays the updated table

Alternative Sequence: Invalid values for the client’s data

- The user inserts a negative value for the id of the client
- The application displays an error message and requests the user to insert a valid id
- The scenario returns to step 3

Use Case: delete client

Primary Actor: employee

Main Success Scenario:

7. The application will display a form in which the client details should be inserted
8. The employee selects the option to delete a client
9. The employee inserts the id
10. The employee clicks on the “Delete” button
11. The application deletes the client data in the database and displays the updated table

Alternative Sequence: Invalid values for the client’s data

- The user inserts a client that is not in the database
- The application displays an error message and requests the user to insert a valid client

- The scenario returns to step 3

Use Case: update client

Primary Actor: employee

Main Success Scenario:

6. The application will display a form in which the client details should be inserted
7. The employee selects the option to update a client
8. The employee inserts the id, the name of the client, the address and the email
9. The employee clicks on the “Update” button
10. The application updates the client data in the database and displays the updated table

Alternative Sequence: Invalid values for the client’s data

- The user inserts a client that is not in the database
- The application displays an error message and requests the user to insert a valid client
- The scenario returns to step 3

Use Case: add order

Primary Actor: client

Main Success Scenario:

1. The application will display a form in which the product details should be inserted
2. The client selects the option to add a new product
3. The client inserts the id of the client, of the product and the amount
4. The client clicks on the “Add” button
5. The application stores the order data in the database and displays an acknowledge message

Alternative Sequence: Invalid values for the order’s data

- The user inserts a negative value for the amount of the order
- The application displays an error message and requests the user to insert a valid amount
- The scenario returns to step 3

Use Case: delete order

Primary Actor: client

Main Success Scenario:

1. The application will display a form in which the product details should be inserted
2. The client selects the option to delete a product
3. The client inserts the id, the id of the client, of the product and the amount
4. The client clicks on the “Delete” button
5. The application deletes the order data in the database and displays the updated table

Alternative Sequence: Invalid values for the order’s data

- The user inserts an order that is not in the database
- The application displays an error message and requests the user to insert an available order
- The scenario returns to step 3

Use Case: update order

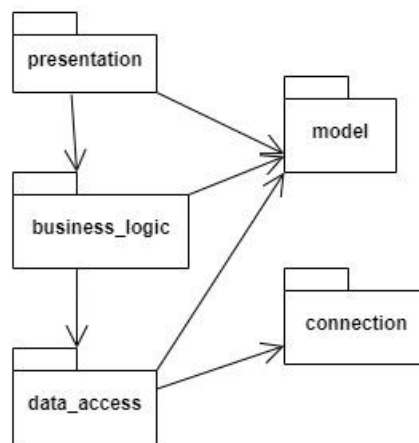
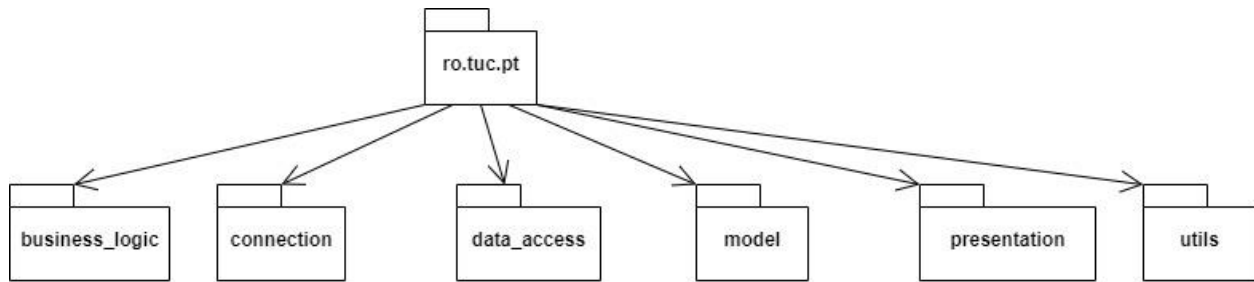
Primary Actor: client

Main Success Scenario:

