GROUP ASSIGNMENT

CT038-3-2-OODJ

OBJECT ORIENTED DEVELOPMENT WITH JAVA

APU2F2411CS(DA/DF)

HAND OUT DATE : 27 NOVEMBER 2024

HAND IN DATE : 22 FEBRUARY 2024

WEIGHTAGE : 50%

GROUP NO: 15

Student ID	Student Name
TP077983	Faiyad Mahabub Tasin (Leader)
TP055673	Batrisyia Binti Mohammad Iqmal
TP079311	Loh Ehung
TP078574	Emmanuel Naranendhya Tyatan
TP079309	Nurul Aiman Binti Raizal Azhar

Table of Contents

1.0 Workload Matrix

Name	ROLES				
	ADMIN	**VENDOR**	**CUSTOMER**	**DELIVERY RUNNER**	**MANAGER**
Faiyad Mahabub Tasin (TP077983)	20%	20%	20%	20%	20%
Batrisyia Binti Mohammad Iqmal (TP055673)	20%	20%	20%	20%	20%
Loh Ehung (TP079311)	20%	20%	20%	20%	20%
Emmanuel Naranendhya Tyatan (TP078574)	20%	20%	20%	20%	20%
Nurul Aiman Binti Raizal Azhar (TP079309)	20%	20%	20%	20%	20%

2.0 Introduction

The Lepak Food Court Management System is a digital platform that helps to streamline the food ordering process within the food court. It helps to boost efficiency in the order and delivery process of the food court, thereby boosting the business output of the food court. The application allows customers to view food options within the food court through the app and is able to connect customer orders to vendors instantly and assign delivery persons to handle the delivery of the food. Customers can be kept informed about the status of their order, reducing the need for manual updates. It also helps the manager to review the performances of vendors through reviews made by the customers. Other than that, the customers do not need to make payment every time they make an order, and instead it is paid through a credit system, where the admin helps the customer to top up money into their account. The system is also able to keep track of the customers' order history, allowing the customer to reorder food items that they have previously ordered during their last visit. This application aims to maximize convenience and efficiency for all processes occurring in a food court.

3.0 Design Solution

3.1 Class Diagram

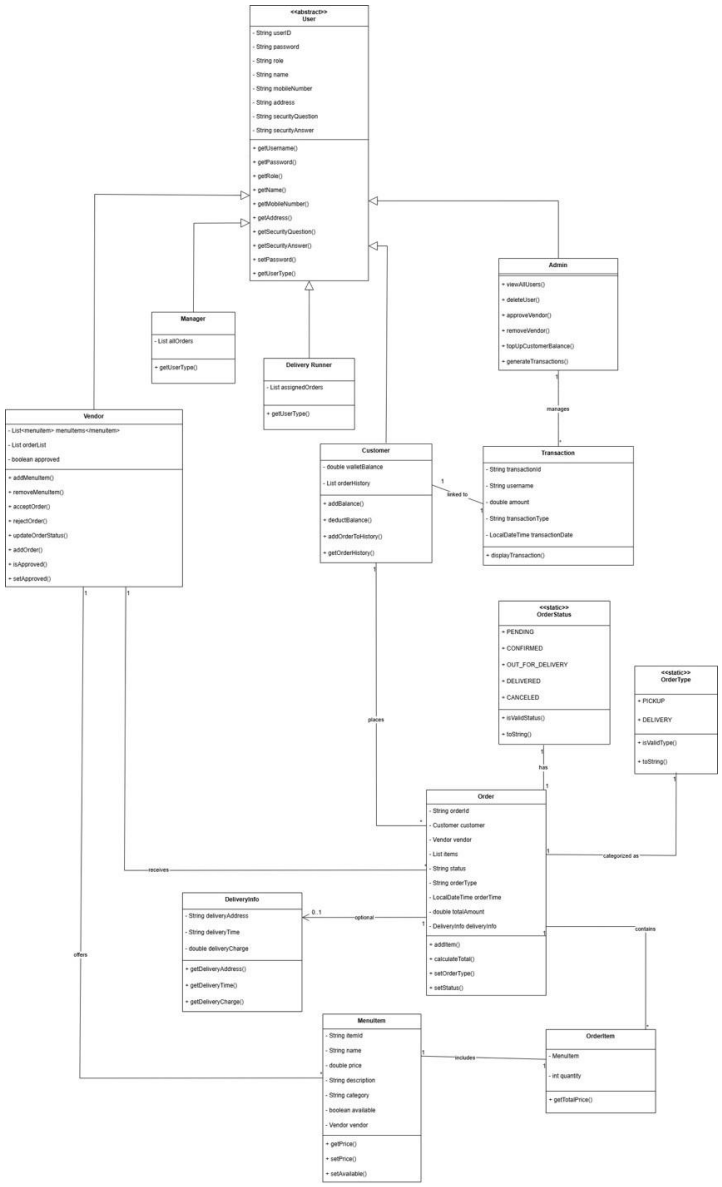


Figure 1.0: Class diagram

The class diagram encapsulates the functionalities of a food ordering and delivery system, encompassing user management, order processing, menu management, and transaction tracking. At its foundation, the ‘User’ class encapsulates shared attributes such as ‘userID’, ‘password’, ‘name’, ‘mobileNumber’, and ‘address’, along with getter methods and ‘getUserType’. Several specialized roles extend from ‘User’, including ‘Manager’, ‘Vendor’, ‘Customer’, ‘DeliveryRunner’, and ‘Admin’.

The ‘Manager’ class provides user-related functionalities through ‘getUserType’, while the ‘Vendor’ class oversees menu items and orders, possessing attributes such as ‘menuItems’ and

‘orders’, as well as methods such as ‘acceptOrder’, ‘rejectOrder’, ‘updateOrderStatus’, and ‘setApproved’. The ‘Customer’ class maintains order history and balance, offering methods like ‘addBalance’, ‘deductBalance’, and ‘getOrderHistory’. The ‘DeliveryRunner’ is responsible for handling assigned deliveries, while the ‘Admin’ class holds system-wide control, with functions such as ‘deleteUser’, ‘approveVendor’, and ‘generateTransaction’.

