JAVA GROUP REPORT

CT038-3-2-OODJ

OBJECT ORIENTED DEVELOPMENT WITH

APU2F2502CS(DA)

LECTURE BY

DR. KADHAR BATCHA NOWSHATH

GROUP MEMBER

NAME	STUDENT ID
OOI DUN TZI	TP071308
YAP BOON SIONG	TP070171
CHEAH JUN HENG	TP071767
CHEN XIN ZE	TP081210
LEE JUIN	TP071920

Contents

1.0 Introduction

Nowadays, along with the business environment growing rapidly, applying automation is crucial to a company, it ensures the procurement processes efficiency, accuracy, and scalability. A company called Omega Wholesale Sdn Bhd (OWSB), it is a growing wholesaler based in Kuala Lumpur and it is aware of this need. It aims to modernize its purchase order operations through a dedicated Java-based system.

A main part of this project is to develop a system that supports core business functions such as allowing users to log in, managing inventory, handling vendors and products and managing purchase requisitions and purchase orders. By integrating role-based access and object-oriented programming (OOP) principles, the system ensures that different user roles such as “Sales Managers”, “Purchase Managers”, “Inventory Managers”, “Finance Managers”, and “Administrators” can perform their specific tasks within a secure and structured environment.

By reducing manual entry errors, ensuring proper stock management, and streamlining supplier interactions to improve OWSB’s internal workflow, the solution is designed. From a technical perspective, the project serves as a hands-on implementation of core OOP concepts such as Encapsulation, Inheritance, and Polymorphism, applied within a real-world business scenario using “Java”.

This documentation clearly states how the system is built, what its design and structure are and what main components it has. This also underlines the usefulness of object-oriented strategies in constructing software that is easy to manage and extend.

2.0 Workload Matrix

TP no & Name	Area of Responsibility	Percentage Contribution
Cheah Jun Heng Tp071767	-Java files (supplier_e, supplier_v, supplier, supplier_list, item, item_e, sales, sales_e, sales_v) **Documentation** -workload matrix -Screenshot Output -OOP -class diagram	20%
Ooi Dun Tzi Tp071308	-Java files (frame, login, login_c, am, sm, fm, im, pm, user_am, user_c, finance_am, pr_pm) **Documentation** -Screenshot Output -OOP -class diagram -use case diagram	20%
Yap Boon Siong Tp070171	-Java files (inventory_e, inventory_c, inventory_e, inventory_v, po_v, pr_v, item_v, finance_r, inventory_r) **Documentation** -Screenshot Output -OOP -Additional function -Conclusion	20%
Chen Xin Ze Tp081210	-Java files (pr_e, pr_e_c., po_e, po_e_c) **Documentation** -Introduction -Screenshot Output -OOP -Additional function	20%
Lee Juin Tp071920	-Java files (po_fm, inventory_fm, finance_c, finance_e) **Documentation** -Screenshot Output -OOP -Additional function -Project Description & Limitation	20%

3.0 Object-Oriented Programming Concepts

3.1 Encapsulation

```
31      private JButton createRoleButton(String role, frame mainFrame) {
32          JButton button = new JButton(role);
33          button.setFont(new Font(name:"Arial", Font.BOLD, size:15));
34          button.setPreferredSize(new Dimension(width:100, height:30));
35
36  >       button.addActionListener(e -> {···
69
70          return button;
71      }
```

Figure 1: Encapsulation login.java

In login class, the createRoleButton() method puts together all the actions needed for creating a button, collecting input, checking credentials and moving to another panel. By putting everything within one method, it's easy to both use and manage often. Handling the username and password is done in this method and no other code can access them.

```
7   public class login_c {
8
9       private String selectedRole;
10      private String enteredUsername;
11      private String enteredPassword;
12
13      // Store user info after login
14      public static String currentUserId;
15      public static String currentUsername;
16      public static String currentRole;
17
18      // Updated constructor to accept username
19      public login_c(String selectedRole, String enteredUsername, String enteredPassword) {
20          this.selectedRole = selectedRole;
21          this.enteredUsername = enteredUsername;
22          this.enteredPassword = enteredPassword;
23      }
24
25  >   public boolean authenticate() {···
60  }
```

Figure 2: Encapsulation login_c.java

Encapsulation is demonstrated in login_c class by holding the login logic and user details like selectedRole, enteredUsername and enteredPassword within the class. These fields are private which prevents them from being accessed from the outside. There is no way to talk to the class except through its constructor and the authenticate() method. Secure management of login data is ensured and other program areas are reduced in the risk of making inadvertent adjustments to it.

```
7       private JPanel contentPanel;
8       private JFrame mainFrame;
9
10  >   public am(JFrame frame) {···
50
51  >   private void switchContent(String className) {···
```

Figure 3: Encapsulation am.java

Encapsulation is used by the am class by marking variables such as contentPanel and mainFrame as private. This prevents info from being accessed directly by those outside the class. These variables can be worked with only by the class methods themselves. Another way the class achieves encapsulation is by making the method switchContent() private.

```
16      private JTextField usernameField, passwordField, contactField, emailField;
17      private JComboBox<String> roleComboBox;
18      private JLabel userIdLabel;
19      private JButton saveButton;
```

```
210     // Save a new user using data from the form
211     private void saveNewUser() {
212         String userId = UserController.generateNextUserId();
213         String username = usernameField.getText().trim();
214         String password = passwordField.getText().trim();
215         String role = roleComboBox.getSelectedItem().toString();
216         String contact = contactField.getText().trim();
217         String email = emailField.getText().trim();
```

```
236     private void resetUserInfoForm() {
237         userIdLabel.setText("User ID: " + UserController.generateNextUserId());
238         usernameField.setText(t:"");
239         passwordField.setText(t:"");
240         contactField.setText(t:"");
241         emailField.setText(t:"");
242         roleComboBox.setSelectedIndex(anIndex:0);
243     }
```

Figure 4: Encapsulation user_am.java

Internal class details are hidden and access is offered by making use of public methods in encapsulation. In the user_am class, JTextField, JButton and JTable are marked as private, so they cannot be reached from anywhere outside of the class. Also, functions such as saveNewUser() and resetUserInfoForm() become the main methods for handling data changes and logic. By doing this, the UI's data and methods are protected and access is managed clearly.

```
15      private String email;
16
```

```
67      public String getEmail() {
68          return email;
69      }
70
71      public void setEmail(String email) {
72          this.email = email;
73      }
```

Figure 5: Encapsulation user_am_c.java

User details such as id, username, password, role, contact and email are declared private in the user_c class and so cannot be seen or changed by outside code. Instead, programming the class with public getter and setter methods helps restrict who can access the fields. By doing this, we let validation or necessary logic be run before putting in or fetching values which protects data integrity. An example is to make email access indirect by giving methods like getEmail() and setEmail(String email).

```
12      private JTable financeMainTable;
13      private JTable financeDetailsTable;
14      private DefaultTableModel mainTableModel;
15      private DefaultTableModel detailsTableModel;
16
17      private JTextField searchField;
18      private JButton searchButton, updateButton;
19
20      private List<String[]> financeData = new ArrayList<>();
21      private static final String[] COLUMNS = {
22          "Finance ID", "PO ID", "Approval Status", "Payment Status", "Payment Date", "Amount", "Verified By"
23      };
24
25  >   public finance_am() {···
84
85      private void loadFinanceData() {
86          financeData.clear();
87  >       try (BufferedReader reader = new BufferedReader(new FileReader(fileName:"TXT/finance.txt"))) {···
95  >       } catch (IOException e) {···
98          refreshMainTable();
99      }
```

Figure 6: Encapsulation finance_am.java

In finance_am, private declarations are given to the UI elements and data (table, buttons and finance records). Using this method, other classes are not able to change these elements directly. Managing the interaction is done using controlled functions such as loadFinanceData(), updateApprovalStatus() and searchFinanceID().

```
11      private static final String PR_FILE = "TXT/pr.txt";
12      private JTable prTable, prDetailsTable;
13      private DefaultTableModel tableModel, detailsTableModel;
14      private JButton updateButton;
15      private JTextField searchField;
16      private JButton searchButton;
17
18  >   public pr_pm() {···
79
80      private void loadPRData() {
81          tableModel.setRowCount(rowCount:0);
82  >       try (BufferedReader br = new BufferedReader(new FileReader(PR_FILE))) {···
90  >       } catch (IOException e) {···
93      }
```

Figure 7: Encapsulation pr_pm.java

In pr_pm class, keep the GUI and data-related code hidden and access them using functions such as loadPRData(), searchPR() and updatePRData().

```
 8      private static final int FRAME_BORDER_SIZE = 30;
 9
10  >   public frame() {···
27
28      // Method to switch panels
29      public void switchPanel(JPanel panel) {
30          getContentPane().removeAll();
31          getContentPane().add(panel, BorderLayout.CENTER);
32          revalidate();
33          repaint();
34      }
35  }
```

Figure 8: Encapsulation frame.java

This class achieves encapsulation by keeping its private state like FRAME_BORDER_SIZE and switchPanel() hidden from other classes. Any class outside the frame can call the public method switchPanel(), because it does not need to understand the inner workings of the switch.

```java
public class po_e extends JPanel {
    private static final String PO_FILE = "TXT/po.txt";
    private JTabbedPane tabbedPane;
    private po_e_c poController = new po_e_c();

    // --- PO Info Components ---
    private JTextField prIDField, itemIDField, supplierIDField, quantityField, orderDateField;
    private JComboBox<String> receivedByDropdown, approvedByDropdown;
    private JButton addButton;
    private String loggedInUser;
    private JComboBox<String> prIDDropdown;

    // --- PO List Components ---
    private JTable poTable;
    private DefaultTableModel tableModel;
    private JTextField searchField;
    private JButton searchButton;
    private TableRowSorter<DefaultTableModel> sorter;
    private JPanel detailsPanel;
    private JTextField editPrIdField, editItemIdField, editSupplierIdField, editQuantityField, editOrderDateField;
    private JComboBox<String> editOrderByDropdown, editReceivedByDropdown, editApprovedByDropdown, editStatusDropdown;
    private JTextField editStatusField;
    private JLabel detailStatusLabel;
    private JLabel detailPoIdLabel;
    private List<String[]> fullPoData;
    private String currentPoIdForEdit = null;
    private JLabel detailOrderByLabel;
```

Figure 9: Encapsulation po_e.java

All the user interface components and data fields in the po_e class are declared as private such as the private JTable poTable, the private JTextField prIDField, itemIDField, supplierIDField, quantityField, orderDateField and the private JComboBox<String> prIDDropdown. Because of this limit, outside the class can't access the purchase order panel's variables directly, so every interaction happens through the class's built-in methods and logic.

```java
public class po_e_c {
    // Private field: only accessible within this class
    private static final String PO_FILE = "TXT/po.txt";

    // Public method: provides controlled access to PO data
    public List<String[]> loadPurchaseOrders() {
        List<String[]> fullPoData = new ArrayList<>();
        File file = new File(PO_FILE);
        // ... file reading logic ...
        return fullPoData;
    }

    // Public method: adds a new purchase order
    public boolean addPurchaseOrder(String prID, String itemID, String supplierID, String quantityStr, String orderDate,
                                    String receivedBy, String approvedBy) {
        // ... validation and file writing logic ...
        return true; // or false on error
    }

    // ... other public methods ...
}
```

Figure 10: Encapsulation po_e_c.java

Encapsulation is shown here by declaring all fields as private within the po_e and po_e_c classes, limit direct access from outside. All interactions with these fields are managed through the class's own public methods, keeping the internal state protected and organized.

```java
public class pr_e_c {
    // Private field: only accessible within this class
    private static final String PR_FILE = "TXT/pr.txt";

    // Reference to the view (UI)
    public final pr_e view;

    public pr_e_c(pr_e view) {
        this.view = view;
    }

    // Public method: only way to add a PR from outside
    public void addPR() {
        // ... method logic ...
    }
}
```

Figure 11: Encapsulation pr_e_c.java

This is encapsulation in action because the pr.txt file is declared as private, so it cannot be accessed directly outside the class. All activities related to purchase requisitions such as adding them, are hidden from users by using methods like addPR().

```java
public class pr_e extends JPanel {
    // Private and protected fields
    private static final String PR_INFO_CARD = "INFO";
    public static final String PR_LIST_CARD = "LIST";

    private CardLayout cardLayout;
    private JPanel cardPanel;
    private JPanel topButtonPanel;

    protected JComboBox<String> itemIDComboBox, supplierIDComboBox;
    protected JTextField quantityField, requiredDateField;
    protected JButton addPrButton;

    // ... other fields and methods ...
}
```

Figure 12: Encapsulation pr_e.java

Encapsulation is demonstrated by declaring fields as private or protected in the pr_e class, which limit direct access from outside the class. Dealing with users and events in these fields is done through class methods which protect and order the internal state of the user interface. It means the details of how the system is put together are hidden and access to the data can be managed carefully.

```java
    private static class PORecord {
        private String poId;
        private String prId;
        private String itemId;
        private String supplierId;
        private int quantityOrdered;
        private String orderDate;
        private int orderBy;
        private int receivedBy;
        private int approvedBy;
        private String status;

        public PORecord(String poId, String prId, String itemId, String supplierId,
                        int quantityOrdered, String orderDate, int orderBy,
                        int receivedBy, int approvedBy, String status) {
            this.poId = poId;
            this.prId = prId;
            this.itemId = itemId;
            this.supplierId = supplierId;
            this.quantityOrdered = quantityOrdered;
            this.orderDate = orderDate;
            this.orderBy = orderBy;
            this.receivedBy = receivedBy;
            this.approvedBy = approvedBy;
            this.status = status;
        }

        public String getPoId() { return poId; }
        public String getPrId() { return prId; }
        public String getItemId() { return itemId; }
        public String getSupplierId() { return supplierId; }
        public int getQuantityOrdered() { return quantityOrdered; }
        public String getOrderDate() { return orderDate; }
        public int getOrderBy() { return orderBy; }
        public int getReceivedBy() { return receivedBy; }
        public int getApprovedBy() { return approvedBy; }
        public String getStatus() { return status; }

        public void setStatus(String status) { this.status = status; }
    }
```

Figure 13: Encapsulation po_fm.java

In the figure above, this shows the Object Oriented Programming concept encapsulation being used to managing access to the “PORecord” class’s internal data. Under the class, it declares all the fields such as poID, prId, itemID, etc. as private to prevent external code from accessing or modifying the values directly. Instead, the class provides public getter methods such as “public String getPoID() { return poId; }” to allow controlled read-only access to these values, making sure that the data can be viewed but not modified. For fields that needs to be set as modifiable, (e.g status), PORecord class provides a public setter method “setStatus()” which allows the users to modify the data. This approach allows the class to enforce rules and maintain consistency by bundling the data with the methods that operate on it while hiding the internal implementation details. The setStatus() method can be updated later on without affecting any external code that uses the method. This method enhances security, flexibility, and maintainability, making the code more robust.

```java
private static class InventoryRecord {
    private String inventoryId;
    private String itemId;
    private int stockLevel;
    private String lastUpdated;
    private int receivedQuantity;
    private int updatedBy;
    private String status;

    public InventoryRecord(String inventoryId, String itemId, int stockLevel,
                           String lastUpdated, int receivedQuantity,
                           int updatedBy, String status) {
        this.inventoryId = inventoryId;
        this.itemId = itemId;
        this.stockLevel = stockLevel;
        this.lastUpdated = lastUpdated;
        this.receivedQuantity = receivedQuantity;
        this.updatedBy = updatedBy;
        this.status = status;
    }

    public String getInventoryId() { return inventoryId; }
    public String getItemId() { return itemId; }
    public int getStockLevel() { return stockLevel; }
    public String getLastUpdated() { return lastUpdated; }
    public int getReceivedQuantity() { return receivedQuantity; }
    public int getUpdatedBy() { return updatedBy; }
    public String getStatus() { return status; }

    public void setStatus(String status) { this.status = status; }
}
```

Figure 14: Encapsulation inventory_fm.java

As shown in figure above, The InventoryRecord class encapsulates inventoryId, itemId, stockLevel, lastUpdated, receivedQuantity, updatedBy, and status field by making them private and they are only exposed via public getters such as getInventoryId, getItemId, etc, while limiting modifications to just the status field using a setter “setStatus()”.

```java
public class finance_e extends JPanel {

    private finance_c financeController;
    private JTabbedPane tabbedPane;
    private JPanel financeInfoPanel;
    private JPanel financeListPanel;

    private JTextField financeIdInfoField;
    private JComboBox<String> poIdComboBox;
    private JTextField paymentStatusInfoField;
    private JTextField paymentDateInfoField;
    private JTextField amountInfoField;
    private JButton addButton;
    private JLabel currentUserLabel;
```

```java
public void populateFinanceListTableTop() {
    financeListTableModelTop.setRowCount(rowCount:0);
    for (finance_c.FinanceRecord record : financeController.getFinanceRecords()) {
        financeListTableModelTop.addRow(new Object[]{
            record.getFinanceId(),
            new ButtonPanel(record)
        });
    }
    refreshPoIdsComboBox();
}
```

```java
private void refreshPoIdsComboBox() {
    poIdComboBox.removeAllItems();
    List<String> poIds = financeController.loadPoIds();
    for (String poId : poIds) {
        poIdComboBox.addItem(poId);
    }
}
```

Figure 15: Encapsulation finance_e.java

Finance_e class shows encapsulation by making its field private, preventing direct modifications. It uses public methods to interact with the fields to ensure data integrity. Furthermore, external codes can only be interacted with finance_e class through defined methods only.

```java
public static class FinanceRecord {
    private String financeId;
    private String poId;
    private String approvalStatus;
    private String paymentStatus;
    private String paymentDate;
    private String amount;
    private int verifiedBy;

    public FinanceRecord(String financeId, String poId, String approvalStatus, String paymentStatus,
                         String paymentDate, String amount, int verifiedBy) {
        this.financeId = financeId;
        this.poId = poId;
        this.approvalStatus = approvalStatus;
        this.paymentStatus = paymentStatus;
        this.paymentDate = paymentDate;
        this.amount = amount;
        this.verifiedBy = verifiedBy;
    }

    public String getFinanceId() { return financeId; }
    public String getPoId() { return poId; }
    public String getApprovalStatus() { return approvalStatus; }
    public String getPaymentStatus() { return paymentStatus; }
    public String getPaymentDate() { return paymentDate; }
    public String getAmount() { return amount; }
    public int getVerifiedBy() { return verifiedBy; }

    public void setPoId(String poId) { this.poId = poId; }
    public void setApprovalStatus(String approvalStatus) { this.approvalStatus = approvalStatus; }
    public void setPaymentStatus(String paymentStatus) { this.paymentStatus = paymentStatus; }
    public void setPaymentDate(String paymentDate) { this.paymentDate = paymentDate; }
    public void setAmount(String amount) { this.amount = amount; }
    public void setVerifiedBy(int verifiedBy) { this.verifiedBy = verifiedBy; }
}
```

Figure 16: Encapsulation finance_c.java

As shown in figure above, FinanceRecord class shows encapsulation by making the data fields as private. Through public getters and setters, it allows controlled access, improving integrity and more flexible to future modifications.

```java
    BOON888, 2 months ago • meeting
private void loadInventoryData() {
    inventoryRecords.clear();
    try (BufferedReader br = new BufferedReader(new FileReader(INVENTORY_FILE))) {
        String line;
        while ((line = br.readLine()) != null) {
            String[] data = line.split(regex:"\\|");
            if (data.length == 7) {
                try {
                    String inventoryId = data[0].trim();
                    String itemId = data[1].trim();
                    int currentStock = Integer.parseInt(data[2].trim());
                    String lastUpdated = data[3].trim();
                    int reorderLevel = Integer.parseInt(data[4].trim());
                    int updatedBy = Integer.parseInt(data[5].trim());
                    String status = data[6].trim();

                    InventoryRecord record = new InventoryRecord(inventoryId, itemId, currentStock, lastUpdated, reorderLevel, updatedBy, status);
                    inventoryRecords.add(record);
                } catch (NumberFormatException e) {
                    System.err.println("Error parsing data in line (inventory.txt): " + line);
                }
            } else {
                System.err.println("Skipping invalid line in inventory.txt: " + line + ". Expected 7 columns.");
            }
        }
    } catch (IOException e) {
        JOptionPane.showMessageDialog(this, "Error reading inventory file: " + e.getMessage(), title:"Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 17: Encapsulation inventory_e.java

The data in the inventory_e class is encapsulated by marking its table model (tableModel) and JTable (inventoryTable) as private fields. So, other parts of the system cannot touch the table's

structure or change the way inventory is put in. loading the inventory data happens safely within the private loadInventoryData() method. This method gets called earlier, only within the initialization, to allow secure changes to the inventory data and keep the display accurate.

```
149     private void clearFields() {
150         itemIdInfoComboBox.setSelectedIndex(-1);
151         lastUpdatedInfoField.setText(t:"DD-MM-YYYY");
152         rQuantityInfoField.setText(t:"Number");
153         stockLevelInfoField.setText(t:"");
154     }
155     You, 2 months ago • really done liao …
156     private int getStockLevelFromItems(String itemId) {
157         try (BufferedReader br = new BufferedReader(new FileReader(ITEMS_FILE))) {
158             String line;
159             while ((line = br.readLine()) != null) {
160                 String[] data = line.split(regex:"\\|");
161                 if (data.length == 6 && data[0].trim().equals(itemId)) {
162                     return Integer.parseInt(data[5].trim());
163                 }
164             }
165         } catch (IOException | NumberFormatException e) {
166             System.err.println("Error reading items file or parsing stock level: " + e.getMessage());
167             e.printStackTrace();
168         }
169         return 0;
170     }
171 }
```

Figure 18: Encapsulation inventory_c.java

External classes cannot directly change the form elements or stock data since inventory_c stores them as private fields (itemComboBox, quantityField, stockMap). Components are controlled by public methods or by adding event listeners. By doing this, fewer bugs occur and internal consistency is ensured such as preventing stock data from becoming invalid.

```
58      private DefaultTableModel tableModel;
59      private JTable table;
60      private TableRowSorter<DefaultTableModel> sorter;
61      private JTextField searchTextField;
```

```
155     // Add listener for the search bar
        BOON888, 2 weeks ago | 1 author (BOON888)
156     searchTextField.addActionListener(new ActionListener() {
157         @Override
158         public void actionPerformed(ActionEvent e) {
159             String text = searchTextField.getText();
160             if (text.trim().length() == 0) {
161                 sorter.setRowFilter(filter:null); // Show all rows if the search field is empty
162             } else {
163                 sorter.setRowFilter(RowFilter.regexFilter("(?i)" + text)); // Filter rows case-insensitively
164             }
165         }
166     });
167 }
```

Figure 19: Encapsulation inventory_v.java

The class contains the table model, table and search field as private members to stop external access. The method filterTable() is privately called to handle how the table is filtered in response to any changes in the search field. Thanks to this design, internal processing and status are protected against accidental mistakes or interruptions.

```
92  private void readInventoryAndSalesDates() {      BOON888, 6 days ago • inventory report done
93      SimpleDateFormat inputDateFormat = new SimpleDateFormat(pattern:"dd-MM-yyyy");
94
95      // Read inventory.txt for dates
96      if (new File(INVENTORY_FILE).exists()) {
97          try (BufferedReader br = new BufferedReader(new FileReader(INVENTORY_FILE))) {
98              String line;
99              while ((line = br.readLine()) != null) {
100                 String[] parts = line.split(regex:"\\|");
101                 // inventory.txt: inventory_id|item_id|stock_level|last_updated|received_quantity|updated_by|sta
102                 if (parts.length >= 4) { // We need at least the last_updated date
103                     try {
104                         Date date = inputDateFormat.parse(parts[3].trim()); // last_updated is at index 3
105                         activityDates.put(date, value:""); // Value doesn't matter, just need the date
106                     } catch (ParseException e) {
107                         System.err.println("Error parsing date in " + INVENTORY_FILE + ": '" + parts[3].trim() +
108                     }
109                 }
110             }
111         } catch (IOException e) {
112             JOptionPane.showMessageDialog(this, "Error reading " + INVENTORY_FILE + ": " + e.getMessage(), title
113         }
114     } else {
115         System.out.println(INVENTORY_FILE + " not found. Skipping inventory data.");
116     }
117
```

Figure 20: Encapsulation inventory_r.java

Inventory_r and InventoryReport keep their workings private and expose only what users of the inventory_r need by through their constructors and methods. Each method is responsible for reading data, processing it and making UI (user interface) components which keeps things private and helps with modularity. As an example, methods like readInventoryAndSalesDates() and processDailyInventory() move the file processing code to just one place, so other parts of the program do not need to worry about the implementation.

```
22      private TreeMap<Date, String> monthYearData = new TreeMap<>(new Comparator<Date>() {
23          SimpleDateFormat monthYearFormat = new SimpleDateFormat(pattern:"yyyyMM");
24
25          @Override
26          public int compare(Date date1, Date date2) {
27              return monthYearFormat.format(date2).compareTo(monthYearFormat.format(date1));
28          }
29      });
```

```
89      private void readFinanceAndSalesData() {
90          SimpleDateFormat inputDateFormat = new SimpleDateFormat(pattern:"dd-MM-yyyy");
91          String financeFilePath = "TXT/finance.txt";
92          String salesFilePath = "TXT/sales_data.txt";
93
94          if (new File(financeFilePath).exists()) {
95              try (BufferedReader br = new BufferedReader(new FileReader(financeFilePath))) {
96                  String line;
97                  while ((line = br.readLine()) != null) {
98                      String[] parts = line.split(regex:"\\|");
99                      if (parts.length >= 5 && parts[3].equals(anObject:"Paid")) {
100                         try {
101                             Date date = inputDateFormat.parse(parts[4]);
102                             monthYearData.put(date, monthYearData.getOrDefault(date, defaultValue:"") + "Finance: "
103                         } catch (ParseException e) {
104                             System.err.println("Error parsing date in finance.txt: " + e.getMessage());
105                         }
106                     }
107                 }
108             } catch (IOException e) {
109                 JOptionPane.showMessageDialog(this, "Error reading finance.txt: " + e.getMessage());
110             }
111         } else {
112             JOptionPane.showMessageDialog(this, message:"finance.txt not found.");
113         }
```

```
324         private Map<String, Double> readItemPrices() {
325             Map<String, Double> itemPrices = new HashMap<>();
326             try (BufferedReader br = new BufferedReader(new FileReader(fileName:"TXT/items.txt"))) {
327                 String line;
328                 while ((line = br.readLine()) != null) {
329                     String[] parts = line.split(regex:"\\|");
330                     if (parts.length >= 5) {
331                         itemPrices.put(parts[0], Double.parseDouble(parts[4]));
332                     }
333                 }
334             } catch (IOException | NumberFormatException e) {
335                 JOptionPane.showMessageDialog(this, "Error reading items.txt: " + e.getMessage());
336             }
337             return itemPrices;
338         }
339     }
340 }
```

Figure 21: Encapsulation finance_r.java

The finance_r class puts its data in private places, like monthYearData and makes use of private methods like readFinanceAndSalesData() and readItemPrices(). Other parts of the program do not see the data or file reading methods; only intended interfaces are shared. Since it keeps its internal data hidden, the class makes it harder for someone to misuse the code and guarantees that changes are managed in a reliable way.

```java
public class Supplier implements Serializable {  ccc0315
    private int id;  2 usages
    private String name;  3 usages
    private String contact;  3 usages
    private String email;  3 usages
    private String address;  3 usages
    private String item;  3 usages

    public Supplier(int id, String name, String contact, String email, String address, String item) {  ccc0315
        this.id = id;
        this.name = name;
        this.contact = contact;
        this.email = email;
        this.address = address;
        this.item = item;
    }

    // Getters
    public int getId() { return id; }  ccc0315
    public String getName() { return name; }  ccc0315
    public String getContact() { return contact; }  no usages  ccc0315
    public String getEmail() { return email; }  no usages  ccc0315
    public String getAddress() { return address; }  no usages  ccc0315
    public String getItem() { return item; }  ccc0315

    // Setters
    public void setName(String name) { this.name = name; }  ccc0315
    public void setContact(String contact) { this.contact = contact; }  no usages  ccc0315
    public void setEmail(String email) { this.email = email; }  no usages  ccc0315
    public void setAddress(String address) { this.address = address; }  no usages  ccc0315
    public void setItem(String item) { this.item = item; }  ccc0315
```

Figure 22: Encapsulation supplier.java

The structure of an object-oriented Supplier class perfectly illustrates encapsulation. With this technique, the class's data elements are hidden from direct view. Instead, interact with and alter them solely via public getter and setter methods. Designating fields as private prevents other classes from inadvertently corrupting Supplier objects. Publicly available methods like getName() and setEmail() are responsible for accessing and changing these elements, and they also offer a convenient place to incorporate any necessary validations or custom logic. This design decision leads to stronger data protection, more predictable system behavior, and easier code management when modifications are required.

```
public class supplier_e extends JPanel {  1 usage  ccc0315 +1

    private JTextField nameField, contactField, emailField, addressField, itemField;  4 usages
    private JTable supplierTable;  5 usages
    private DefaultTableModel tableModel;  22 usages
    private static final String FILE_NAME = "TXT/suppliers.txt";  3 usages
    private int currentId = 3001;  3 usages
```

Figure 23: Encapsulation supplier_e.java

Examples of encapsulation include using private variables such as nameField, contactField, emailField, addressField, itemField, supplierTable, tableModel, FILE_NAME and currentId in the supplier_e class. The idea of encapsulation is to keep the internal working of an object private and force all communication to go through its methods. If these fields are private, only approved functions like addSupplier() or loadSuppliers() can add or change their data. With this, the data can't be altered by chance, remains correct and is easier for programmers to update.

```
public class supplier_v extends JDialog {  2 usages  ccc0315 +1
    private JTextField nameField, contactField, emailField, addressField, itemField;  3 usages
    private boolean saved = false;  2 usages
```

Figure 24: Encapsulation supplier_v.java

Encapsulation is shown in the `supplier_v` class by setting the variables `nameField`, `contactField`, `emailField`, `addressField`, `itemField` and `saved` to `private`. That means users from outside the class cannot get to or edit these fields. Most of the time, access to fields is given through public methods called getters or setters. Having this encapsulation means the internal details of the dialog (such as user data and save status) are safe from outside changes which maintains the code's security and helps improve its modularity and maintenanc

```java
public class supplier_list extends JPanel {  no usages  ccc0315
    private static final String SUPPLIER_FILE = "TXT/suppliers.txt";  3 usages
    private static final int MARGIN = 20;  4 usages
    private static final Font GLOBAL_FONT = new Font( name: "Arial", Font.PLAIN,  size: 16);  1 usage
    private static final Font HEADER_FONT = new Font( name: "Arial", Font.BOLD,  size: 18);  1 usage
    private static final Font TITLE_FONT = new Font( name: "Arial", Font.BOLD,  size: 25);  1 usage

    private static class Supplier {  5 usages  ccc0315
        private String id, name, phone, email, address, category;  2 usages

        public Supplier(String id, String name, String phone, String email, String address, String category) {  1 usage
            this.id = id;
            this.name = name;
            this.phone = phone;
            this.email = email;
            this.address = address;
            this.category = category;
        }

        public String getId() { return id; }  1 usage  ccc0315
        public String getName() { return name; }  1 usage  ccc0315
        public String getPhone() { return phone; }  1 usage  ccc0315
        public String getEmail() { return email; }  1 usage  ccc0315
        public String getAddress() { return address; }  1 usage  ccc0315
        public String getCategory() { return category; }  1 usage  ccc0315
    }
```

Figure 25: Encapsulation supplier_list.java

Within the supplier_list class, superspecialization is used by declaring a private static inner class called Supplier and putting all its attributes—id, name, phone, email, address and category—as private. So, these items in the Supplier class aren't available for use directly by other code. As an alternative, access to data is only possible through public getter methods which can be called using getId(), getName() and other similar names. As a result, the data is protected and is altered only through approved methods. It supports clear-looking and low-error code, as the information is protected to be accessed only by authorized parts of the program.

```java
public class Item implements Serializable {  ccc0315
    private String id;  3 usages
    private String name;  4 usages
    private String supplierId;  4 usages
    private String category;  4 usages
    private double price;  4 usages
    private int stockQuantity;  4 usages

    public Item(String id, String name, String supplierId, String category, double price, int stockQuantity) {  ccc0315
        this.id = id;
        this.name = name;
        this.supplierId = supplierId;
        this.category = category;
        this.price = price;
        this.stockQuantity = stockQuantity;
    }

    // Getters
    public String getId() { return id; }  ccc0315
    public String getName() { return name; }  ccc0315
    public String getSupplierId() { return supplierId; }  8 usages  ccc0315
    public String getCategory() { return category; }  ccc0315
    public double getPrice() { return price; }  4 usages  ccc0315
    public int getStockQuantity() { return stockQuantity; }  4 usages  ccc0315

    // Setters
    public void setName(String name) { this.name = name; }  ccc0315
    public void setSupplierId(String supplierId) { this.supplierId = supplierId; }  1 usage  ccc0315
    public void setCategory(String category) { this.category = category; }  ccc0315
    public void setPrice(double price) { this.price = price; }  1 usage  ccc0315
    public void setStockQuantity(int stockQuantity) { this.stockQuantity = stockQuantity; }  1 usage  ccc0315
```

Figure 26: Encapsulation item.java

Encapsulation is carried out in the Item class by making all attributes (id, name, supplierId, category, price and stockQuantity) private, so they can't be accessed directly by code outside the class. In fact, access and modification are limited using getter and setter methods that are accessible to all. When we want to access the item's name, we call getName() and to set or update its name, we pass a String to setName(). Thanks to this design, the data inside the Item class is confined to intended interactions and changes. Because all interactions with the data use set interfaces, encapsulation allows the code to be maintained and debugged more easily.

```java
public class item_e extends JPanel {  2 usages  ccc0315 +1

    private JTextField nameField, supplierIdField, supplierNameField, categoryField, priceField, stockQuantityField;
    private JTable itemTable;  9 usages
    private DefaultTableModel tableModel;  11 usages
    private static final String FILE_NAME = "TXT/items.txt";  3 usages
    private static final String SUPPLIER_FILE = "TXT/suppliers.txt";  4 usages
    private List<Item> items = new ArrayList<>();  8 usages
    private int nextId = 2000;  4 usages
```

Figure 27: Encapsulation item_e.java

Encapsulation in the `item_e` class is demonstrated by setting `nameField`, `supplierIdField`, `itemTable`, `items` and others as private variables. To protect the data and stop it from being modified accidentally, Java makes these fields private. By using this design, just the methods inside `item_e` can work directly with these fields, helping to control them, ease debugging and maintain the code better. By doing this, the logic is organized and it's not possible for unknown code to impact the private data within the class.

```java
public class Sales implements Serializable {  10 usages  ccc0315
    private String salesId;  3 usages
    private String itemId;  3 usages
    private LocalDate salesDate;  3 usages
    private int quantitySold;  3 usages
    private int remainingStock;  3 usages
    private String salesPerson;  3 usages

    public Sales(String salesId, String itemId, LocalDate salesDate, int quantitySold, int remainingStock, String salesPerson) {
        this.salesId = salesId;
        this.itemId = itemId;
        this.salesDate = salesDate;
        this.quantitySold = quantitySold;
        this.remainingStock = remainingStock;
        this.salesPerson = salesPerson;
    }

    // Getters
    public String getSalesId() { return salesId; }  no usages  ccc0315
    public String getItemId() { return itemId; }  ccc0315
    public LocalDate getSalesDate() { return salesDate; }  no usages  ccc0315
    public int getQuantitySold() { return quantitySold; }  no usages  ccc0315
    public int getRemainingStock() { return remainingStock; }  no usages  ccc0315
    public String getSalesPerson() { return salesPerson; }  no usages  ccc0315
```

Figure 28: Encapsulation sales.java

The Sales class provides a clear example of encapsulation in action. All its core variables—salesId, itemId, salesDate, quantitySold, remainingStock, and salesPerson—are marked as private, meaning they can't be directly accessed from outside the class. Instead, the class offers public methods specifically for reading this data, ensuring every interaction with the data is carefully controlled.

```
public class sales_e extends JPanel {  2 usages  ccc0315 +1

    private JTextField itemNameField, quantityField;  7 usages
    private JComboBox<String> salesPersonCombo;  5 usages
    private JTable salesTable;  7 usages
    private DefaultTableModel tableModel;  6 usages
    private static final String SALES_FILE = "TXT/sales_data.txt";  2 usages
    private static final String ITEMS_FILE = "TXT/items.txt";  6 usages
    private static final String USERS_FILE = "TXT/users.txt";  2 usages
    private List<Sales> salesList = new ArrayList<>();  8 usages
    private List<String[]> itemsList = new ArrayList<>();  5 usages
    private List<String> salesPersons = new ArrayList<>();  4 usages
    private int nextSalesId = 4000;  4 usages
    private static final DateTimeFormatter DISPLAY_FORMAT = DateTimeFormatter.ofPattern("dd-MM-yyyy");  5 usages
```

Figure 29: Encapsulation sales_e.java

The `sales_e` class uses encapsulation to make sure that data like `itemNameField`, `quantityField` and both file paths, are private. With this design, outside users can only change or see the inside state of the object using the provided methods. The `salesList`, `itemsList` and `salesPersons` lists observed by the class make sure that only the code within the class can access and update their contents. Because of this design, the program is safer, simpler to upgrade and less complex to maintain.

```
public class sales_v extends JPanel {  no usages  ccc0315 +1
    private static final String SALES_FILE = "TXT/sales_data.txt";  1 usage
    private static final String ITEMS_FILE = "TXT/items.txt";  1 usage
    private DefaultTableModel tableModel;  4 usages
    private JTable table;  6 usages
    private TableRowSorter<DefaultTableModel> sorter;  4 usages
    private JTextField searchTextField;  5 usages
    private static final Font GLOBAL_FONT = new Font( name: "Arial", Font.PLAIN,  size: 16);  3 usages
    private static final Font HEADER_FONT = new Font( name: "Arial", Font.BOLD,  size: 18);  1 usage
    private static final Font TITLE_FONT = new Font( name: "Arial", Font.BOLD,  size: 25);  1 usage
    private static final int MARGIN = 20;  2 usages
```

Figure 30: Encapsulation sales_v.java

In `sales_v`, its variables are marked as `private` which is an example of encapsulation, as shown by `SALES_FILE`, `ITEMS_FILE`, `tableModel`, `table`, `sorter` and `searchTextField`. So, these fields can't be altered or read from another class, helping to keep the class's state reliable. With the data stored in its own file paths, fonts and UI components, the class stops any external method from modifying or using it. This approach encourages programmers to write flexible, safe and convenient code.

3.2 Inheritance

```
5   public class login extends JPanel {
```

Figure 31: Inheritance login.java

Login class extends JPanel and gets all layout and component features from Java Swing. Therefore, this allow login operates like any Swing panel and can simply be put into the frame, taking advantage of the existing GUI features.

