UE21CS352B - Object Oriented Analysis & Design using Java

Mini Project Report

"Inventory Management System"

Submitted by:

Niranjan Mayur S	**PES1UG21CS390**
Nischal Kashyap	**PES1UG21CS394**
Pavani.R.Acharya	**PES1UG21CS409**
Poorvi Tambakad	**PES1UG21CS412**

6th *Semester* ***G*** *Section*

Prof. Raghu BA
Associate Professor

January - May 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

1. Problem Statement

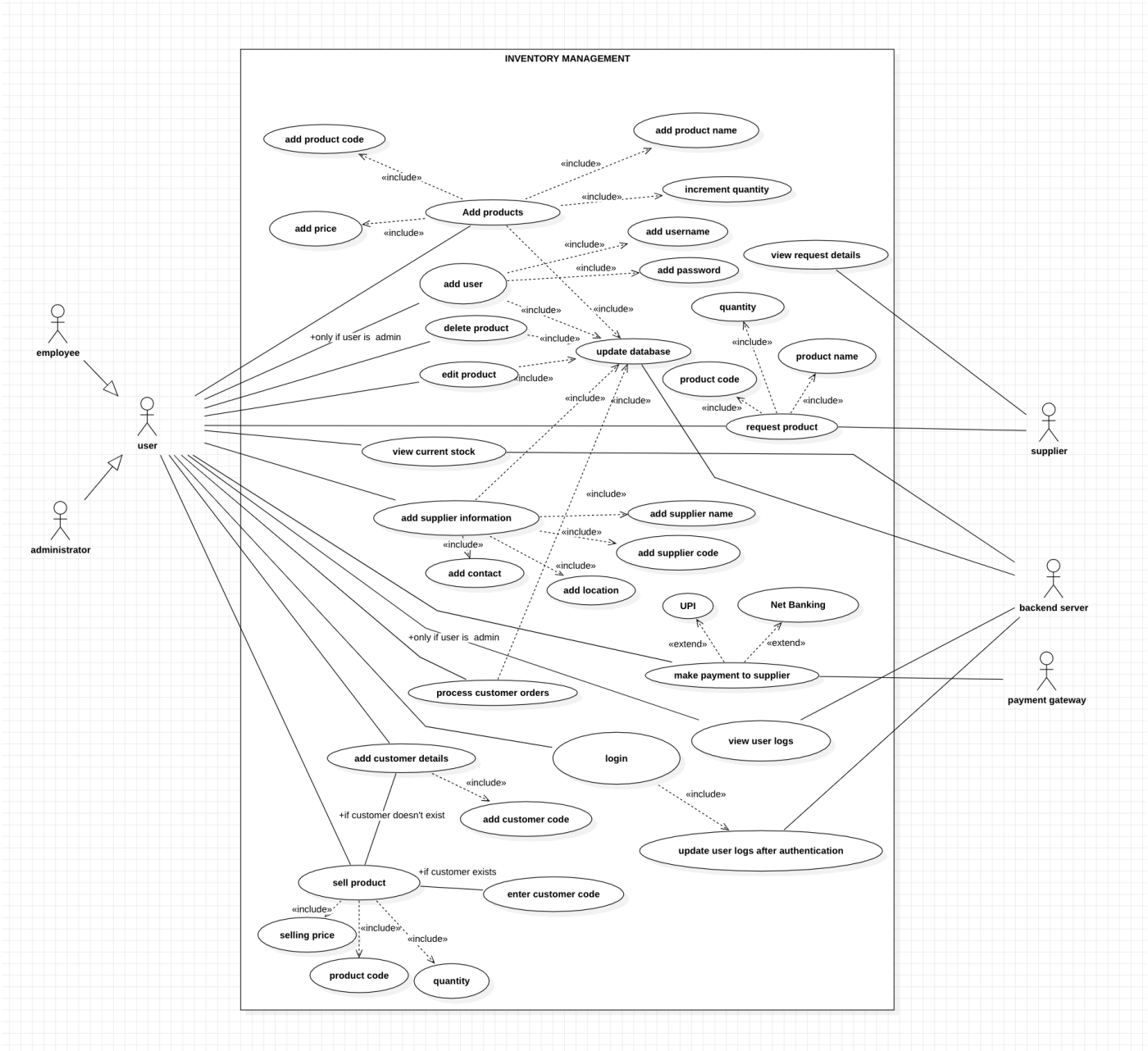
The problem statement revolves around the development of an Inventory Management System (IMS), which is a crucial component for businesses to efficiently handle their inventory or stock levels. The primary goal is to design and implement a software application or platform that offers a comprehensive suite of tools and functionalities to manage the entire inventory lifecycle seamlessly.

In this project, Java has been utilized as the programming language to develop the IMS. By leveraging Java's robust features and capabilities, the team aims to construct a user-friendly system with an intuitive and interactive Graphical User Interface (GUI). The GUI is essential for enhancing user experience and facilitating easy navigation and interaction with the system.

Throughout the development process, several design principles and design patterns have been employed to ensure the system's effectiveness, maintainability, and scalability.