The ‘Order’ class represents customer transactions and comprises attributes such as ‘orderID’, ‘customer’, ‘vendor’, ‘items’, ‘status’, ‘orderType’, ‘orderTime’, ‘totalAmount’, and ‘deliveryInfo’. It provides methods for adding orders, calculating totals, and updating status. The ‘OrderStatus’ class defines static order states (‘PENDING’, ‘CONFIRMED’, ‘OUT_FOR_DELIVERY’, ‘DELIVERED’, and ‘CANCELLED’), while the ‘OrderType’ categorizes orders as either ‘PICKUP’ or ‘DELIVERY’.

The ‘Transaction’ class records payments, tracking details such as ‘transactionType’, ‘username’, ‘amount’, ‘transactionDate’, and ‘displayTransaction’. Delivery-related information, including ‘deliveryAddress’, ‘deliveryTime’, and ‘deliveryCharge’, is stored in the ‘DeliveryInfo’ class. The ‘MenuItem’ class holds food item details, including ‘menuId’, ‘name’, ‘price’, ‘category’, and ‘availability’, with methods for updating price and availability. The ‘OrderItem’ class connects menu items to orders, managing ‘quantity’ and calculating total price through ‘getTotalPrice’.

This class diagram efficiently models the system’s structure, ensuring seamless user interactions, order processing, and financial management.

3.2 Use Case Diagram

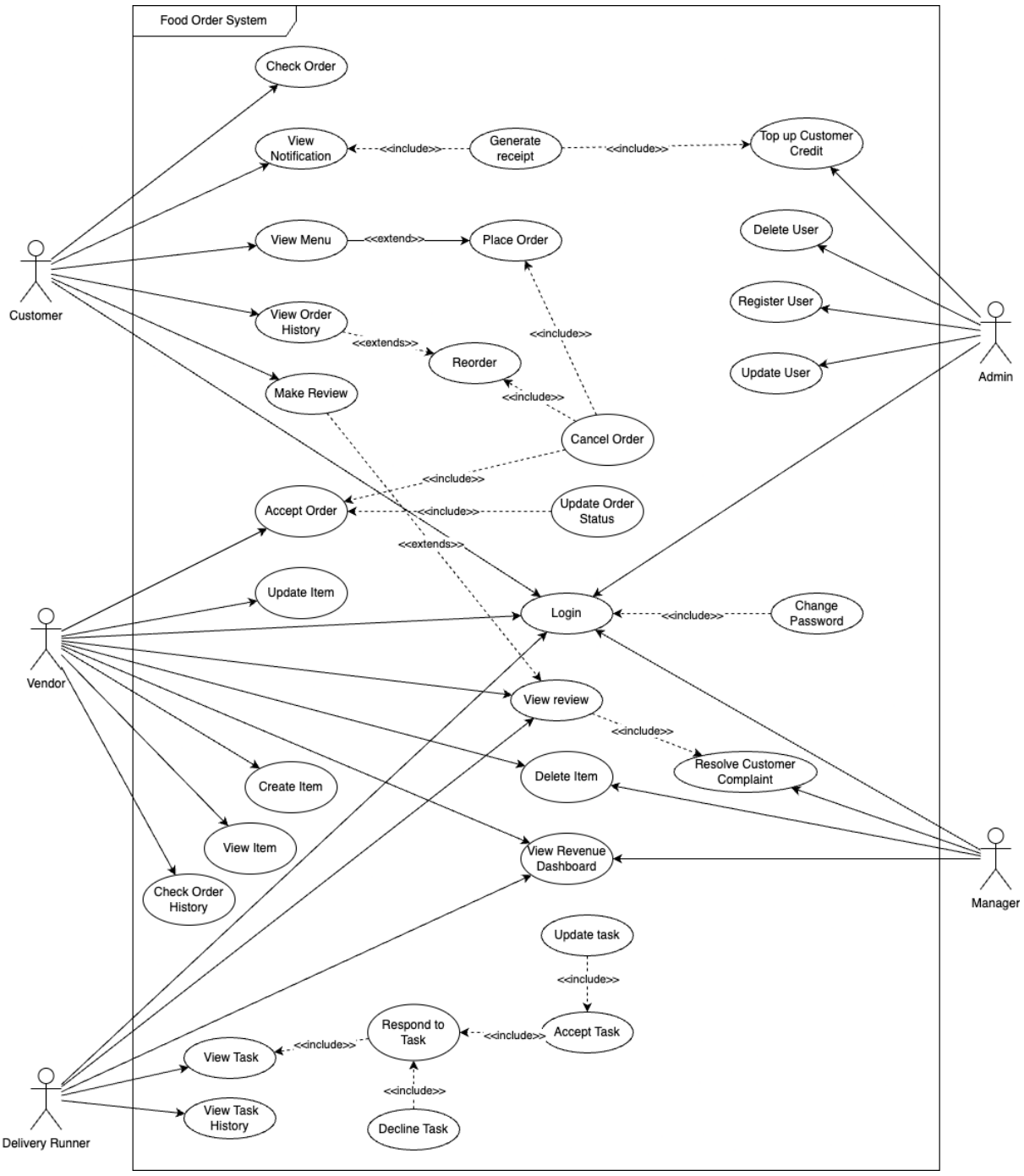


Figure 1.1: Use Case Diagram

The use case diagram for the Lepak Food Court Management System shows how each actor interacts with the system to handle various tasks. The 5 actors in the system being: (1) Customer, (2) Vendor, (3) Delivery Runner, (4) Admin, (5) Manager. Customers have the ability to place orders, browse menus, reorder previously ordered items, check notifications,

write reviews, and keep track of their order history. They can also choose whether to cancel orders. Other than establishing, editing, and removing menu items, vendors can also monitor order histories, take orders, and create or delete items in their menu. Deliveries are handled by delivery runners who also view and reply to tasks assigned to them, accept or reject deliveries, and keep track of task history. Admins oversee user accounts by adding, editing, removing users, and also helping to top up customer’s in-app credits for transactions. Operations are managed by managers who also handle revenue dashboards, monitor system activity, and address consumer complaints. Dependencies between use cases, such as creating a receipt upon making an order or changing order status as part of the order process, are depicted in the diagram using "include" and "extend" connections.