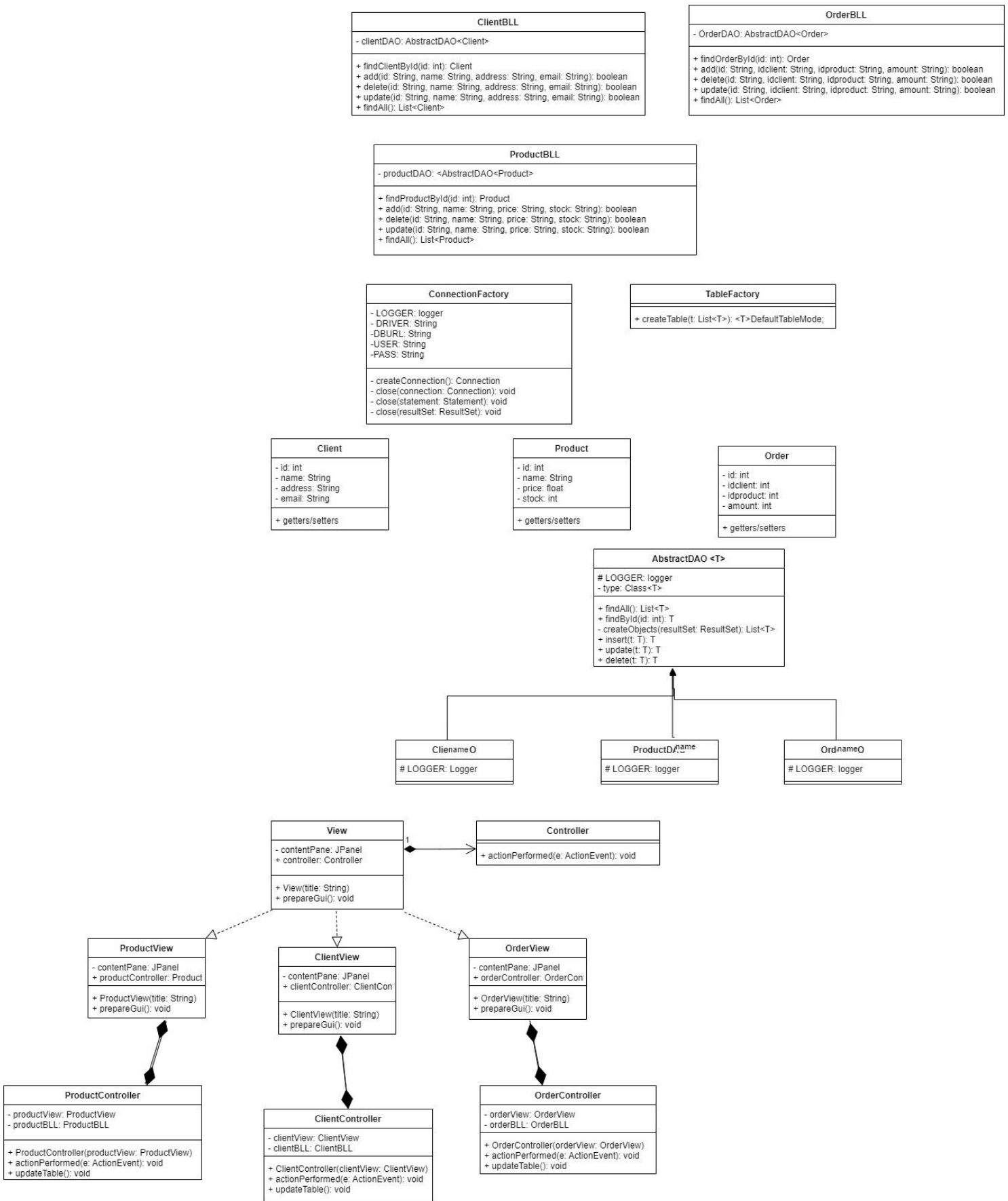
1. The application will display a form in which the product details should be inserted
2. The client selects the option to update a order
3. The client inserts the id, the id of the client, of the product and the amount
4. The client clicks on the “Update” button
5. The application updates the order in the database and displays the updated table

Alternative Sequence: Invalid values for the order's data

- The user inserts an order that is not in the database
- The application displays an error message and requests the user to insert an available order
- The scenario returns to step 3



~ Package diagram~



~Class diagrams~

3. Description

The application uses the layered architectural pattern to realize an Orders Management system. The purpose of this architectural pattern is the division of the components into layers, according to their functionality. The top layer is the presentation layer, presented by the package with the same name. It contains the View and Controller classes, and also the utils package with the TableModel. The second layer is the business logic, so the business_logic package with the BLL classes. The third layer is the data access. In this layer we have the data_access package with the AbstractDAO class performing the SQL queries according to the user's needs. Also, the ClientDAO, ProductDAO and OrderDAO classes belong to the same layer, as well as the ConnectionFactory class from the connection package. The bottom layer is the MySQL database itself, the order_management schema with the three tables. The additional package, the model represents the different types of data introduced into these tables. The fields of these classes from the model package are the same as the columns in the tables.

