DOCUMENTATION

ASSIGNMENT *3*

STUDENT NAME: Iulia-Olivia Cipleu

GROUP: 30423

# CONTENTS

[1. Assignment Objective 3](#_Toc128043139)

[2. Problem Analysis, Modeling, Scenarios, Use Cases 3](#_Toc128043140)

[3. Design 5](#_Toc128043141)

[4. Implementation 7](#_Toc128043142)

[5. Results 7](#_Toc128043143)

[6. Conclusions 13](#_Toc128043144)

[7. Bibliography 13](#_Toc128043145)

# Assignment Objective

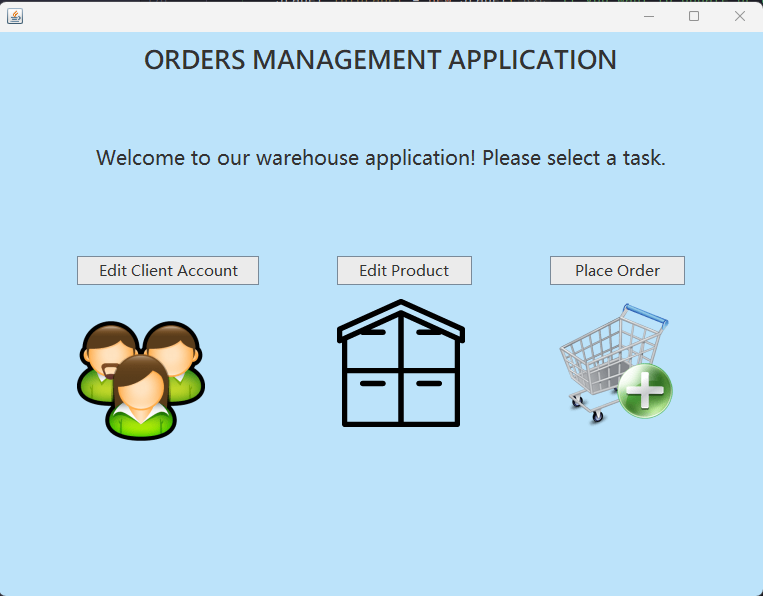
The aim of this assignment is the implement an application for managing a warehouse ordering system using databases.

On the other hand, the second aim of this work is to use the reflection technique.

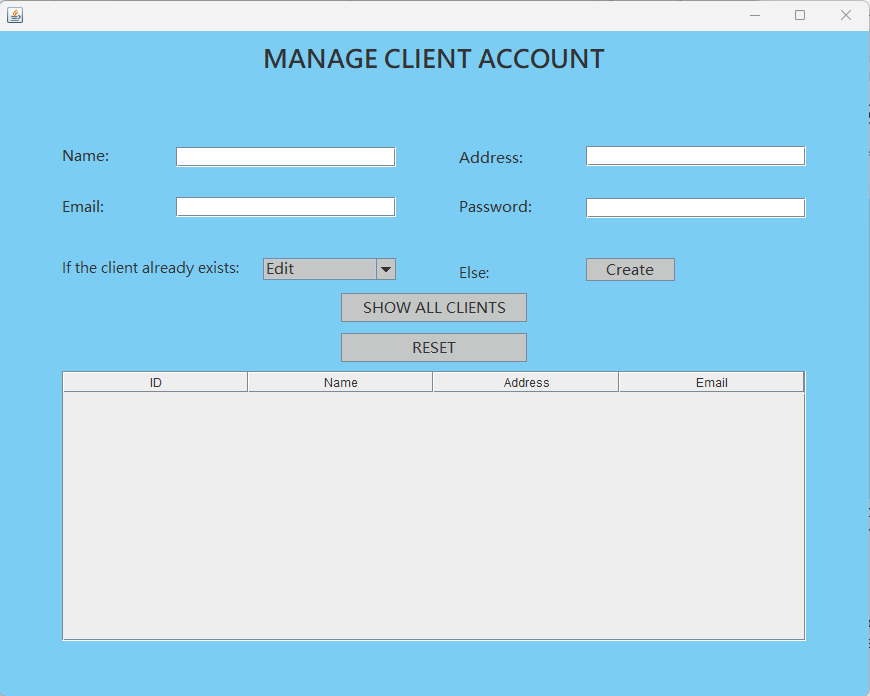
Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program.