4.0 Screenshots of Output of the Program with Appropriate Explanations

4.1 Sign UP

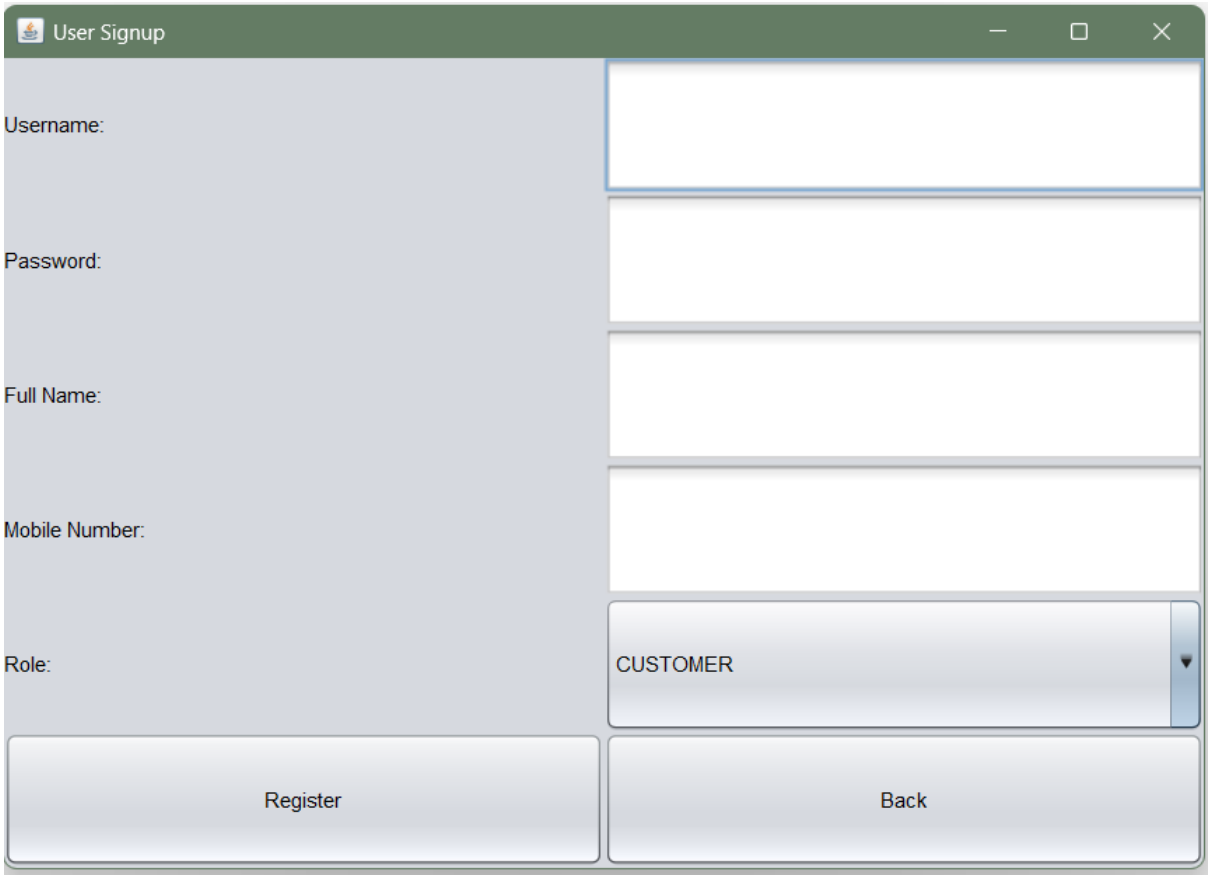


Figure 2.0: Sign Up Page.

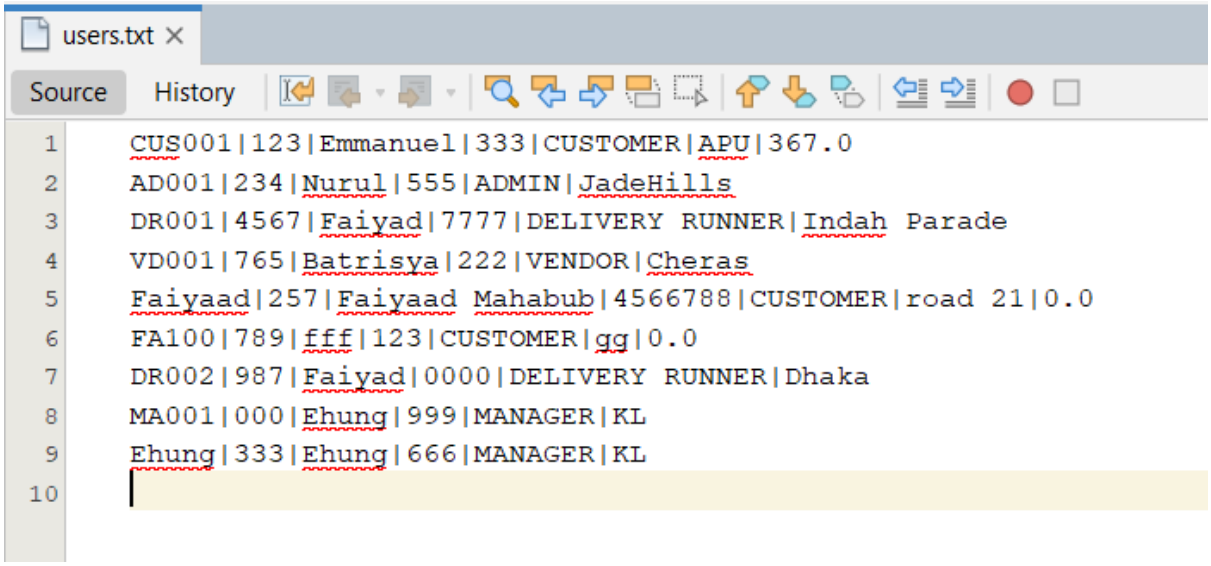


Figure 2.1: All Data Present in User.txt with new Registries.

