
# ASSIGNMENT

## TECHNOLOGY PARK MALAYSIA

## CT038-3-2-OODJ

## OBJECT ORIENTED DEVELOPMENT WITH JAVA

## APD2F2502CS(DA)

## GROUP ASSIGNMENT

**HAND OUT DATE: 24 MARCH 2025**

**HAND IN DATE: 3 JUNE 2025**

## GROUP 33 MEMBERS:

| | |
|---|---|
| OOI WEI LUN (Leader) | TP070209 |
| WONG SHI HONG | TP071359 |
| PHUAH KAI WEN | TP071427 |
| LAY PEAK QI | TP070028 |
| YAP YANG MIN | TP069866 |

# Table of Contents

# 1. Introduction

Omega Wholesale Sdn Bhd (OWSB) is a fast-growing wholesaler based in Kuala Lumpur that specializes in supplying groceries, fresh produce, and other goods to retailers across Malaysia. As the business starts to expand, the existing manual procedures have become increasingly difficult and inefficient to manage. OWSB realized that it needs to automate the Purchase Order Management to increase efficiency and accuracy. Without the implementation of an automated Purchase Order Management System, OWSB will face several challenges as manual procedures start to get more inefficient and prone to error.

Firstly, entering the data manually is very time-consuming and it will increase the chances of having errors in quantities, item codes, delivery dates, and so on, often leading to anomalies. Furthermore, manual documentation is challenging when it comes to generating reports or supporting clear audit trails for financial records with a paper-based or spreadsheet-driven system like Excel and Google Sheets. Additionally, without a centralized platform, the coordination between departments often breaks down, leading to miscommunications and redundant work.

To overcome these challenges, Omega Wholesale Sdn. Bhd. (OWSB) has implemented an automated Purchase Order Management System. This system will serve as the centralized platform to improve the procurement process. The key roles for this system include Sales Manager, Purchase Manager, Finance Manager, Inventory Manager and Admin. Sales Manager uses the system to generate Purchase Requisitions (PR) only when the items are below the reorder level. Purchase Manager will then receive and approve those PR requests and generate Purchase Orders, which will then be verified and approved by the Finance Manager who also processes the payments to the supplier.

Inventory Manager manages stock levels by updating inventory after receiving goods and tracks low-stock alerts. Finally, the admin has full control of the entire system by managing user accounts and ensures every operation runs smoothly. The main functions for this system include a user-based access system, automated document generation, real-time stock status tracking, and many more. Additionally, it also provides financial, and stock report features to support informed decision-making with relevant information.

Thus, this new system will help OWSB to minimize human errors, improve internal communication and massively boost the efficiency and the effectiveness of the system.

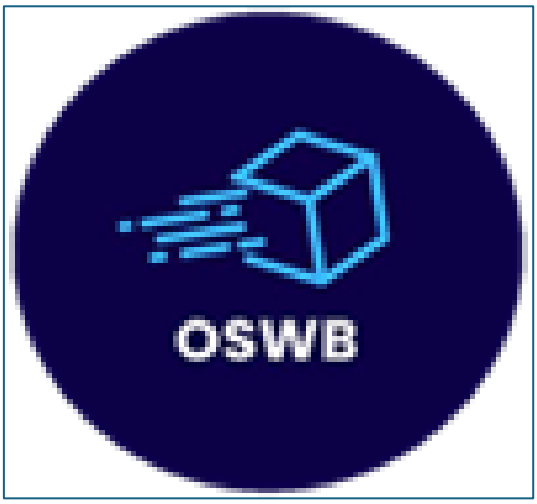
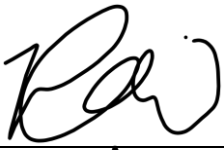


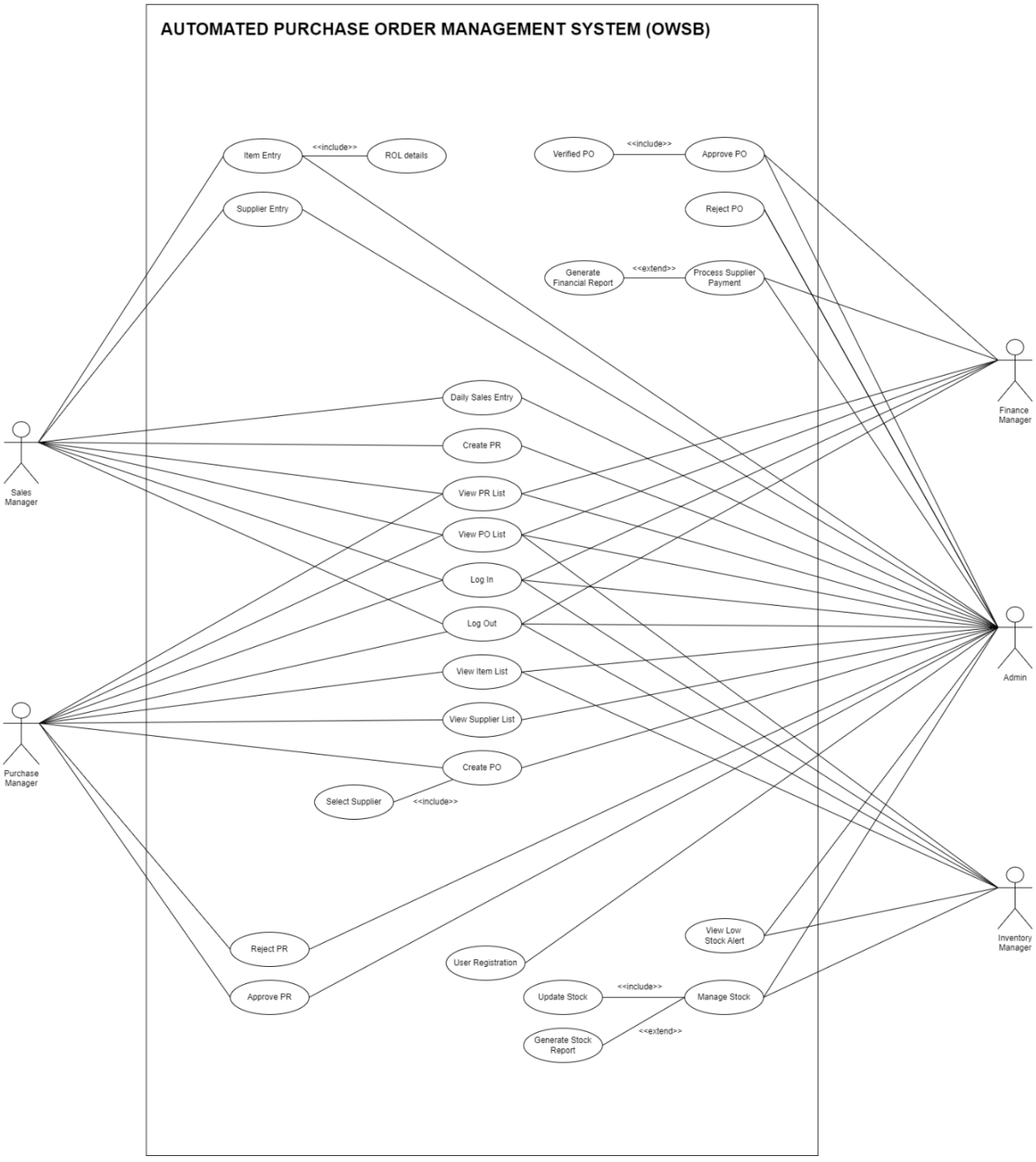

**Figure 1.0: Logo of OSWB**

# 2. Workload Matrix

| Team Member | Contribution % | Signature |
|---|---|---|
| Ooi Wei Lun (TP070209) | - Involved in Diagram, Coding, and Documentation part<br>- **22%** |  |
| Wong Shi Hong (TP071359) | - Involved in Diagram, Coding, and Documentation part<br>- **20%** |  |
| Phuah Kai Wen (TP071427) | - Involved in Diagram, Coding, and Documentation part<br>- **20%** |  |
| Lay Peak Qi (TP070028) | - Involved in Diagram, Coding, and Documentation part<br>- **20%** |  |
| Yap Yang Min (TP069866) | - Involved in Diagram, Coding, and Documentation part<br>- **18%** |  |

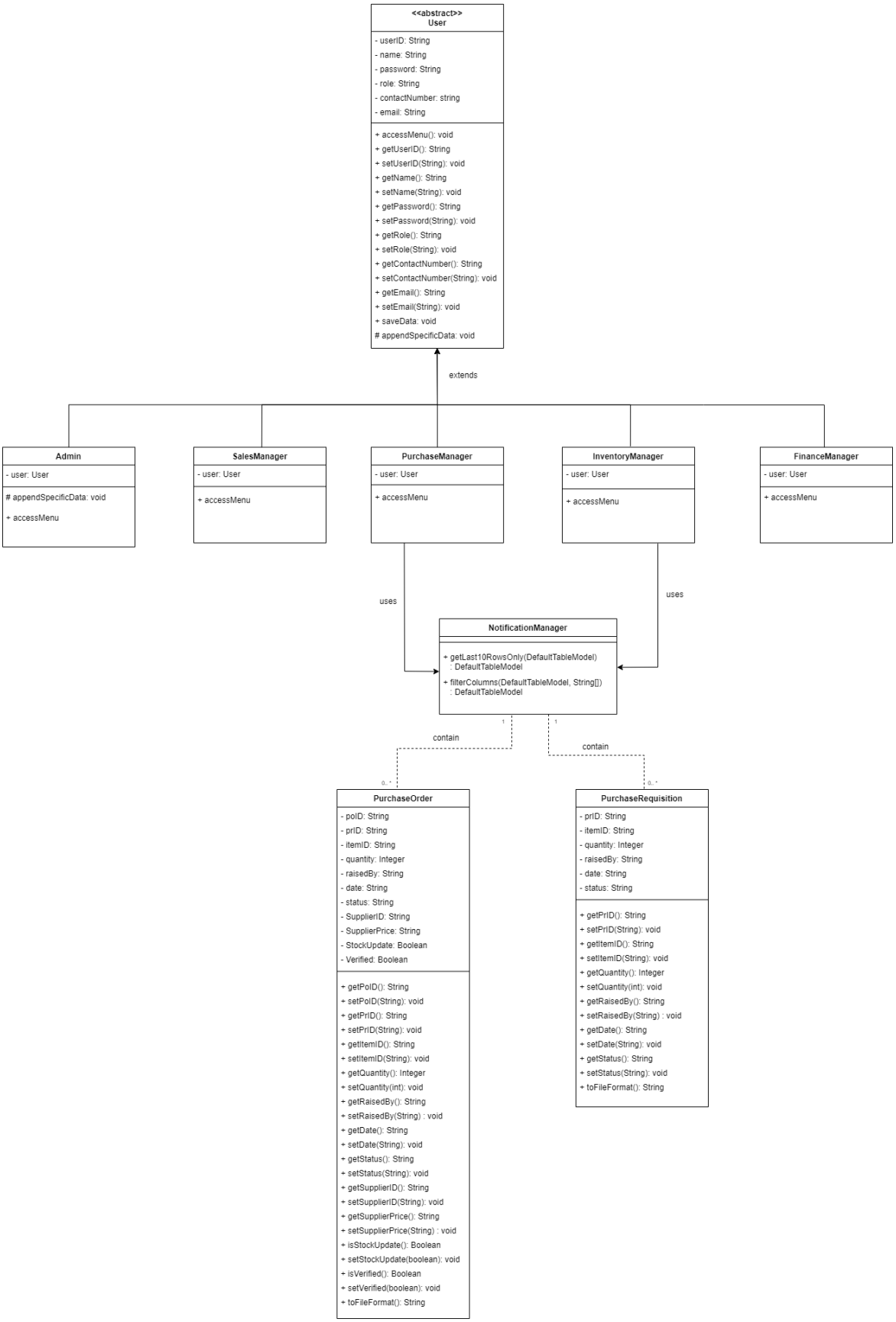
# 3. Design Diagrams

## 3.1 Use Case Diagram

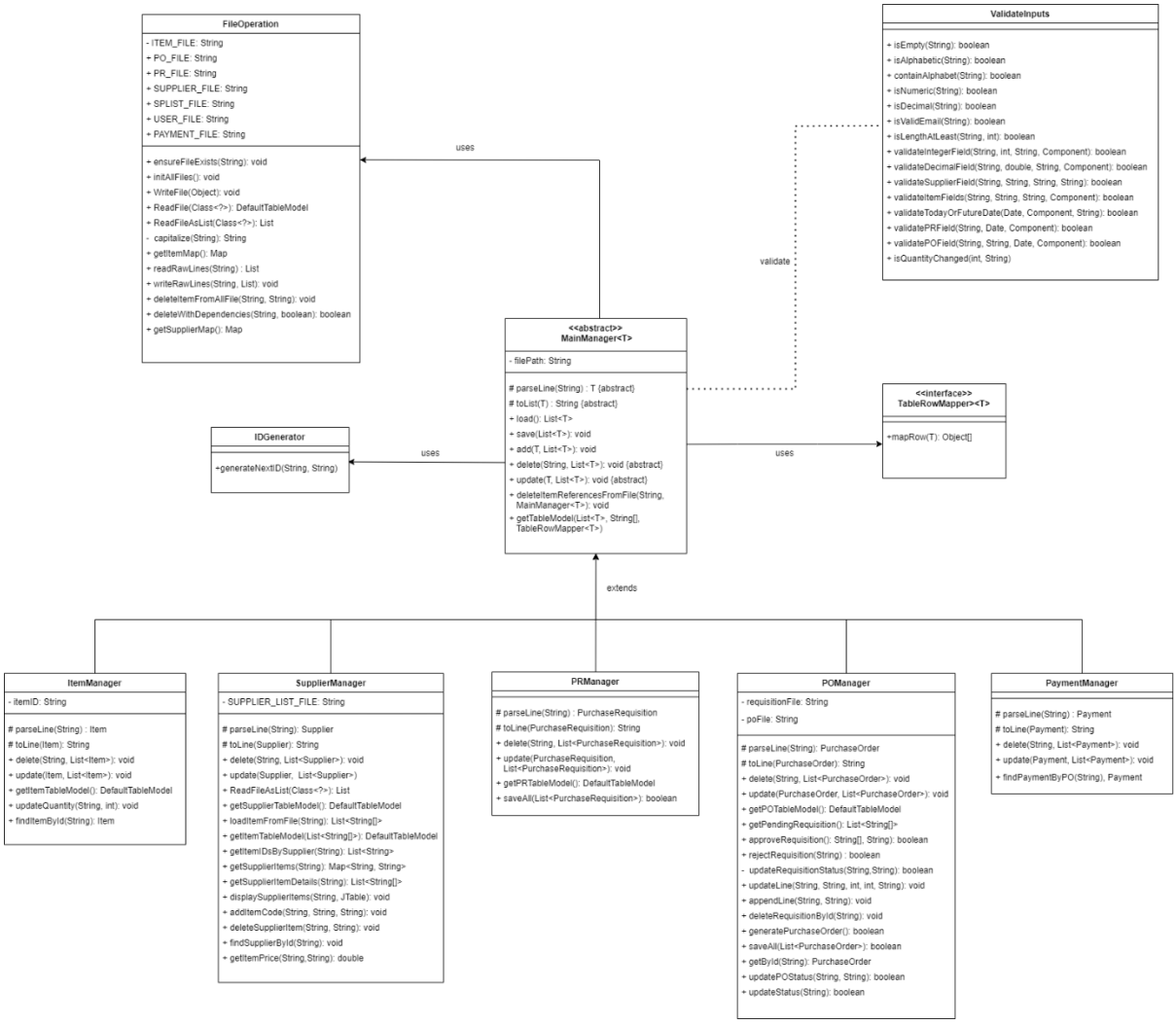


# 3.2 Class Diagram

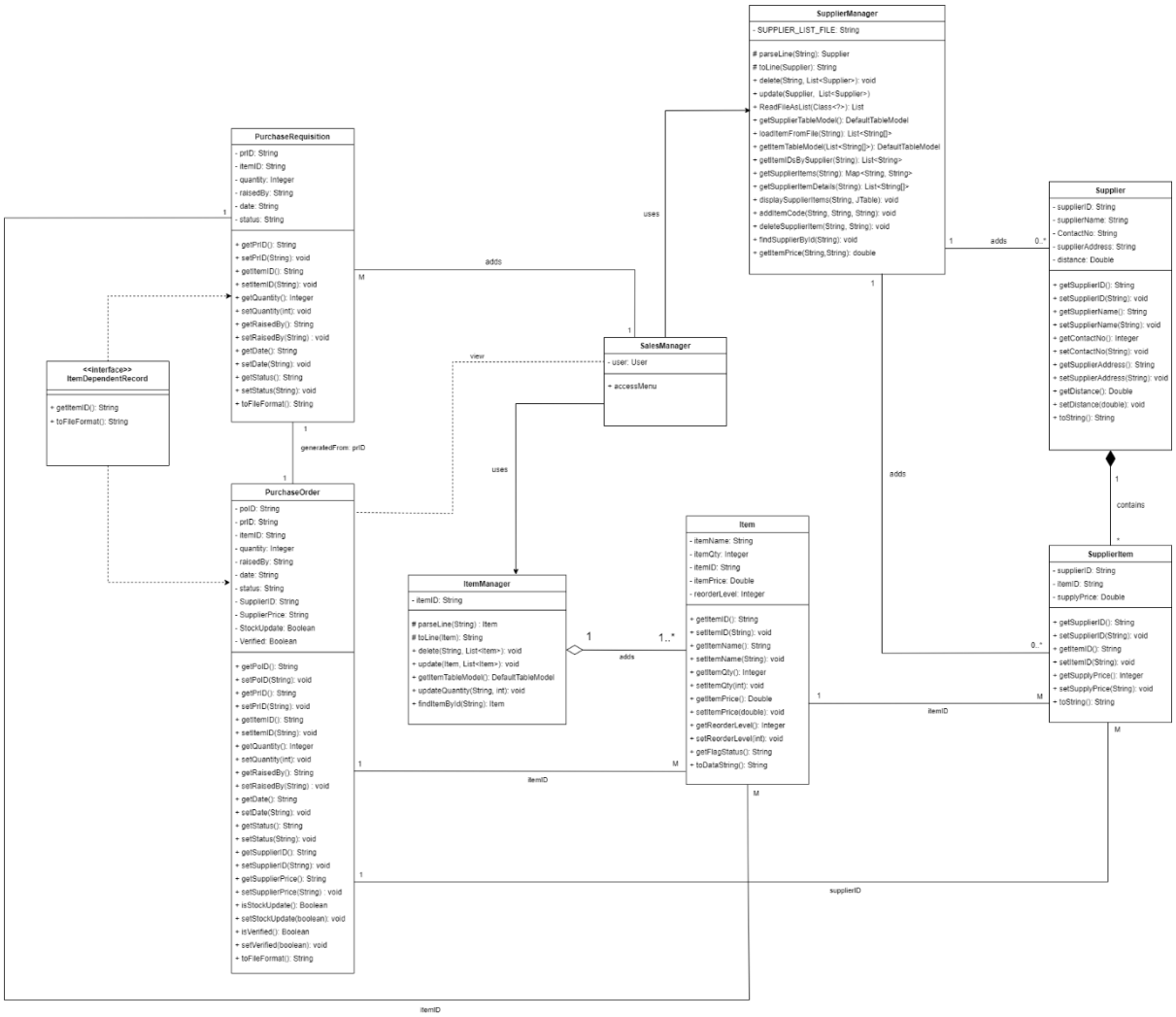
## 3.2.1 User Class Diagram



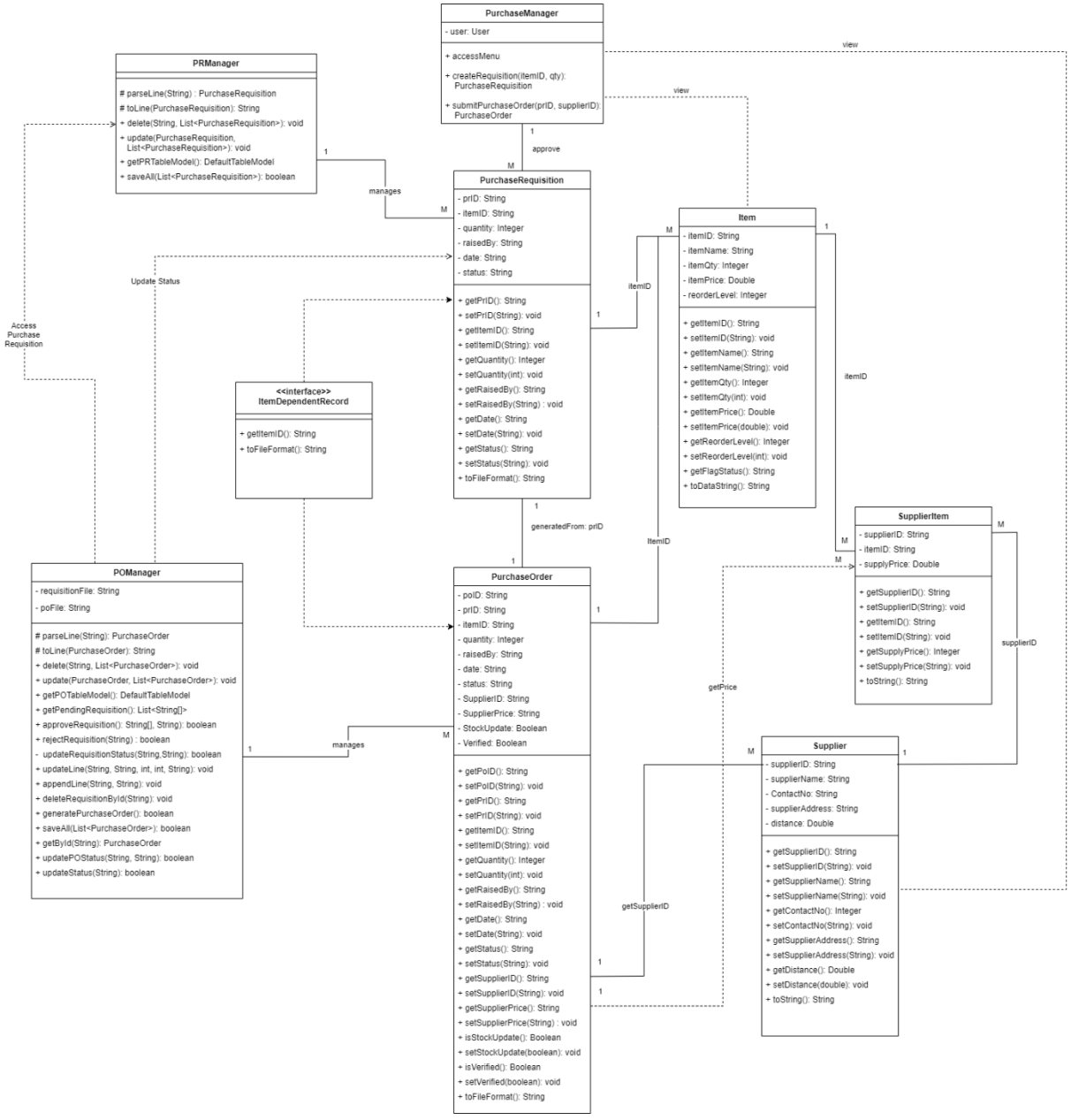
## 3.2.2 Service Manager Class Diagram



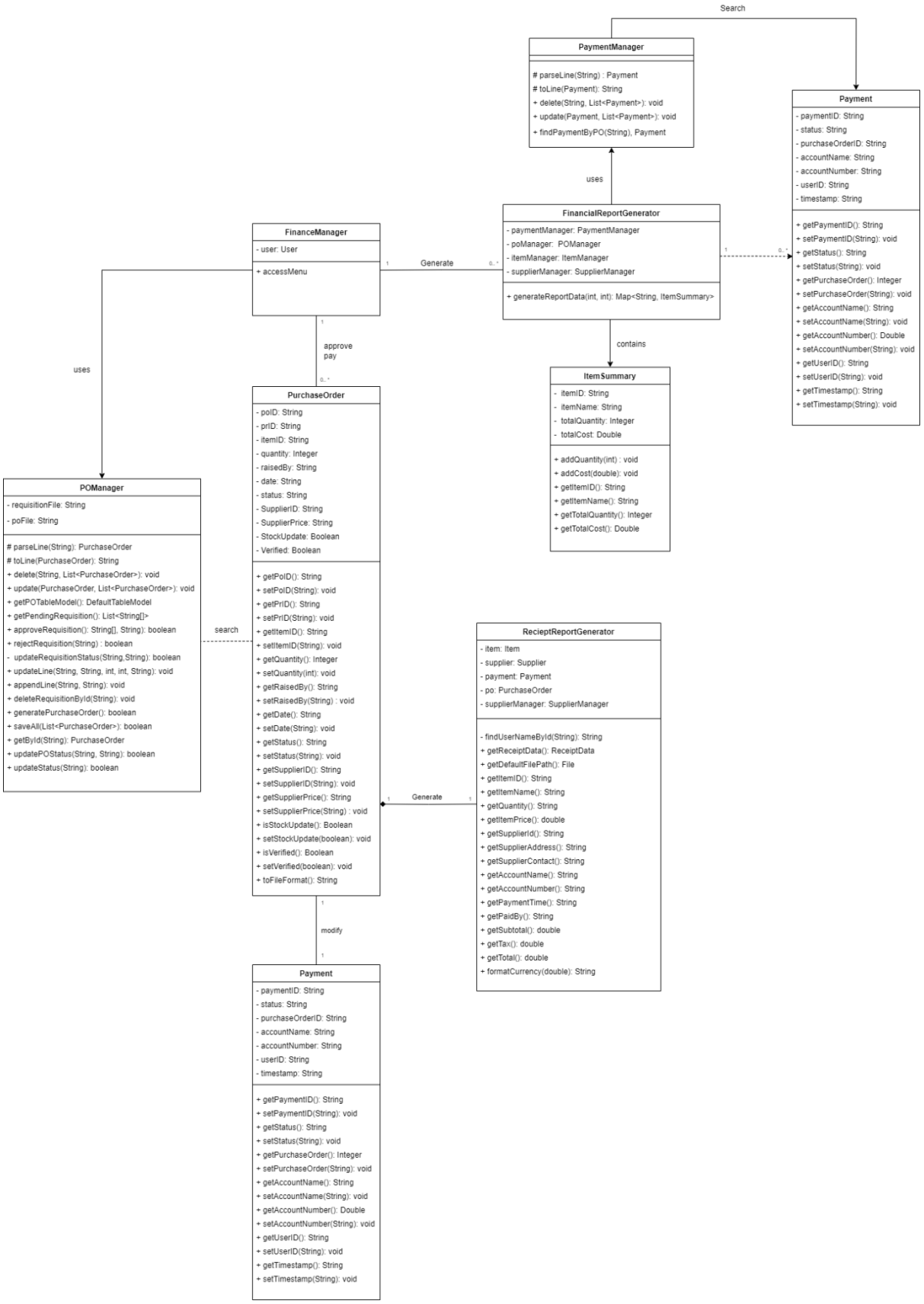
## 3.2.3 Sales Manager Class Diagram



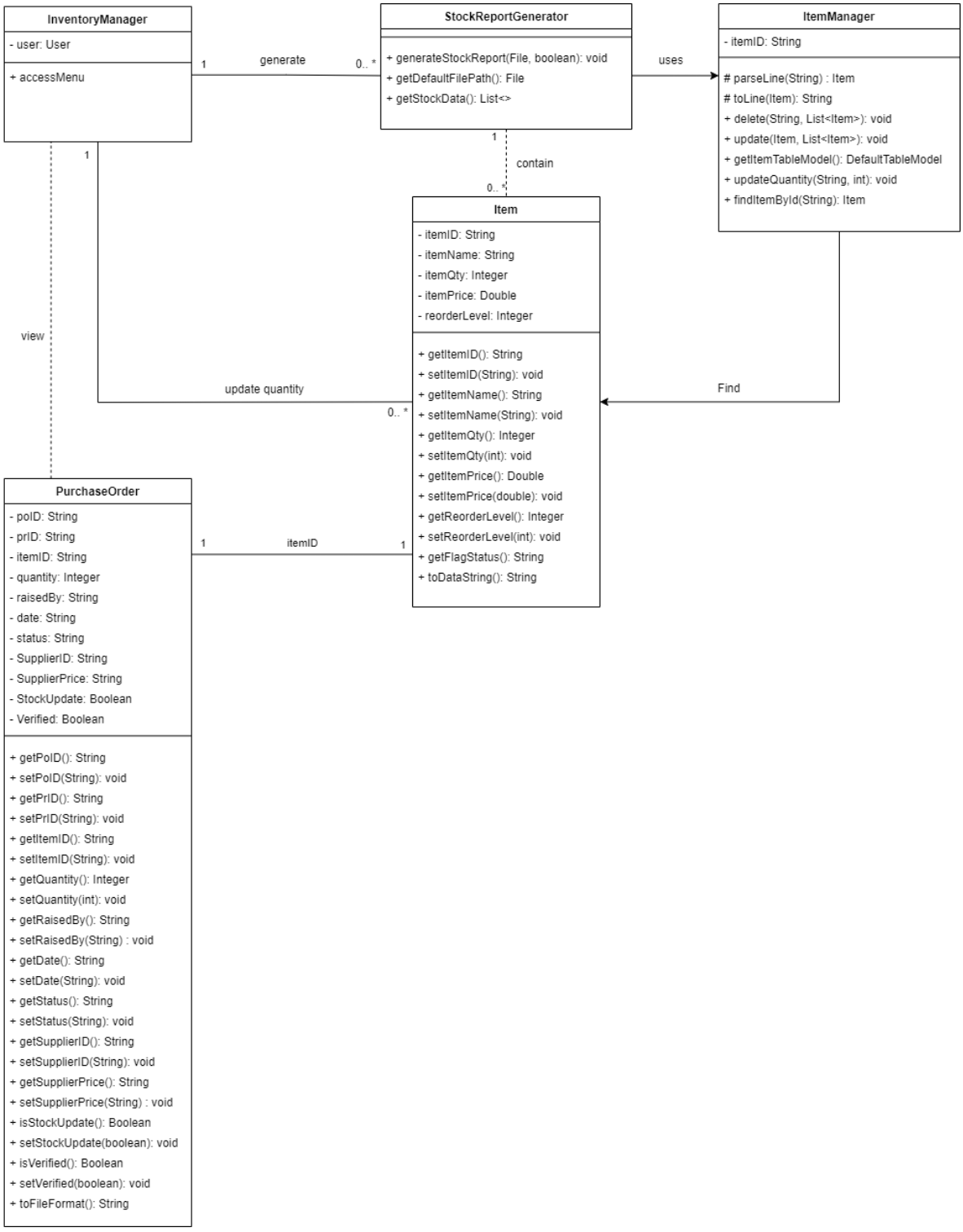
## 3.2.4 Purchase Manager Class Diagram



## 3.2.5 Finance Manager Class Diagram



## 3.2.6 Inventory Manager Class Diagram



# 4. OOP Techniques

Object-Oriented Programming (OOP) is a software development approach that revolves around the use of objects and classes (Zhang et al., 2023). It emphasizes modularity, data hiding, and code reusability through key principles such as encapsulation, inheritance, polymorphism, and abstraction (Liu et al., 2023). Additionally, supporting principles such as constructor overloading, classes and objects, access modifiers, exception handling, coupling, composition, association, and aggregation enhance the structure and readability of large-scale applications. This section documents how these Object-Oriented Programming (OOP) principles are demonstrated within the system.

## 4.1 Classes and Objects

```
public class Admin extends User{
    private User user;

    public Admin(String userId, String name, String password, String role, String contactNumber, String email) {
        super(userId, name, password, "Admin", contactNumber, email);
    }

    public Admin(){};

    public Admin(String userId, String name){
        super(userId, name);
    };

    @Override
    public void accessMenu() {
        new AdminMenu(user).setVisible(true);
    }
```

**Figure 4.1: Admin class declaration, constructors, and behavior definition**

Classes and Objects are foundational to OOP. A class is a blueprint that defines an entity's structure and behaviors. An object is a runtime instance of a class that embodies real data and executes defined methods (GeeksforGeeks, 2025). In the Admin class, constructors allow for flexible instantiation of objects with varying detail. The method accessMenu() illustrates how the object behaves once constructed. This reflects the essence of OOP: defining reusable templates (classes) and creating functional entities (objects) from them. By invoking admin.accessMenu(), the object executes a behavior specific to the Admin role, showing how objects represent real-world entities and perform logic defined in their classes. This compact interaction highlights how identity and behavior are encapsulated within objects while structured through their class definitions.

