

DOCUMENTATION

ASSIGNMENT 3

STUDENT NAME: Dincă Diana
GROUP: 30223

CONTENTS

| | | |
|----|---|----|
| 1. | Assignment Objective | 3 |
| 2. | Problem Analysis, Modeling, Scenarios, Use Cases..... | 3 |
| 3. | Design | 4 |
| 4. | Implementation | 5 |
| 5. | Results..... | 13 |
| 6. | Conclusions..... | 13 |
| 7. | Bibliography | 13 |

1. Assignment Objective

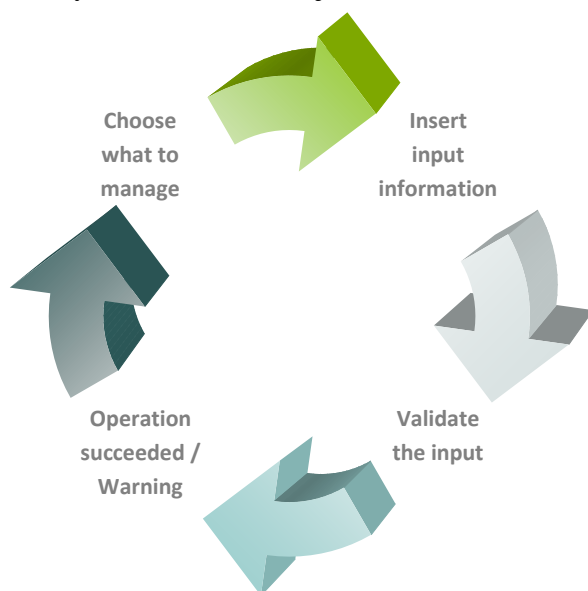
The main objective of the assignment is to design and implement an application which helps employees to manage the products, the clients and the orders of a warehouse.

To develop this project, I pursued the following sub-objectives:

- Analyze the problem and identify requirements:
The application allows users to choose between managing Clients, Products or placing Orders. After selecting a category, the user is able to perform different activities. For the result to be valid, the input must be valid. (detailed at point “2. Problem Analysis, Modeling, Scenarios, Use Cases”)
- Design the orders management application:
The application design must be intuitive and aesthetically pleasing. (detailed at point “3. Design”)
- Implement the orders management application:
The development of the code must be well-organized into packages and classes. (detailed at point “4. Implementation”)
- Test the orders management application:
For proper functionality, every function of the application must be verified with examples. (detailed at point “5. Results”)

2. Problem Analysis, Modeling, Scenarios, Use Cases

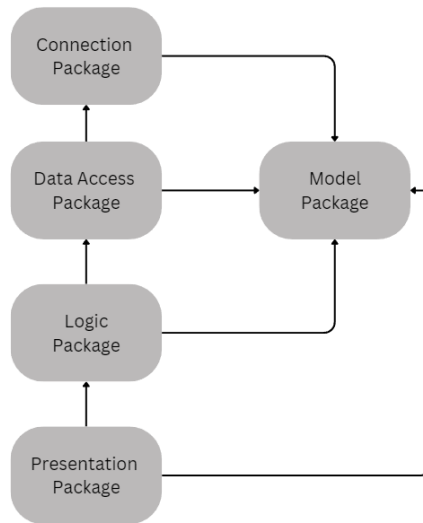
The current manual process of managing products, clients, and orders in a warehouse is inefficient and time-consuming. To address this issue, we can identify a system with the main entities: Clients, Products and Orders. The user is able to manage Clients and Products in the same way: he can add a new entity, update or delete an existing entity, or view the complete list of each entity. The user can place an order for a client, containing a specific product and the quantity. In this system, the main objective is to minimize the effort of the employees of a warehouse.