```
7   public class user_am extends JPanel {
```

```
298     class ButtonEditor extends DefaultCellEditor {
299
300         private final JPanel panel;
301         private final JButton viewButton;
302         private final JButton deleteButton;
303         private String userId;
304
305 >       public ButtonEditor(user_am parent) {···
335
336         @Override
337 >       public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int row, int column) {···
341
342         @Override
343 >       public Object getCellEditorValue() {···
346     }
```

Figure 32: Inheritance user_am.java

In user_am class, the code mentions that it extends JPanel which is a Swing type of container. Because of the inheritance, user_am can appear like a JPanel, making use of layout managers, adding components and joining frames. In addition, the ButtonEditor inner class is derived from DefaultCellEditor and now allows editing cells in JTables, while adding buttons that will either show or remove items.

```
6   public class frame extends JFrame {
7
8       private static final int FRAME_BORDER_SIZE = 30;
9
10      public frame() {
11          setTitle(title:"JAVA FRAME");
12          setSize(width:1920, height:1080); //scale 100%
13          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14          setLocationRelativeTo(c:null);
```

Figure 33: Inheritance frame.java

In this class, the code extends JFrame which is part of the Swing framework. Thus, frame class will automatically get all the methods and properties of JFrame like setTitle() and can then change or improve them. Although this example uses a Java library, it clearly shows examples of inheritance.

```
public class pr_e extends JPanel {
    // ... class body ...
}
```

```
class ButtonEditor extends AbstractCellEditor implements TableCellEditor {
    // ... class body ...
}
```

Figure 34: Inheritance pr_e.java

Inheritance is demonstrated by having the pr_e class extend JPanel, which means pr_e inherits all the properties and methods of the JPanel class from the Java Swing library. This allows pr_e to be used as a panel in a graphical user interface and to override or extend its behavior. In a similar way, ButtonEditor extends AbstractCellEditor and implements TableCellEditor which allows it to customize how editing works in table cells. This way, code is reused, and different specialized components can be made using already existing classes.

```
private class ButtonEditor extends DefaultCellEditor {
    private ButtonPanel panel;

    public ButtonEditor(JTable table) {
        super(new JTextField());
        setClickCountToStart(count:1);
    }

    @Override
    public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int row, int column) {
        if (value instanceof ButtonPanel) {
            panel = (ButtonPanel) value;
        }
        return panel;
    }

    @Override
    public Object getCellEditorValue() {
        return panel;
    }

    @Override
    public boolean isCellEditable(java.util.EventObject e) {
        return true;
    }

    @Override
    public boolean shouldSelectCell(java.util.EventObject anEvent) {
        return false;
    }
}
```

Figure 35: Inheritance po_fm.java

As shown in figure above, the ButtonEditor class extends to DefaultCellEditor which means that it inherits from Swing's DefaultCellEditor, allowing it to reuse its core cell-editing functions while customizing it for further button interactions. By overriding "getTableCellEditorComponent()", it maintains compatibility with JTable while implementing a specialized editing behaviour. Furthermore, the "super(new JTextField())" call ensures proper

parent class initialization. This inheritance example enables code reuse without modifying base functionality and provides a smooth Swing integration while adding new features.

```
public class inventory_fm extends JPanel {
```

Figure 36: Inheritance inventory_fm.java

As shown in the figure above, inventory_fm class shows inheritance by extending to Swing's JPanel. By inheriting and using panel behaviour from JPanel, inventory_fm class will then be able to avoid rewriting basic panel behaviour and can add additional features instead.

```
private class ButtonRenderer extends DefaultTableCellRenderer {

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column) {
        if (value instanceof ButtonPanel) {
            return (ButtonPanel) value;
        }
        return new JLabel();
    }
}
```

Figure 37: Inheritance finance_e.java

In the figure above, ButtonRenderer inherits from Swing's DefaultTableCellRenderer, overriding its getTableCellRendererComponent(…) method to manage custom button rendering while keeping default behaviour for other scenarios. This avoids code duplication by reusing existing parent class logic and using method overriding to add additional features.

```
69      public inventory_e(List<InventoryRecord> inventoryRecords, Map<String, ItemDetails> itemDetailsMap) {
70          this.inventoryRecords = inventoryRecords;
71          this.itemDetailsMap = itemDetailsMap;
72          loadItemDetailsForDropdown();
73          setLayout(new BorderLayout());
74
75          tabbedPane = new JTabbedPane();
76          Font tabTitleFont = new Font(name:"Arial", Font.BOLD, size:20);
77          tabbedPane.setFont(tabTitleFont);
78
79          inventoryInfoPanel = createInventoryInfoPanel();
80          tabbedPane.addTab(title:"Inventory Info", inventoryInfoPanel);
81
82          inventoryListPanel = createInventoryListPanel();
83          tabbedPane.addTab(title:"Inventory List", inventoryListPanel);
84
85          add(tabbedPane, BorderLayout.CENTER);
86
87          loadInventoryData();
88          populateInventoryListTableTop();
89      }
```

Figure 38: Inheritance inventory_e.java

This class is a subclass of JPanel which is a Swing container, so it behaves like a panel component that can add to any Swing window or container. Because of this, inventory_e does not have to start from the beginning or test the basics for containers and user interfaces; it builds

on a reliable class. Using super constructors or methods like paintComponent are usual ways to use inheritance when required.

```
public class inventory_c extends JPanel {

    private static final String INVENTORY_FILE = "TXT/inventory.txt";
    private static final String ITEMS_FILE = "TXT/items.txt";

    private JComboBox<String> itemIdInfoComboBox;
    private JTextField lastUpdatedInfoField;
    private JTextField rQuantityInfoField;
    private JTextField stockLevelInfoField;
    private JButton addButton;

    private List<inventory_e.InventoryRecord> inventoryRecords;
    private Map<String, inventory_e.ItemDetails> itemDetailsMap;
    private inventory_e parentPanel;

    public inventory_c(List<inventory_e.InventoryRecord> inventoryRecords, Map<String, inventory_e.ItemDetails> itemDetailsMap, inventory_e parentPanel) {
        this.inventoryRecords = inventoryRecords;
        this.itemDetailsMap = itemDetailsMap;
        this.parentPanel = parentPanel;
        setLayout(new GridBagLayout());
        initComponents();
    }
```

Figure 39: Inheritance inventory_c.java

The class is based on JPanel and implements ActionListener, so it is a specialized panel and can handle user actions. Because of this, it can place a Swing component in any container and connect events to it automatically by adding itself as a listener, so there is less code to write.

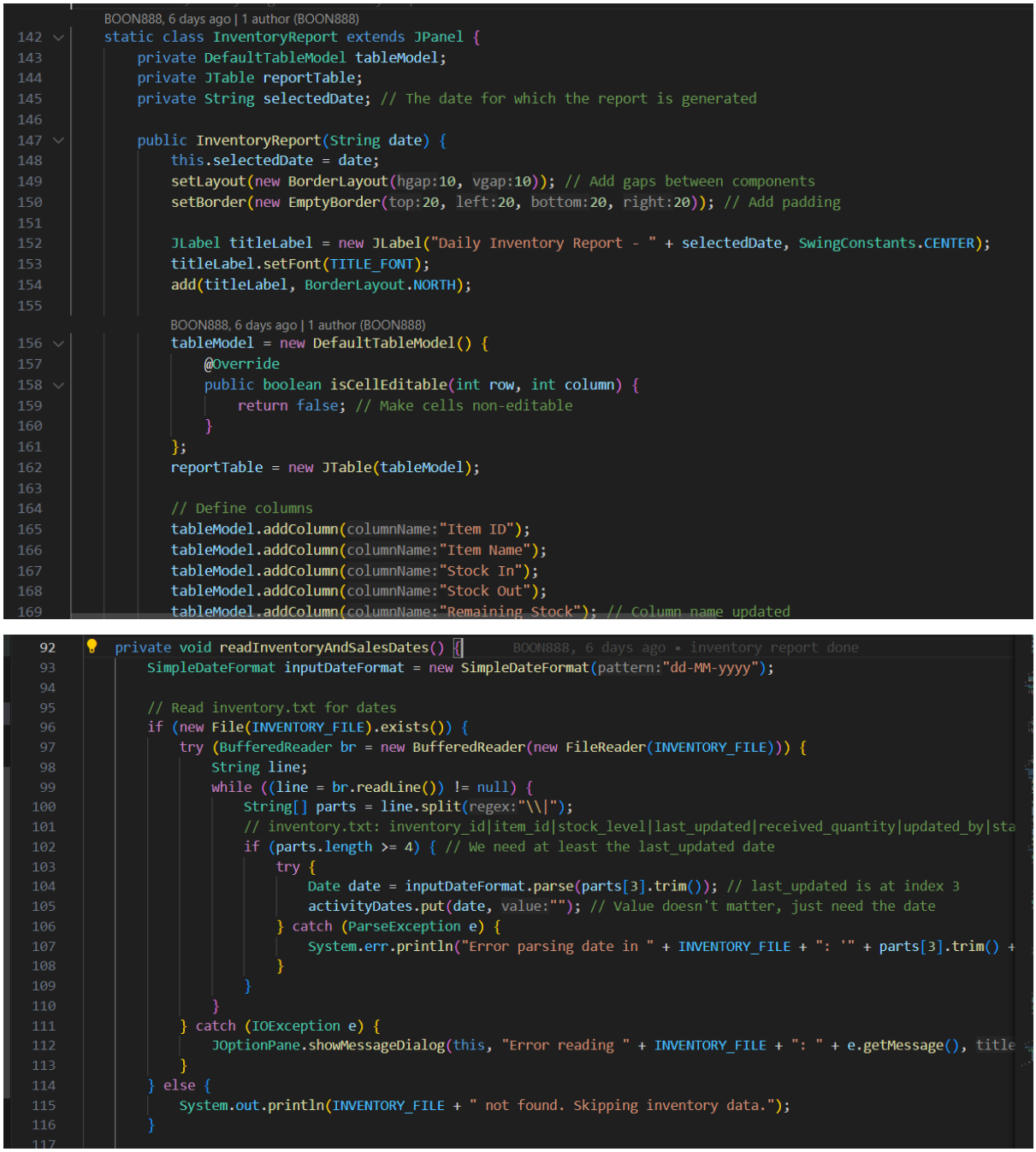
```
BOON888, 2 weeks ago | 2 authors (BOON888 and one other)
public class inventory_v extends JPanel {

    private static final String INVENTORY_FILE = "TXT/inventory.txt"; // Path to the inventory file
    private static final Font GLOBAL_FONT = new Font(name:"Arial", Font.PLAIN, size:16);
    private static final Font HEADER_FONT = new Font(name:"Arial", Font.BOLD, size:18);
    private static final Font TITLE_FONT = new Font(name:"Arial", Font.BOLD, size:25);
    private static final int MARGIN = 20; // Consistent margin

    BOON888, 2 months ago | 1 author (BOON888)
    private static class InventoryItem {
        private int inventoryId;
        private int itemId;
        private int stockLevel;
        private Date lastUpdated;
        private int receivedQuantity;
        private int updatedBy;
        private String status;      BOON888, 2 months ago • inventory_v done
```

Figure 40: Inheritance inventory_v.java

By extending JPanel, this class can be a Swing component panel, making it possible to use and embed it in several different GUI layouts. It uses every aspect of JPanel plus extra functions for looking at and filtering inventory.



```
BOON888, 6 days ago | 1 author (BOON888)
static class InventoryReport extends JPanel {
    private DefaultTableModel tableModel;
    private JTable reportTable;
    private String selectedDate; // The date for which the report is generated

    public InventoryReport(String date) {
        this.selectedDate = date;
        setLayout(new BorderLayout(hgap:10, vgap:10)); // Add gaps between components
        setBorder(new EmptyBorder(top:20, left:20, bottom:20, right:20)); // Add padding

        JLabel titleLabel = new JLabel("Daily Inventory Report - " + selectedDate, SwingConstants.CENTER);
        titleLabel.setFont(TITLE_FONT);
        add(titleLabel, BorderLayout.NORTH);

        BOON888, 6 days ago | 1 author (BOON888)
        tableModel = new DefaultTableModel() {
            @Override
            public boolean isCellEditable(int row, int column) {
                return false; // Make cells non-editable
            }
        };
        reportTable = new JTable(tableModel);

        // Define columns
        tableModel.addColumn(columnName:"Item ID");
        tableModel.addColumn(columnName:"Item Name");
        tableModel.addColumn(columnName:"Stock In");
        tableModel.addColumn(columnName:"Stock Out");
        tableModel.addColumn(columnName:"Remaining Stock"); // Column name updated
```

```
92  private void readInventoryAndSalesDates() {      BOON888, 6 days ago • inventory report done
93      SimpleDateFormat inputDateFormat = new SimpleDateFormat(pattern:"dd-MM-yyyy");
94
95      // Read inventory.txt for dates
96      if (new File(INVENTORY_FILE).exists()) {
97          try (BufferedReader br = new BufferedReader(new FileReader(INVENTORY_FILE))) {
98              String line;
99              while ((line = br.readLine()) != null) {
100                 String[] parts = line.split(regex:"\\|");
101                 // inventory.txt: inventory_id|item_id|stock_level|last_updated|received_quantity|updated_by|sta
102                 if (parts.length >= 4) { // We need at least the last_updated date
103                     try {
104                         Date date = inputDateFormat.parse(parts[3].trim()); // last_updated is at index 3
105                         activityDates.put(date, value:""); // Value doesn't matter, just need the date
106                     } catch (ParseException e) {
107                         System.err.println("Error parsing date in " + INVENTORY_FILE + ": '" + parts[3].trim() +
108                     }
109                 }
110             }
111         } catch (IOException e) {
112             JOptionPane.showMessageDialog(this, "Error reading " + INVENTORY_FILE + ": " + e.getMessage(), title
113         }
114     } else {
115         System.out.println(INVENTORY_FILE + " not found. Skipping inventory data.");
116     }
117
```

Figure 41: Inheritance inventory_r.java

Inheritance allows inventory_r to use the features and methods of the Swing panel class by inheriting from JPanel. This feature lets inventory_r act like a personalized panel inside an overall GUI application, at the same time offering new ways to cover inventory reports functions, for example by displaying a dynamically updated view or making the content scrollable.

```
18  public class finance_r extends JPanel {
19
20      private static final Font TITLE_FONT = new Font(name:"Arial", Font.BOLD, size:25);
21      private static final Font CONTENT_FONT = new Font(name:"Arial", Font.PLAIN, size:18);
        BOON888, 2 months ago | 2 authors (BOON888 and one other)
22      private TreeMap<Date, String> monthYearData = new TreeMap<>(new Comparator<Date>() {
23          SimpleDateFormat monthYearFormat = new SimpleDateFormat(pattern:"yyyyMM");
24
25          @Override
26          public int compare(Date date1, Date date2) {
27              return monthYearFormat.format(date2).compareTo(monthYearFormat.format(date1));
28          }
29      });
```

Figure 42: Inheritance finance_r.java

The finance_r class extends JPanel and gets all the properties and methods necessary to operate inside the Swing GUI framework. The connection means that finance_r can be used easily as a panel in a larger application, since it uses existing tools for layout, event management and rendering, helping to keep code consistent.

```
private static class IntegerDocument extends PlainDocument {  1 usage  ccc0315 +1
    @Override  ccc0315 +1
    public void insertString(int offset, String str, AttributeSet attr) throws BadLocationException {
        if (str == null) {
            return;
        }

        for (int i = 0; i < str.length(); i++) {
            if (!Character.isDigit(str.charAt(i))) {
                return;
            }
        }
        super.insertString(offset, str, attr);
    }
}
```

Figure 43: Inheritance sales_e.java

A class named IntegerDocument which is based on PlainDocument. This marks that IntegerDocument inherits from PlainDocument, so it adds specific features to it. It keeps the features of PlainDocument but changes the insertString method, permitting only numbers to be added, so the document holds only integer values.

3.3 Polymorphism

```
73      private JPanel getRolePanel(String role, frame mainFrame) {
74          switch (role) {
75              case "AM": ···
77              case "FM": ···
79              case "IM": ···
81              case "SM": ···
83              case "PM": ···
85              default: ···
87          }
88      }
```

Figure 44: Polymorphism login.java

In the getRolePanel() method, each role-based panel (am, fm, im, sm, pm) is made into a common type JPanel. It means mainFrame.switchPanel(...) can handle any role panel, no matter its details, as long as it is a subclass of JPanel. All roles get the same treatment when swap, even if each role panel behaves differently.

```
51      private void switchContent(String className) {
52          try {
53              contentPanel.removeAll();
54              Class<?> clazz = Class.forName(className);
55              JPanel panel = (JPanel) clazz.getDeclaredConstructor().newInstance();
56              contentPanel.add(panel, BorderLayout.CENTER);
57              contentPanel.revalidate();
58              contentPanel.repaint();
59          } catch (Exception ex) {
60              ex.printStackTrace();
61              JOptionPane.showMessageDialog(parentComponent:null, "Error opening " + className, title:"Error", JOptionPane.ERROR_MESSAGE);
62          }
63      }
```

Figure 45: Polymorphism am.java

In the switchContent() method are making different panel classes dynamically depend on the class name. At compilation, the specific type is unknown, though every new panel is handled as a JPanel. This case illustrates polymorphism which allows various classes (user_am, item_e, finance_am, etc.) to be accessed as a single JPanel.

```
289         @Override
290         public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column) {
291             JPanel panel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
292             panel.add(new JButton(text:"View"));
293             panel.add(new JButton(text:"Delete"));
294             return panel;
295         }
296     }
```

Figure 46: Polymorphism user_am.java

In the user_am class code, polymorphism is shown by making the ButtonRenderer class provide a way to render buttons differently which appears in the JTable. They also use polymorphism, since rather than using a separate class, ActionListener is implemented using

lambda expressions and when a button is clicked, it triggers a different action even though all buttons use the same interface.

```
110             bw.write(user.toFileString());
```

Figure 47: Polymorphism user_c.java

User_c has methods such as write() or readLine(). Java determines which version of the method to apply according to the object it is assigned to. As another example of polymorphism, use the same toFileString() method everywhere, since the method name will be reused if the class is subclassed.

```
169         @Override
170         public Component getTableCellRendererComponent(JTable table, Object value,
171                 boolean isSelected, boolean hasFocus, int row, int column) {
172             return this;
```

```
197         @Override
198         public Component getTableCellEditorComponent(JTable table, Object value,
199                 boolean isSelected, int row, int column) {
200             financeId = table.getValueAt(row, column:0).toString();  // Get ID
201             return panel;
202         }
```

Figure 48: Polymorphism finance_am.java

This code utilizes polymorphism by making the ButtonRenderer and ButtonEditor inner classes override methods taken from their superclasses which is TableCellRenderer and DefaultCellEditor. It lets adjust how table cells appear and can be used. The overridden methods that work with the table allow a button that reads "View" in each row.

```java
private static class ActionButtonsRenderer extends JPanel implements TableCellRenderer {
    private final JButton viewButton = new JButton(text:"View");
    private final JButton deleteButton = new JButton(text:"Delete");

    public ActionButtonsRenderer() {
        setLayout(new FlowLayout(FlowLayout.CENTER, hgap:5, vgap:5));
        add(viewButton);
        add(deleteButton);
    }

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected,
                                                   boolean hasFocus, int row, int column) {
        JPanel panel = new JPanel(new FlowLayout(FlowLayout.CENTER, hgap:5, vgap:5));
        JButton viewBtn = new JButton(text:"View");
        JButton deleteBtn = new JButton(text:"Delete");
        panel.add(viewBtn);
        panel.add(deleteBtn);
        panel.setOpaque(isOpaque:true);
        if (isSelected) {
            panel.setBackground(table.getSelectionBackground());
        } else {
            panel.setBackground(table.getBackground());
        }
        return panel;
    }
}
```

Figure 49: Polymorphism po_e.java

Polymorphism is used in this code by making the ActionButtonsRenderer class implement TableCellRenderer and overridden its getTableCellRendererComponent method. This allows the renderer to be used wherever a TableCellRenderer is expected, but with custom behavior for displaying action buttons in table cells. The actual method that gets executed at runtime depends on the specific class instance, illustrating runtime polymorphism. With this design, the same data can be shown in many types of table cells in the user interface.

```java
// Renders the View/Delete button panel
class ButtonRenderer extends JPanel implements TableCellRenderer {
    final JButton viewButton = new JButton(text:"View");
    final JButton deleteButton = new JButton(text:"Delete");

    public ButtonRenderer() {
        super(new FlowLayout(FlowLayout.CENTER, hgap:5, vgap:2)); // Center buttons
        setOpaque(isOpaque:true);
        viewButton.setMargin(new Insets(top:2, left:5, bottom:2, right:5));
        deleteButton.setMargin(new Insets(top:2, left:5, bottom:2, right:5));
        add(viewButton);
        add(deleteButton);
    }

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column) {
        // Set background based on selection
        if (isSelected) {
            setForeground(table.getSelectionForeground());
            setBackground(table.getSelectionBackground());
        } else {
            setForeground(table.getForeground());
            setBackground(UIManager.getColor(key:"Button.background"));
        }
        return this;
    }
}
```

Figure 50: Polymorphism pr_e.java

Polymorphism is demonstrated in this code by implementing the TableCellRenderer interface and overriding its getTableCellRendererComponent method in the ButtonRenderer class. This allows instances of ButtonRenderer to be used wherever a TableCellRenderer is required, but with custom behavior for rendering action buttons in table cells. The actual method that is executed at runtime depends on the specific class instance, illustrating runtime polymorphism. It allows cells in a table to be changed and shaped on the user interface.

```java
private class ButtonRenderer extends DefaultTableCellRenderer {
    @Override
    public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column) {
        if (value instanceof ButtonPanel) {
            return (ButtonPanel) value;
        }
        JLabel label = (JLabel) super.getTableCellRendererComponent(table, value, isSelected, hasFocus, row, column);
        label.setBorder(BorderFactory.createEmptyBorder(top:0, left:5, bottom:0, right:5));
        return label;
    }
}
```

```java
poListTableTop.getColumn(identifier:"Actions").setCellRenderer(new ButtonRenderer());
```

Figure 51: Polymorphism po_fm.java

As shown in figure above, the ButtonRenderer class shows polymorphism through overriding "getTableCellRendererComponent()" from DefaultTableCellRenderer class. This type of overriding is runtime polymorphism where the subclass provides its own implementation while still able to call the parent class's version using "super". Depending on the input type, the method will use the parent class's implementation to create JLabel with modified formatting unless the value if "ButtonPanel". Additionally, this polymorphic behaviour allows the same method call to come up with different results depending on the object type, sticking to the

substitution principle where ButtonRenderer can be used anywhere where "DefaultTableCellRenderer" are. This allows the same method to have a more consistent interface for table rendering while managing multiple objects.

```
addButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String financeId = financeIdInfoField.getText().trim();
        String poId = (String) poIdComboBox.getSelectedItem();
        String paymentStatus = paymentStatusInfoField.getText().trim();
        String paymentDate = paymentDateInfoField.getText().trim();
        String amountText = amountInfoField.getText().trim();
```

```
updateButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String financeId = financeIdUpdateField.getText();
        String newPaymentDate = paymentDateUpdateField.getText().trim();

        // Validate payment date format
        if (!DATE_PATTERN.matcher(newPaymentDate).matches()) {
            JOptionPane.showMessageDialog(finance_e.this,
                    message:"Invalid payment date format. Please use dd-mm-yyyy format.",
                    title:"Error", JOptionPane.ERROR_MESSAGE);
            return;
        }
```

```
viewButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        populateUpdateFields(record);
        tabbedPane.setSelectedIndex(index:1);
    }
});
```

```
deleteButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        int confirm = JOptionPane.showConfirmDialog(
                finance_e.this,
                "Are you sure you want to delete Finance ID: " + record.getFinanceId() + "?",
                title:"Confirm Delete",
                JOptionPane.YES_NO_OPTION);
```

Figure 52: Polymorphism finance_e.java

As shown above, these code displays polymorphism through implementations of Swing's ActionListener. Each of the actionPerformed() method provides different usage despite the same method signature. There are 3 different buttons which are add, delete, and view, each with different functions and confirmation dialogs. This approach shows that similar interface contract allows different respond to click events.

```
try (BufferedReader br = new BufferedReader(new FileReader(FINANCE_FILE))) {
```

```
try (FileWriter writer = new FileWriter(FINANCE_FILE)) {
```

Figure 53: Polymorphism finance_c.java

As shown above, the code shows polymorphism as BufferedReader and FileWriter are used interchangeably with their parent types. This grants the file operations to work uniformly across different reader and writer implementations.

```
272     public void populateInventoryListTableTop() {
273         inventoryListTableModelTop.setRowCount(rowCount:0);
274         for (InventoryRecord record : inventoryRecords) {
275             inventoryListTableModelTop.addRow(new Object[]{record.getInventoryId() + " - (" + record.getStatus() + ")", new ButtonPanel(record)});
276         }
277     }
```

Figure 54: Polymorphism inventory_e.java

The table model and sorter are handled in the class using their parent classes and interfaces which are TableModel and RowSorter. If the actual implementation is switched (for example, to a custom sorter), the code that deals with these parts does not need to be modified. The class can be used with any appropriate object without needing to change the core code.

```
106             itemIdInfoComboBox.addActionListener(new ActionListener() {
107                 @Override       You, 2 months ago • Uncommitted changes
108                 public void actionPerformed(ActionEvent e) {
109                     String selectedItem = (String) itemIdInfoComboBox.getSelectedItem();
110                     if (selectedItem != null) {
111                         String itemId = selectedItem.split(regex:" ")[1];
112                         int stockLevel = getStockLevelFromItems(itemId);
113                         stockLevelInfoField.setText(String.valueOf(stockLevel));
114                     }
115                 }
116             });
117 
                You, 2 months ago | 1 author (You)
118             addButton.addActionListener(new ActionListener() {
119                 @Override
120                 public void actionPerformed(ActionEvent e) {
121                     String selectedItem = (String) itemIdInfoComboBox.getSelectedItem();
122                     if (selectedItem == null) {
123                         JOptionPane.showMessageDialog(inventory_c.this, message:"Please select an item.", title:"Error", JOptionPane.ERROR_MESSAGE);
124                         return;
125                     }
126 
127                     String itemId = selectedItem.split(regex:" ")[1];
128                     String lastUpdated = lastUpdatedInfoField.getText();
129                     String rQuantityText = rQuantityInfoField.getText().trim();
130 
131                     try {
132                         int rQuantity = Integer.parseInt(rQuantityText.isEmpty() ? "0" : rQuantityText);
133                         String newInventoryId = parentPanel.generateNewInventoryId();
```

Figure 55: Polymorphism inventory_c.java

Polymorphism works here by using the ActionListener interface so that just one actionPerformed method responds to events from various UI components. JComboBox and other Swing components use polymorphism because they are typically referred to by their abstract base classes (JComponent, AbstractButton) which makes it flexible to respond to events.

```java
// TreeMap to store unique dates with inventory/sales activity, sorted in descending order
BOON888, 6 days ago | 1 author (BOON888)
private TreeMap<Date, String> activityDates = new TreeMap<>(new Comparator<Date>() {
    @Override
    public int compare(Date date1, Date date2) {
        // Sort by date in descending order
        return date2.compareTo(date1);
    }
});
```

```java
BOON888, 6 days ago | 1 author (BOON888)
tableModel = new DefaultTableModel() {
    @Override
    public boolean isCellEditable(int row, int column) {
        return false; // Make cells non-editable
    }
};
reportTable = new JTable(tableModel);

// Define columns
tableModel.addColumn(columnName:"Item ID");
tableModel.addColumn(columnName:"Item Name");
tableModel.addColumn(columnName:"Stock In");
tableModel.addColumn(columnName:"Stock Out");
tableModel.addColumn(columnName:"Remaining Stock"); // Column name updated
```

Figure 56: Polymorphism inventory_r.java

It can be notice polymorphism by using Java interfaces and method overriding such as for event handling in GUIs and ActionListener classes. it can be seen by clicking the “View Report” button; it uses an ActionListener object to define “date-specific” behaviour for that button, but lets many buttons act the same with a common interface. It results in code that is easy to change and use again.

```java
BOON888, 2 months ago | 1 author (BOON888)
monthYearButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        FinanceReport financeReportPanel = new FinanceReport(entry.getKey());
        JDialog dialog = new JDialog();
        dialog.add(financeReportPanel);
        dialog.setSize(width:1600, height:1000);
        dialog.setLocationRelativeTo(finance_r.this);
        dialog.setVisible(b:true);
    }
});
}
```

Figure 57: Polymorphism finance_r.java

It is polymorphism that is employed by using event listeners and the Swing component model. An anonymous inner class implements ActionListener in the monthYearButton to add

behaviour when the button is clicked. As a result, several buttons can participate in the same event processing (through ActionListener) and yet handle events differently based on individual buttons and situations, making the GUI flexible and easy to update or adapt as necessary.

```
contactField.setDocument((PlainDocument) insertString(offset, str, attr) -> {
        if (str == null) {
            return;
        }

        String digitsOnly = str.replaceAll( regex: "[^0-9]", replacement: "");
        if (digitsOnly.isEmpty()) {
            Toolkit.getDefaultToolkit().beep();
            return;
        }

        if ((getLength() + digitsOnly.length()) > 11) {
            Toolkit.getDefaultToolkit().beep();
            return;
        }

        super.insertString(offset, digitsOnly, attr);
});
```

Figure 58: Polymorphism supplier_e.java

PlainDocument is the parent and its insertString is rewritten to check only for digits. Calling insertString on contactField (a JTextField) will result in the PlainDocument override provided being performed, not the default method in PlainDocument. This is a typical case of runtime polymorphism which picks the actual method to run based on the object's actual type during execution.

```
tableModel = new DefaultTableModel(columnNames, rowCount: 0) {  ccc0315
    @Override  ccc0315
    public boolean isCellEditable(int row, int column) {
        return column == 7; // Only actions column is editable
    }
};
```

Figure 59: Polymorphism item_e.java

I am overriding the isCellEditable method in the DefaultTableModel superclass to make it possible for only the "Actions" column to be changed. Calling isCellEditable on tableModel in Java makes the JVM use the overridden version created in this class instead of the one in DefaultTableModel.

```
table.setDefaultRenderer(Object.class, new DefaultTableCellRenderer() {  ccc0315
    @Override  ccc0315
    public Component getTableCellRendererComponent(JTable table, Object value,
            boolean isSelected, boolean hasFocus, int row, int column) {
        Component c = super.getTableCellRendererComponent(table, value, isSelected, hasFocus, row, column);

        // Check if total stock is below 10
        int totalStock = (Integer) table.getValueAt(row,  column: 2);
        if (totalStock < 10) {
            c.setForeground(Color.RED);
            c.setFont(c.getFont().deriveFont(Font.BOLD));
        } else {
            c.setForeground(Color.BLACK);
            c.setFont(c.getFont().deriveFont(Font.PLAIN));
        }
```

Figure 60: Polymorphism sales_e.java

DefaultTableCellRenderer is used to generate simple versions of table cells. Add extra behaviour such as changing the text colour and font based on stock quantities, by using getTableCellRendererComponent and still allowing the table to use its default rendering abilities.

3.4 Abstraction

```
34          for (int i = 0; i < buttonNames.length; i++) {
35              JButton button = new JButton(buttonNames[i]);
36              final String className = classNames[i];
37              button.addActionListener(e -> switchContent(className));
38              bottomPanel.add(button);
39          }
```

Figure 61: Abstraction am.java

Instead of having to handle reflection calling Class.forName and newInstance, switchContent() deals with this and loads and displays a new panel. When look at the button, all it has to do is trigger switchContent() to function. In programming, abstraction means hiding the details of what happens inside and only showing what is necessary on the outside.

```
187     private void loadUserData() {
188         tableModel.setRowCount(rowCount:0);
189         for (user_c user : UserController.loadUsers()) {
190             tableModel.addRow(new Object[]{user.getId(), "Buttons"});
191         }
192     }
193
194     // Search for a user by ID
195     private void searchUser() {
196         String searchId = searchField.getText().trim();
197         if (searchId.isEmpty()) {
198             loadUserData();
199             return;
200         }
201         tableModel.setRowCount(rowCount:0);
202         for (user_c user : UserController.loadUsers()) {
203             if (user.getId().equals(searchId)) {
204                 tableModel.addRow(new Object[]{user.getId(), "Buttons"});
205                 break;
206             }
207         }
208     }
```

Figure 62: Abstraction user_am.java

UserController contains the methods that deal with user actions like adding, deleting or loading users in user_am class. The user interface doesn't care how users are handled—that is done by making calls to UserController functions like addUser() and loadUsers(). As a result, the interface and backend may be worked on independently.

```java
class UserController {

    private static final String USER_FILE = "TXT/users.txt";

    // Load users from the file
    public static List<user_c> loadUsers() {
        List<user_c> users = new ArrayList<>();
        try (BufferedReader br = new BufferedReader(new FileReader(USER_FILE))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] parts = line.split(regex:"\\|");
                if (parts.length == 6) {
                    user_c user = new user_c(parts[0], parts[1], parts[2], parts[3], parts[4], parts[5]);
                    users.add(user);
                }
            }
        } catch (IOException e) {
            System.out.println("Error reading user file: " + e.getMessage());
        }
        return users;
    }
```

Figure 63: Abstraction user_c.java

This class he UserController class handles all the ways users are managed—adding them, editing them, removing them, validating their info and reading users from the file. In this code stays easy to work with because the controller alone handles user storage and validation and the UI does not have to worry about it. Instead, they simply make calls to methods like addUser(user) or loadUsers().

```java
    private void viewPR(String prId) {
        detailsTableModel.setRowCount(rowCount:0);
        try (BufferedReader br = new BufferedReader(new FileReader(PR_FILE))) {…
        } catch (IOException e) {…
    }

    private void updatePRData() {
        if (detailsTableModel.getRowCount() == 0) {…
```

Figure 64: Abstraction pr_pm.java

Abstraction removes the details of how something works and just gives the user useful information. When in this class uses don't need to think about how the updates and reading/writing to files or tables are handled. All those difficult aspects are concealed by clear method names such as updatePRData(), searchPR() and viewPR(). This shows how abstraction is useful in practical settings.

```java
private po_e_c poController = new po_e_c();

private void loadPurchaseOrders() {
    tableModel.setRowCount(0);
    fullPoData.clear();
    List<String[]> dataList = poController.loadPurchaseOrders();
    for (String[] data : dataList) {
        if (data.length >= 1) {
            tableModel.addRow(new Object[]{data[0], null});
            fullPoData.add(data);
        }
    }
    clearDetailsPanel();
}
```

```java
public class po_e_c {
    // ... other code ...
    public List<String[]> loadPurchaseOrders() {
        // ... file reading logic ...
        return fullPoData;
    }
    // ... other methods ...
}
```

Figure 65: Abstraction po_e_c.java

Abstraction is applied by dividing the system into distinct classes with clear responsibilities. The po_e class interacts with the user and delegates data operations to the po_e_c class, which abstracts away the details of file handling and purchase order management. By separating these elements allows the user interface to be simple and focused. The data processing can be maintained away from it. So, the UI is not disturbed by file management or data checking which makes the full codebase more manageable.

```java
public class pr_e extends JPanel {
    // ...
    protected pr_e_c controller;

    public pr_e() {
        // ...
        controller = new pr_e_c(this);
        // ...
        controller.loadPRData(); // Load initial data
    }
    // ...
}
```

```java
public class pr_e_c {
    private static final String PR_FILE = "TXT/pr.txt";
    public final pr_e view;

    public pr_e_c(pr_e view) {
        this.view = view;
    }

    public void loadPRData() {
        // ... logic to load data from file and update the view ...
    }

    public void addPR() {
        // ... logic to add a new PR ...
    }
    // ... other logic methods ...
}
```

Figure 66: Abstraction pr_e_c.java

Abstraction is demonstrated by separating the user interface (pr_e) from the business logic and data handling (pr_e_c). The pr_e class manages the user interface, and the pr_e_c class takes care of actions such as loading, adding, updating and deleting purchase requisitions. The UI interacts with the controller through simple method calls (like controller.loadPRData()), without needing to know the details of file handling or data processing. The design shields the UI from complex details and simplifies the process of updating and supporting the code.

```
public class po_fm extends JPanel {

    private static final String PO_FILE = "TXT/po.txt";
    private static final String ITEMS_FILE = "TXT/items.txt";

    private JTabbedPane tabbedPane;
```

```
public po_fm() {
    loadItemDetailsForDropdown();
    setLayout(new BorderLayout());

    tabbedPane = new JTabbedPane();
    Font tabTitleFont = new Font(name:"Arial", Font.BOLD, size:20);
    tabbedPane.setFont(tabTitleFont);

    poListPanel = createPOListPanel();
    tabbedPane.addTab(title:"PO List", poListPanel);

    add(tabbedPane, BorderLayout.CENTER);
```

Figure 67: Abstraction po_fm.java

As shown in figure above, “private JTabbedPane” shows abstraction by only exposing an essential and simple interface “addTab()” without showing the complex implementation of tab management such as rendering and event handling. The users can add a tab with a title and panel without prior understanding to its mechanisms. Additionally, the key principle of abstraction is the separation of functionality from implementation.

```java
private void loadInventoryData() {
    inventoryRecords.clear();
    try (BufferedReader br = new BufferedReader(new FileReader(INVENTORY_FILE))) {
        String line;
        while ((line = br.readLine()) != null) {
            String[] data = line.split(regex:"\\|");
            if (data.length == 7) {
                try {
                    String inventoryId = data[0].trim();
                    String itemId = data[1].trim();
                    int stockLevel = Integer.parseInt(data[2].trim());
                    String lastUpdated = data[3].trim();
                    int receivedQuantity = Integer.parseInt(data[4].trim());
                    int updatedBy = Integer.parseInt(data[5].trim());
                    String status = data[6].trim();

                    InventoryRecord record = new InventoryRecord(
                        inventoryId, itemId, stockLevel,
                        lastUpdated, receivedQuantity,
                        updatedBy, status
                    );
                    inventoryRecords.add(record);
                } catch (NumberFormatException e) {
                    System.err.println("Error parsing data in line (inventory.txt): " + line);
                }
            } else {
                System.err.println("Skipping invalid line in inventory.txt: " + line + ". Expected 7 columns.");
            }
        }
    } catch (IOException e) {
        JOptionPane.showMessageDialog(this, "Error reading Inventory file: " + e.getMessage(), title:"Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

```java
private void saveInventoryData() {
    try (FileWriter writer = new FileWriter(INVENTORY_FILE)) {
        for (InventoryRecord record : inventoryRecords) {
            writer.write(record.getInventoryId() + "|"
                    + record.getItemId() + "|"
                    + record.getStockLevel() + "|"
                    + record.getLastUpdated() + "|"
                    + record.getReceivedQuantity() + "|"
                    + record.getUpdatedBy() + "|"
                    + record.getStatus() + "\n");
        }
    } catch (IOException e) {
        JOptionPane.showMessageDialog(this, "Error saving Inventory file: " + e.getMessage(), title:"Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 68: Abstraction inventory_fm.java

As shown in figure above, loadInventoryData() and saveInventoryData() method practices abstraction by hiding complex operations and only showing simple interfaces. These methods encapsulate both the file reading and file writing logic, only showing clean method calls.