The snippets above indicates the Sign Up dashboard and the “users.txt” file. The first snippet is the UI that display the sign-up method, where it can be used by all roles (Customer, Vendor, Manager, Admin, and Delivery Runner). Then after signing up, the data will be automatically updated inside users text file.

4.2 Login

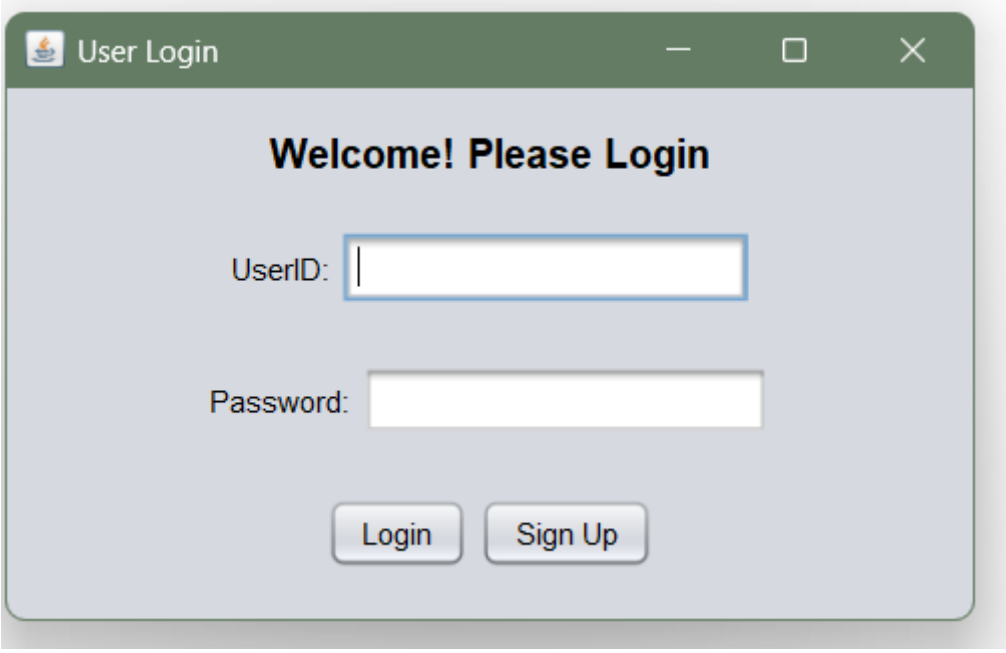


Figure 3.0: Login Page

The snippet above is the User Login Menu where all roles can log in to their own system with their designated username and password.

4.3 Admin

4.3.0 Admin Dashboard

Figure 3.1: Admin Dashboard

First thing we're going to do is to log in as Admin. In the Admin Dashboard, we have 6 feature that we can work on, we have Manager User to edit user's information, View Vendors to view who is the available vendor, View Delivery Runners to view who is currently active for delivery, Top Up Credit where Admin can update / give wallet balance to the Customer, Generate Receipt is to make receipt for transactions that has been made, and last is Log Out.

4.4.1 Manage User

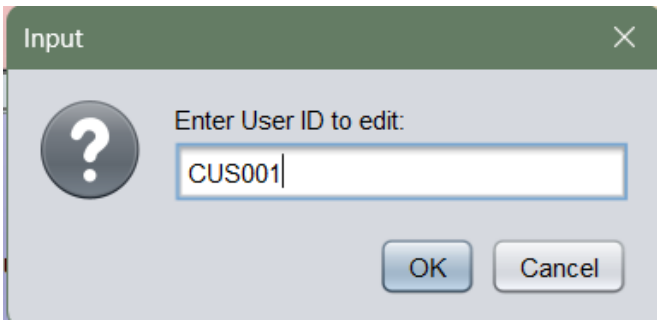


Figure 3.2: Editing Customer "CUS001"

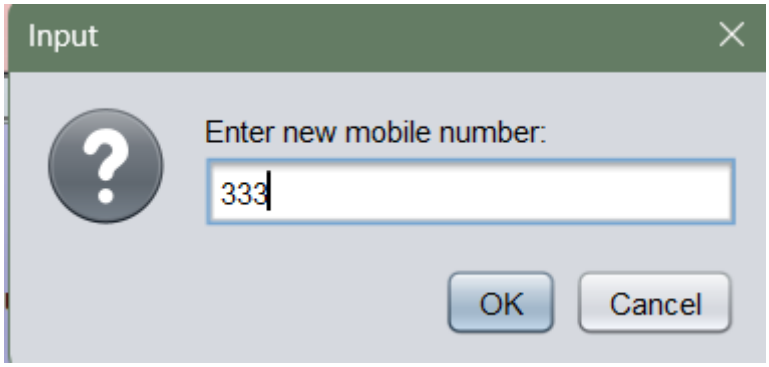


Figure 3.3: Editing Mobile Number Of "CUS001"

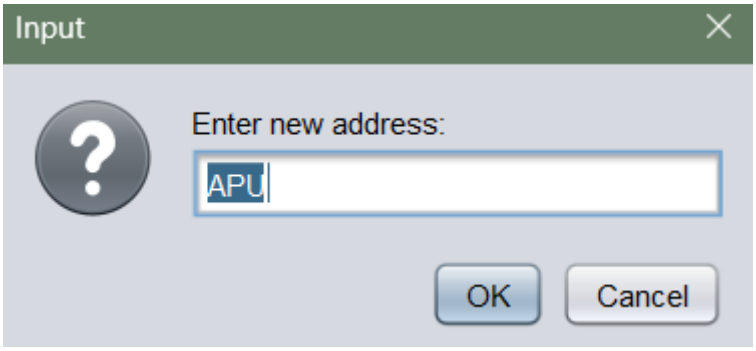


Figure 3.4: Editing Address Of "CUS001"