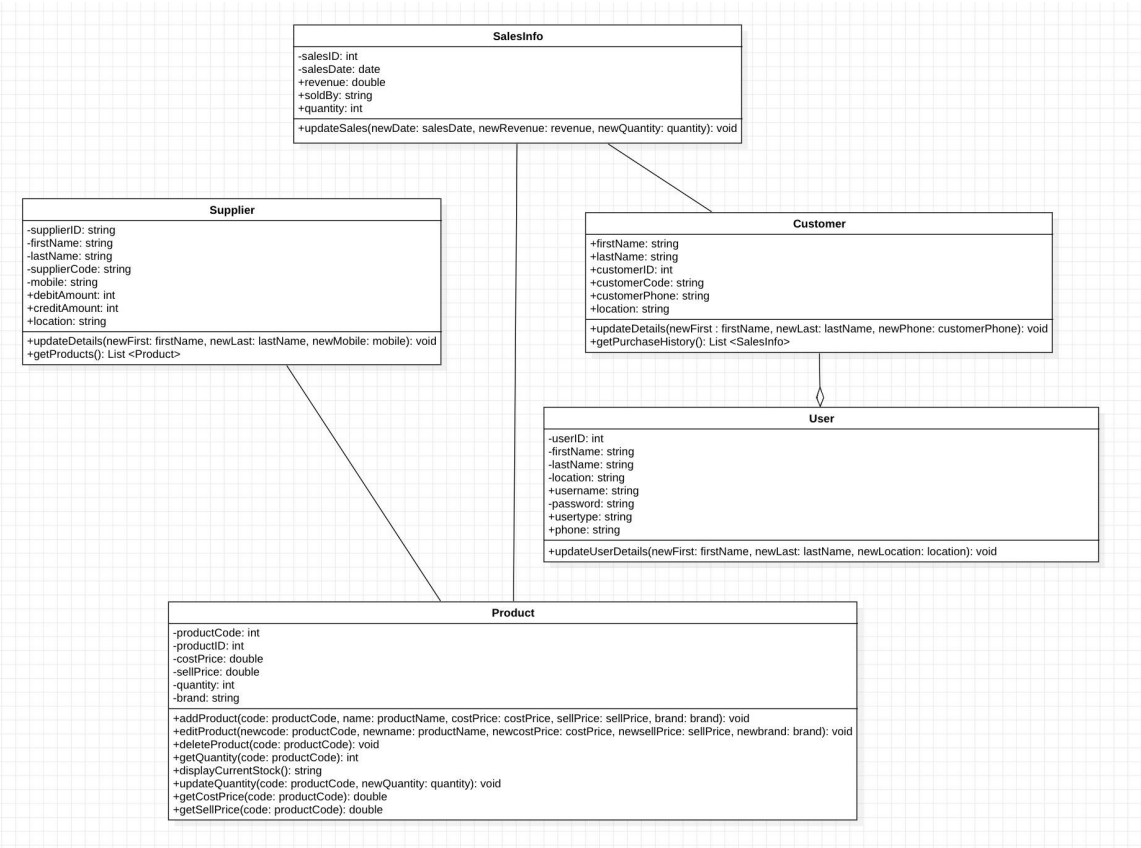
2. Model

2.1 Use Case Diagram -

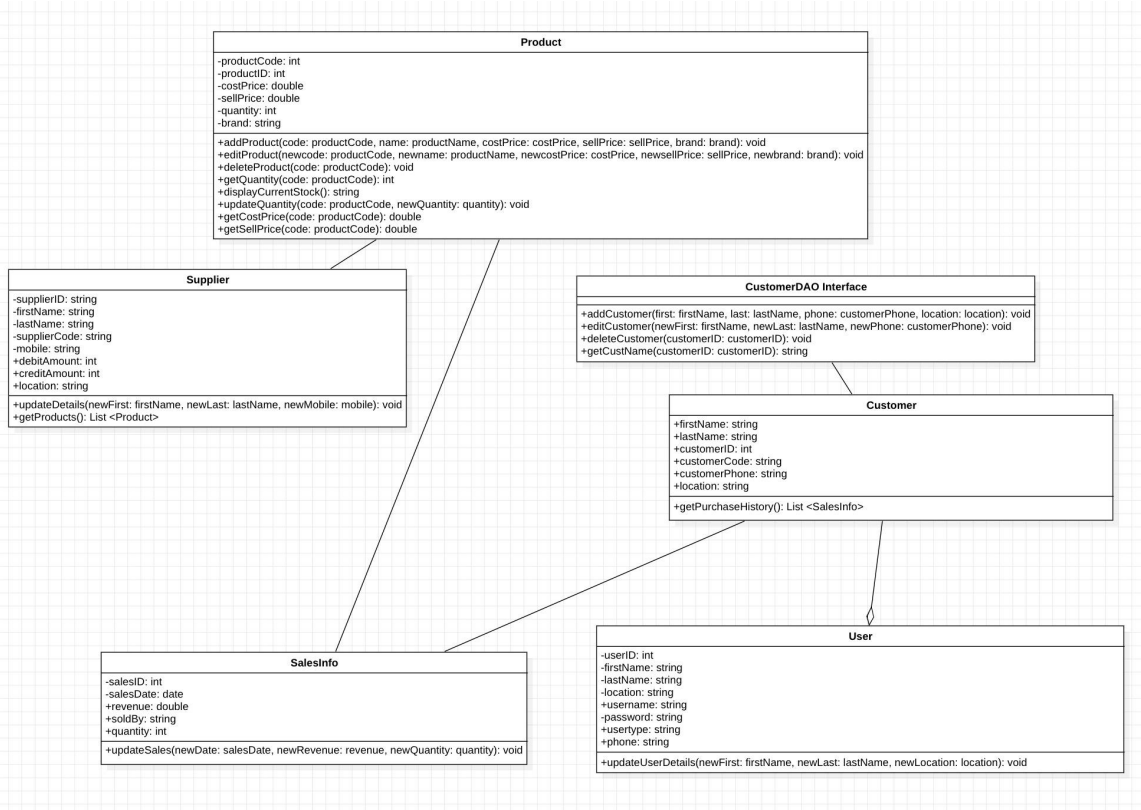


2.2 Class Diagram-

- BEFORE applying the Design principles

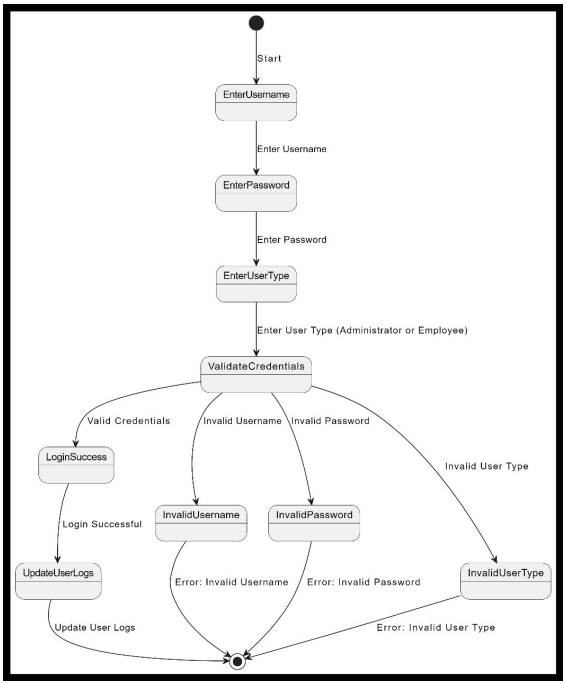


- AFTER applying the Design principles

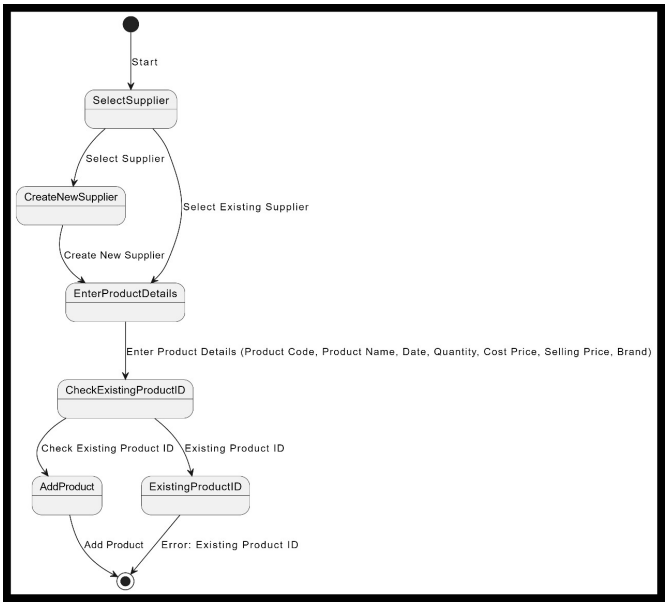


2.3 State Diagram -

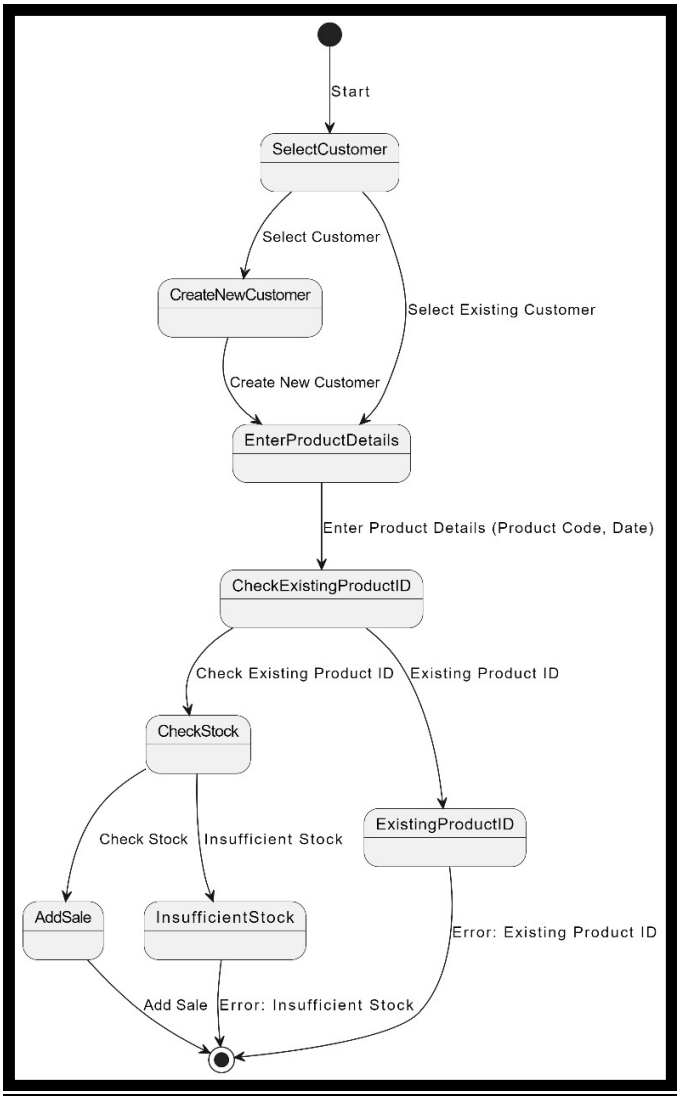
1. User Login



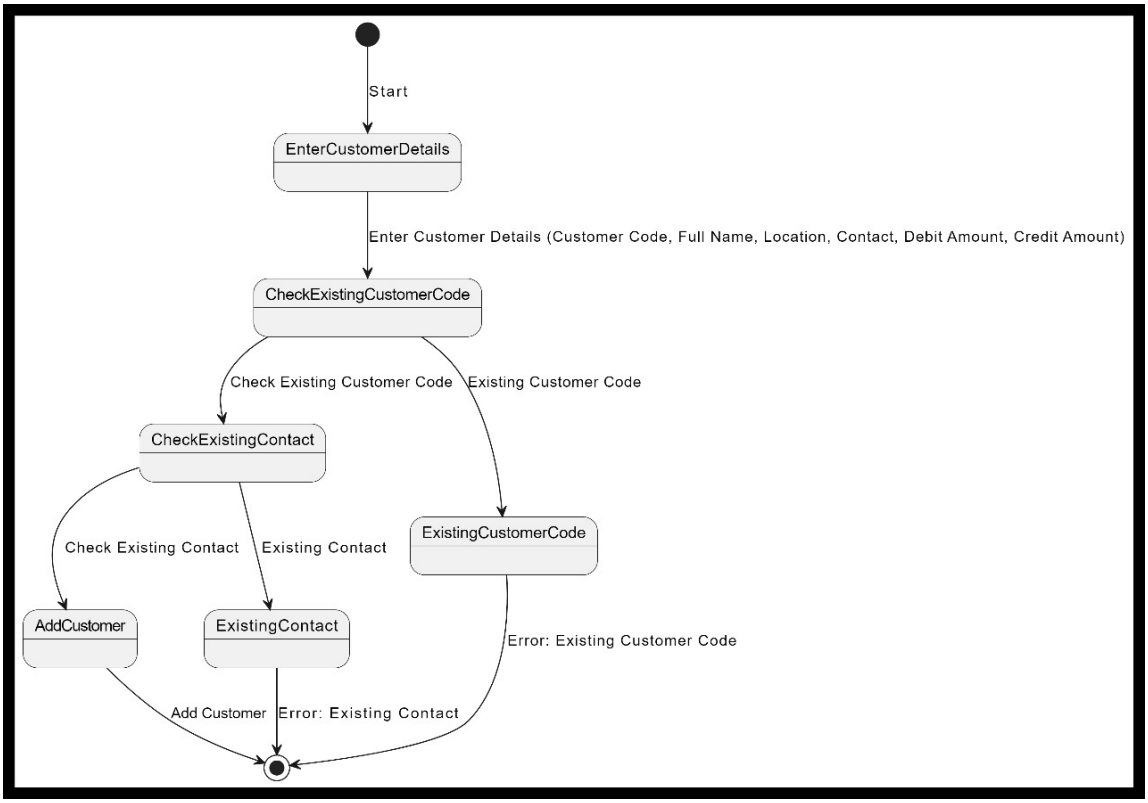
2. Updating product details



3. Updating a Sale

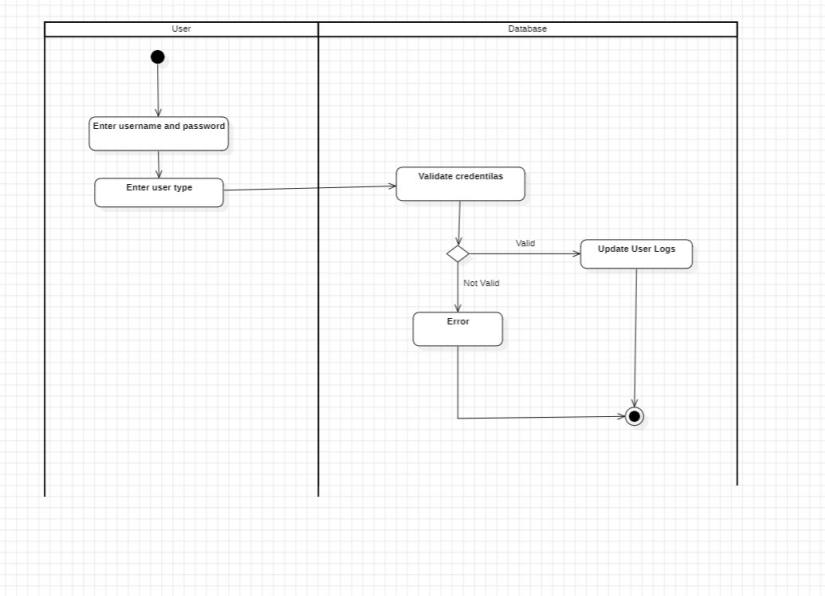


4. Adding a Customer

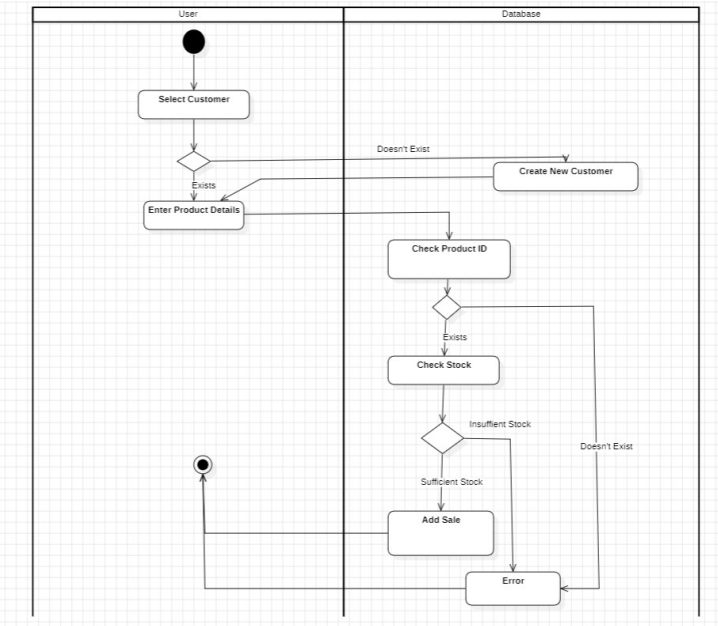


2.4 Activity Diagram -

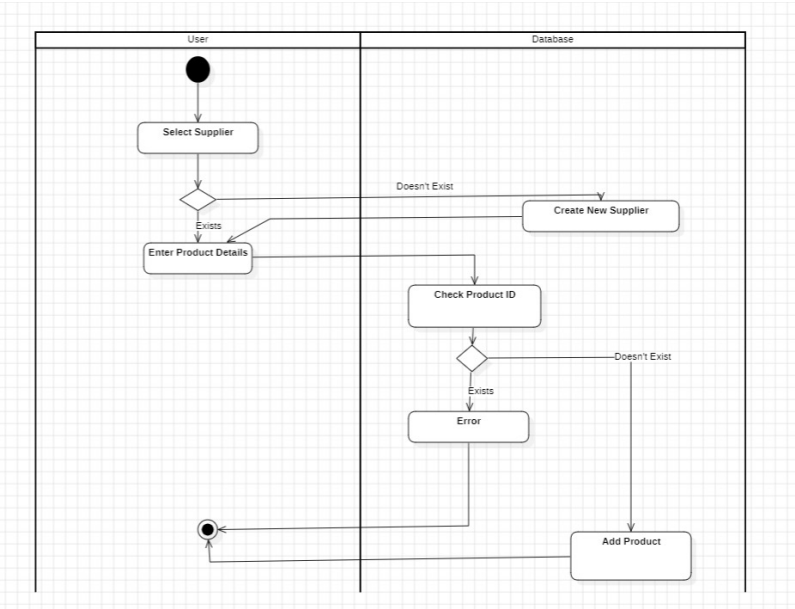
1. Login Page



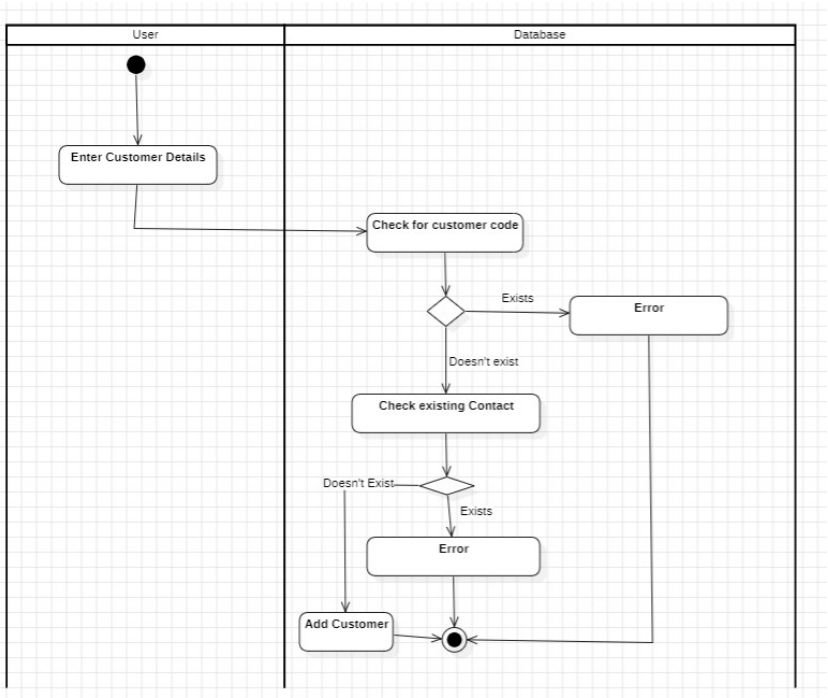
2. Add Sales



3. Update Product



4. Customer Details



3. Architecture Patterns

MVC, or Model-View-Controller, is a software architectural pattern commonly used for developing user interfaces.

1. Model: The model represents the application's data and business logic. In this code, the UserDTO class represents the data model for a user, containing attributes such as full name, location, phone number, username, password, and user type. The UserDAO class acts as the interface to interact with the underlying database, encapsulating database operations related to users.

2. View: The view represents the presentation layer of the application, responsible for displaying data to the user and capturing user input. In this code, the Swing UI components (JPanel, JTable, JTextField, JComboBox, etc.) within the UserLogsPage and UsersPage classes serve as the views. These classes handle the layout and display of user interface elements, presenting data retrieved from the model and capturing user interactions.

3. Controller: The controller acts as an intermediary between the model and the view, handling user input, updating the model, and updating the view accordingly. In this code, event listeners and handlers (e.g., ActionPerformed methods) within the UserLogsPage and UsersPage classes serve as controllers. These event handlers respond to user actions such as button clicks, capture user input, invoke appropriate methods in the model (UserDAO), and update the view accordingly.