# Problem Analysis, Modeling, Scenarios, Use Cases

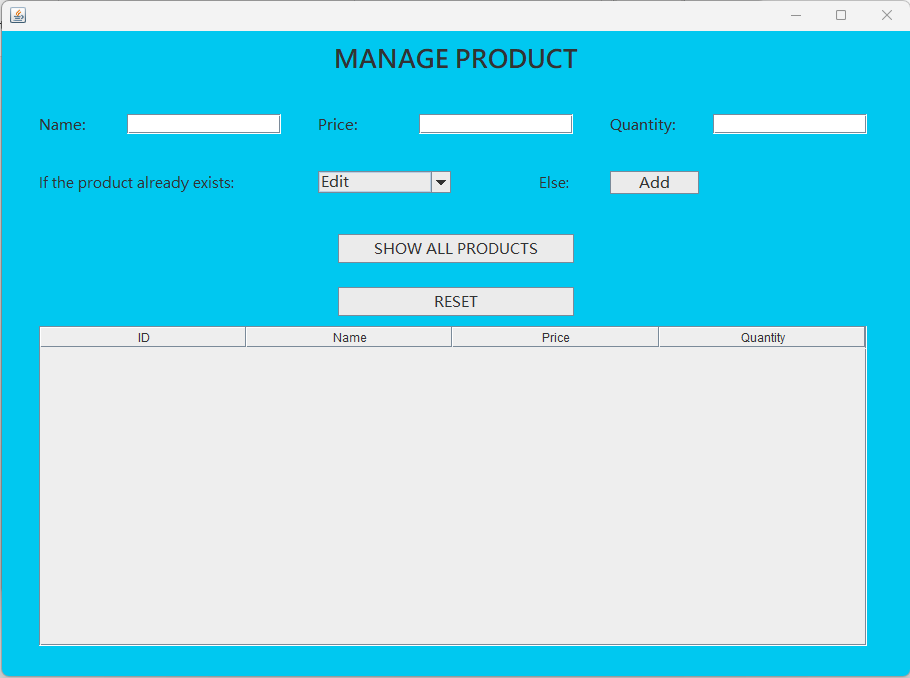
The user interacts with a GUI interface. In the Home Page, they choose the task they want to perform: manage client account, manage product or manage order.

**

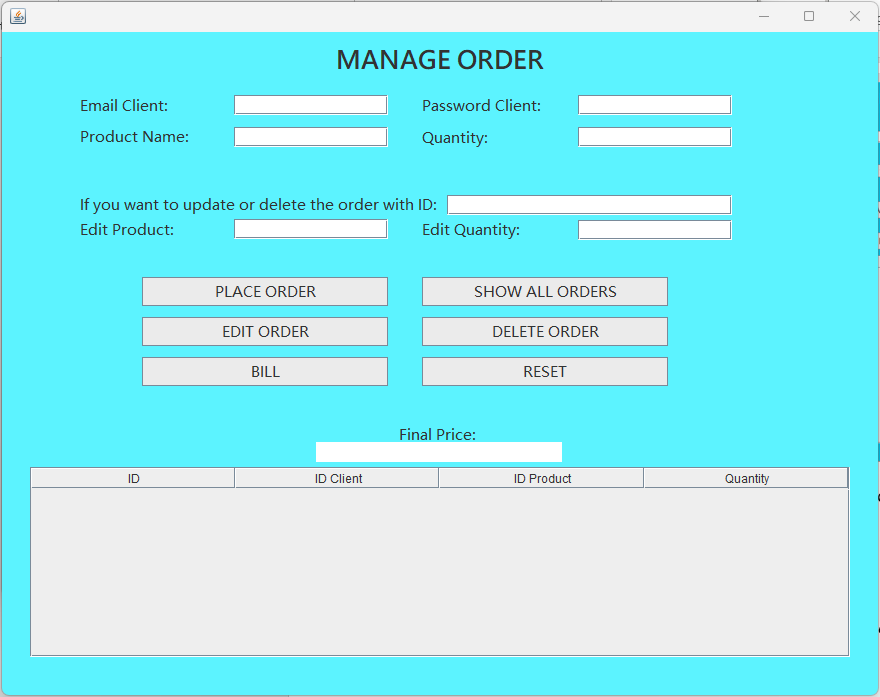
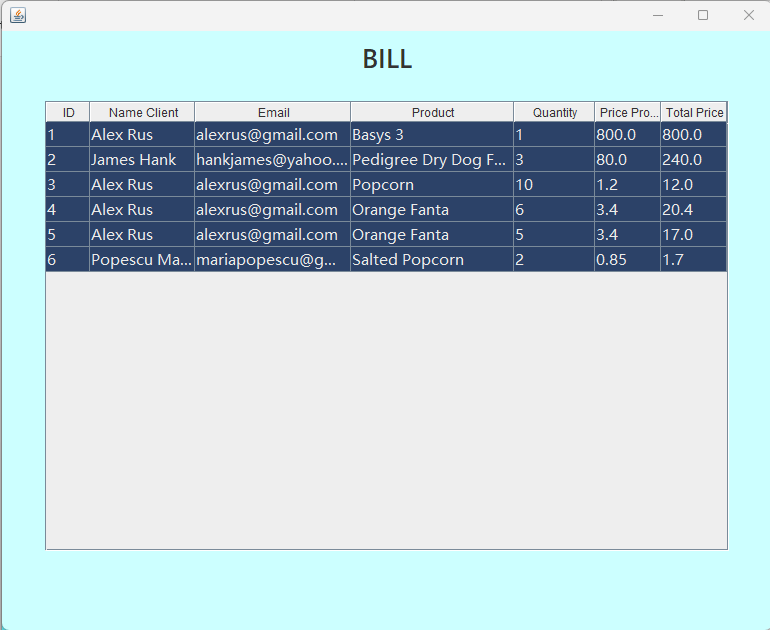
If they choose the first option, a new page will appear. Here, the user can add, edit or delete an account defined by the information the type in the text fields, or just visualize all the clients in the database. There is also a button for reset.



In the Product page, the user can do the same things as for accounts.



If the user chooses the Order Page, they can place, edit, delete an order, or see all the active orders that exist in the database. Also, the is an option for Bill, which is an history for the orders made, even the ones updated or annulated.