```java
finance_c.FinanceRecord newRecord = new finance_c.FinanceRecord(
        financeId,
        poId,
        finance_c.STATUS_PENDING,
        paymentStatus,
        paymentDate,
        amountText,
        Integer.parseInt(login_c.currentUserId)
);
```

```java
public static class FinanceRecord {
    private String financeId;
    private String poId;
    private String approvalStatus;
    private String paymentStatus;
    private String paymentDate;
    private String amount;
    private int verifiedBy;

    public FinanceRecord(String financeId, String poId, String approvalStatus, String paymentStatus,
                         String paymentDate, String amount, int verifiedBy) {
        this.financeId = financeId;
        this.poId = poId;
        this.approvalStatus = approvalStatus;
        this.paymentStatus = paymentStatus;
        this.paymentDate = paymentDate;
        this.amount = amount;
        this.verifiedBy = verifiedBy;
    }

    public String getFinanceId() { return financeId; }
    public String getPoId() { return poId; }
    public String getApprovalStatus() { return approvalStatus; }
    public String getPaymentStatus() { return paymentStatus; }
    public String getPaymentDate() { return paymentDate; }
    public String getAmount() { return amount; }
    public int getVerifiedBy() { return verifiedBy; }

    public void setPoId(String poId) { this.poId = poId; }
    public void setApprovalStatus(String approvalStatus) { this.approvalStatus = approvalStatus; }
    public void setPaymentStatus(String paymentStatus) { this.paymentStatus = paymentStatus; }
    public void setPaymentDate(String paymentDate) { this.paymentDate = paymentDate; }
    public void setAmount(String amount) { this.amount = amount; }
    public void setVerifiedBy(int verifiedBy) { this.verifiedBy = verifiedBy; }
}
```

Figure 69: Abstraction finance_e.java & finance_c.java

As shown in figure above, the code shows abstraction in the FinanceRecord class, it hides complex implementation details behind a simple interface of getters and setters.

```java
private void loadInventoryData() {
    inventoryRecords.clear();
    try (BufferedReader br = new BufferedReader(new FileReader(INVENTORY_FILE))) {
        String line;
        while ((line = br.readLine()) != null) {
            String[] data = line.split(regex:"\\|");
            if (data.length == 7) {
                try {
                    String inventoryId = data[0].trim();
                    String itemId = data[1].trim();
                    int currentStock = Integer.parseInt(data[2].trim());
                    String lastUpdated = data[3].trim();
                    int reorderLevel = Integer.parseInt(data[4].trim());
                    int updatedBy = Integer.parseInt(data[5].trim());
                    String status = data[6].trim();

                    InventoryRecord record = new InventoryRecord(inventoryId, itemId, currentStock, lastUpdated, reorderLevel, updatedBy, status);
                    inventoryRecords.add(record);
                } catch (NumberFormatException e) {
                    System.err.println("Error parsing data in line (inventory.txt): " + line);
                }
            } else {
                System.err.println("Skipping invalid line in inventory.txt: " + line + ". Expected 7 columns.");
            }
        }
    } catch (IOException e) {
        JOptionPane.showMessageDialog(this, "Error reading inventory file: " + e.getMessage(), title:"Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 70: Abstraction inventory_e.java

For external users (such as other classes or GUI components), inventory_e handles all the details of loading and displaying inventory data. Users add the panel to a frame and the inventory appears, making file formats and data processing details irrelevant. LoadInventoryData() keeps the details private, so the interface needed to show the inventory is straightforward and clean.

```
156         private int getStockLevelFromItems(String itemId) {
157             try (BufferedReader br = new BufferedReader(new FileReader(ITEMS_FILE))) {
158                 String line;
159                 while ((line = br.readLine()) != null) {
160                     String[] data = line.split(regex:"\\|");
161                     if (data.length == 6 && data[0].trim().equals(itemId)) {
162                         return Integer.parseInt(data[5].trim());
163                     }
164                 }
165             } catch (IOException | NumberFormatException e) {
166                 System.err.println("Error reading items file or parsing stock level: " + e.getMessage());
167                 e.printStackTrace();
168             }
169             return 0;
170         }
171     }
```

Figure 71: Abstraction inventory_c.java

The class takes care of the challenge of loading items and updating stock by using simple UI tools. The rest of the program can be setup to just add this panel to the interface and trust it to manage all the validation and updates automatically.

```
76          // Search Bar (Top Right, Full Width)
77          JPanel searchContainer = new JPanel(new FlowLayout(FlowLayout.RIGHT));
78          JLabel searchLabel = new JLabel(text:"Search:");
79          searchLabel.setFont(GLOBAL_FONT);
80          searchTextField = new JTextField(columns:30); // Increased width for better visibility
81          searchTextField.setFont(GLOBAL_FONT);
82          searchContainer.add(searchLabel);
83          searchContainer.add(searchTextField);
84          topPanel.add(searchContainer, BorderLayout.EAST);
```

```
155         // Add listener for the search bar
            BOON888, 2 weeks ago | 1 author (BOON888)
156         searchTextField.addActionListener(new ActionListener() {
157             @Override
158             public void actionPerformed(ActionEvent e) {
159                 String text = searchTextField.getText();
160                 if (text.trim().length() == 0) {
161                     sorter.setRowFilter(filter:null); // Show all rows if the search field is empty
162                 } else {
163                     sorter.setRowFilter(RowFilter.regexFilter("(?i)" + text)); // Filter rows case-insensitively
164                 }
165             }
166         });
167     }
```

Figure 72: Abstraction inventory_v.java

Search results are possible because of the complex filtering process which is hidden behind a simple search box. Persons using the class just have to type one keyword; the class filters the

table model only inside the class and doesn't make users notice or deal with the filtering algorithms or UI event wiring. Makes it easier for users to interact with the feature and keeps it up-to-date.

```
54          // Iterate through sorted dates and create report buttons
55          for (Map.Entry<Date, String> entry : activityDates.entrySet()) {
56              JPanel rowPanel = new JPanel(new GridLayout(rows:1, cols:2));
57              rowPanel.setBorder(new LineBorder(Color.LIGHT_GRAY, thickness:1));
58              rowPanel.setMaximumSize(new Dimension(Integer.MAX_VALUE, height:50)); // Limit height
59
60              JLabel dateLabel = new JLabel(dateFormatForDisplay.format(entry.getKey()));
61              dateLabel.setFont(CONTENT_FONT);
62              JButton viewButton = new JButton(text:"View Report");      BOON888, 6 days ago • inventory report done
63              viewButton.setFont(CONTENT_FONT);
64
65              rowPanel.add(dateLabel);
66              rowPanel.add(viewButton);
67
68              mainPanel.add(rowPanel);
69              mainPanel.add(Box.createRigidArea(new Dimension(width:0, height:5))); // Add a small gap between rows
70
71              String selectedDate = dateFormatForDisplay.format(entry.getKey());
                BOON888, 6 days ago | 1 author (BOON888)
72              viewButton.addActionListener(new ActionListener() {
73                  @Override
74                  public void actionPerformed(ActionEvent e) {
75                      InventoryReport inventoryReportPanel = new InventoryReport(selectedDate);
76                      JDialog dialog = new JDialog();
77                      dialog.setTitle("Inventory Report - " + selectedDate);
78                      dialog.add(inventoryReportPanel);
79                      dialog.setSize(width:1000, height:600); // Increased size for better table display
80                      dialog.setLocationRelativeTo(inventory_r.this);
81                      dialog.setModal(modal:true); // Make it modal so user has to close it
82                      dialog.setVisible(b:true);
83                  }
84              });
85          }
```

Figure 73: Abstraction inventory_r.java

Using abstraction, the report system highlights the key features such as its daily sales and stock report and handles advanced tasks such as breaking up and summarizing data without users being aware. The panel users can only work with the buttons and tables, not seeing or knowing how the files are read and the data is formatted on the back end. Because of this, the screen stays free of clutter as most of the work is carried out by the data-handling section.

```
emailField = new JTextField( columns: 15);
emailField.setInputVerifier((input) -> {
        String text = ((JTextField) input).getText().trim();
        if (!text.isEmpty() && !isValidEmail(text)) {
            JOptionPane.showMessageDialog(input,
                    message: "Invalid email format!\nExample: name@example.com",
                    title: "Invalid Email",
                    JOptionPane.ERROR_MESSAGE);
            return false;
        }
        return true;
});
```

Figure 74: Abstraction supplier_e.java

An InputVerifier abstract class (or interface, conceptually) states the rules for verifying user input before a component loses focus. Implementing the abstract contract means user are using their own email validation rules by making a subclass and overriding verify. As soon as the cursor moves away from emailField, Swing just calls verify() on the InputVerifier associated with the field. Only the result matters to Main," whether it's true or false, not the method it called or how JOptionPane was used.

```
    public String[] getEditedData() {  1 usage   ccc0315
        return new String[]{
            nameField.getText(),
            contactField.getText(),
            emailField.getText(),
            addressField.getText(),
            itemField.getText()
        };
    }
}
```

Figure 75: Abstraction supplier_v.java

It returns all the edited supplier data as a String array in an abstract way. Other elements of application (like supplier_e) do not need to be aware of how dialog.getEditedData() collects data (using which JTextFields and in what order). It accepts an array of strings to process, because it does not need to deal with the internal definitions in the supplier_v sourcedialog. It makes it easier for the caller to reach the agent.

```
private static class Supplier {  5 usages  ccc0315
    private String id, name, phone, email, address, category;  2 usages

    public Supplier(String id, String name, String phone, String email, String address, String category) {  1 usage  ccc0315
        this.id = id;
        this.name = name;
        this.phone = phone;
        this.email = email;
        this.address = address;
        this.category = category;
    }

    public String getId() { return id; }  1 usage  ccc0315
    public String getName() { return name; }  1 usage  ccc0315
    public String getPhone() { return phone; }  1 usage  ccc0315
    public String getEmail() { return email; }  1 usage  ccc0315
    public String getAddress() { return address; }  1 usage  ccc0315
    public String getCategory() { return category; }  1 usage  ccc0315
}
```

Figure 76: Abstraction supplier_list.java

The class Supplier is created to represent a Supplier. Through the Supplier object, don't have to work with the raw String[] arrays or single String variables for the suppliers' attributes. This object makes it so we do not have to see how each piece of data (id, name, phone, etc.) is handled under the hood. Calling methods on a Supplier object (like supplier.getName()) lets ask "what is the name of this supplier?" instead of trying to figure out "which integer in the array keeps the string of the name? It gives a clearer and more structured way to manage supplier data.

```java
@Override  ccc0315 +1
public Object getCellEditorValue() {
    if (isPushed) {
        int option = JOptionPane.showOptionDialog(
                parentComponent: item_e.this,
                message: "Choose action for " + items.get(itemTable.getEditingRow()).getName(),
                title: "Item Action",
                JOptionPane.YES_NO_CANCEL_OPTION,
                JOptionPane.QUESTION_MESSAGE,
                icon: null,
                new String[]{"View", "Edit", "Delete", "Cancel"},
                initialValue: "Cancel");

        if (option == 0) {
            viewItem(itemTable.getEditingRow());
        }else if (option == 1) {
            editItem(itemTable.getEditingRow());
        }else if (option == 2) {
            deleteItem(itemTable.getEditingRow());
        }
    }
    isPushed = false;
    return label;
}
```

Figure 77: Abstraction item_e.java

Like TableCellRenderer, TableCellEditor sets out the abstract rules for how a cell can be changed by the user. TableButtonEditor is based on DefaultCellEditor (a TableCellEditor implementation) and it provides its own versions of getTableCellEditorComponent and getCellEditorValue. As a result, now have custom behavior set up for editing the "Actions" cell (producing a button and a dialog when it is clicked). Whenever someone tries to edit a cell, the JTable uses the TableCellEditor abstract class without needing to know the details of editing.

```
private JPanel createSalesListPanel() {  1 usage   ccc0315
    JPanel panel = new JPanel(new BorderLayout());
    panel.setBorder(BorderFactory.createTitledBorder("Sales List"));

    String[] columns = {"Sales ID", "Item ID", "Item Name", "Date", "Qty Sold", "Remaining", "Sales Person", "Actions"};
    tableModel = new DefaultTableModel(columns,  rowCount: 0) {  ccc0315
        @Override  ccc0315
        public boolean isCellEditable(int row, int column) {
            return column == 7;
        }
    };

    salesTable = new JTable(tableModel);
    salesTable.setRowHeight(30);
    salesTable.getColumn( identifier: "Actions").setCellRenderer(new ButtonRenderer());
    salesTable.getColumn( identifier: "Actions").setCellEditor(new ButtonEditor(new JCheckBox()));

    panel.add(new JScrollPane(salesTable), BorderLayout.CENTER);
    return panel;
}
```

Figure 78: Abstraction sales_e.java

DefaultTableModel provides an abstract way to work with table data shown in a JTable. It establishes the way data is handled and used. There are telling TableView to not allow editing for other columns, by using isCellEditable in salesTable's model. The JTable uses isCellEditable to see if a cell can be edited and then allows this logic to do the specific checks on its own. It prevents from easily seeing the rules behind setting cell editable status.

```
    // Add listener for the search bar
    searchTextField.addActionListener(new ActionListener() {  BOON888
        @Override  BOON888
        public void actionPerformed(ActionEvent e) {
            String text = searchTextField.getText();
            if (text.trim().length() == 0) {
                sorter.setRowFilter(null); // Show all rows if the search field is empty
            } else {
                sorter.setRowFilter(RowFilter.regexFilter( regex: "(?i)" + text)); // Filter rows case-insensitively
            }
        }
    });
}
```

Figure 79: Abstraction sales_v.java

The interface ActionListener has one abstract method, actionPerformed which explains what occurs each time an action takes place (for example, pressing Enter in a text field). Using anonymous inner classes to implement ActionListener give the real code needed for this kind of behavior. The searchTextField (or Swing's event handling system) invokes the actionPerformed method on the ActionListener when the user presses Enter. It doesn't care about the internal code that generates the filter (the RowFilter); it's only concerned about invoking the actionPerformed method from the listener.

4.0 Design Solution

4.1 Use Case Diagram

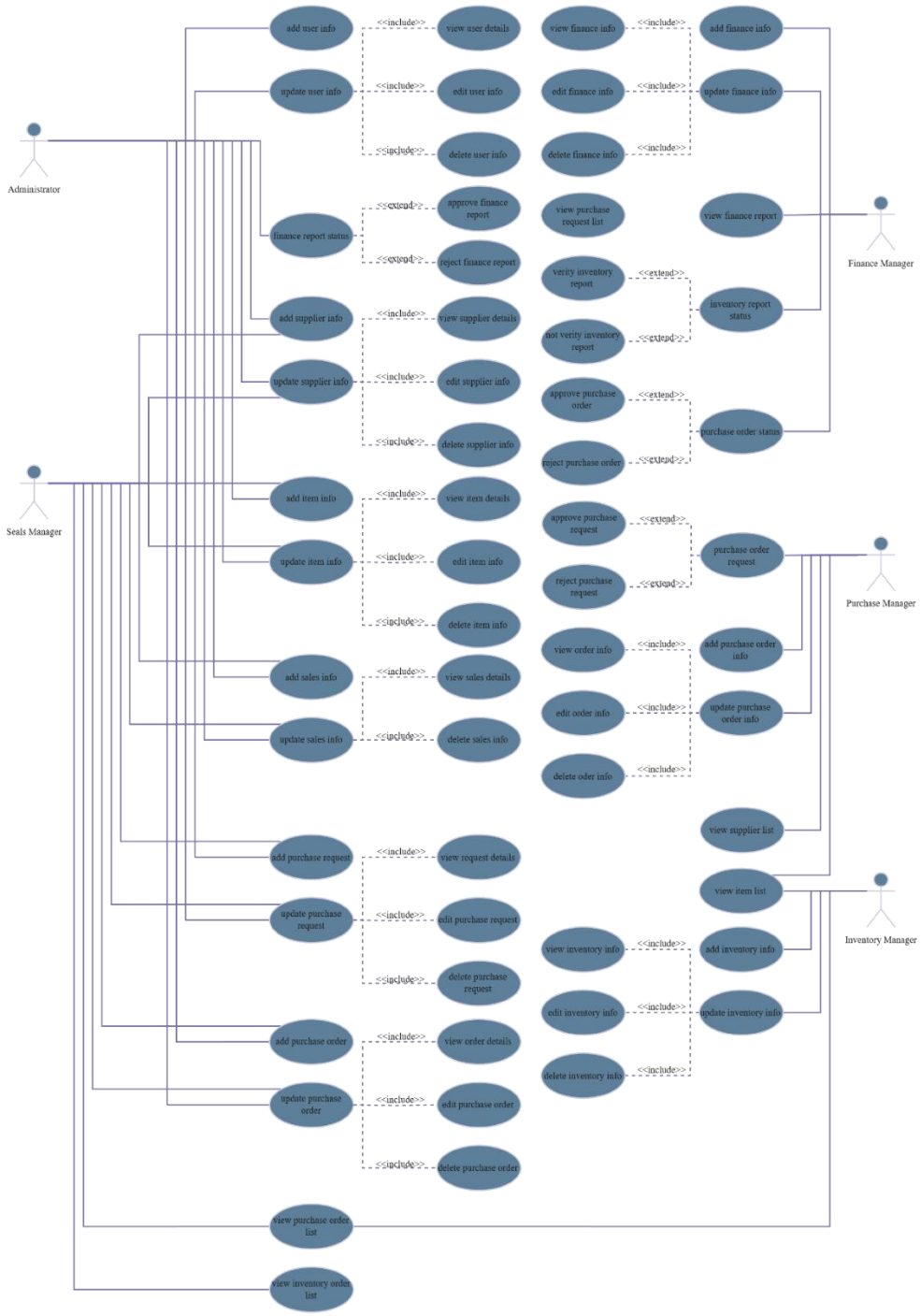


Figure 80: Use Case Diagram

4.2 Class Diagram

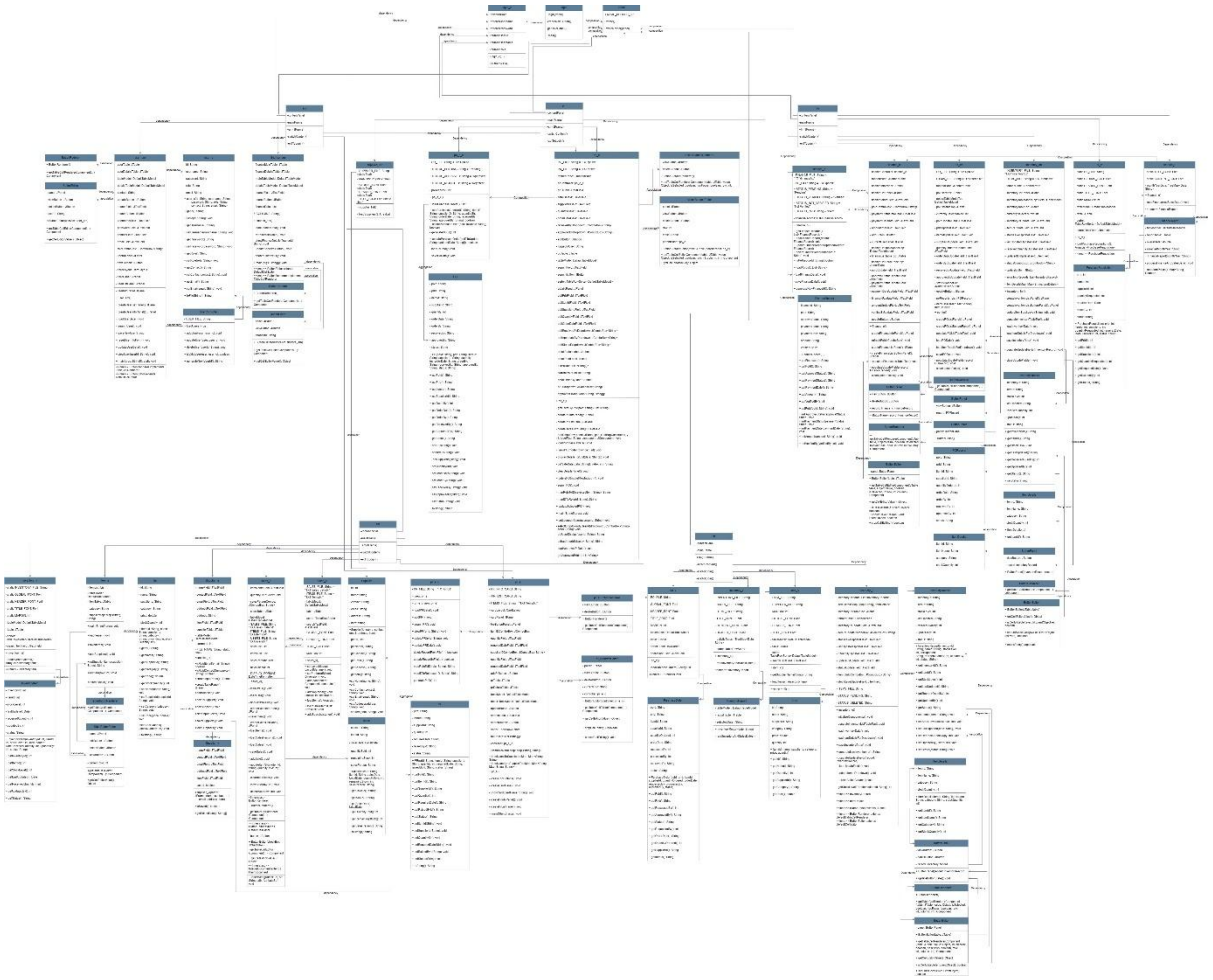


Figure 81: Class Diagram

https://drive.google.com/file/d/1Swcpstxe_lSUWVxZXVk9ILVY2h9xn3xW/view?usp=sharing

5.0 Screenshots Output

Login:

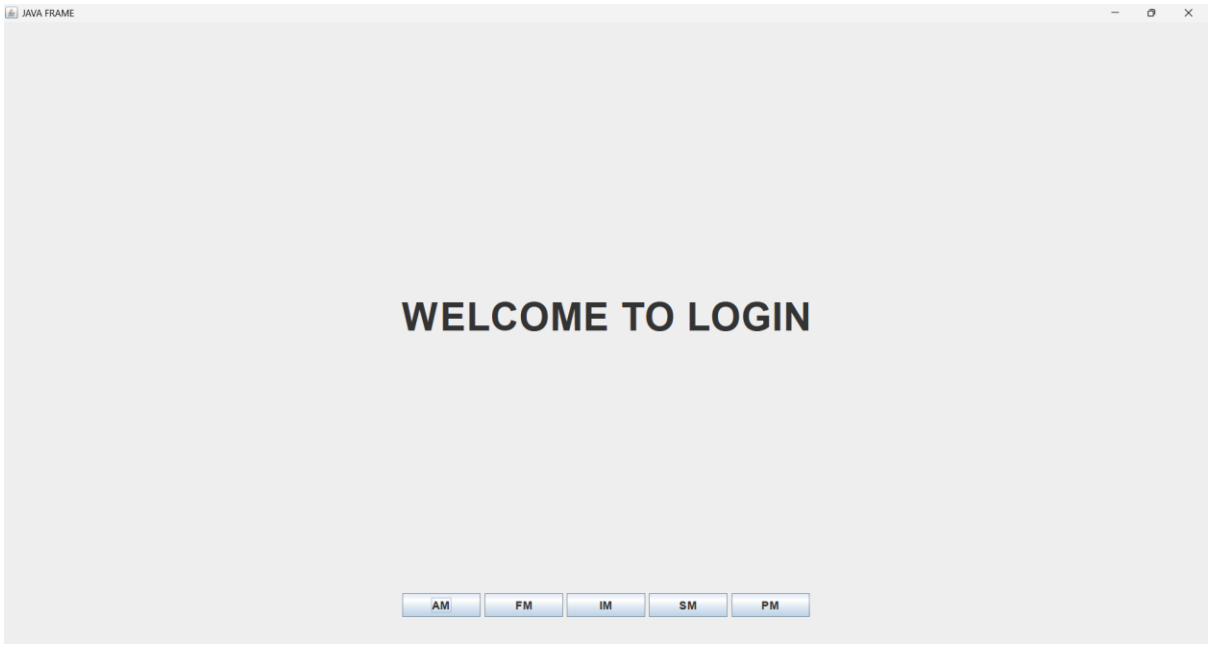
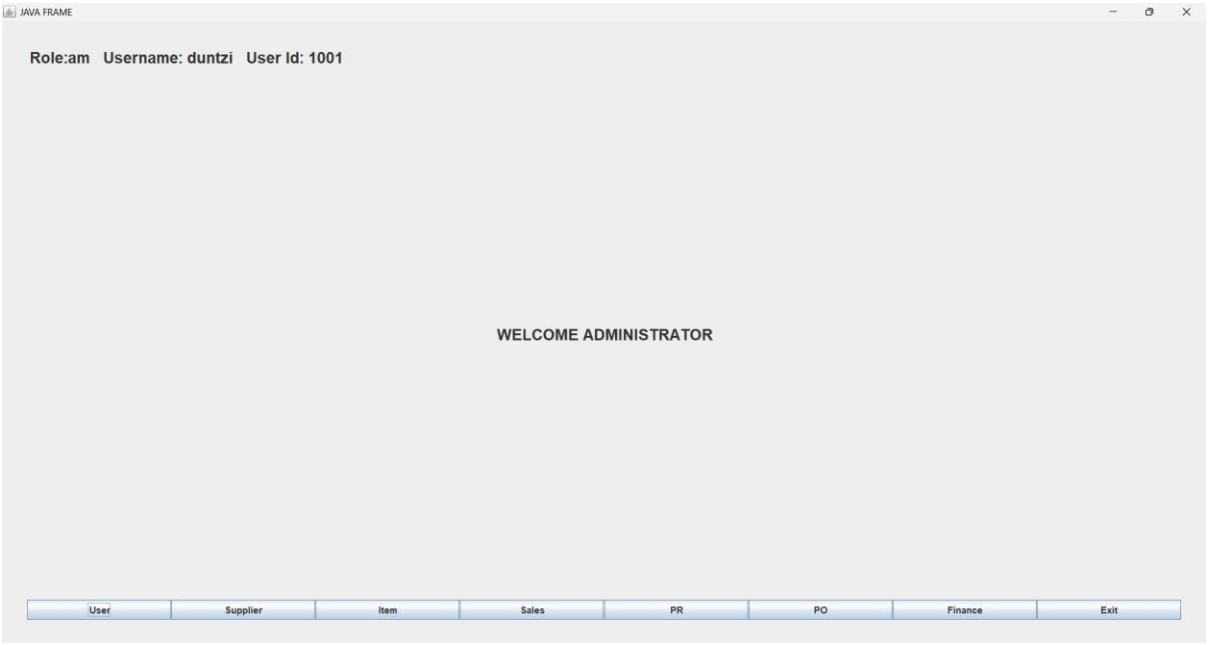
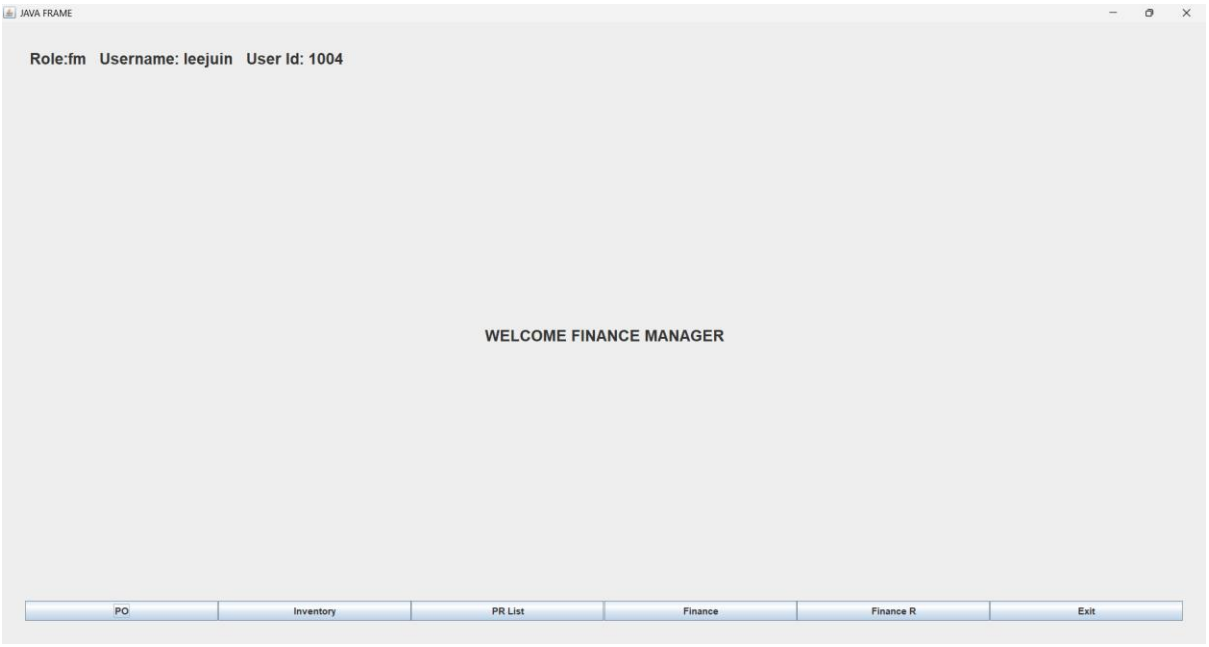


Figure 82: Login

During login, users are presented with several role choices to pick from. A role only allows users with its own username and password to enter. If a person chooses the Financial Manager role and submits the Administrator's information, the system will show a message saying the credentials are incorrect. In addition, if someone enters the wrong password and username, "Invalid Credentials" will be displayed.

Dashboard of AM PM SM IM FM:






JAVA FRAME
Role:im Username: boon User Id: 1003
WELCOME INVENTORY MANAGER
Item List
PO List
Inventory
Inventory Report
Exit


JAVA FRAME
Role:sm Username: junheng User Id: 1002
WELCOME SALES MANAGER
Supplier
Item
Sales
PR
PO List
Inventory List
Exit

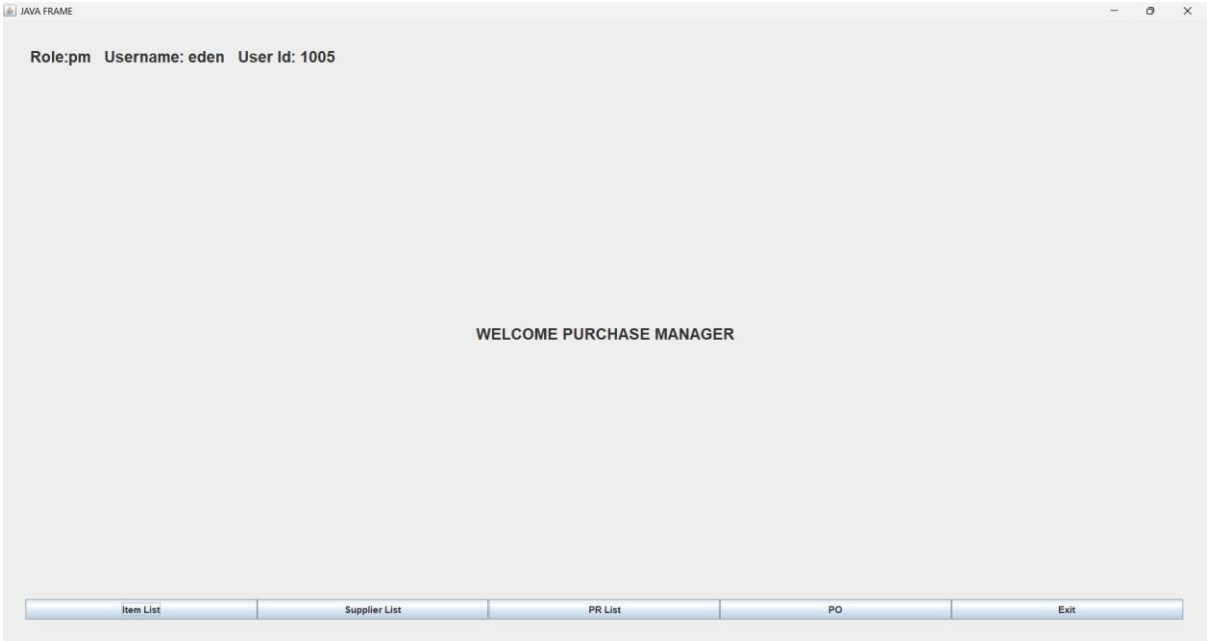
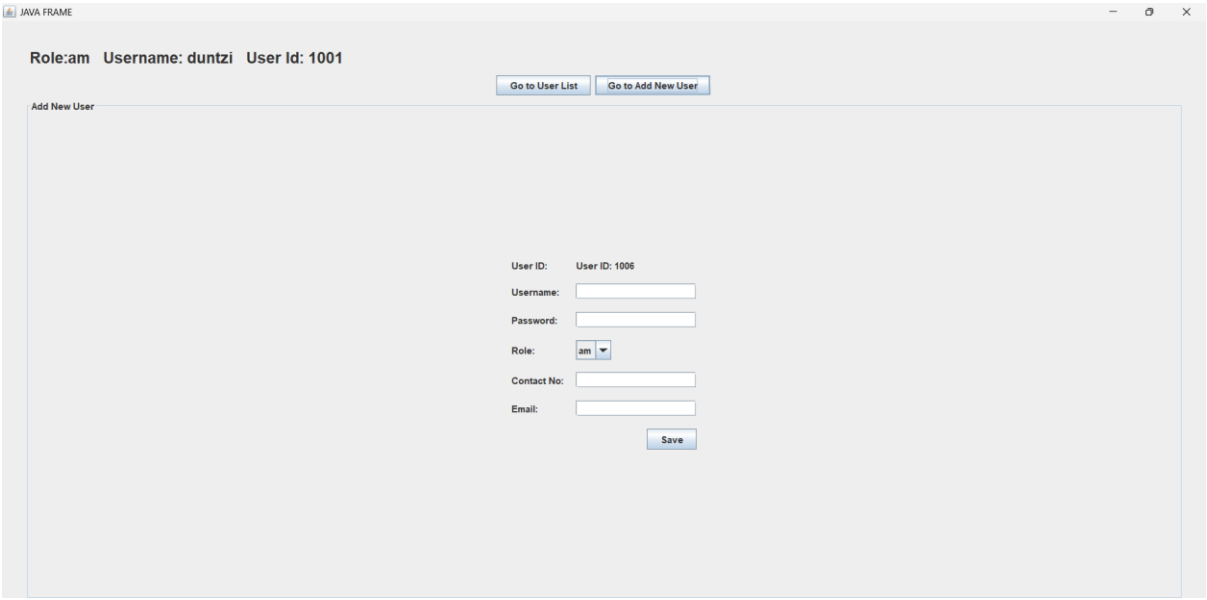
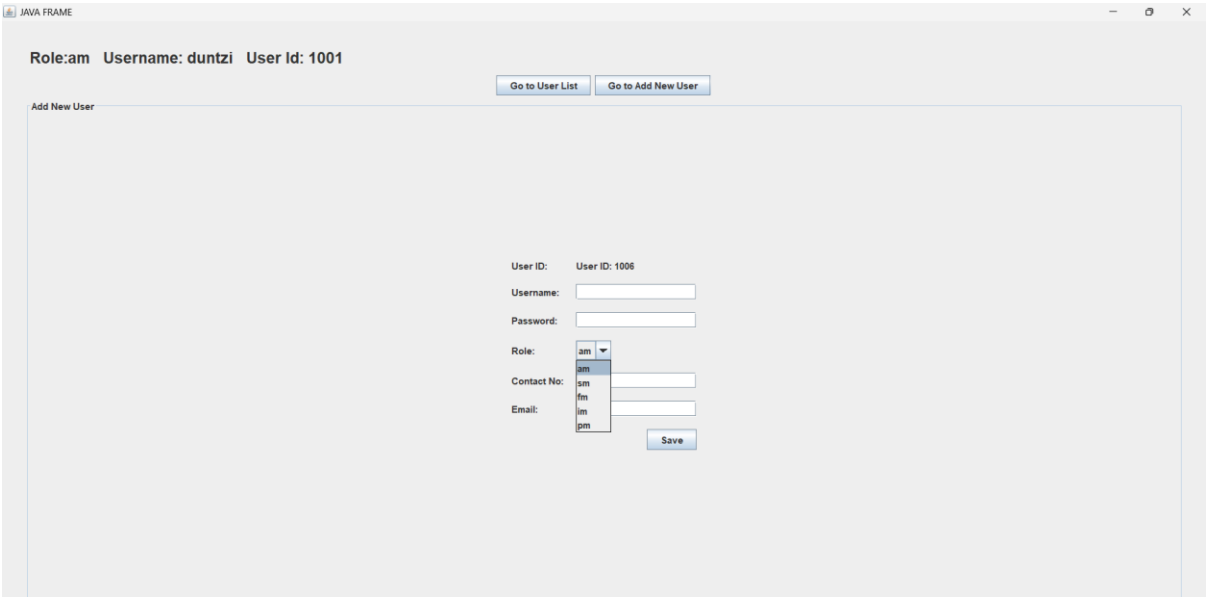


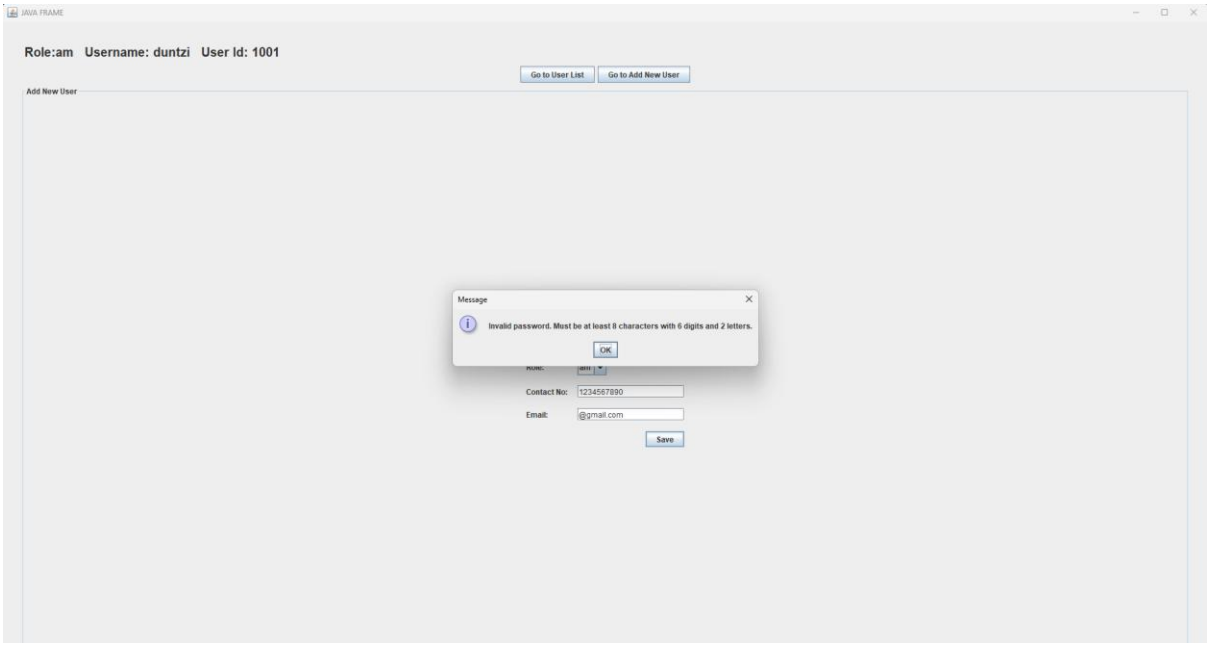
Figure 83: Dashboard of Each Role

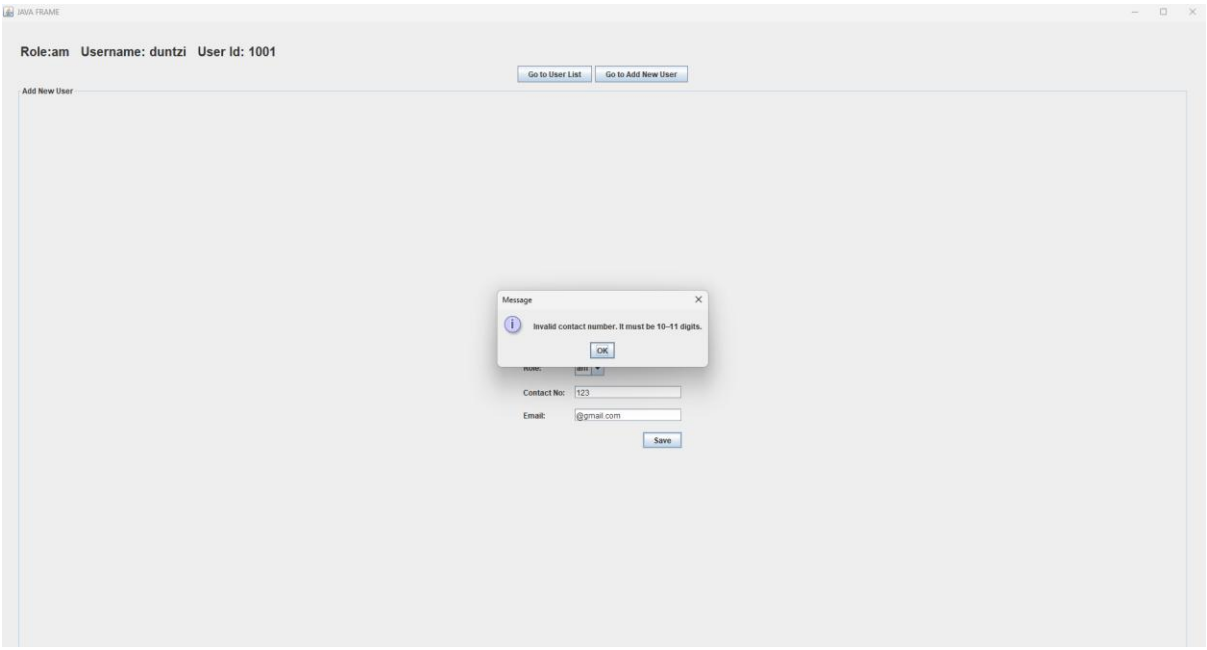
There is a dashboard prepared for each person with Administrator, Purchase Manager, Finance Manager, Inventory Manager and Sales Manager roles. For every user, the welcome title on the dashboard matches their role and the role, username and user ID are shown at the top left corner. Furthermore, every role's dashboard leads to pages that are unique to what that user can see. Furthermore, the detailed explanation of each page will provide below.