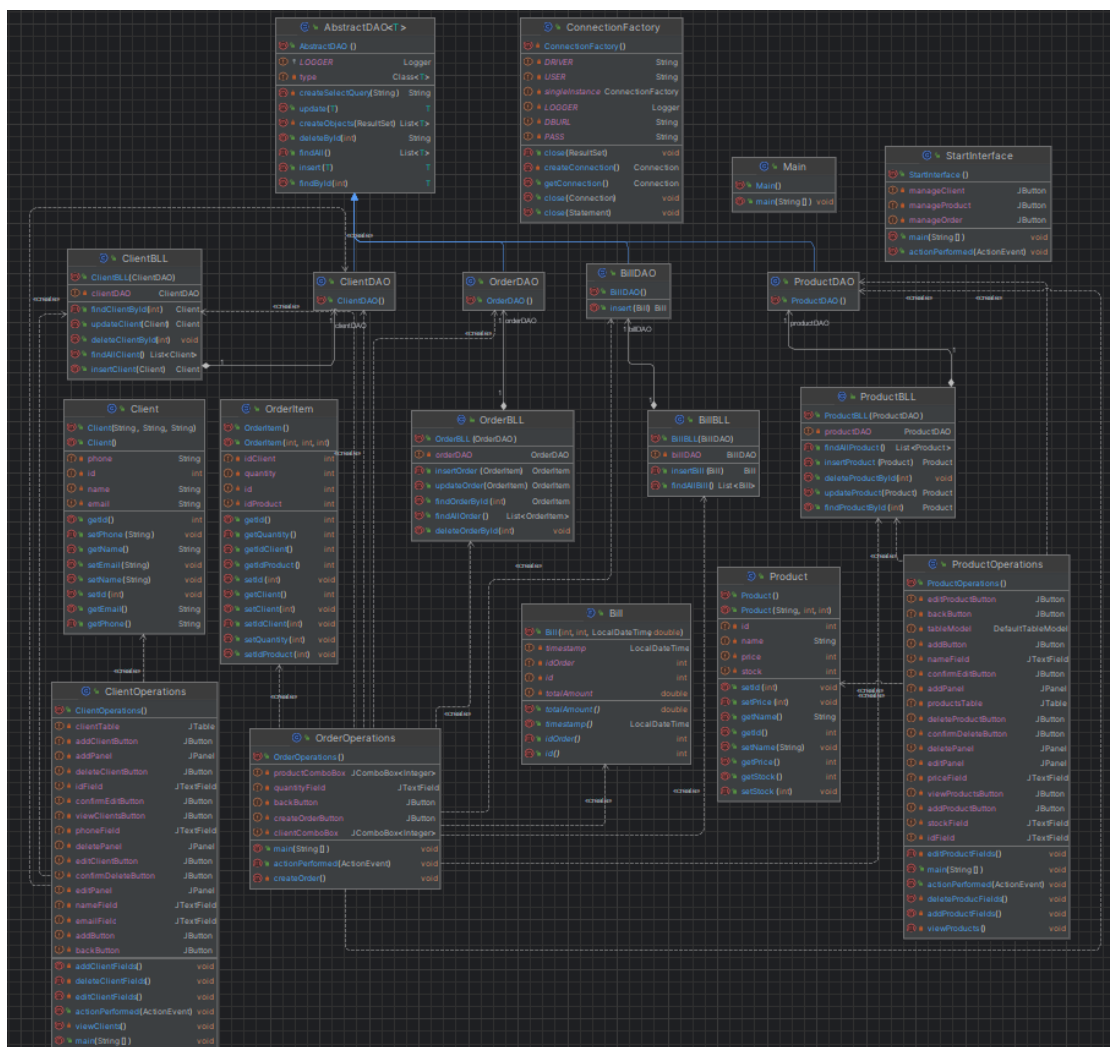
In order for the application to work, the user must provide a valid input, such as: the name of the clients must contain only letters and the phone number must contain only 10 digits; the product price and stock fields must be filled with positive numbers; the quantity of a product wanted by a client must be smaller or equal to the stock number of the item. If the input does not correspond, a specific error message will be shown.

3. Design

Package Diagram:



Classes Diagram:



The application's implementation is organized into five distinct packages:

- *model package*: within this package, you'll find the *Client*, *Product*, *OrderItem* and *Bill* classes, essential for representing and manipulating the orders for a warehouse;
- *presentation package*: this package houses the *StartInterface* class, which manages the graphical user interface for managing each entity, *ClientOperation* and *ProductOperation* classes, which allow the user to perform specific changes on the entities, and *OrderOperation* class, which allows the user to place an order for a client, with a specific quantity of a product;
- *data access package*: contains the *AbstractDAO* class that creates queries using Reflection, and it's implemented by *ClientDAO*, *ProductDAO*, *OrderDAO* and *BillDAO* classes.
- *logic package*: this package includes *ClientBLL*, *ProductBLL*, *OrderBLL*, *BillBLL* classes implement all CRUD methods using DAO classes.

4. Implementation

ConnectionFactory Class- makes the connection with SQL possible.

Client Class- stores the data for each Client: id, name, email, phone.

Product Class- stores the data for each Product: id, name, price, stock.

OrderItem Class- stores the data for each Order: id, idClient, idProduct, quantity.