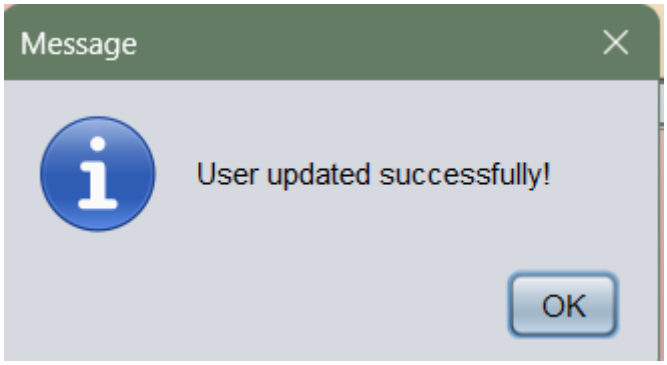


Figure 3.5: Notification Pop Up For Update Successfully

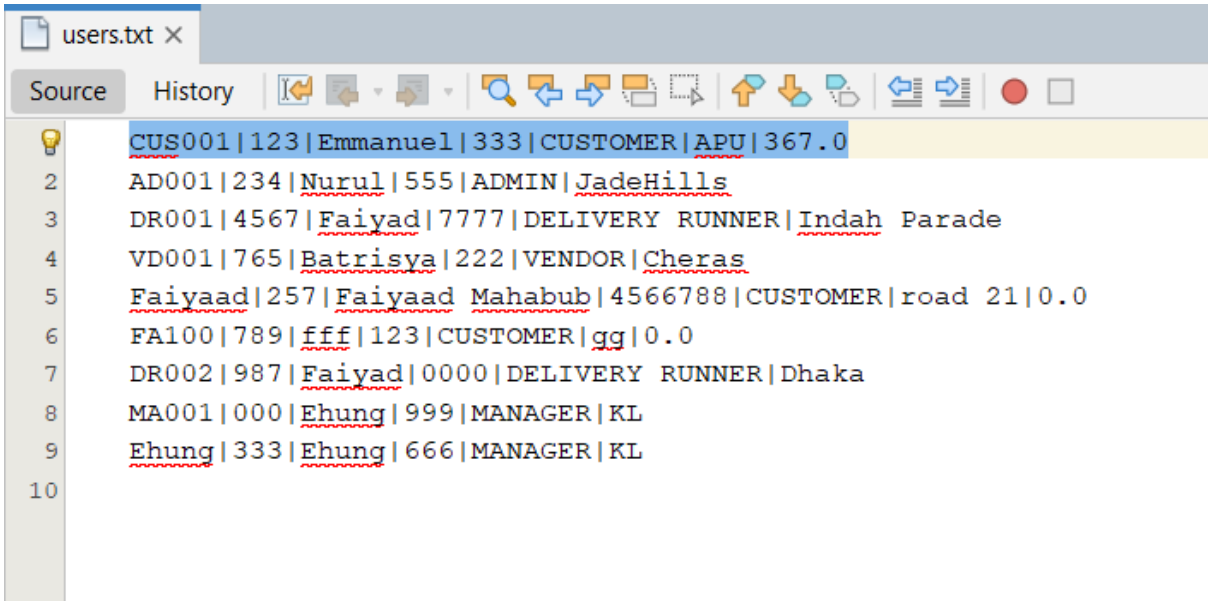


Figure 3.6: Checking "User.txt" to check if "CUS001" edited successfully.

This is where Admin run "Manage User" Menu, where he/she has to enter any role ID that is available (In the image is Customer with CUS001 as the userId), then Admin can edit phone number and address, then it will give a success notification and it will directly change in the users.txt file.

4.4.2 View Vendor

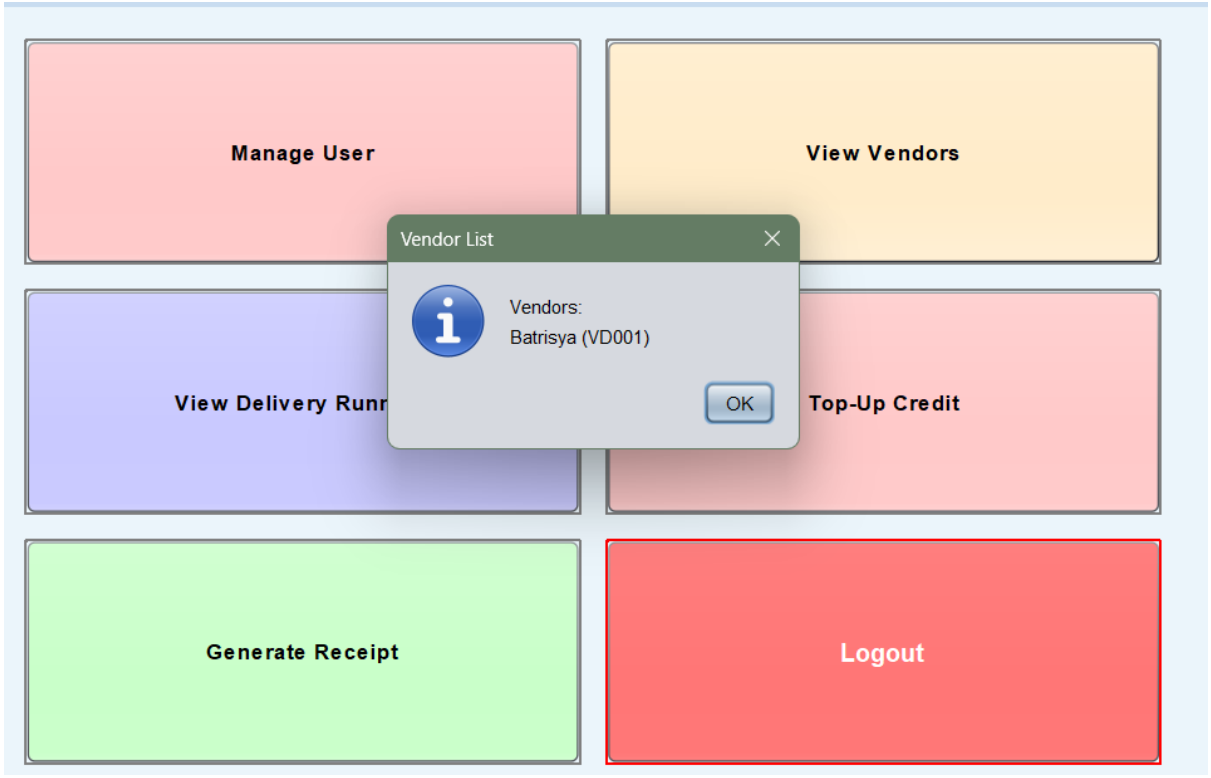


Figure 3.7: Viewing All Available Vendors.

