# 

ASSIGMENT NUMBER 3

**-Programming Techniques-**

Orders Management

STUDENT NAME: Tat Teodora

GROUP: 30423

# CONTENT

1. Assignment Objectives
2. Problem analysis, modelling, scenarios, use cases
3. Design
4. Implementation
5. Results and testing
6. Conclusions
7. Bibliography
8. Assigment Objectives
9. Main objective

*Project specification:*

Consider an application Orders Management 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

1. Sub-objectives:

|  |  |  |
| --- | --- | --- |
| Objectiv | Description | The Chapter in which it is described |
| Problem Analisys | In order to translate the problem into code, it is first necessary to extract from the statement relevant information about the functionalities and use cases. | II |
| Projection of the application | Once the necessary information has been extracted, you can start mapping it to the programming paradigm provided (Object Oriented Programming in Java in this case) | [III](file:///C:\Users\VivoBook\Downloads\Documentatie%20depozit.docx#_Proiectare) |
| Implementation of the application | In this step, the code is written to solve the required problem, based on the previous mapping. | IV |
| Testing of the application | In order to ensure that the final product or even partial results required for a next step work correctly, the implemented code must in turn be subjected to tests that simulate the usage scenarios identified in the analysis step. | [V](file:///C:\Users\VivoBook\Downloads\Documentatie%20depozit.docx#_Testare_și_rezultate) |

1. Problem analysis, modelling, scenarios, use cases

**Functional requirements**

*Persistence of information*. Customer, product, and order information must persist after closing the application, which requires the use of a relational database.

*Managing operations from a graphical interface*. It is more natural that in such a scenario where users are not programmers to make a graphical interface for managing operations. From a menu, you can choose whether you want operations on customers products or orders.

*Automatically create a table*. It is necessary to display in the graphical interface the information from the database in a JTable. Furthermore, its header containing the displayed attribute names must be generated automatically by Reflection

*Data validation*. This aspect refers to the verification of the correctness of the data before entering them in the database. When designing an application for a user, the chances are high that a person will make a mistake when he wants to assign values ​​to some fields, either inadvertently or out of pure sociopathic intentions. Because the table fields in the database depend on implementation, cases of erroneous data and their actual treatment will be discussed later.

**Non-functional requirements**

*Issuing a receipt for an order*. A receipt containing the information related to any order, as well as real-time orders, would be a good thing to have so that an employee does not have to do it manually every time.

*Query automation.* The syntax of MySQL queries can be reduced to a pattern that depends only on the type of operation (delete, edit, select, insert). Therefore, knowing Java Reflection techniques, they can be easily generated as long as the tables are mapped in Java with exactly the same name as in MySQL.

**Use case scenarios**

Graphical user interface, application

Description automatically generated

The application must be designed for use by a warehouse manager. In general, the usage steps are simple: the user opens the main menu and selects the desired option, which will open a menu with a table containing the information related to the required category. After that, select the desired operation (for example, insert), possibly enter additional data and press a special button. After these steps, he can return to the main menu to perform another operation. An example scenario is:

Usage scenario: insert a new product

Main actor: user (warehouse manager)

Main use case:

1. The main menu opens

2. Press the Choose operation on “Product” option button

3. Press button “Add” on the new window to create a new area for product attributes

4. The user writes the data for the product to be entered in a dedicated area

5. Press enter to confirm the data

6. The entered data is verified

7. Insert in the corresponding table

8. The user either returns to the main menu to operate on something else, or closes the application

Alternative scenarios:

a. Incorrect data is entered

- Nothing is entered in the database

- The user is warned

Most use cases are reduced to operation on tables in the database in the graphical interface, checking backwards the correctness of the information entered.

1. Design

**Architectural patterns**

**Singleton Pattern**. It is an OOP design mode that involves using a single instance of an object to solve a problem. In this case, it was used to manage the application's connection to the database, including establishing a connection and closing it or other objects such as Statements and ResultSets.