Bill Class- is an immutable class that stores the data for each Bill generated after ordering a product: id, idOrder, timestamp, total amount of money spend.

AbstractDAO Class- creates queries using Reflection, and it's extended by *ClientDAO*, *ProductDAO*, *OrderDAO* and *BillDAO* classes.

- *createSelectQuery*- creates a Select query for SQL;
- *deleteById*- deletes an object by its id;
- *findAll*- returns a list of all the objects in one table;
- *findById*- finds in the database an object based on its id;
- *createObjects*- manages to create objects based on a resulted set of data;
- *insert*- constructs an SQL INSERT query;
- *update*- constructs an SQL UPDATE query;

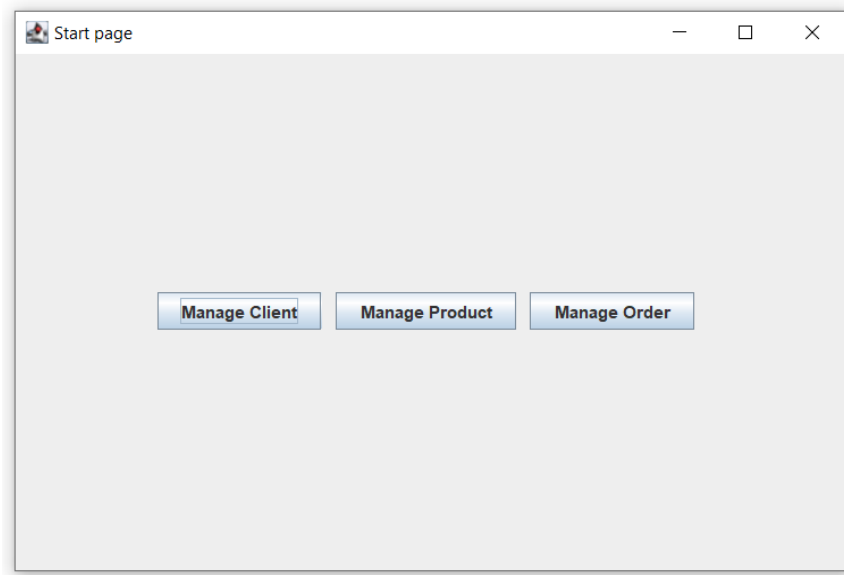
ClientBLL Class- implements all CRUD methods for Clients using DAO classes.

ProductBLL Class- implements all CRUD methods for Products using DAO classes.

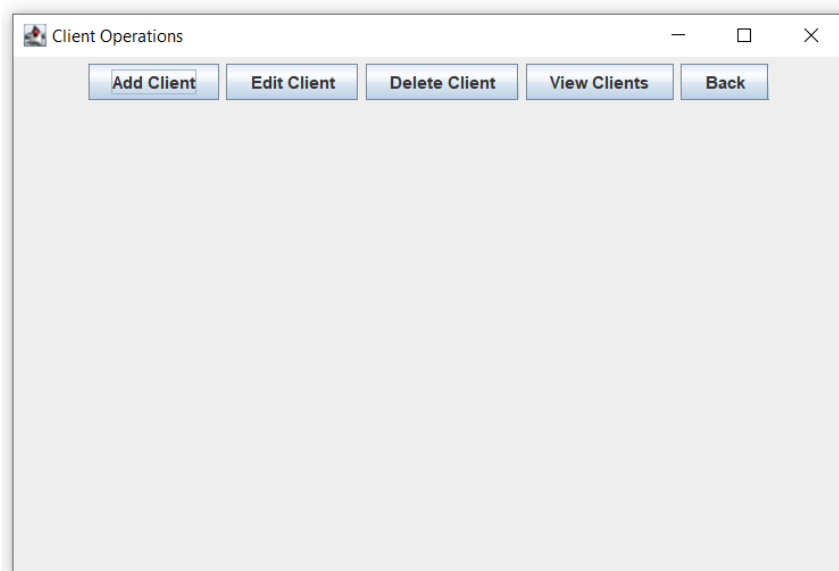
OrderItemBLL Class- implements all CRUD methods for Orders using DAO classes.

Bill ClassBLL- implements insert methods for Bills using DAO classes.

StartInterface Class- allows the user to select which subjects he wants to manage: Clients, Products or Orders.



ClientOperations Class- implements a window for client operations: add new client, edit client, delete client, view all clients in a table.



- When *Add Client* button is pressed, it uncovers 3 labels with 3 text fields, where the user can write the necessary data for a client. The method checks if the input is valid: the name must contain only letters and the phone must contain only 10 digits, otherwise an error message will be shown.