4.4.3 View Deliver Runner

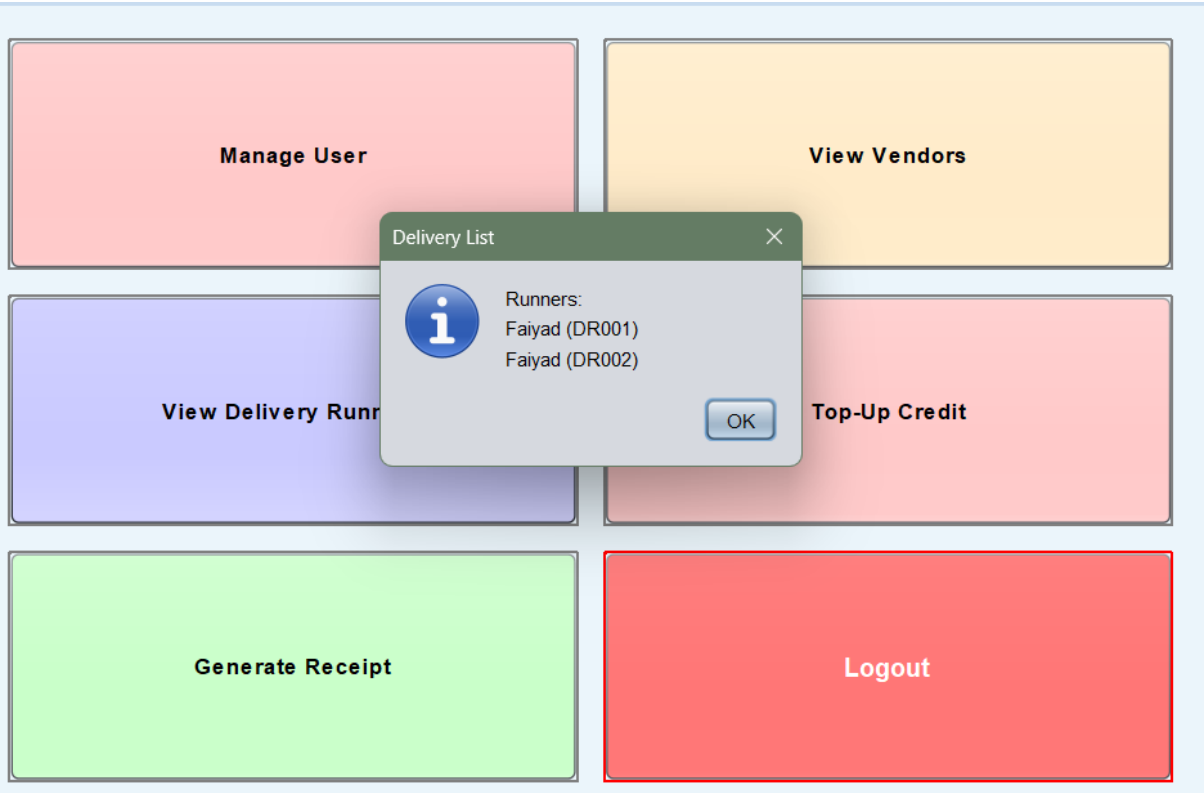


Figure 3.8: Viewing All Available Runners

Next, Admin can see which vendor and Delivery Runner are available. The first image indicates that only one vendor available under username of VD001. The second image shows there are 2 delivery Runners online that can be assigned to deliver the order.

4.4.4 Top Up Credit for Customer

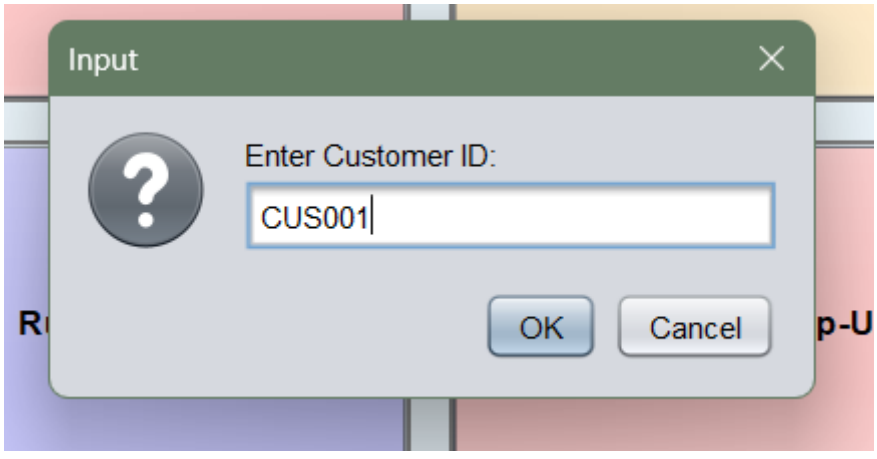


Figure 3.9: Entering Customer ID for Top Up Credit.