## 4.2 Constructor Overloading

```
public abstract class User {
    private String UserID;
    private String Name;
    private String Password;
    private String role;
    private String contactNumber;
    private String email;


    public abstract void accessMenu();

    public User(String UserID, String Name, String Password, String role, String contactNumber, String email) {
        this.UserID = UserID;
        this.Name = Name;
        this.Password = Password;
        this.role = role;
        this.contactNumber = contactNumber;
        this.email = email;
    }

    public User(String Name, String UserID) {
        this.Name = Name;
        this.UserID = UserID;
    }
```

**Figure 4.2: Overloaded constructors providing flexible initialization in User**

Constructor overloading exemplifies compile-time polymorphism by allowing multiple constructors in the same class with different parameter lists. This enables objects to be instantiated in different ways depending on available data (Sharma et al., 2012). In the example above, the User class provides two constructors—one that accepts full user details and another that only takes a name and ID. This design increases flexibility and supports modular system development by allowing partial initialization where full details are not yet available. Such patterns are especially useful in layered systems, temporary user creation, or abstracted object factories.

## 4.3 Access Modifiers

```
public class Supplier{
    private String supplierID;
    private String supplierName;
    private String ContactNo;
    private String supplierAddress;
    private double distance;
```

**Figure 4.3: Example of private and public access modifiers in Java**

```
public String getSupplierID() {
    return supplierID;
}

public void setSupplierID(String supplierID) {
    this.supplierID = supplierID;
}

public String getSupplierName() {
    return supplierName;
}

public void setSupplierName(String supplierName) {
    this.supplierName = supplierName;
}

public String getContactNo() {
    return ContactNo;
}

public void setContactNo(String ContactNo) {
    this.ContactNo = ContactNo;
}
```

**Figure 4.4: Example of getters and setters used for access and modification control**

Access modifiers play a crucial role in defining the visibility scope of classes, attributes, and methods. They are instrumental in controlling how the internal components of a class can be accessed from other parts of the program, promoting secure and maintainable code structures (Sutherland, 2014). In the Supplier class, attributes like supplierID and supplierName are marked as private, which means they cannot be accessed directly from outside the class. Instead, public getters and setters are used to control access and modify these values. Access modifiers define visibility scope. For example, marking a field private prevents external access, while public methods like getters and setters offer controlled entry points. By keeping fields private and exposing behavior through public methods, developers can enforce validation, logging, or other business rules, ensuring system consistency and reducing risk from unintended external interference (Yu et al., 2021).

## 4.4 Encapsulation

```
public class Supplier{
    private String supplierID;
    private String supplierName;
    private String ContactNo;
    private String supplierAddress;
    private double distance;

public Supplier() {}

public Supplier(String supplierID, String supplierName, String ContactNo, String supplierAddress, double distance){
    this.supplierID = supplierID;
    this.supplierName = supplierName;
    this.ContactNo = ContactNo;
    this.supplierAddress = supplierAddress;
    this.distance = distance;
}

public String getSupplierID() {
    return supplierID;
}

public void setSupplierID(String supplierID) {
    this.supplierID = supplierID;
}

public String getSupplierName() {
    return supplierName;
}

public void setSupplierName(String supplierName) {
    this.supplierName = supplierName;
}

public String getContactNo() {
    return ContactNo;
}

public void setContactNo(String ContactNo) {
    this.ContactNo = ContactNo;
}

public String getSupplierAddress() {
    return supplierAddress;
}

public void setSupplierAddress(String supplierAdress) {
    this.supplierAddress = supplierAdress;
}

public double getDistance(){
    return distance;
}

public void setDistance(double distance){
    this.distance = distance;
}
```

**Figure 4.5: Encapsulation in Supplier with controlled field access**

```
public class PurchaseOrder implements ItemDependentRecord{
    private String poID;
    private String prID;
    private String itemID;
    private int quantity;
    private String raisedBy;
    private String date;
    private String status;
    private String SupplierID;
    private String SupplierPrice;

    public PurchaseOrder() {}

    public PurchaseOrder(String poID, String prID, String itemID, int quantity, String raisedBy, String date, String status, String SupplierID, String SupplierPrice) {
        this.poID = poID;
        this.prID = prID;
        this.itemID = itemID;
        this.quantity = quantity;
        this.raisedBy = raisedBy;
        this.date = date;
        this.status = status;
        this.SupplierID = SupplierID;
        this.SupplierPrice = SupplierPrice;
    }
```

**Figure 4.6: Encapsulation in Practice within PurchaseOrder**

Encapsulation is the principle of combining data and the methods that operate on that data within a single unit while shielding internal details from direct external access. This object-oriented feature helps ensure that data is not modified or accessed unintentionally by external code (Braunschweig, 2018). Instead, all access must go through controlled interfaces such as getters and setters. In the Supplier class, all fields like supplierID, supplierName, and distance are declared private, and access to them is granted through public getter and setter methods. This restricts direct external modification, allowing the class to maintain control over its internal state.

The PurchaseOrder class further demonstrates encapsulation in practice. Rather than exposing supplier data directly, it includes a Supplier object internally and accesses its information only through the public interface provided by the Supplier class. This reinforces the encapsulation boundary—each class manages its own data integrity while collaborating through controlled points of contact. Together, these classes highlight how encapsulation promotes modularity, prevents data corruption, and enhances maintainability in an object-oriented system (Chodrow et al., 2021).

## 4.5 Inheritance

```java
public abstract class User {
    private String UserID;
    private String Name;
    private String Password;
    private String role;
    private String contactNumber;
    private String email;


    public abstract void accessMenu();

    public User(String UserID, String Name, String Password, String role, String contactNumber, String email) {
        this.UserID = UserID;
        this.Name = Name;
        this.Password = Password;
        this.role = role;
        this.contactNumber = contactNumber;
        this.email = email;
    }

    public User(String Name, String UserID) {
        this.Name = Name;
        this.UserID = UserID;
    }
```

**Figure 4.7: User superclass with abstract structure and shared attributes**

```java
public class Admin extends User{
    private User user;

    public Admin(String userId, String name, String password, String role, String contactNumber, String email) {
       super(userId, name, password, "Admin", contactNumber, email);
    }

    public Admin(){};

    public Admin(String userId, String name){
        super(userId, name);
    };

    @Override
    public void accessMenu() {
        new AdminMenu(user).setVisible(true);
    }
```

**Figure 4.8: Admin subclass inheriting from User and extending functionality**

Inheritance is a core principle in Object-Oriented Programming that allows a class (the subclass) to derive fields and methods from another class (the superclass) (Naqvi, 2024). It promotes code reuse, simplifies system design, and enables polymorphism by forming a hierarchical relationship between classes. In the first snippet, User is defined as an abstract superclass with shared attributes like UserID, Name, and Password, and includes abstract behavior through the accessMenu() method. In the second snippet, the Admin class extends User, automatically gaining access to its fields and constructors. It also overrides the abstract method accessMenu(), enabling role-specific behavior. This demonstrates how inheritance allows a subclass like Admin to reuse and specialize the foundational structure of User, reducing duplication and enabling consistent functionality across different user roles.

## 4.6 Abstraction

```
public abstract class User {
    private String UserID;
    private String Name;
    private String Password;
    private String role;
    private String contactNumber;
    private String email;


    public abstract void accessMenu();

    public User(String UserID, String Name, String Password, String role, String contactNumber, String email) {
        this.UserID = UserID;
        this.Name = Name;
        this.Password = Password;
        this.role = role;
        this.contactNumber = contactNumber;
        this.email = email;
    }

    public User(String Name, String UserID) {
        this.Name = Name;
        this.UserID = UserID;
    }
```

**Figure 4.9 Abstract User class defining structure and required behavior**

```
public class FinanceManager extends User{
    private User user;

    public FinanceManager(String userId, String name, String password, String role, String contactNumber, String email) {
        super(userId, name, password, "Finance Manager", contactNumber, email);
    }

    @Override
    public void accessMenu() {
        new FinanceManagerMenu(user).setVisible(true);
    }
}
```

**Figure 4.10: FinanceManager subclass implementing the abstract method**

```
package models;

public interface ItemDependentRecord {
    String getItemID();
    String toFileFormat();
}
```

**Figure 4.11: Interface-based abstraction using ItemDependentRecord**

Abstraction is an object-oriented programming principle that focuses on exposing only the essential characteristics of an object while hiding the internal complexities of its implementation. It helps manage system complexity by defining a simplified interface for interacting with different types of objects (Bhat, 2024). Java supports abstraction through both abstract classes and interfaces. In the system, the User class demonstrates abstraction by defining the abstract method accessMenu(). This method acts as a contract that all subclasses—such as FinanceManager, Admin, and SalesManager—must implement. Each subclass

provides its own behavior for accessMenu(), allowing flexibility in user interface design while enforcing structural consistency across user roles.

Additionally, the ItemDependentRecord interface represents a more lightweight form of abstraction. It declares two methods—getItemID() and toFileFormat()—which must be implemented by any class that depends on item-related data, such as PurchaseOrder and PurchaseRequisition. This enforces a common protocol without dictating how the data is handled internally. Interfaces allow for high-level abstraction and decoupling, enabling unrelated classes to interact through a shared contract even if they do not share a superclass. By combining abstract classes and interfaces, the system leverages abstraction to enforce uniform behavior, support polymorphism, and reduce interdependence between components.

## 4.7 Polymorphism

```
public class SalesManager extends User{
    private User user;

    public SalesManager(String userId, String name, String password, String role, String contactNumber, String email) {
        super(userId, name, password, "Sales Manager", contactNumber, email);
    }

    @Override
    public void accessMenu() {
        new SalesManagerMenu(user).setVisible(true);
    }
}
```

**Figure 4.12: Overridden accessMenu() methods in different subclasses of User (SalesManager)**

```
public class InventoryManager extends User{
    private User user;

    public InventoryManager(String userId, String name, String password, String role, String contactNumber, String email) {
        super(userId, name, password, "Inventory Manager", contactNumber, email);
    }

    @Override
    public void accessMenu() {
        new InventoryManagerMenu(user).setVisible(true);
    }
}
```

**Figure 4.13: Overridden accessMenu() methods in different subclasses of User (InventoryManager)**

Polymorphism enables a unified interface to represent different underlying forms. It allows one method to operate differently based on the object's runtime type, providing flexibility and dynamic behavior in systems (Taylor, 2023). In the above example, both SalesManager and InventoryManager override the accessMenu() method of the User class. Although the method

name is the same, the behavior is specific to the subclass. This ensures that the correct menu interface appears depending on the user role, fulfilling OOP's goal of writing extensible, maintainable code where new user types can be introduced with minimal changes to existing logic.

## 4.8 Exception Handling

```
public static void WriteFile(Object obj) {
    Class<?> classObj = obj.getClass();
    String fileName = classObj.getSimpleName() + ".txt";
    File file = new File(fileName);

    try {
        boolean fileExists = file.exists();
        PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter(file, true)));

        Field[] fields = classObj.getDeclaredFields();

        // Write CSV header only if file is new
        if (!fileExists) {
            StringBuilder header = new StringBuilder();
            for (Field field : fields) {
                header.append(field.getName()).append(",");
            }
            pw.println(header.substring(0, header.length() - 1)); // remove trailing comma
        }

        // Write object data row
        StringBuilder row = new StringBuilder();
        for (Field field : fields) {
            String getterName = "get" + capitalize(field.getName());
            try {
                Method getter = classObj.getMethod(getterName);
                Object value = getter.invoke(obj);
                row.append(value).append(",");
            } catch (NoSuchMethodException ignored) {}
        }
        pw.println(row.substring(0, row.length() - 1)); // remove trailing comma
        pw.close();

    } catch (IOException | IllegalAccessException | InvocationTargetException e) {
        e.printStackTrace();
    }
}
```

**Figure 4.14: Exception handling in file-writing logic**

Exception handling is a structured mechanism in Java that allows developers to manage errors gracefully and maintain application stability (Drabent, 2023). It separates normal logic from error-handling logic, ensuring that runtime issues do not cause abrupt failures. In the example from FileOperation.java, the WriteFile method uses a try-catch block to manage exceptions that might occur during file writing or reflective method invocation. Checked exceptions like

IOException, IllegalAccessException, and InvocationTargetException are caught and logged using e.printStackTrace(). This prevents the application from crashing and allows developers to trace what went wrong. Additionally, nested try-catch blocks handle missing getter methods (NoSuchMethodException), which may arise from reflection. This design demonstrates how OOP uses exception handling to enforce robustness and decouples core logic from fault recovery, allowing systems to operate reliably even when faced with unexpected input or I/O issues.

## 4.9 Coupling

```
public class FinanceManagerMenu extends javax.swing.JFrame {
    private User user;

    /**
     * Creates new form FinancialManagerMenu
     */
    public FinanceManagerMenu(User user) {
        this.user = user;
        initComponents();
        setLocationRelativeTo(null);
    }
```

**Figure 4.15: Loose coupling in FinanceManagerMenu through dependency injection**

Coupling refers to the degree of interdependence between classes. Loose coupling—where classes interact through well-defined interfaces or abstractions—is encouraged in OOP for greater flexibility and easier maintainability (Vats, 2025). In the snippet above, the FinanceManagerMenu class accepts a User object through its constructor. It stores the reference and initializes the UI using that object without knowing or depending on the concrete subclass (e.g., Admin, FinanceManager). This design pattern, known as dependency injection, promotes separation of concerns. It allows components to be developed, tested, and maintained independently while still collaborating effectively. Loose coupling reduces the risk of changes in one class causing unintended effects in others.

## 4.10 Composition

```
public class PurchaseOrder implements ItemDependentRecord{
    private String poID;
    private String prID;
    private String itemID;
    private int quantity;
    private String raisedBy;
    private String date;
    private String status;
    private String SupplierID;
    private String SupplierPrice;

    public PurchaseOrder() {}

    public PurchaseOrder(String poID, String prID, String itemID, int quantity, String raisedBy,
            String date, String status, String SupplierID, String SupplierPrice) {
        this.poID = poID;
        this.prID = prID;
        this.itemID = itemID;
        this.quantity = quantity;
        this.raisedBy = raisedBy;
        this.date = date;
        this.status = status;
        this.SupplierID = SupplierID;
        this.SupplierPrice = SupplierPrice;
    }
```

**Figure 4.16: Composition in PurchaseOrder using multiple composed fields**

```
public class Item {
    private String itemID;
    private String itemName;
    private int itemQty;
    private double itemPrice;
    private int reorderLevel;

    // Constructor initializes item attributes
    public Item(String itemID, String itemName, int itemQty, double itemPrice, int reorderLevel) {
        this.itemID = itemID;
        this.itemName = itemName;
        this.itemQty = itemQty;
        this.itemPrice = itemPrice;
        this.reorderLevel = reorderLevel;
    }

    public String getItemID() {
        return itemID;
    }
```

**Figure 4.17: Item class as a composable object within container classes**

Composition is a strong form of association in object-oriented programming where one class is made up of one or more objects from other classes. These component objects are essential to the larger object's function and typically share the same lifecycle (GeeksforGeeks, 2025). In this example, the PurchaseOrder class is composed of multiple attributes, such as itemID, SupplierID, and quantity, representing parts that define a complete order. While the current system uses IDs as placeholders, the logical structure still groups item-related data within the PurchaseOrder class. In a more object-oriented implementation, composition would directly

include Item and Supplier objects as part of the class. When a PurchaseOrder is created or destroyed, the integrity of its related components is crucial, reflecting how composition models part-whole relationships in real-world systems.

This compositional concept is further supported by the design of the Item class itself, which encapsulates key item attributes and behaviors in a reusable object. With clearly defined fields, constructors, and accessor methods, the Item class is structured to be embedded within other classes, such as orders or inventory records. This makes it ideal for composition, as objects like PurchaseOrder can instantiate and own Item objects directly, ensuring a tight and functional integration that reflects their lifecycle and dependency.

## 4.11 Association

```
public class SupplierEntry extends javax.swing.JFrame {
    private User user;
    private SupplierManager sm = new SupplierManager();
    private List<Supplier>supplierList = new ArrayList<>();
    private DefaultTableModel tableModel;
    private TableRowSorter<DefaultTableModel> sorter;

    /**
     * Creates new form SupplierEntry
     */
    public SupplierEntry(User user) {
        this.user = user;
        initComponents();
        setLocationRelativeTo(null);
        supplierList = sm.load();
        tableModel = sm.getSupplierTableModel();
        tblSupplier.setModel(tableModel);
        sorter = new TableRowSorter<>(tableModel);
        String itemID = IDGenerator.generateNextID("SP", "Supplier.txt");
        txtSupplierID.setText(itemID);
```

**Figure 4.18: Association between SupplierEntry and SupplierManager/Supplier objects**

Association describes a relationship where one object interacts with another without owning or managing its lifecycle. In this example, the SupplierEntry class uses a SupplierManager object and a list of Supplier objects. The relationship is formed through field declarations and method calls such as sm.load() and sm.getSupplierTableModel().

Here, SupplierEntry depends on SupplierManager to manage supplier data and on Supplier objects to populate the table model, but it does not create or destroy these objects directly. This reflects a use relationship where objects work together while remaining independent. Association supports modular, maintainable system design where components collaborate through clearly defined roles (Gessenharter, 2009).

## 4.12 Aggregation

```
public class PurchaseRequisition implements ItemDependentRecord{
    private String prID;
    private String itemID;
    private int quantity;
    private String raisedBy;
    private String date;
    private String status;

    public PurchaseRequisition() {}

    public PurchaseRequisition(String prID, String itemID, int quantity,
            String raisedBy, String dateNeeded, String status) {
        this.prID = prID;
        this.itemID = itemID;
        this.quantity = quantity;
        this.raisedBy = raisedBy;
        this.date = dateNeeded;
        this.status = status;
    }
```

**Figure 4.19: Aggregation between PurchaseRequisition and Item via itemID reference**

```
public class Item {
    private String itemID;
    private String itemName;
    private int itemQty;
    private double itemPrice;
    private int reorderLevel;

    // Constructor initializes item attributes
    public Item(String itemID, String itemName, int itemQty, double itemPrice, int reorderLevel) {
        this.itemID = itemID;
        this.itemName = itemName;
        this.itemQty = itemQty;
        this.itemPrice = itemPrice;
        this.reorderLevel = reorderLevel;
    }

    public String getItemID() {
        return itemID;
    }
```

**Figure 4.20 itemID used as reference from Item**

Aggregation is a form of association that represents a "has-a" relationship between two classes, where one class maintains a reference to another class without taking full ownership of its lifecycle. This relationship is considered a weaker form of composition because the associated object can exist independently of the class that references it (Bergenti et al., 2011). In this system, the PurchaseRequisition class demonstrates aggregation through its use of the itemID field. Instead of directly containing or instantiating an Item object, the requisition simply stores the itemID as a reference to an item that exists elsewhere in the system. The actual Item class maintains its own properties, such as itemName, itemQty, itemPrice, and reorderLevel, and it is managed independently of any requisitions that reference it.

This relationship is meaningful because a purchase requisition depends on an item's data to be valid, but it does not control the item's creation, deletion, or internal behavior. Items can be created and maintained in the inventory system without being tied to any specific requisition. Likewise, a single item can be associated with multiple requisitions. This loose linkage exemplifies aggregation: both classes are related, but neither class is strongly bound to the other’s existence. By relying on identifiers rather than full object embedding, the system promotes flexibility, separation of concerns, and modularity. It allows each class to evolve independently while still supporting logical collaboration between purchasing and inventory records. Aggregation in this case makes the system more maintainable and aligns with real-world structures, where a requisition points to an item in stock without necessarily owning or controlling it.

# 5. Screenshot of System Flow

## 5.1 Login

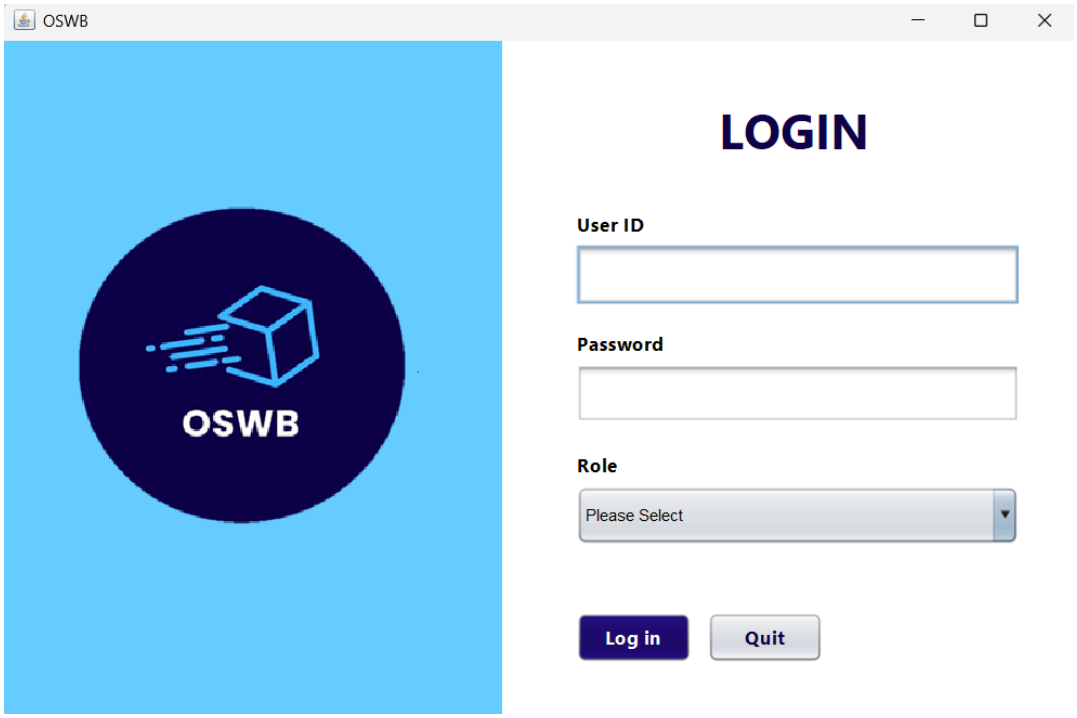



**Figure 5.1.1 Login Page**

Figure 5.1.1 shows the login page of the Omega Wholesale Sdn Bhd Purchase Order Management System. After executing the program, users will be led to the login page for login purposes. Users are required to key in User ID and Password and select the specific role at the combo box below. Users then can click on the "Login" button to access their specific role panel. If users decide to quit the system, they can click on the "Quit" button to quit the system.

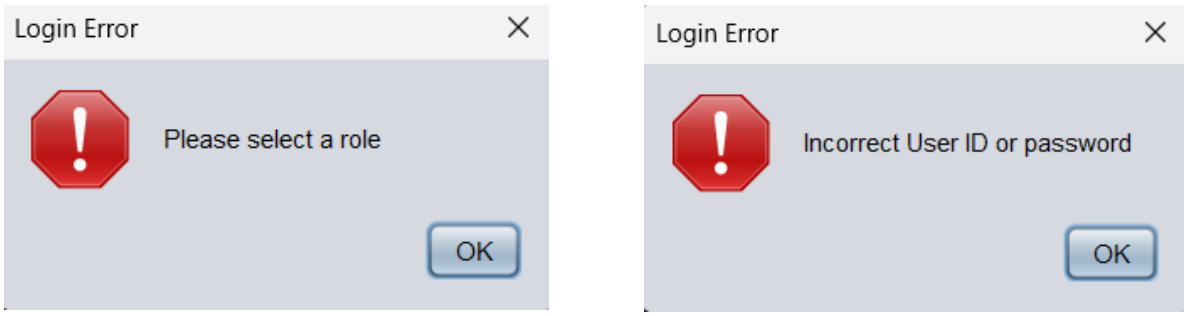



**Figure 5.1.2 Login Error Message**

Figure 5.1.2 shows the error messages that the user will receive if the login credentials are not valid. If users don't select a role and click on the "Login" button, a message of "Please select a role" will be displayed for the user. If the user didn't key in the correct login credentials, a message of "Incorrect IC number and password" will be shown.

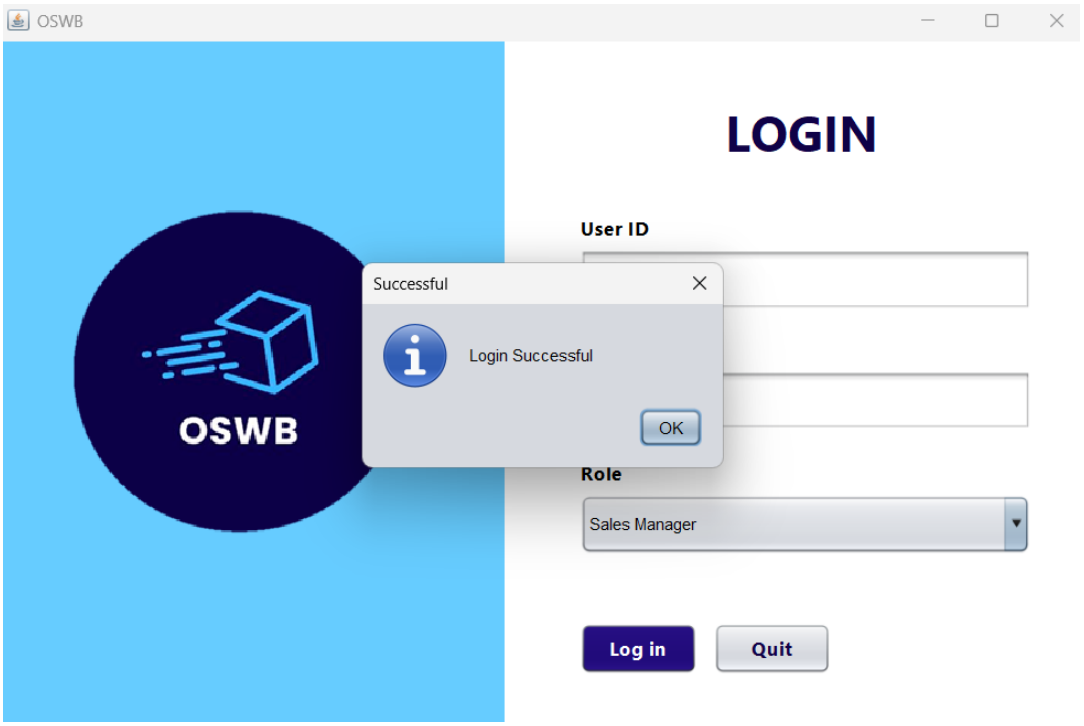



**Figure 5.1.3 Login Successful**

Figure 5.1.3 shows the login successful message if users key in the valid login credentials. Users will be led to their specific role menu after clicking on the 'OK' button.

## 5.2 User Registration

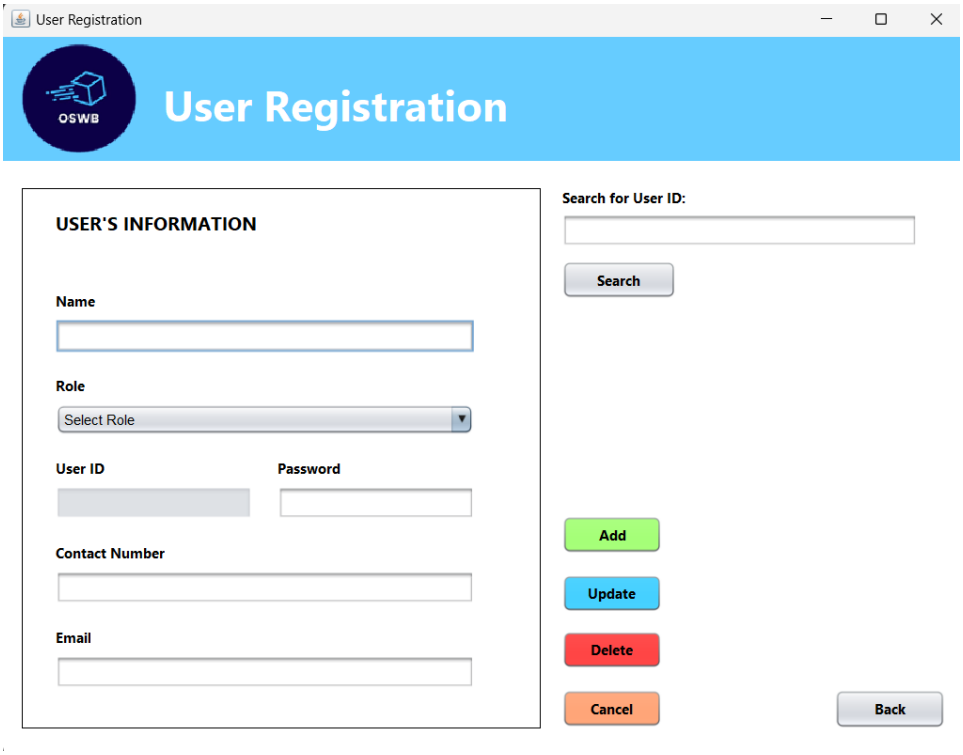



**Figure 5.5.1 User Registration Option**

Figure 5.5.1 shows the interface for the user registration where only the admin role has access to. Over here, the admin can add, update and delete roles. If admin wants to add a certain role, the admin has the option of choosing the roles between Sales Manager, Purchase Manager, Inventory Manager, Finance Manager and Admin from the combo box. After filling in all the input fields, admins are required to click on the Add button to register the user. If admins choose to update a certain role, they have to enter the User ID of the certain user in the search bar and click the "Search" button to search for the user. After modifying the details of the user, admins can click on the "Update" button to update the user. If admins choose to delete a user instead, they are also required to enter the User ID of a certain user in the search bar and click the "Delete" button to delete the user.

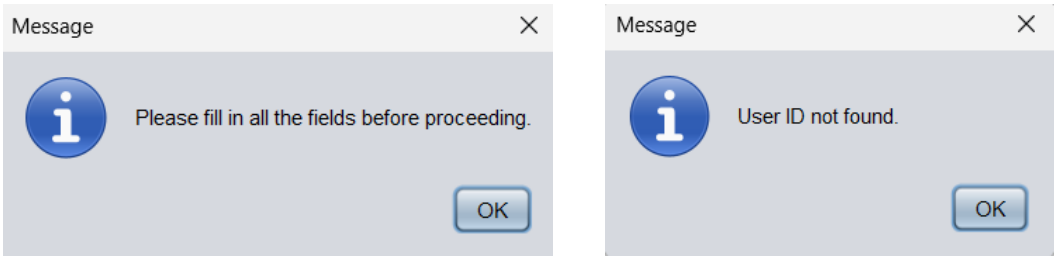



**Figure 5.2.2 Validation Message**

Figure 5.2.2 shows various types of validation messages for user validation purposes when the admin failed to fill in all the input fields or failed to fill in the correct User ID when searching for the certain user.

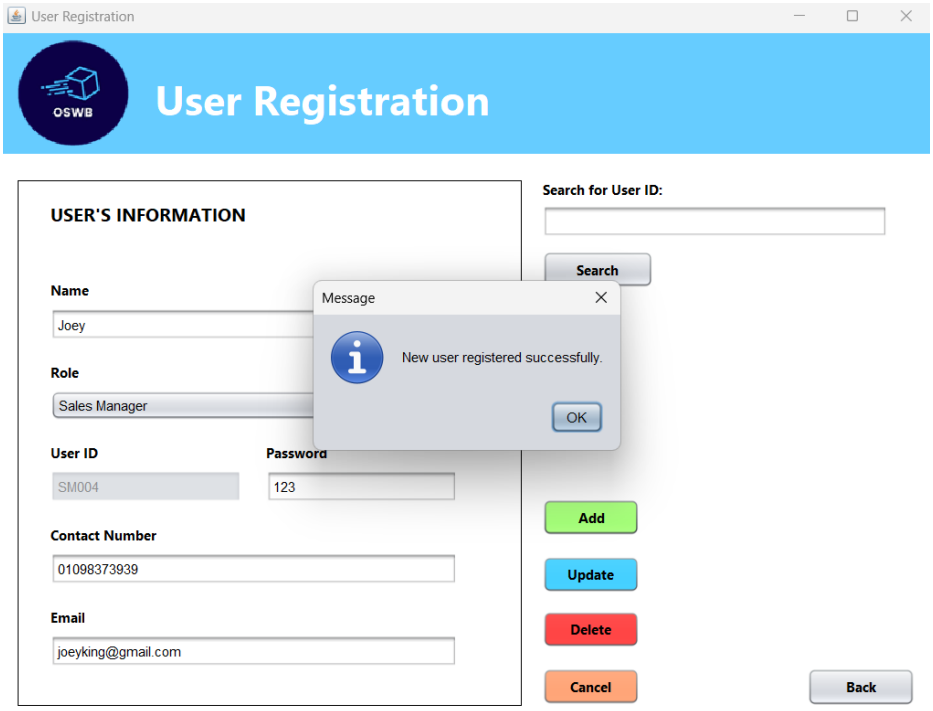



**Figure 5.5.3 User Registered Successfully**

Figure 5.5.3 shows an example of user registration being successful if all the validation is passed.