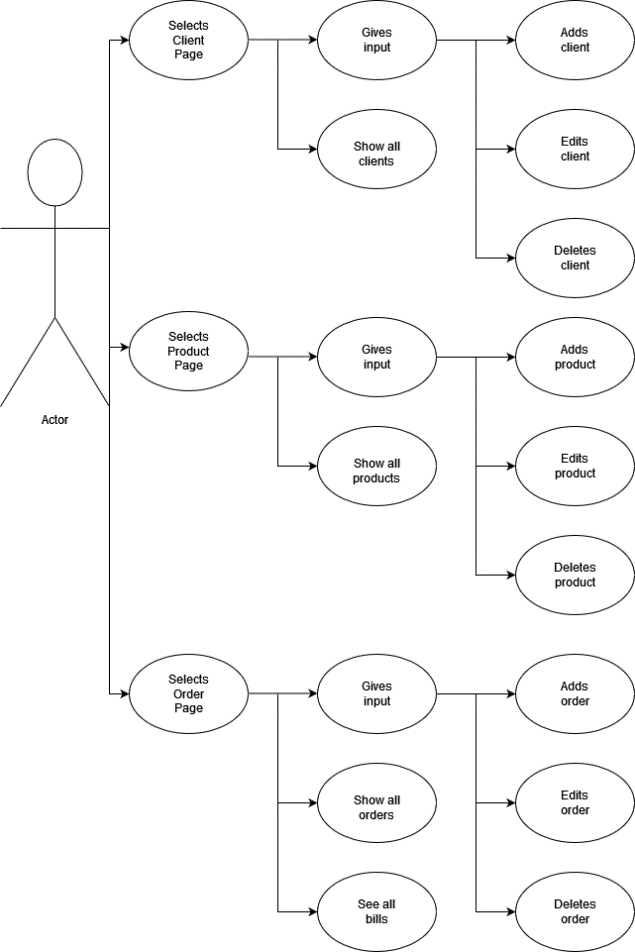
Use Case: Navigate through the site

Primary Actor: User

Alternative cases which will generate an error pop-out message:

* The user writes invalid characters (string characters in the fields destinated only for numbers)
* Incorrect password
* Non-existent account, product, order given to be updated/deleted
* Empty fields for create

Here is the Use Cases Diagram:



# Design

3.1 OOP Design of Application

OOP principles are rules for how to properly build programs within an object-oriented paradigm. The four main principles are encapsulation, abstraction, inheritance, and polymorphism.

Encapsulation says that each object should keep its internal state private, so that other objects can only access it when programmers explicitly provide that permission.

Abstraction says that the internal workings of the object should be hidden as well. Other objects ca not go inside the object and change its state — or even see its state — unless the programmer has created a method. This ensures that objects always interact with each other in a predictable fashion, even when the program becomes large and complex.

Inheritance is an OOP method to deal with a large number of objects that are similar, but not identical. Objects inherit all their traits from parent objects, which can be supplemented by their own traits.

Polymorphism enables treating multiple child classes in the same way, by using the same methods within the child classes.

3.2 UML Diagrams

The UML Class diagram is a graphical notation used to construct and visualize object-oriented systems. A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's:

- classes,

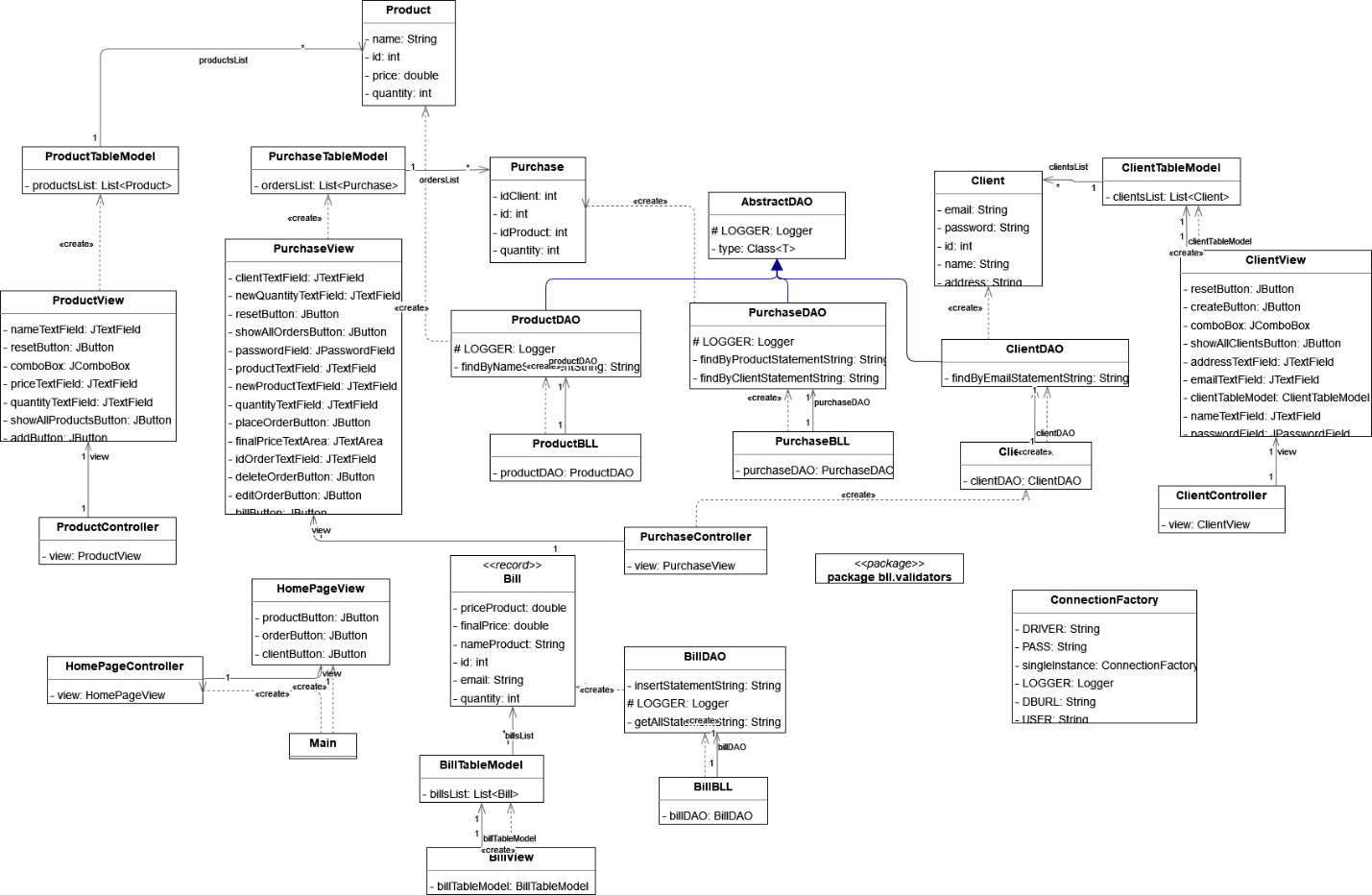
- their attributes,

- operations (or methods),

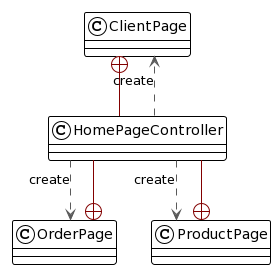
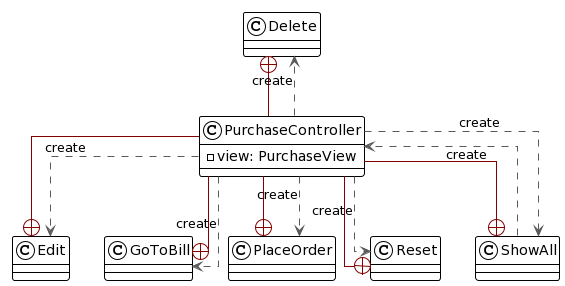
- and the relationships among objects.

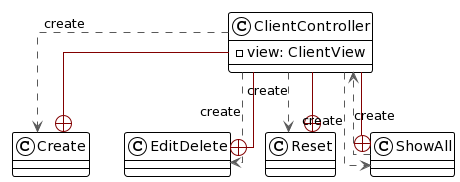
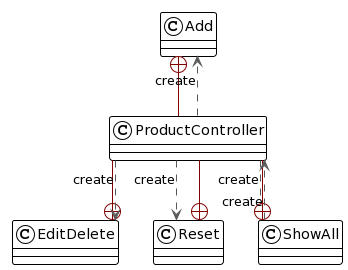
A Class is a blueprint for an object. Objects and classes go hand in hand. We can't talk about one without talking about the other. And the entire point of Object-Oriented Design is not about objects, it is about classes, because we use classes to create objects. So, a class describes what an object will be, but it is not the object itself.

In fact, classes describe the type of objects, while objects are usable instances of classes. Each Object was built from the same set of blueprints and therefore contains the same components (properties and methods). The standard meaning is that an object is an instance of a class and object - Objects have states and behaviors.



Here are the controllers with their inner classes:



*3.3 Used Data Structures*

The data types structures used in this project are both primitive (int), non-primitive (String, ArrayList, Logger).

A Logger object is used to log messages for a specific system or application component. Loggers are normally named, using a hierarchical dot-separated namespace. Logger names can be arbitrary strings, but they should normally be based on the package name or class name of the logged component, such as java.net or javax.swing. In addition, it is possible to create "anonymous" Loggers that are not stored in the Logger namespace.

Logger objects may be obtained by calls on one of the getLogger factory methods. These will either create a new Logger or return a suitable existing Logger. It is important to note that the Logger returned by one of the getLogger factory methods may be garbage collected at any time if a strong reference to the Logger is not kept.

* 1. Defined Interfaces

The User Interface, presented above, contains basic GUI Elements:

* Frame
* Panels
* Labels
* Text Fields
* Text Panes
* Text Area
* Scroll
* Buttons
* Combo Boxes
* Tables

The Validator Interface is implemented by the “EmailValidator”, which verifies if an email has the correct form.

# Implementation

*4.1* *Client*

This class represent the client account, having as fields: ID, name, email address, password, address. It has 3 constructors, which are used in different contexts. Also, it has getters and setters.

*4.2 Product*

This class represent the product, having as fields: ID, name, price, quantity. It has 3 constructors, which are used in different contexts. Also, it has getters and setters.

*4.3 Purchase*

This class represent the order, having as fields: ID, ID client, ID product, quantity. It has 3 constructors, which are used in different contexts. Also, it has getters and setters.