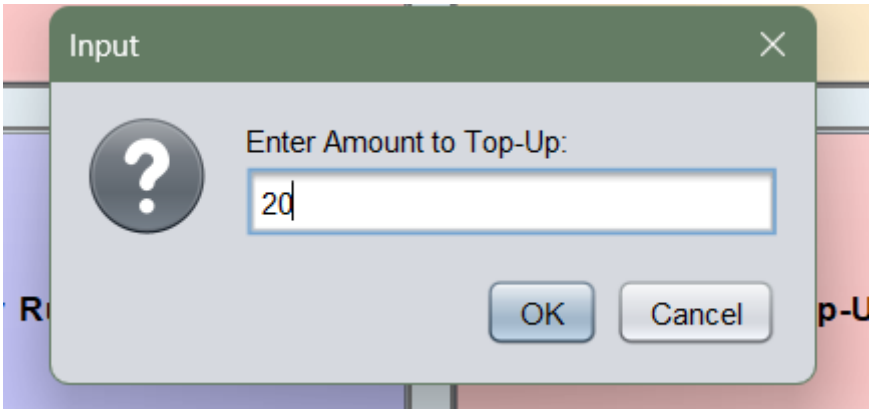


Figure 3.10: Entering Top Up Balance.

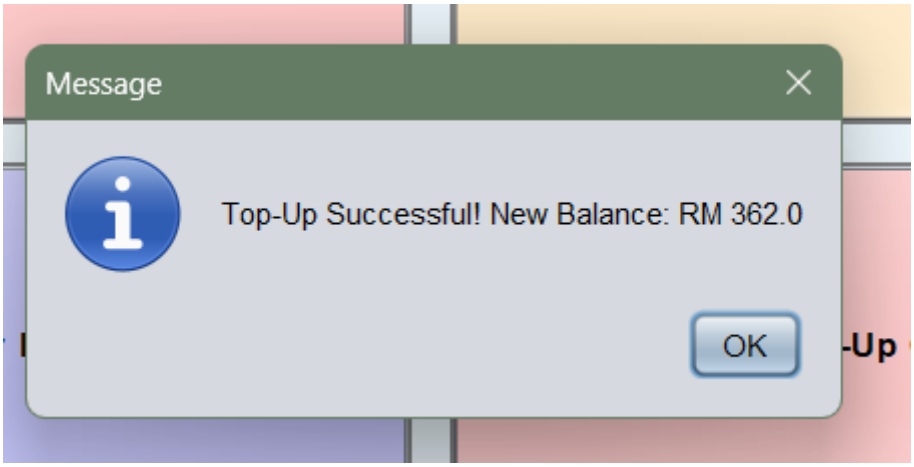


Figure 3.11: Notification Pop up for Top-Up Successful.

Admin can also top up customer's wallet balance by entering Customer ID who needs top up, after that enter the amount of balance and click "OK" (it will give you notification that the top up is success).

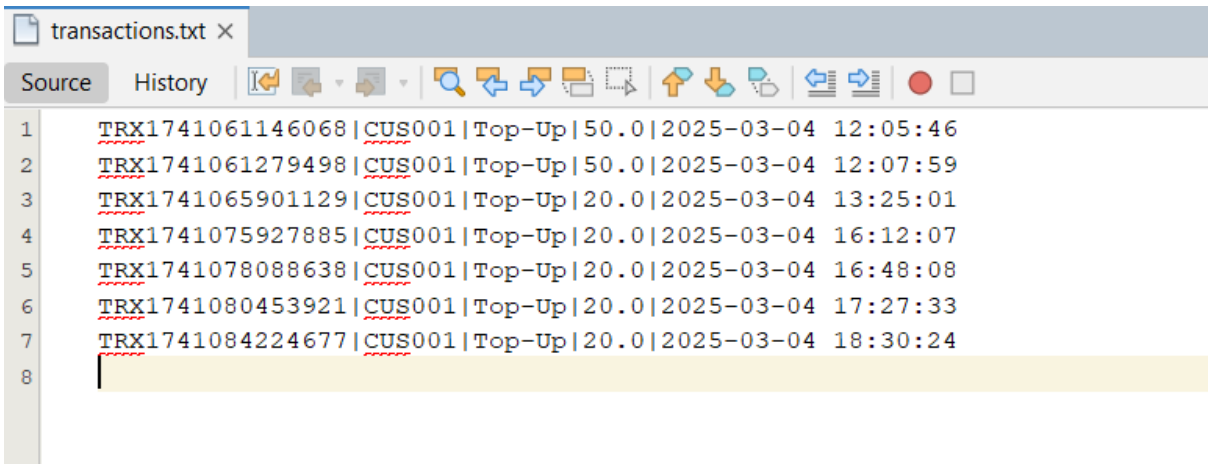


Figure 3.12: All Of the new and old transaction IDs present in "transaction.txt".