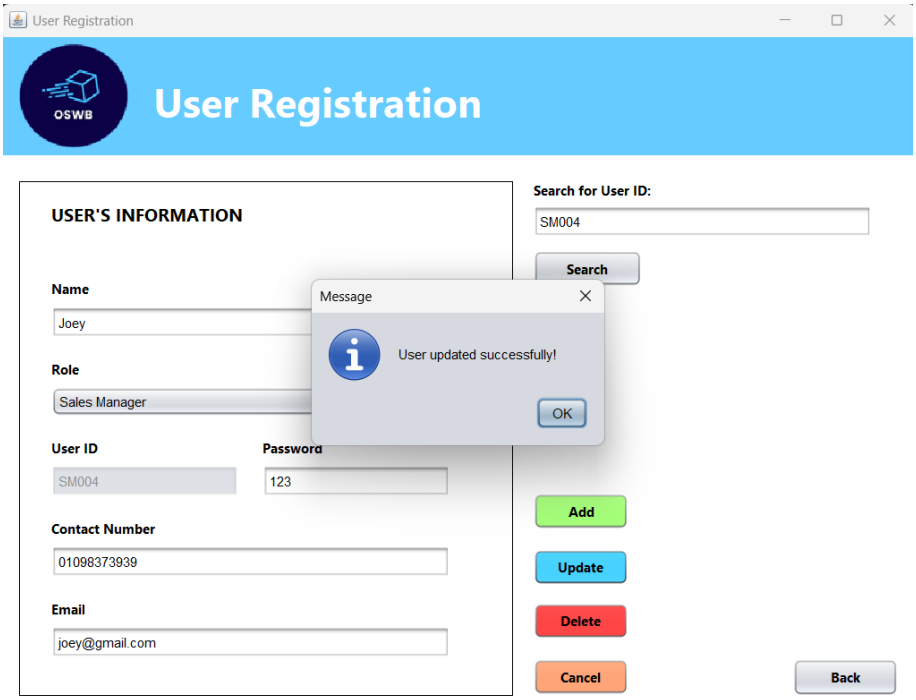



**Figure 5.5.4 User Updated Successfully**

Figure 5.5.4 shows an example of user update being successful if all the validation is passed.

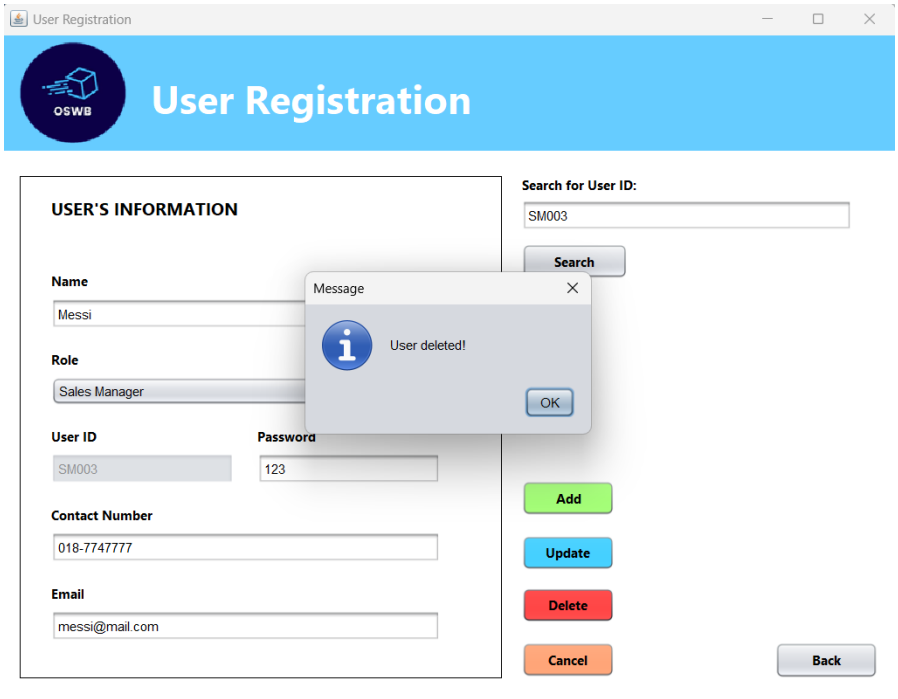



**Figure 5.5.5 User Deleted Successfully**

Figure 5.5.5 shows an example of user deletion being successful if all the validation is passed.

## 5.3 Menu Roles

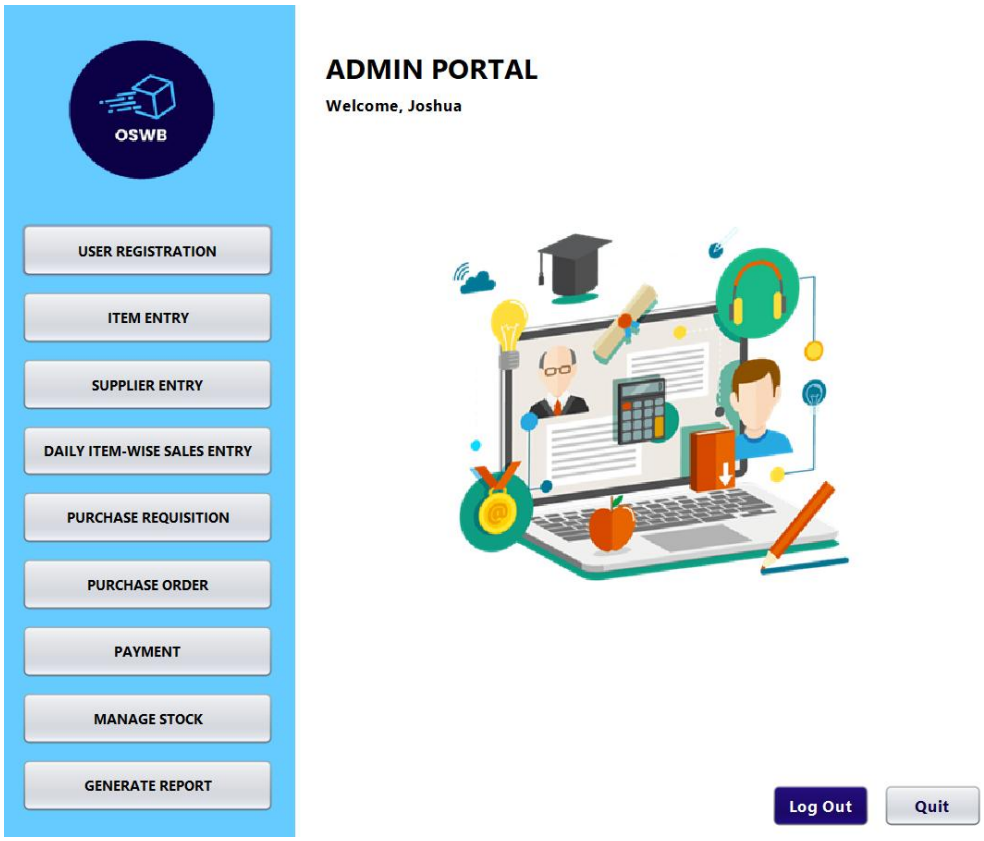



**Figure 5.3.1 Admin Menu**

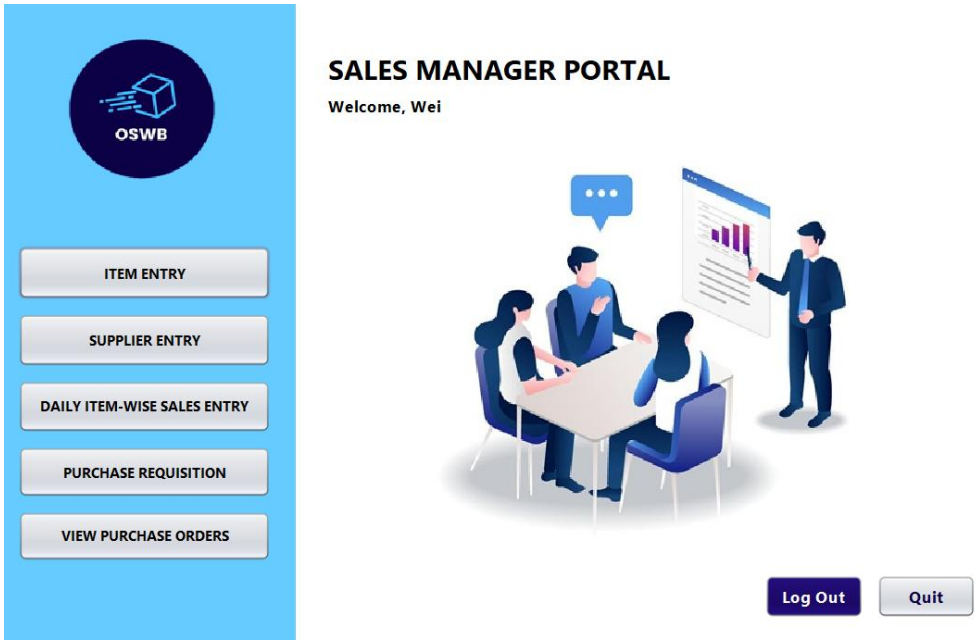



**Figure 5.3.2 Sales Manager Menu**

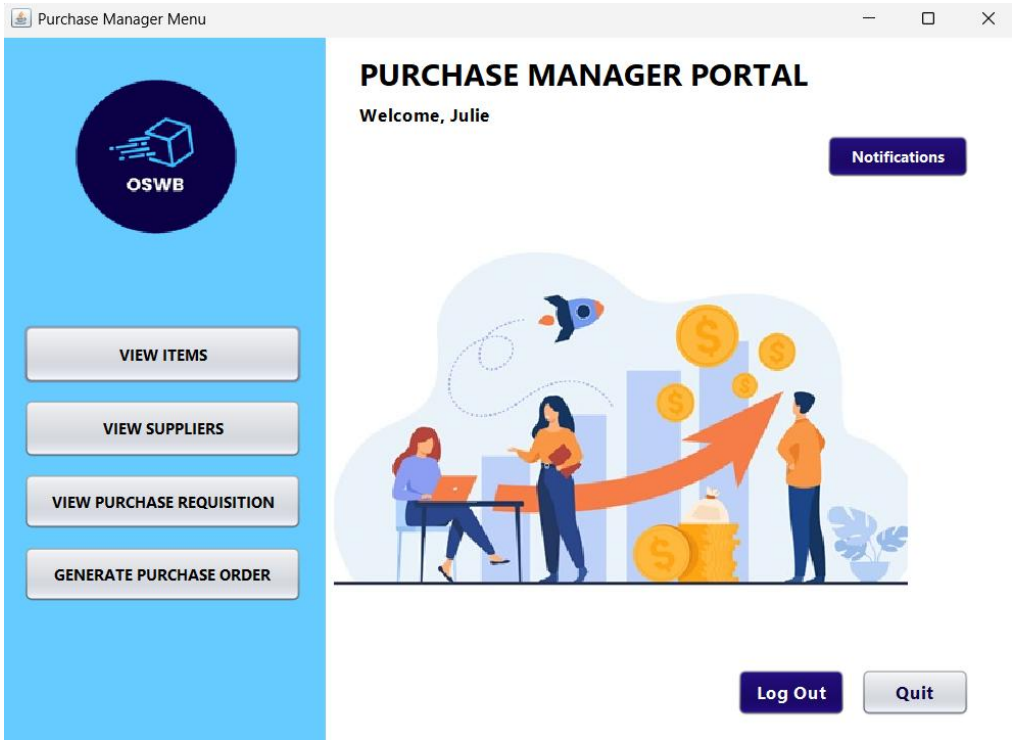



Figure 5.3.3 Purchase Manager Menu

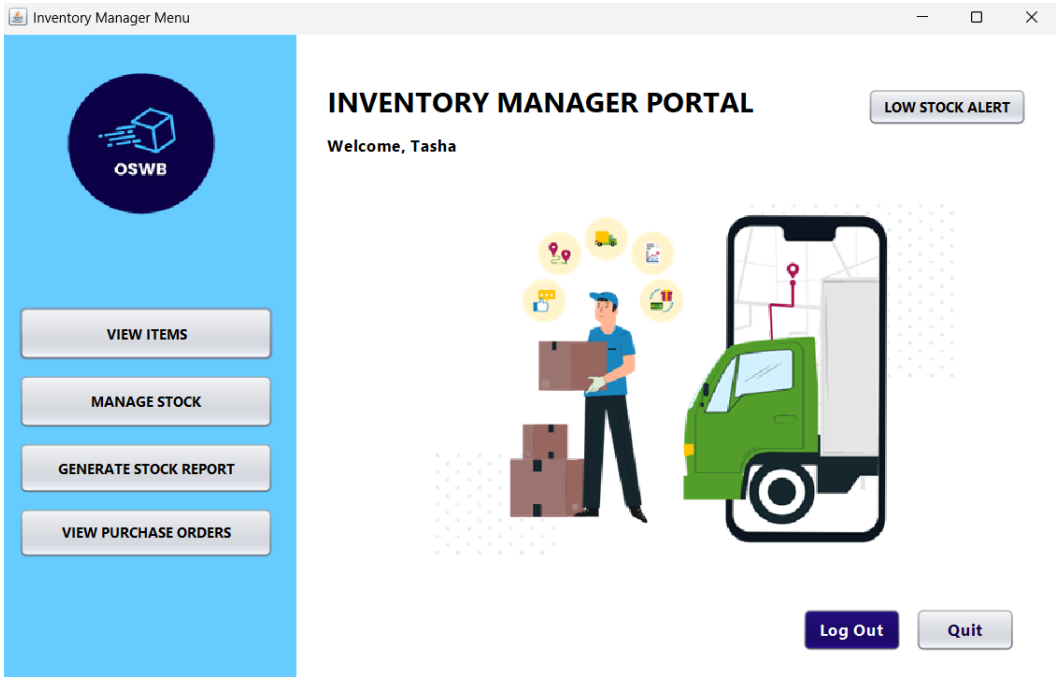



Figure 5.3.4 Inventory Manager Menu

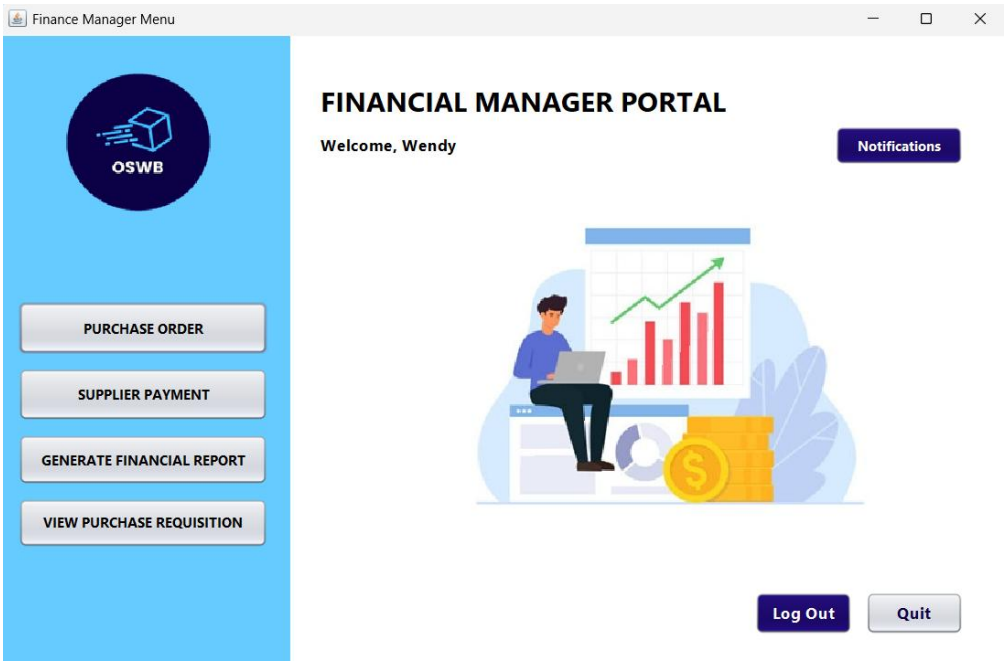



**Figure 5.3.5 Financial Manager Menu**

Figure 5.3.1 to Figure 5.3.5 shows the menu roles of all the user. In the sales manager menu, sales managers can add new items, suppliers and daily sales entry, create purchase requisition and view purchase orders. For the purchase manager menu, purchase managers can view items, supplier, purchase requisition and generate purchase order. For the inventory manager menu, inventory managers can view items, track low stock alert, generate stock report and view purchase orders. For the financial manager menu, financial managers can handle purchase orders approval, process supplier payment, generate financial report and view purchase requisitions. For the administrator menu, the admin has the access to all the functionalities as it withheld the rights to access and update all the application functionalities and data.

After clicking on the button of any specific function, the user will be led to that specific feature. The “Logout” and “Quit” buttons are provided for the user, if the user clicks on the “Log Out” button, they will be led back to the login page, whereas if the user clicks on the “Quit” button, the system will be terminated.

# 5.4 Item Entry

## 5.4.1 View Items

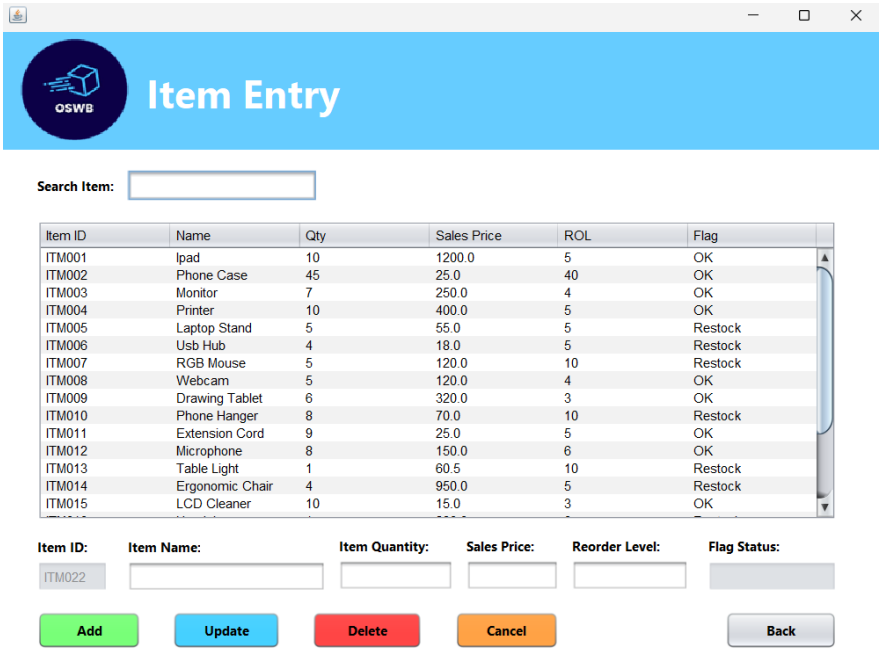



**Figure 5.4.1.1 Item Entry Interface**

Figure 5.4.1.1 shows the interface the sales manager and purchase manager will enter once the sales manager clicks on the “Item Entry” button and purchase manager clicking on the “View Items” button. In this interface, the users can view details of the items for the purchase management order system such as Item ID, Item Name, Quantity, Sales Price, Reorder Level and Flag Status. Only the sales manager has the access to add, update and delete items from the item list whereas the purchase manager is strictly restricted to viewing the items. If the sales manager wants to add a new item, the item name, item quantity, sales price and reorder level must be filled before the item is added.

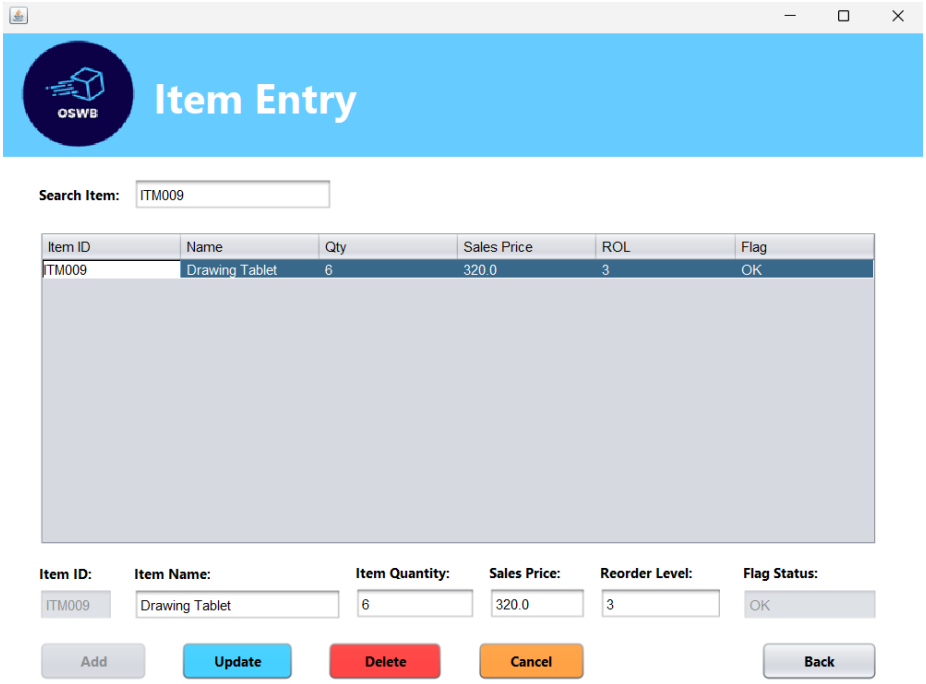
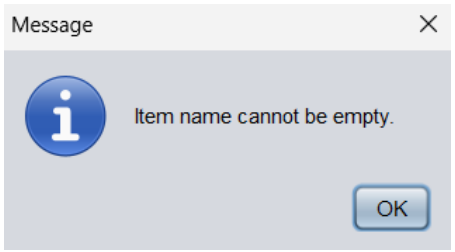
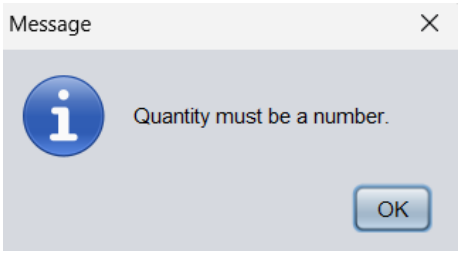


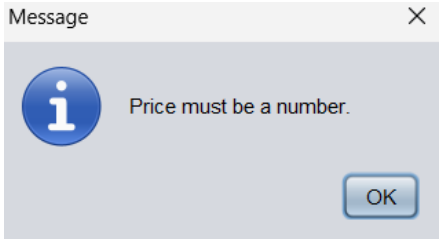

**Figure 5.4.1.2 Item Entry Information Prompt**

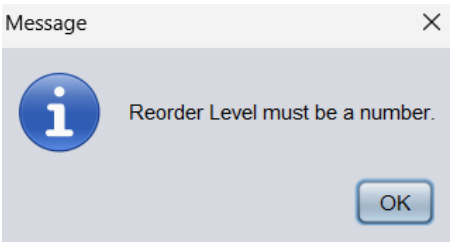
Figure 5.4.1.2 shows if the users want to search for a certain item from the item list, the users could type the item ID or item name in the search bar and if the item ID or name exists, the item will then be listed below.

## 5.4.2 New Items Entry



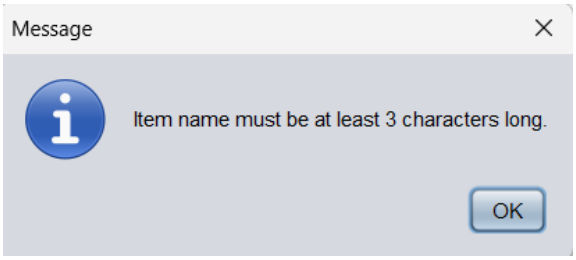



**Figure 5.4.5.1 Error Validation Notification**

Figure 5.4.5.1 shows various error validation notification for validation purposes where only the sales manager has access to. If the sales manager failed to fill in the item name, the message "Item name cannot be empty" will be shown. If they tried to enter a character in the quantity, price and reorder level input field, the respective input field with the "must be a number" will be shown. Lastly, if they tried to enter an item name with 2 characters or less, the "Item name must be at least 3 characters long" message will be shown.

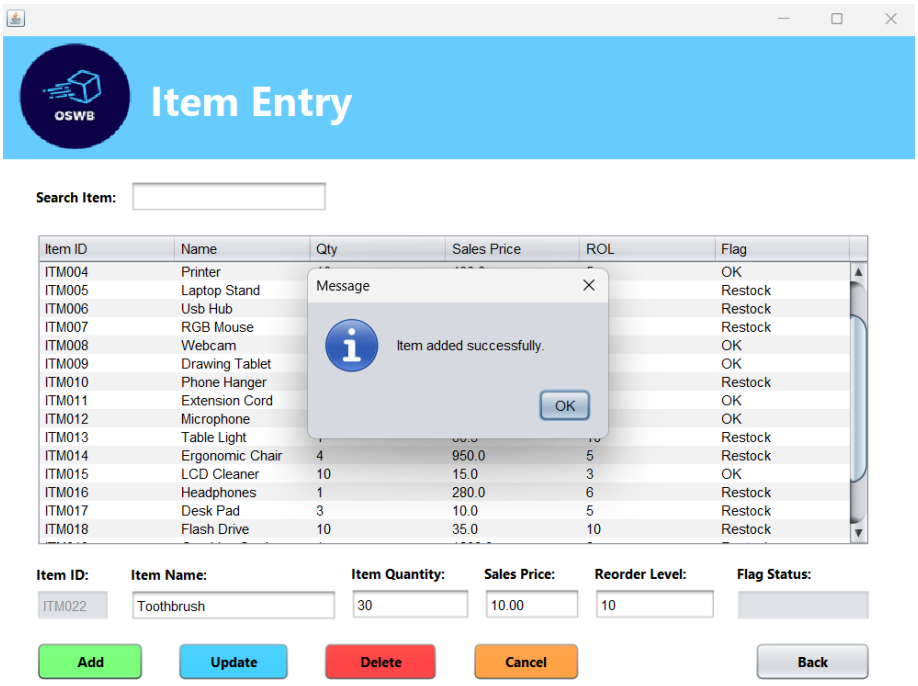



**Figure 5.4.5.2 Add New Item Entry**

Figure 5.4.5.2 shows the message of the item added successfully when the sales manager and purchase manager clicked on the "Add" button. If all the required fields are filled in, the item will be successfully added to the item list.

### 5.4.3 Update Existing Items

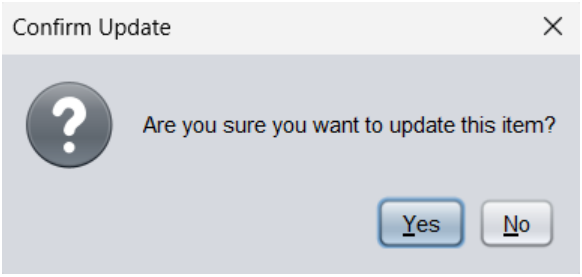



**Figure 5.4.3.1 Item Update Confirmation**

Figure 5.4.3.1 shows the message of update confirmation. If the users select any one of the items, edits the details of the item and presses the "Update" button, the message of update confirmation will be shown for the sales manager, if the users click on the "Yes" button, the item will be successfully updated in the item list.

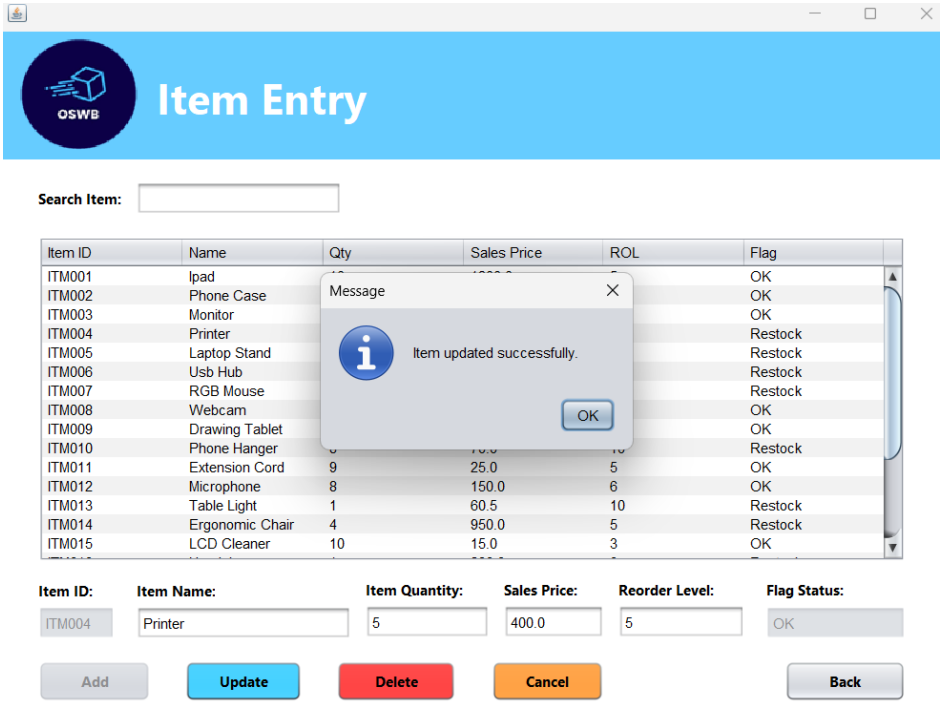



**Figure 5.4.3.2 Item Update Successful Message**

Figure 5.4.3.2 shows the message of the item updated successfully after clicking the "Update" button. If all the edited input fields are valid, the item will be successfully edited in the item list.

### 5.4.4 Delete Existing Items

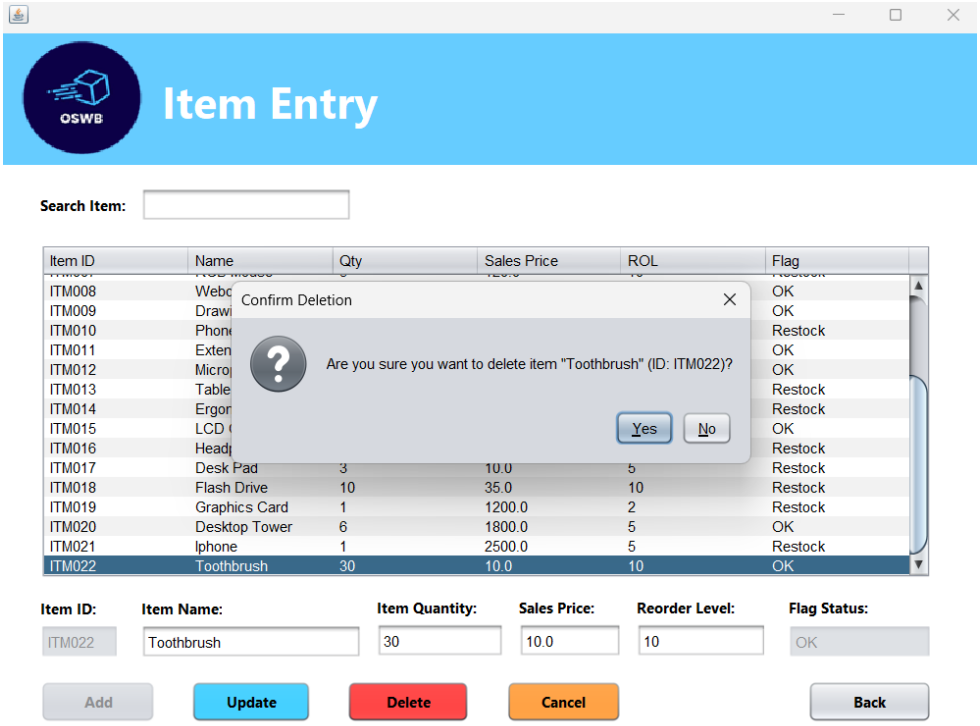



**Figure 5.4.4.1 Item Deletion Confirmation Message**

Figure 5.4.4.1 shows the message of item deletion confirmation. If the users select any one of the items and presses the “Delete” button, the message of delete confirmation will be shown to the sales manager. If the users click on the “Yes” button, the item will be successfully removed from the item list.

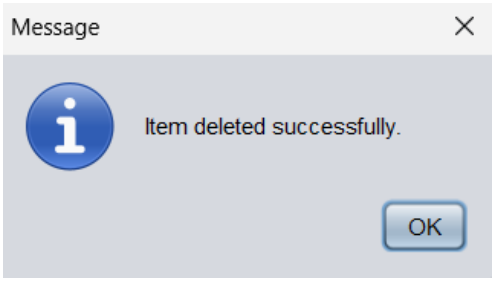



**Figure 5.4.4.2 Item Deletion Successful Message**

Figure 5.4.4.2 shows the message of the item successfully deleted after the users clicked the “Yes” button in the confirmation validation message.

