Technical University of Cluj-Napoca

Fundamental Programming Techniques

Laboratory – Assignment 3

Order Management

Logo, company name

Description automatically generated

Teacher: prof. Ioan Salomie

Teacher Assistant: Ciprian Adrian Stan

Student: Itu Anca

Group: 30422

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 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

This application should provide an interface for the user which he can use to perform some operations on the data stored in a relational MySQL database.

1. Problem analysis, modeling, scenarios, use cases

Problem analysis

A database is a collection of data that is specially organized for rapid search and retrieval by a computer. It is structured to facilitate the storage, retrieval, modification and deletion of data using various data-processing operations.

In this application, a database is used in order to store the products, the clients and the orders needed for processing client orders for a warehouse.

Modelling

In the main stage there are 3 buttons that the user has to click on in order to choose which one of the three tables he wants to manage: the products, the customers or the orders table.

If the user chooses to manage the “customers” table:

In the following scene, on the left there are 4 buttons: one button is destinated to the operation of adding a anew customer to the table, another one to edit an existing customer in the table, another button is used for deletion of a customer from the table and the last one is used to view all the data included in the customers table.

* If the user wants to add a new customer to the table, he should click on the corresponding button and then to introduce the new customer’s data in the textfields from the opened scene.
* If the user wants to edit an existing customer, he has to click on the corresponding button and then to choose from the combo box the name of the customer whose data he wants to edit. Then, some editable textfields containing the data of the customer show up and the user can do any modification he prefers if it is valid.
* If the user wants to delete a customer from the table, he should click on the corresponding button and then to choose the customer he wants to delete from the combo box. Some textfields with the selected customer’s data will appear and a button to click on to delete the customer from the table.
* If the user wants to view the data from the table, he has to click on the corresponding button and the customers table with all the information about them shows up on the screen.

If the user chooses to manage the “products” table:

The operations are the same as in the case the user chooses to manage the customers table.

If the user chooses to manage the “orders” table:

The scene that shows up includes at the beginning a combo box for choosing a customer from “customers” table, a combo box for choosing a product to be ordered from “products” table, a textfield to enter the desired quantity and a button for computing the total price of the order. After a product is selected, two more textfields appear on the screen: one that shows the price of a unit of that product and one that shows the units of that products that are in stock. By clicking on the “Total price” button, the total price will be displayed in a textfield. If the quantity preferred by the user exists, after clicking on the “Create order” button the order will be successfully registered and included in the “orders” table.

Use cases

A use case is a definition of a specific objective that the system needs to accomplish. The two types of use-cases can be described at an abstract level (business use-case) or at an implementation-specific level (system use-case).

The **business use-cases** in my project are “choose one table to manage”, “choose an operation to perform on that table”, “fill in the input data”, “view the modified table”.

The **system use-cases** are written at a lower level of detail than the business use case and refer to the specific processes that will be carried out by different parts of the system. In my project, such use-cases are “add element to table”, “delete element from table”, “edit element”, “view table”, “create order”.

Scenarios

The main success scenario would be the following one:

-the user selects the table he wants to manage;

-if he chooses to manage the “customers” or “products” table, then he chooses the operation he wants to perform on the table;

-in case he chooses to add or to update a client he has to introduce valid data and then click on the “Add” or “Update” button;

-in case he chooses to delete a client he selects an existing client and then clicks on the “Delete” button

-in case he chooses to view the table, all the data from the table is displayed;

-if he chooses to manage the “customers” table he has to select an option from each one of the two combo boxes, then he has to choose a valid quantity, view the total price he has to pay and then click one the “Create order” button;

If the operations performed succeed, an appropriate message is displayed in a combo box. If not, an alert box with an error message is displayed and the user must change the filled in data in order to perform the desired operation.

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

When a customer or a product has to be added or updated, the introduced data is taken from the textfields and an object of the class Customer or Product is created. Then, in order to be added to or updated in the table from the database, it is checked if all the attributes of the created object are valid and then the appropriate changes are made in the table. When the user decides to delete an element from a table he has to choose a name from the combo box, name which is a String. The element to be deleted is searched in the table according to the name, and then it is deleted.

As data structures, I used arrays of type **ArrayList**. Normal arrays have the disadvantage that they can store only a fixed set of elements and they don’t grow their size at the runtime. To solve this problem, I preferred to use ArrayList, which is a resizable-array implementation of the List interface. I also used **ObservableList,** which is a list that allows listeners to track changes when they occur.