User Creation:






JAVA FRAME
Role:am Username: duntzi User Id: 1001
Go to User List
Go to Add New User
Add New User
Message
Invalid password. Must be at least 8 characters with 6 digits and 2 letters.
OK
Contact No:
1234567890
Email:
@gmail.com
Save


JAVA FRAME
Role:am Username: duntzi User Id: 1001
Go to User List
Go to Add New User
Add New User
Message
Invalid contact number. It must be 10-11 digits.
OK
Contact No:
123
Email:
@gmail.com
Save

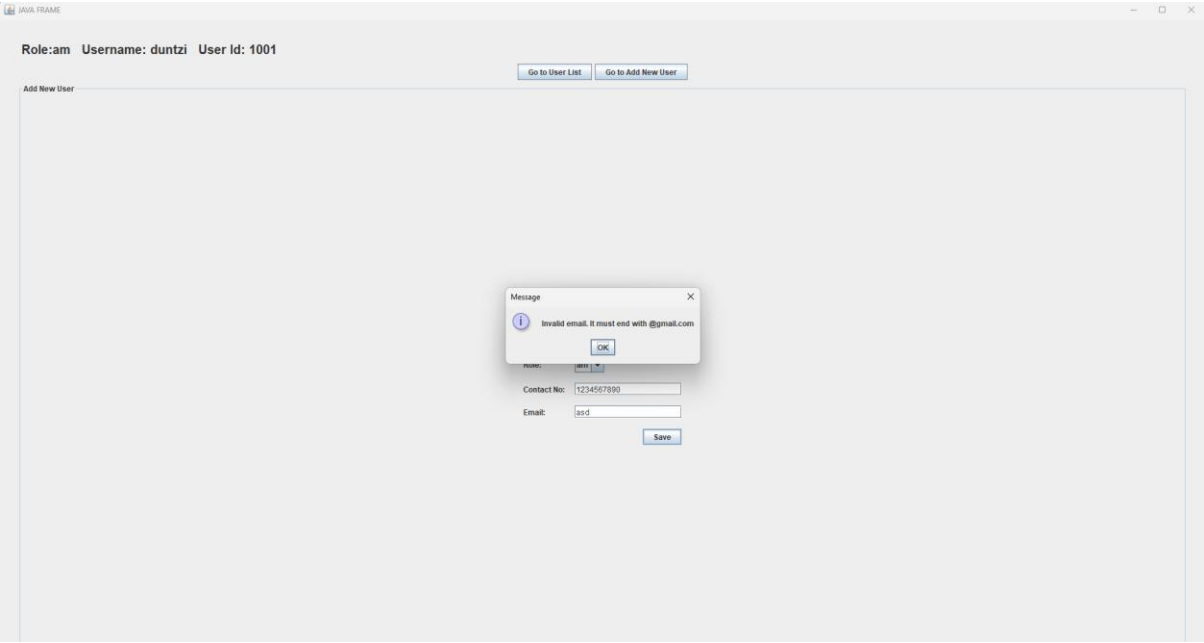
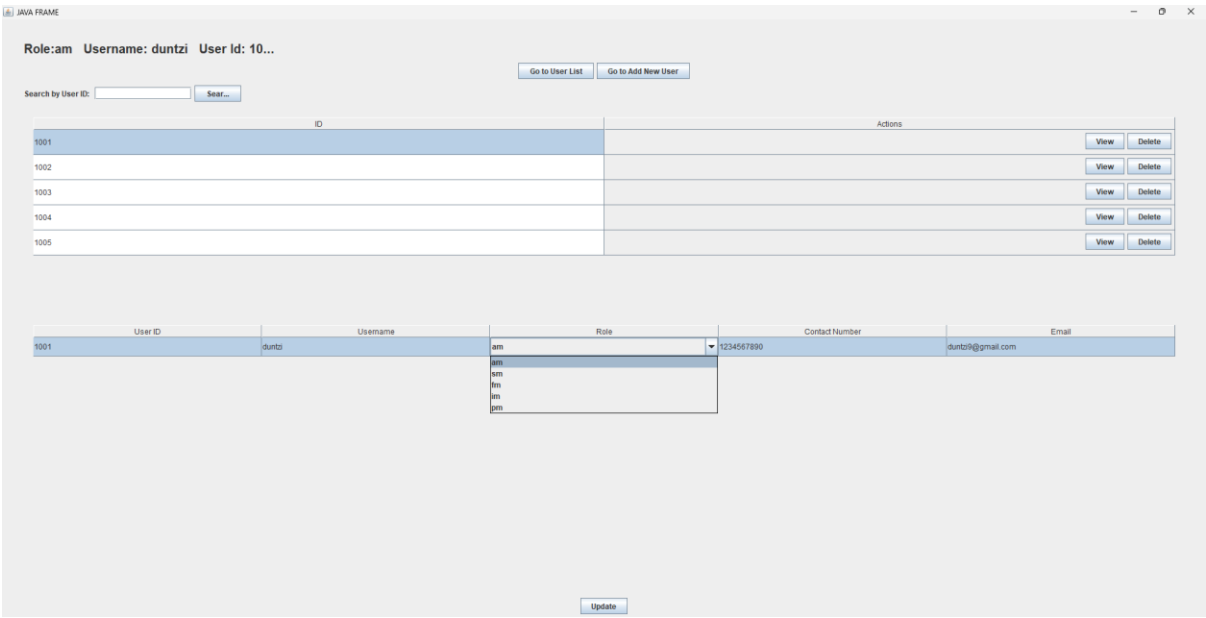


Figure 84: Add New User

Administrators can use the "Add New User" page to add a new user by entering the username, password, choosing a role and giving their contact number and email address. A valid password is made of 8 characters, and it includes 6 digits and 2 letters. Moreover, each contact number must be 10 to 11 digits long and every email address needs to end in "@gmail.com". Should any of the above conditions not be met, a suitable error message will be shown showing what the problem is. Additionally, the user ID all is a unique primary key, no two people will have the same user ID.



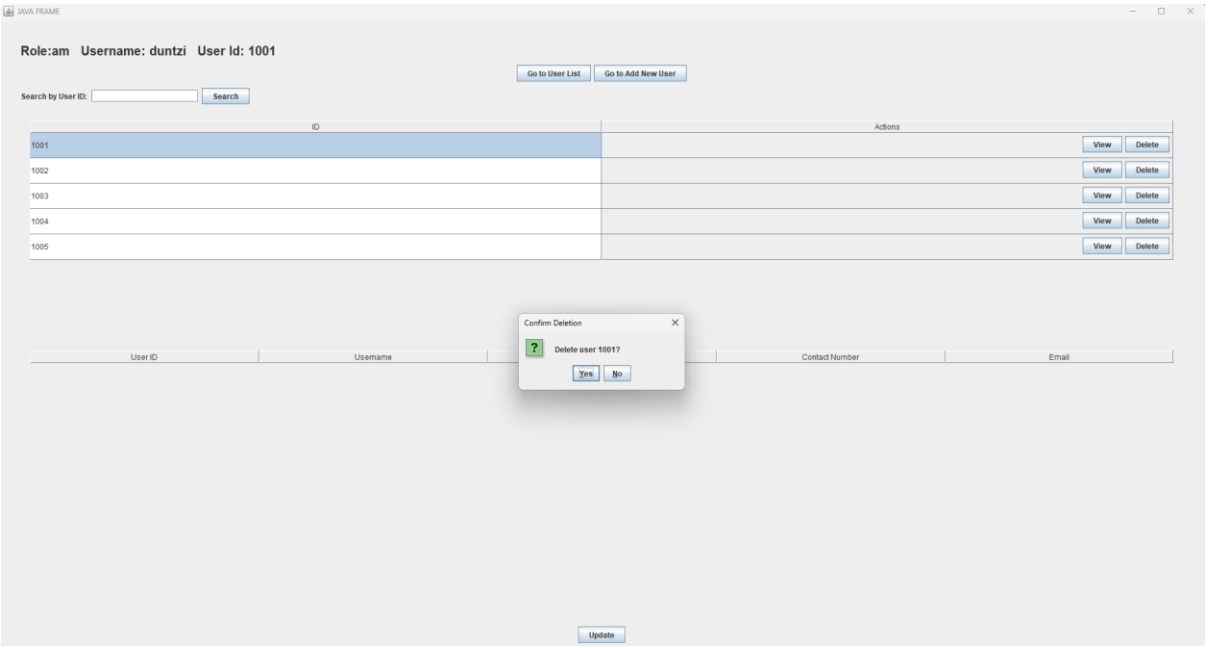


Figure 85: User List Update or Delete

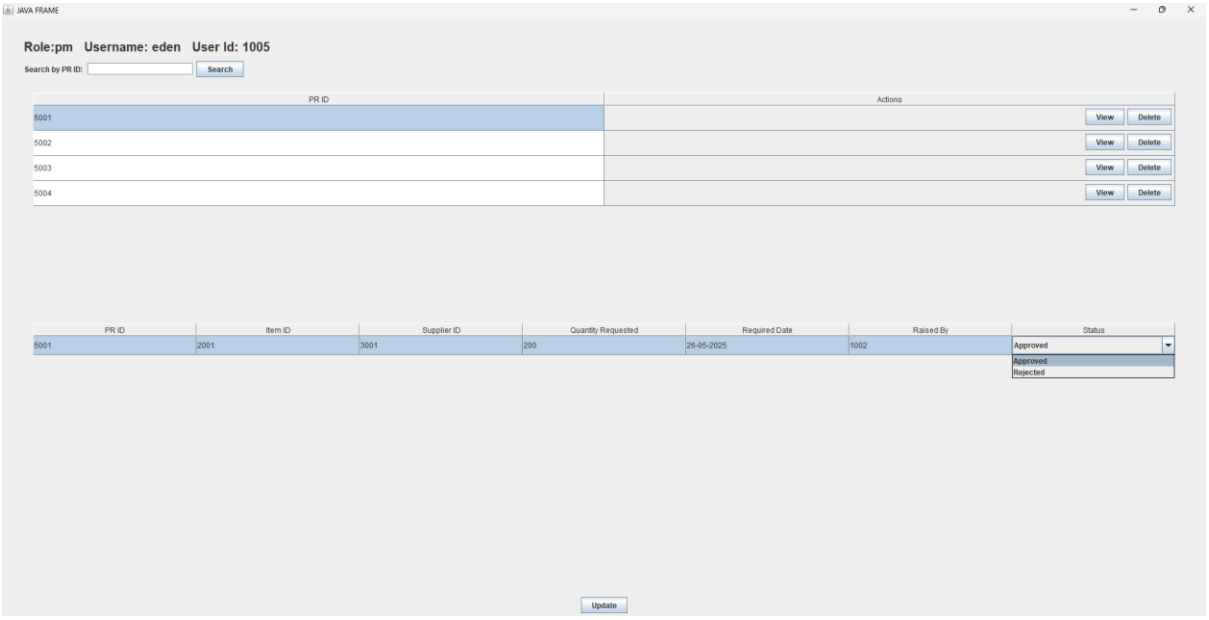
On the User List page, the administrator is able to view at or delete user accounts. Therefore, clicking "View" opens the chosen user's details in the table, giving the administrator an option to update the username, role and both the contact info and email address which all have to meet the acceptable form. Also, confirmation pop-up appears after clicking "Delete" before the account gets permanently removed. Additionally, a search bar appears on top left, allowing the administrator to look up any user by their User ID with ease.

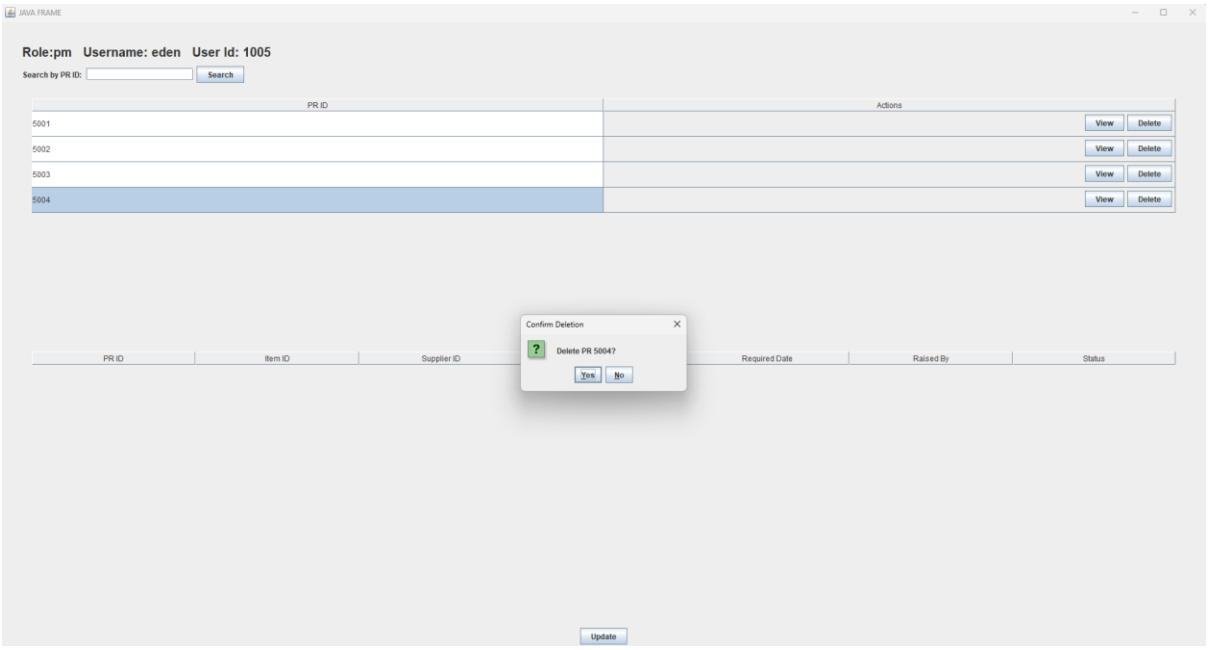
Finance Status:

Figure 86: Finance Status

On the Finance Status page, the administrator goes through every transaction submitted by the Finance Manager and marks the approval. Checking these items helps ensure nothing is wrong with the financial transactions on user account. A search function is available to help the administrator quickly find a transaction by looking up its Finance ID.

PR status:


JAVA FRAME
Role:pm Username: eden User Id: 1005
Search by PR ID:
Search
PR ID
Actions
5001
5002
5003
5004
View
Delete
PR ID
Item ID
Supplier ID
Quantity Requested
Required Date
Raised By
Status
5001
2001
3001
200
26-05-2025
1002
Approved
Approved
Rejected
Update


JAVA FRAME
Role:pm Username: eden User Id: 1005
Search by PR ID:
Search
PR ID
Actions
5001
5002
5003
5004
View
Delete
Confirm Deletion
Delete PR 5004?
Yes
No
PR ID
Item ID
Supplier ID
Required Date
Raised By
Status
Update

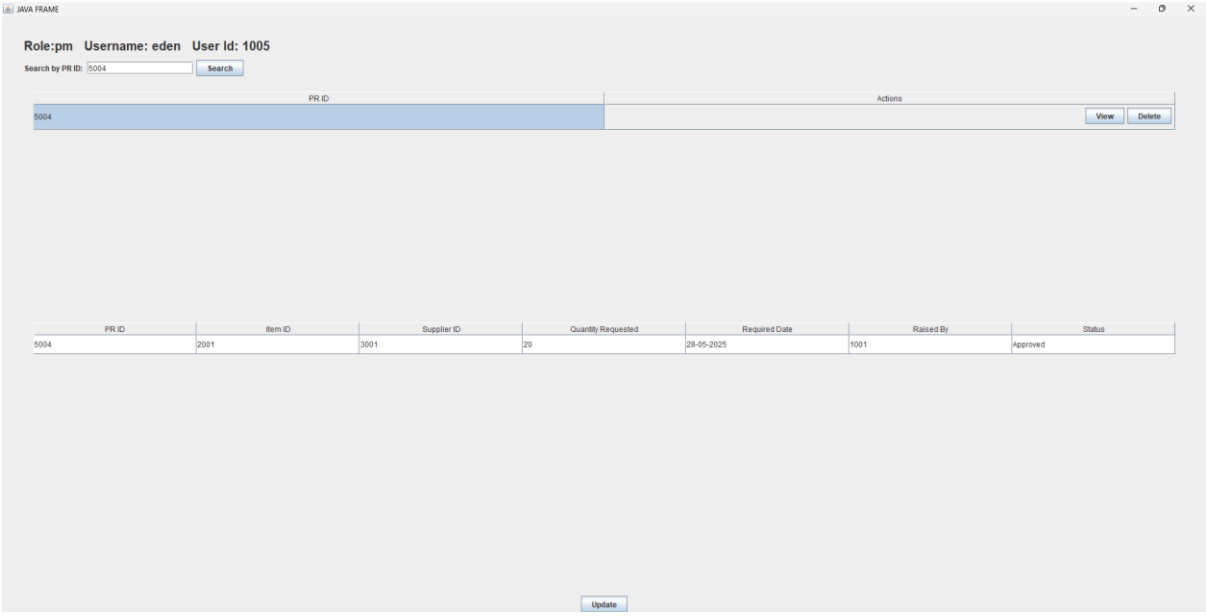


Figure 87: Purchase Request Status

The Purchase Manager can access all the details of a request by pushing the “View” button and selects “Approved” or “Rejected” depending on the request’s status. Selecting “Delete” will let the Purchase Manager remove a request, but a confirmation will be needed before it deletes the request. Besides that, users can find specific requests by typing in the PR ID in the search bar.

Supplier Creation:

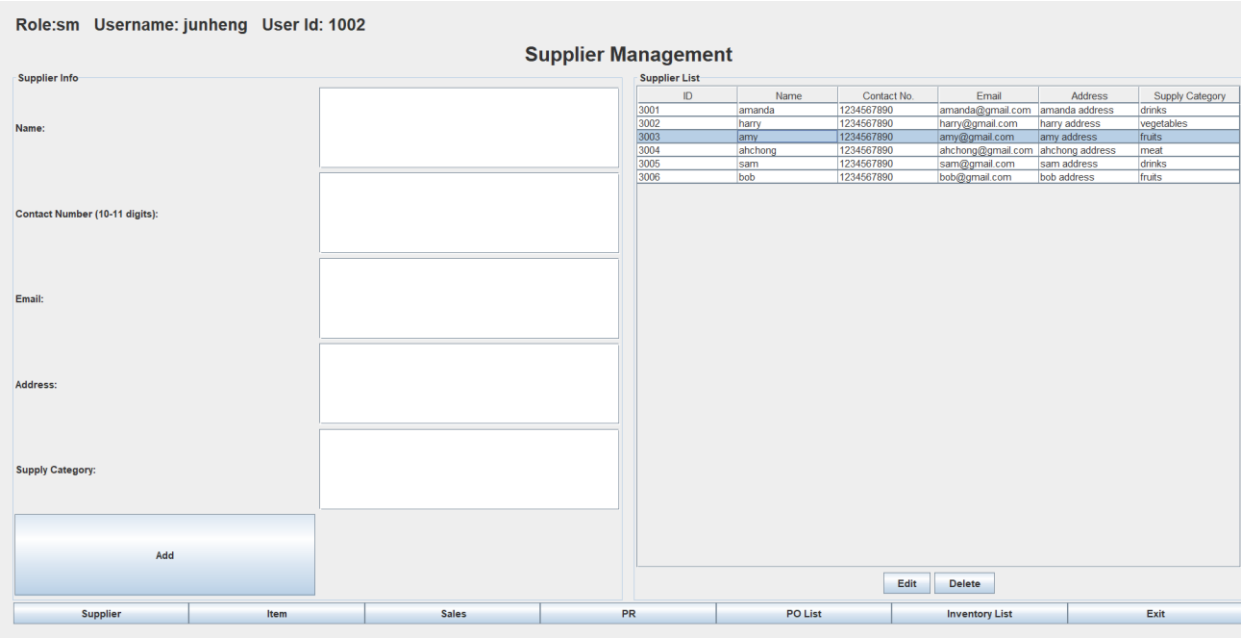


Figure 88: Supplier Page

This is supplier creation and view page for sales manager and administrator so sales they can insert their name, contact number, email, address and supply category after that press add it will display and the right side that is view supplier tables. Sales manager and admin must input the right format of the contact number (10-11digit only) and email (@ gmail.com), if the sales manager input wrong format it will ask to input again with the right format.

At the supplier view tables, sales manager and admin can view the supplier information if want to delete can press the column that sales manager wants to delete, and press delete or edit button.

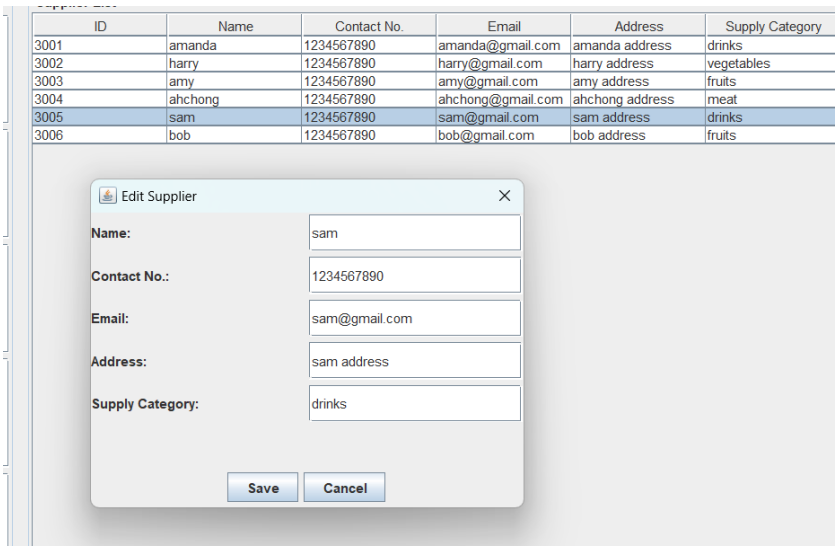


Figure 89: Supplier Edit button

This is the interface that when clicking the edit button.

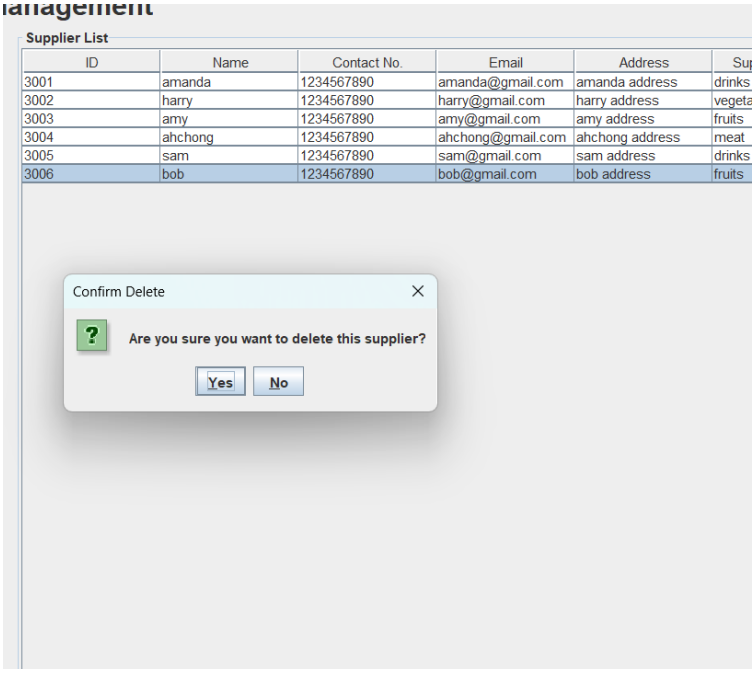


Figure 90: Supplier Page Confirm Delete

This is the interface that want to delete, it will let sales manager and administrator to double confirm that want to delete or not.

Item Creation:

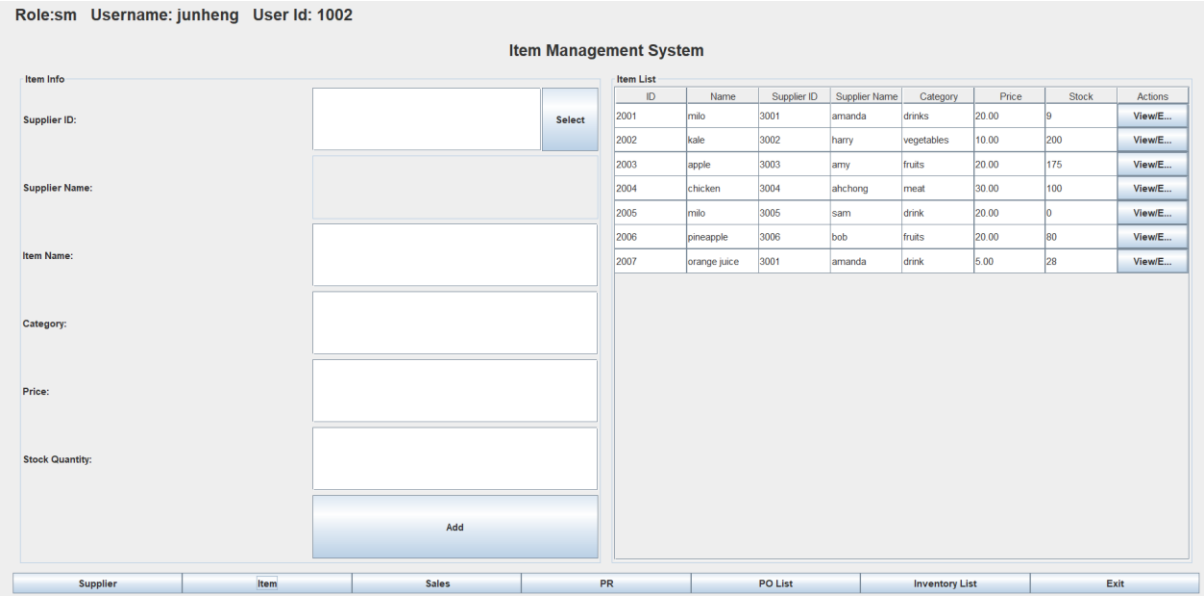


Figure 91: Item Page

This is the item page that only allow for sales manager so sales manager and administrator need to choose the supplier id and the supplier's name will automatically show at the supplier's name

column, insert the supply item, categories, price (integer only) and stock quantity (integer only) that supplied by the supplier. After adding the item info, it will display the item list at the right-hand side that is item list table. Sales manager and admin can see the supplier that supplied what item and edit or delete the item list.

Sales Creation:

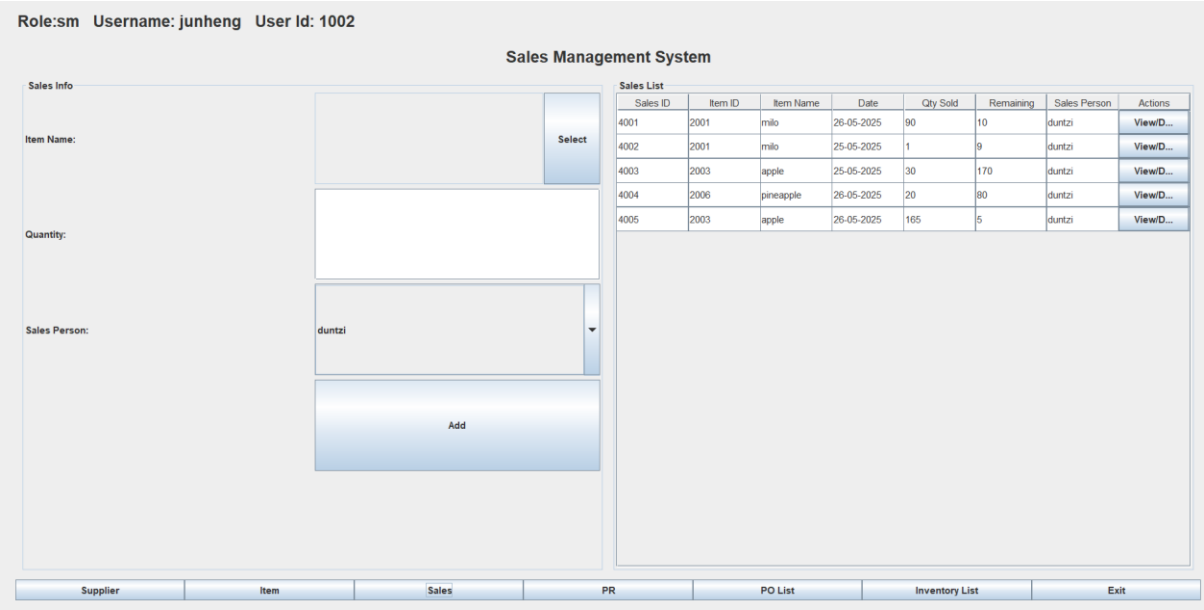


Figure 92: Sales Page

This is sales creation and view sales page that for sales manager and administrator, so they will need to select the item that been sell to customers, input the quantity that have been sold (integer only) and choose the salesperson.

After adding the sales, it will display at right hand side that is the sales list, so sales manager and admin can view the sales list and edit the or delete the list if adding wrong information

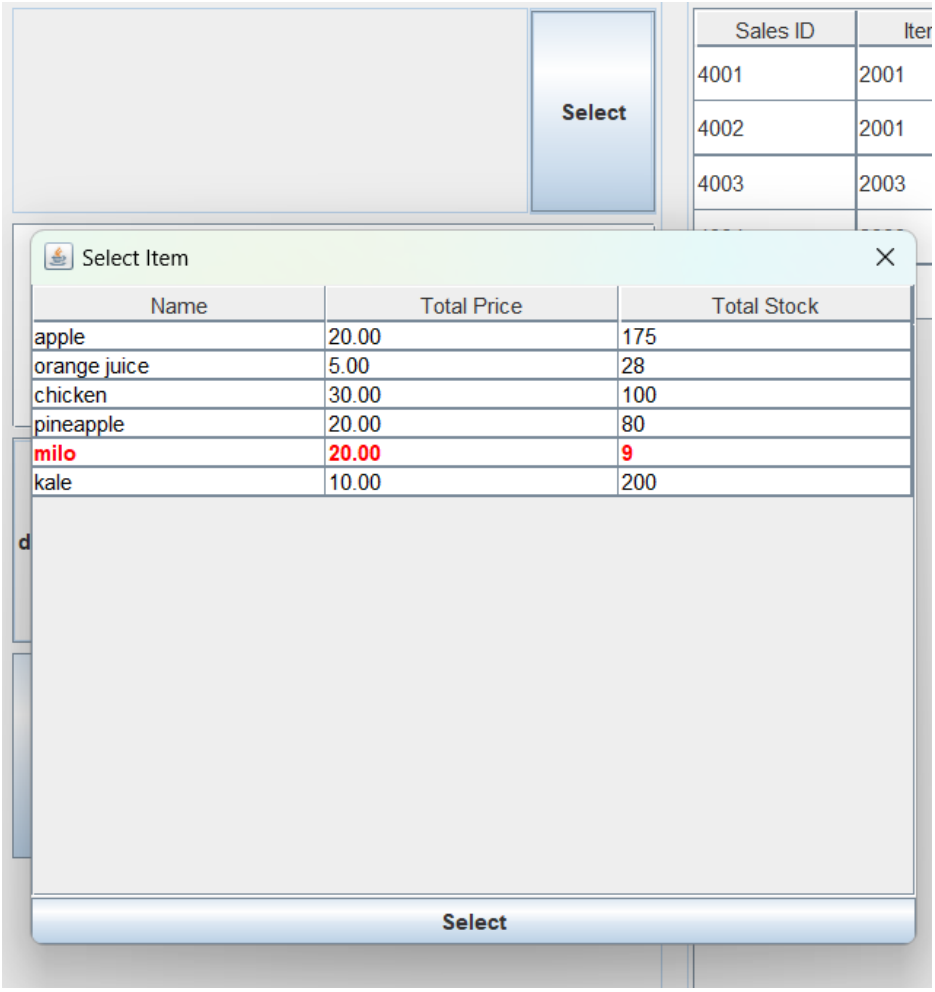


Figure 93 Sales select item

So, when sales manager and admin selecting the item when the stock is below 10 it will turn red to let them know the stock is less and going to ask for more product. When it has two suppliers supplying one product also will combine into one row and show the total stock of the two products.

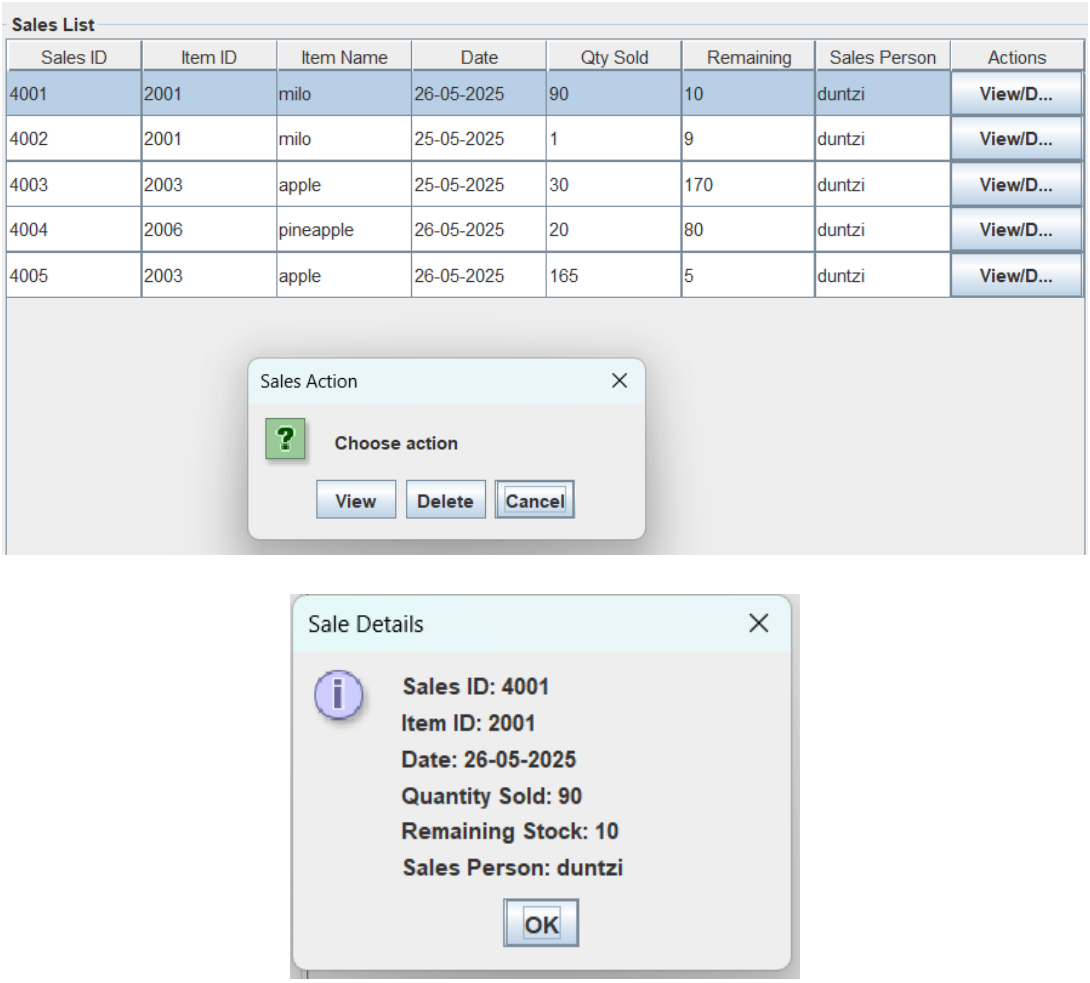


Figure 94 sales action

So, at the right-hand side that is the sales list so when sales manager and admin can click the action button to view for clearly details or delete the sales row.