Additional knowledge that was required for this project is the query language so the Java code can be interpreted as an SQL command and executed of the database. The table mapping from the database to the screen also required to implement this Java application in a Maven project form. Every time the application interacts with the database, the connection between them is established first, then the query gets executed and returns a ResultSet type of object, and at the end the connection is closed.

For this project, I use reflection technique, which is a unique technique in Java language. It is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its members and display them. I have used the generic types provided by Java to avoid code repetition and make the code easier to understand and to write. In generalizes for example the add, delete, update and findById operations when we can simply call them from the AbstractDAO class and there is no need for a separate implementation in case of different tables.

4. Implementation

About every class in details:

Connection package:

ConnectionFactory

This class represents the basis of the connection between the Java application and the order_management schema. It has several functions like createConnection() or close() for closing the connection, the statement or the resultSet returned by the database query. To be able to realize this connection, the programmer needs to provide the necessary parameters, like the database url link, the mysql driver and also the username and password that he would use to visualize the database from the Workbench.

```
private Connection createConnection() {
    Connection connection = null;
    try {
        connection = DriverManager.getConnection(DBURL, USER, PASS);
    } catch (SQLException e) {
        LOGGER.log(Level.WARNING, msg: "An error occurred while trying to connect to the database");
        e.printStackTrace();
    }
    return connection;
}
```


Model package:

Client

This class models the client table from order_management schema. It has fields that correspond to the columns of the database table: id, name, address, and email. It also provides getter and setter methods for all of these private fields.

Product

This class models the client table from order_management schema. It has fields that correspond to the columns of the database table: id, name, price, and stock. It also provides getter and setter methods for all of these private fields.

Order

This class models the client table from order_management schema. It has fields that correspond to the columns of the database table: id, idclient, idproduct, and amount. It also provides getter and setter methods for all of these private fields.

Data_access package:

ClientDAO

This class extends the AbstractDAO class, providing as generic parameter the Client type. It has a single static field, the Logger field, used by the AbstractDAO to insert the name of the child class.

ProductDAO

This class extends the AbstractDAO class, providing as generic parameter the Product type. It has a single static field, the Logger field, used by the AbstractDAO to insert the name of the child class.

OrderDAO

This class extends the AbstractDAO class, providing as generic parameter the Order type. It has a single static field, the Logger field, used by the AbstractDAO to insert the name of the child class.

AbstractDAO

This class is the parent of all other DAO classes. It performs operations on the Client, Product and Order tables. It has several methods to create different sql queries. It creates them using reflection technique, like the .getSimpleName() method as shown below. There are separate methods for select queries, delete queries, update queries and find all queries because their sql syntax differs.

Then there are several methods that call their stringBuilder methods and establish a connection with the database to execute these queries. The add, delete, update, findById and findAll methods are based on this principle. Their structure is quite similar, they follow the same fashion. At the end of the methods we close the connections with the database management system.

```
/**
 * The createSelectQuery method uses a StringBuilder and creates the select SQL query to be executed by
 * other methods that work with the database tables
 * @param field is the criteria of the selection
 * @return the select type SQL query
 */
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();
}
```

```

/**
 * The insert method creates the connection to the database. Then, it executes the created insert query.
 * In the end, it closes the connection.
 * @param t the generic parameter that could be a client, product or order
 * @return The instance that was inserted successfully
 * @throws IllegalAccessException when trying to access certain fields declared private
 */
public T insert(T t) throws IllegalAccessException {
    Connection dbConnection = ConnectionFactory.getConnection();

    PreparedStatement insertStatement = null;
    String query = createInsertQuery(t);
    try {
        insertStatement = dbConnection.prepareStatement(query, Statement.RETURN_GENERATED_KEYS);
        Field[] fields = t.getClass().getFields();
        for (int i = 0; i < fields.length - 1; i++){
            PropertyDescriptor propertyDescriptor = new PropertyDescriptor(fields[i].getName(), type);
            Method method = propertyDescriptor.getReadMethod();
            insertStatement.setObject(i, method.invoke(t));
        }
        insertStatement.executeUpdate();
    } catch (SQLException | IntrospectionException e) {
        LOGGER.log(Level.WARNING, msg: "AbstractDAO:insert " + e.getMessage());
    } catch (InvocationTargetException | IllegalAccessException e) {
        e.printStackTrace();
    } finally {
        ConnectionFactory.close(insertStatement);
        ConnectionFactory.close(dbConnection);
    }
    return t;
}