# 5.5 Supplier Entry

## 5.5.1 View Suppliers

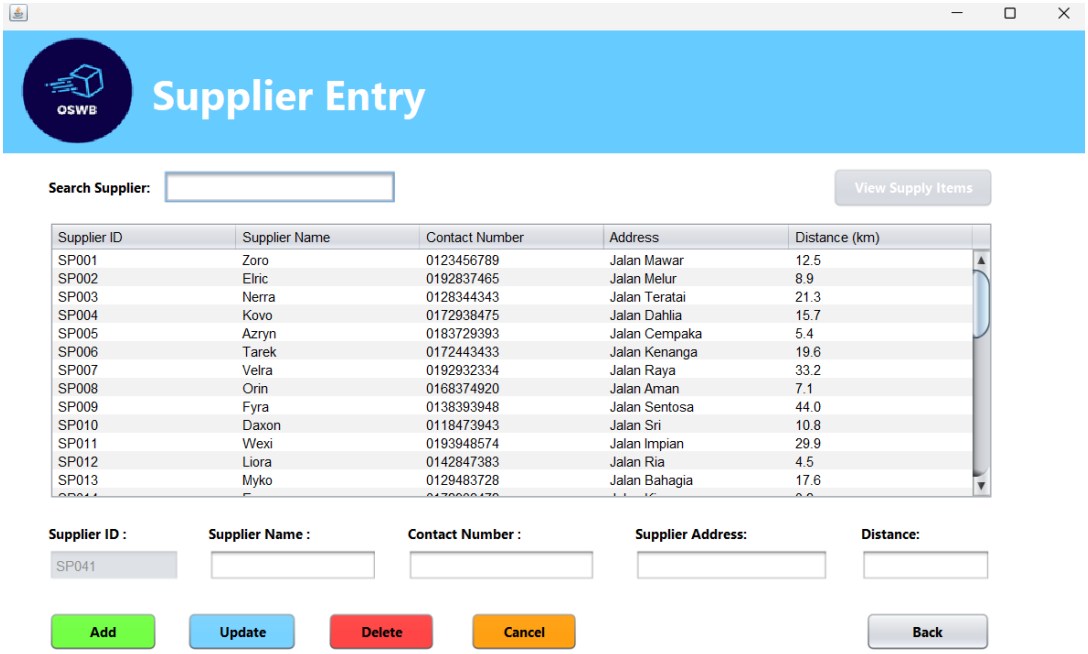



**Figure 5.5.1.1 Supplier Entry Interface**

Figure 5.5.1.1 shows the interface the sales manager and purchase manager will enter once they click on the “Supplier Entry” button. Only the sales manager has the options to add, update and delete suppliers from the suppliers list whereas the purchase manager also is restricted to only viewing the suppliers. If the sales manager wants to add a new supplier, the supplier’s name, contact number, supplier address and distance must be filled before the supplier is added.

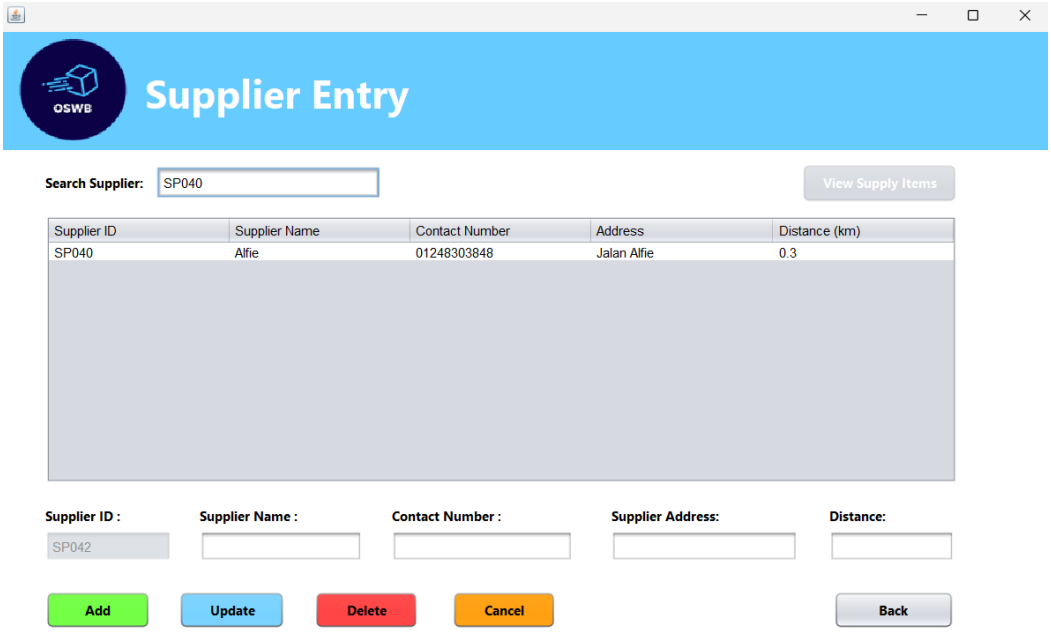



**Figure 5.5.1.2 Supplier Entry Information Prompt**

Figure 5.5.1.2 shows if the users want to search for a certain supplier from the supplier list, the users could type the supplier ID or supplier name in the search bar and if the supplier ID or name exists, the supplier will then be listed below.

## 5.5.2 New Supplier Entry

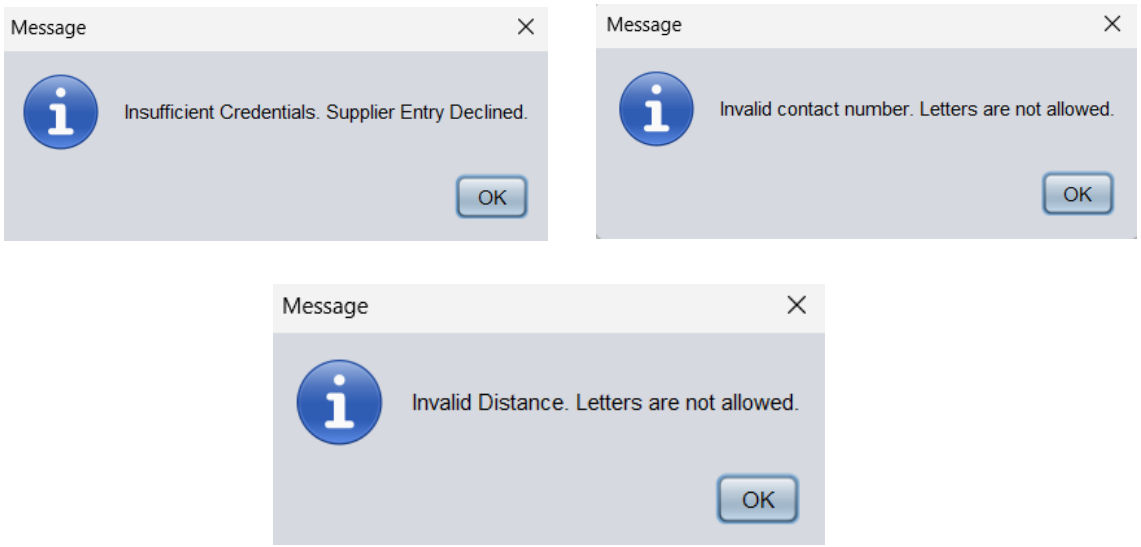



**Figure 5.5.5.1 Error Validation Notification**

Figure 5.5.5.1 shows various error validation notifications for validation purposes. If the users failed to fill in the any of the input field, the message “Insufficient Credentials. Supplier Entry Declined.” will be shown. If the supplier manager tries to enter a character in the Contact Number or Distance input field, the respective input field with the “Letters are not allowed” will be shown.

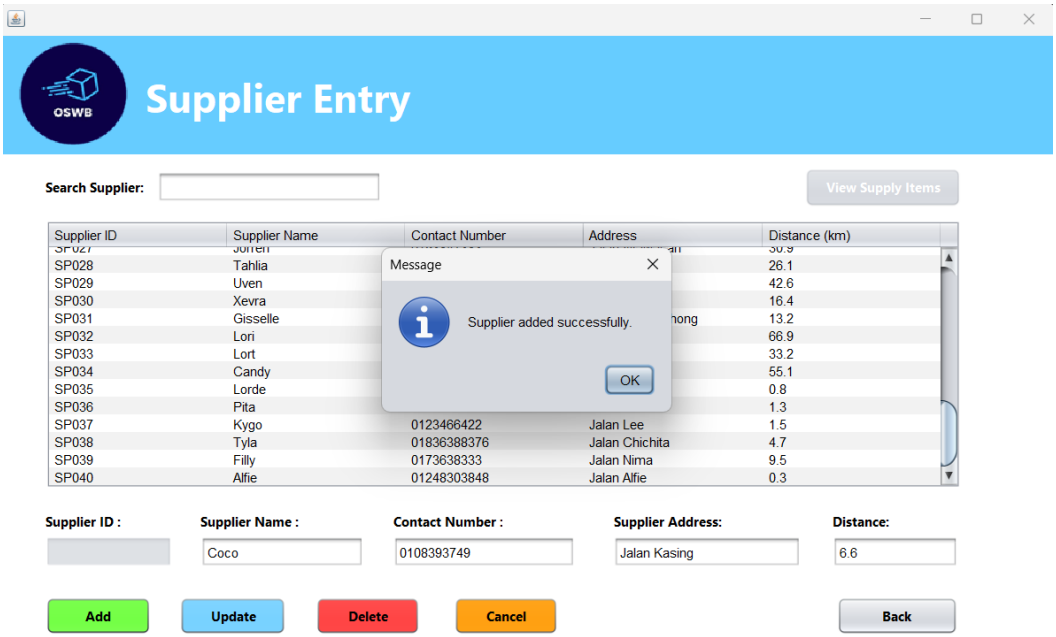



**Figure 5.5.5.2 Add New Supplier Entry**

Figure 5.5.5.2 shows the message of the new supplier added successfully. If all the required fields are filled in, the supplier will be successfully added to the supplier list.

## 5.5.3 Update Existing Supplier

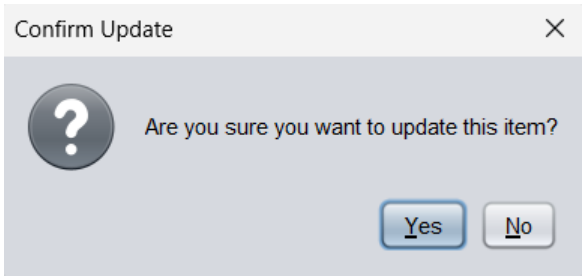



**Figure 5.5.3.1 Supplier Update Confirmation**

Figure 5.5.3.1 shows the message of update confirmation. If the users select any one of the suppliers, edits the details of the supplier and presses the “Update” button, the message of update confirmation will be shown for the users. If the users click on the “Yes” button, the supplier will be successfully updated in the item list.

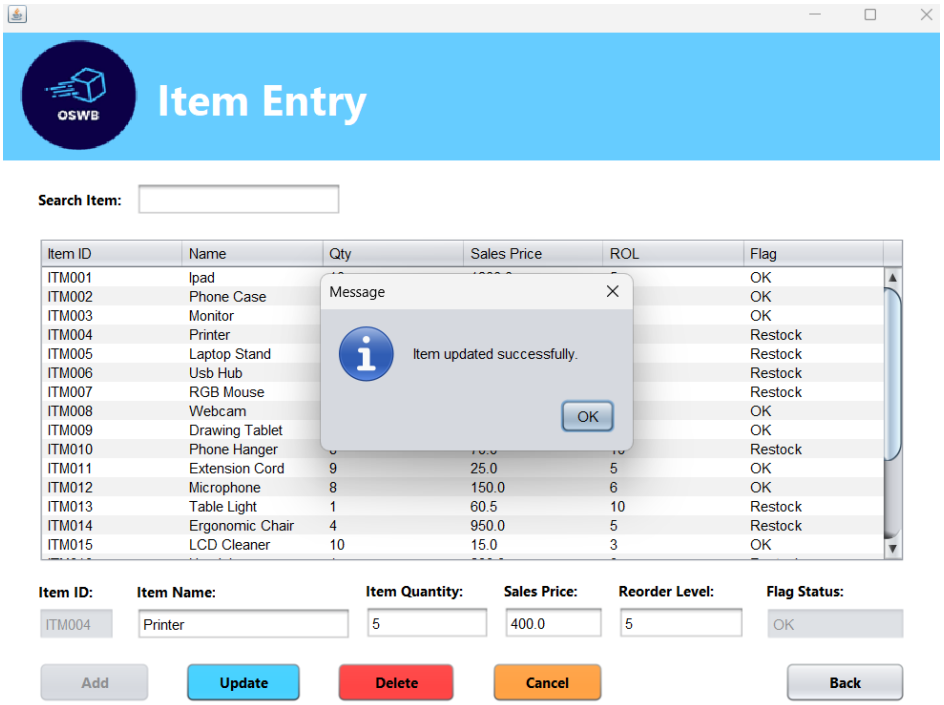



**Figure 5.5.3.2 Supplier Update Successful Message**

Figure 5.5.3.2 shows the message of the supplier updated successfully. If all the edited input fields are valid, the supplier will be successfully updated in the item list.

### 5.5.4 Delete Existing Supplier

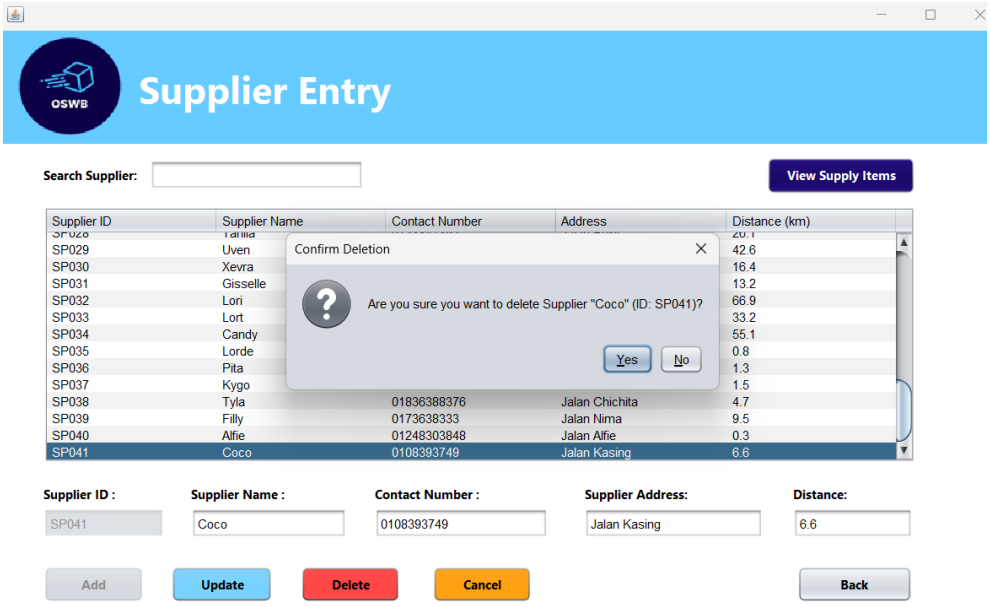



**Figure 5.5.4.1 Supplier Deletion Confirmation Message**

Figure 5.5.4.1 shows the message of supplier deletion confirmation. If the users select any one of the suppliers and presses the “Delete” button, the message of delete confirmation will be shown to the users. If the users click on the “Yes” button, the supplier will be successfully removed from the supplier list.

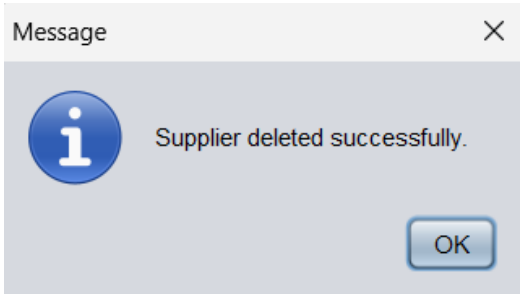



**Figure 5.5.4.2 Supplier Deletion Successful Message**

Figure 5.5.4.2 shows the message of the supplier successfully deleted after the users clicking the “Yes” button in the confirmation validation message.

### 5.5.5 View Supplier's Items

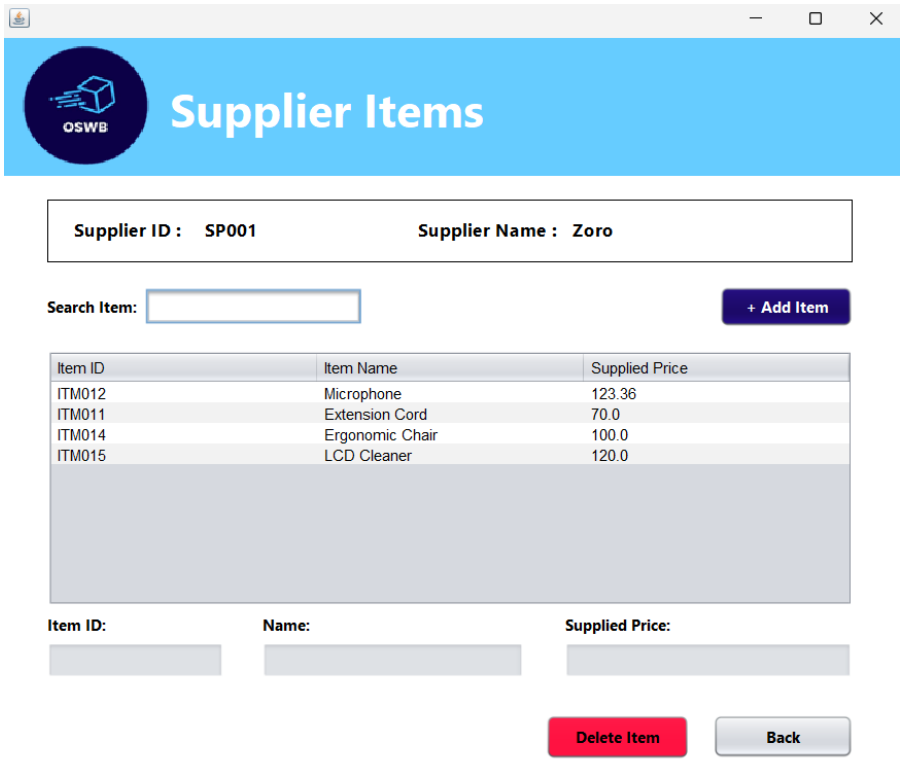



**Figure 5.5.5.1 Supplier's Item Interface**

Figure 5.5.5.1 shows the interface where the sales manager and purchase manager will enter once they click on the "View Supply Items" button. In this interface, the sales manager and purchase manager can view list of items supplied by the supplier. However, only the sales manager can add or delete items from the supplier's item list. If the sales manager wants to delete an item from the supplier's item list, the sales manager can choose an item from the list and click the "Delete" Item button to delete the item.

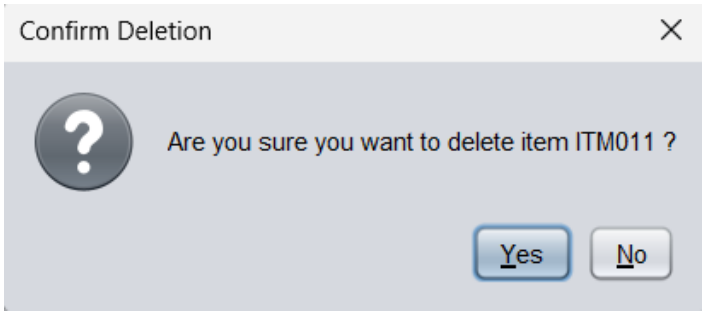



**Figure 5.5.5.2 Supplier's Item Deletion Confirmation**

Figure 5.5.5.2 shows the message of item deletion confirmation. If the sales manager clicks on the “Yes” button, the item will be successfully removed from the supplier’s item list.

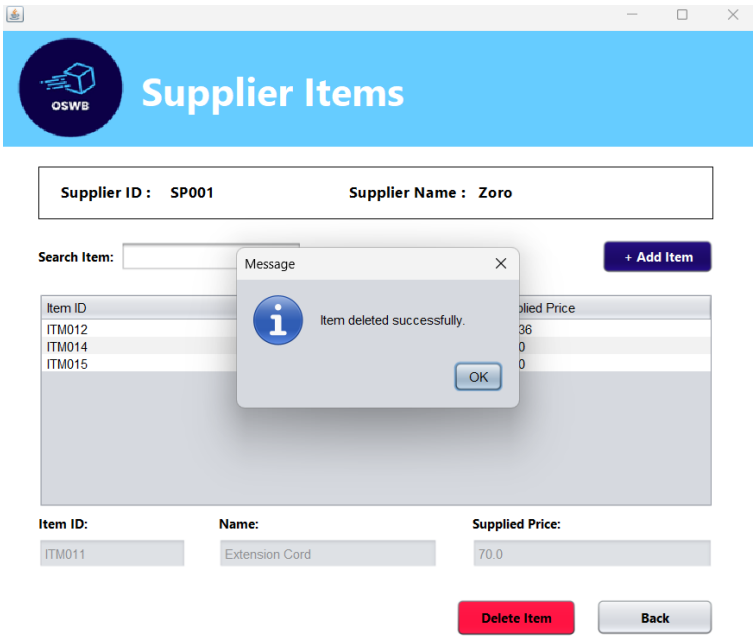



**Figure 5.5.5.3 Supplier’s Item Deletion Successful Message**

Figure 5.5.5.3 shows the message of the supplier’s item successfully deleted after the sales manager clicking the “Yes” button in the confirmation validation message.

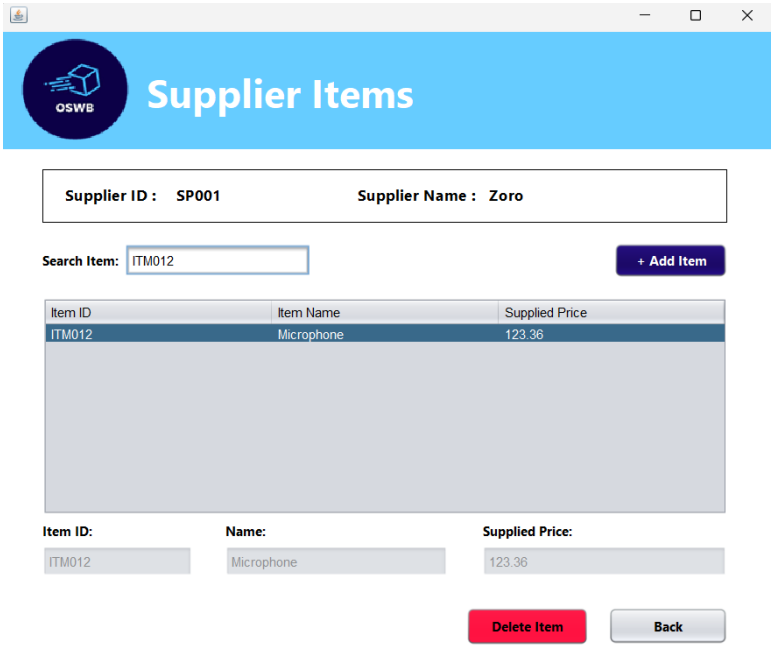



**Figure 5.5.5.4 Supplier’s Item List Information Prompt**

Figure 5.5.5.4 shows if the sales manager wants to search for a certain supplier's item from the supplier's item list, the sales manager could type the item ID or item name in the search bar and if the item ID or name exists, the item will then be listed below.

## 5.5.6 Supplier's Items Addition

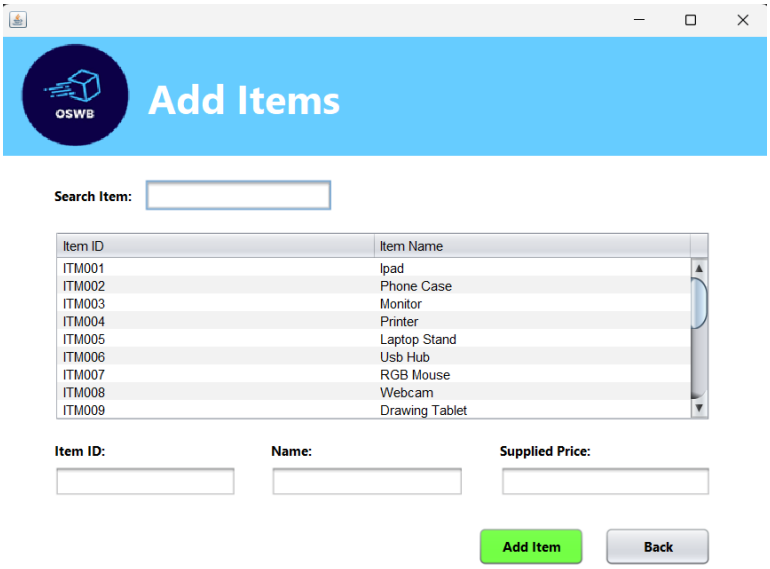



**Figure 5.5.6.1 Supplier Item Addition Interface**

Figure 5.5.6.1 shows the interface only the sales manager will enter once they click on the "Add Item" button. In this interface, the sales manager can view list of items available and they can add items available from the item list. If the sales manager wants to add an item from the system list, the sales manager can choose an item from the list, enter the desired supplied price and click the "Add Item" button to add the item.

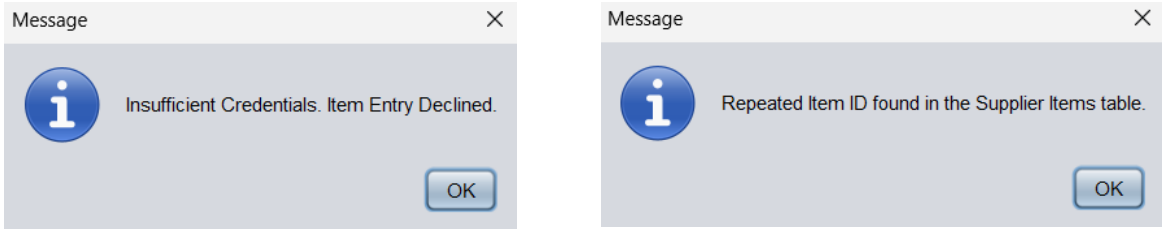



**Figure 5.5.6.2 Error Validation Notification**

Figure 5.5.6.2 shows various error validation notifications for validation purposes. If the sales manager failed to fill in the price in the input field, the message “Insufficient Credentials. Item Entry Declined.” will be shown. If the supplier manager tries to add an item that has already existed in the supplier’s item list table, the input field with the “Repeated Item ID found in the Supplier Items table” will be shown.

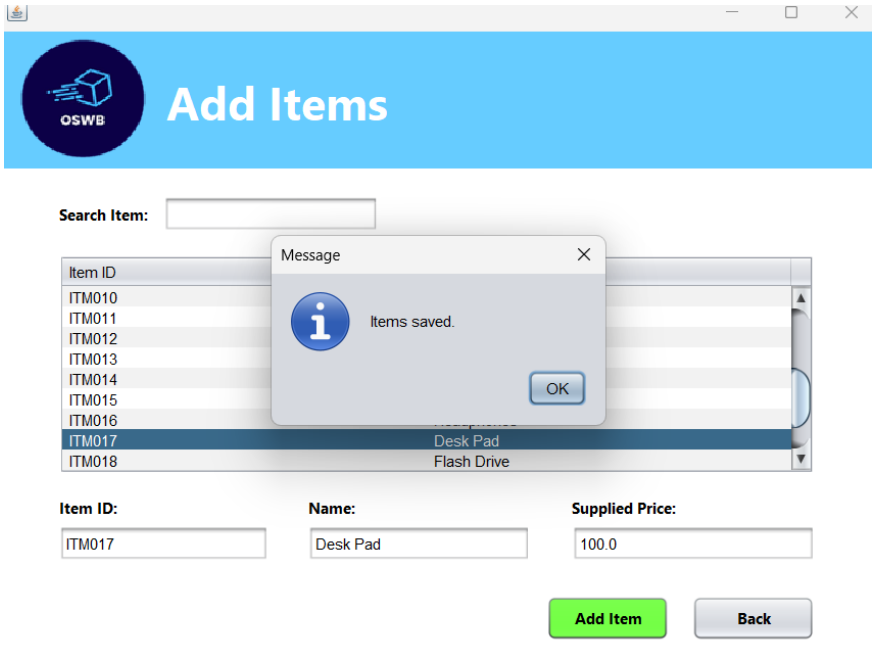



**Figure 5.5.6.2 Item Addition Successful Message**

Figure 5.5.6.2 shows the message of the supplier’s item was added successfully. If the price input fields are valid, the item will be successfully added to the item list.

# 5.6 Daily Sales Entry

## 5.6.1 View Daily Sales Entry

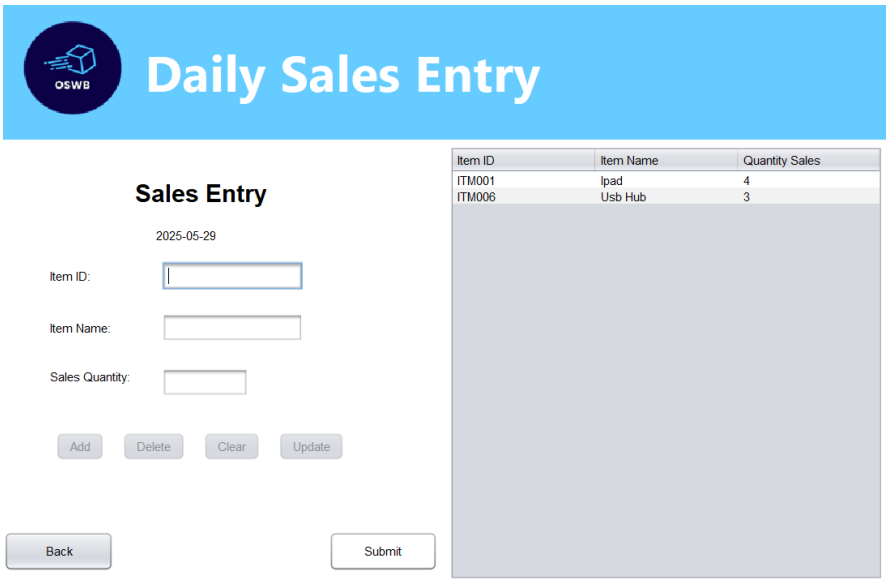



**Figure 5.6.1.1 Daily Sales Entry Interface**

Figure 5.6.1.1 shows the interface the sales manager will enter once they click on the “Daily Sales Entry” button. In this interface, the sales manager can view the daily item sales of the management system with details including the Item ID, Item Name and Quantity Sales. The sales manager also has the options to add, update and delete daily item sales.

## 5.6.2 Add Daily Sales Entry

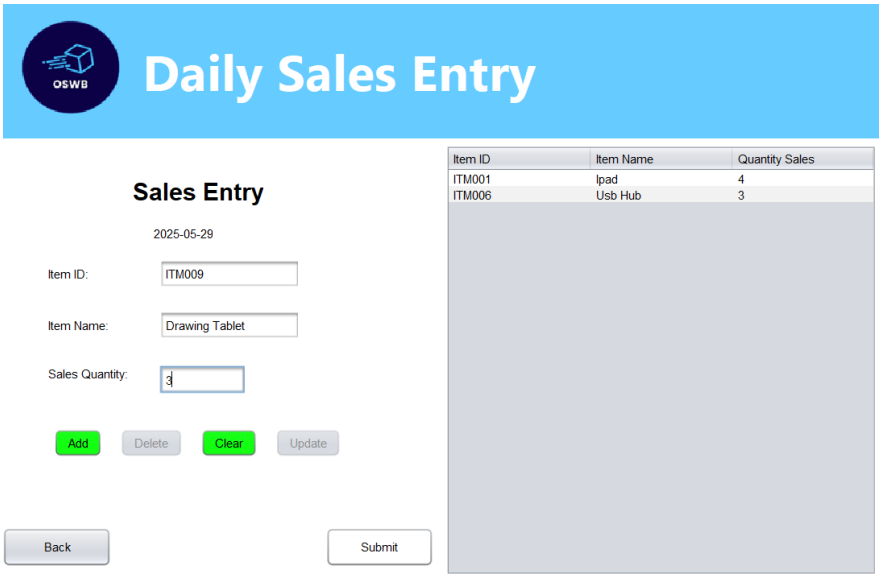



**Figure 5.6.5.1 Adding Daily Sales Entry**

Figure 5.6.5.1 shows that if the sales manager wants to add a new daily sales entry, the Item ID must be filled correctly so that the Item Name of the Item ID will show up alongside it. The sales manager then also needs to fill the sales quantity input field before clicking the "Add" button.