Supplier List:

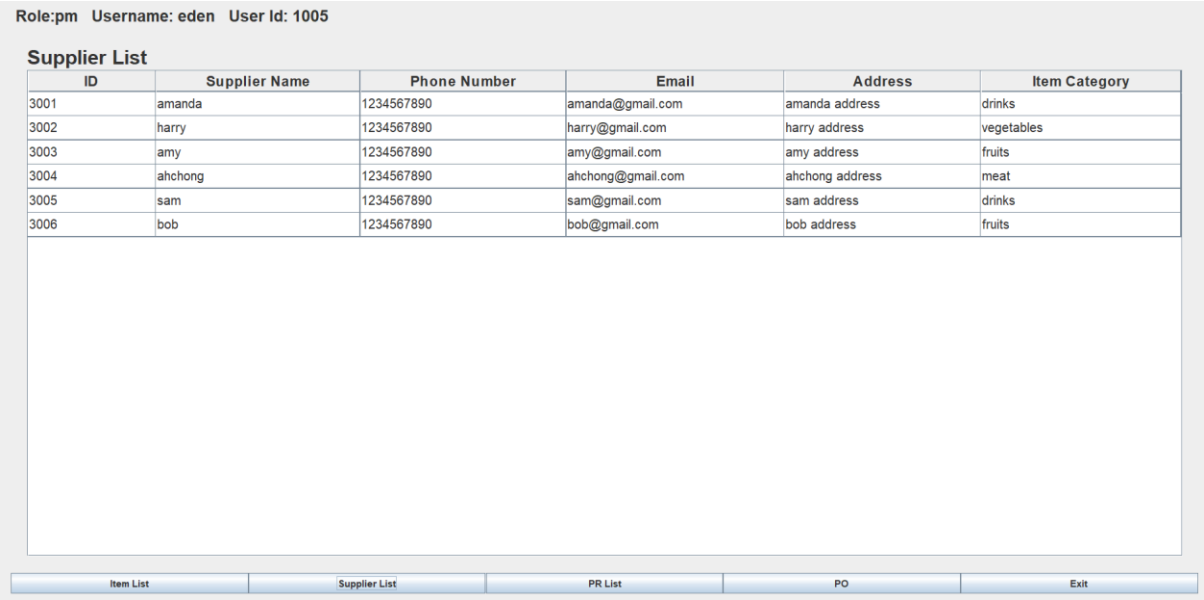


Figure 95: Supplier list Page

This is the supplier view list for purchase manager, so in this page purchase manager can view the id, supplier name, phone number, email, address and the item category.

Purchase Request:

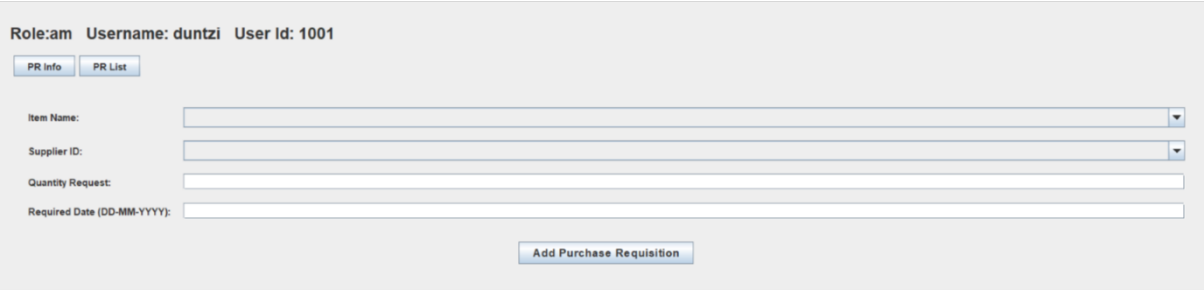


Figure 96: Output of pr_e.java

This is output of pr_e.java, and it is related with items.txt. Since the items.txt file doesn't have item below 10 quantities, so the pr_e.java file "PR Info" will not appear any data. If there are items are same (milo) supplied from different supplier, it will automatically total up, and if the quantities below 10, it will show in pr_e.java.

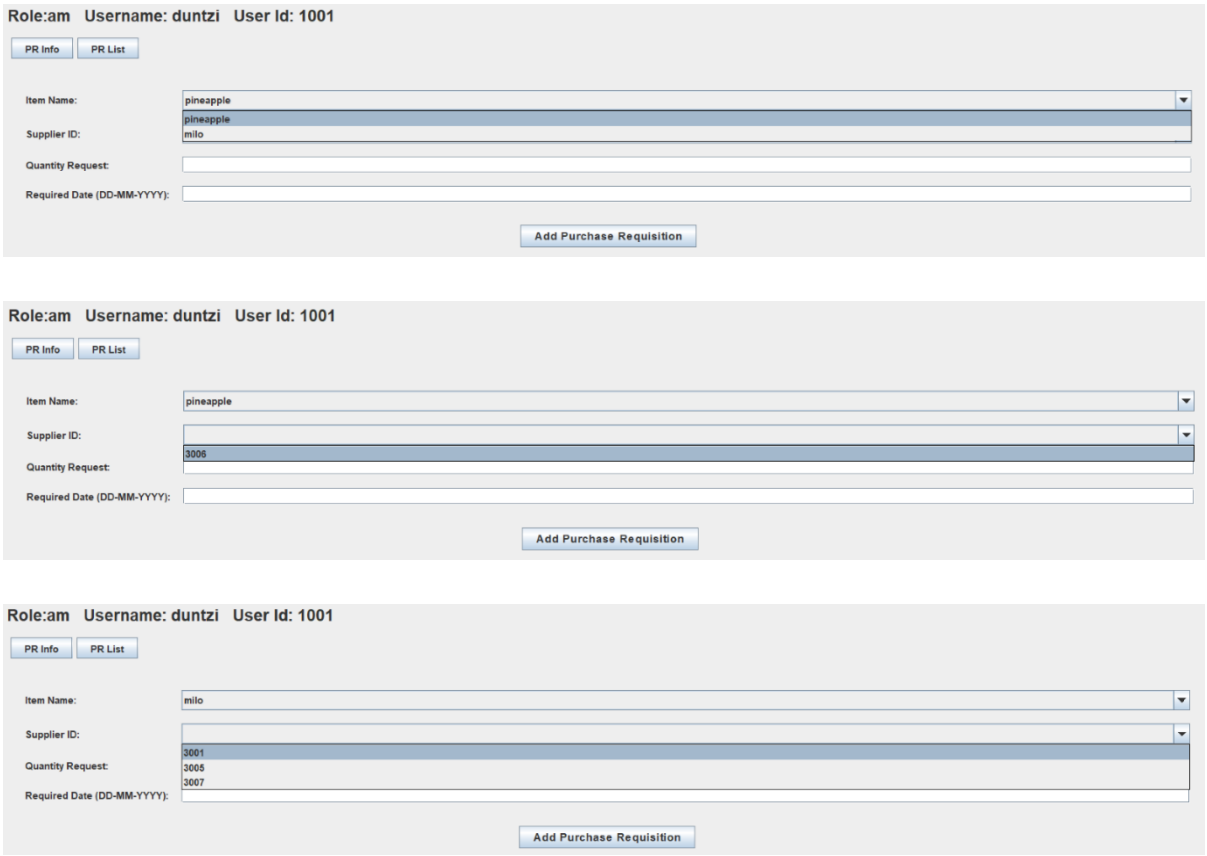


Figure 97: Output of pr_e.java

The items.txt file “milo” stock quantity has only 6; “pineapple” stock quantity has only 3, so the system automatically raise a signal. Now Administrators/Sales Managers can choose the item they want to add purchase requisition first, Item Name now will have two selection “milo” and “pineapple”, and they can choose the supplier they want to purchase from based on the Supplier Id.

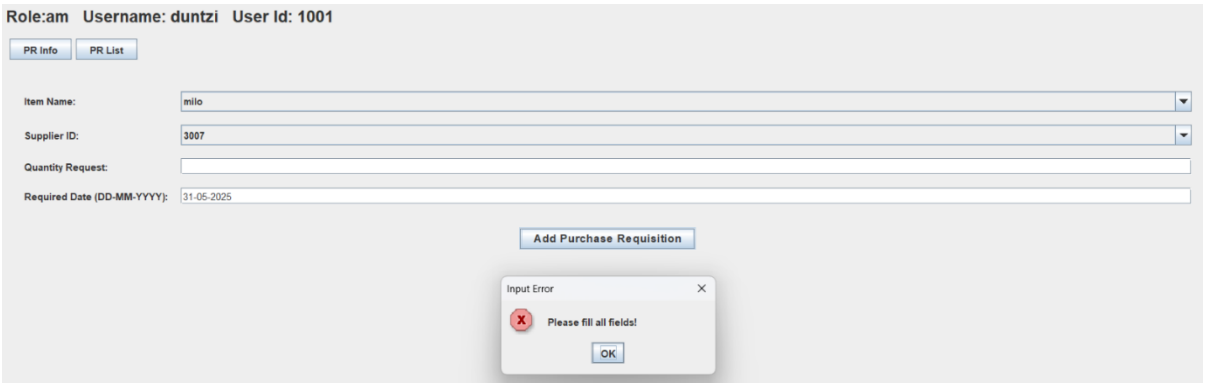


Figure 98: Output of pr_e.java

Administrators/Sales Managers is required to fill in all the information, else it will show error message.

Figure 99: Output of pr_e.java

After Administrators/Sales Managers selected and filled in all the information, they now can click the button “Add Purchase Requisition”.

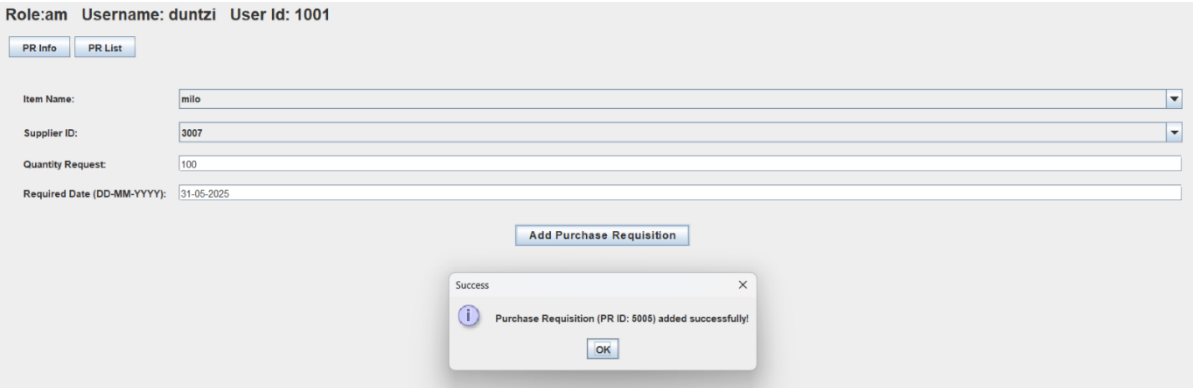


Figure 100: Output of pr_e.java

After Administrators/Sales Managers click the button “Add Purchase Requisition”, it will show the message that telling “Purchase Requisition added successfully!” and store the data into the pr.txt file.

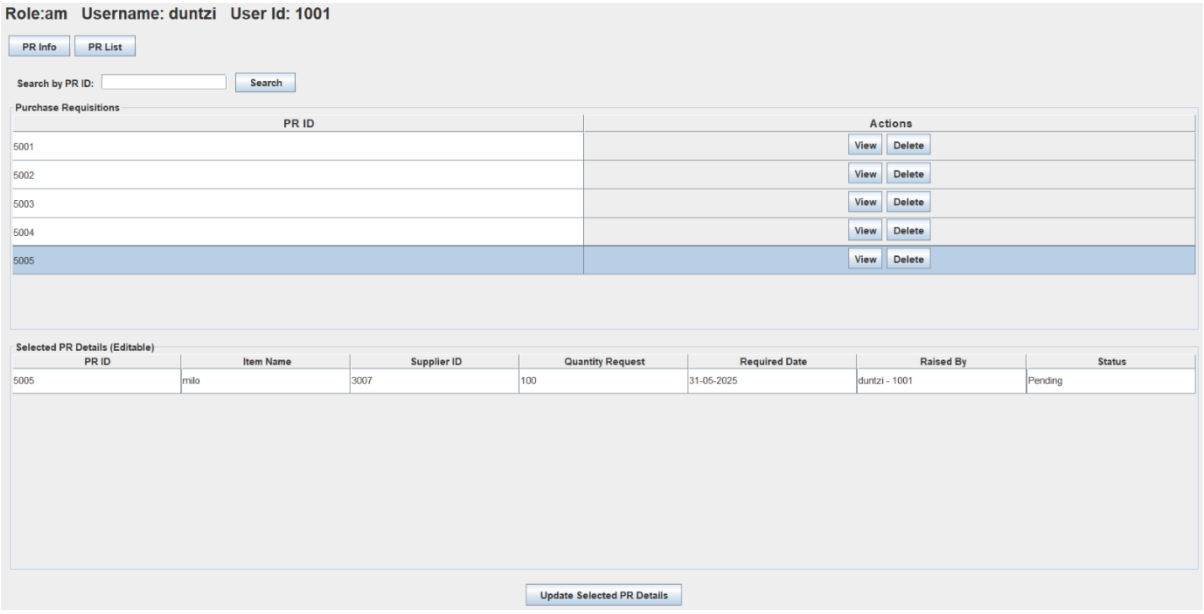


Figure 101: Output of pr_e.java

In the “PR List” page Administrators/Sales Managers can view the Purchase Requisition by clicking on the “View” button. Only the “Quantity Request” can be modified and update the pr.txt file by Administrators/Sales Managers. The “Raised By” will automatically detect the person who Purchase Requisition.

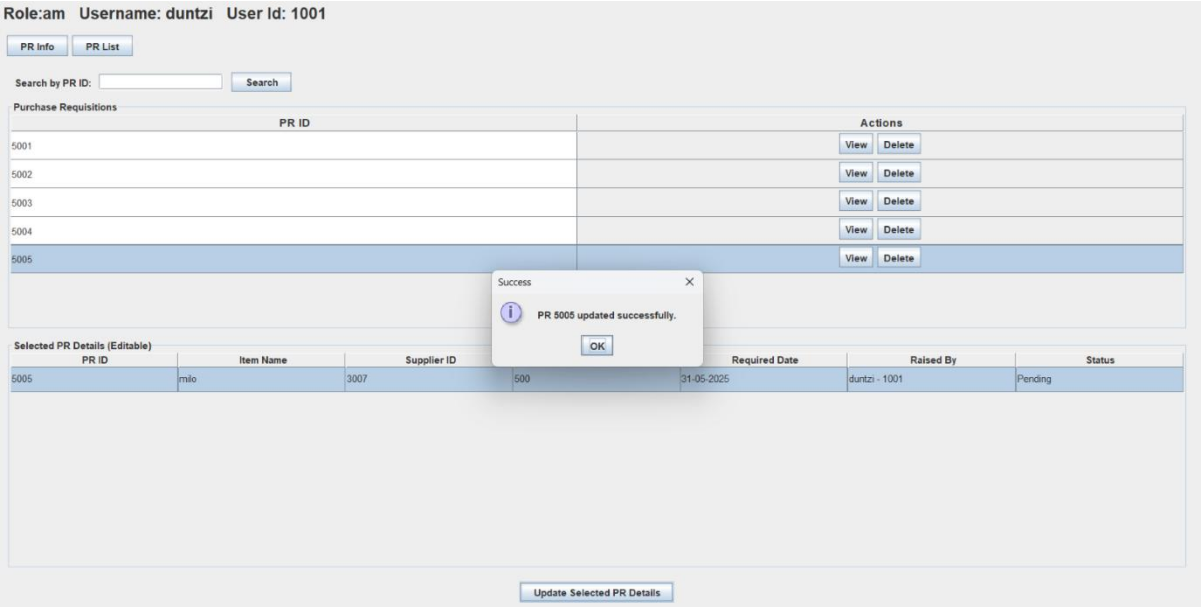


Figure 102: Output of pr_e.java

Now the “Quantity Request” from “100” --> “500”.

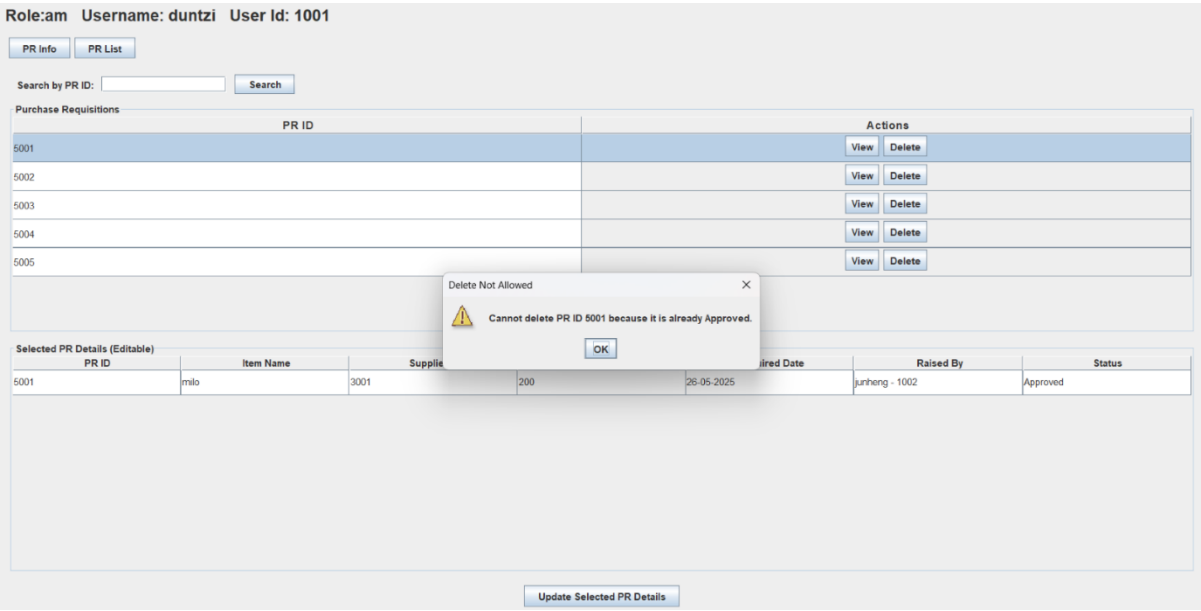


Figure 103: Output of pr_e.java

Administrators/Sales Managers unable to delete the Purchase Requisition that Status is “Approved”. After they click the button “Delete”, it will be showing them a message “Cannot delete because it is already Approved”. Else “Pending” and “Rejected” can be deleted.

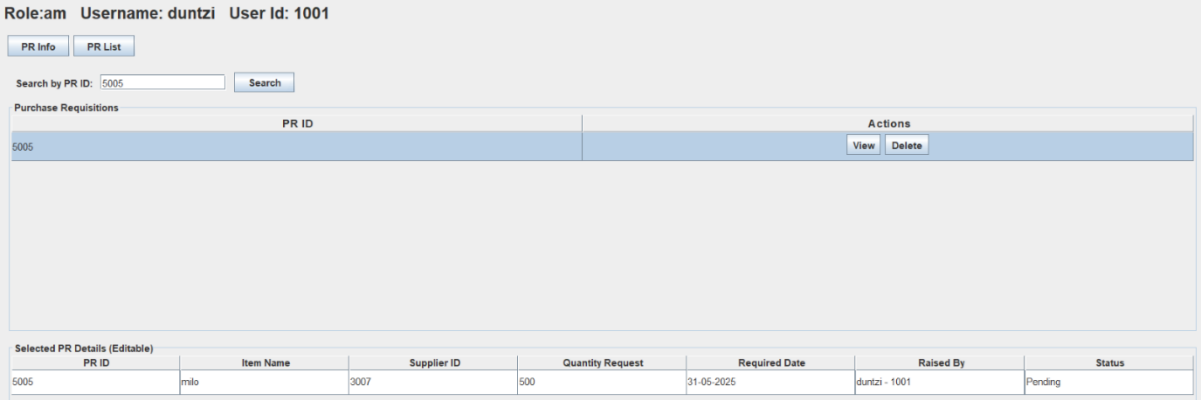


Figure 104: Output of pr_e.java

Administrators/Sales Managers can search the Purchase Requisition by typing the PR ID. Imagine if there are thousands of Purchase Requisitions, it will be wasting a lot of time to find that Purchase Requisition one by one, by this function it can reduces the time wasting.

Purchase Order:

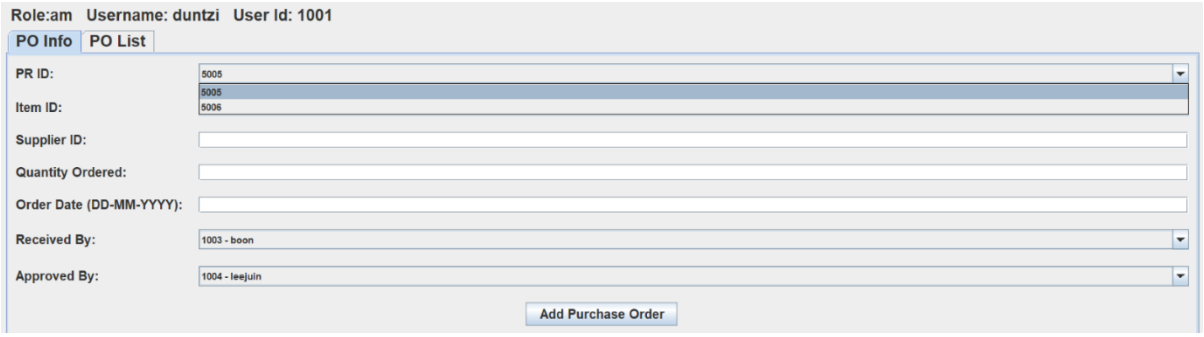


Figure 105: Output of po_e.java

They are two Purchase Requisitions which are “5005” and “5006” waiting to Purchase Order, and they are ready to Purchase Order because in the pr.txt file their status is “Approved”.

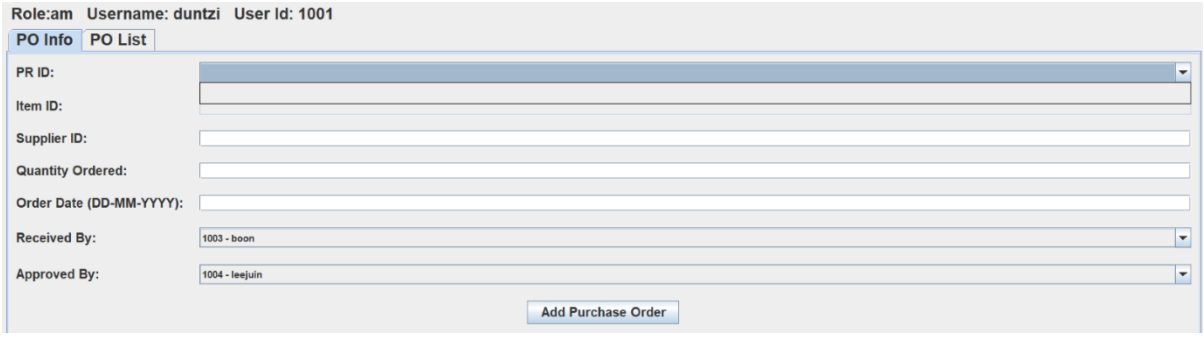


Figure 106: Output of po_e.java

If the pr.txt file status is not “Approved” the data will never show in the po_e.java “PO Info”. The others PR ID such as: “5001”, “5002”, “5003”, and “5004” is not showing although they are “Approved” because Administrators/Sales Managers have been raised Purchase Order and “Approved” by Financial Managers.

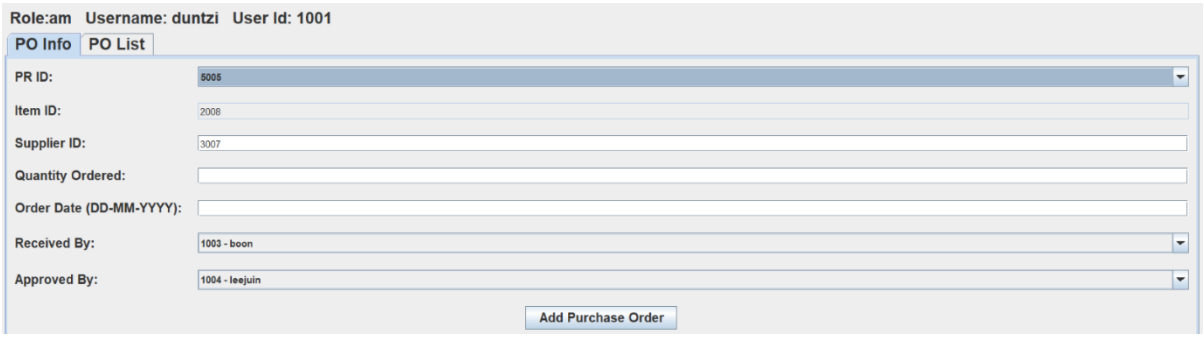


Figure 107: Output of po_e.java

Now Administrators/Purchase Managers can raise the Purchase Order, after they selected the PR ID, the system will automatically fill in “Item ID” and “Supplier ID”. The Item ID is not editable; the Supplier ID is editable.

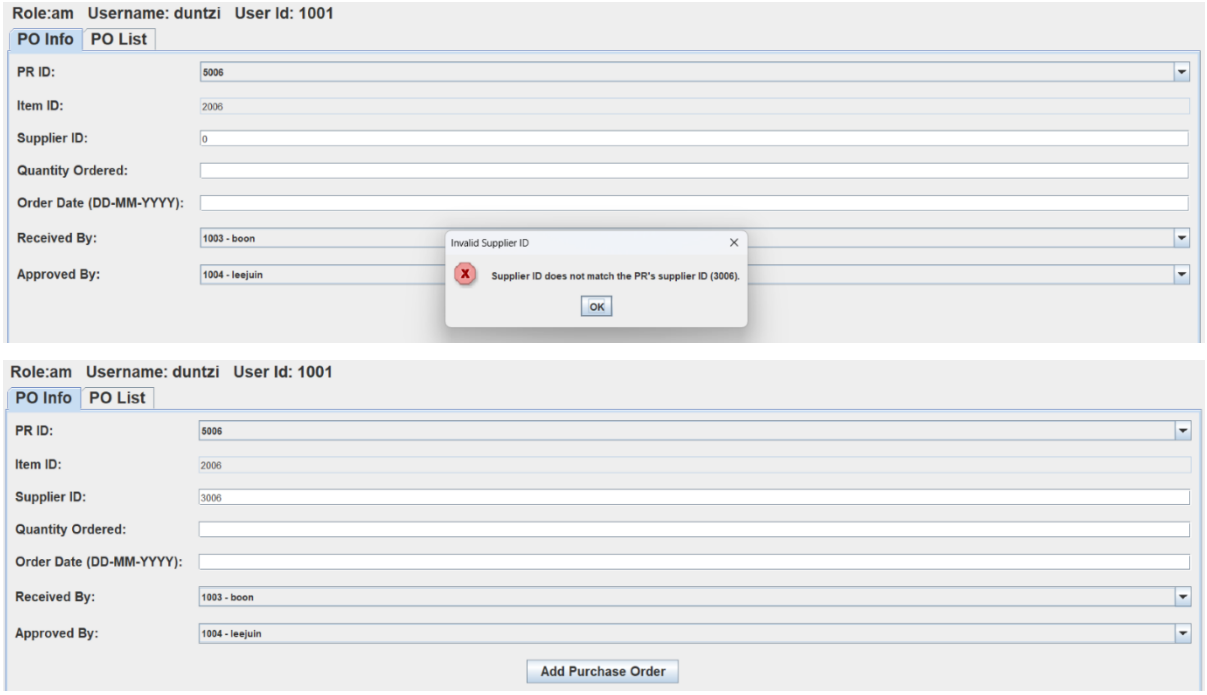


Figure 108: Output of po_e.java

But for the prevent human error typing, it designed to auto fill in the correct Supplier ID after detected human error typing, so it ensures the system is reliable.

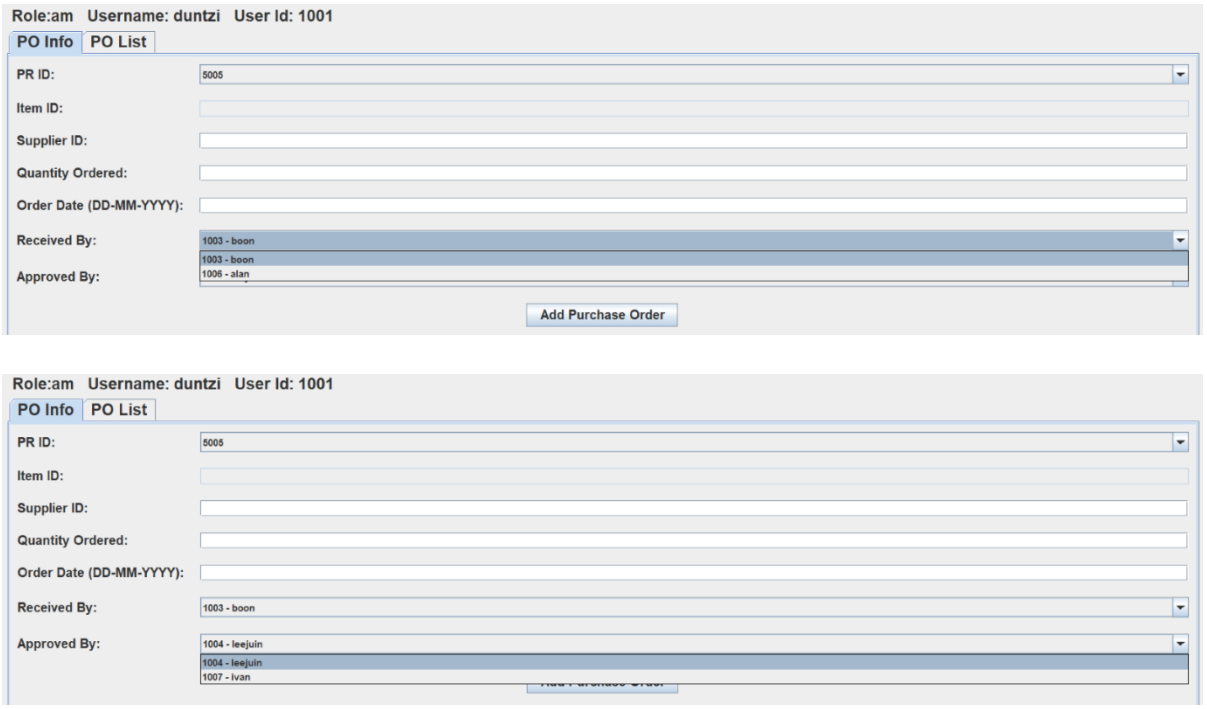


Figure 109: Output of po_e.java

Based on users.txt file, Administrators/Purchase Managers can select the person who "Received By" and "Approved By".

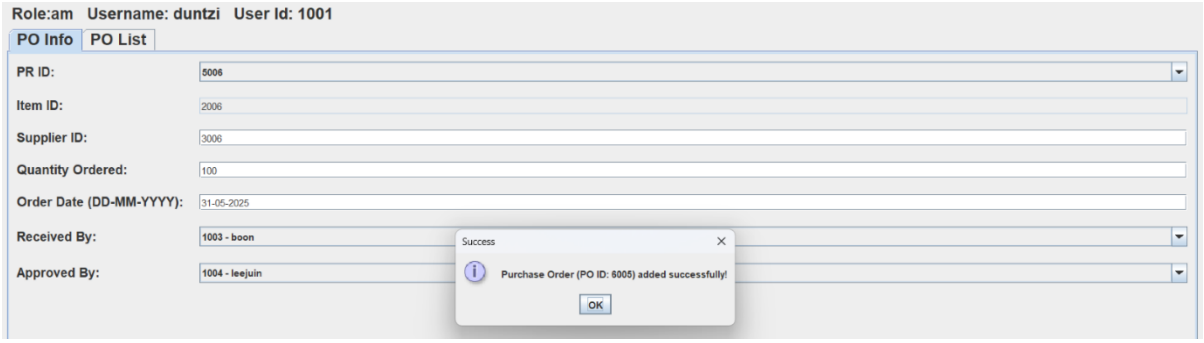


Figure 110: Output of po_e.java

After Administrators/Purchase Managers click the button “Add Purchase Order”, it will show the message “Purchase Order added successfully!”. So, it indicates the data is successful stored in the po.txt file and the new PO ID is waiting to be Approved by Financial Managers.

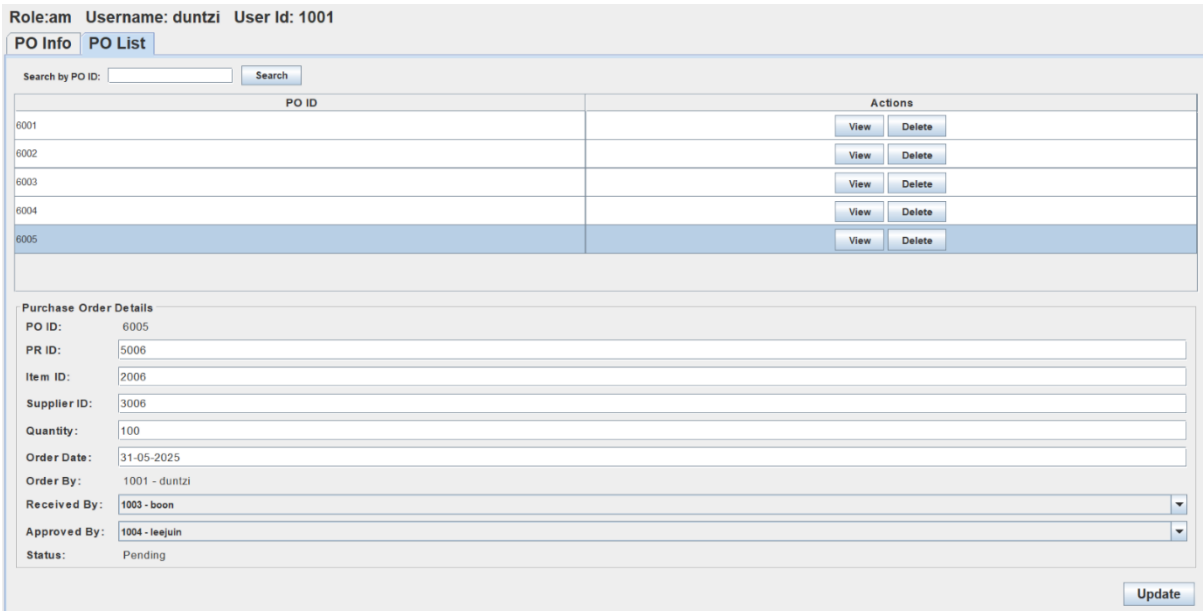


Figure 111: Output of po_e.java

In the “PO List” page, Administrators/Purchase Managers can click on the button “View”, to check the information, and Administrators/Purchase Managers able to edit the data but except for “PO ID”, “Order By” and “Status” is unable to edit. The “Order By” will automatically detect the person who Purchase Order.

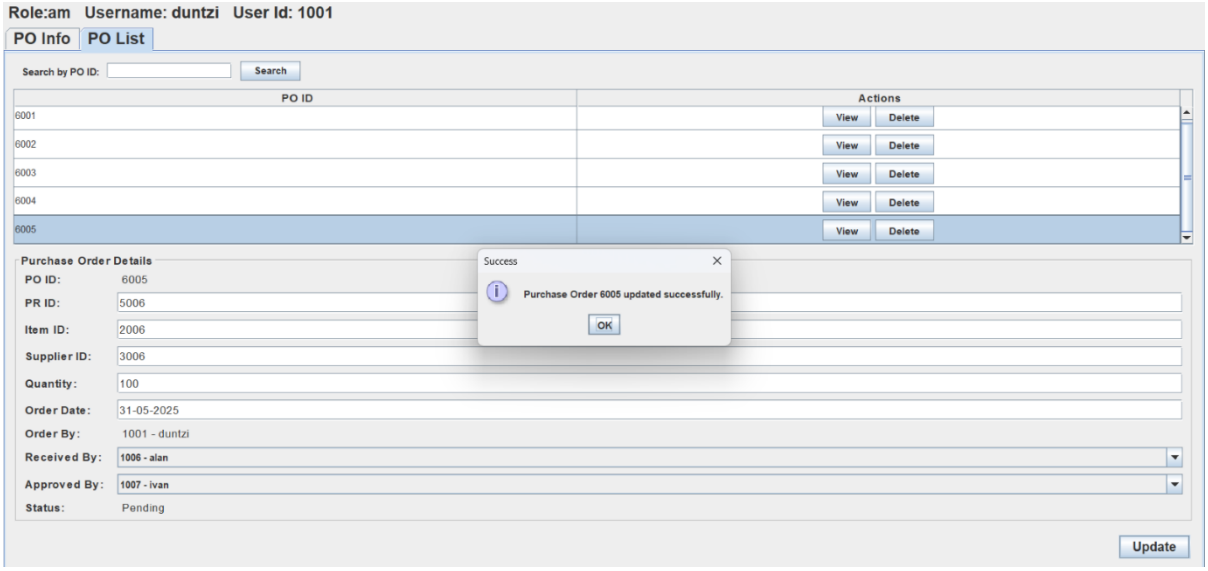
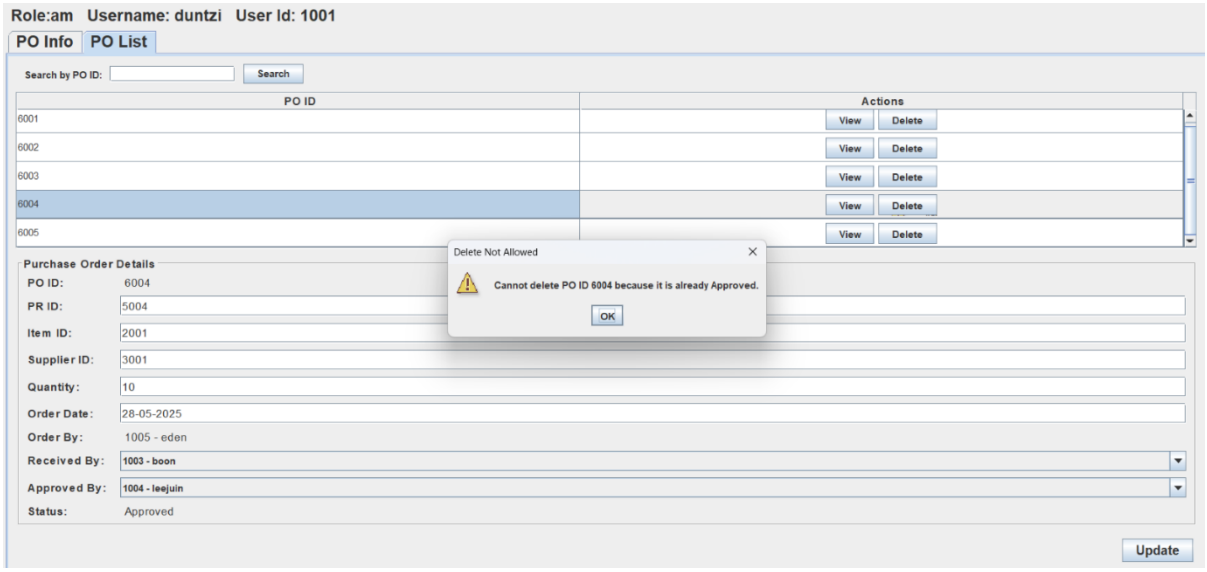


Figure 112: Output of po_e.java

After Administrators/Purchase Managers click the button “Update”, it will show a message about “Purchase Order updated successfully”, the po.txt file has changed.