**Layered Architecture**

For a better organization of the classes, the project was broken into several packages, each dealing with a different aspect of the application:

The **model** package. Contains Java mapping of entities and additional data types in the database.

Diagram

Description automatically generated

**DataAccess** package. Here are the classes responsible for accessing data from the database and connection management.Diagram

Description automatically generated

The **bll** package. It involves validating the information before entering it into the database. The erroneous data that can be entered for each class are the following:

|  |  |  |
| --- | --- | --- |
| Clients | Orders | Product |
| * Attempting to change the ID to any value * Age is checked if the client is over 18 years old * Email does not follow a pattern with letters@address.domain   Phone number if valid (starts with 0, has 10 digits, etc.) | * Attempting to change ID * Enter a date with the wrong format | * Attempting to change ID * Enter a price with an incorrect pattern: more than 2 decimal places and more than 15 digits in total * Enter a quantity that is not an integer |

Presentation package. It includes the graphical interface part and the controller that connects the user to it

**Algorithms**

No special algorithms were used other than going through the instance variables of a class and possibly checking some annotations.

**Data structures**

Saving objects in a ResultSet was performed in a list because their number is not known in advance.

1. Implementation
   1. AbstractDAO implementation

Here we have the methods for creating the queries: createSelectByID(), createInsertQuery(), createEditQuery(). Here is one example:

Text

Description automatically generated

Here is also the function for creating and preparing the data for the table:

private List<T> createObjects(ResultSet resultSet) {  
 List<T> list = new ArrayList<T>();  
 Constructor[] ctors = type.getDeclaredConstructors();  
 Constructor ctor = null;  
 for (int i = 0; i < ctors.length; i++) {  
 ctor = ctors[i];  
 if (ctor.getGenericParameterTypes().length == 0)  
 break;  
 }  
 try {  
 while (resultSet.next()) {  
 ctor.setAccessible(true);  
 T instance = (T)ctor.newInstance();  
 for (Field field : type.getDeclaredFields()) {  
 String fieldName = field.getName();  
 Object value = resultSet.getObject(fieldName);  
 PropertyDescriptor propertyDescriptor = new PropertyDescriptor(fieldName, 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();  
 }  
 return list;  
 }  
}

Then I have implemented the Reflection method for all the three classes: selectByID(), selectAll(), insert(), update() and delete(). The AbstractDAO method is called in the ClientDAO, OrderDAO and ProductDAO. Here is the function update():

public boolean update(T object){  
 Connection connection = null;  
 PreparedStatement statement = null;  
  
 String query = createEditQuery();  
 try {  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(query);  
 int position = 1;  
 for(Field field : type.getDeclaredFields()) {  
 field.setAccessible(true);  
 statement.setObject(position++,field.get(object));  
 }  
 Field field = type.getDeclaredFields()[0];  
 field.setAccessible(true);  
 statement.setObject(position++,field.get(object));  
 statement.executeUpdate();  
 return true;  
 } catch (SQLException | IllegalAccessException e) {  
 *LOGGER*.log(Level.*WARNING*, type.getName() + " DAO:insert "+ e.getMessage());  
 return false;  
 }  
 finally {  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
}

* 1. Bbl package implementation

Here are used the methods from DataAccess package, where they are called and error messages are shown if the validation of the Class Client (the attributes email, phone number and age are verified in the Verification package with the help of an interface). In the OrderBLL is presented also a method generateBill() containing the details of an order when it is placed in the interface.

* 1. Model package implementation

Here I have created the classes Client, Product, Order with their setters and getters, along with a

Override for the toString() method to help writing the bill into a fileWriter.

* 1. ConnectionFactory implementation

Here I have made the connection with the database. There are also implemented some methods to help populate the database.

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/ordersdb";  
private static final String *USER* = "root";  
private static final String *PASS* = "andreea2002!";  
  
private static ConnectionFactory *singleInstance* = new ConnectionFactory();  
  
private ConnectionFactory() {  
 try {  
 Class.*forName*(*DRIVER*);  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
}

* 1. Presentation implementation: The Graphical User Interface

Here I have used JavaFX to help me with the four windows that put together the application.