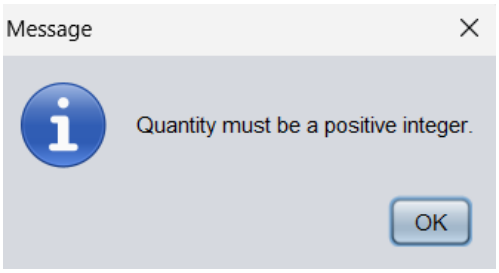



**Figure 5.6.5.2 Error Validation Notification**

Figure 5.6.5.2 shows various error validation notifications for validation purposes. If the sales manager filled an invalid data into the quantity input field, the message "Quantity must be a positive integer." will be shown.

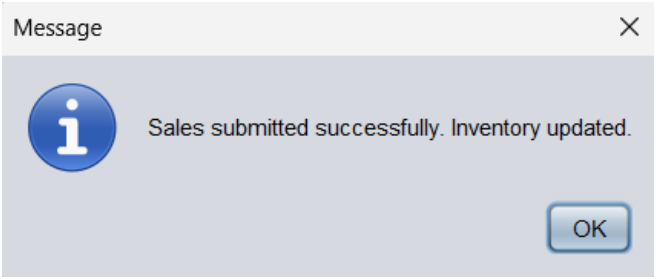



**Figure 5.6.5.3 Item Sales Successfully Added**

Figure 5.6.5.3 shows that the item added is successfully submitted into the inventory after the sales manager clicked the “Submit” button.

## 5.6.3 Delete Daily Sales Entry

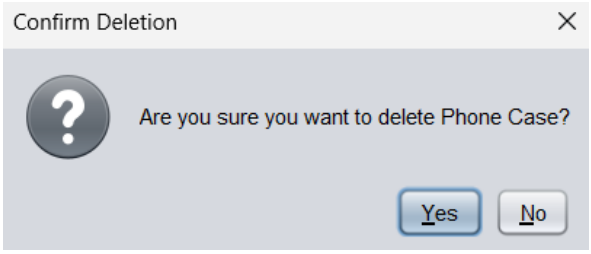



**Figure 5.6.3.1 Item Daily Sales Deletion Confirmation Message**

Figure 5.6.3.1 shows the message when the users click on the “Delete” button clicking on one of the item sales in the item sales table. After clicking on the button, a confirmation notification will pop up on the screen notifying the users to confirm on the deletion of the item daily sales entry. The sales manager who presses the “Yes” button will then delete the item sales from the table.

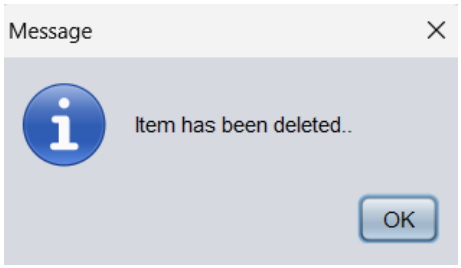



**Figure 5.6.3.2 Item Daily Sales Deletion Successful Message**

Figure 5.6.3.2 shows the message of the sales manager successfully deleted the item sales from the table after clicking the “Yes” button in the confirmation message.

## 5.6.4 Update Daily Sales Entry

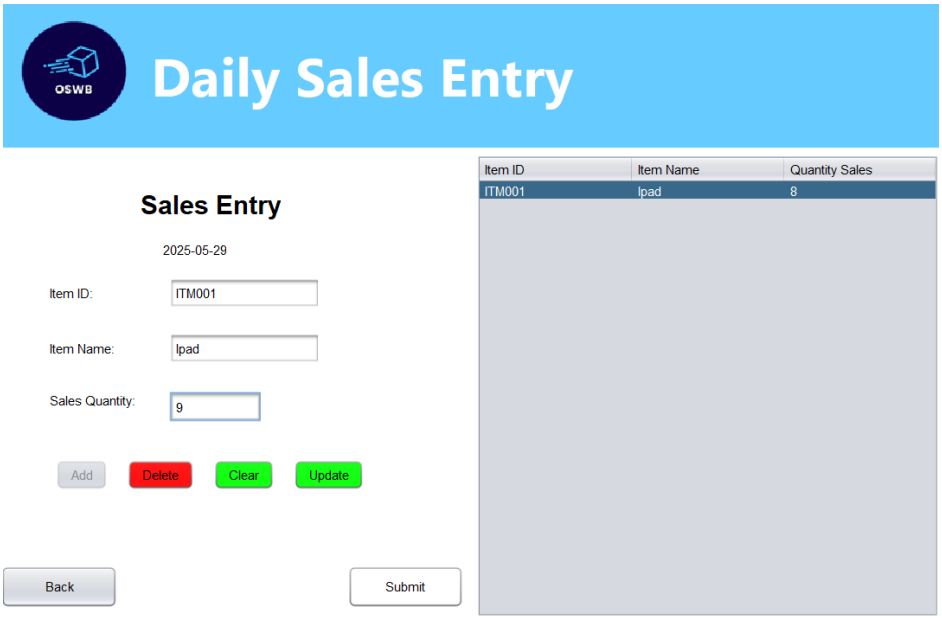



**Figure 5.6.4.1 Item Daily Sales Update**

Figure 5.6.4.1 shows that if the sales manager wants to update any items of the item sales entry table, the sales manager must click on the items in the table and update the latest sales quantity in the input field. Then, the sales manager can click on the “Update” button to save the changes.

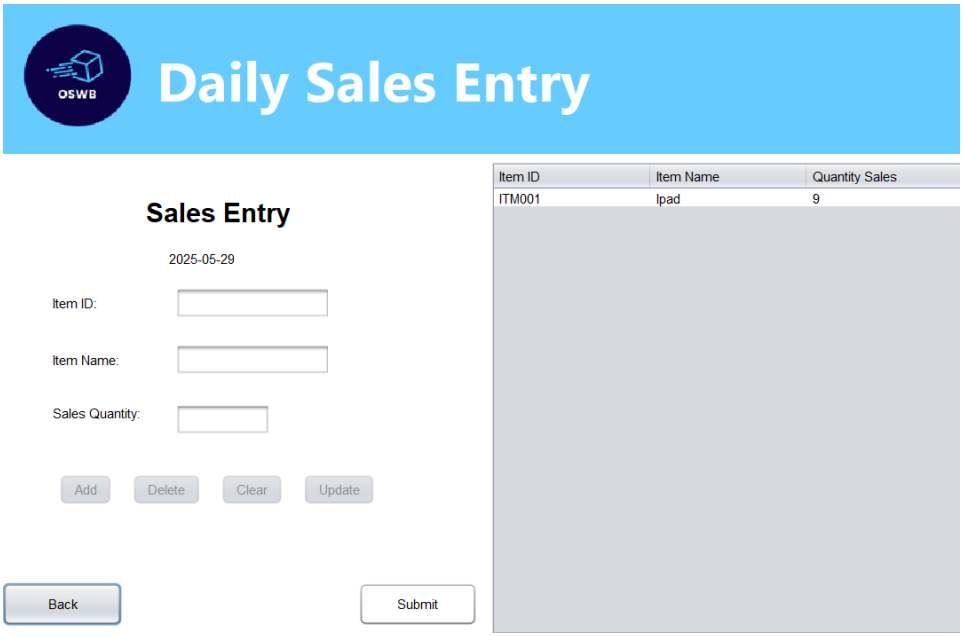



**Figure 5.6.4.2 Successful Item Daily Sales Update**

Figure 5.6.4.2 shows the sales manager successfully updated the item sales from the table after clicking the “Update” button.

# 5.7 Purchase Requisition

## 5.7.1 View Purchase Requisition

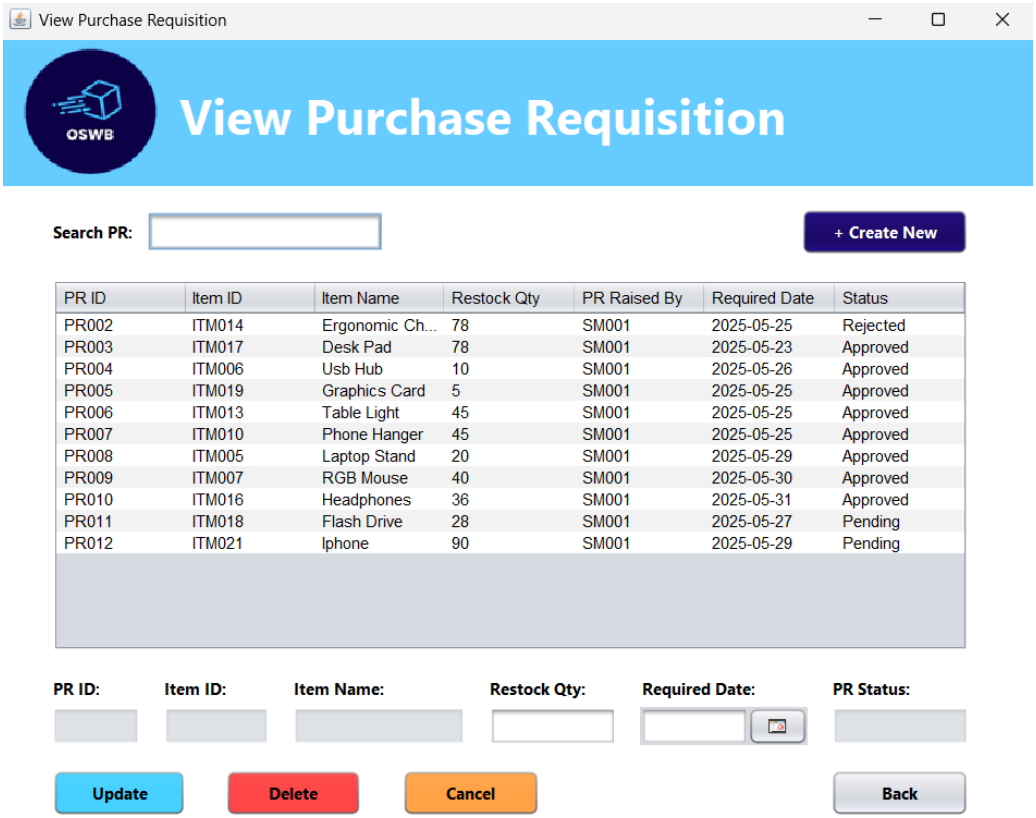



**Figure 5.7.1.1 View Purchase Requisition Interface**

Figure 5.7.1.1 shows the View Purchase Requisition interface where the roles of Admin, Sales Manager, Purchase Manager and Finance Manager have access to. These users will enter this interface once they click on the “Purchase Requisition” button in their home menu. In this interface, the users can view PR details like PR ID, Item ID, Item Name, Restock Quantity, PR Raised By, Required Date and Status. However, only the Sales Manager have the options to update or delete purchase requisitions from the purchase requisition list by clicking on any of the purchase requisitions from the list and then clicking on the respective buttons below to save the changes.

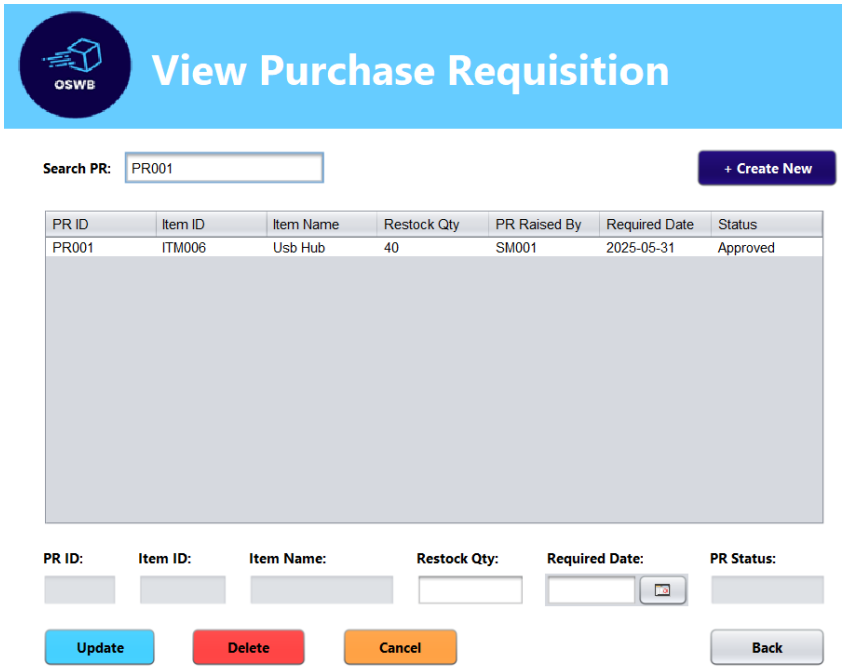
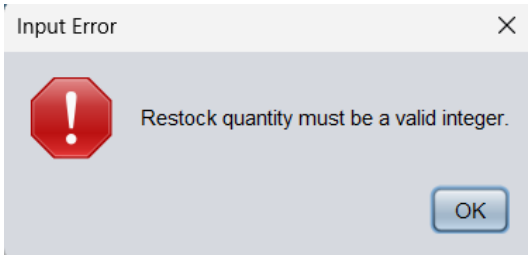
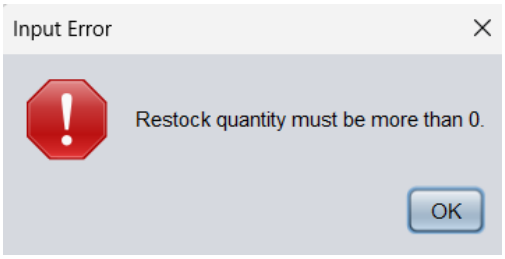



**Figure 5.7.1.2 Purchase Requisition Entry Information Prompt**

Figure 5.7.1.2 shows if the users want to search for a certain purchase requisition from the PR list, the users could type the PR ID or item ID in the search bar and if the ID exists, the PR will then be listed below.

## 5.7.2 Update Purchase Requisition



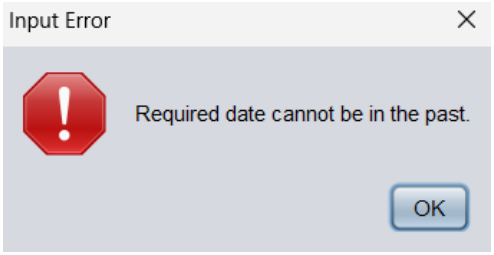



**Figure 5.7.5.1 Error Message**

Figure 5.7.5.1 shows the error message when the sales manager entered characters into the restock quantity input field or have selected a date from the past when trying to update the purchase requisition.

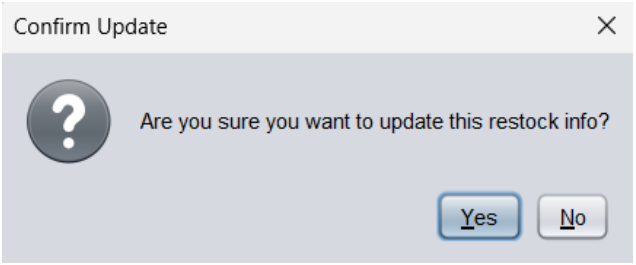



**Figure 5.7.5.2 Purchase Requisition Update Confirmation**

Figure 5.7.5.2 shows the interface when the users click on the “Update” button after updating the restock quantity and required date. After clicking on the button, a “Are you sure you want to update this restock info?” notification will pop up on the screen notifying the users to confirm on the update restock action. Users who press the “Yes” button will update the restock info and save it back to the purchase requisition list.

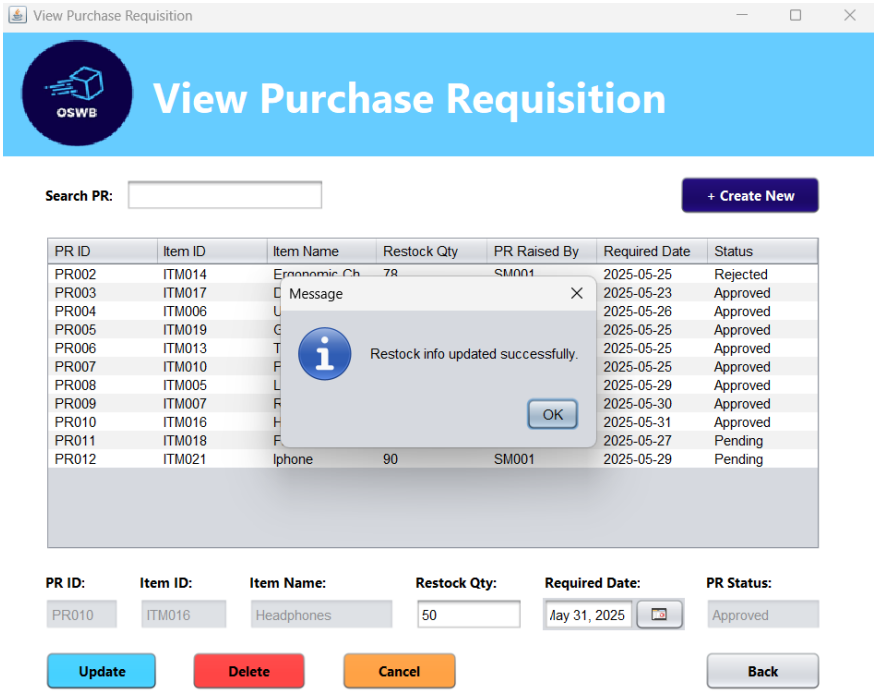


**Figure 5.7.5.3 Purchase Requisition Update Successful Message**

Figure 5.7.5.3 shows the message of the users successfully updated after clicking the “Yes” button in the confirmation validation message.

## 5.7.3 Delete Purchase Requisition

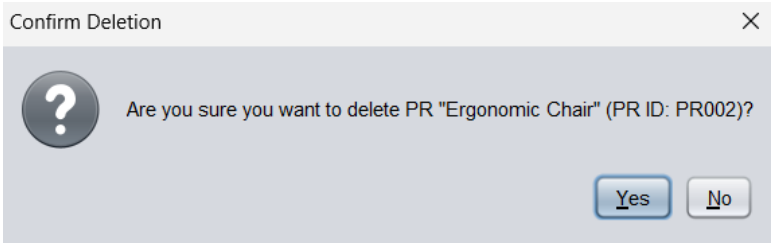



**Figure 5.7.3.1 Purchase Requisition Delete Confirmation**

Figure 5.7.3.1 shows the interface when the users click on the “Delete” button after selecting a purchase requisition. After clicking on the button, a confirmation notification will pop up on the screen notifying the users to confirm on the Item Name along with the PR ID is the one that is going to be deleted. Users who press the “Yes” button will delete the purchase requisition from the purchase requisition list.

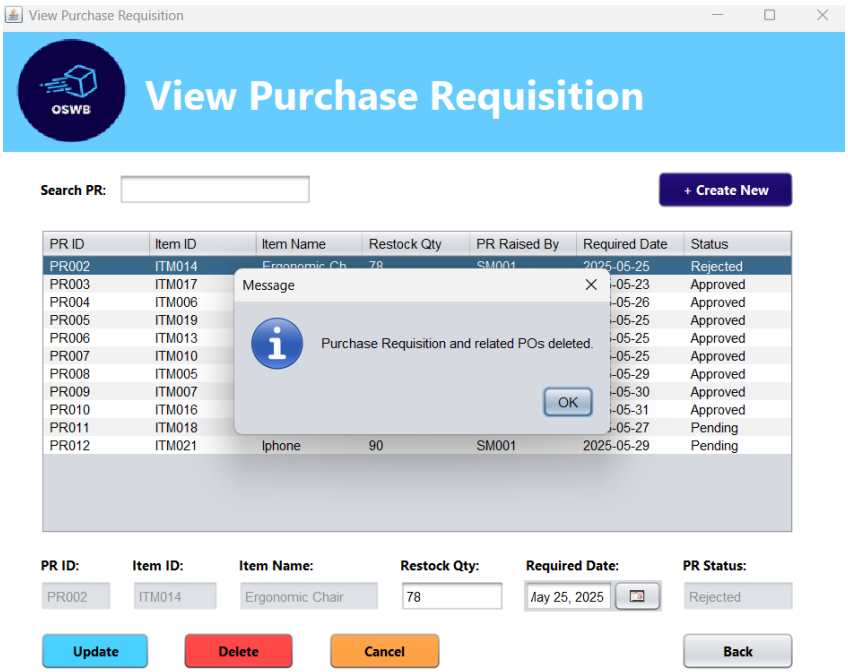


**Figure 5.7.3.2 Purchase Requisition Deletion Successful Message**

Figure 5.7.3.2 shows the message of the users successfully deleting the purchase requisition after clicking the "Yes" button in the confirmation validation message.

## 5.7.4 Create Purchase Requisition

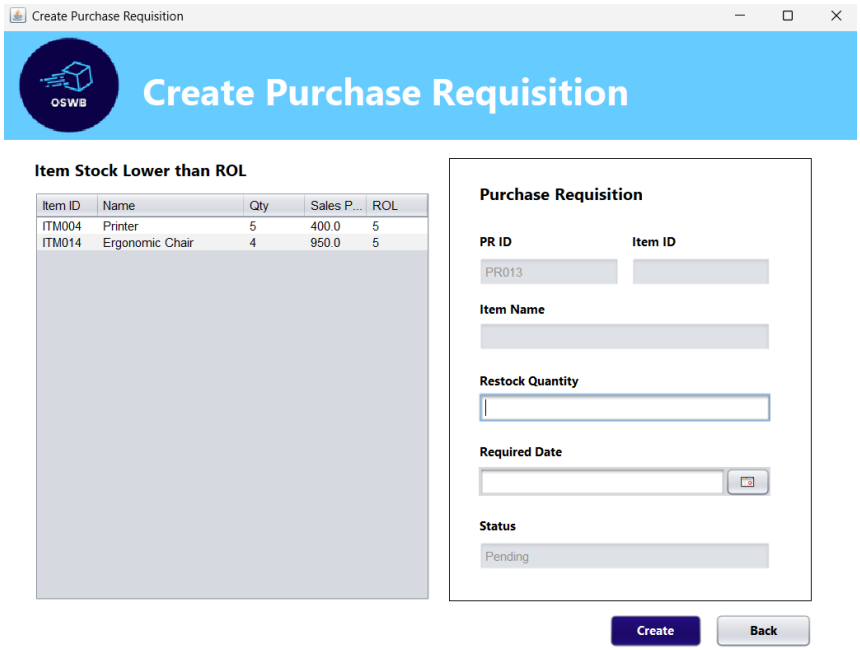



**Figure 5.7.4.1 Create Purchase Requisition Interface**

Figure 5.7.4.1 shows the Create Purchase Requisition interface where the Sales Manager will enter this interface once they click on the "Create Purchase Requisition" button in the View Purchase Requisition interface. In this interface, the sales manager can create purchase requisition by clicking on any one of the items from the item list where their stock is lower than the reorder level (ROL). Sales manager then would have to fill in the restock quantity and required date input field and click on the "Create" button to create the purchase requisition.

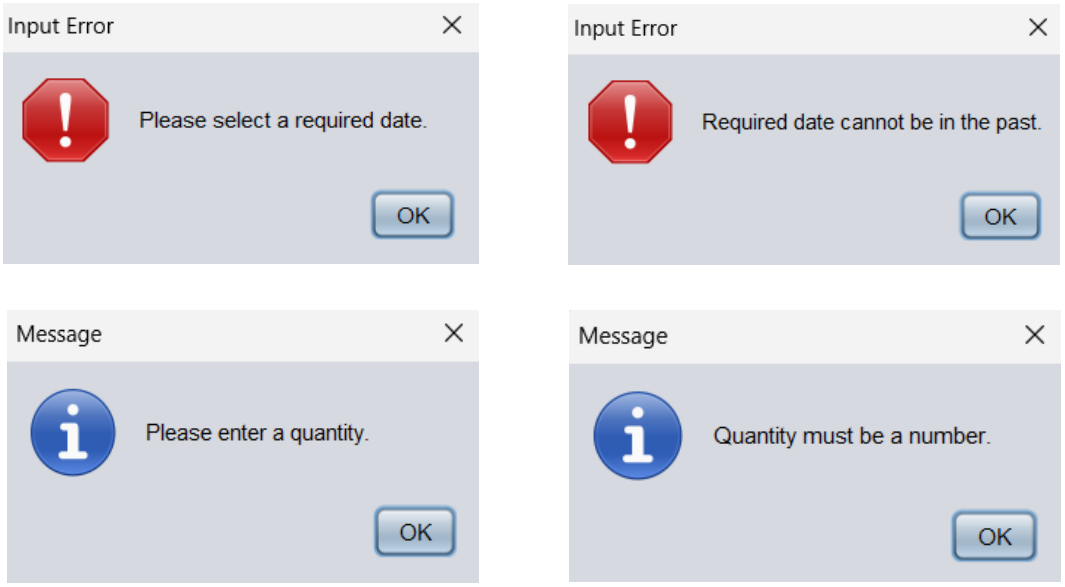



**Figure 5.7.4.2 Validation and Error Confirmation**

Figure 5.7.4.2 shows the validation and error message when the sales manager has entered invalid data into input fields and clicked on the "Create" button. An error message of "Please select a required date" or "Required date cannot be in the past" will pop up when the sales manager did not select a date or have selected a date in the past. A validation confirmation message also pops up when the sales manager did not enter the quantity input field or has entered characters into the quantity input field.

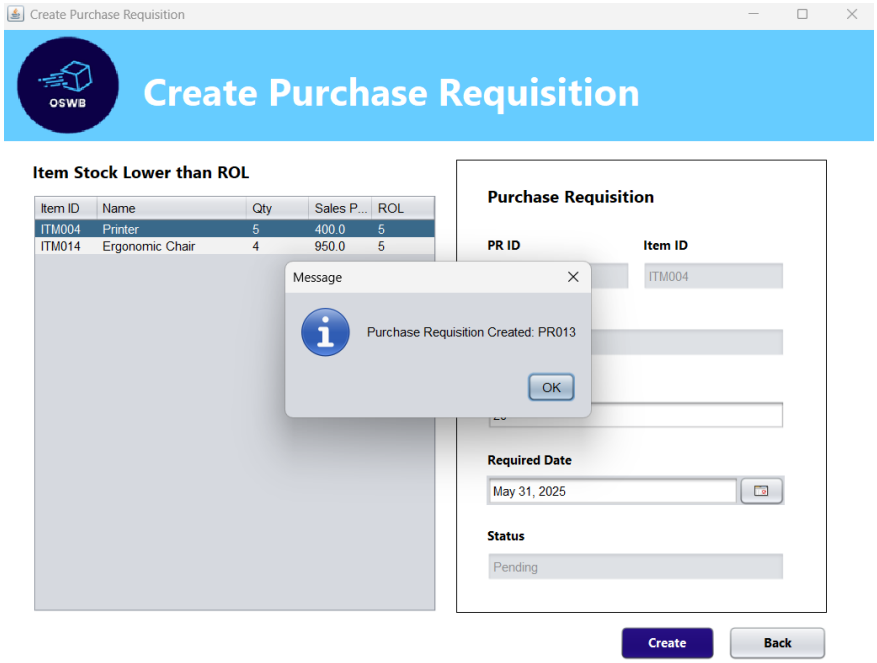


**Figure 5.7.4.3 Purchase Requisition Creation Successful Message**

Figure 5.7.4.3 shows the message of the sales manager successfully created the purchase requisition after clicking the "Create" button.

# 5.8 Purchase Orders

## 5.8.1 View Purchase Orders

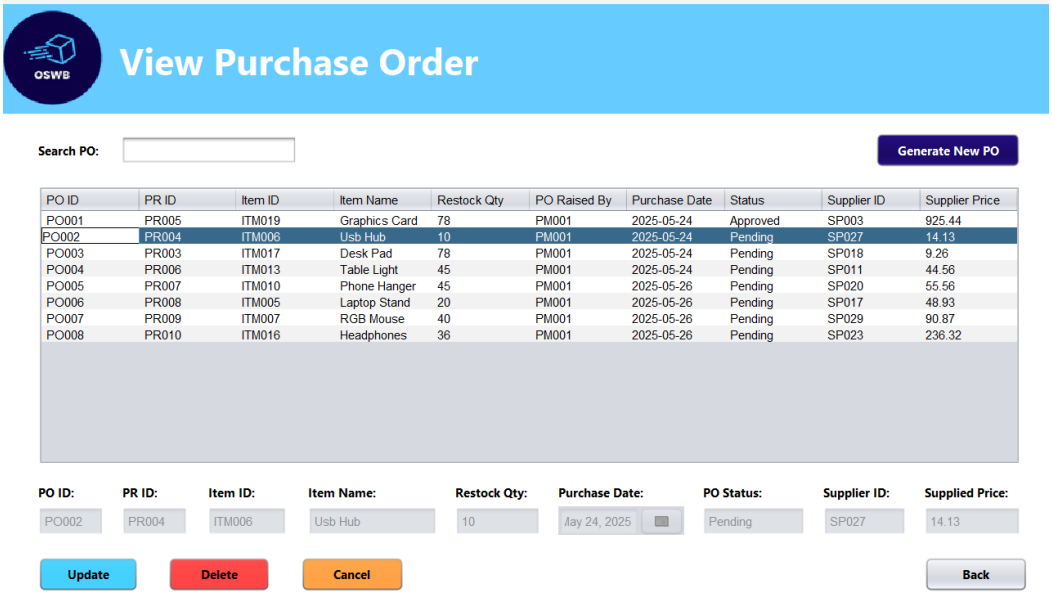



**Figure 5.8.1.1 View Purchase Order Interface**

Figure 5.8.1.1 shows the View Purchase Order interface where the roles of Admin, Sales Manager, Purchase Manager, Finance Manager and Inventory Manager have access to. The purchase manager will enter this interface by clicking on the "Generate Purchase Order" button, while others will enter through the "View Purchase Order" button in their home menu. In this interface, the users can view PO details like PO ID, PR ID, Item ID, Item Name, Restock Quantity, PO Raised By, Purchase Date, Status, Supplier ID and Supplier Price by clicking on any one of the purchase orders in the purchase table. However, only the purchase managers have the option to update and delete the purchase orders, whereas the other users are only restricted to viewing the list.

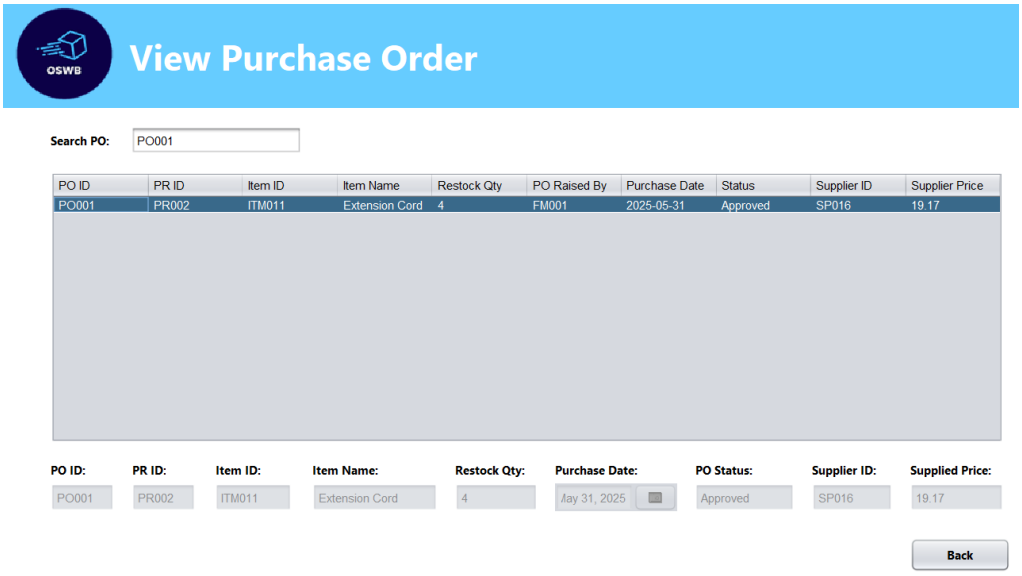



**Figure 5.8.1.2 Purchase Order Entry Information Prompt**

Figure 5.8.1.2 shows if the users want to search for a certain purchase order from the PR list, the users could type the PO ID, PR ID or Item ID in the search bar and if the ID exists, the PR will then be listed below.