The 'Client Operations' window features a title bar with standard window controls. Below the title bar, there are five buttons: 'Add Client', 'Edit Client', 'Delete Client', 'View Clients', and 'Back'. The main area of the window contains a form with three labels: 'Name:', 'Email:', and 'Phone:'. Each label is followed by a text input field. The 'Name' field contains 'George', the 'Email' field contains 'george123@yahoo.com', and the 'Phone' field contains '0752023862'. At the bottom of the form, there is a blue 'Add' button.

| | |
|--------|---------------------|
| Name: | George |
| Email: | george123@yahoo.com |
| Phone: | 0752023862 |

Add

- When the *Edit Client* button is pressed, it uncovers 4 labels with 4 text fields, where the user can write the necessary data for a client. The method checks the id of the Client, and it replaces the existing data with the data provided in the other 3 text fields. All of the text fields are being checked after making an update.

The 'Client Operations' window is shown with the 'Edit Client' button highlighted. A 'Success' dialog box is displayed in the center, indicating that the client was updated successfully. The dialog box has a title bar with a close button, an information icon, and the text 'Updated successfully'. Below the text is an 'OK' button. In the background, the 'Edit Client' form is visible, showing four labels: 'Client ID:', 'Name:', 'Email:', and 'Phone:'. Each label is followed by a text input field. The 'Client ID' field contains '15', the 'Name' field contains 'Marcel', the 'Email' field contains 'marcel9876@yahoo.com', and the 'Phone' field contains '0772932042'. At the bottom of the form, there is a blue 'Confirm' button.

Success

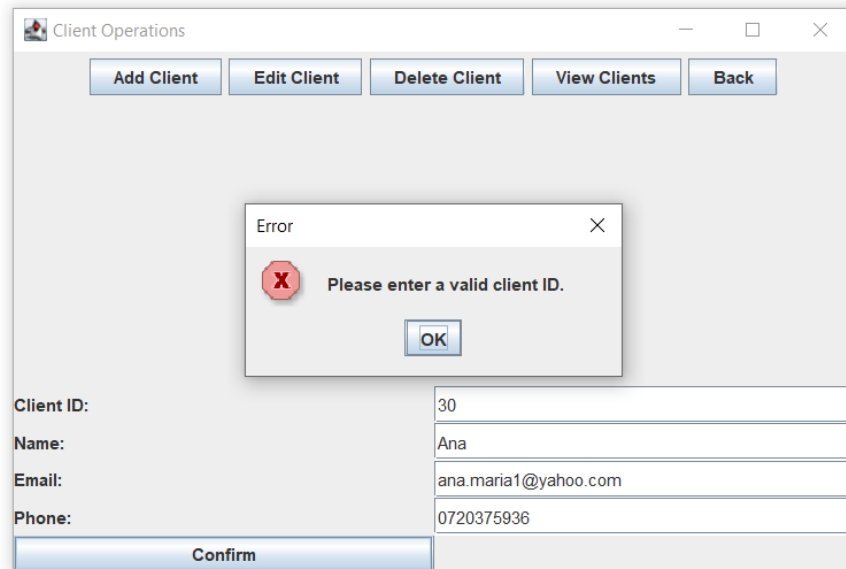
Updated successfully

OK

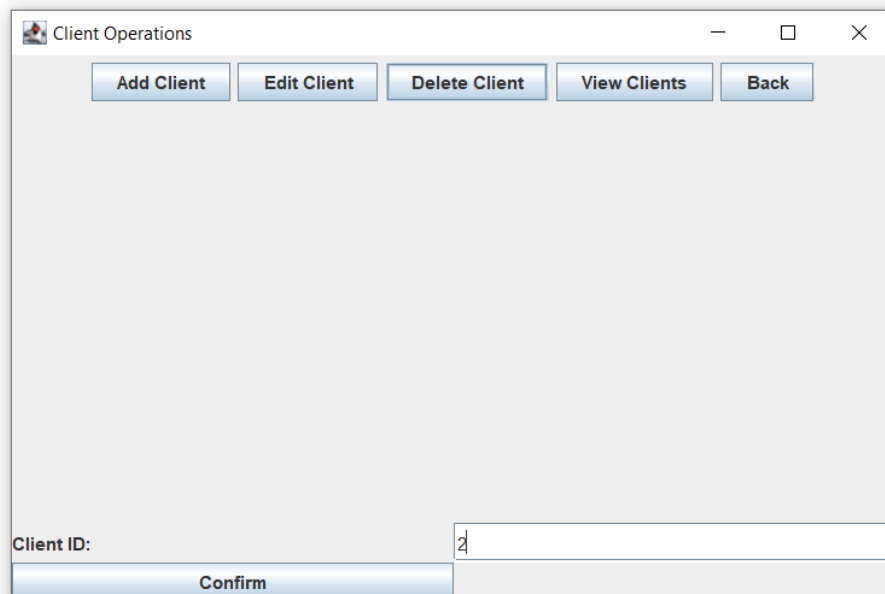
| | |
|------------|----------------------|
| Client ID: | 15 |
| Name: | Marcel |
| Email: | marcel9876@yahoo.com |
| Phone: | 0772932042 |