```

Business_logic package:

ClientBLL

This class calls the methods from the AbstractDAO class to perform the operations on the tables. Every method verifies at first if all the parameters provided by the user are in the correct range and are not null. If some parameters are given incorrectly, then a message is displayed on the screen and the user must provide other input parameters. If the parameters are given correctly, a new object instance is created and transmitted to the function as a parameter. After any operation the table on the graphical user interface is updated as well, so the user can see if the operation's result is the expected one.

```

/**
 * The method calls the add method of the clientDAO class and verifies if the client was found.
 * Firstly, it checks the parameters to have a correct value, other than null.
 * Then it creates a new client and inserts it into the Client table.
 * @param id the id of the client
 * @param name the name of the client
 * @param address the address of the client
 * @param email the email of the client
 * @return true if the insert was completed successfully, and false otherwise
 */
public boolean add(String id, String name, String address, String email){
    if(id.isEmpty() || name.isEmpty() || address.isEmpty() || email.isEmpty())
        return false;
    Client client = new Client(Integer.parseInt(id), name, address, email);
    try {
        clientDAO.insert(client);
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
    return true;
}

```

```

/**
 * The method finds all the rows of the database table Client.
 * @return a list of clients found in the database table
 */
public List<Client> findALL() {
    List<Client> clients = clientDAO.findAll();
    return clients;
}

```

ProductBLL

This class calls the methods from the AbstractDAO class to perform the operations on the tables. Every method verifies at first if all the parameters provided by the user are in the correct range and are not null. If some parameters are given incorrectly, then a message is displayed on the screen and the user must provide other input parameters. If the parameters are given correctly, a new object instance is created and transmitted to the function as a parameter. After any operation the table on the graphical user interface is updated as well, so the user can see if the operation's result is the expected one.

The stock of the products is updated even when the user wants to manipulate the order table. For every order the stock is decremented, intuitively. If the order gets updated, so the customer wants to order more from the product, then the stock gets decremented again, and if the order gets deleted the stock cannot be incremented, because the stores do not take back the products already sold.

OrderBLL

This class calls the methods from the AbstractDAO class to perform the operations on the tables. Every method verifies at first if all the parameters provided by the user are in the correct range and are not null. If some parameters are given incorrectly, then a message is displayed on the screen and the user must provide other input parameters. If the parameters are given correctly, a new object instance is created and transmitted to the function as a parameter. After any operation the table on the graphical user interface is updated as well, so the user can see if the operation's result is the expected one.

If the customer performs any operation on this table, the product table gets updated as well. A customer can only buy products, not return any, therefore, the correct stock from a certain product can only decrease. If there are no more available items in stock from the product specified, then the customer gets a notification, as in case of incorrect input.

Utils package:

TableFactory

From the perspective of the graphical user interface, this class creates the table that the user can see on the interface. It gets updated after every operation on the table(add, delete and update functions all have to call updateTable after their execution.