*4.4 Bill*

The Bill class represents a bill in the system. It contains information about the bill, such as the ID, name of the client, email of the client, name of the product, quantity, price of the product, and final price. It has two constructors, one that constructs a new bill with default values for all fields, and another that constructs a bill with the specified values for all fields. The class is implemented using the record feature in Java, which provides concise syntax for defining classes that are mainly used to store data.

*4.5 ConnectionFactory*

This is the implementation of the ConnectionFactory class, which is used to create a connection to a MySQL database and provide methods to close the connection, statement, and result set.

The class contains a private constructor, which ensures that only one instance of the class can be created. It also contains a createConnection() method, which creates a new connection to the database using the driver, database URL, username, and password specified in the class constants.

The class also contains three static methods to close the connection, statement, and result set. These methods check whether the object passed as an argument is null before attempting to close it, and log a warning message if an error occurs while closing the object.

Overall, this class provides a simple way to create and manage a database connection in a Java application.

*4.6 Validator*

This package contains an interface called `Validator` which defines a method for validating an object of a specified type. Any class that implements this interface must provide an implementation for the `validate` method which takes an object of the specified type and throws an `IllegalArgumentException` if the object is not valid according to some criteria. This interface can be used to define custom validation logic for various classes in the system, such as input validation for user input or data validation for database objects.

*4.7 EmailValidator*

This class is used to validate an email address of a Client object. It uses a regular expression pattern to determine whether an email address is valid or not.

*4.8 AbstractDAO*

This class represents a generic Data Access Object (DAO) for accessing and manipulating data stored in a database table. It provides basic CRUD operations such as insert, select, update and delete. The class uses reflection to automatically generate SQL queries based on the fields and methods of the entity type, which is passed as a generic parameter to the class. The DAO uses JDBC to execute SQL queries against the database and return results as a list of objects. The DAO is designed to be extended by concrete DAO implementations that specify the entity type and any additional operations specific to that type.

*4.9 ClientDAO*

This is a Java code for a data access object (DAO) that manages the persistence of client data in a database. It contains a class named `ClientDAO` that extends `AbstractDAO` and has a method named `findByEmail()` which takes an email address as a parameter and returns a `Client` object representing the client whose email matches the given email. The DAO uses JDBC to connect to the database, prepare and execute a SQL query to find the client by email, and return the client data as a `Client` object. It also handles exceptions and closes resources appropriately using try-catch-finally blocks.

*4.10 ProductDAO*

‘ProductDAO’ provides methods for interacting with the database table "product". It extends the ‘AbstractDAO’ class and inherits its methods and properties. The ‘ProductDAO’ class has a method called ‘findByName’, which finds a product in the database by its name. It takes a name parameter as input and returns the Product object found by the name, or null if no such product exists in the database.

The’ findByName’ method connects to the database using the’ ConnectionFactory’ class, creates a prepared statement with the SQL statement for finding a product by name, sets the name parameter on the prepared statement, executes the query, and retrieves the’ ResultSet’. If the ‘ResultSet’ contains a row, it extracts the product details from the row and creates a Product object with the retrieved details. Finally, the method closes the ‘ResultSet’, prepared statement, and database connection before returning the Product object. If an exception occurs during the execution of the method, it logs a warning message with the Logger object.

*4.11 PurchaseDAO*

The `Purchase` class allows reading and writing `Purchase` objects to and from a database. The `PurchaseDAO` class extends the `AbstractDAO` class and provides two methods for finding purchases: `findByClient` and `findByProduct`. Both methods take an ID as a parameter and return a list of `Purchase` objects that match the ID. The `findByClient` method finds all purchases made by a given client, while the `findByProduct` method finds all purchases of a given product.

The `PurchaseDAO` class contains two static string fields that define the SQL queries used to find purchases by client ID or product ID. The `findByClient` and `findByProduct` methods both use a `PreparedStatement` to execute the appropriate SQL query with the ID passed as a parameter. They both create a new `Purchase` object for each row returned by the query, add the new object to a list, and return the list when all rows have been processed.

Like the other DAO classes, `PurchaseDAO` uses a `ConnectionFactory` to obtain a connection to the database and to close the connection, `ResultSet`, and `PreparedStatement` objects when they are no longer needed. The class also defines a logger that is used to log any warnings or errors that occur during database operations.

*4.12 BillDAO*

This class handles communication with the database for the `Bill` model. It contains two methods: `insert` and `findAll`.

The `insert` method inserts a new `Bill` object into the database and returns the ID of the newly inserted `Bill` or -1 if the insertion failed. It takes a `Bill` object as a parameter and uses a SQL `INSERT` statement with prepared statements to insert the `Bill` object's data into the database. It also retrieves the ID of the newly inserted `Bill` using the `getGeneratedKeys` method of the `PreparedStatement` object.

The `findAll` method retrieves all the `Bill` objects from the database and returns them as an `ArrayList`. It uses a SQL `SELECT` statement with prepared statements to retrieve all the `Bill` objects from the database and then creates a `Bill` object for each row returned from the database. Finally, it returns an `ArrayList` of all the `Bill` objects.

This class also contains a logger to log any warnings that occur during execution. It uses the `ConnectionFactory` class to get a database connection and close the connection and statements after execution.