Confirm

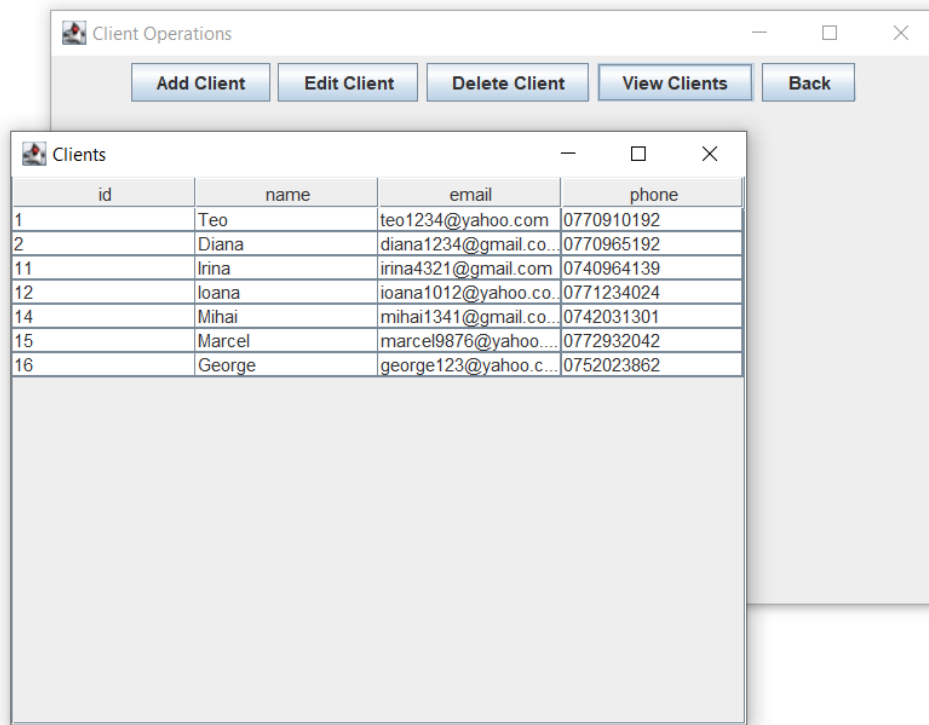
- If the id dose not correspond to a client in the database, there will be an error message.



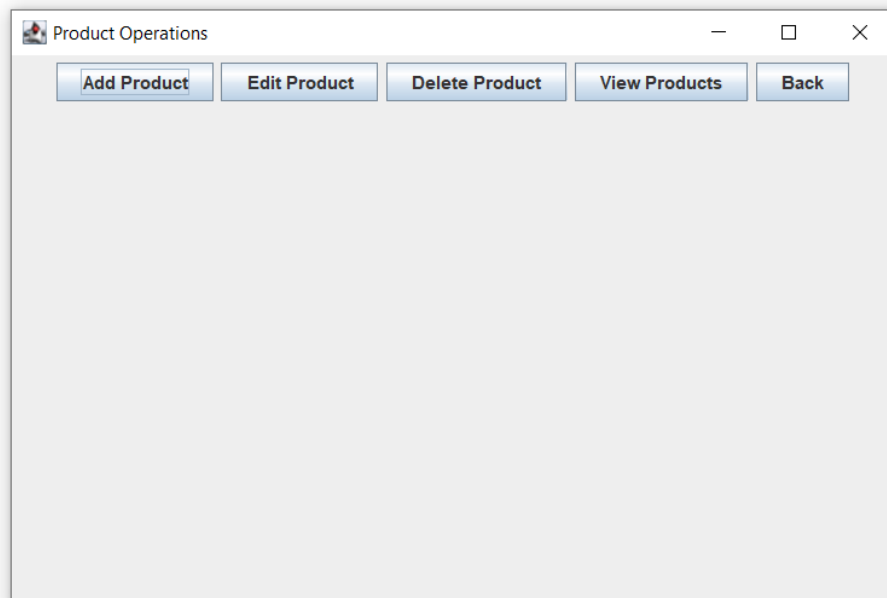
- When the *Delete Client* button is pressed, it uncovers 1 label with 1 text field, where the user can input the id of the Client, soon to be deleted. The method validates the input, the id must be an int, and it must correspond to an existing Client. If the id dose not correspond to a client in the database, there will be an error message, like the one above.