MVC promotes a separation of concerns, making it easier to manage complex applications by dividing them into distinct components. For example, changes to the user interface (handled by the view) can be made independently of changes to the underlying data or business logic (handled by the model). This separation also promotes code reusability and maintainability, as each component can be developed and tested separately.

4. Design Principles

4.1 Single Responsibility Principle (SRP):

Each class in the application has a single responsibility. For example, UserLogsPage.java is responsible for displaying user logs, UsersPage.java is responsible for managing users and UserDAO.java is responsible for database operations related to users. This ensures that each class is focused and has a clear purpose.

UserDAO.java : code snippet shows the implementation of a function that adds a user to the database

```
// Methods to add new user
public void addUserDAO(UserDTO userDTO, String userType) {
    try {
        String query = "SELECT * FROM users WHERE name='"
                +userDTO.getFullName()
                +"' AND location='"
                +userDTO.getLocation()
                +"' AND phone='"
                +userDTO.getPhone()
                +"' AND usertype='"
                +userDTO.getUserType()
                +"'";
        resultSet = statement.executeQuery(query);
        if(resultSet.next())
            JOptionPane.showMessageDialog(null, "User already exists");
        else
            addFunction(userDTO, userType);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

UserPage.java : code snippet of a function that shows us how we're managing user by deleting a user.

```
private void deleteButtonActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_deleteButtonActionPerformed
    if (userTable.getSelectedRow()<0)
        JOptionPane.showMessageDialog(null, "Please select an entry from the table");
    else{
        int opt = JOptionPane.showConfirmDialog(
                null,
                "Are you sure you want to delete this user?",
                "Confirmation",
                JOptionPane.YES_NO_OPTION);
        if(opt==JOptionPane.YES_OPTION) {
            new UserDAO().deleteUserDAO(
                    String.valueOf(
                            userTable.getValueAt
                                    (userTable.getSelectedRow(), 4)));
            loadDataSet();
        }
    }
}//GEN-LAST:event_deleteButtonActionPerformed
```

4.2 Dependency Inversion Principle

The Dependency Inversion Principle suggests that high-level modules/classes should not depend on low-level modules/classes. Both should depend on abstractions. Additionally, it states that abstractions should not depend on details; rather, details should depend on abstractions

To apply the Dependency Inversion Principle, we introduce abstractions/interfaces to decouple the high-level CustomerDAO class from low-level implementation details

```java
package com.inventory.DAO;

import com.inventory.DTO.CustomerDTO;

import java.sql.ResultSet;

public interface CustomerDAOInterface {
    void addCustomer(CustomerDTO customerDTO);
    void editCustomer(CustomerDTO customerDTO);
    void deleteCustomer(String custCode);
    ResultSet getQueryResult();
    ResultSet getCustomerSearch(String text);
    ResultSet getCustName(String custCode);
    ResultSet getProdName(String prodCode);
}
```

By introducing the interface, we decouple the CustomerDAO class from specific implementations of the database-related classes (ConnectionFactory, PreparedStatement, Statement, etc.), thus adhering to the Dependency Inversion Principle. This makes the code more flexible, maintainable, and easier to test

```java
public class CustomerDAO implements CustomerDAOInterface {
    private Connection conn = null;
    private PreparedStatement prepStatement = null;
    private Statement statement = null;
    private ResultSet resultSet = null;

    public CustomerDAO() {
        try {
            conn = new ConnectionFactory().getConn();
            statement = conn.createStatement();
        } catch (SQLException e) {
            e.printStackTrace();
        }
```

The CustomerDAO class implements the CustomerDAOInterface, which provides an abstraction for the DAO operations. This separation allows for easier maintenance, testing, and potential future changes in the implementation details of the DAO operations without affecting the high-level components that use the CustomerDAO class.

4.3 Creator Principle :

The Creator Principle is a fundamental concept in object-oriented programming (OOP) that emphasizes the encapsulation of object creation logic within the classes themselves. According to this principle, the class responsible for creating an object should also be responsible for managing its life cycle. This means that the process of creating and initializing an object should be handled within the class or within closely related classes.

```
// The ProductDAO class creates instances of ProductDAO and manages database connect
public ProductDAO() {
    try {
        conn = new ConnectionFactory().getConn();
        statement = conn.createStatement();
        statement2 = conn.createStatement();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

In the provided code snippet, the ProductDAO class is responsible for creating instances of ConnectionFactory and statements. This aligns with the Creator Principle because ProductDAO logically requires a database connection (ConnectionFactory) and statements to interact with the database. Therefore, it is appropriate for ProductDAO to create these instances within its constructor.

4.4 Seperation of Concerns (SoC) :

Separation of Concerns (SoC) is a design principle that encourages dividing a computer program into distinct sections, each addressing a separate concern or aspect of functionality. The goal is to isolate different responsibilities or concerns within the system to make the codebase more modular, maintainable, and comprehensible.

```
// Method to load data into table
public void loadDataSet() {
    try {
        ProductDAO productDAO = new ProductDAO();
        purchaseTable.setModel(productDAO.buildTableModel(productDAO.getPurchaseInfo()));
    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
}
```

The Separation of Concerns (SoC) principle, exemplified in the provided code snippet, advocates for the distinct division of responsibilities within a software system. Here, the loadDataSet() method encapsulates the UI interaction logic, while the actual data retrieval and conversion tasks are delegated to the ProductDAO class. This separation ensures that UI-related operations are isolated from database interactions, promoting modularity, code reuse, and easier maintenance. By adhering to SoC, each component focuses on a specific concern, enhancing the overall clarity and maintainability of the codebase.

5. Design Patterns

5.1 Observer

The Observer pattern is implicitly used in the event handling mechanism of Swing. For instance, when a user clicks on a button (addButton, deleteButton, etc.), an event is triggered, and registered listeners (event handlers) are notified to perform the corresponding actions. This decouples the event source from the event handlers, promoting loose coupling and easier maintenance.

Code snippet that shows us how the addButtonActionPerformed( ) function is calling UserDAO() to add a new user. This decouples the event of adding a user when the add button action is performed

```
private void addButtonActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_addButtonActionPerformed
    UserDTO userDTO = new UserDTO();

    if (nameText.getText().equals("") || locationText.getText().equals("") || phoneText.getText().equals(""))
        JOptionPane.showMessageDialog(null, "Please fill all the required fields.");
    else {
        userType = (String) userTypeCombo.getSelectedItem();
        userDTO.setFullName(nameText.getText());
        userDTO.setLocation(locationText.getText());
        userDTO.setPhone(phoneText.getText());
        userDTO.setUsername(usernameText.getText());
        userDTO.setPassword(passText.getText());
        userDTO.setUserType(userType);
        new UserDAO().addUserDAO(userDTO, userType);
        loadDataSet();
    }
```

5.2 Data Access Object Pattern

The DAO pattern abstracts and encapsulates all access to a data source, such as a database. It separates the business logic from the data access logic, providing a clean interface for accessing data while hiding the implementation details of how the data is actually fetched or persisted.