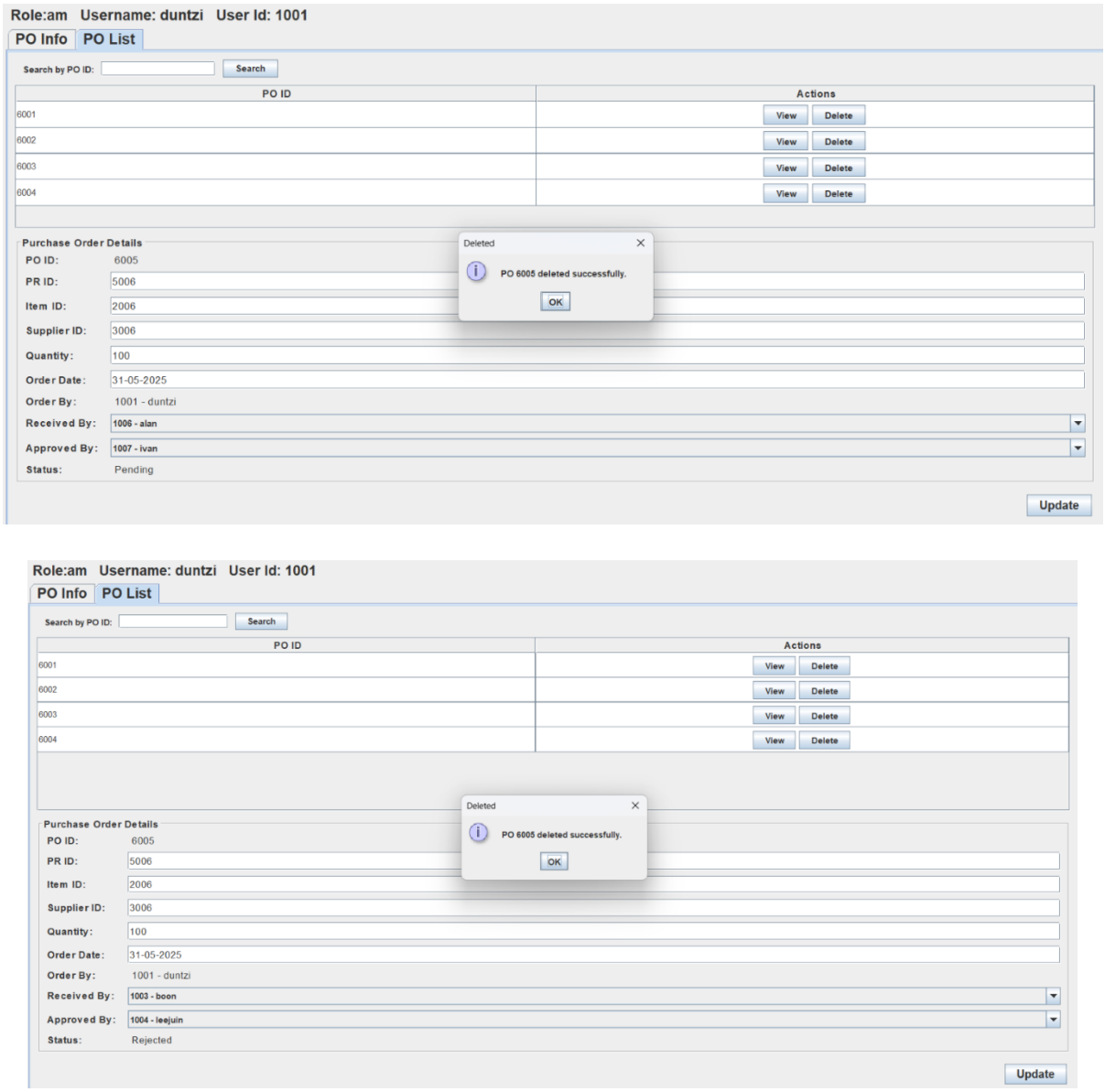


Figure 113: Output of po_e.java

The Purchase Order cannot be deleted by Administrators/Purchase Managers if the status is “Approved”, only if the status is “Pending” or “Rejected”.

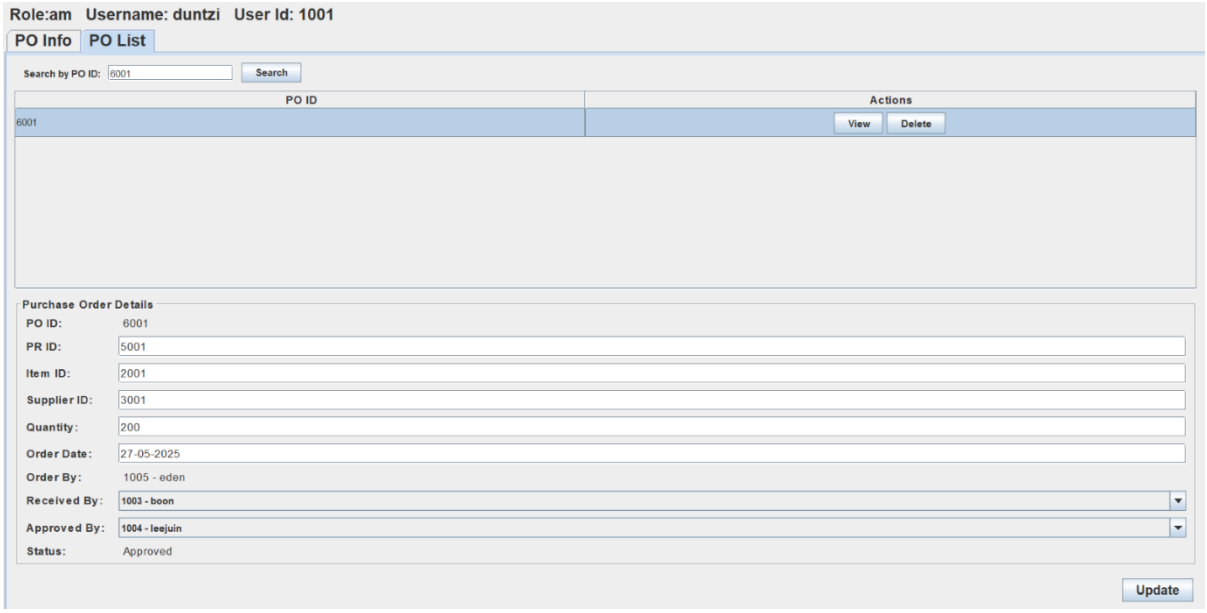


Figure 114: Output of po_e.java

The system provides Administrators/Purchase Managers for search the Purchase Order, based on the PO ID. This search function can make reduce time wasting when handling many of the Purchase Order.

Purchase Request List:

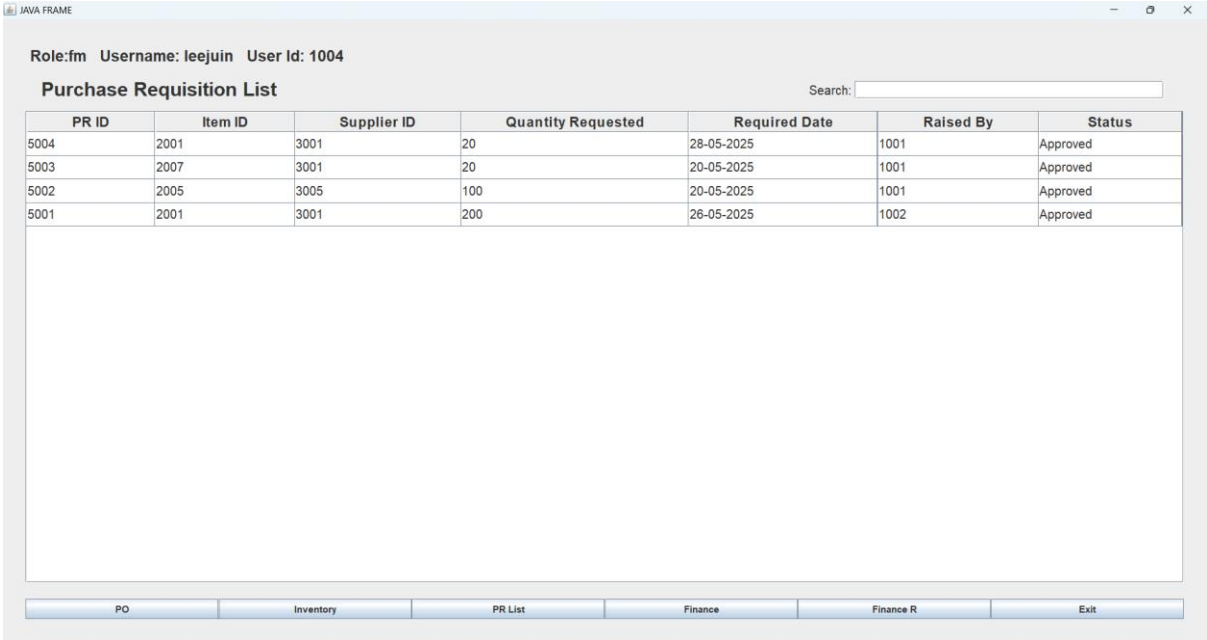


Figure 115: Purchase Request List Page

This is "Purchase Requisition List" page, it is accessible by finance manager only. The first parts of the bar, known as static labels, tell us that the user is an employee ("fm"), their name is "leejuin" and their ID is "1004". Underneath, the main "Purchase Requisition List" title is shown. Next to the data table, there is a "Search" field that people can use to quickly filter the table's data while typing. A clear and concise table forms the basis of the display and it contains key header labels: "PR ID", "Item ID", "Supplier ID", "Quantity Requested", "Required Date", "Raised By" and "Status", with all fields in a readable typeface. For each row in the table, the details of a purchase requisition are shown, arranged so that PR IDs are sorted highest to lowest.

Purchase Order List:

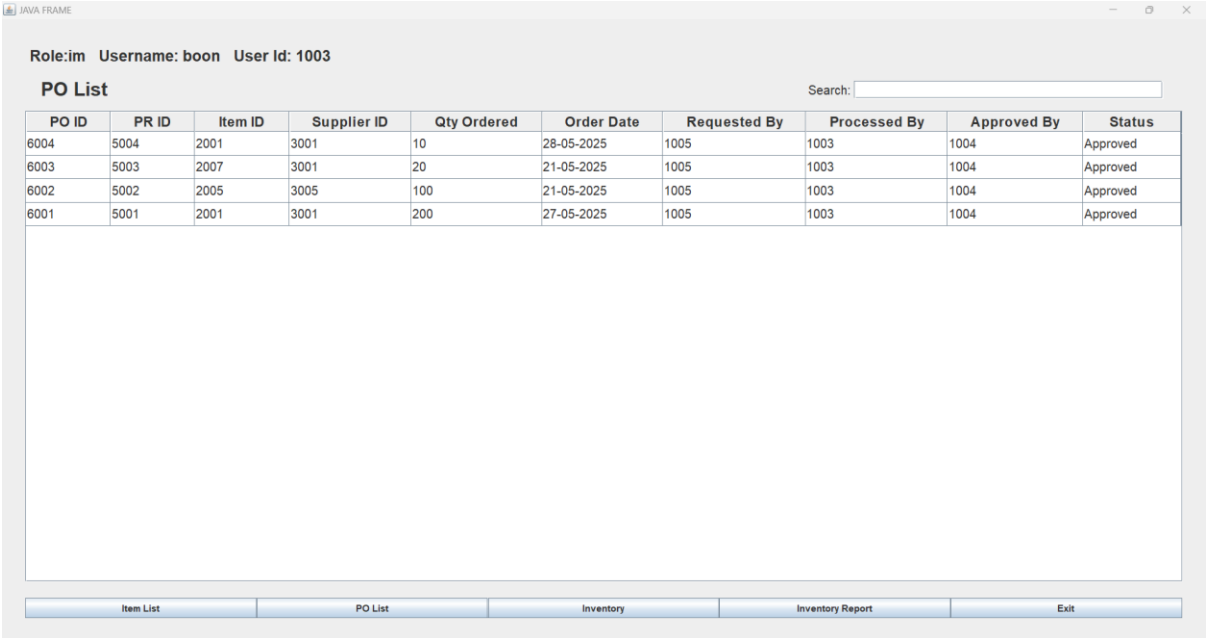


Figure 116: Purchase Order List Page

This is “Purchase Order List” page, it is accessible by inventory manager only. At the very top, the application shares details that only the logged-in user can see, like "Role:im", "Username: boon" and "User Id: 1003". The main part of the interface, the "PO List," has a title, with a "Search:" input bar on the right to let the user search for purchase orders by typing what they are looking for. In the main area, there is a well-ordered JTable that presents all the important Purchase Order information in columns, namely: “PO ID”, “PR ID”, “Item ID”, “Supplier ID”, “Qty Ordered”, “Order Date”, “Requested By”, “Processed By”, “Approved By” and “Status”. Sorted by PO ID, this makes finding the newest records easier.

Item List:

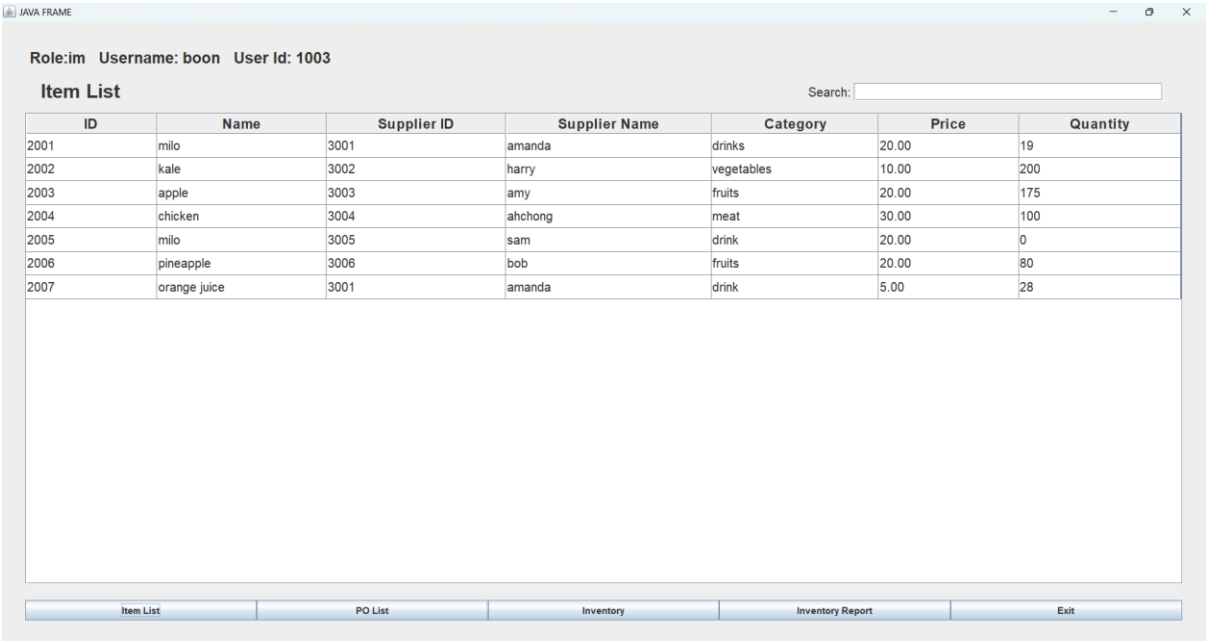


Figure 117: Item List Page

This is "Item List" page, it is accessible by purchase manager and inventory manager. The top of the screen shows the user is logged in and includes details such as "Role:im", "Username: boon" and "User Id: 1003." The "Item List" title is placed under the map, aligned to the left and a search bar on the right with a text field allows users to search tirelessly. The table in the main area has seven columns labeled "ID," “Name,” “Supplier ID,” “Supplier Name,” “Category,” “Price,” and “Quantity,” using easy-to-read fonts and bigger rows.

Inventory Status:

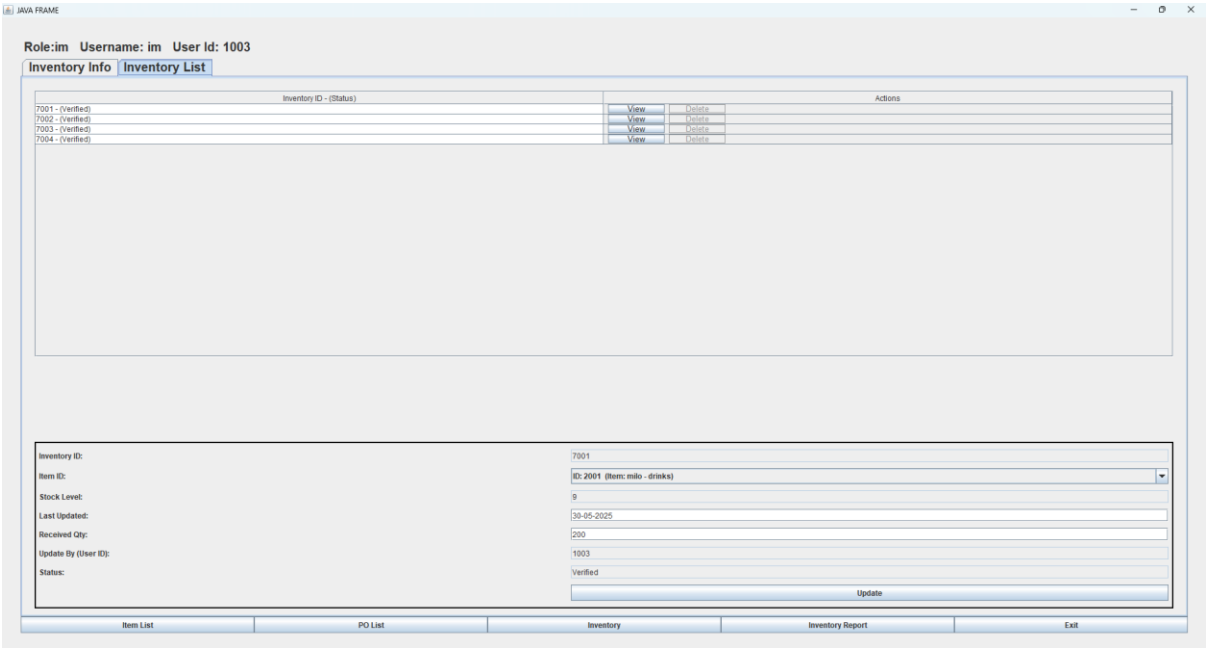


Figure 118: Inventory Status Page

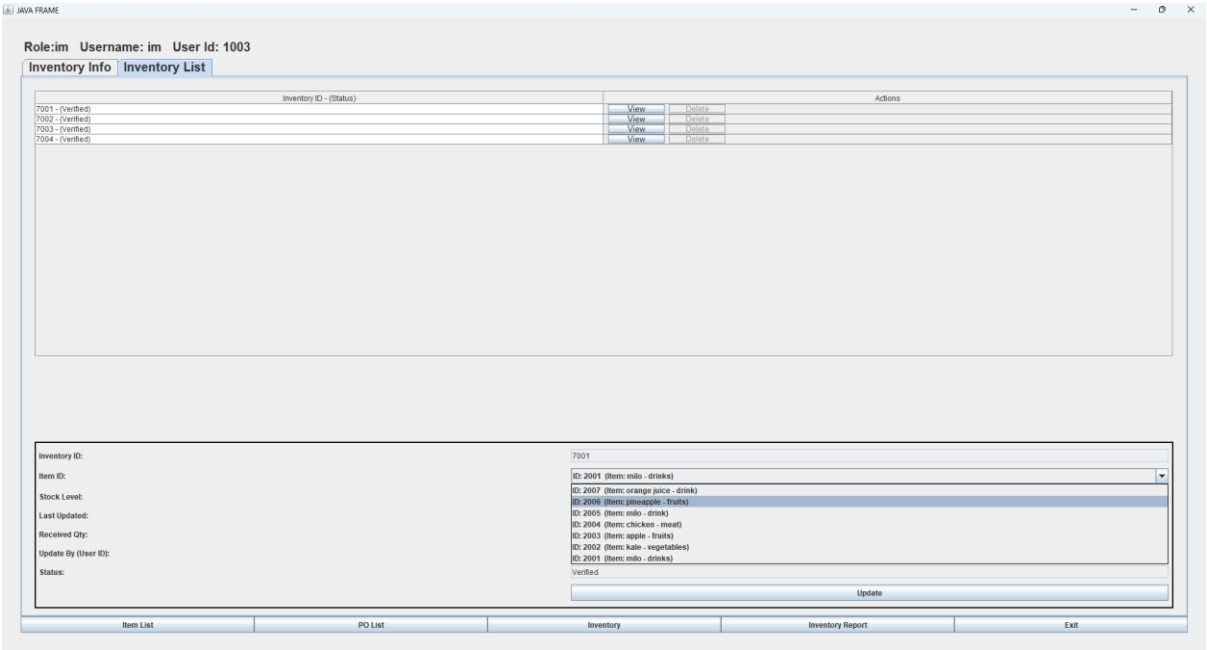


Figure 119: Dropbox for ItemID

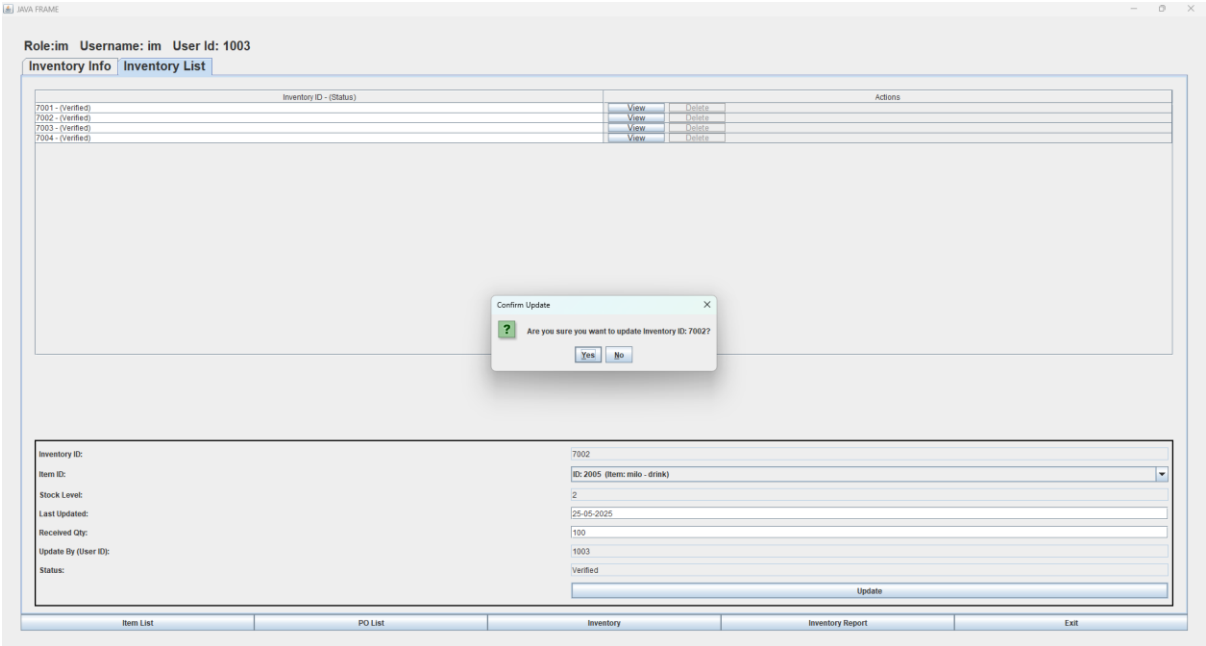


Figure 120: Confirmation for Update Data

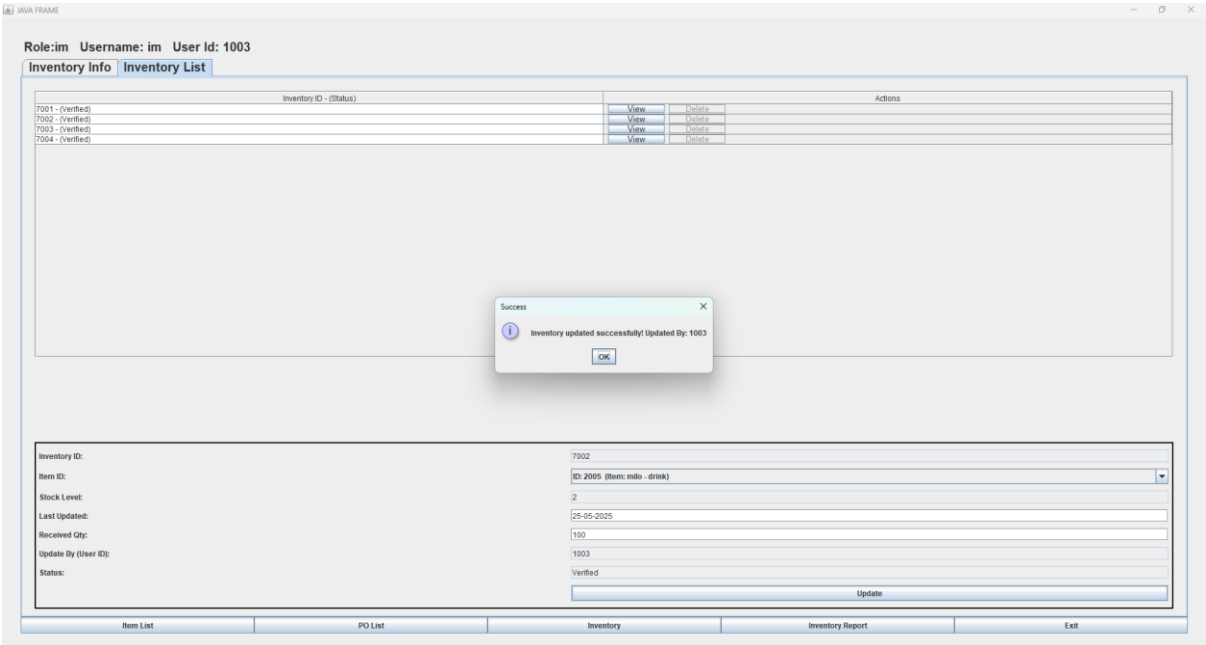


Figure 121: Success Message Appear

This is “Inventory List” page, it is accessible by inventory manager only, which gives a tidy and straightforward way to view and manage what is in the inventory. Right at the top of the page, the role ("Role: im"), username ("Username: boon") and user ID ("User Id: 1003") give immediate information to the user. Underneath these, the tabs "Inventory Info" and "Inventory List" show which functions are available here and "Inventory List" is currently picked.

The central section of the page has a table of listings, showing columns for "Inventory ID - (Status)" and "Actions". Because each row is an inventory ID (such as 7001 or 7002) and its state (Verified), users can notice its condition quickly. The "Actions" column gives viewers an interactive option to look at or delete any product they include in the inventory record. If there is a (Verified) status next to the name, means that not able to delete it.

After the main table, there is a detailed form that enables users to look at and change individual inventory details. The fields are called "Inventory ID," "Item ID," "Stock Level," "Last Updated," "Received Qty," "Update By (User ID)," and "Status". The list of Items is displayed as a dropdown, so users can choose different items and “Received Qty” is a field that can be filled in by the user. User can use the “Update” button to save the new entries in the inventory. When trying to make an update, user can see the dialog "Confirm Update" asking if are sure user want to update Inventory ID: [ID]. with two options: “Yes” and “No”. After updating the inventory correctly, the message box appears and says "Inventory updated successfully! The content was reviewed and revised by [User ID].

Finance Report:

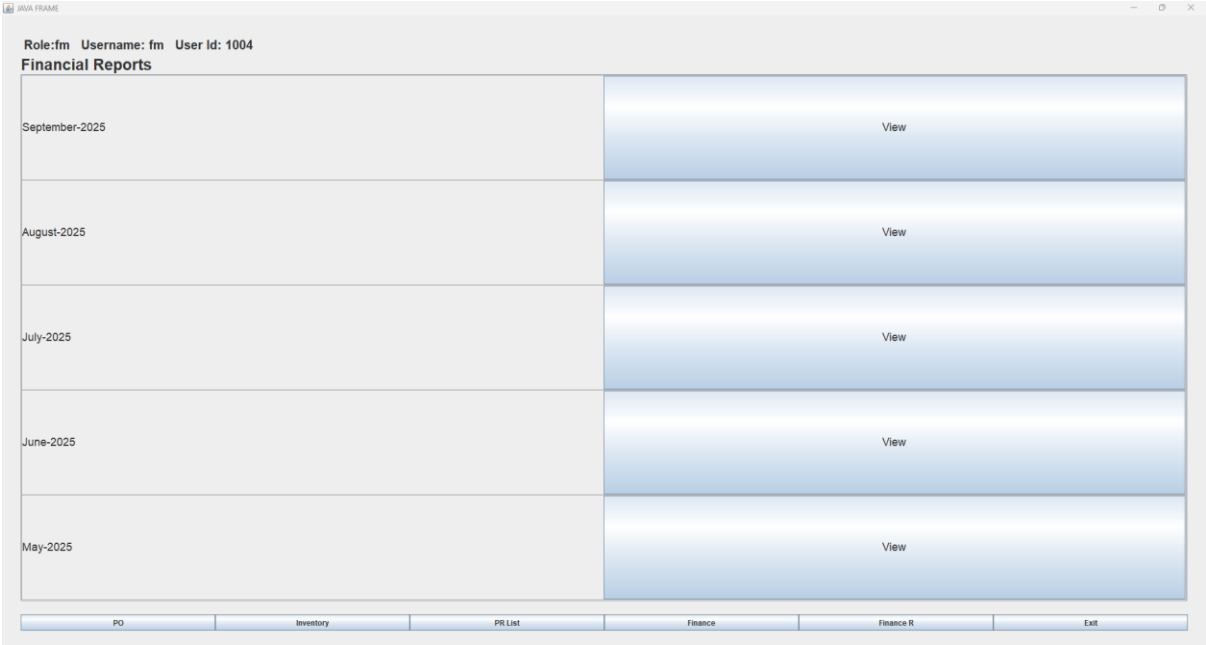


Figure 122: Finance Report Page

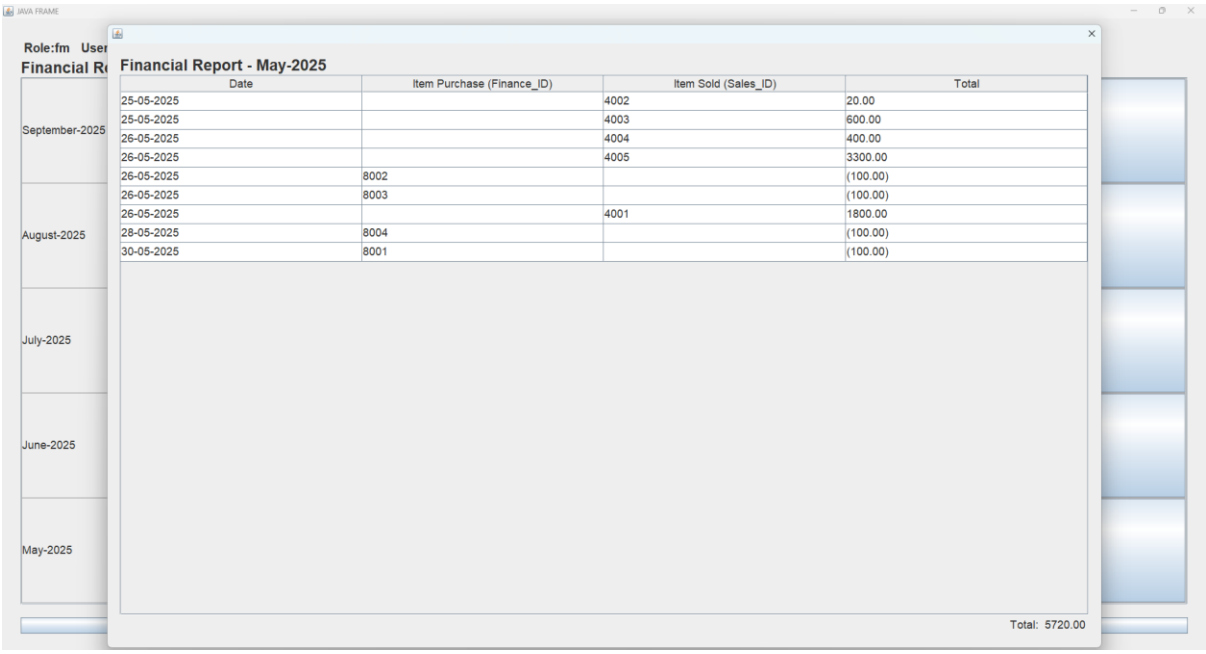


Figure 123: Details for Finance Report

This is "Financial Reports" page, it is accessible by finance manager only and at the top, it able to see the role, username and ID. This interface is centered around a panel that scrolls, automatically showing a list of months and years (e.g., "September-2025," "May-2025"). For each time period, user can see the month and year and there is a “View” button behind. These buttons are very important, because users can select a month and year and the detailed report

will appear instantly. The financial report will be displayed in a separate window, for a better info-rientation.

The FinanceReport displays windows show the important details of the report. It will automatically show what period is shown by the window's title: "Financial Report - [Month-Year]. It shows the financial data using a JTable with four columns: Date, Item Purchase (Finance_ID), Item Sold (Sales_ID) and Total. It is made for clarity since the font is clearly legible and the rows are all the same height. When user scroll down to the bottom of the table, a "Total" label and figure give a quick summary of the total amount for the period displayed.