As data types, I used both primitive data types, like **int, double** and **boolean**, and non-primitive data types, such as String, Integer, Double and Classes.

I used in my application a **layered architecture** because I had to work with a database and a UI. The used layers are the following ones:

* Presentation layer – UI takes the action made by the user and sends it to the controller and at the end the result is taken from the controller and send to the UI; here, I used the **Model View Controller** approach to implement UI:
* **Model** is the central component of the pattern and it is the application’s dynamic data structure, independent of the user interface and receives user input from the controller.
* **View** is used for the UI logic of the application.
* **Controller** receives the input, inserted by the user and converts it to commands for the **Model** and **View**. It is like an interface between the other two components.
* Business logic – handles the application logic; it takes information from the user and sends it to the Database layer and vice versa.
* Data access – is the database handling layer of the application; it contains the classes containing the queries and the database connection
* Model – contains the classes mapped to the database table (the entities)

My project contains classes organized in packages the following way:

* application
  + Main – here I create the object controller of the class Controller and I call the start() method from the Controller
* businessLayer
  + validators
    - Validator
    - CustomerFieldValidator
    - EmailValidator
    - PhoneValidator
    - ProductsFieldsValidator
    - ProductUnitPriceValidator
    - UnitsInStockValidator
* CustomerBLL
* OrderBLL
* PDFGenerator
* ProductBLL
* connection
* ConnectionFactory
* dataAccessLayer
  + CustomerDAO
  + OrderDAO
  + ProductDAO
* model
  + Customer
  + Order
  + Product
* pdf
* presentationLayer
* controller
  + AddCustomerController
  + AddProductController
  + Controller
  + CustomersController
  + DeleteCustomerController
  + DeleteProductController
  + EditCustomerController
  + EditProductController
  + ObjectsController
  + OrdersController
  + ProductsController
  + TableCustomersController
  + TableProductsController
* fxmlFiles
* view
  + FxmlLoader
  + Interface

Diagram

Description automatically generated with medium confidenceDiagram

Description automatically generated

Text

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generated

Text

Description automatically generatedText

Description automatically generated with medium confidence

1. Implementation

* businessLayer
* validators
* **Validator** - this interface is implemented by the following classes that all have the aim to validate the input

public boolean validate(T t);

* **CustomerFieldValidator** – checks if the fields are not empty and if the name and the city have only alphabets, spaces or hyphens
* **EmailValidator** – checks if the email is valid
* **PhoneValidator** – checks if the phone number is valid
* **ProductFieldsValidator** – checks if the unit price is of type double, the number of units in stock is integer, and if these numbers are positive
* **ProductUnitPriceValidator** – checks if the unit price is in a required interval
* **UnitsInStockValidator** – checks if the number of units in stock is in the required interval
* **CustomerBLL**

Constructor: public CustomerBLL() - here the object of the class CustomerDAO is created and the list of validators of the object of the Customer class is populated

public Customer findCustomerById(int id)

public Customer findCustomerByName(String name)

public int insertCustomer(Customer customer)

public int updateCustomer(Customer customer)

public int deleteCustomer(Customer customer)

* + - In the methods from above the object customer is possibly validated and then transmitted as a parameter to the corresponding method from CustomerDAO class

public ArrayList<String> getNames()

public ArrayList<Customer> getCustomers()

* + - Each one of these two methods call a method from CustomerDAO to get all the names or the customers from the table
* **OrderBLL**

public int createOrder(Order order)

public Order getLastOrder()

* Each one of these two methods call a method from CustomerDAO to get all the names or the customers from the table
* **PDFGenerator**

public void generatePDF(Order order, Customer customer, Product product)

* This method is used to generate the pdf containing all the details of the order as a pdf file in the “pdf” package
* **ProductBLL**

Constructor: public ProductBLL() – here the object of the class ProductDAO is created and the list of validators of the object of the Product class is populated

public Product findProductById(int id)

public Product findProductByName(String name)

public int insertProduct(Product product)

public int updateProduct(Product product)

public int deleteProduct(Product product)

-In the methods from above the object product is possibly validated and then transmitted as a parameter to the corresponding method from ProductDAO class

public ArrayList<String> getNames()

public ArrayList<Product> getProducts()