Our CustomerDAO class encapsulates all database operations related to customers, providing methods like addCustomer, editCustomer, deleteCustomer, etc. These methods abstract away the details of interacting with the database and provide a clear interface for other parts of your system to use.

```java
public CustomerDAO() {
    try {
        conn = new ConnectionFactory().getConn();
        statement = conn.createStatement();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

@Override
public void addCustomer(CustomerDTO customerDTO) {
    try {
        String query = "SELECT * FROM customers WHERE fullname='"
                + customerDTO.getFullName()
                + "' AND location='"
                + customerDTO.getLocation()
                + "' AND phone='"
                + customerDTO.getPhone()
                + "'";
        resultSet = statement.executeQuery(query);
        if (resultSet.next())
            JOptionPane.showMessageDialog(parentComponent:null, message:"Customer already exists.");
        else
            addFunction(customerDTO);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

5.3 Singleton Pattern

The Singleton pattern ensures that only one instance of a class exists and gives a way to access it globally. It does this by making sure the class's constructor is private so that it can't be created from outside, and it provides a static method to get the instance. This method creates the instance if it doesn't exist yet and returns it if it does. This pattern is handy when you need to manage resources like database connections or configuration settings across your program. It keeps things simple by making sure there's only one instance of the class, which can be accessed from anywhere in the code.

```java
public class UserDAO {
    Connection conn = null;
    PreparedStatement prepStatement = null;
    Statement statement = null;
    ResultSet resultSet = null;

    // Constructor method
    public UserDAO() {
        try {
            conn = new ConnectionFactory().getConn();
            statement = conn.createStatement();
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
```

5.4 Table data getaway

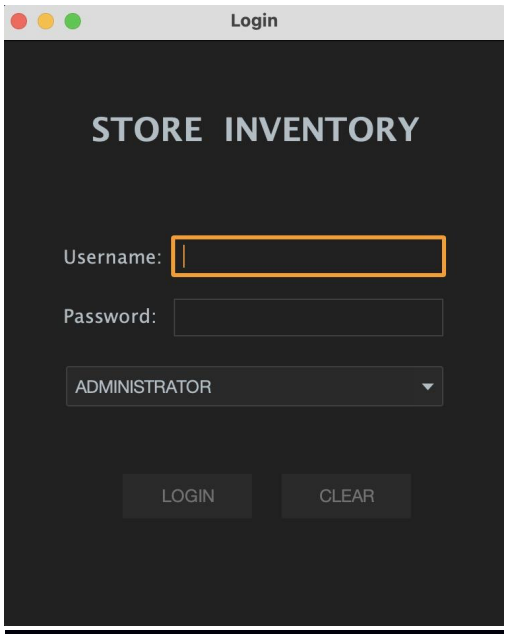
The table data gateway pattern is utilized through the buildTableModel method, which is responsible for converting a ResultSet into a DefaultTableModel. This method is a clear implementation of the table data gateway pattern, as it abstracts the details of accessing the database and provides a way to easily display the data in a tabular format in a Swing application.

The buildTableModel method is used in the context of a GUI application where a JTable is populated with data retrieved from a database. The method takes a ResultSet object as input, retrieves the column names and data from the ResultSet, and uses them to populate a DefaultTableModel object. This DefaultTableModel object is then set to a JTable to display the data in a tabular format. This process abstracts the details of database access and data manipulation, providing a clean separation between the database and the user interface components

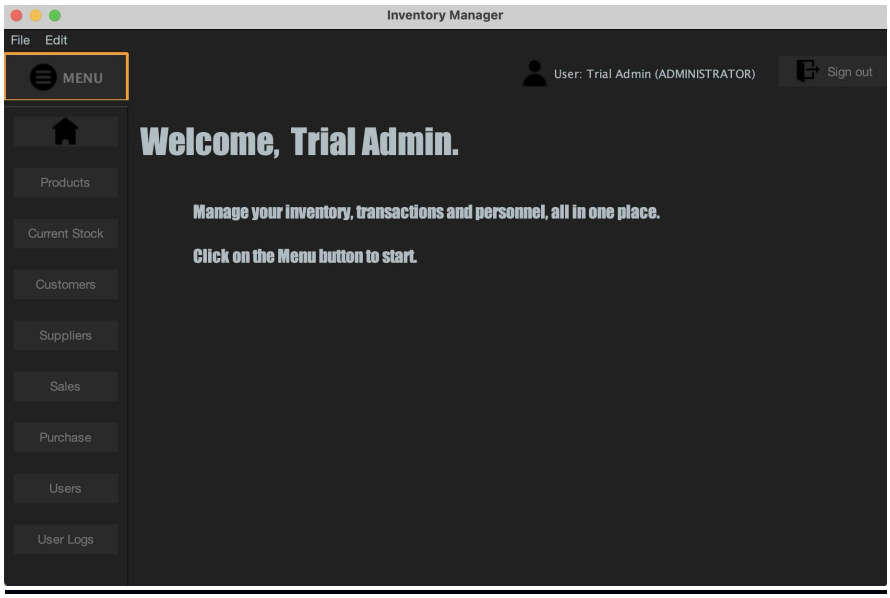
```java
    // Method to display data set in tabular form
    public DefaultTableModel buildTableModel(ResultSet resultSet) throws SQLException {
        ResultSetMetaData metaData = resultSet.getMetaData();
        Vector<String> columnNames = new Vector<String>();
        int colCount = metaData.getColumnCount();

        for (int col=1; col <= colCount; col++){
            columnNames.add(metaData.getColumnName(col).toUpperCase(Locale.ROOT));
        }