Inventory Creation:

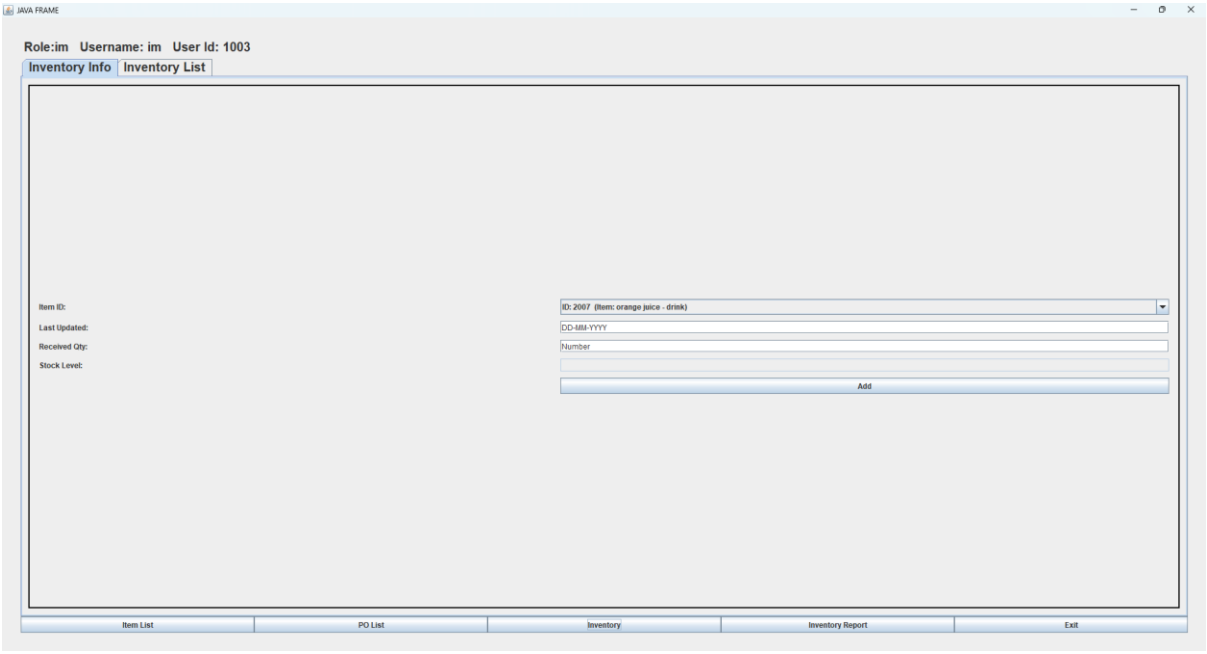


Figure 124: Inventory Info Page

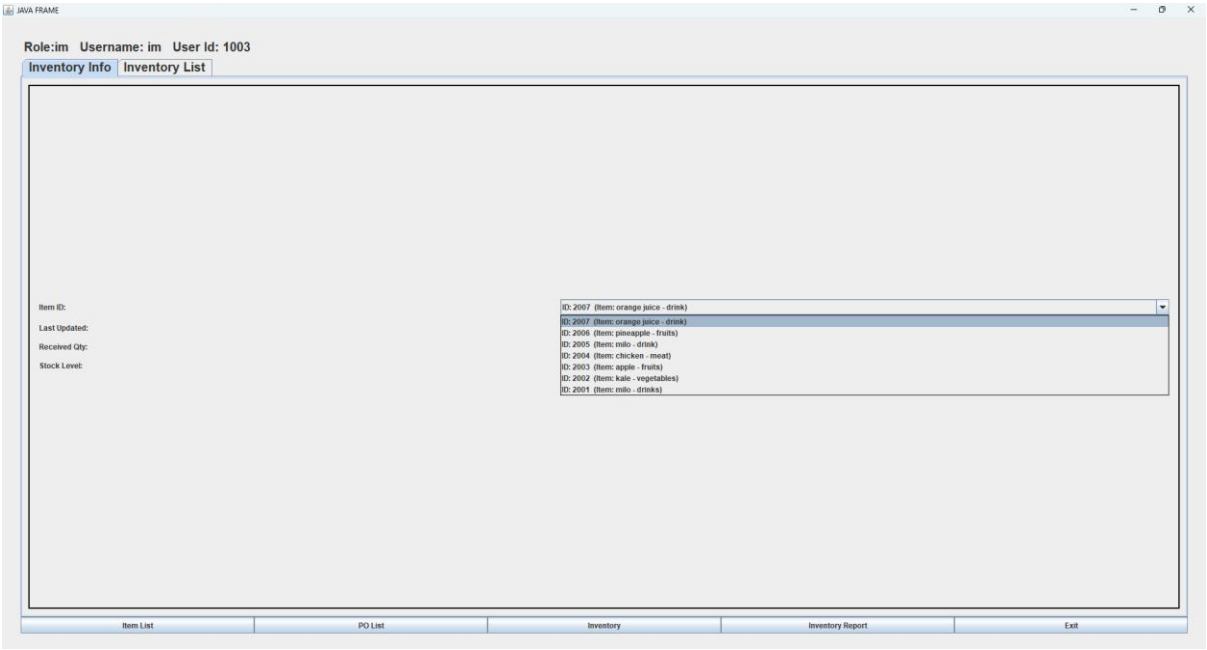


Figure 125: Dropbox for ItemID for Adding Inventory Info

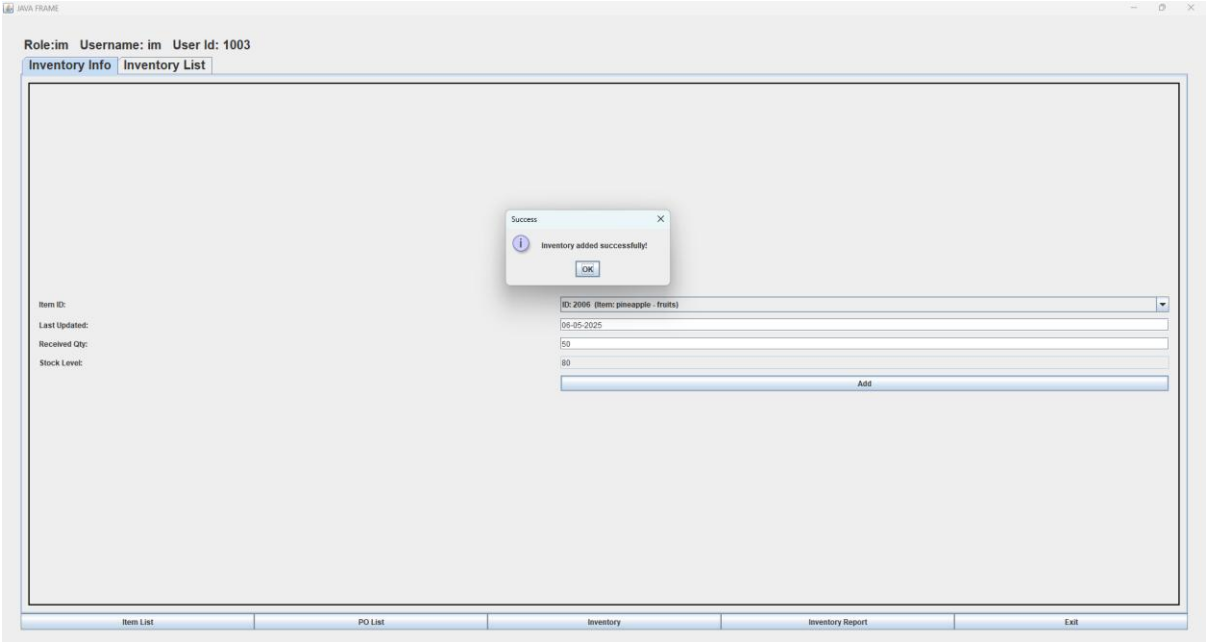


Figure 126: Successful Message after Adding Inventory Data

This “Inventory Info” page, it is accessible by inventory manager only. It able to add new inventory information. The user role ("Role: im"), username ("Username: im") and user ID ("User Id: 1003") are placed at the top of the frame, ready to guide the logged-in user. Just below that, two tabs, "Inventory Info" and "Inventory List," represent different functions or ways to use the inventory module and "Inventory Info" is selected at this time.

There are input fields for "Item ID" with a dropdown to choose from, "Last Updated" (in date format), "Received Qty," and "Stock Level". After all the details are entered, a user needs to click the “Add” button and the system pops up “Success” with the message “Inventory added successfully!”

Inventory Report:

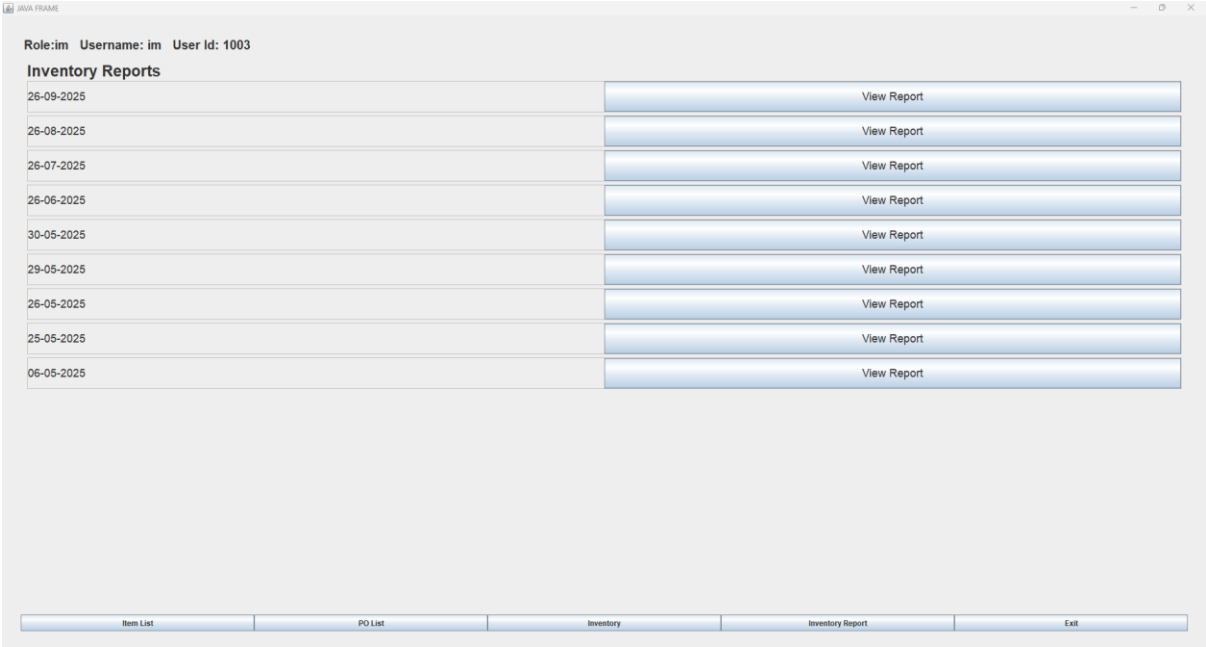


Figure 127: Inventory Report Page

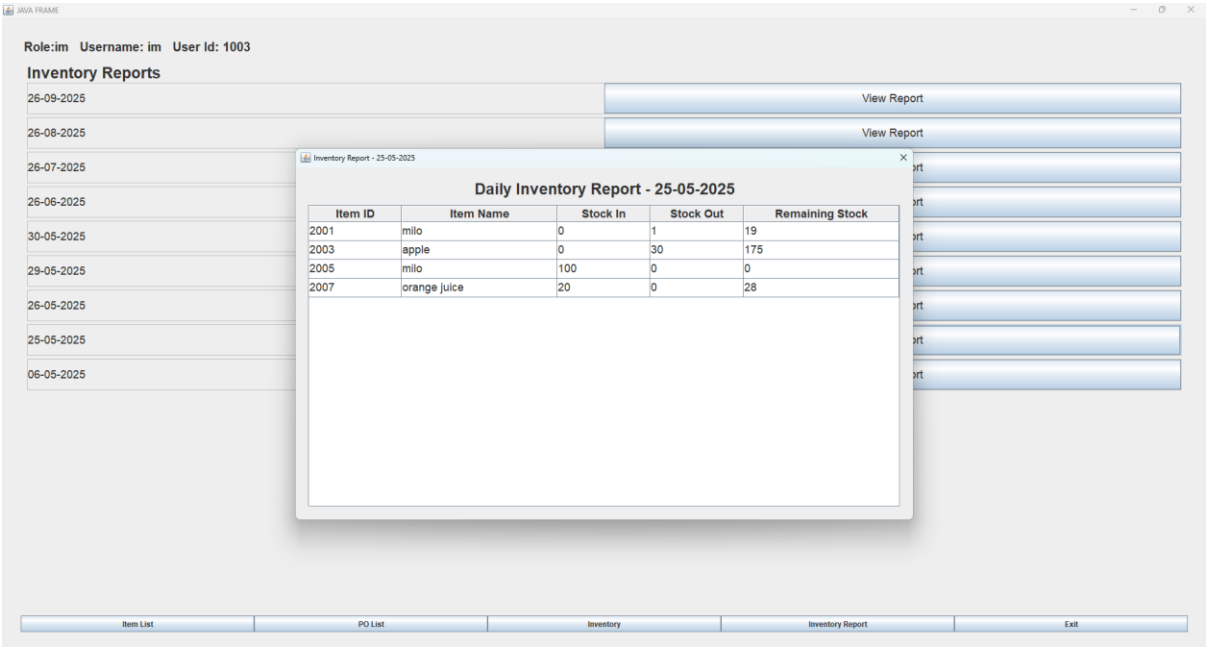


Figure 128: Inventory Report Details

This is “Inventory Reports”, it is accessible by inventory manager only. This page is formatted to show dates of past inventories and sales, plus a button that opens a report for each one. User can quickly check recent reports because the entries are displayed from newest to oldest with scrolling available. When user click the "View Report" button for a date, a new modal dialog appears called "Inventory Report - [Date Chosen]". This dialog contains a detailed table where

the first five columns (Item ID, Item Name, Stock In, Stock Out and Remaining Stock) cannot be edited, but the width of all columns and the row height can be adjusted for better readability and use.

Purchase Order Status:

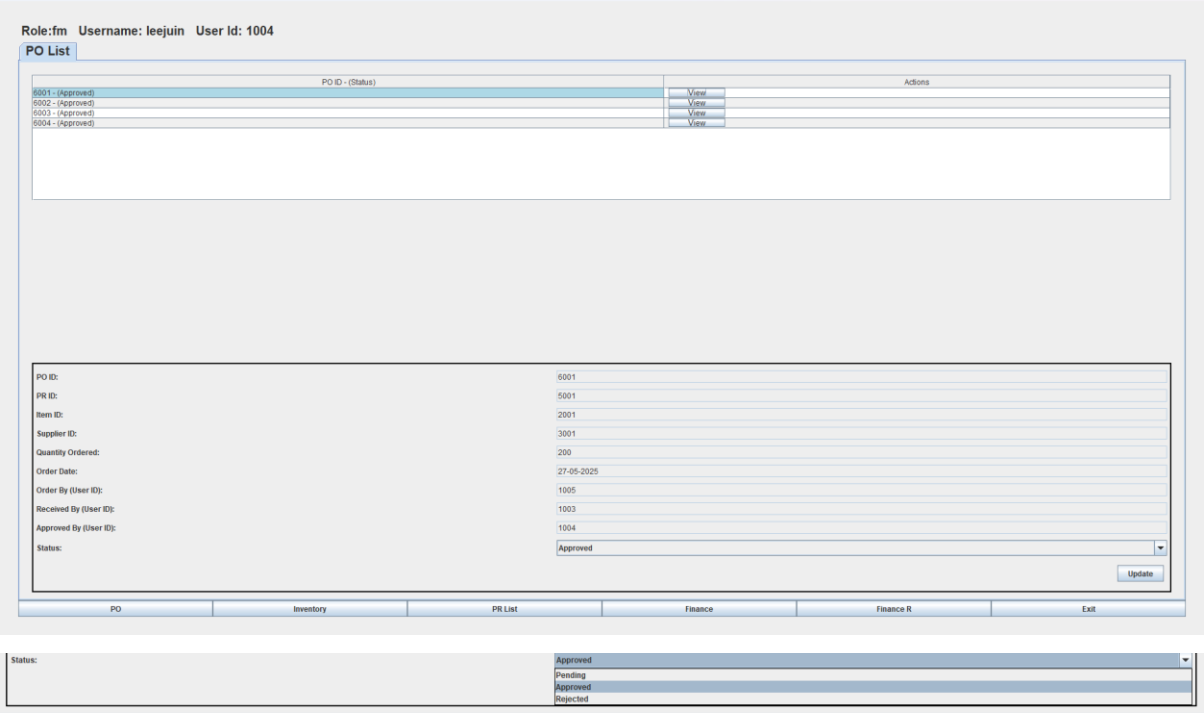


Figure 129: Purchase Order Status

This is the Purchase Order Status page. This page is used and accessible to only Finance Manager. In this page, the finance manager can view the status of purchase order and update its status to either approved or rejected. As shown in figure above, it also shows the role, username, and user id of the finance manager that is currently log onto. Furthermore, finance manager can view the purchase order list just by clicking on the view button and it will show details below. To indicate which purchase order is being viewed, the PO_ID will be highlighted in blue, giving the user a clear view. After choosing one of the purchase orders from the PO list, the finance manager will be able to change the status to either approve or rejected and click on the update button to confirm the changes. At the bottom of the page, there will be navigation bar so that the finance manager can change page conveniently. This page features a dropdown combobox which provides the user a list of choices (pending, approved, rejected) when clicked onto.

Inventory Status:

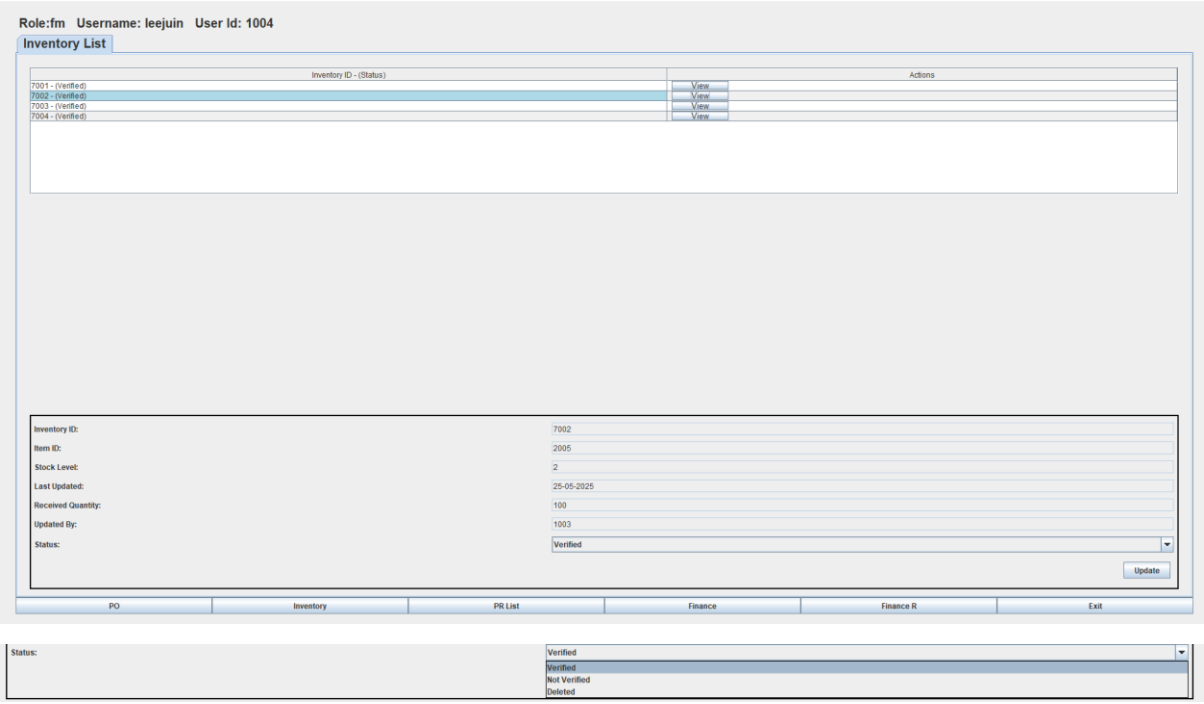
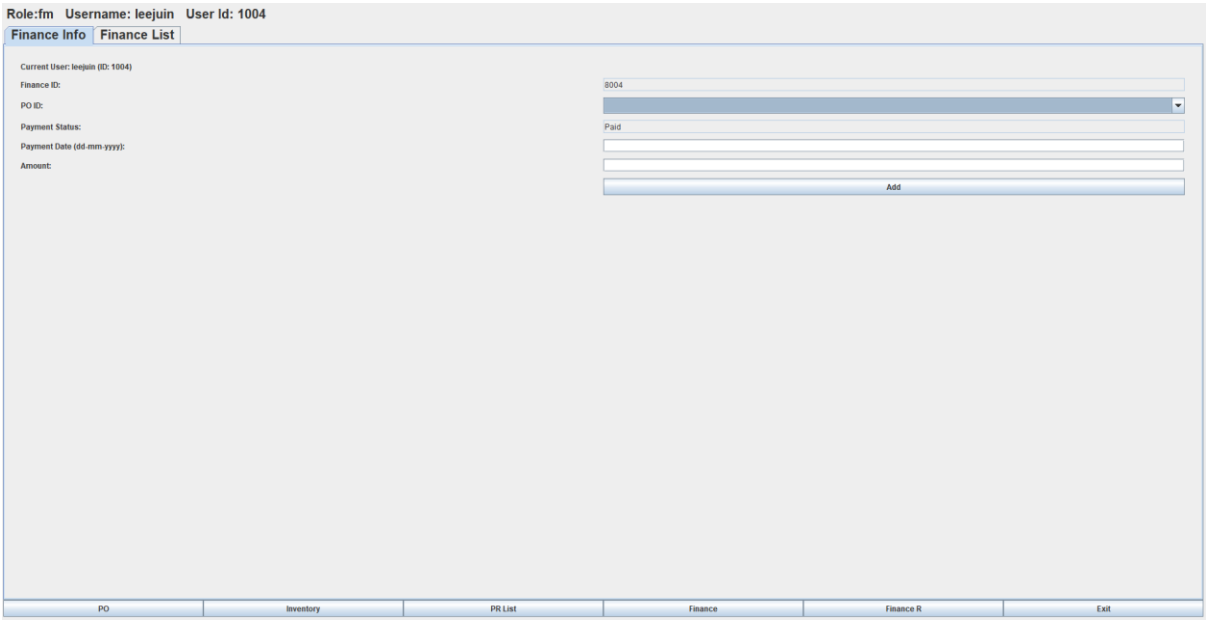
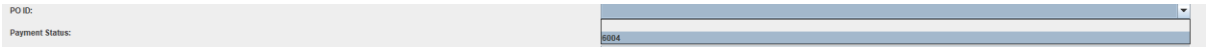


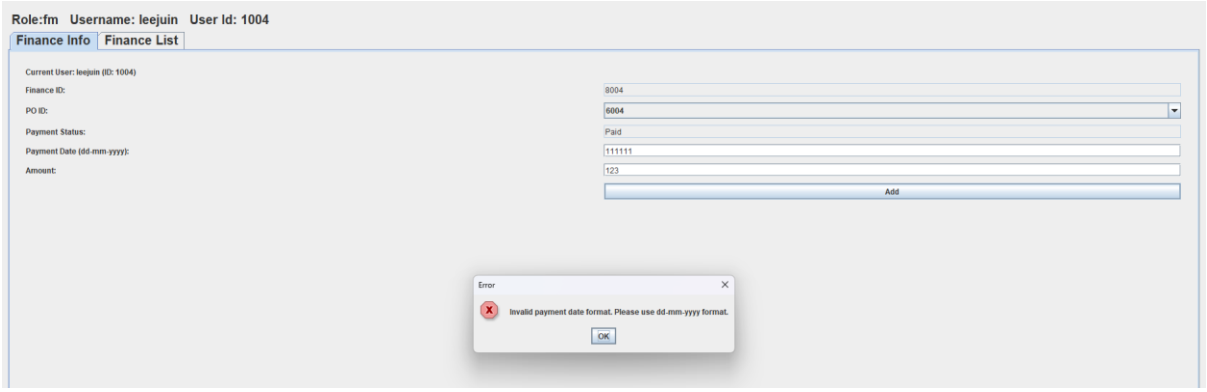
Figure 130: Inventory Status

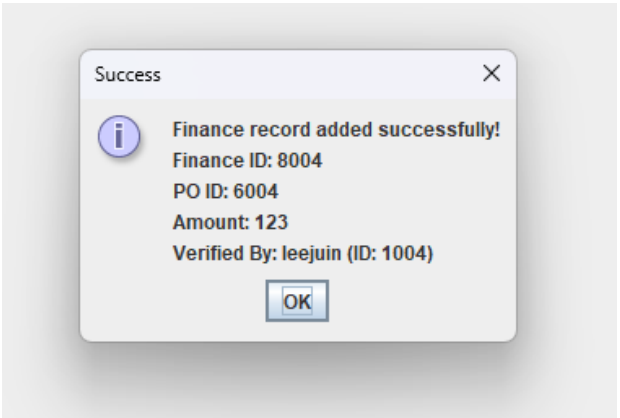
This is the inventory status page. Similarly, this page is only accessible by finance managers as they are required to verify inventory status and payments. At the top left of the page, it will also show the role, username and user id of the finance manager that is currently logged onto. At the bottom of the page, it will also have a navigation panel for the convenience of finance manager to access other pages. In this page, the finance manager will be able to click onto the view button to view one of the inventory list statuses and perform update accordingly. For example, currently the finance manager chose to view the inventory id 7002. At the bottom of the page, the system will then provide information about the inventory id, including inventory id, item id, stock quantity, last updated, received quantity, updated by, and status. After that, the finance manager will just need to update the status from “not verified” to either “verified” or “deleted”. After choosing the status to update, the user will then click on the update button to save the changes made. This is to mainly verify the inventory status so that once verified, the stock quantity will only be updated. This page also features a dropdown combobox which provides the user a list of choices (verified, not verified, deleted) when clicked onto.

Finance Creation:









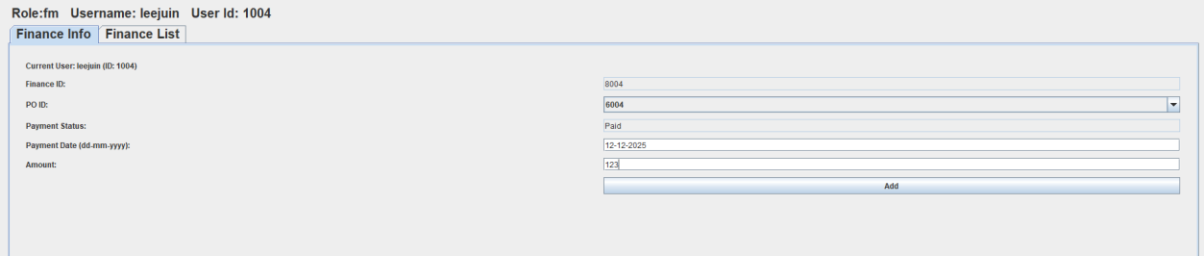


Figure 131: Finance Info

This is the finance info tab of finance creation page. This page is only accessible by finance manager. In this page, it allows the finance manager to create a finance statement which marks the chosen purchase order ID as paid. Once created, the chosen PO ID will be marked as paid in the finance list, verifying that the purchase order has been paid off. After choosing the PO ID, the payment status has been set to paid, and the finance manger will need to type in the payment date and amount. The user will need to click onto the add button to add the finance record. The additional feature that this page contains is the auto increment of Finance ID. The finance ID will then increase by 1 based on every finance record created. Furthermore, the payment date and amount have a validation rule that requires the user to only type the date in the format of ‘dd-mm-yyyy’ and the amount in only integer form. If the format is not correct, the system will then prompt the user about the invalid format and will tell them to change the input format. Once the format has been corrected, the user will then be able to add the finance record, and the system will prompt ‘Finance record added successfully!’. Lastly, it has also been set that the PO ID will only be displayed if it is marked as unpaid. Once finance record has been added and the corresponding PO ID has been marked as paid, it will not be displayed here under the PO ID combobox.

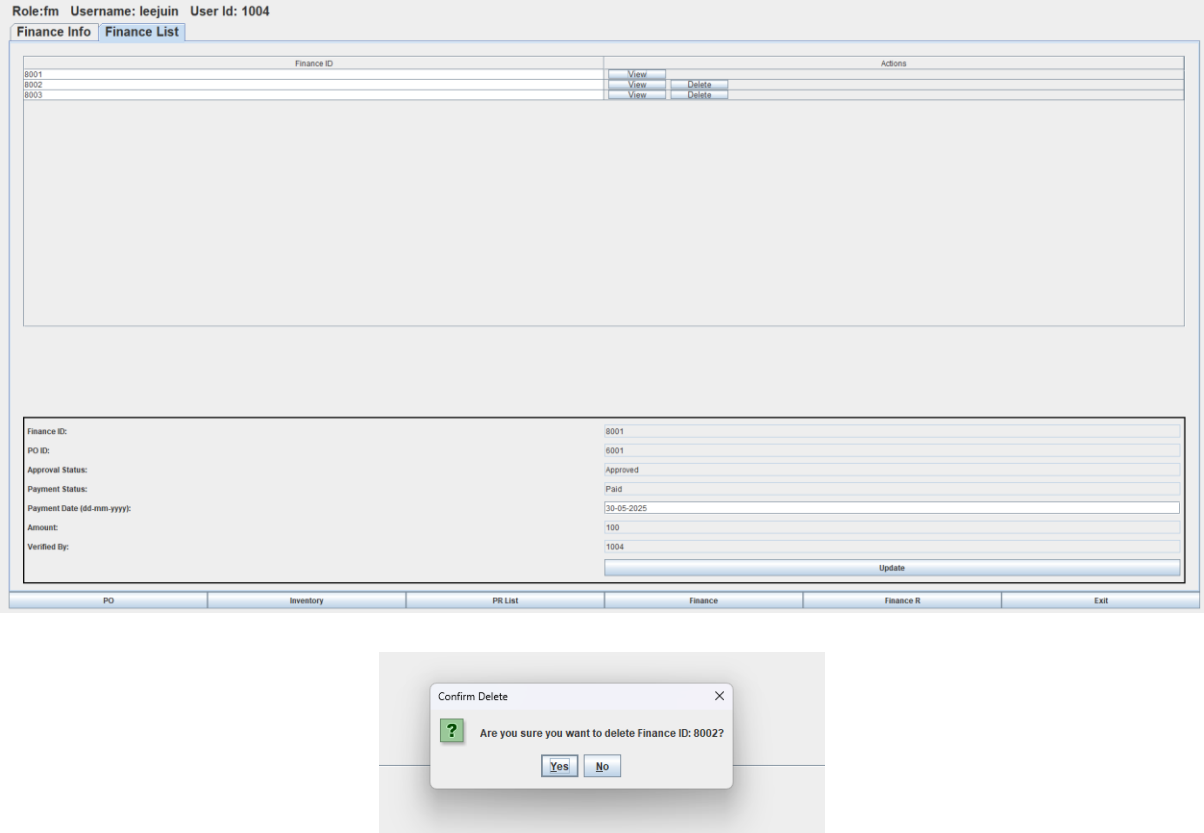


Figure 132: Finance List

This is the finance list tab of finance creation page. This page is only accessible by finance manager. This allows the finance manager to view, delete, and update the existing finance records. As shown in figure above, the finance manager can view one of the finance records, including its corresponding finance ID, PO ID, approval status, payment status, payment date, amount, and verified by. Furthermore, the finance manager can change the payment date of the chosen finance record by typing it in the text box. After that, the finance manager will then click on update button the save the changes made and the changes will be done to the chosen finance record accordingly. Lastly, the finance manager can choose to delete the finance record by clicking the delete button beside the view button of the corresponding finance record row. Once clicked, the system will prompt the user to confirm if the user wants to delete the finance record. Once clicked on yes, the system will then delete the finance record and changes will be made. However, the finance manager will not be able to delete the finance record that has already been approved by administrator. As shown in figure above, only finance ID 8001 cannot be deleted because it has been verified by the administrator. Additionally, the current finance manager user id that is logged on will be set as the user that has verified the finance record. For example, if the finance manager user id is 1005, once they create the finance record, the “verified by” row will show that 1005 has verified the finance record.

Page Name	Accessible Roles
Login	AM, PM, SM, IM, FM
Dashboard of AM PM SM IM FM	AM, PM, SM, IM, FM
User Creation	AM
Finance Status	AM
PR status	PM
Supplier Creation	AM, SM
Item Creation	AM, SM
Sales Creation	AM, SM
Supplier List	PM
Purchase Request	AM, SM
Purchase Order	AM, PM
Purchase Request List	FM
Purchase Order List	IM
Item List	IM
Inventory Status	IM
Finance Report	FM
Inventory Creation	IM
Inventory Report	IM
Purchase Order Status	PM
Inventory Status	IM
Finance Creation	FM

6.0 Additional Features

Role-Based Credential Validation at Login

Figure 133: Additional Features Login

Role-based verification is put in place to ensure authorized login and stop any unauthorized users. Users have to choose their role from a menu and give their correct username and password. For example, if a user tries to login as Finance Manager with Administrator credentials, the system will stop the login and display a “Invalid Credentials” message. Because of this feature, only the pages and data needed by a user based on their role are visible to them which helps preserve the system’s integrity and privacy.

Formatted Input Validation for Password, Phone, and Email

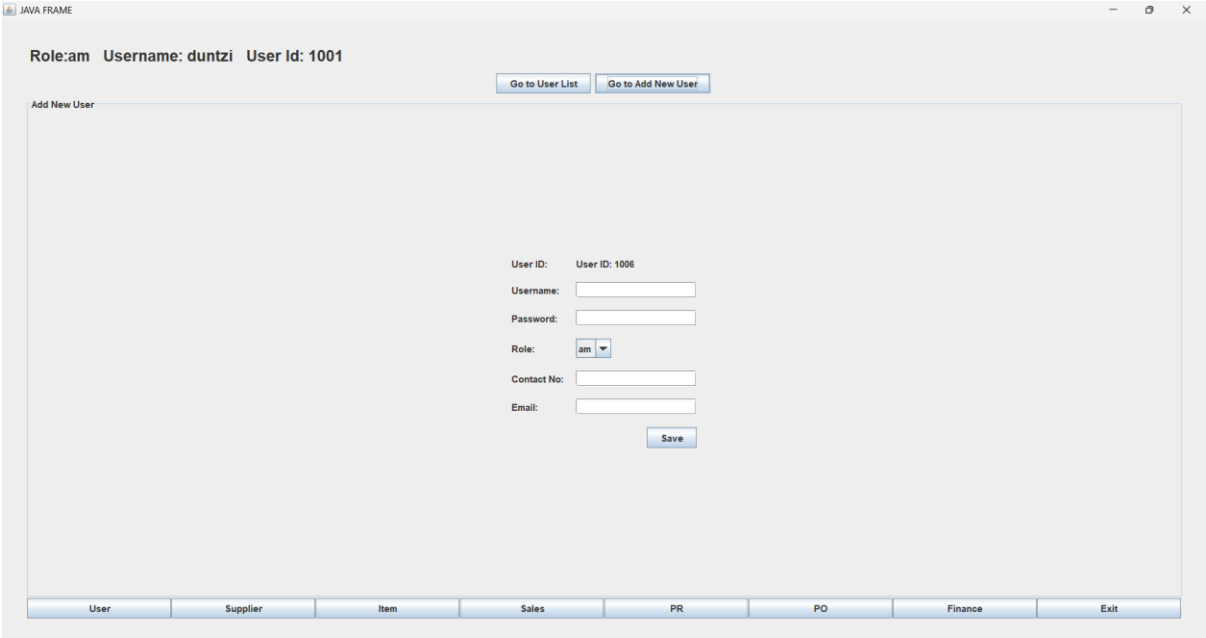


Figure 134: Additional Features Add User

For reliable and secure user information, careful validation rules were put in place during registration and editing. A password must be exactly 8 characters, with six digits and two letters, to guarantee it is not too easy to guess. Contact numbers should contain numbers only and be between 10 and 11 characters in length and all email addresses should have "@gmail.com" to uniform communication formats. Depending on the criteria, the program will give a relevant error alert, prompting the administrator to fix the entered data. These validations stop users from adding inaccurate or unsecure data.

Edit Functionality for User Details

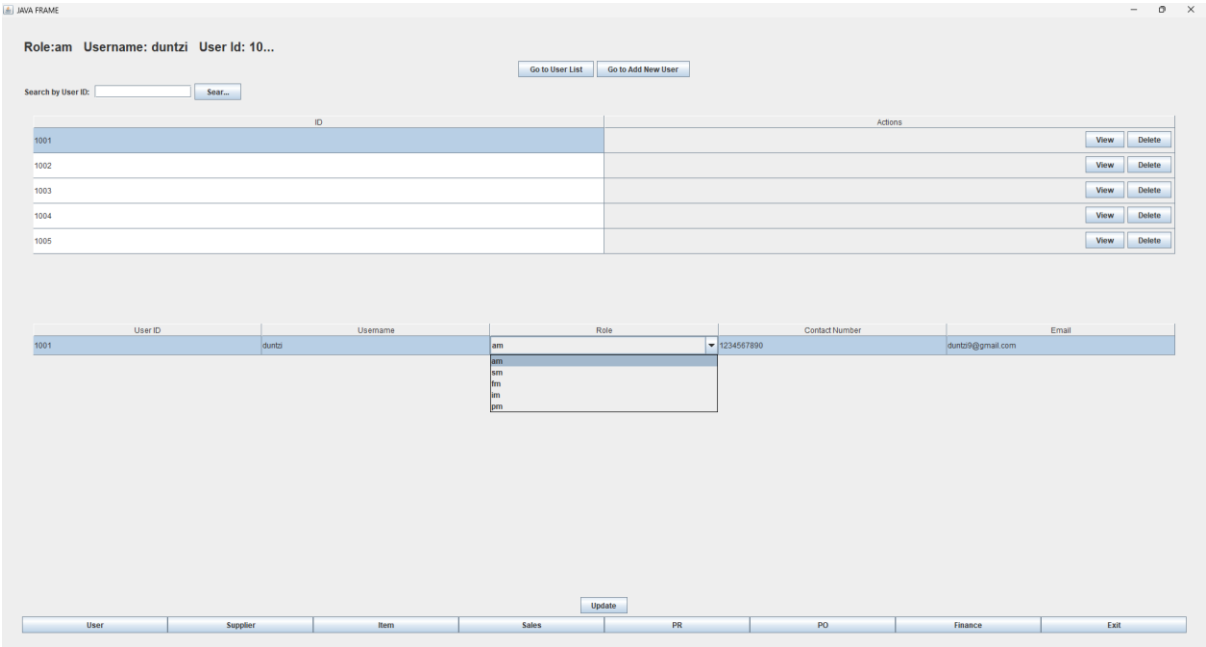


Figure 135: Additional Features User Update

As well as allowing registration and user removal, the system lets the Administrator look at and change any user data. When a user profile is chosen, users can access fields for editing the username, role, contact number and email. The system helps administrators make updates or changes without having to redo the creation of user accounts. All changes are checked using the same format rules as the registration, keeping data unchanged. It makes managing users more efficient.

Search Bar for Filtering Records

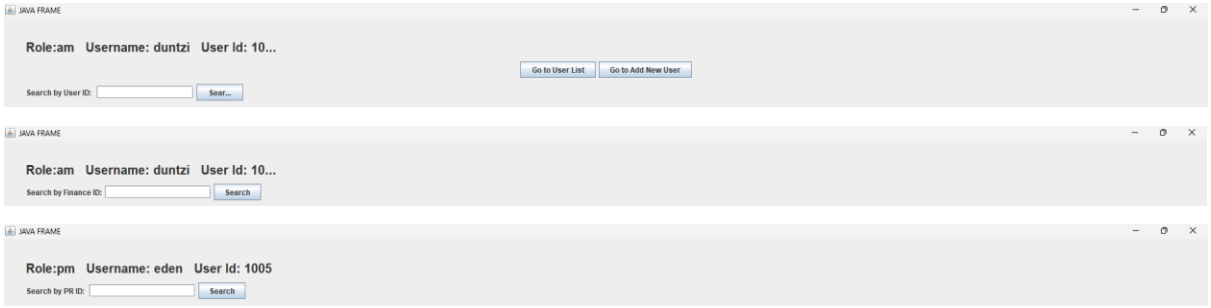


Figure 136: Additional Features Search Bar

The search bar has been put on the top-left corner of key pages including User List, Finance Status and Purchase Request Status. A user can enter the User ID, Finance ID or PR ID and the search bar will return the related record quickly for the respective manager or administrator. This helps a lot in systems that include many entries, since it means user do not have to scroll through each entry manually. It lets users quickly and easily reach the specific details they are looking for.