## 5.8.2 Update Purchase Orders

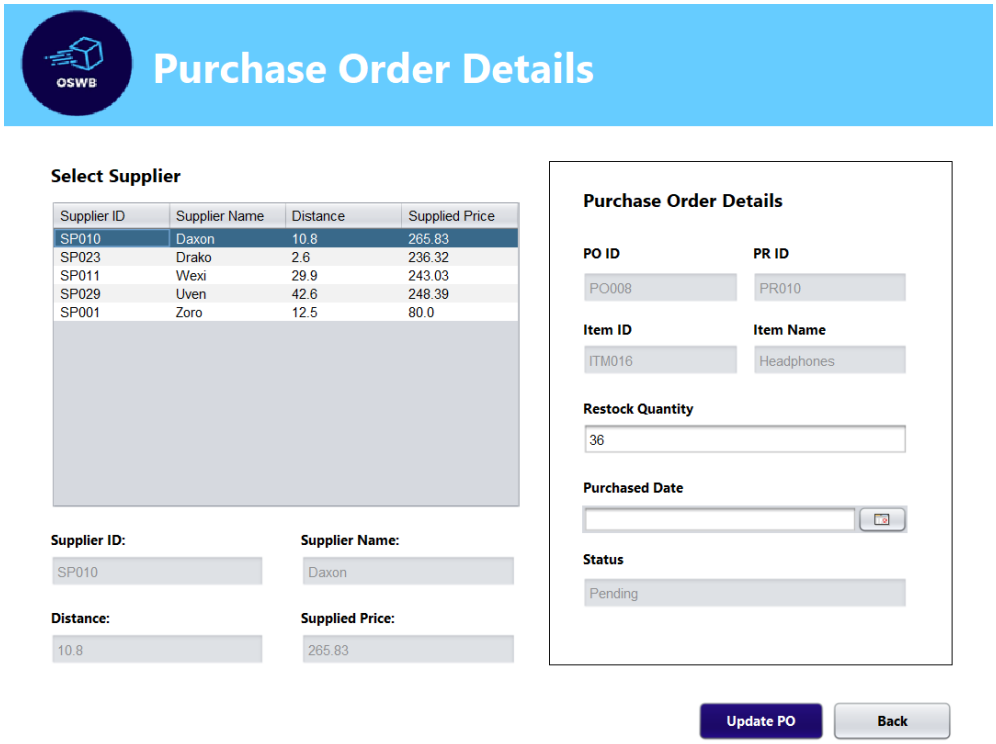
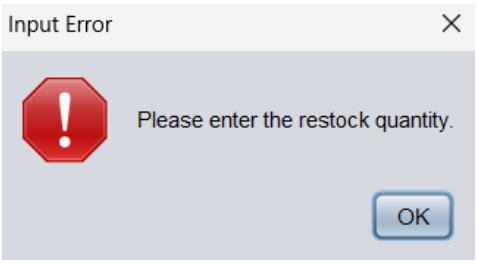
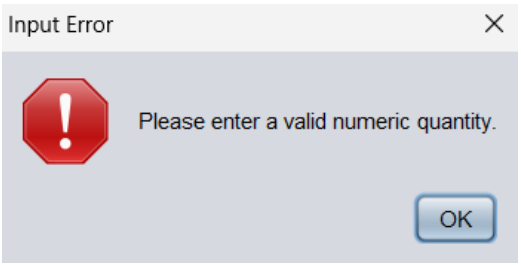



**Figure 5.8.5.1 Update Purchase Order Interface**

Figure 5.8.5.1 shows the Update Purchase Order interface where the purchase manager will enter this interface once they click on the "Update" button in the View Purchase Requisition interface. Meanwhile, the finance manager can also enter this interface by clicking the "Edit PO" button in the PO Approval interface. In this interface, the users can update PO by selecting other suppliers from the suppliers table for the item chosen. Users can also update the restock quantity and purchased date by filling in the details into the input fields. Users can click the "Update PO" button to update the purchase order.



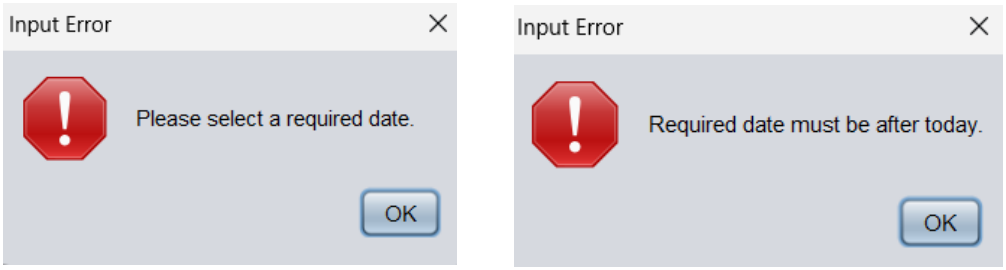



**Figure 5.8.5.3 Error Confirmation**

Figure 5.8.5.3 shows error messages when the users have entered invalid data into input fields and clicked on the "Update PO" button. Error messages of "Please enter the stock quantity" or "Please enter a valid numeric quantity" will pop up when the users misfiled the quantity input field. An error message of "Please select a required date" or "Required date must be after today" will pop up when the users did not select a date or have selected a date in the past.

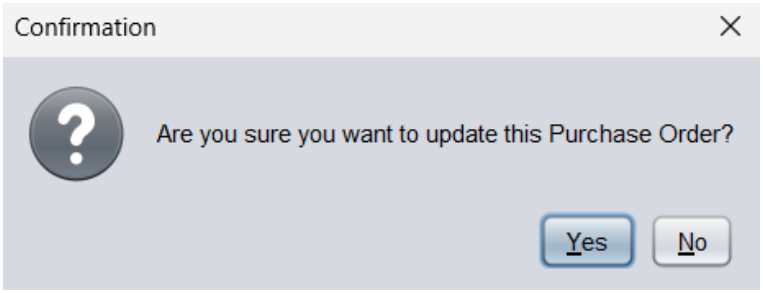



**Figure 5.8.5.4 Update Purchase Order Confirmation Message**

Figure 5.8.5.4 shows the interface when the users click on the "Update PO" button after updating the purchase order details. After clicking on the button, a confirmation notification will pop up on the screen notifying the users to confirm on the update of the purchase order. Users who press the "Yes" button will update the details in the purchase order.

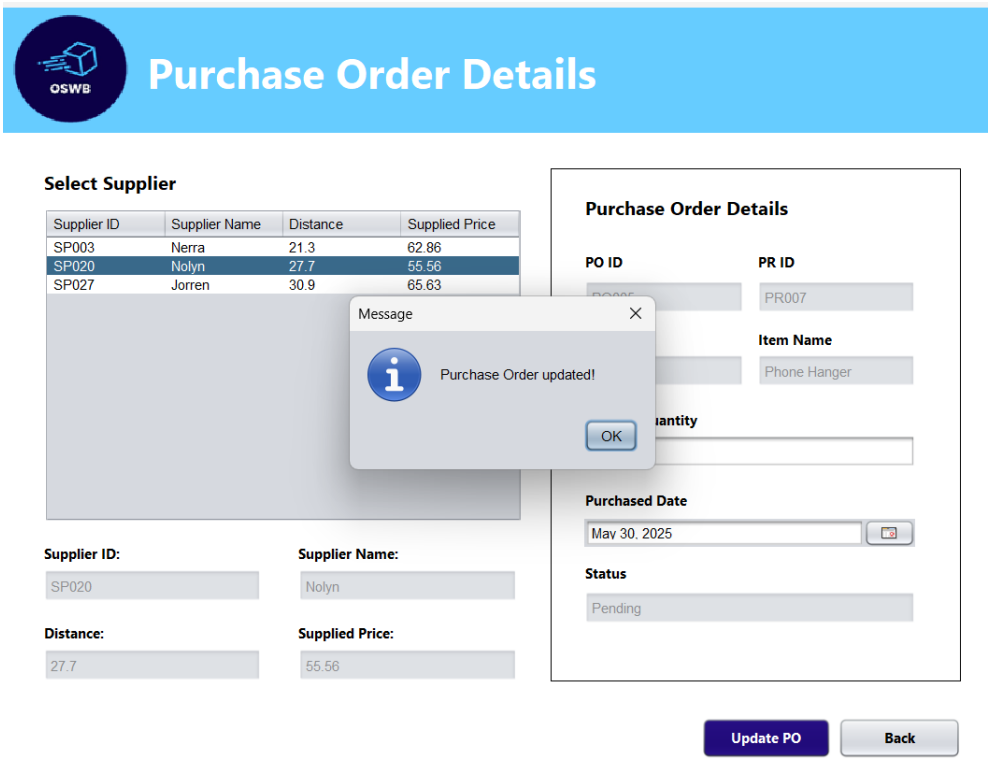



**Figure 5.8.5.5 Update Purchase Order Successful Message**

Figure 5.8.5.5 shows the message of the users successfully updated the details of the purchase order after clicking the “Yes” button in the confirmation message.

## 5.8.3 Delete Purchase Orders

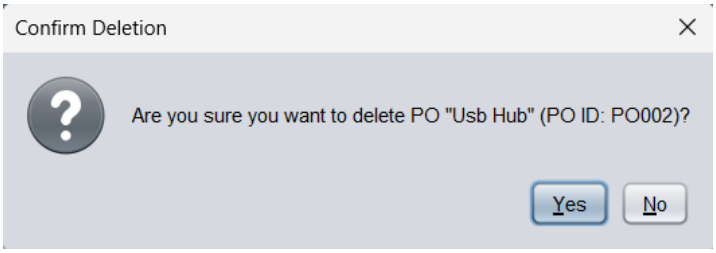



**Figure 5.8.3.1 Purchase Requisition Delete Confirmation**

Figure 5.8.3.1 shows the interface when the users click on the “Delete” button after selecting a purchase order. After clicking on the button, a confirmation notification will pop up on the screen notifying the users to confirm on the Item Name along with the PO ID is the one that is going to be deleted. Users who press the “Yes” button will delete the purchase order from the purchase order list.

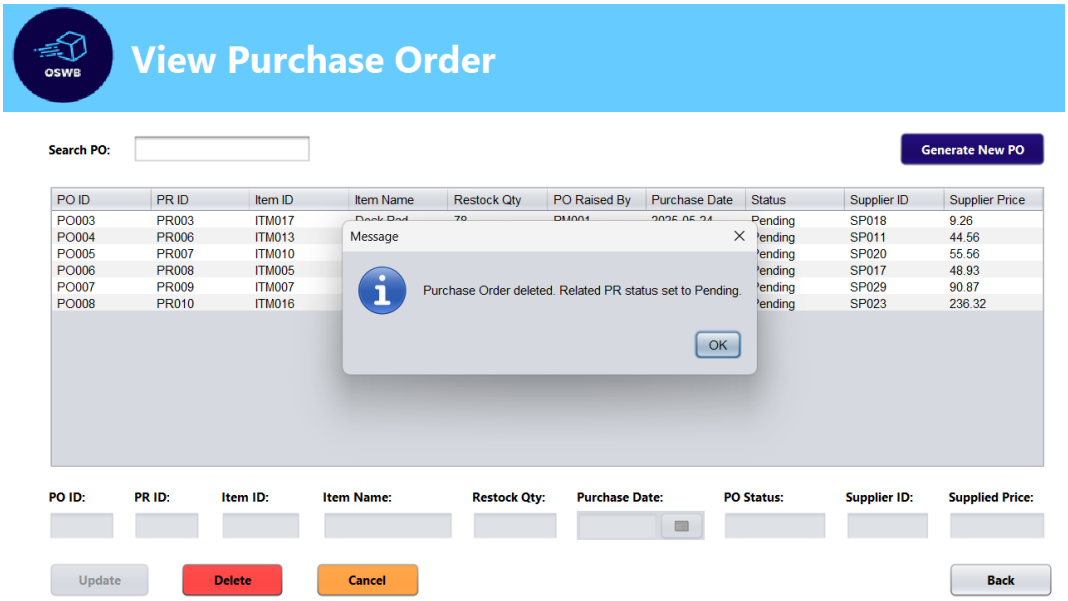



**Figure 5.8.3.2 Purchase Order Deletion Successful Message**

Figure 5.8.3.2 shows the message of the users successfully deleted the purchase order after clicking the "Yes" button in the confirmation validation message.

## 5.8.4 Purchase Requisition Approval

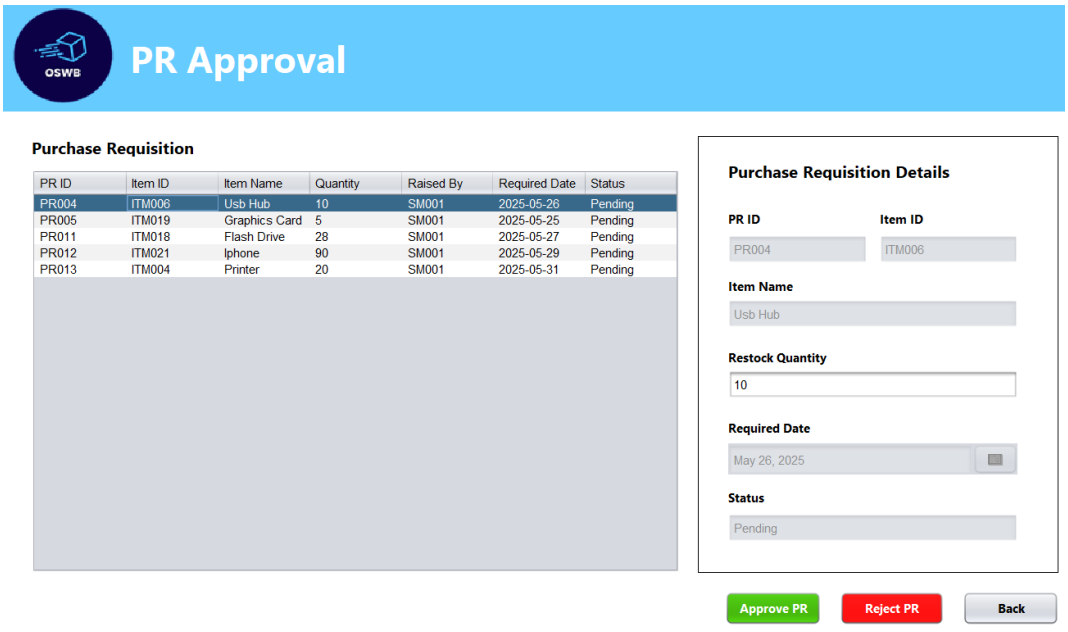



**Figure 5.8.4.1 PR Approval Interface**

Figure 5.8.4.1 shows the Purchase Requisition Approval interface where the purchase manager will enter this interface once they click on the "Generate New PO" button in the View Purchase Order interface. In this interface, the users can approve or reject PR by selecting any of the purchase requisitions from the table and clicking on the "Approve PR" or "Reject PR" button.


**Figure 5.8.4.2 Purchase Requisition Approval Reject Confirmation**

Figure 5.8.4.2 shows the message when the users click on the "Reject PR" button after selecting a purchase requisition. After clicking on the button, a confirmation notification will pop up on the screen notifying the users to confirm the purchase requisition rejection. Users who press the "Yes" button will reject the purchase requisition approval and be removed from the purchase requisition list.

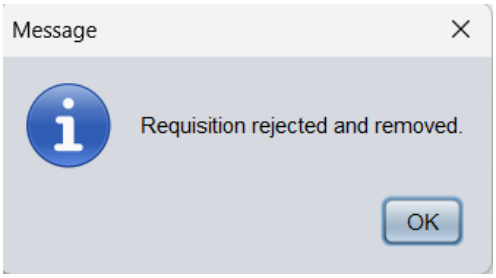



**Figure 5.8.4.3 Purchase Requisition Approval Rejection Successful Message**

Figure 5.8.4.3 shows the message of the users successfully rejected the approval of the purchase requisition after clicking the "Yes" button in the confirmation message.

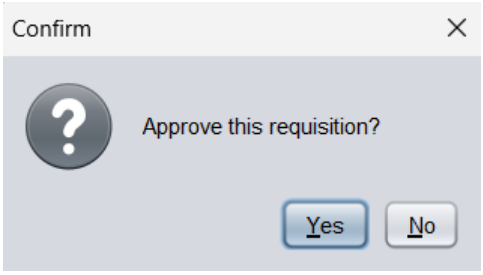



**Figure 5.8.4.4 Purchase Requisition Approval Confirmation**

Figure 5.8.4.4 shows the interface when the users click on the "Approve PR" button after selecting a purchase requisition. After clicking on the button, a confirmation notification will pop up on the screen notifying the users to confirm the purchase requisition approval. Users who press the "Yes" button will approve the purchase requisition approval and led to the purchase order details interface.

## 5.8.5 Generate Purchase Order

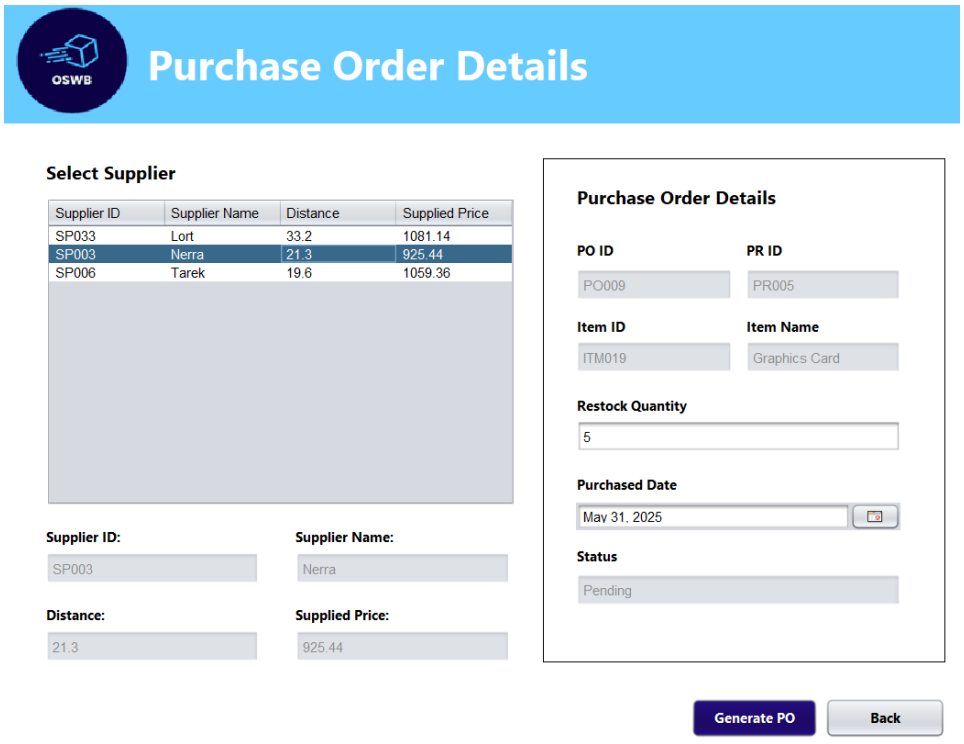



**Figure 5.8.5.1 Purchase Order Details Interface**

Figure 5.8.5.1 shows the Purchase Order Details interface where the purchase manager will enter this interface once they click on the "Yes" button in the Purchase Requisition Approval

Confirmation message. In this interface, the users can generate PO by selecting any of the suppliers of the given item from the table by comparing their distance and supplied price. Users will then need to select the purchased date from the calendar and have the option to edit the restock quantity before clicking on the "Generate PO" button.

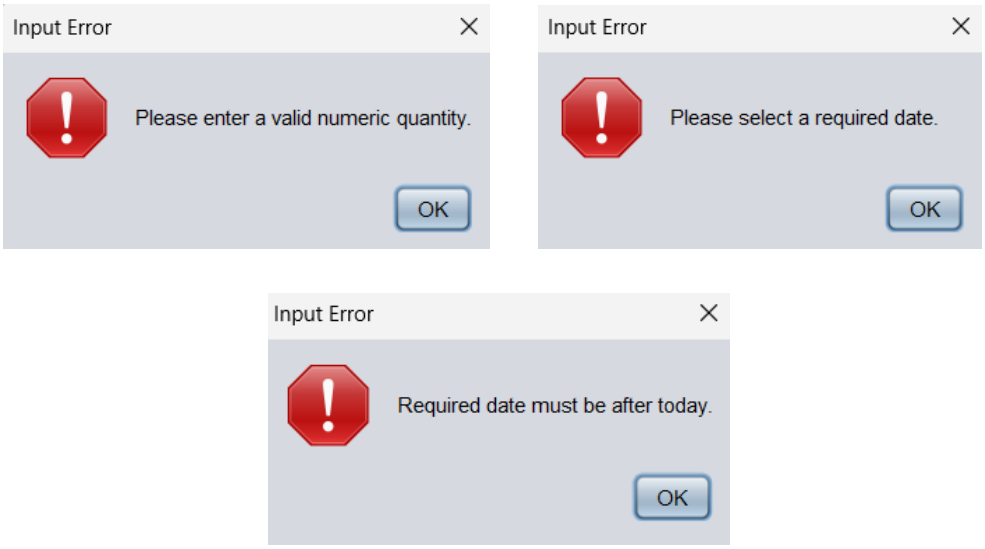



**Figure 5.8.5.2 Error Confirmation**

Figure 5.8.5.2 shows error messages when the users have entered invalid data into input fields and clicked on the "Generate PO" button. Error messages of "Please enter a valid numeric quantity" will pop up when the users misfiled the quantity input field. An error message of "Please select a required date" or "Required date must be after today" will pop up when the users did not select a date or have selected a date in the past.

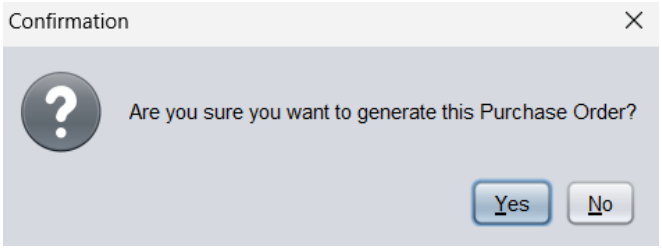



**Figure 5.8.5.3 Purchase Order Generation Confirmation**

Figure 5.8.5.3 shows the message when the users click on the "Generate PO" button. After clicking on the button, a confirmation notification will pop up on the screen notifying the users

to confirm on generating a new purchase order. Users who press the “Yes” button will then generate a new purchase order.

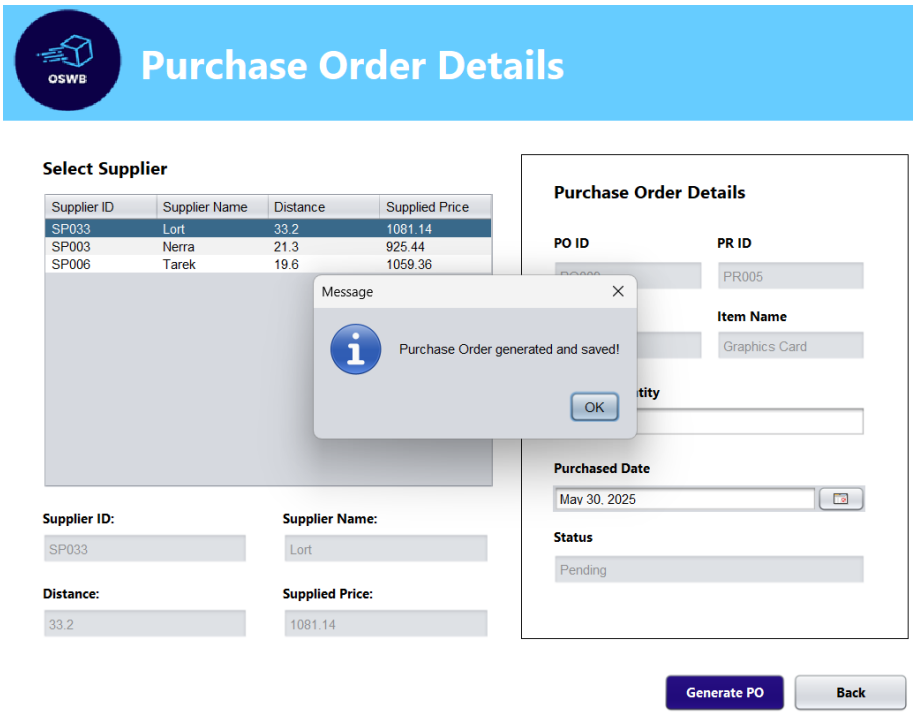



**Figure 5.8.5.4 Purchase Order Generation Successful Message**

Figure 5.8.5.4 shows the message of the users successfully generated the purchase order after clicking the Yes button in the confirmation validation message.

## 5.8.6 Purchase Order Approval

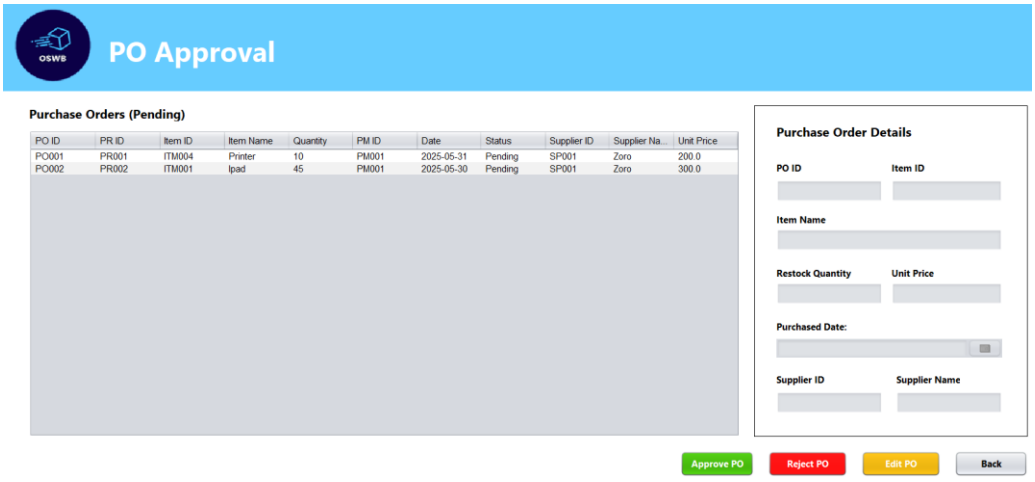



**Figure 5.8.6.1 PO Approval Interface**

Figure 5.8.6.1 shows the Purchase Order Approval interface where only the finance manager will enter this interface once they click on the “PO Approval” button in the home menu. In this interface, the finance manager can approve or reject PO by selecting any of the purchase orders from the table and clicking on the “Approve PO” or “Reject PO” button.

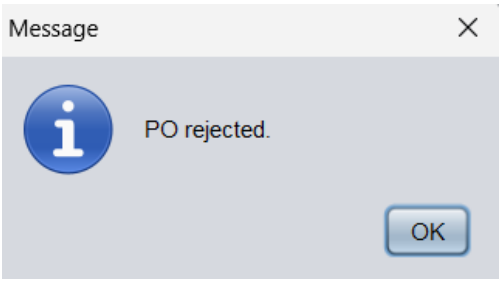



**Figure 5.8.6.2 PO Rejected Message**

Figure 5.8.6.2 shows the message when the finance manager clicks on the “Reject PO” button after selecting a purchase order. After clicking on the button, a message notification will pop up on the screen notifying the finance manager that the purchase order is rejected.

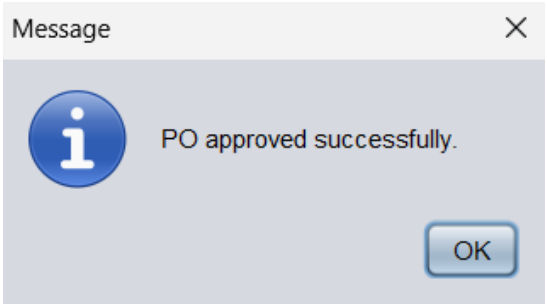



**Figure 5.8.6.3 PO Approval Message**

Figure 5.8.6.3 shows the message when the finance manager clicks on the “Approve PO” button after selecting a purchase order. After clicking on the button, a message notification will pop up on the screen notifying the finance manager that the purchase order is approved.

## 5.8.7 Purchase Order Verification

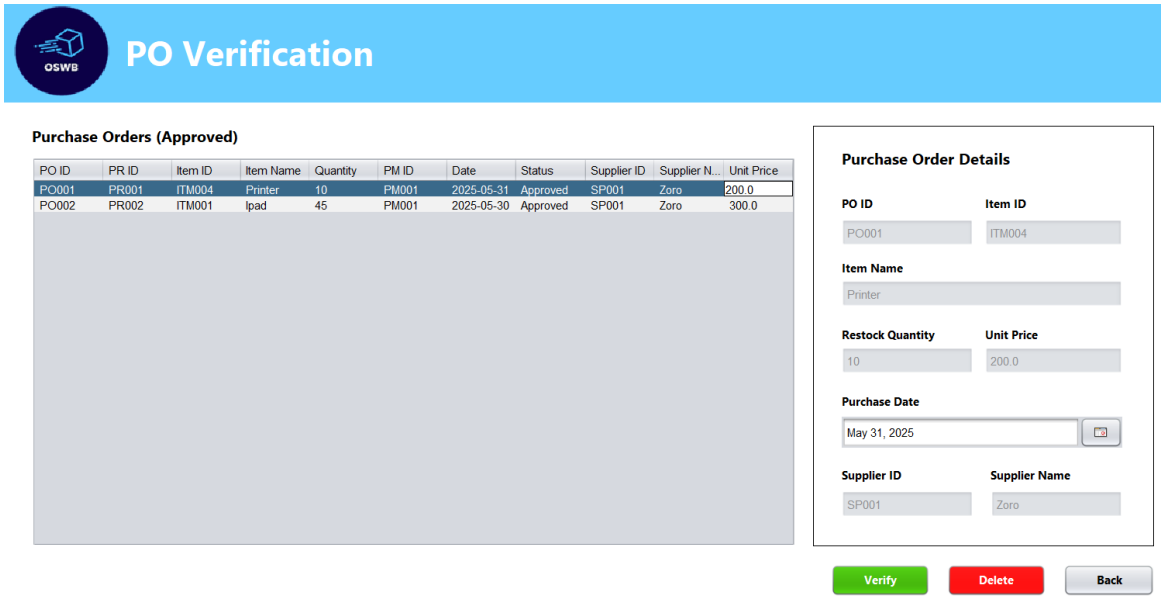



**Figure 5.8.7.1 PO Verification Interface**

Figure 5.8.7.1 shows the Purchase Order Verification interface where the finance manager will enter this interface once they click on the “PO Verification” button in the home menu. In this interface, the finance manager can verify or delete PO by selecting any of the approved purchase orders from the table and clicking on the “Verify” or “Delete” button.

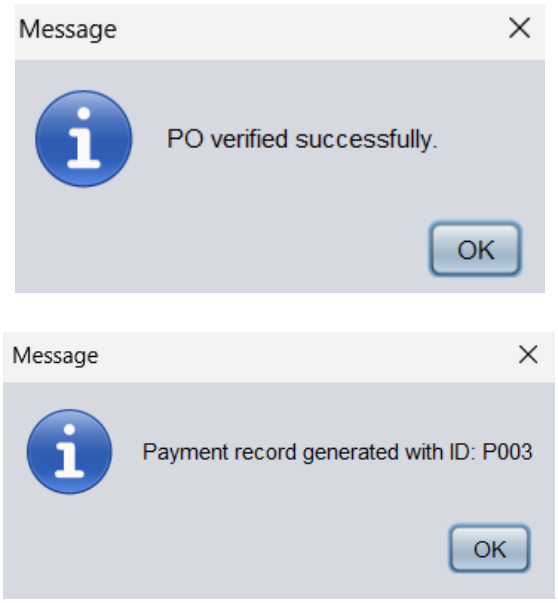



**Figure 5.8.7.2 PO Verification Successful Message**

Figure 5.8.7.2 shows the message when the finance manager clicks on the "Verify" button after selecting an approved purchase order. After clicking on the button, a message notification will pop up on the screen notifying the finance manager that the approved purchase order is now verified. There will also be a message notifying the purchase order is generated with a payment record of its own ID.

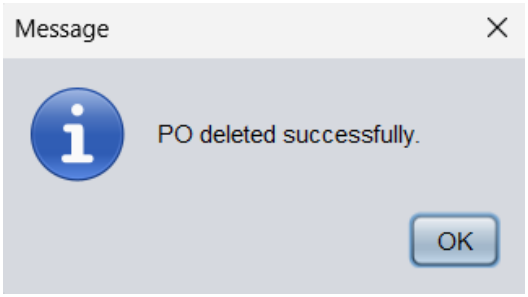



**Figure 5.8.7.3 PO Deletion Successful Message**

Figure 5.8.7.3 shows the message when the finance manager clicks on the "Delete" button after selecting an approved purchase order. After clicking on the button, a message notification will pop up on the screen notifying the finance manager that the approved purchase order is now deleted.