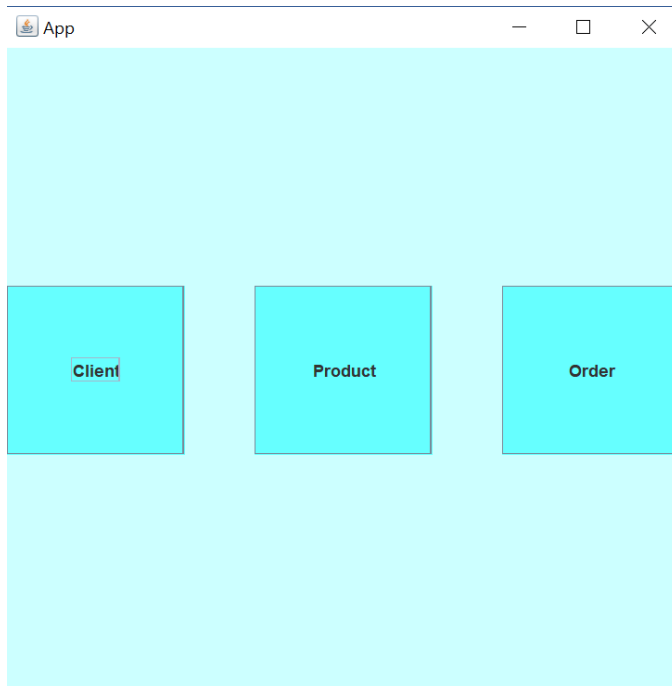
```
public static <T>DefaultTableModel createTable(List<T> t){
    if(t.size() == 0)
        return new DefaultTableModel();
    String[] columns = new String[t.get(0).getClass().getDeclaredFields().length];
    String[][] rows = new String[t.size() + 1][t.get(0).getClass().getDeclaredFields().length + 1];
    int i = 0;
    for(Field f:t.get(0).getClass().getDeclaredFields()){
        columns[i] = f.getName();
        rows[0][i] = f.getName();
        i++;
    }
    i = 1;
    int j = 0;
    for(T tt: t){
        for(Field ff:tt.getClass().getDeclaredFields()){
            try {
                ff.setAccessible(true);
                rows[i][j] = ff.get(tt) + "";
                ff.setAccessible(false);
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
            j++;
        }
        j = 0;
        i++;
    }
    DefaultTableModel table = new DefaultTableModel(rows, columns);
    return table;
}
```

Presentation package:

The classes make up the user interface and call the corresponding methods if a button gets pushed on one of the windows of the interface.

View

The View class is the first window that pops up when the application starts running. The user can select his role, therefore, he gets the information relevant to what he wants to do, the operations he wants to perform (the employee of the company can do different operations and a client can do different operations). To protect the access to these windows, one should add a login window. The users should have a username and password to protect their identity and for the application to work correctly.



Controller

In the controller class the clicking of a button is detected. This class implements the ActionListener interface and therefore, overrides the actionPerformed() method. It opens one of the three windows for the user: the client table, the product table or the order table.

ClientView

This class has a similar structure to the View class. It extends the JPanel superclass and is part of the Java Swing interface. It uses the DefaultTableModel as well as some buttons to indicate the operations that need to be performed. To manipulate the data in the tables, one can click on the add, delete or update buttons and introduce the correct parameters.

Client window

id:

name:

address:

email:

add

delete

update

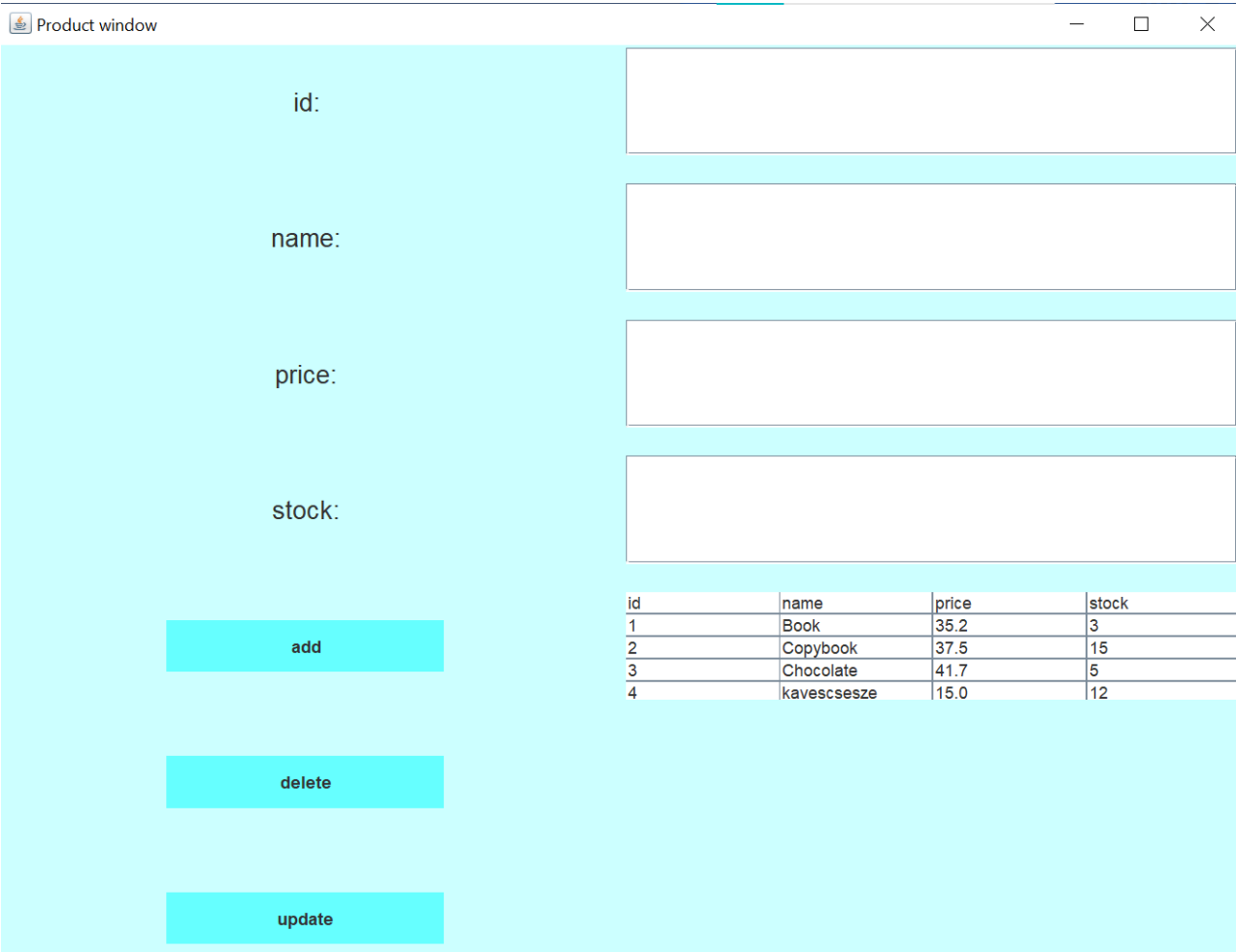
id	name	address	email
1	Sarah J Maas	Transylvania	sarah@gmail.com
2	John Green	Pennsylvania	john_green@yah...
3	Sarolta	Portile de fier	sarolta@gmail.com
4	Balazssss	Piata Abator 4-7	vitusbalazs01@v...

ClientController

The Controller classes have a similar structure to the Controller class. This class also implements the ActionListener interface and overrides, therefore, the actionPerformed() method. It calls the add, delete or update methods, depending on which button was selected by the user. After the query is executed, the updateTable method is called to project these changes into the user interface.

ProductView

This class has a similar structure to the View class. It extends the JPanel superclass and is part of the Java Swing interface. It uses the DefaultTableModel as well as some buttons to indicate the operations that need to be performed. To manipulate the data in the tables, one can click on the add, delete or update buttons and introduce the correct parameters.



The screenshot shows a Java Swing window titled "Product window". The window has a light blue background. On the left side, there are four labels: "id:", "name:", "price:", and "stock:". To the right of each label is a white text input field. Below these input fields, there are three red buttons: "add", "delete", and "update". To the right of the buttons, there is a table with four columns: "id", "name", "price", and "stock". The table contains four rows of data.

id	name	price	stock
1	Book	35.2	3
2	Copybook	37.5	15
3	Chocolate	41.7	5
4	kavescsesze	15.0	12

ProductController

The Controller classes have a similar structure to the Controller class. This class also implements the ActionListener interface and overrides, therefore, the actionPerformed() method. It calls the add, delete or update methods, depending on which button was selected by the user. After the query is executed, the updateTable method is called to project these changes into the user interface.

OrderView

This class has a similar structure to the View class. It extends the JPanel superclass and is part of the Java Swing interface. It uses the DefaultTableModel as well as some buttons to indicate the operations that need to be performed. To manipulate the data in the tables, one can click on the add, delete or update buttons and introduce the correct parameters.

Order window

id:

idclient:

idproduct:

amount:

add

delete

update

id	idclient	idproduct	amount
1	2	3	1
2	1	2	1

OrderController

The Controller classes have a similar structure to the Controller class. This class also implements the ActionListener interface and overrides, therefore, the actionPerformed() method. It calls the add, delete or update methods, depending on which button was selected by the user. After the query is executed, the updateTable method is called to project these changes into the user interface.