The file “mainWindow.fxml” is controlled by the class MainWindowController and it is shown three buttons for the user to choose on which table it wants to operate:  
Diagram

Description automatically generated

The file “client.fxml” is controlled by the class ClientsWindowController and it is shown three buttons for the user to choose what operation wants to perform:

Graphical user interface, application, table

Description automatically generated

When “add” button is pressed, a new DEFAULT field (which has a random generated ID, the string fields being called DEFAULT exception being the email and phone which have a validated form so that the app may continue) is shown so that the new values could be added to the table. For editing, the user can select any row, change the value of the field which is saved when he presses the ENTER key. When deleting, the whole selected row is deleted permanently.

Below is shown how a row can be added and updated.

Graphical user interface, application

Description automatically generated

The file “product.fxml” is controlled by the class ProductsWindowController and it is shown three buttons for the user to choose what operation wants to perform:

Graphical user interface, application

Description automatically generated

This interface has the same properties as the Client interface. Here is shown how the product Sunglasses is permanently deleted.

A screenshot of a computer

Description automatically generated

The file “order.fxml” is controlled by the class OrderWindowController and it is shown three buttons for the user to choose what operation wants to perform, but it is different from the Client and Product window because when pressing the “add” button, the window “place-order.fxml” controlled by the PlaceOrderWindowController.

A screenshot of a computer

Description automatically generated

Here is presented the place-order functionality:

Graphical user interface, table

Description automatically generated

The user can select the client and the product the client wants to order, and also the quantity. If the quantity is bigger than the stock, an error message is shown. If not, the order is added in the order window and the stock in the product class is decreased. In case the stock is equal to the quantity, the product is deteled.

Graphical user interface

Description automatically generated

Graphical user interface, table

Description automatically generated

Graphical user interface

Description automatically generated

1. Testing

For testing, I introduced manually many values so that I can perform the operations described.

1. Conclusions

This project was a good exercise in remembering the OOP concepts learned in the first semester, but also learning new ones, I found it very useful and challenging at first. There are a few learned things which I would present next.

his project has been a great help in getting used to Java Reflection. Java Reflection is a unique technique, specific to Java, but very useful, allowing even the abstraction of extracting a class from a ResultSet. Dynamic invocation of methods and obtaining certain markers placed on a variable instance have a very high applicability in projects with certain very repetitive things, such as a database application, where queries and obtaining an object from it have a similar pattern.

Another thing I learned from this project is working with Annotations, which again are very useful, especially used in Reflection to mark certain things about a variable instance, method, or class.

Regarding working with a Java database, I refreshed my memory related to creating and closing a connection to a database, along with basic knowledge about queries (their structure, their Java call, trigger creation)

I also realized that Tables in JavaFX are much more complex and versatile than I thought. In trying to customize Tables with TableCellRenderer and TableCellEditor, we realized that they can replace or make editing much easier than JPanel or JTable with multiple editable components.

And creating PDF documents was interesting. Although I've worked with PdfBox in the past (which is a bit lower-level in the sense that it doesn't have methods for tables by itself, it still needs a Maven dependency for this), this project made me realize that I don't need a license of iText for use in projects.

Validating the data to be entered, I was kept up to date with regex and ways to extract data

One regret is that I did not study Java Annotations more closely. Annotations have great potential, they can also have values ​​assigned in the back, and if we analyse a little more, maybe I managed to abstract the extraction of an OrderItem from the table.

1. Bibliography
   1. Object-Oriented Programming - Lecture Slides of prof. Marius JOLDOS
   2. Programming Techniques – Lectures of prof. Cristina POP
   3. [www.stackoverflow.com](http://www.stackoverflow.com)
   4. [www.wikipedia.org](http://www.wikipedia.org)
   5. <http://whatis.techtarget.com/definition/model-view-controller-MVC>
   6. <https://www.draw.io/> (for diagrams)