## 5.8.8 View Approved Purchase Orders

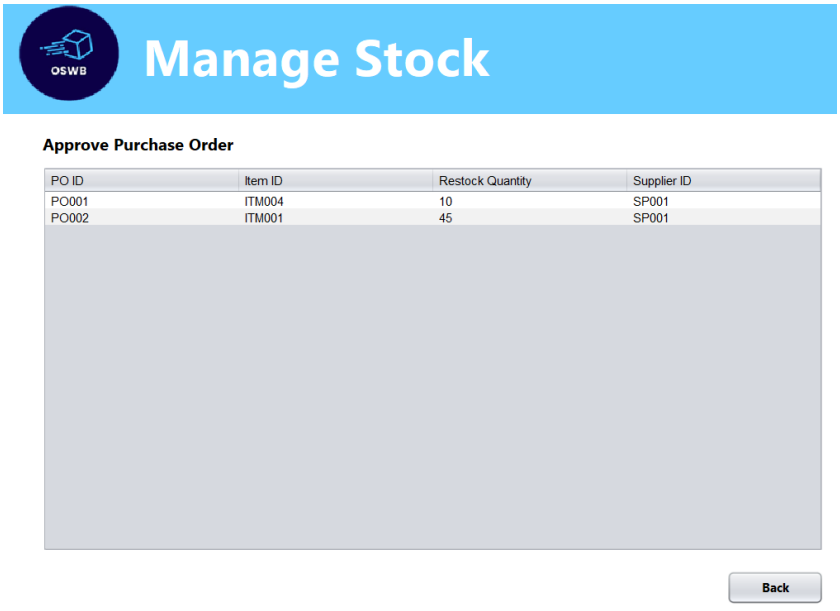


**Figure 5.8.8.1 Approved PO Interface**

Figure 5.8.8.1 shows the Approved Purchase Order interface where the finance manager will enter this interface once they click on the "View Approved PO" button in the home menu. In this interface, the finance manager can view the details of the approved purchase order that includes the PO ID, Item ID, Restock quantity and Supplier ID.

## 5.9 Manage Stock of Approved Purchase Order

### 5.9.1 Manage Stock Update

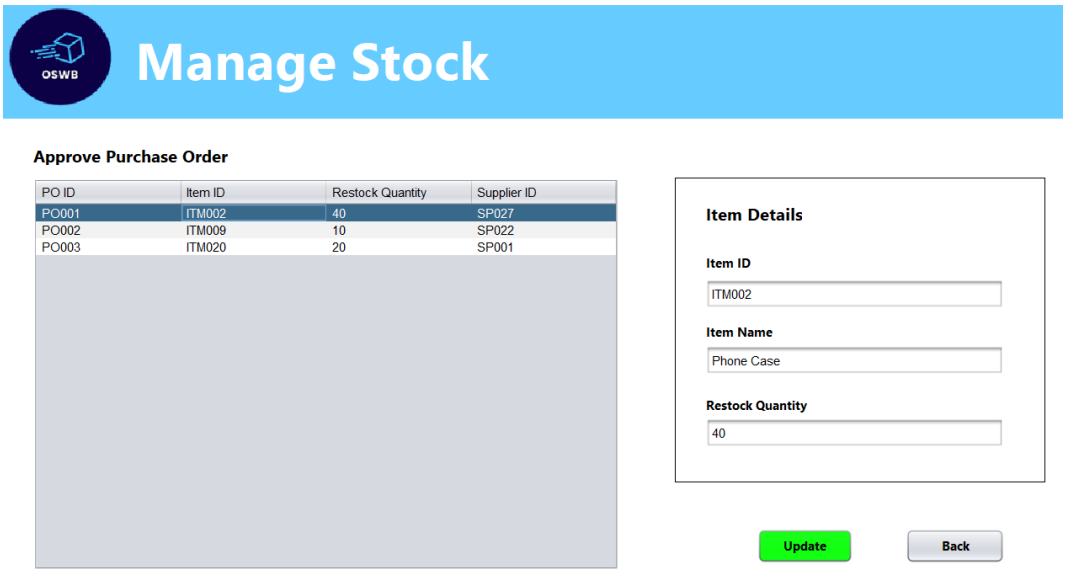



**Figure 5.9.1.1 Manage Stock Interface**

Figure 5.9.1.1 shows the Manage Stock interface where the inventory manager will enter this interface once they click on the "Manage Stock" button in the home menu. In this interface, the inventory manager can view the details of the approved purchase order that includes the PO ID, Item ID, Restock quantity and Supplier ID. The inventory manager then can select any one of the approved PO and click the "Update" button to update the quantity by adding the items from the PO to the inventory.

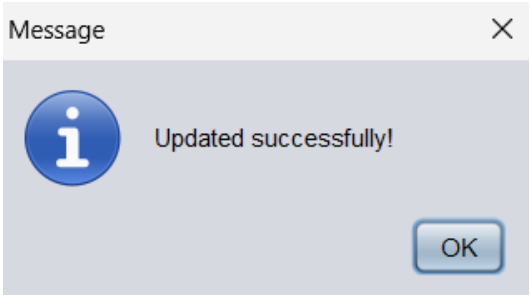



**Figure 5.9.1.2 Update Successful Message**

Figure 5.9.1.2 shows the message when the inventory manager clicks on the "Approve PO" button after selecting a purchase order. After clicking on the button, a message notification will pop up on the screen notifying the finance manager that the purchase order is approved.

## 5.10 Supplier Payment

### 5.10.1 View Supplier Payment

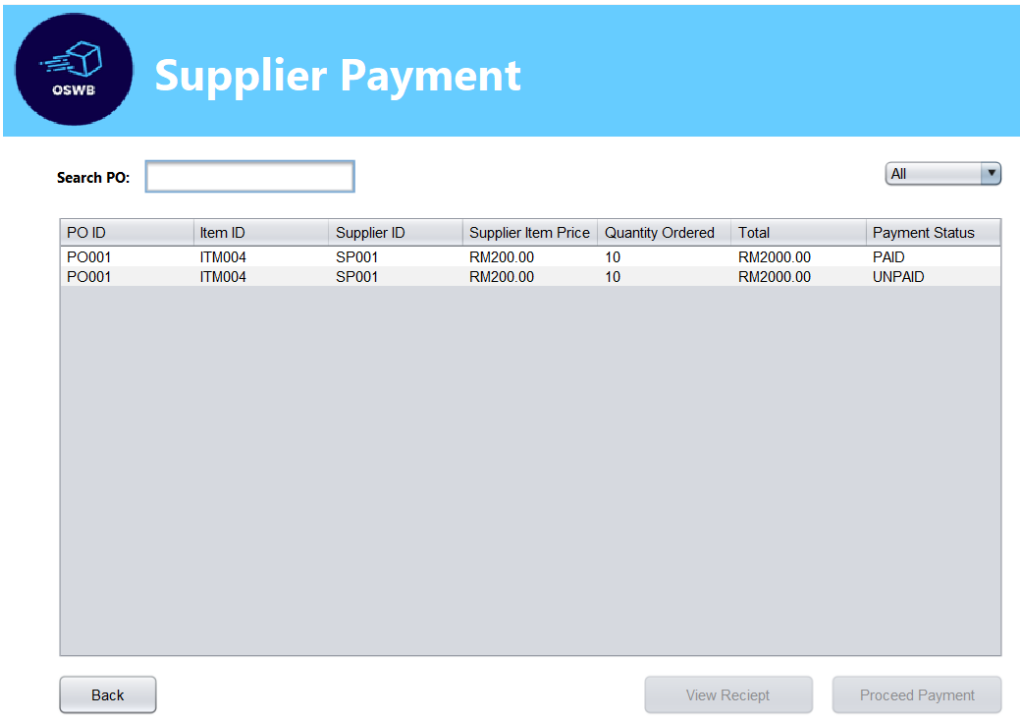



**Figure 5.10.1.1 Supplier Payment Interface**

Figure 5.10.1.1 shows the View Supplier Payment interface where the finance manager will enter this interface once they click on the "Supplier Payment" button in their home menu. In

this interface, the users can view details like PO ID, Item ID, Supplier ID, Supplier Item Price, Quantity Ordered, Total and Payment Status.

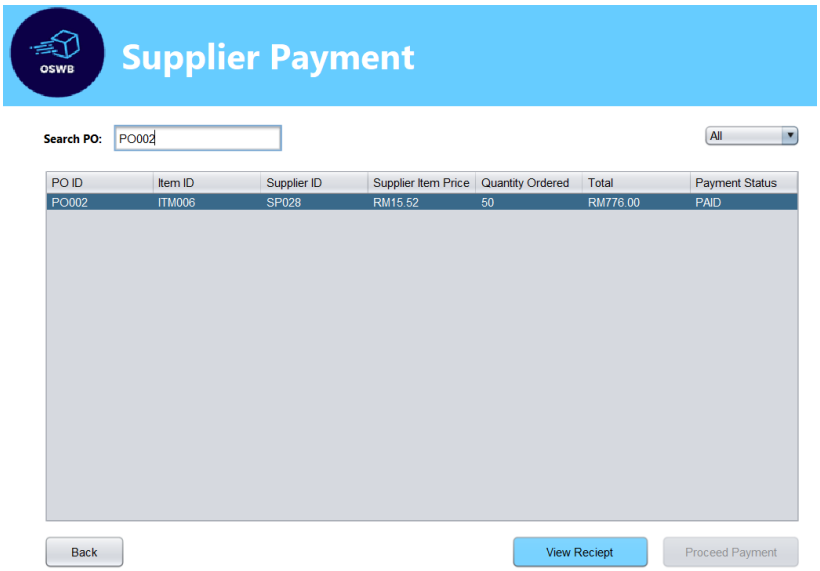



**Figure 5.10.1.2 Supplier Payment Entry Information Prompt**

Figure 5.10.1.2 shows if the users want to search for a supplier payment from the list, the users could type the PO ID, Item ID or Supplier ID in the search bar and if the ID exists, the supplier payment will then be listed below.

## 5.10.2 Proceed Supplier Payment

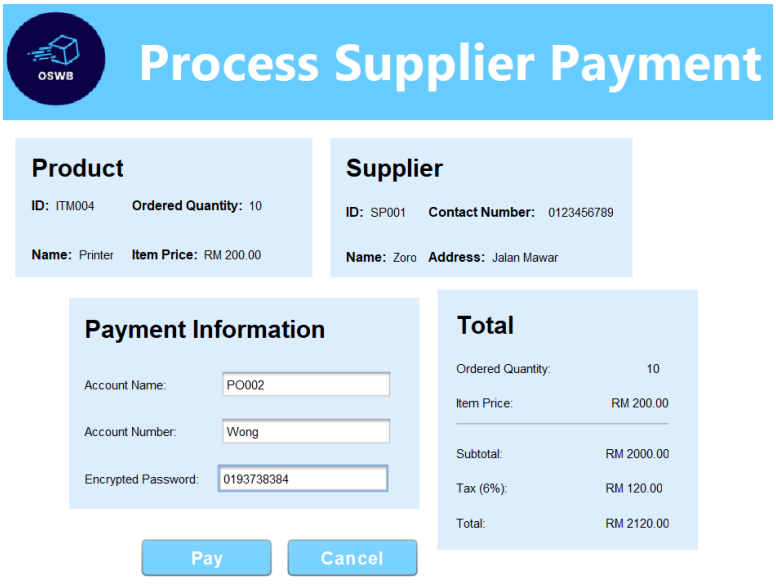


**Figure 5.10.5.1 Process Supplier Payment Interface**

Figure 5.10.5.1 shows the Process Supplier Payment Interface where the finance manager will enter this interface once they click on the “Proceed Payment” button in the supplier payment interface. In this interface, the finance manager can view details of the product, supplier and also the total of the cost for the product. They will also need to fill in the account name, account number and encrypted password input field to process the payment.

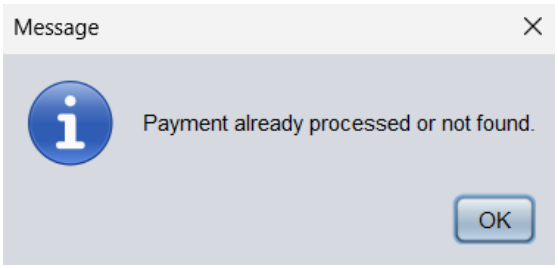



**Figure 5.10.5.2 Error Confirmation**

Figure 5.10.5.2 shows the message of the finance manager being rejected of processing the payment due to the possibility of the payment not being found or being paid.

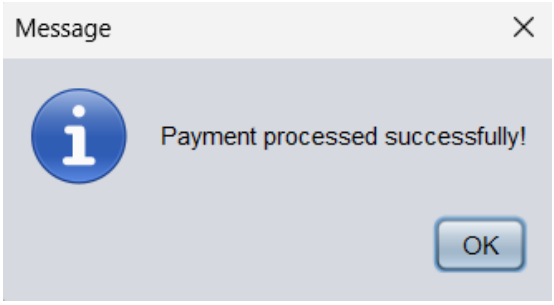



**Figure 5.10.5.3 Payment Process Successful Message**

Figure 5.10.5.3 shows the message when the finance manager clicks on the “Pay” button after filling in the required input field for the payment. After clicking on the button, a message notification will pop up on the screen notifying the finance manager that the payment has been successfully processed.

### 5.10.3 View Receipt Payment

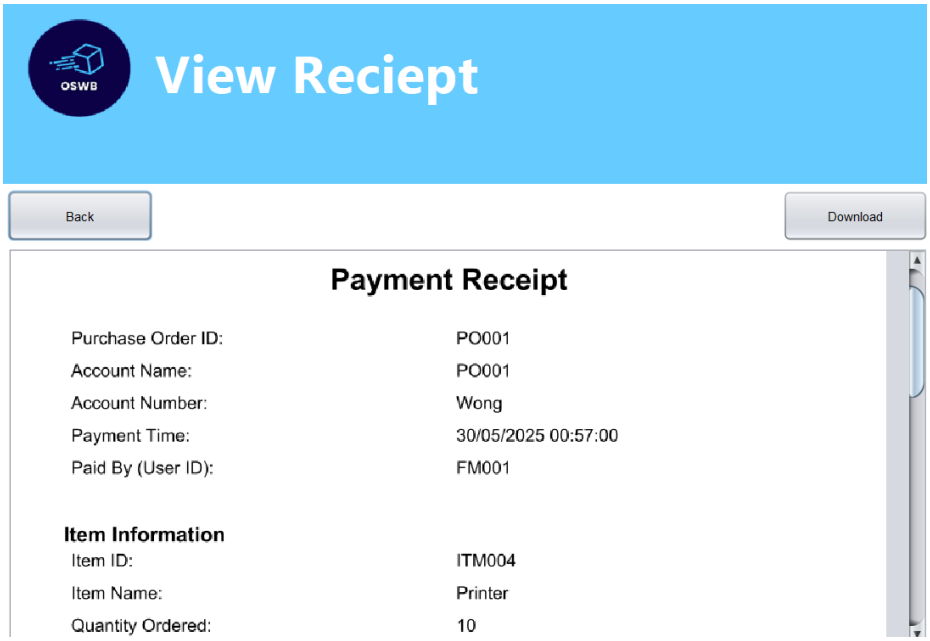



**Figure 5.10.3.1 View Receipt Payment Interface**

Figure 5.10.3.1 shows the View Receipt Payment Interface where the finance manager will enter this interface once they click on the "View Receipt" button in the supplier payment interface. In this interface, the finance manager can view details of the receipt that includes the information of the purchase order, item bought, supplier and the cost summary. The finance manager can also click on the "Download" button to download the receipt.

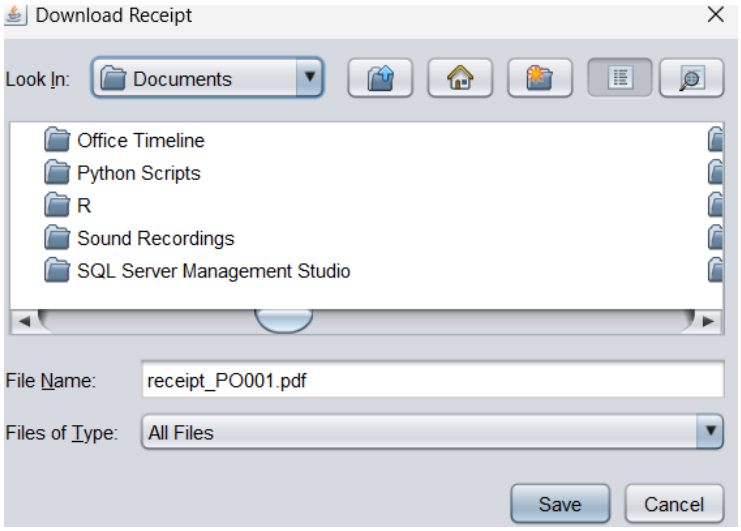



**Figure 5.10.3.2 Download Receipt Page**

Figure 5.10.3.2 shows the Download Receipt Page where the finance manager will download the receipt into their own files after clicking on the “Download” button in the View Receipt interface.

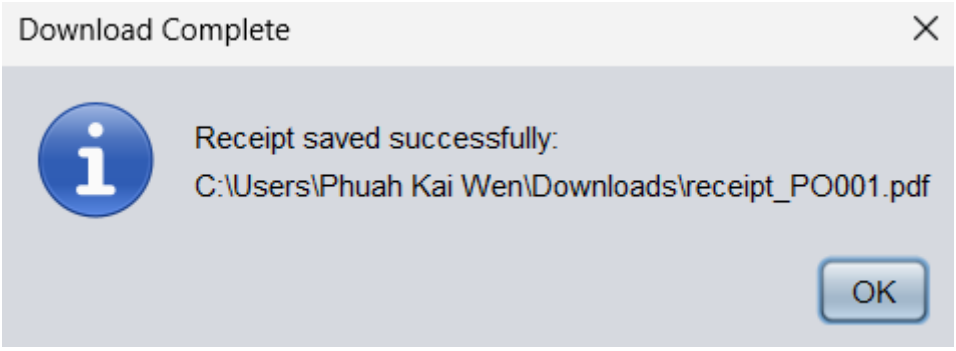



**Figure 5.10.3.3 Download Receipt Successful Message**

Figure 5.10.3.3 shows the message when the finance manager clicks on the “Save” button to save the receipt in their own desktop files. After clicking on the button, a message notification will pop up on the screen notifying the finance manager that the receipt is now successfully saved.

# 5.11 Alerts and Notifications

## 5.11.1 Low Stock Alert

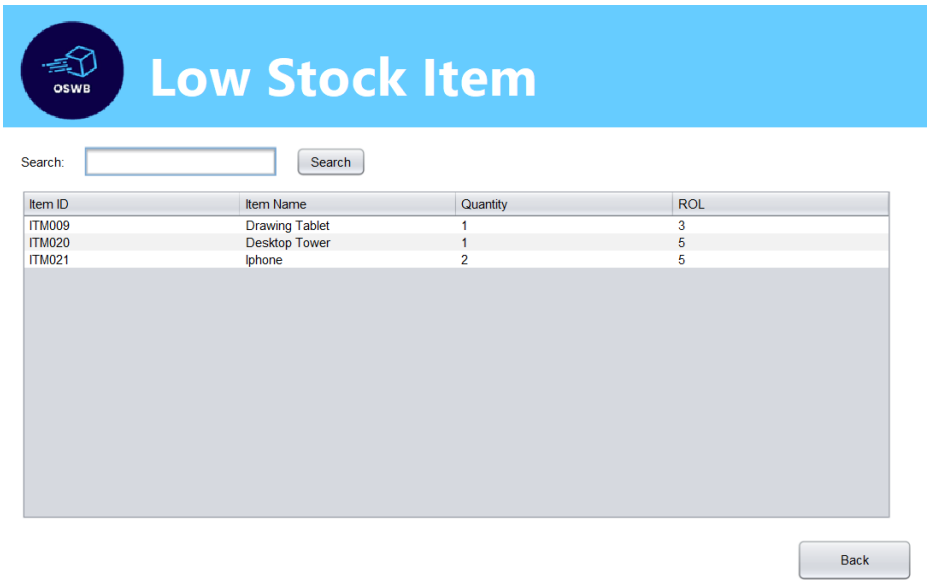



**Figure 5.11.1.1 Low Stock Item Pop-Up Screen**

Figure 5.11.1.1 shows the Low Stock Item pop-up screen where the inventory manager will enter this interface once they click on the “Low Stock Alert” button in their home menu. In this interface, the users can view items whose quantities are below the reorder level with details like Item ID, Item Name, Quantity and Reorder level.

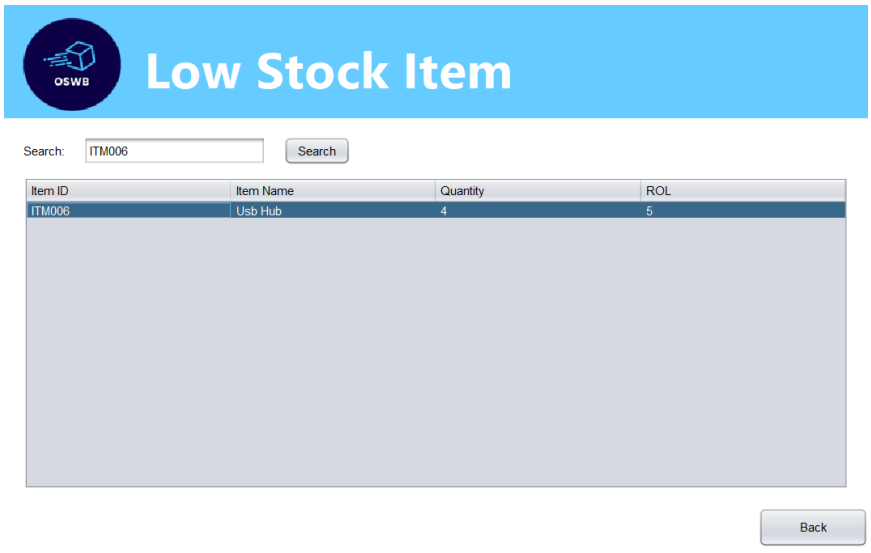


**Figure 5.11.1.2 Item Entry Information Prompt**

Figure 5.11.1.2 shows if the users want to search for a item from the low stock item list, the users could type the Item ID in the search bar and if the ID exists, the item will then be listed below.

## 5.11.2 Purchase Requisition Notification

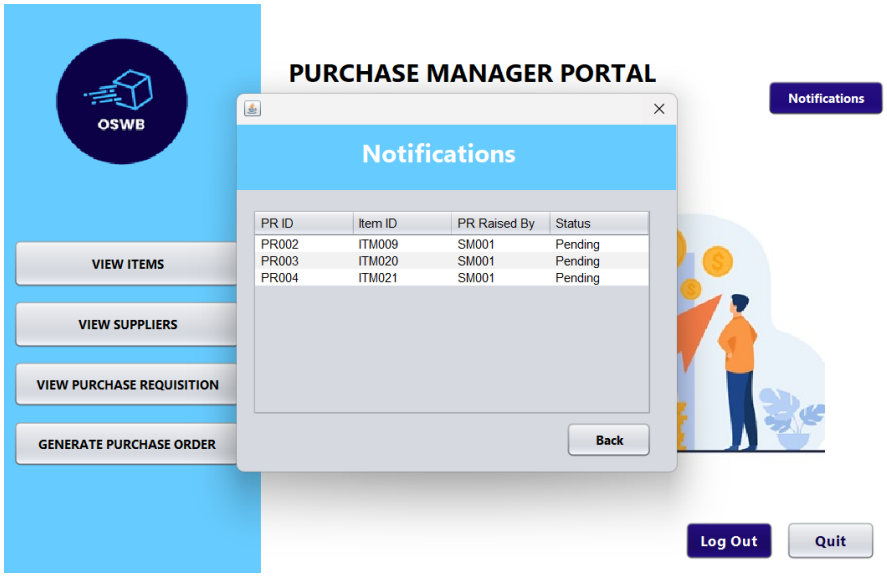



**Figure 5.11.5.1 Purchase Requisition Pop-Up Screen**

Figure 5.11.5.1 shows the Purchase Requisition pop-up screen where the purchase manager will enter this interface once they click on the “Notifications” button in their home menu. In this interface, the users can view purchase requisition that has been created by the sales manager and needed to be approved by them with details including PR ID, Item ID, PR Raised By and Status.

### 5.11.3 Purchase Order Notification

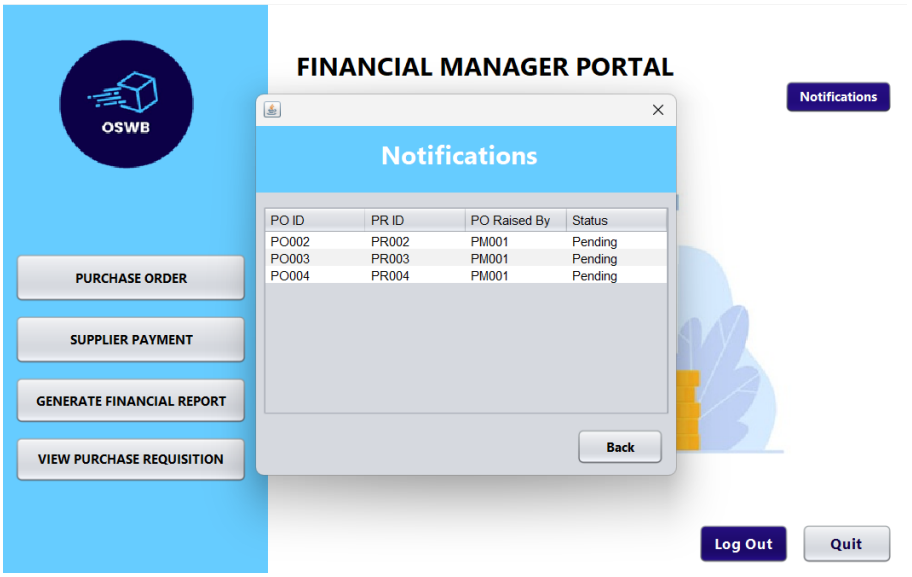



**Figure 5.11.3.1 Purchase Order Pop-Up Screen**

Figure 5.11.3.1 shows the Purchase Order pop-up screen where the sales manager will enter this interface once they click on the “Notifications” button in their home menu. In this interface, the users can view purchase order that has been created by the purchase manager and needed to be approved by them with details including PO ID, PR ID, PO Raised By and Status.

## 5.12 Report Generation

### 5.15.1 Generate Stock Report

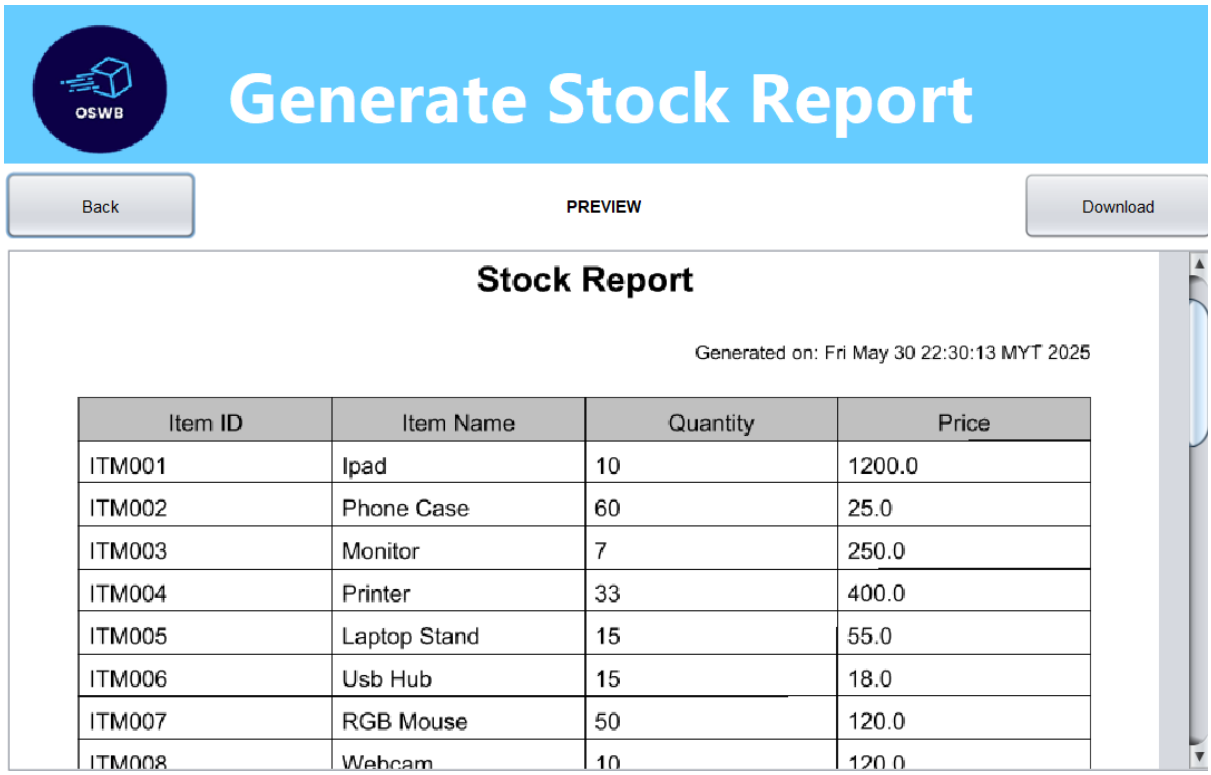



**Figure 5.15.1.1 Generate Stock Report Interface**

Figure 5.15.1.1 shows the Generate Stock Report Interface where the inventory manager will enter this interface once they click on the “Generate Stock Report” button in their home menu. In this interface, a stock report regarding the details of all the items in the inventory will be generated into a document format for the inventory manager. The inventory manager can also click on the “Download” button to download the report if needed.

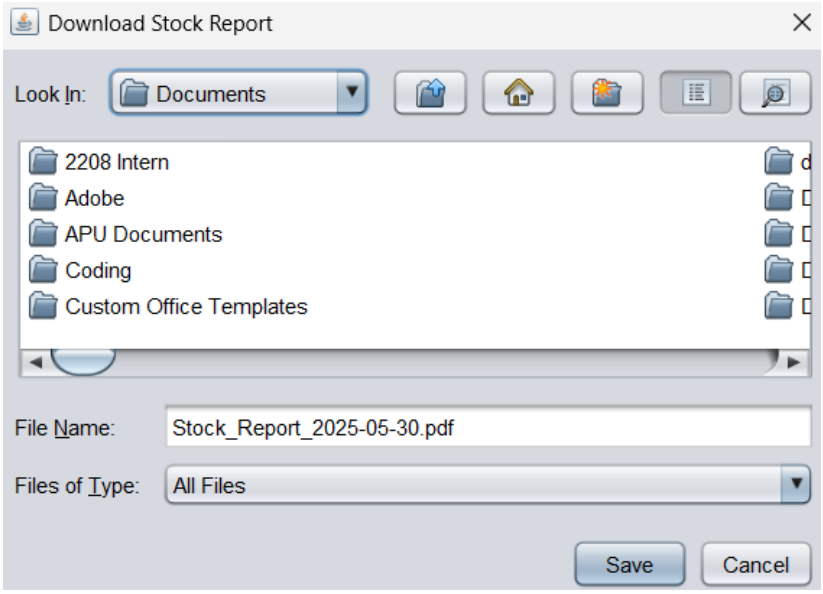



**Figure 5.15.1.2 Download Report Page**

Figure 5.15.1.2 shows the Download Report Page where the inventory manager will download the report into their own files after clicking on the "Download" button in the Generate Stock Report interface.

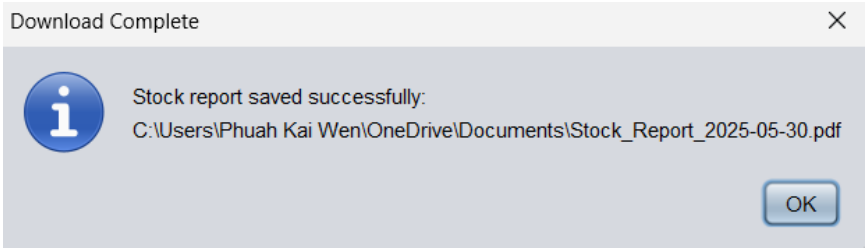



**Figure 5.15.1.3 Download Report Successful Message**

Figure 5.15.1.3 shows the message when the inventory manager clicks on the "Save" button to save the report in their own desktop files. After clicking on the button, a message notification will pop up on the screen notifying that the report is now successfully saved.