*4.13 ClientBLL*

This class provides methods to interact with the `ClientDAO` class and perform CRUD (Create, Read, Update, Delete) operations on client data. Here's a summary of the methods provided by this class:

- `public ClientBLL() throws SQLException`: A constructor method that initializes the `clientDAO` instance variable by creating a new `ClientDAO` object. It throws an SQLException if there's an error creating the `ClientDAO` object.

- `public Client findClientById(int id) throws SQLException, NoSuchElementException`: This method retrieves a client from the database by their ID. It takes an integer ID as a parameter and returns a `Client` object. It throws an `SQLException` if there's an error executing the SQL query and a `NoSuchElementException` if the client with the given ID is not found in the database.

- `public Client findClientByEmail(String email) throws SQLException, NoSuchElementException`: This method retrieves a client from the database by their email address. It takes an email address as a parameter and returns a `Client` object. It throws an `SQLException` if there's an error executing the SQL query and a `NoSuchElementException` if the client with the given email address is not found in the database.

- `public int insertClient(Client client) throws SQLException`: This method inserts a new client into the database. It takes a `Client` object as a parameter and returns the ID of the inserted client. It throws an `SQLException` if there's an error executing the SQL query.

- `public void deleteClient(Client client) throws SQLException`: This method deletes a client from the database. It takes a `Client` object as a parameter and returns nothing. It throws an `SQLException` if there's an error executing the SQL query.

- `public void editClient(Client client, int oldId) throws SQLException`: This method updates a client in the database. It takes a `Client` object and an integer ID as parameters and returns nothing. The integer ID is used to identify the client to update in the database. It throws an `SQLException` if there's an error executing the SQL query.

- `public List<Client> getAllClients() throws NoSuchElementException`: This method retrieves a list of all clients in the database. It returns a list of `Client` objects. It throws a `NoSuchElementException` if there are no clients in the database.

*4.14 ProductBLL*

This is a Java class called "ProductBLL" that represents the business logic layer for the "Product" model. It provides methods to access and manipulate product data from the database.

The class has a constructor that initializes a "ProductDAO" object. The "ProductDAO" is a data access object that provides methods to access the database.

The class has methods to find a product by its name or ID, insert a new product into the database, delete a product from the database, update a product in the database, and retrieve all products from the database.

The methods throw SQLException and NoSuchElementException exceptions if there is an error with the database or if no product is found. The insertProduct method returns the ID of the inserted product, and if the insertion was unsuccessful, it returns -2. The getAllProducts method returns a list of all products and throws a NoSuchElementException if there are no products in the database.

*4.15 PurchaseBLL*

This is a Java class for the ‘PurchaseBLL’ (Business Logic Layer) which serves as an intermediary between the Purchase model and the ‘PurchaseDAO’ (Data Access Object). It contains methods to perform CRUD operations (create, read, update, delete) on purchases, and throws ‘SQLException’ and ‘NoSuchElementException’ if there are any problems with the database connection or if the purchase cannot be found.

The class contains a constructor that initializes the ‘PurchaseDAO’ object, and several public methods for finding purchases by ID, client, or product, inserting a new purchase into the database, deleting a purchase from the database, updating a purchase in the database, and getting all purchases in the database.

The ‘findOrderById()’ method finds a purchase with the specified ID, throws a ‘NoSuchElementException’ if the purchase is not found, and returns the Purchase object if found. The ‘findOrderByClient()’ and ‘findOrderByProduct()’ methods find all purchases made by a specific client or containing a specific product respectively, and return a List of Purchase objects or null if no purchases are found.

The ‘insertOrder()’ method inserts a new Purchase object into the database and returns its ID. The ‘deleteOrder()’ method deletes a Purchase object from the database. The ‘editOrder()’ method updates a Purchase object in the database and takes the old ID of the purchase to be updated as a parameter.

The ‘getAllOrders()’ method returns a List of all Purchase objects in the database and throws a ‘NoSuchElementException’ if there are no purchases in the database.

*4.16 BillBLL*

The ‘BillBLL’ class provides business logic methods for interacting with Bill objects in the database. The class has two methods:

* ‘getAllBills()’: This method retrieves a list of all Bill objects in the database using the ‘BillDAO’ object, and returns the list. If there are no Bill objects in the database, a ‘NoSuchElementException’ is thrown.
* ‘insertBill(Bill bill)’: This method inserts a new Bill object into the database using the ‘BillDAO’ object, and returns the ID of the newly inserted Bill object.

*4.17 ClientTableModel*

This is a class definition for a table model used to display a list of clients in a JTable. The `ClientTableModel` extends the `AbstractTableModel` class, which is an abstract class that provides basic implementations for the `TableModel` interface methods.

The class has a `clientsList` instance variable, which is a list of `Client` objects to display in the table. It has three implemented methods: `getRowCount()`, `getColumnCount()`, and `getValueAt()`.

`getRowCount()` returns the number of rows in the table, which is the size of the `clientsList`. `getColumnCount()` returns the number of columns in the table, which is always 4. `getValueAt()` returns the value of the specified cell in the table. The method takes two arguments, `rowIndex` and `columnIndex`, which represent the row and column index of the cell, respectively. The method first retrieves the `Client` object at the specified row index, and then returns the value of the column specified by the column index.

The class also has a method `putClients()`, which sets the list of clients to display in the table. It takes a list of `Client` objects as its argument and sets the `clientsList` instance variable to the argument value.