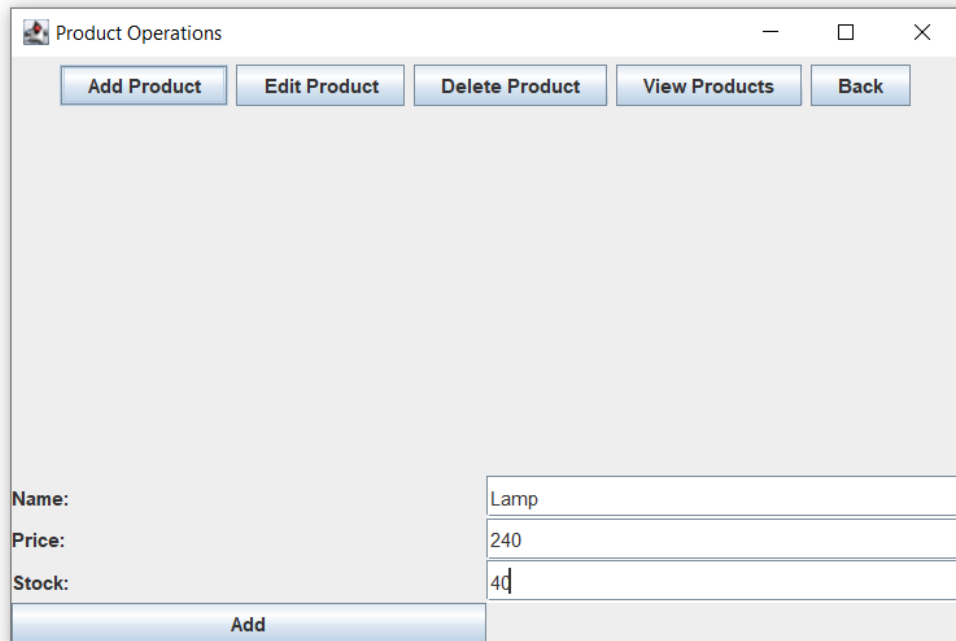
- When the *View Clients* button is pressed, it uncovers a table containing all Clients and all their fields: id, name, email, phone.



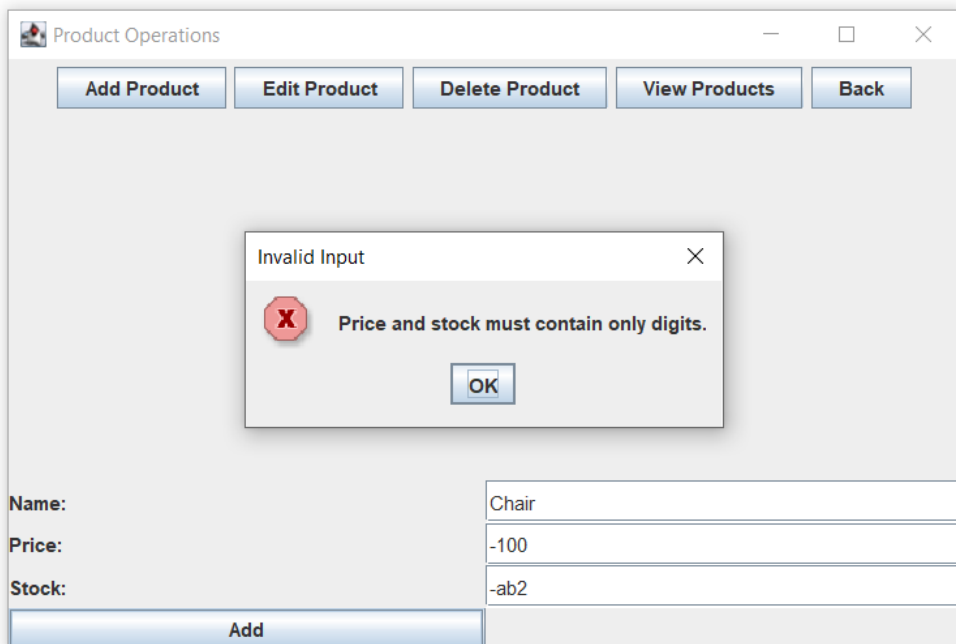
ProductOperations Class- implements a window for product operations: add new product, edit product, delete product, view all products in a table.