## 5.15.2 Generate Financial Report

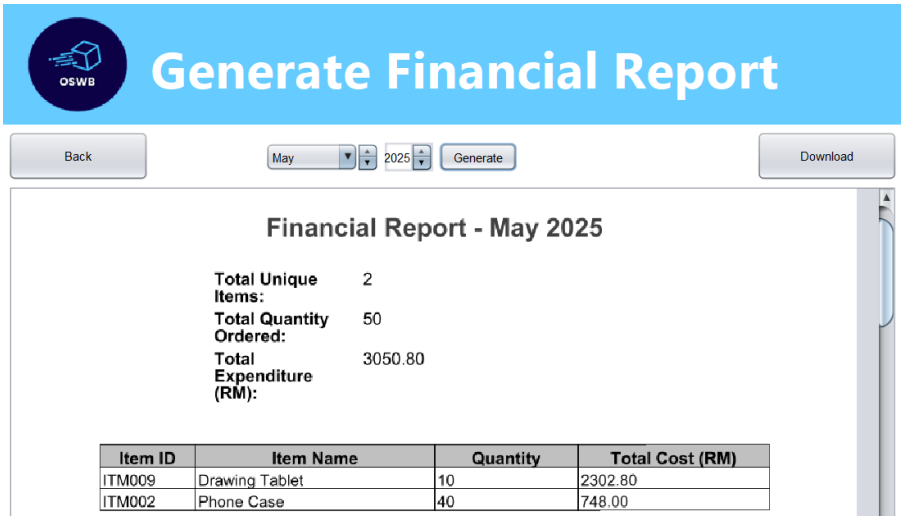



**Figure 5.15.5.1 Generate Financial Report Interface**

Figure 5.15.1.1 shows the Generate Financial Report Interface where the financial manager will enter this interface once they click on the "Generate Financial Report" button in their home menu. In this interface, a financial report regarding all the item quantities and total expenditure of that selected will be generated into a document format by clicking on the "Generate" button. The sales manager can also click on the "Download" button to download the report if needed.

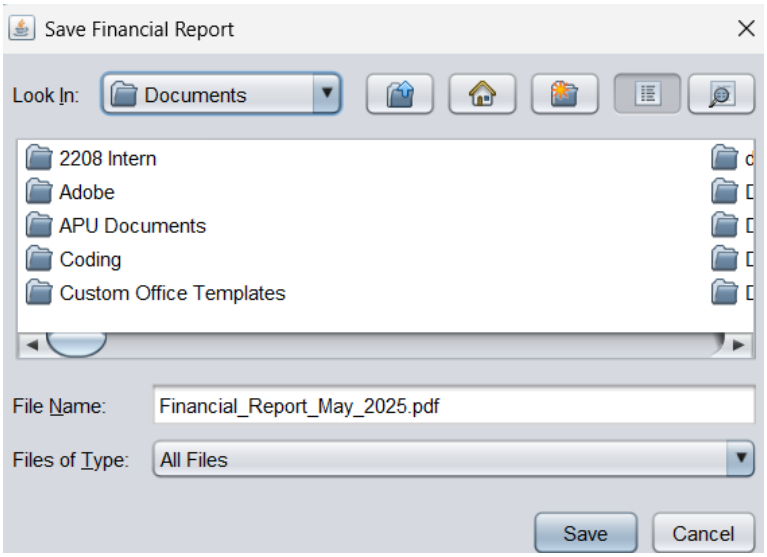



**Figure 5.15.2.2 Download Report Page**

Figure 5.15.2.2 shows the Download Report Page where the sales manager will download the report into their own files after clicking on the “Download” button in the Generate Financial Report interface.

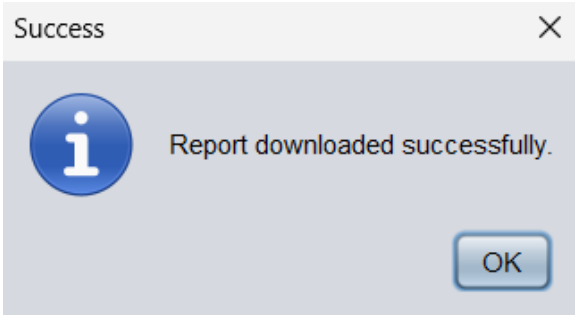



**Figure 5.15.5.3 Download Report Successful Message**

Figure 5.15.5.3 shows the message when the sales manager clicks on the “Save” button to save the report in their own desktop files. After clicking on the button, a message notification will pop up on the screen notifying that the report is now successfully saved.

# 6. Additional Features: PDF Generator

## 6.1 PDF Generator

The PDF Generator enables users to:

- Preview the generated reports.
- Download the reports to their desired location.

### 6.1.1 Receipt

The Receipt report provides detailed payment information, including:

- Payment Details: Payment Order ID, Bank Account Name, and Account Number.
- User Information: Details of the user who made the payment.
- Item Information: Item ID, Item Name, and Quantity Ordered.
- Supplier Information: Supplier Name, Supplier ID, Supplier Address, Contact Number, and Supply Item Price.
- Summary: Subtotal, Tax, and Total Price.

### 6.1.2 Stock Report

The Stock Report offers a comprehensive view of current inventory, including:

- Item ID
- Item Name
- Stock Quantity
- This allows users to efficiently monitor stock levels.

### 6.1.3 Financial Report

The Financial Report displays all payments made within a selected month.

Users can:

- Choose a specific month and year to view.
- Review a summary of all financial transactions made during that period.

## 6.2 Abstract Class

```java
public abstract class PdfGenerator {
    protected final String outputPath;
    protected final boolean previewAfterSave;
    protected Document document;

    public PdfGenerator(String outputPath, boolean previewAfterSave) {
        this.outputPath = outputPath;
        this.previewAfterSave = previewAfterSave;
    }

    public void generate() {
        try {
            document = new Document();
            PdfWriter.getInstance(document, new FileOutputStream(outputPath));
            document.open();

            addLogo();
            addTitle(document);
            addTimestamp();
            addContent(document);
            addFooter();

            document.close();

            if (previewAfterSave) {
                // open file
                java.awt.Desktop.getDesktop().open(new File(outputPath));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    protected void addLogo() throws DocumentException {}

    protected abstract void addTitle(Document document) throws DocumentException;

    protected abstract void addTimestamp();

    protected abstract void addContent(Document document) throws DocumentException;

    protected abstract void addFooter();
}
```

**Figure 6.1: PDF Generator abstract class**

The PdfGenerator class is an abstract base class used to create different types of PDF reports, such as receipts, stock reports, and financial summaries. It uses the iText PDF library and follows a fixed process using the generate() method. Some parts of the PDF, like the title and content, are left for subclasses to define by writing their own methods. This makes the code easier to reuse and extend for different types of reports.

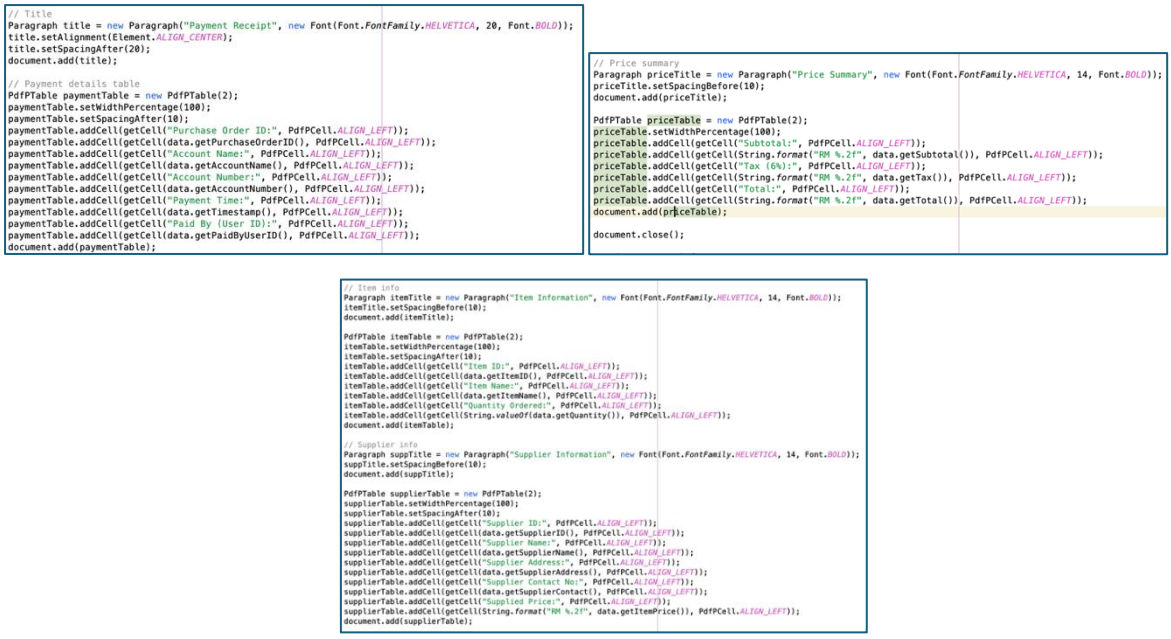
# 6.3 Receipt

## 6.3.1 Receipt Pdf Generator

```
public class RecieptPdfGenerator {

    public void generateReceipt(String purchaseOrderId, File outputFile) {
        try {
            POManager poManager = new POManager();
            PurchaseOrder po = poManager.getById(purchaseOrderId);
            if (po == null) {
                System.out.println("Purchase order not found for ID: " + purchaseOrderId);
                return;
            }

            // Create parent directories if they don't exist
            File parentDir = outputFile.getParentFile();
            if (parentDir != null && !parentDir.exists()) {
                parentDir.mkdirs();  // Create folders recursively
            }

            // Create the file if it does not exist
            if (!outputFile.exists()) {
                outputFile.createNewFile();
            }

            RecieptReportGenerator generator = new RecieptReportGenerator(po);
            ReceiptData data = generator.getReceiptData();

            Document document = new Document();
            PdfWriter.getInstance(document, new FileOutputStream(outputFile));
            document.open();

            try {
                Image logo = Image.getInstance("/img/smallOSWB.png");
                logo.scaleToFit(100, 100);
                logo.setAlignment(Element.ALIGN_LEFT);
                document.add(logo);
            } catch (Exception e) {}
```

**Figure 6.2: Receipt Pdf Generator class**

The RecieptPdfGenerator class creates a receipt PDF based on a purchase order. It follows the structure of the PdfGenerator abstract class, which defines the steps for PDF creation. It fills in details like payment info, item and supplier details, and a price summary using tables.



```
// Title
Paragraph title = new Paragraph("Payment Receipt", new Font(Font.FontFamily.HELVETICA, 20, Font.BOLD));
title.setAlignment(Element.ALIGN_CENTER);
title.setSpacingAfter(20);
document.add(title);

// Payment details table
PdfPTable paymentTable = new PdfPTable(2);
paymentTable.setWidthPercentage(100);
paymentTable.setSpacingAfter(10);
paymentTable.addCell(getCell("Purchase Order ID:", PdfPCell.ALIGN_LEFT));
paymentTable.addCell(getCell(data.getPurchaseOrderID(), PdfPCell.ALIGN_LEFT));
paymentTable.addCell(getCell("Account Name:", PdfPCell.ALIGN_LEFT));
paymentTable.addCell(getCell(data.getAccountName(), PdfPCell.ALIGN_LEFT));
paymentTable.addCell(getCell("Account Number:", PdfPCell.ALIGN_LEFT));
paymentTable.addCell(getCell(data.getAccountNumber(), PdfPCell.ALIGN_LEFT));
paymentTable.addCell(getCell("Payment Time:", PdfPCell.ALIGN_LEFT));
paymentTable.addCell(getCell(data.getTimestamp(), PdfPCell.ALIGN_LEFT));
paymentTable.addCell(getCell("Paid By (User ID):", PdfPCell.ALIGN_LEFT));
paymentTable.addCell(getCell(data.getPaidByUserID(), PdfPCell.ALIGN_LEFT));
document.add(paymentTable);
```

```
// Price summary
Paragraph priceTitle = new Paragraph("Price Summary", new Font(Font.FontFamily.HELVETICA, 14, Font.BOLD));
priceTitle.setSpacingBefore(10);
document.add(priceTitle);

PdfPTable priceTable = new PdfPTable(2);
priceTable.setWidthPercentage(100);
priceTable.addCell(getCell("Subtotal:", PdfPCell.ALIGN_LEFT));
priceTable.addCell(getCell(String.format("RM %.2f", data.getSubtotal()), PdfPCell.ALIGN_LEFT));
priceTable.addCell(getCell("Tax (6%):", PdfPCell.ALIGN_LEFT));
priceTable.addCell(getCell(String.format("RM %.2f", data.getTax()), PdfPCell.ALIGN_LEFT));
priceTable.addCell(getCell("Total:", PdfPCell.ALIGN_LEFT));
priceTable.addCell(getCell(String.format("RM %.2f", data.getTotal()), PdfPCell.ALIGN_LEFT));
document.add(priceTable);

document.close();
```

```
// Item info
Paragraph itemTitle = new Paragraph("Item Information", new Font(Font.FontFamily.HELVETICA, 14, Font.BOLD));
itemTitle.setSpacingBefore(10);
document.add(itemTitle);

PdfPTable itemTable = new PdfPTable(2);
itemTable.setWidthPercentage(100);
itemTable.setSpacingAfter(10);
itemTable.addCell(getCell("Item ID:", PdfPCell.ALIGN_LEFT));
itemTable.addCell(getCell(data.getItemID(), PdfPCell.ALIGN_LEFT));
itemTable.addCell(getCell("Item Name:", PdfPCell.ALIGN_LEFT));
itemTable.addCell(getCell(data.getItemName(), PdfPCell.ALIGN_LEFT));
itemTable.addCell(getCell("Quantity Ordered:", PdfPCell.ALIGN_LEFT));
itemTable.addCell(getCell(String.valueOf(data.getQuantity()), PdfPCell.ALIGN_LEFT));
document.add(itemTable);

// Supplier info
Paragraph suppTitle = new Paragraph("Supplier Information", new Font(Font.FontFamily.HELVETICA, 14, Font.BOLD));
suppTitle.setSpacingBefore(10);
document.add(suppTitle);

PdfPTable supplierTable = new PdfPTable(2);
supplierTable.setWidthPercentage(100);
supplierTable.setSpacingAfter(10);
supplierTable.addCell(getCell("Supplier ID:", PdfPCell.ALIGN_LEFT));
supplierTable.addCell(getCell(data.getSupplierID(), PdfPCell.ALIGN_LEFT));
supplierTable.addCell(getCell("Supplier Name:", PdfPCell.ALIGN_LEFT));
supplierTable.addCell(getCell(data.getSupplierName(), PdfPCell.ALIGN_LEFT));
supplierTable.addCell(getCell("Supplier Address:", PdfPCell.ALIGN_LEFT));
supplierTable.addCell(getCell(data.getSupplierAddress(), PdfPCell.ALIGN_LEFT));
supplierTable.addCell(getCell("Supplier Contact No:", PdfPCell.ALIGN_LEFT));
supplierTable.addCell(getCell(data.getSupplierContact(), PdfPCell.ALIGN_LEFT));
supplierTable.addCell(getCell("Supplied Price:", PdfPCell.ALIGN_LEFT));
supplierTable.addCell(getCell(String.format("RM %.2f", data.getItemPrice()), PdfPCell.ALIGN_LEFT));
document.add(supplierTable);
```

**Figure 6.3: Receipt PDF Layout Code**

This part of the code creates and adds content to the receipt PDF. It starts with a centered title called **"Payment Receipt"**. Then, it adds several tables to display different kinds of information. The first table shows payment details like the purchase order ID, account name and number, payment time, and user ID. After that, it adds an "Item Information" section with details such as item ID, name, and quantity ordered. Next, it shows the "Supplier Information" section, which includes supplier ID, name, address, contact number, and item price. Finally, it adds a **"Price Summary"** that shows the subtotal, 6% tax, and total amount.

## 6.3.2 Receipt Report Generator

```
public class RecieptReportGenerator {
    private Item item;
    private Supplier supplier;
    private Payment payment;
    private PurchaseOrder po;
    private SupplierManager supplierManager;

    private static final String USER_FILE = "userData.txt";
    private static final DecimalFormat df = new DecimalFormat("0.00");

    public RecieptReportGenerator(PurchaseOrder po) {
        ItemManager itemManager = new ItemManager();
        supplierManager = new SupplierManager();
        PaymentManager paymentManager = new PaymentManager();
        POManager purchaseorder = new POManager();

        this.item = itemManager.findItemById(po.getItemID());
        System.out.println("item:" + item);
        this.supplier = supplierManager.findSupplierById(po.getSupplierID());
        System.out.println("Fetched Supplier: " + supplier);
        this.payment = paymentManager.findPaymentByPO(po.getPoID());
        this.po = purchaseorder.getById(po.getPoID());
        System.out.println("Supplier: " + supplier.getSupplierID() + "\nItem: " + item.getItemID());
        System.out.println("SupplierItemPrice: " + supplierManager.getItemPrice(supplier.getSupplierID(),item.getItemID()));


    }
```

**Figure 6.4: ReceiptReportGenerator Class**

The constructor of the RecieptReportGenerator class initializes essential components by retrieving the item, supplier, payment, and purchase order details related to the provided purchase order. It utilizes dedicated manager classes to fetch this data and includes debug outputs to verify successful data retrieval. This ensures that the generator has all necessary information to create an accurate payment receipt.

```
public String getItemId() { return item.getItemID(); }
public String getItemName() { return item.getItemName(); }
public int getQuantity() { return po.getQuantity(); }
public double getItemPrice() { return supplierManager.getItemPrice(supplier.getSupplierID(),item.getItemID());}
public String getSupplierId() { return supplier.getSupplierID(); }
public String getSupplierName() { return supplier.getSupplierName(); }
public String getSupplierAddress() { return supplier.getSupplierAddress(); }
public String getSupplierContact() { return supplier.getContactNo(); }
public String getAccountName() { return payment.getAccountName(); }
public String getAccountNumber() { return payment.getAccountNumber(); }
public String getPaymentTime() { return payment.getTimestamp(); }
public String getPaidBy() { return payment.getUserID(); }
public double getSubtotal() { return getItemPrice() * getQuantity(); }
public double getTax() { return getSubtotal() * 0.06; }
public double getTotal() { return getSubtotal() + getTax(); }
public String formatCurrency(double value) { return "RM " + df.format(value); }

public models.ReceiptData getReceiptData() {
    models.ReceiptData data = new models.ReceiptData();

    data.setPurchaseOrderID(po.getPoID());
    data.setAccountName(getAccountName());
    data.setAccountNumber(getAccountNumber());
    data.setTimestamp(getPaymentTime());
    data.setPaidByUserID(getPaidBy());

    data.setItemID(getItemId());
    data.setItemName(getItemName());
    data.setQuantity(getQuantity());
    data.setItemPrice(getItemPrice());

    data.setSupplierID(getSupplierId());
    data.setSupplierName(getSupplierName());
    data.setSupplierAddress(getSupplierAddress());
    data.setSupplierContact(getSupplierContact());

    data.setSubtotal(getSubtotal());
    data.setTax(getTax());
    data.setTotal(getTotal());

    return data;
}
```

**Figure 6.5: getReceiptData Method**

The class provides methods to access detailed information about the item, supplier, payment, and purchase order, including IDs, names, contact details, quantities, and prices. It also calculates the subtotal, tax (at a fixed rate of 6%), and the total payment amount. Finally, the `getReceiptData` method consolidates all relevant information into a `ReceiptData` object, which serves as the basis for generating a complete payment receipt.

### 6.3.3 Receipt Data

```
public class ReceiptData {
    private String purchaseOrderID;
    private String accountName;
    private String accountNumber;
    private String timestamp;
    private String paidByUserID;

    private String itemID;
    private String itemName;
    private int quantity;
    private double itemPrice;

    private String supplierID;
    private String supplierName;
    private String supplierAddress;
    private String supplierContact;

    private double subtotal;
    private double tax;
    private double total;

    // Getters and Setters

    public String getPurchaseOrderID() {
```

**Figure 6.6: ReceiptData Encapsulation and Getters/Setters**

The `ReceiptData` class holds all the information needed to make a payment receipt. It has details like purchase order ID, account name, account number, payment time, and who made the payment. It also stores information about the item and supplier, such as their names, IDs, and contact details. The class also has fields for subtotal, tax, and total amount. There are getter and setter methods to get and set these values easily.

## 6.4 Stock Report

### 6.4.1 Stock PDF Generator

```
public class StockPdfGenerator extends PdfGenerator {

    private final List<String> headers;
    private final List<List<String>> data;

    public StockPdfGenerator(String outputPath, boolean previewAfterSave, List<String> headers, List<List<String>> data) {
        super(outputPath, previewAfterSave);
        this.headers = headers;
        this.data = data;
    }
```

**Figure 6.7: StockPDFGenerator Class**

The StockPdfGenerator class creates a stock report in PDF format. It extends the PdfGenerator abstract class and follows its structure for making PDF documents. It uses a list of headers and data rows to fill the stock table. The report includes a title, the time it was created, the table of stock items, and a footer showing the total number of items.

```
@Override
protected void addTitle(Document document) throws DocumentException {
    Font titleFont = new Font(Font.FontFamily.HELVETICA, 18, Font.BOLD);
    Paragraph titlePara = new Paragraph("Stock Report", titleFont);
    titlePara.setAlignment(Element.ALIGN_CENTER);
    titlePara.setSpacingAfter(20f);
    document.add(titlePara);
}

@Override
protected void addTimestamp() {
    try {
        Font timeFont = FontFactory.getFont(FontFactory.HELVETICA, 10);
        Paragraph timestamp = new Paragraph("Generated on: " + new java.util.Date(), timeFont);
        timestamp.setAlignment(Element.ALIGN_RIGHT);
        timestamp.setSpacingAfter(10f);
        document.add(timestamp);
    } catch (Exception e) {
        Logger.getLogger(StockPdfGenerator.class.getName()).log(Level.SEVERE, null, e);
    }
}

@Override
protected void addContent(Document document) throws DocumentException {
    PdfPTable table = new PdfPTable(headers.size());
    table.setWidthPercentage(100);

    for (String header : headers) {
        PdfPCell cell = new PdfPCell(new Phrase(header));
        cell.setBackgroundColor(BaseColor.LIGHT_GRAY);
        cell.setHorizontalAlignment(Element.ALIGN_CENTER);
        cell.setPadding(5f);
        table.addCell(cell);
    }

    for (List<String> row : data) {
        for (String cellData : row) {
            PdfPCell cell = new PdfPCell(new Phrase(cellData));
            cell.setPadding(5f);
            table.addCell(cell);
        }
    }
```

**Figure 6.8: Stock PDF Layout Code**

This part of the code adds content to the stock report. First, it adds a title at the top that says "Stock Report", centered on the page. Then, it adds the current date and time on the top right. After that, it creates a table. The first row of the table shows the headers like "Item ID", "Item Name", "Stock", etc., in grey background. Below the headers, each row in the table shows the actual stock data. At the bottom, a footer is added that shows how many items are listed in the report.

## 6.4.2 Stock Report Generator

```
public final class StockReportGenerator {

    private StockReportGenerator() {}

    public static void generateStockReport(File outputFile, boolean preview) throws IOException, Exception {
        List<String> headers = Arrays.asList("Item ID", "Item Name", "Quantity", "Price");
        List<List<String>> data = getStockData();

        StockPdfGenerator pdfGenerator = new StockPdfGenerator(outputFile.getAbsolutePath(), preview, headers, data);
        pdfGenerator.generate();
    }

    public static File getDefaultFilePath() {
        String fileName = "Stock_Report_" + LocalDate.now() + ".pdf";
        return new File(fileName);
    }
```

**Figure 6.9: StockReportGenerator Class**

The StockReportGenerator class is a utility class that creates a PDF report showing all stock items. It uses the StockPdfGenerator class to build and save the report. The class is final and

has a private constructor, which means it cannot be extended or instantiated. It provides static methods to prepare the stock data, create the PDF, and suggest a default file name.

```
private static List<List<String>> getStockData() {
    List<List<String>> data = new ArrayList<>();

    try {
        List<String> lines = FileOperation.ReadFileAsList(Item.class);
        for (String line : lines) {
            String[] parts = line.split(",");
            if (parts.length >= 4) {
                data.add(Arrays.asList(parts[0], parts[1], parts[2], parts[3]));
            }
        }
    } catch (Exception e) {
        JOptionPane.showMessageDialog(null, "Error reading stock data or generating report.");
        e.printStackTrace();
    }

    return data;
}
```

**Figure 6.10: Stock Report generation code**

This part of the code first defines the table headers: "Item ID", "Item Name", "Quantity", and "Price". It then reads the stock data from a file using the FileOperation.ReadFileAsList method, expecting each line to contain these four pieces of information. If the data is valid, it adds each item as a row in a list. The generateStockReport method passes this list and the headers to the StockPdfGenerator, which creates the actual PDF.

## 6.4 Financial Report Generator

```
public class FinancialReportGenerator {

    private final PaymentManager paymentManager;
    private final POManager poManager;
    private final ItemManager itemManager;
    private final SupplierManager supplierManager;

    public FinancialReportGenerator() {
        this.paymentManager = new PaymentManager();
        this.poManager = new POManager();
        this.itemManager = new ItemManager();
        this.supplierManager = new SupplierManager();
    }
```

**Figure 6.11: FinancialReportGenerator class**

The FinancialReportGenerator class prepares the data needed for the monthly financial report. It collects payment records and checks if the payment was made in the selected month and year. It only includes payments that are marked as "PAID". It uses managers to get the purchase order, item, and supplier details. For each item, it adds up the total quantity and total cost.

```
public Map<String, ItemSummary> generateReportData(int selectedMonth, int selectedYear) {
    Map<String, ItemSummary> summaryMap = new HashMap<>();
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");

    List<Payment> payments = paymentManager.load();
    boolean foundData = false;

    for (Payment payment : payments) {
        if (!"PAID".equalsIgnoreCase(payment.getStatus()) || payment.getTimestamp() == null) continue;

        try {
            LocalDateTime dateTime = LocalDateTime.parse(payment.getTimestamp(), formatter);
            if (dateTime.getMonthValue() == selectedMonth && dateTime.getYear() == selectedYear) {
                foundData = true;

                PurchaseOrder po = poManager.getById(payment.getPurchaseOrderID());
                if (po == null) continue;

                Item item = itemManager.findItemById(po.getItemID());
                if (item == null) continue;

                String itemID = item.getItemID();
                String itemName = item.getItemName();
                int quantity = po.getQuantity();
                String supplierID = po.getSupplierID();

                double price = supplierManager.getItemPrice(supplierID, itemID);
                double totalCost = quantity * price;

                ItemSummary summary = summaryMap.getOrDefault(itemID,
                    new ItemSummary(itemID, itemName, 0, 0.0));

                summary.addQuantity(quantity);
                summary.addCost(totalCost);

                summaryMap.put(itemID, summary);
            }
        } catch (DateTimeParseException e) {
            System.err.println("Invalid date format for payment timestamp: " + payment.getTimestamp());
        }
    }
```

**Figure 6.12: Financial Report Data Generation Code**

This part of the code loads all payment records. It checks the date of each payment and matches it with the selected month and year. If the payment is valid, it finds the related purchase order, item, and supplier. Then, it calculates the item's total cost by multiplying the quantity by the item price. It stores this information in a summary map. Each item has a summary that includes the item ID, name, total quantity ordered, and total cost.

# 7. Conclusion & Limitation

In conclusion, the automated Purchase Order Management System was built using Object-Oriented Programming (OOP) principles such as encapsulation, inheritance, polymorphism, and abstraction. Building the system based on these fundamental principles allows the system to stay organized as related data and functions are grouped into classes. Therefore, this makes it simpler for the system to be managed as it provides a clear role-based access, ensuring each of the roles have the maximum guarantee that its operation will be run smoothly.

Having an automated Purchase Order Management System is important nowadays as it helps streamlining procurement operations in real-life business settings by digitizing the procurement workflow. It reduces the need for manual input, minimizes human errors, and ensures accurate processing of purchase orders. This helps strengthen the inventory management and contribute to cost reductions across various departments (Monczka et al., 2015). By centralizing procurement data, the system allows a better decision-making process and provides a clear audit trail, which are essential in maintaining transparency and accountability (Laudon & Laudon, 2020).

However, despite this system offers good operational benefits, it also comes with some limitations. For example, the system's effectiveness heavily relies on accurate data input; any inaccurate entries will disrupt the procurement process. Moreover, some small businesses will struggle with the limited IT infrastructure or lack of staff training (Heizer et al., 2020). Additionally, the system security also proves to be a huge limitation due to the large amount of handling sensitive financial and supplier private information, this information is highly prone and vulnerable to cyber-attacks if OWSB does not follow and implement any safety measures. (Stair & Reynolds, 2020)

# References

Bergenti, F., Chiarabini, L., & Rossi, G. (2011). Programming with partially specified aggregates in Java. *Computer Languages Systems & Structures*, *37*(4), 178–192. https://doi.org/10.1016/j.cl.2011.07.002

Braunschweig, D. (2018, December 15). *Encapsulation*. Pressbooks. https://press.rebus.community/programmingfundamentals/chapter/encapsulation/

Bhat, H. (2024, November 22). *What is Abstraction in OOPs? Definition, Types, Advantages*. AlmaBetter. https://www.almabetter.com/bytes/articles/abstraction-in-oops

Cengage Learning. (n.d.). https://faculty.cengage.com/works/9781285869681 Laudon, K. C., & Laudon, J. P. (2020). Management information systems: Managing the digital firm (16th ed.). Pearson. https://www.pearsonhighered.com/assets/preface/0/1/3/5/0135173620.pdf

Chodrow, P. S., Veldt, N., & Benson, A. R. (2021). Generative hypergraph clustering: From blockmodels to modularity. *Science Advances*, *7*(28). https://doi.org/10.1126/sciadv.abh1303

Drabent, W. (2023, May 25). *Implementing backjumping by means of exception handling*. arXiv.org. https://arxiv.org/abs/2305.16137

GeeksforGeeks. (2025, May 7). *Association, composition and aggregation in Java*. GeeksforGeeks. https://www.geeksforgeeks.org/association-composition-aggregation java/

GeeksforGeeks. (2025, March 27). *Classes and objects in Java*. GeeksforGeeks. https://www.geeksforgeeks.org/classes-objects-java/

Gessenharter, D. (2009). Implementing UML associations in Java. *RAOOL '09: Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages*, 17 24. https://doi.org/10.1145/1562100.1562104

Monczka, R. M., Handfield, R. B., Giunipero, L. C., & Patterson, J. L. (2021). Purchasing & supply chain management (7th ed.). Cengage Learning. https://www.cengage.com/c/purchasing-and-supply-chain-management-7e-monczka/

Naqvi, S. (2024, November 25). A deep dive Object-Oriented Programming (OOP) in dart and flutter. *Medium*. https://medium.com/@shahzebnaqvi/a-deep-dive-object-oriented programming-oop-in-dart-and-flutter-70e2ef6fd2ac

Sharma, M., Sharma, C., Bhardwaj, A., Singh, N., & Singh, L. (2012). COMPARATIVE STUDY OF STATIC METRICS OF PROCEDURAL AND OBJECT ORIENTED PROGRAMMING LANGUAGES. *INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY*, *2*(1), 15–19. https://doi.org/10.24297/ijct.v2i1.2607

Singh, G., Sharma, M., Arora, A., & Kaur, P. (2012). A comparative study of static object oriented metrics. *ResearchGate*. https://www.researchgate.net/publication/277106128_A_Comparative_Study_of_Sc_Object_Oriented_Metrics

Stair, R., & Reynolds, G. (2020). Principles of information systems (13th ed.). Cengage Learning. https://api.pageplace.de/preview/DT0400.9781292403571_A42098351/preview 9781292403571_A42098351.pdf

Sutherland, B. (2014). *Controlling Data with Access Modifiers*. https://www.semanticscholar.org/paper/Controlling-Data-with-Access-Modifiers Sutherland/112674392c10b319655bf305d53ccfb5d9443c30

Taylor, T. (2023, October 20). *Understanding the role of polymorphism in OOP*. Search App Architecture. https://www.techtarget.com/searchapparchitecture/tip/Understanding the-role-of-polymorphism-in-OOP

Vats, R. (2025, May 9). *Loose vs Tight Coupling in Java: Key Differences & Examples*. upGrad Blog. https://www.upgrad.com/blog/loose-coupling-vs-tight-coupling-in-java/

Yu, C., Zhang, X., Zhang, X., Li, G., & Tang, Z. (2021). Reversible data hiding with hierarchical embedding for encrypted images. *IEEE Transactions on Circuits and*

*Systems for Video Technology*, *32*(2), 451–466. https://doi.org/10.1109/tcsvt.2021.3062947

Zhang, X., Guo, G., He, X., & Hu, Z. (2023). Bidirectional Object-Oriented Programming: towards programmatic and direct manipulation of objects. *Proceedings of the ACM on Programming Languages*, *7*(OOPSLA1), 230–255. https://doi.org/10.1145/3586035