*4.18 ProductTableModel*

The ProductTableModel class is a Java class that extends the AbstractTableModel class and represents a table model used to display a list of products in a JTable. It contains a private instance variable "productsList" which is a List of Product objects. The class overrides the methods "getRowCount", "getColumnCount", and "getValueAt" from the AbstractTableModel class to define the behavior of the table model. The "getRowCount" method returns the number of rows in the table, while the "getColumnCount" method returns the number of columns in the table. The "getValueAt" method returns the value at the specified cell in the table based on the row and column indices. The class also contains a public method "setProducts" which sets the list of products to be displayed in the table.

*4.19 PurchaseTableModel*

The `PurchaseTableModel` class is a Java class that extends the `AbstractTableModel` abstract class and is used to display a list of purchases in a JTable. The class contains a list of `Purchase` objects, and implements methods to get the number of rows and columns in the table, as well as to get the value at a specific cell in the table. The class also provides a method to set the list of purchases to be displayed in the table. The table has four columns: `id`, `idClient`, `idProduct`, and `quantity`.

*4.20 BillTableModel*

The `BillTableModel` class is a table model used to display a list of bills in a `JTable`. It extends the `AbstractTableModel` class and has an instance variable `billsList` which is a list of `Bill` objects. It overrides three methods of the `AbstractTableModel` class: `getRowCount()`, `getColumnCount()`, and `getValueAt()`.

`getRowCount()` returns the number of rows in the table and simply returns the size of the `billsList`.

`getColumnCount()` returns the number of columns in the table, which is seven.

`getValueAt()` returns the value at the specified cell in the table. It takes two parameters: the row index and the column index. It uses these indices to retrieve the corresponding `Bill` object from the `billsList`. It then uses a switch statement to determine which column to display, and returns the appropriate value from the `Bill` object.

The `putBills()` method sets the list of bills to display in the table. It takes a list of `Bill` objects as a parameter and sets the `billsList` instance variable to this list.

*4.21 HomePageView*

This is a Java Swing class that extends JFrame and represents the main page of an application for managing orders. It has three buttons to perform different tasks: editing a client account, editing a product, and placing an order.

*4.21 ClientView*

The class is a Java Swing GUI for managing client accounts. It contains various GUI components such as buttons, labels, and text fields. The class also has a JComboBox for selecting whether to edit or delete an existing client, and a ClientTableModel object for displaying the clients in a JTable.

The constructor of the class takes an ArrayList of Client objects as a parameter. It sets up the GUI components, including the JTable that displays the clients.

The createButton and showAllClientsButton buttons are used to create a new client and display all clients, respectively. The JComboBox is used to select whether to edit or delete an existing client.

*4.22 ProductView*

The class is a Java Swing GUI for managing products. It contains various GUI components such as buttons, labels, and text fields. The class also has a JComboBox for selecting whether to edit or delete an existing product, and a ProductTableModel object for displaying the products in a JTable.

The constructor of the class takes an ArrayList of Product objects as a parameter. It sets up the GUI components, including the JTable that displays the clients.

The createButton and showAllProductsButton buttons are used to create a new client and display all products, respectively. The JComboBox is used to select whether to edit or delete an existing product.

*4.23 PurchaseView*

This class defines a graphical user interface for managing purchase orders. The ‘PurchaseView’ class extends the ‘JFrame’ class and contains several components such as text fields, buttons, and labels to collect and display information about purchase orders.

The constructor of the ‘PurchaseView’ class takes an ArrayList of ‘Purchase’ objects as an argument, which is used to initialize the view with the existing purchase orders.

The ‘PurchaseView’ class creates a graphical user interface with labels and text fields to collect and display information about purchase orders. The buttons in the interface allow the user to place orders, reset the text fields, display all the existing orders, calculate the final price of the order, edit orders, and delete orders. The view is initialized with the existing purchase orders passed to the constructor.

*4.24 BillView*

This is the implementation of a class located in the "presentation" package. It extends the ‘JFrame’ class, which is a top-level container that represents a window. The purpose of this class is to display a table of bills, and it receives an ArrayList of ‘Bill’ objects as a parameter in its constructor.

*4.25 HomePageController*

The ‘HomePageController’ class is responsible for controlling the main page of the application. It has three inner classes that implement the ActionListener interface and listen to user events when the corresponding buttons are clicked.

When the "Client" button is clicked, the ‘ClientPag’e inner class is called, which creates an empty ArrayList of Client objects, creates a new ‘ClientView’ object with the empty ArrayList, and creates a new ClientController object with the ‘ClientView’ object. The ‘ClientController’ is responsible for controlling the ‘ClientView’ and its interactions with the user.

Similarly, when the "Product" button is clicked, the ‘ProductPage’ inner class is called, which creates an empty ArrayList of Product objects, creates a new ‘ProductView’ object with the empty ArrayList, and creates a new ‘ProductController’ object with the ‘ProductView’ object.

Finally, when the "Order" button is clicked, the ‘OrderPage’ inner class is called, which creates an empty ArrayList of Purchase objects, creates a new ‘PurchaseView’ object with the empty ArrayList, and creates a new ‘PurchaseController’ object with the ‘PurchaseView’ object.

*4.26 ClientController*

The ‘ClientController’ class contains inner classes that implement ActionListener interface to handle the specific events and to perform the operations according to the event.