- When the *Add Product* button is pressed, it uncovers 3 labels with 3 text fields, where the user can write the necessary data for a Product (name, price, stock). The method checks if the input is valid: the name must contain only letters, the price and the stock must contain only digits, otherwise an error message will be shown.



The screenshot shows a window titled "Product Operations" with a standard Windows-style title bar (minimize, maximize, close buttons). Below the title bar is a horizontal bar containing five buttons: "Add Product", "Edit Product", "Delete Product", "View Products", and "Back". The "Add Product" button is highlighted. Below this bar, the "Add" button is visible. The form contains three labels and text fields: "Name:" with the value "Lamp", "Price:" with the value "240", and "Stock:" with the value "40".



The screenshot shows the same "Product Operations" window, but with an error message displayed. The error message is in a small dialog box titled "Invalid Input" with a close button. It contains a red "X" icon and the text "Price and stock must contain only digits." Below the text is an "OK" button. The form data in the background is: "Name:" with the value "Chair", "Price:" with the value "-100", and "Stock:" with the value "-ab2".

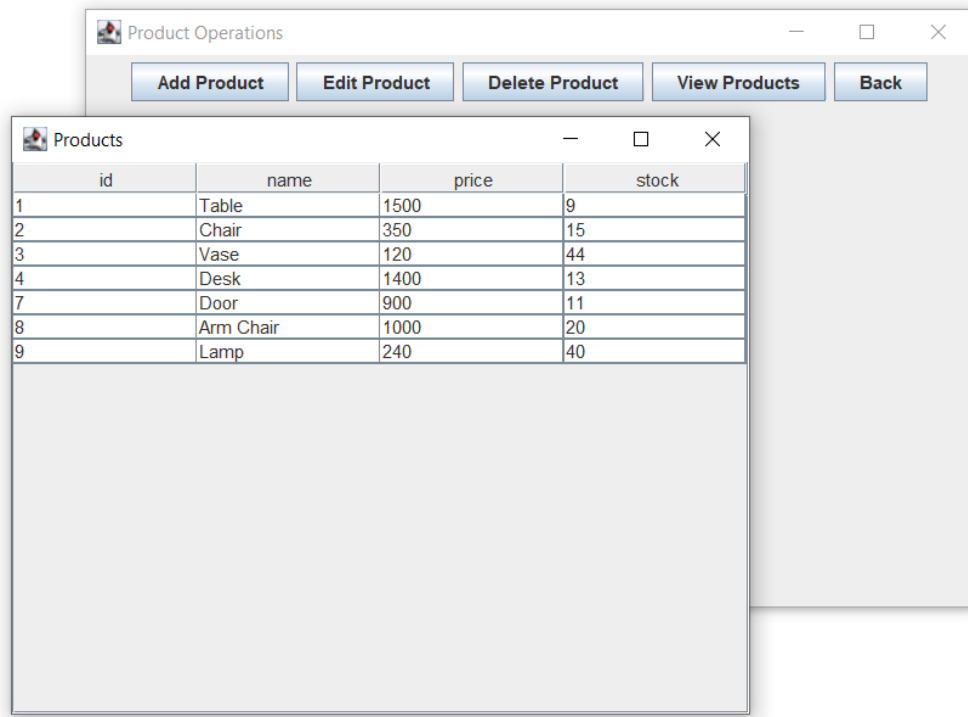
- When the *Edit Product* button is pressed, it uncovers 4 labels with 4 text fields, where the user can write the necessary data for a Product (id, name, price, stock). The method checks the id of the Product, and it replaces the existing data with the data provided in the other 3 text fields. All the text fields are being checked after making an update and if the input does not correspond, there will be an error message.

The screenshot shows a window titled "Product Operations" with standard Windows window controls (minimize, maximize, close). At the top, there are five buttons: "Add Product", "Edit Product", "Delete Product", "View Products", and "Back". The "Edit Product" button is highlighted. Below the buttons, the form is divided into two columns. The left column contains four labels: "Product ID:", "Name:", "Price:", and "Stock:". The right column contains four corresponding text input fields. The "Product ID" field contains the value "3". The "Name" field contains the value "Lamp". The "Price" field contains the value "140". The "Stock" field contains the value "3". At the bottom of the form, there is a "Confirm" button.