5. Results

All Classes and Interfaces

Classes	
Class	Description
AbstractDAO<T>	Tool to access the database.
Client	The Client class models a client from the client table of the database.
ClientBLL	Client business logic class that calls the methods of the ClientDAO class.
ClientController	The ClientController class implements ActionListener interface and its method actionPerformed is called every time the user selects any of the operations insert, delete or update.
ClientDAO	Tool to access the client table from the database.
ClientView	The ClientView class extends JFrame and creates the client window with the client table.
ConnectionFactory	The ConnectionFactory class connects the Java code with the Order_management MySQL database.
Controller	The Controller class implements ActionListener interface and its method actionPerformed is called every time the user selects any of the 3 buttons.
Main	
Order	The Order class models an order from the order table of the database.
OrderBLL	Order business logic class that calls the methods of the OrderDAO class.
OrderController	The OrderController class implements ActionListener interface and its method actionPerformed is called every time the user selects any of the operations insert, delete or update.
OrderDAO	Tool to access the order table from the database.
OrderView	The OrderView class extends JFrame and creates the order window with the order table.
Product	The Product class models a product from the product table of the database.
ProductBLL	Product business logic class that calls the methods of the ProductDAO class.
ProductController	The ProductController class implements ActionListener interface and its method actionPerformed is called every time the user selects any of the operations insert, delete or update.
ProductDAO	Tool to access the product table from the database.
ProductView	The ProductView class extends JFrame and creates the product window with the product table.
TableFactory	The TableFactory class realizes the connection to the mySQL Order_Management database and creates a table model for each table to print on the screen.
View	The View class extends JFrame and creates the initial window of the application.

As part of the project’s requirements, I have documented all the public classes and methods using JavaDoc comments. These comments have helped me and will help in the future development of the application. After generating the JavaDoc file, the programmer can simply verify the internal structure of the whole project, the packages, inside them the classes with the public fields and methods, having a short explanation of their usecase and when could they be invoked.

Because this project involves a database schema, I have generated the dump file of the order_management schema. This file shows us the operations executed in the database, the creation of the tables, the manipulation of the data and so on. An example of the file’s contents:

All Packages	
Package Summary	
Package	Description
ro.tuc.pt	
ro.tuc.pt.business_logic	
ro.tuc.pt.connection	
ro.tuc.pt.data_access	
ro.tuc.pt.model	
ro.tuc.pt.presentation	
ro.tuc.pt.utils	

```

18  --
19  -- Table structure for table `client`
20  --
21
22  • DROP TABLE IF EXISTS `client`;
23  • /*!40101 SET @saved_cs_client      = @@character_set_client */;
24  • /*!50503 SET character_set_client = utf8mb4 */;
25  • CREATE TABLE `client` (
26      `id` int NOT NULL,
27      `name` varchar(45) DEFAULT NULL,
28      `address` varchar(45) DEFAULT NULL,
29      `email` varchar(45) DEFAULT NULL,
30      PRIMARY KEY (`id`),
31      UNIQUE KEY `id_UNIQUE` (`id`),
32      UNIQUE KEY `name_UNIQUE` (`name`)
33  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
34  • /*!40101 SET character_set_client = @saved_cs_client */;
35

```

I have also created a bill for every order that a client places. It shows the information about the client (the name in this case), shows the product's name, the unit price and the amount ordered, and at the end it calculates the total cost that needs to be paid:

	BON FISCAL
Client:	Sarah J Maas
Product:	Book
Amount:	1
Unit price:	35.2
Total:	35.20

6. Conclusion

This application is a design for a more complex application that could model a whole store or a chain of stores that represent a brand. The costumers and employees could be able to do different operation on the data, for example, they clients could be able to buy different products together, in a single order. Or the tables could hold more information about the customers, for example their favorite items or previous orders. The possibilities are countless. They could rate the products or write a small review as well.

The graphical user interface could improve tremendously. Because of the time limit, I did not put any emphasis on the design part, but it would have to be way more appealing for a customer to use it on a regular basis.

This project was a great way to learn more about Java techniques, like reflection, to use MySQL for the first time and to understand how can a database communicate with the application through queries.

7. Bibliography

The resources for this project were the pdf-s provided by the professor, the presentation of the project and the pdf guide.

I have also tackled several problems with the help of different websites explaining different Java concepts.