The constructor of the ‘ClientController’ class takes an instance of the ‘ClientView’ class as an argument and adds action listeners to the view's buttons.

The ‘EditDelete’ class is an inner class that implements ActionListener interface to handle the edit and delete event. The ‘actionPerformed()’ method of this class checks the type of event (edit or delete) and calls the appropriate method of the ‘ClientBLL’ class to modify or delete the client data.

The ‘Create’ class is an inner class that implements ActionListener interface to handle the create event. The ‘actionPerformed()’ method of this class creates a new instance of the Client class and calls the ‘insertClient()’ method of the’ ClientBLL’ class to insert the client data.

The ‘ShowAl’l class is an inner class that implements ActionListener interface to handle the show event. The actionPerformed() method of this class retrieves all clients from the database using the ‘getAllClients()’ method of the ‘ClientBLL’ class and creates a new instance of the ‘ClientView’ class to display the list of clients.

The Reset class is an inner class that implements ActionListener interface to handle the reset event. The ‘actionPerformed()’ method of this class calls the ‘refresh()’ method of the view object to reset the form fields.

*4.27 ProductController*

This class is responsible for controlling the flow of information between the ‘ProductView’ and the ‘ProductBLL’.

The ‘ProductController’ has four inner classes that implement the ActionListener interface to handle the user interactions in the view.

The ‘EditDelete’ class handles the edit and delete actions. If the user selects "edit", it creates a ‘Product’ object with the data from the view and passes it to the ‘ProductBLL’ to be updated in the database. If the user selects "delete", it first checks if there are any ‘Purchase’ objects associated with the ‘Product’. If there are, it deletes them and then deletes the ‘Product’. If there aren't any, it simply deletes the ‘Product’.

The ‘Add’ class handles the add action. It creates a ‘Product’ object with the data from the view and passes it to the ‘ProductBLL’ to be inserted in the database. If the insertion is successful, it shows a success message. If the product already exists, it shows an error message. If any of the input values are negative, it also shows an error message.

The ‘ShowAll’ class handles the show action. It retrieves all the products from the database using the ‘ProductBLL’ and creates a new ‘ProductView’ with the retrieved data. It then disposes the current view and creates a new ‘ProductController’ with the new view.

The Reset class handles the reset action. It simply refreshes the view by calling the refresh method of the ‘ProductView’.

The constructor of the ‘ProductController’ takes a ‘ProductView’ object as a parameter and sets up the event listeners for the edit/delete, add, show, and reset actions.

*4.28 PurchaseController*

The ‘PurchaseController’ class is responsible for controlling the user interactions with the orders management system. It handles the events triggered by the user interface of the purchase view and interacts with the business logic layer. This class implements ActionListener interface to listen to the events generated by the user interface. It contains inner classes to handle specific events and to perform the operations according to the event.

The methods and inner classes in the ‘PurchaseController’ class are:

* ‘ PurchaseController(PurchaseView view)’: Constructor method that initializes the ‘PurchaseController’ object with the specified view object and adds action listeners to the view's buttons.
* ‘isAuthenticated()’: A method that returns a Boolean value indicating whether the client is authenticated by checking the email and password fields entered in the GUI against those stored in the database.
* ‘PlaceOrder’: An inner class that implements ActionListener interface to handle the add event.
* ‘ShowAl’l: An inner class that implements ActionListener interface to handle the show event.
* ‘GoToBill’: An inner class that implements ActionListener interface to handle the go event.
* ‘Delete’: An inner class that implements ActionListener interface to handle the delete event.
* ‘Edit’: An inner class that implements ActionListener interface to handle the edit event.

*4.29 Main*

The Main class serves as the entry point to the program and contains the main method.

# Results

The application works as expected.

# Conclusions

This application is functional, giving correct result and also provides an easy way to be used, the interface being as simple as possible, but not in a way to become impractical.

The aim of this assignment was to learn to work with databases and reflection, concepts not so used until now.

For future, there can be some modification: there can be added more inputs, which means more constraints for the application, adding more fields to the classes, or a better design of the interface, in terms a more pleasant look or for the ease of use.

# Bibliography

The references that were consulted by the student during the implementation of the homework will be added.

Example:

1. *Bruce Eckel, Thinking in Java (4th Edition), Publisher: Prentice Hall PTRUpper Saddle River, NJUnited States, ISBN:978-0-13-187248-6 Published:01 December 2005.*
2. *What are Java classes? -* [*www.tutorialspoint.com*](http://www.tutorialspoint.com)
3. *Reflection -* [*https://www.oracle.com/technical-resources/articles/java/javareflection.html*](https://www.oracle.com/technical-resources/articles/java/javareflection.html)
4. *OOP -* [*https://blog.proto.io/how-object-oriented-design-principles-relate-to-mobile-app-design/*](https://blog.proto.io/how-object-oriented-design-principles-relate-to-mobile-app-design/)
5. *UML -* [*https://www.lucidchart.com/pages/uml-package-diagram*](https://www.lucidchart.com/pages/uml-package-diagram)
6. *Class Diagram -* [*https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/*](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/)
7. *Logger -* [*https://docs.oracle.com/javase/8/docs/api/java/util/logging/Logger.html*](https://docs.oracle.com/javase/8/docs/api/java/util/logging/Logger.html)