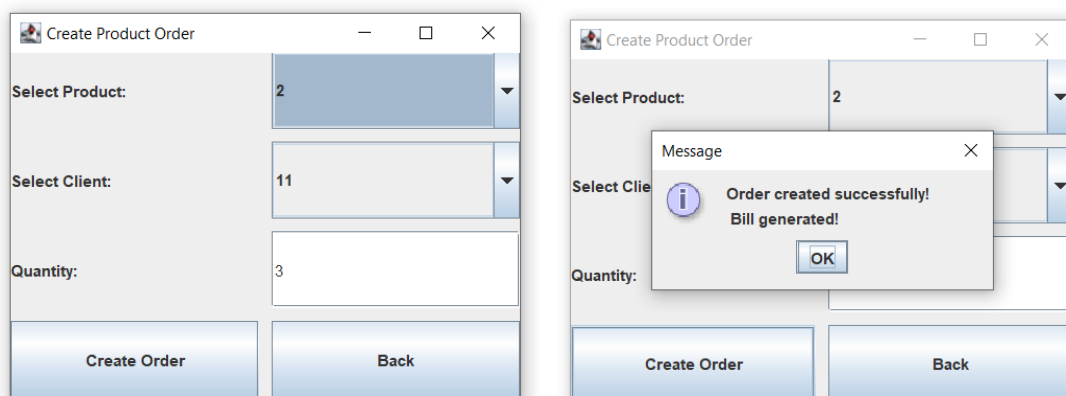
- When the *Delete Product* button is pressed, it uncovers 1 label with 1 text field, where the user can input the id of the Product, soon to be deleted. The method validates the input, the id must be an int, and it must correspond to an existing Product.

The screenshot shows the same "Product Operations" window. The "Delete Product" button is now highlighted. The form has been updated to show a single label "Product ID:" on the left and a single text input field on the right. The input field contains the value "3". The "Confirm" button remains at the bottom.

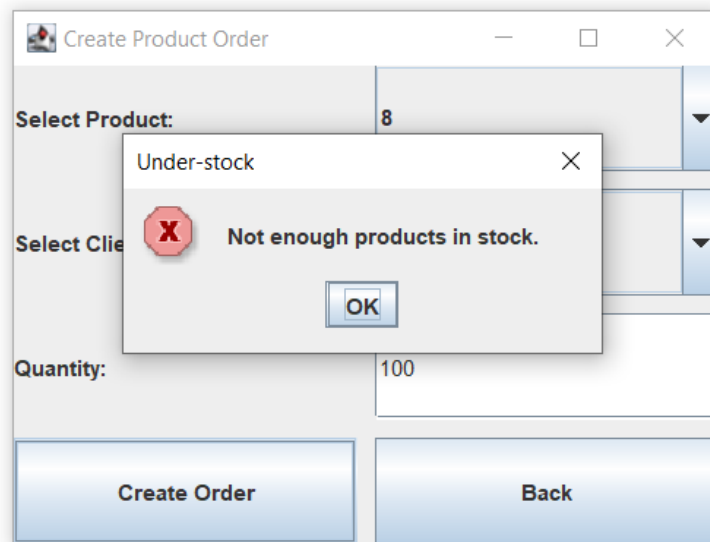
- When the *View Products* button is pressed, it uncovers a table containing all Products and all their fields: id, name, price, stock.



OrderOperations Class- implements a window for creating product orders. The user will be able to select an existing product, select an existing client, and insert a desired quantity for the product to create a valid order. A bill will be generated and written in the database.



- In case there are not enough products, an under-stock message will be displayed. After the order is finalized, the product stock is decremented.



5. Results

I have conducted many tests. First, I tested the validation functionality and performance by introducing invalid data. Secondly, I tested various cases on adding, updating and deleting entities, checking with the view option if the table changed in the right way.

6. Conclusions

The development of an application Orders Management for processing client orders for a warehouse highlighted the inefficiencies of manual processes and the benefits of implementing automated solutions.

I've gained insights into utilizing Reflection in Java, for inspecting and manipulating classes, methods, and fields. Also, I have deepened my understanding of database design principles, SQL queries, and data manipulation techniques.

Future developments could focus on implementing security measures to protect data (authentication, authorization), order tracking or even inventory forecasting.

In conclusion, this project has provided valuable hands-on experience in database integration, Reflection in Java, and use case modeling.

7. Bibliography

- <http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>
- https://gitlab.com/utcn_dsrl/pt-reflection-example
- <http://tutorials.jenkov.com/java-reflection/index.html>
- <https://www.baeldung.com/javadoc>