-Each one of these two methods call a method from CustomerDAO to get all the names or the customers from the table

* connection
  + **ConnectionFactory**

private ConnectionFactory()

private Connection createConnection()

public static Connection getConnection()

public static void close(Connection connection)

public static void close(Statement statement)

public static void close(ResultSet resultSet)

* + - The methods from this class are used to create the connection with the database, to close it and to close the created statement and the result set
* dataAccessLayer
* **CustomerDAO**

public Customer findById(int customerID) – method used to find customer by id

public Customer findByName(String customerName) – method used to find customer by name

public static int insert(Customer customer) – method used to insert customer to table

public static int updateCustomer(Customer customer) – method used to update customer in table

public static int deleteCustomer(Customer customer) – method used to delete customer from table

public static ArrayList<String> selectNames() – method used to select all the customers’ names from the table

public static ArrayList<Customer> findAll() – method used to find all the customers from the table and create a list with them

* **OrderDAO**

public static int createOrder(Order order) – method used to insert a new order to the “orders” table

public static Order getLastOrder() – method used to return the order that was the last one added to the table

* **ProductDAO**

public Product findById(int productID)

public Product findByName(String productName)

public static int insert(Product product)

public static int update Product (Product product)

public static int delete Product (Product product)

public static ArrayList<String> selectNames()

public static ArrayList<Product> findAll()

* pdf – here is located the generated pdf file that represents the bill of the order
* model – it contains the entities
  + **Customer**
  + **Order**
  + **Product**
* presentationLayer
  + controller
    - **Controller**

public void start()

public void manageCustomers(ActionEvent actionEvent)

public void manageProducts(ActionEvent actionEvent)

public void manageOrders(ActionEvent actionEvent)

* + - **CustomersController**

public void addCustomer(ActionEvent actionEvent)

public void editCustomer(ActionEvent actionEvent)

public void deleteCustomer(ActionEvent actionEvent)

public void viewCustomersTable(ActionEvent actionEvent)

* + - **ProductsController**

public void addProduct(ActionEvent actionEvent)

public void editProduct(ActionEvent actionEvent)

public void deleteProduct(ActionEvent actionEvent)

public void viewProductsTable(ActionEvent actionEvent)

* + - **OrdersController**
    - **ObjectsController**

public void createTable(ObservableList<T> objects, ArrayList<TableColumn<Object, String>> columns, ScrollPane scrollPane, TableView tableView )

* + - * this method is used to create a table in the table view format containing the data of all the objects from the database table, using reflection techniques
    - **AddCustomerController**
    - **AddProductController**
    - **EditCustomerController**
    - **EditProductController**
    - **DeleteCustomerController**
    - **DeleteProductController**
    - **TableCustomersController**
    - **TableProductsController**
      * These classes contain methods that receive the input inserted by the user and converts it to commands for the business logic and data access
  + fxmlFiles
  + view
    - **FxmlLoader**

public Pane getPage(String filename)

* + - * Here, the pane corresponding to the fxml file specified by the path including the parameter filename is generated
    - **Interface**

public void start(Stage stage)

* + - * Here, the main stage is opened

Graphical user interface, application

Description automatically generatedGraphical user interface

Description automatically generated

Graphical user interface

Description automatically generated

Graphical user interface, table

Description automatically generated

Graphical user interface, application

Description automatically generated

1. Results

The customer or product added by the user appears in the table along with all its data. If a record is updated, then the modifications are applied on the table in the database. If it is deleted, it doesn’t appear in the table anymore. If an order is made, it is inserted in the table and a bill is generated.

1. Conclusions

In conclusion, this application provides the user with a friendly interface which facilitates the storage, retrieval, modification and deletion of data from a database and this could be achieved by creating a Connection between the MySQL database server and IntelliJ, creating a Statement for holding a SQL command, writing a SQL query and executing it via the Statement and Connection created, processing the query result and then closing the Statement and the Connection.

1. Bibliography

<https://docs.oracle.com/javase/9/docs/api/javafx/collections/ObservableList.html>

<file:///D:/Facultate/An%202/Sem%202/PT/Laboratory/Assignment3/ASSIGNMENT_3_SUPPORT_PRESENTATION.pdf>

<https://dzone.com/articles/layers-standard-enterprise>

<https://www3.ntu.edu.sg/home/ehchua/programming/java/JDBC_Basic.html>