        Vector<Vector<Object>> data = new Vector<Vector<Object>>();
        while (resultSet.next()) {
            Vector<Object> vector = new Vector<Object>();
            for (int col=1; col<=colCount; col++) {
                vector.add(resultSet.getObject(col));
            }
            data.add(vector);
        }
        return new DefaultTableModel(data, columnNames);
    }
```

6. Output Screenshots

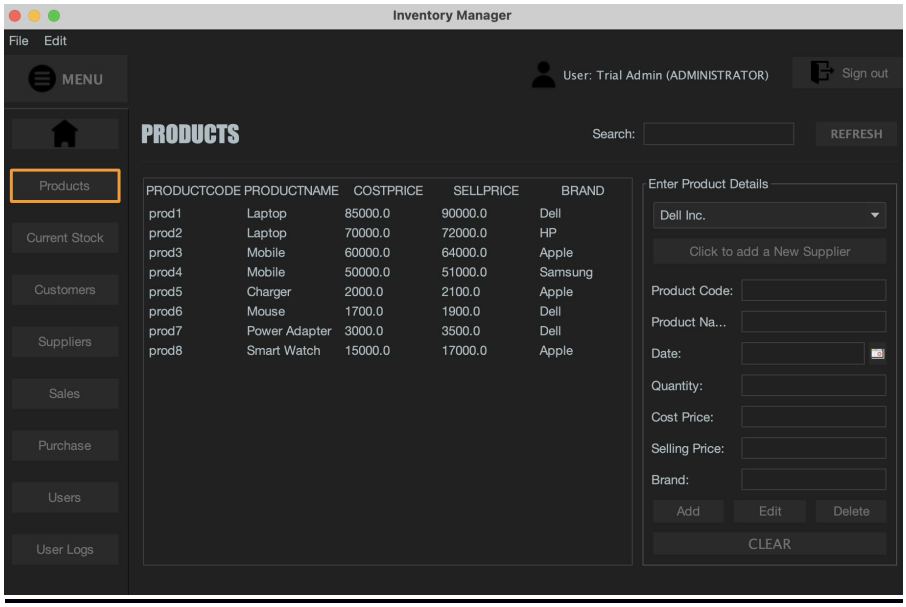
6.1 Login



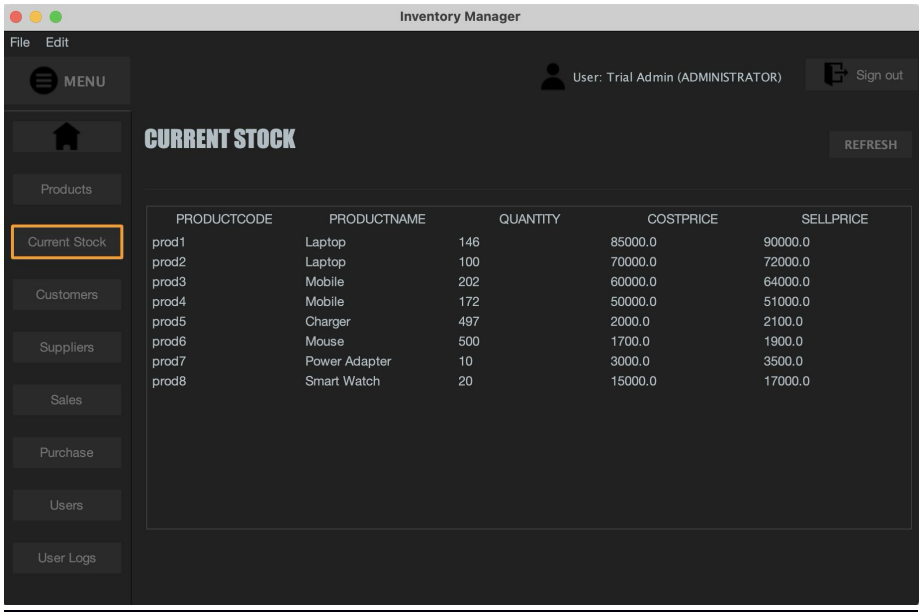
6.2 Dashboard - Admin



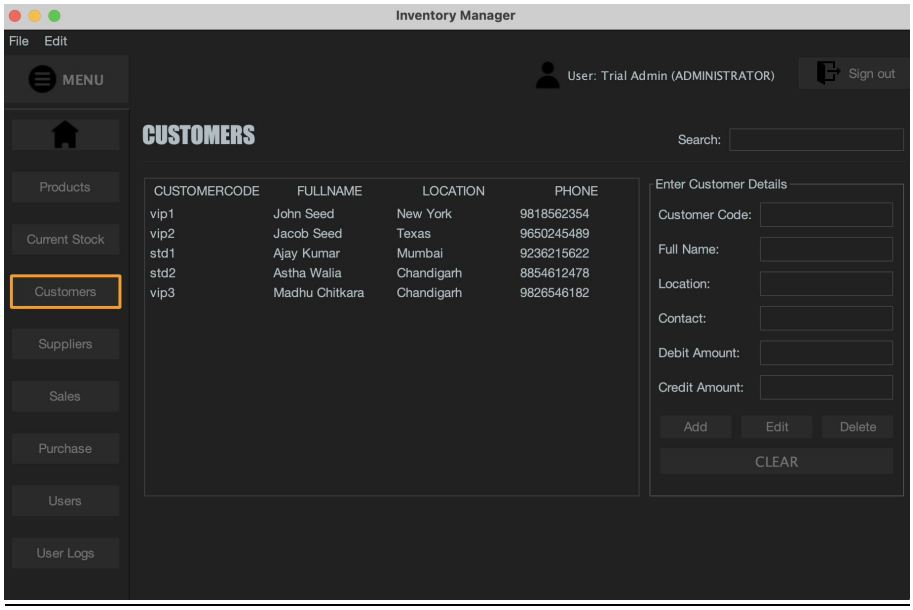
6.3 Products View



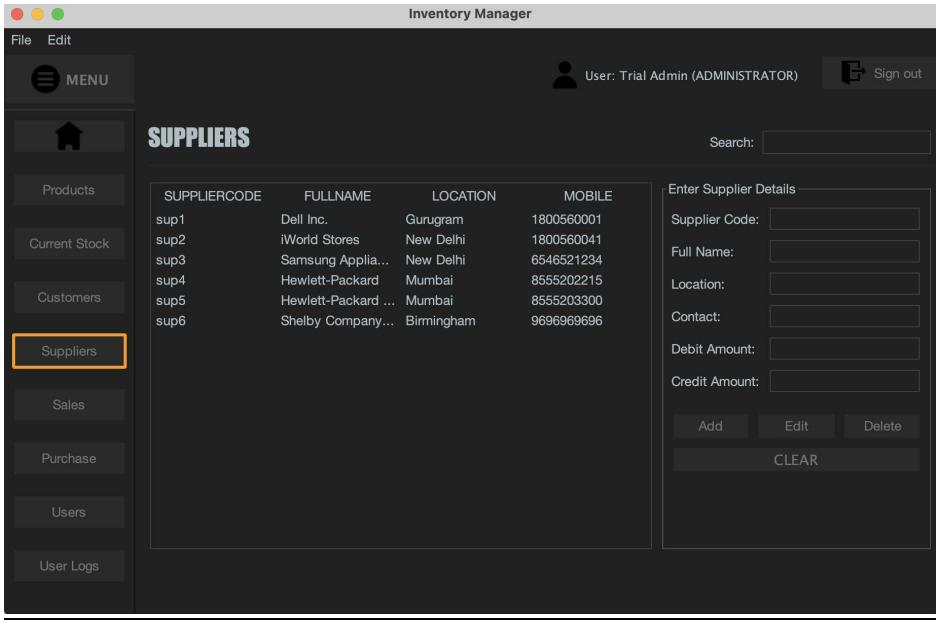
6.4 Current Stock View



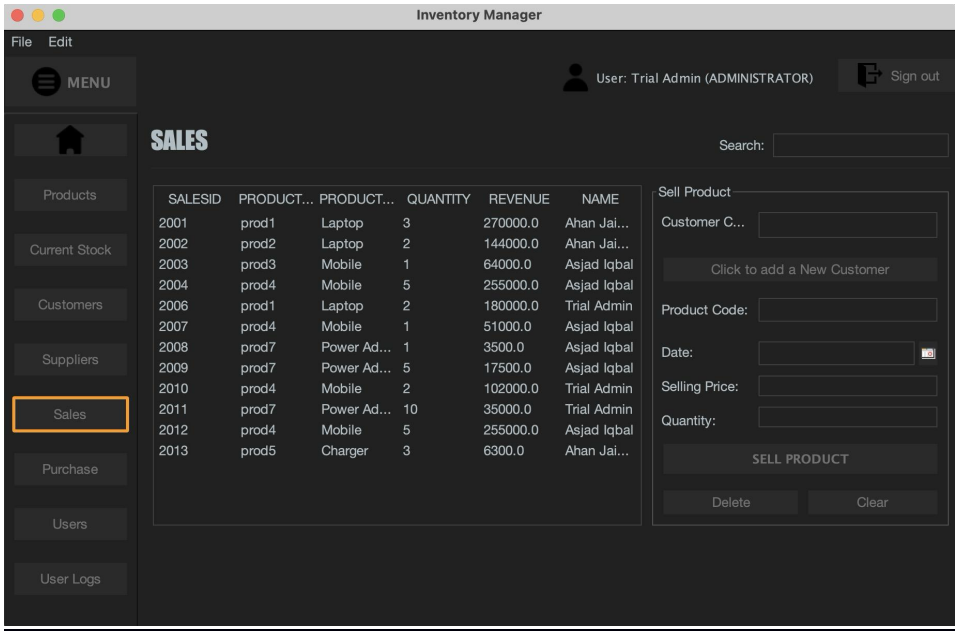
6.5 Customers View