Confirmation Prompts Before Deletion

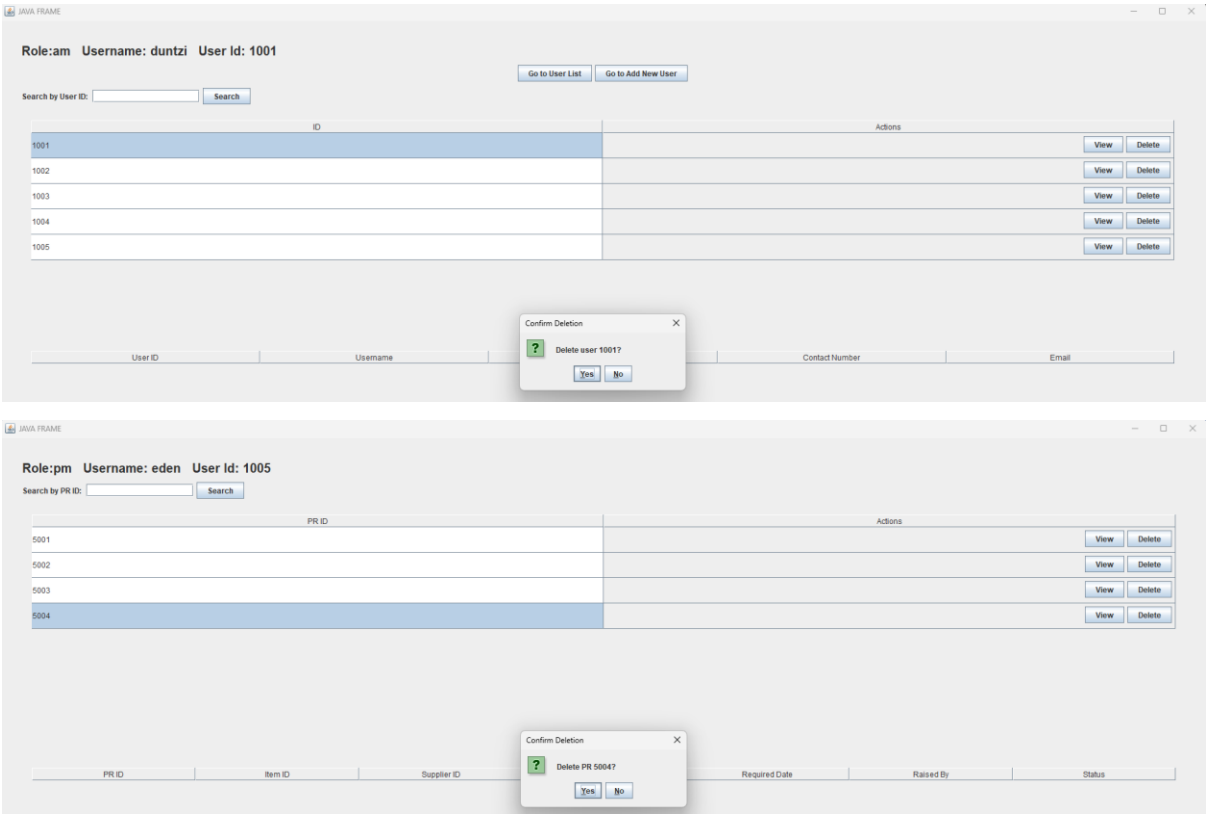
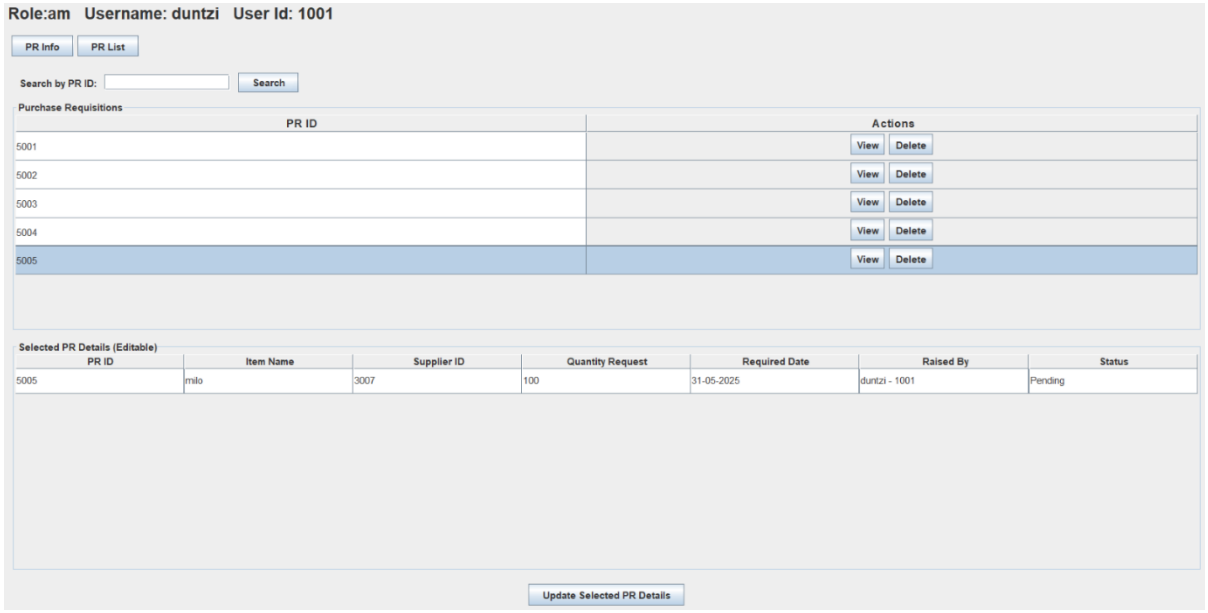
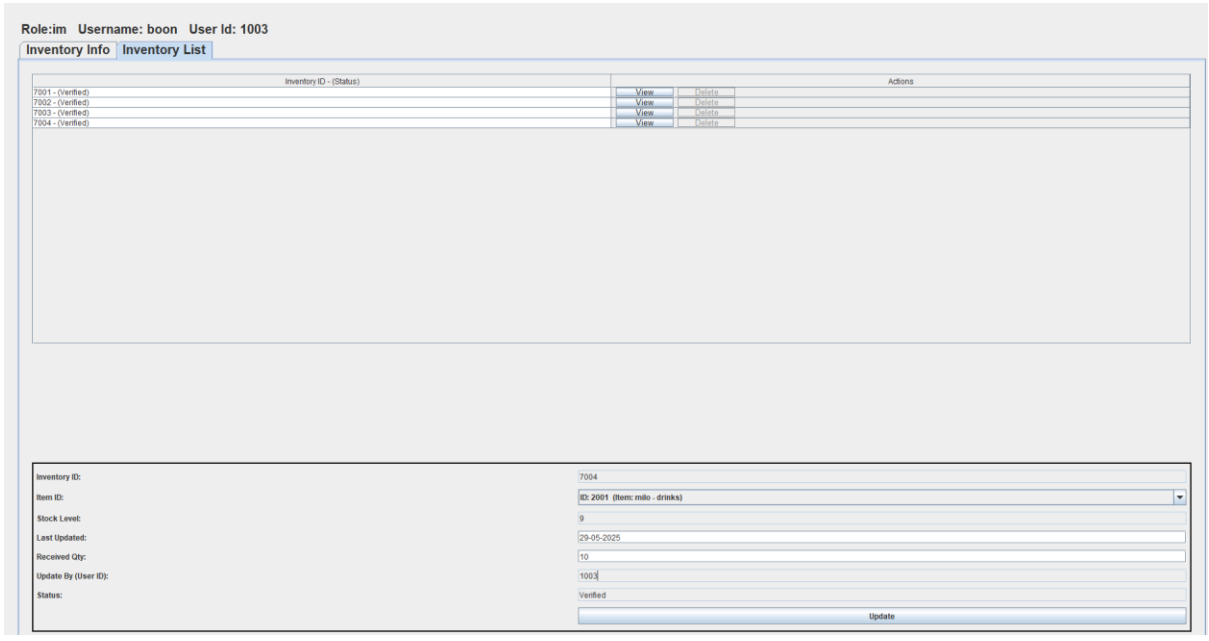


Figure 137: Additional Features Confirmation Prompts

Deleting anything in the system will ask for confirmation before it is removed. A warning message appears asking the user to confirm the decision before any delete or removal happens. Because of this, accidental deletions due to errors are avoided and user accountability goes up. Avoiding losing important data requires using the proper prompts which helps maintain document integrity.

Auto Detect User





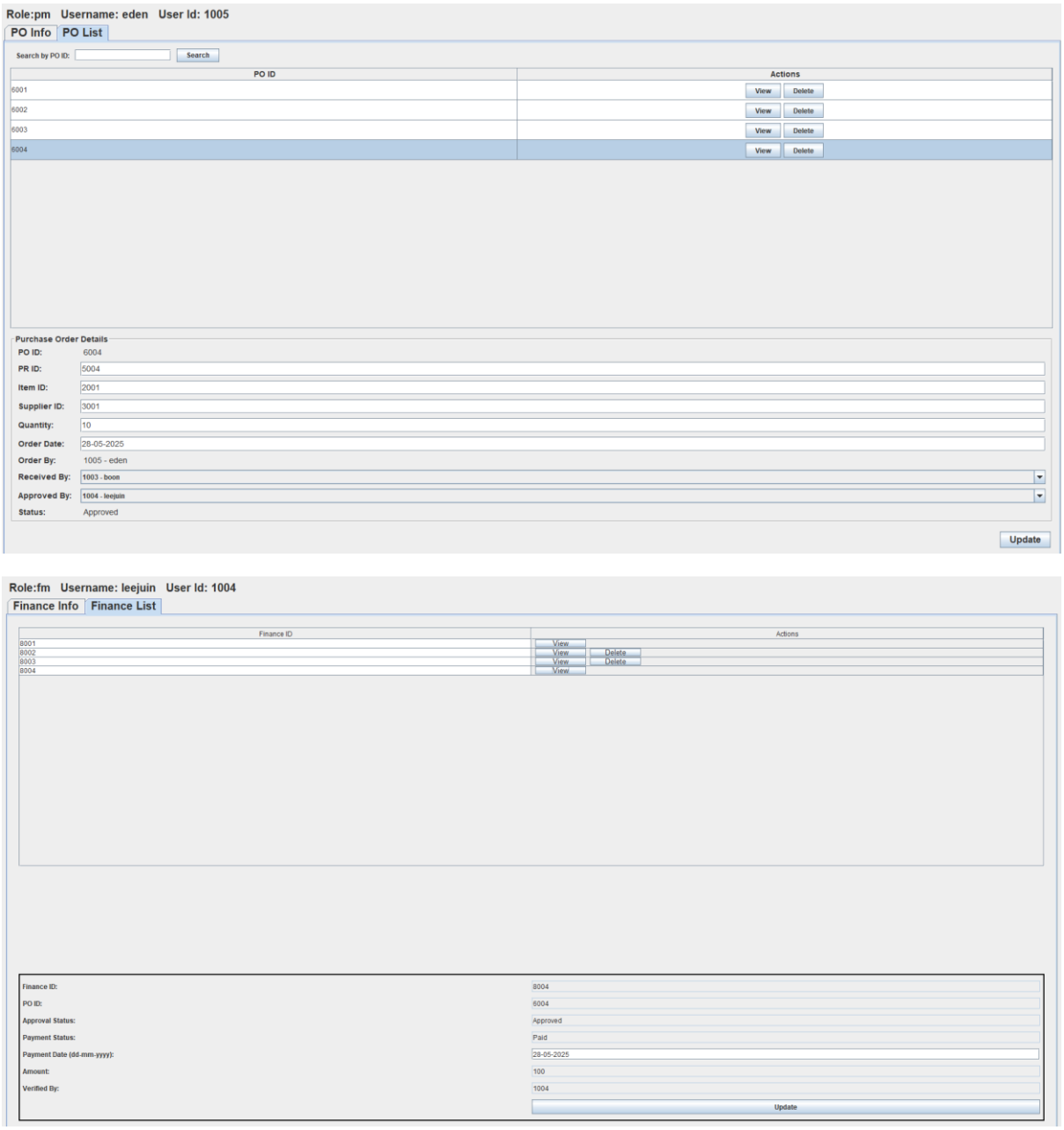


Figure 138: Additional Features Auto detect user

The “Raised By”, “Order By”, “Update By”, “Verified By”, will automatically detect the user id who raised the Purchase Requisition, Purchase Order, Inventory Creation, Finance Creation.

Finance Transaction Approval System for Administrators

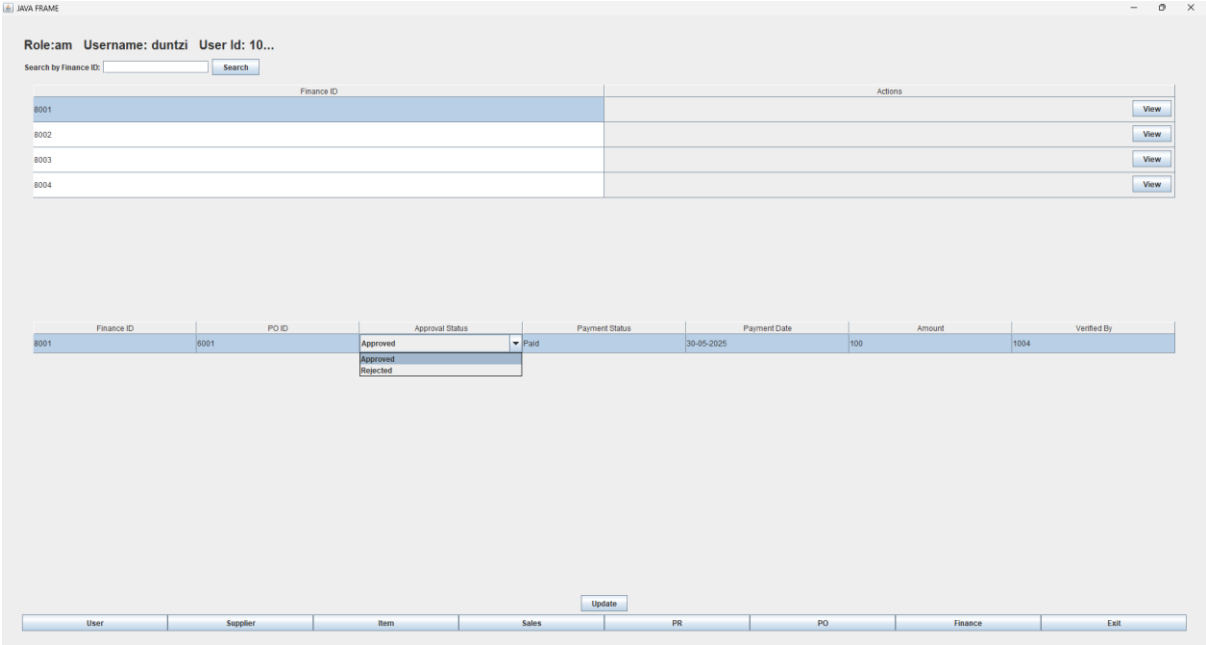


Figure 139: Additional Features Finance Status

An extra check was set up on the Finance Status page, so the Administrator can review transactions started by the Finance Manager. This page helps the Administrator check each transaction and mark it as approved or not. This is not necessary for the original assignment but gives the company extra internal control. Before the financial transaction is complete, it is examined by someone else which increases transparency and helps prevent errors or fraud.

Mandatory Field Validation

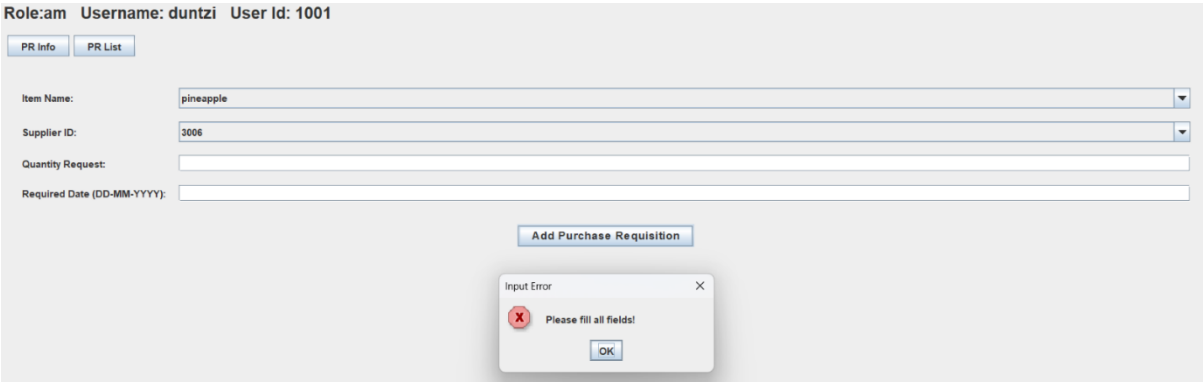


Figure 140: Ensure data filled in feature

Administrators/Sales Managers should fill in the all the information, if they did not fill in, after clicking the “Add Purchase Requisition” it will show a message telling “Please fill all fields!”. So, this feature ensures that data record correctly and prevent human error.

PR ID Search Feature

Figure 141: Search Feature

Administrators/Sales Managers can search the Purchase Requisition by typing the PR ID, when there are thousands of Purchase Requisition this search feature will be very helpful and reduce time wasting for user.

Approved PR Deletion Prevention

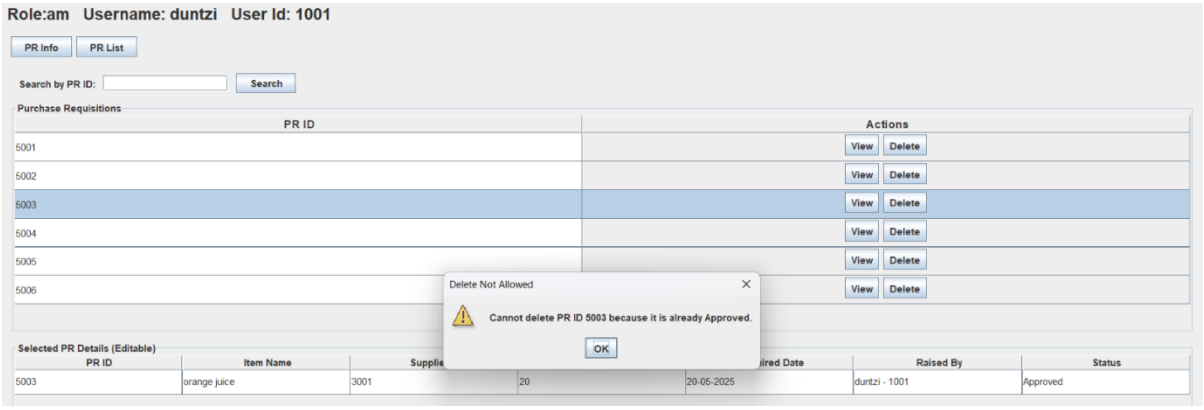


Figure 142: Prevent Human Error Delete

If the Purchase Requisition status is “Approved”, Administrators/Sales Managers will be unable to delete, this feature can prevent accidently delete the Purchase Requisition.

Auto-Fill for PR Selection

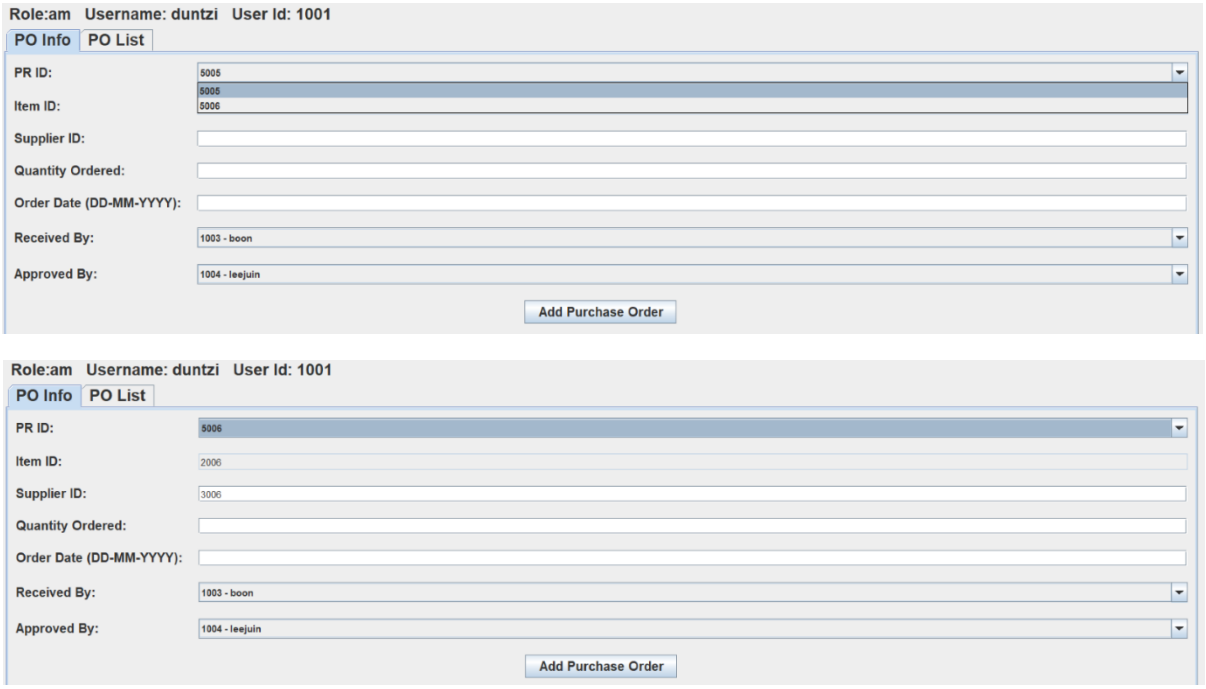


Figure 143: Auto Fill In Feature

After Administrators/Purchase Managers selected the PR ID, the Item ID and Supplier ID will automatically fill in, this can make sure the data accurately, prevent human error, and reduce time wasting.

Supplier ID Mismatch Detection

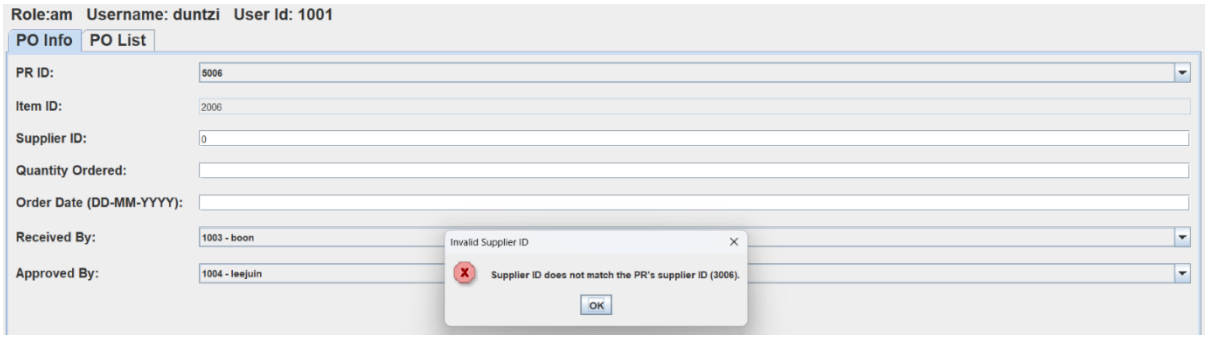


Figure 144: Error detection

If Administrators/Purchase Managers entered an impossible data, the system will be detected and show a message “Supplier ID does not match PR’s supplier ID”, so this feature ensures that all information entered by user is correctly.

PO ID Search Feature

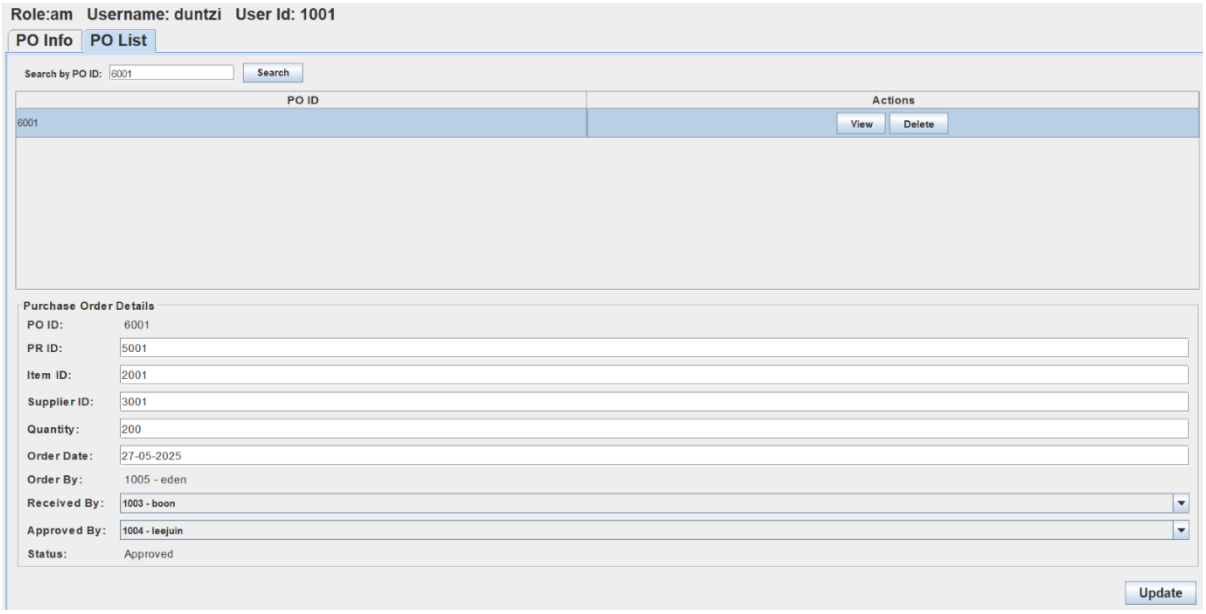


Figure 145: Search feature

Administrators/Purchase Managers can search the Purchase Order by typing the PO ID, when there are thousands of Purchase Order this search feature will be very helpful and reduce time wasting.

Approved PO Deletion Prevention

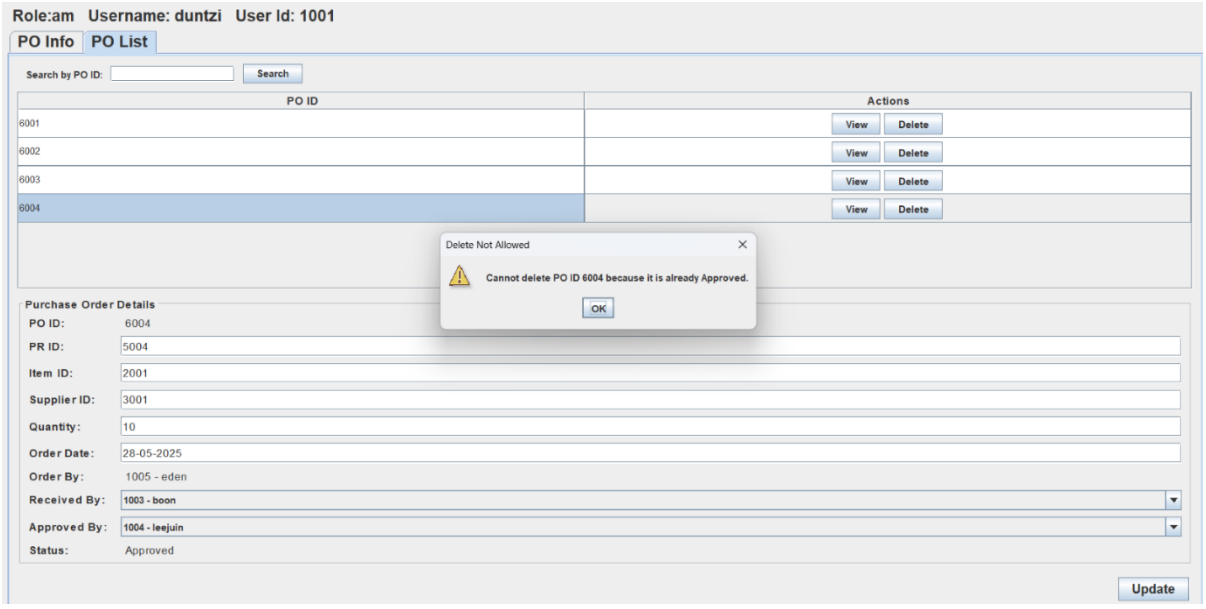


Figure 146: Prevent human error delete

If the Purchase Order status is “Approved”, Administrators/Purchase Managers will be unable to delete, this feature can prevent accidently delete the Purchase Order.

Dynamic Keyword Search

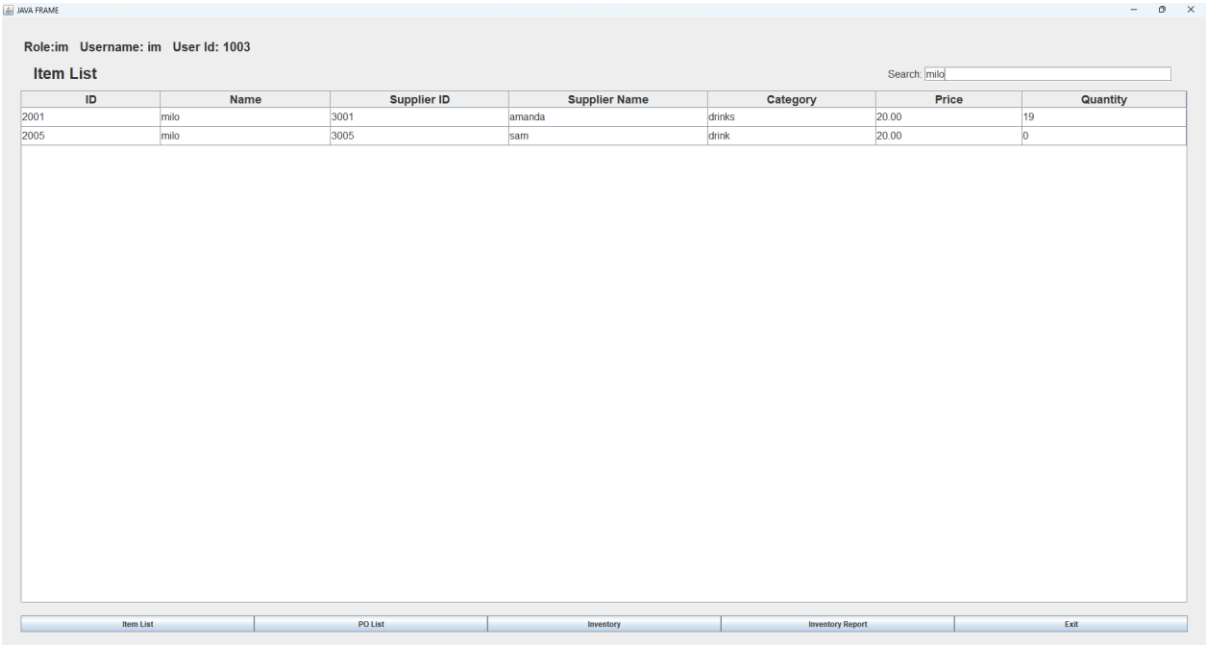


Figure 147: Search Bar Showing "milo"

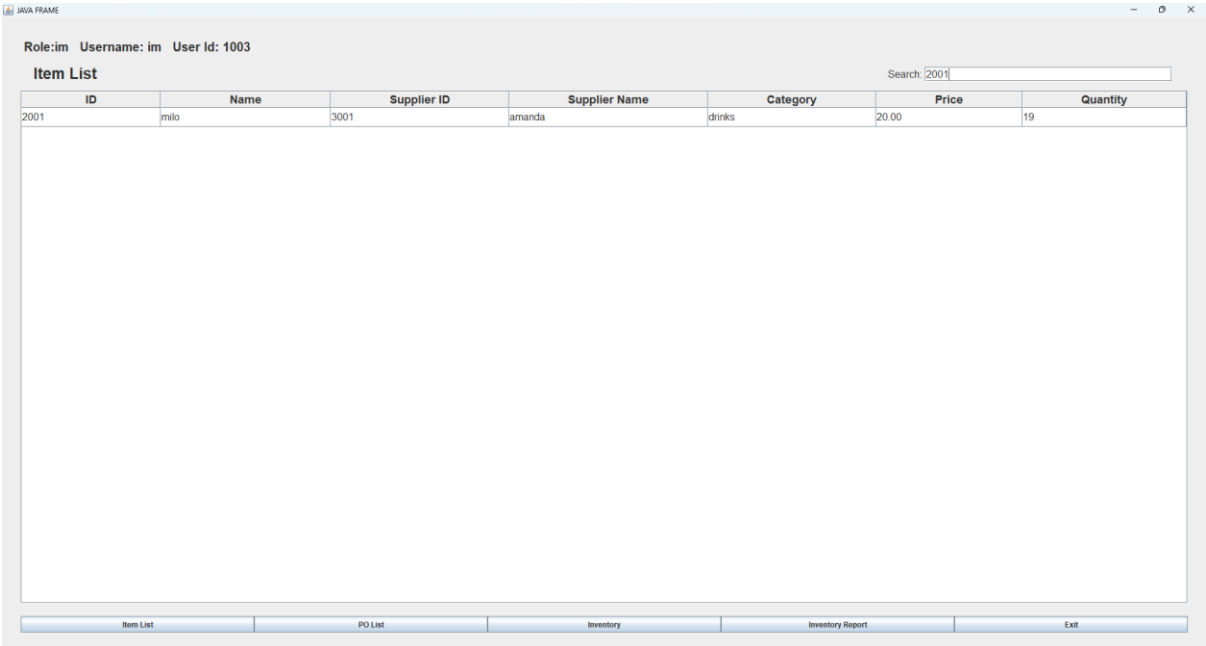


Figure 148: Search Bar Showing "2001"

Search Bar Functionality is implemented. "Item List", "PO List" and "Purchase Requisition List" display a "Search:" button at the top-right. Users can use keywords in the search box to quickly filter the data on the table and find records they need faster. Example give, if search "milo" in the "Item List"; this will bring up a table of items that have "milo" in their names. Giving a similar example, trying "2001" as a query will get items with the same ID.

Validation Rule in finance creation

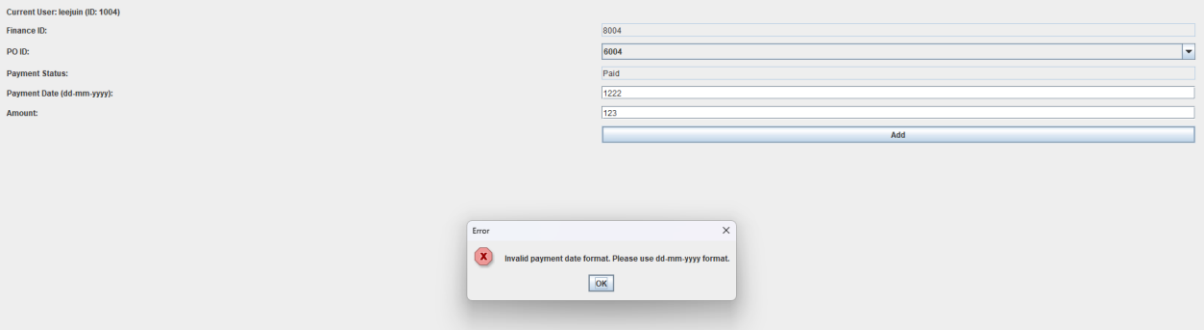


Figure 149: Finance Creation Validation Rule

In the finance info tab of finance creation page, a validation rule has been set for payment date and amount. The payment date format is required to be in the form of dd-mm-yyyy, else it will prompt error message to the user. Other than that, the amount format is set to the form of integer, meaning that the user can only type numerical values as input. These validation rules have been set to ensure data consistency and accuracy, preventing confusion due to different format when reviewing the finance records.

PO ID display based on Payment Status

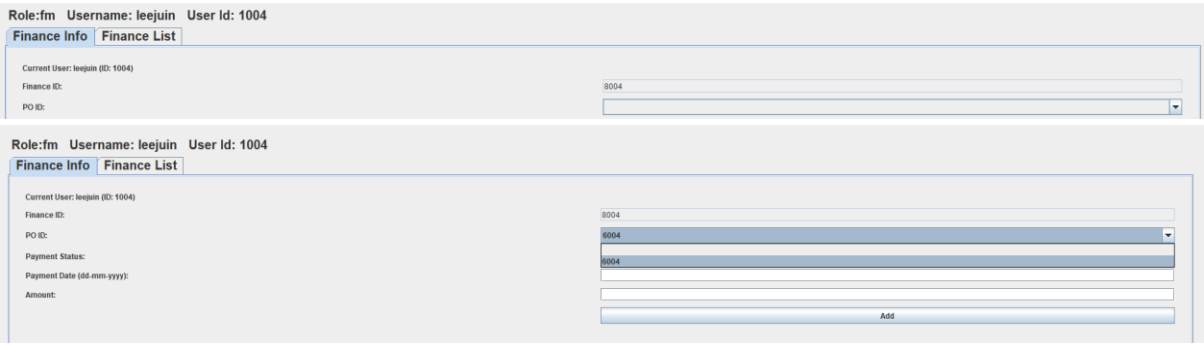


Figure 150: Relation between PO ID with Payment Status

In the finance info tab of finance creation page, the additional feature here is that the PO ID will be displayed based on the payment status of the existing PO ID. If the payment status is unpaid, it will be displayed in the PO ID list, so that the finance manager can select the unpaid PO ID and check. On the other hand, if the payment status of the corresponding PO ID is paid, it will not be displayed on the list of PO ID.

7.0 Limitation

Problem 1 (No real time code syncing)

As member are working on coding for each different positions and profiles, the code needs to be shared together to make sure that the code integrates with each other. However, we are not able to integrate the code together as each code files are done on different machines.

Solution 1

To tackle the issue of not being able to sync the codes, all the members have downloaded github desktop, which allows the user to push and pull their code to the cloud. Then other users will be able to pull the code from the cloud for modifications. Once modifications are done, the latest version will be pushed to github, therefore the other users will be able to see the changes made and use the latest files for editing. This lets everyone in the group to sync their codes together and make sure that every java code file can integrate with each other.

Problem 2 (No backup or code recovery)

When typing code in visual studio code, sometimes there are bugs and the code did not save, forcing us to waste time troubleshoot and try to type back the same code. Other than that, one of our group mate accidentally delete the java file, making it an issue for us as we need to type back the code again.

Solution 2

Similar to the solution of problem 1, we save and upload our code onto github desktop. After we type our code, we push the latest code onto github to save it. Then if there are cases of corruption, we can refer back to github desktop for the previous history of code, and we can pull back the code onto visual studio code. And if the java file are deleted accidentally, we will refer back to the change history in github desktop to recover back the original code.

8.0 Conclusion

By using OOP, system had successfully created an integrated procurement and inventory system for Omega Wholesale Sdn Bhd (OWSB). It was formulated to imitate real business activities such as purchasing request, generating orders, tracking inventory, keeping track of sales and preparing financial statements. Because of good organization and the use of modules, encapsulation, abstraction and class structure, the application becomes easy to maintain and able to grow as needed.

Roles are set up for users in the system which means there are interfaces for Admins, Sales Managers, Purchase Managers, Inventory Managers and Finance Managers. Every role has particular permissions and tools in the application which makes it more secure and helps organize tasks. The user interface was created with Java Swing which made it easy to use. Data was stored in text files which made accessing and saving information easier while the game was being developed.

Overall, the project mainly shows close collaboration, proper use of Java and good problem-solving while developing a desktop system with multiple functions. It gives a good base for upcoming advancements such as using databases or converting to a web version.