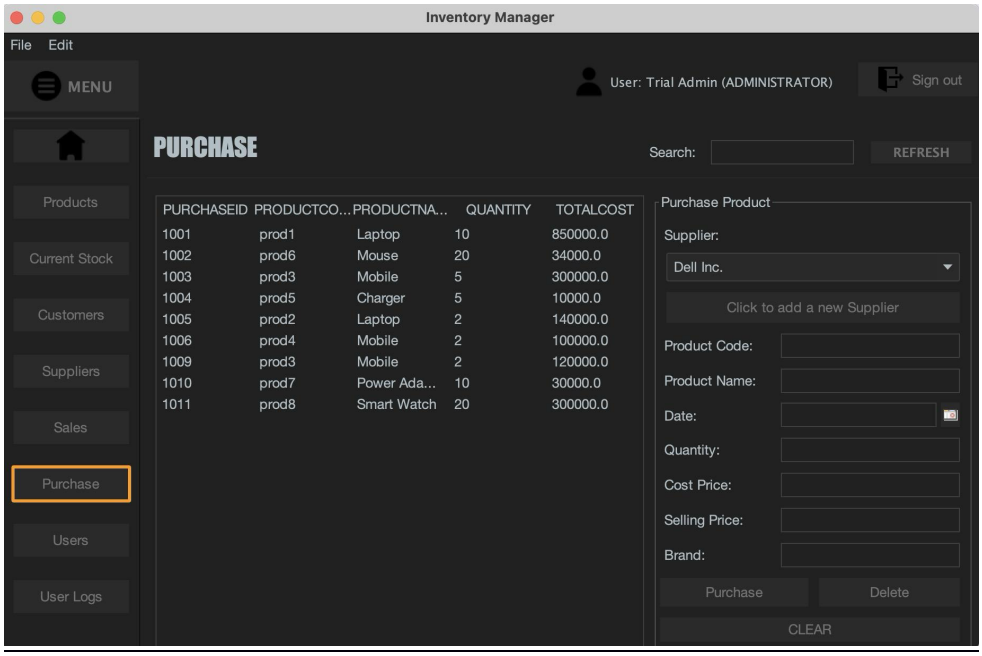
6.6 Suppliers View



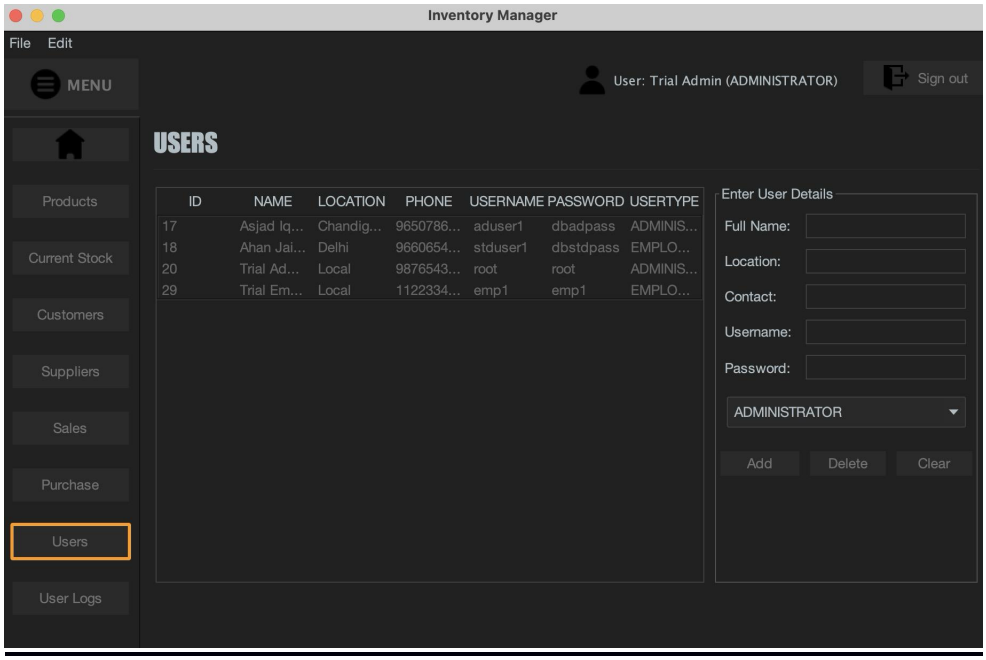
6.7 Sales View



6.8 Purchases View



6.9 Users View



7. Individual Contributions

a. Niranjan Mayur (PES1UG21CS390) -
   i. Use Case Diagram
   ii. Activity Diagram - Sales Details
   iii. State Diagram - Updating a Sale
b. Nischal Kashyap (PES1UG21CS394) -
   i. Class Diagram
   ii. Activity diagram - Login page
   iii. State Diagram - User Login
c. Pavani.R.Acharya (PES1UG21CS409) -
   i. Use Case Diagram
   ii. Activity diagram - Product update
   iii. State Diagram - Updating Product details
d. Poorvi Tambakad (PES1UG21CS412) -
   i. Class Diagram
   ii. Activity diagram - Customer Details
   iii. State Diagram - Adding a new Customer