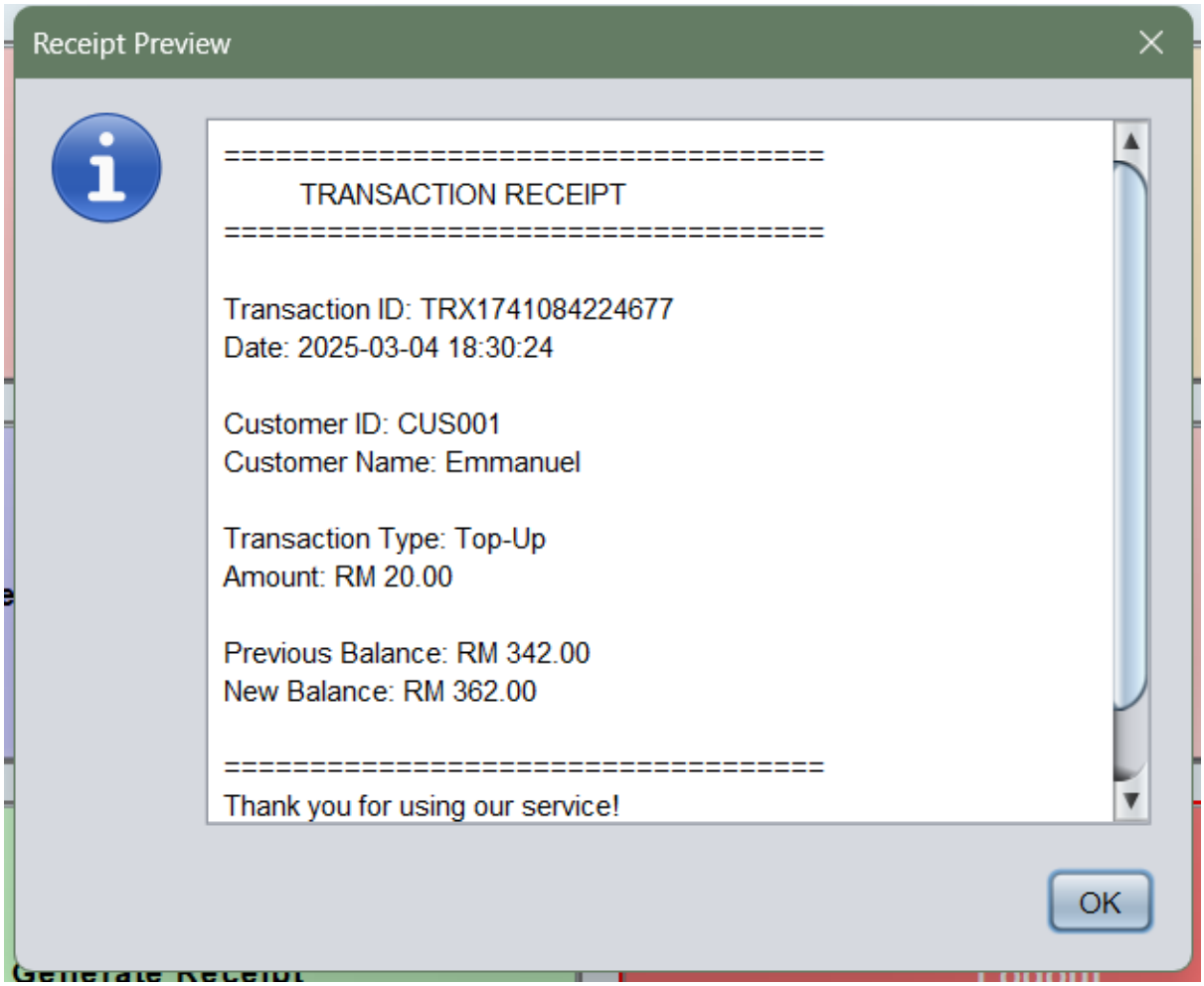


Figure 3.13: Receipt Preview Before Sending Customer.

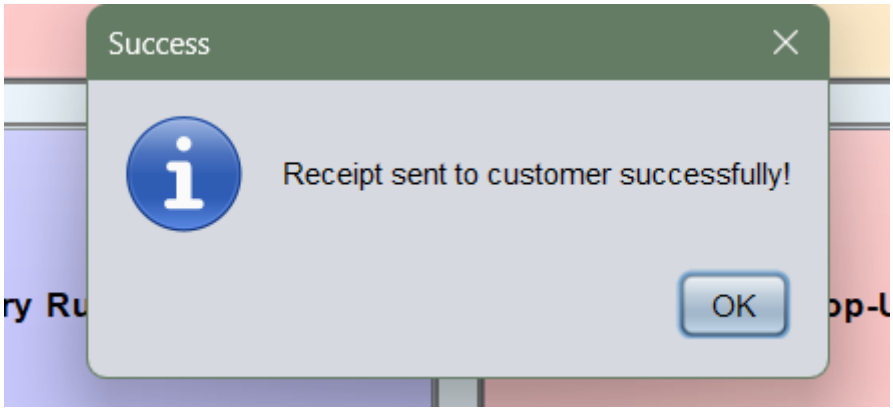


Figure 3.14: Pop up notification for sending receipt successfully to Customer.

After Admin successfully do the top up, it will give an option to Admin whether to send the receipt to the Customer or no. If Admin says “Yes”, it will give Admin a window to view the **Transaction Receipt** before sending it to Customer. If it’s done, then Admin can click OK and it will give notification that it is send successfully.

4.4.5 Generate Receipt

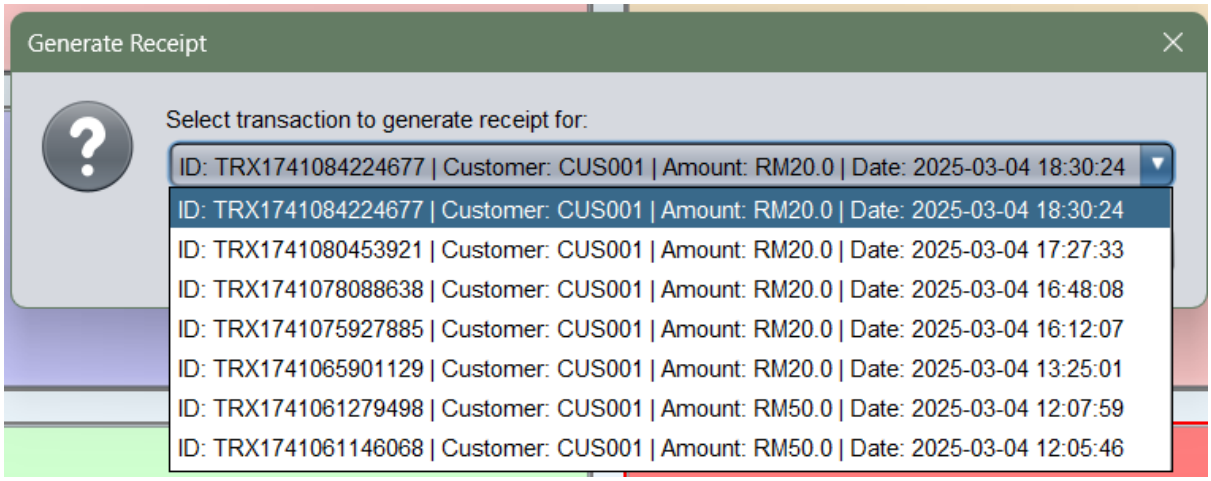


Figure 3.15: List of Previous Transaction History Showcased In A Combo Box.

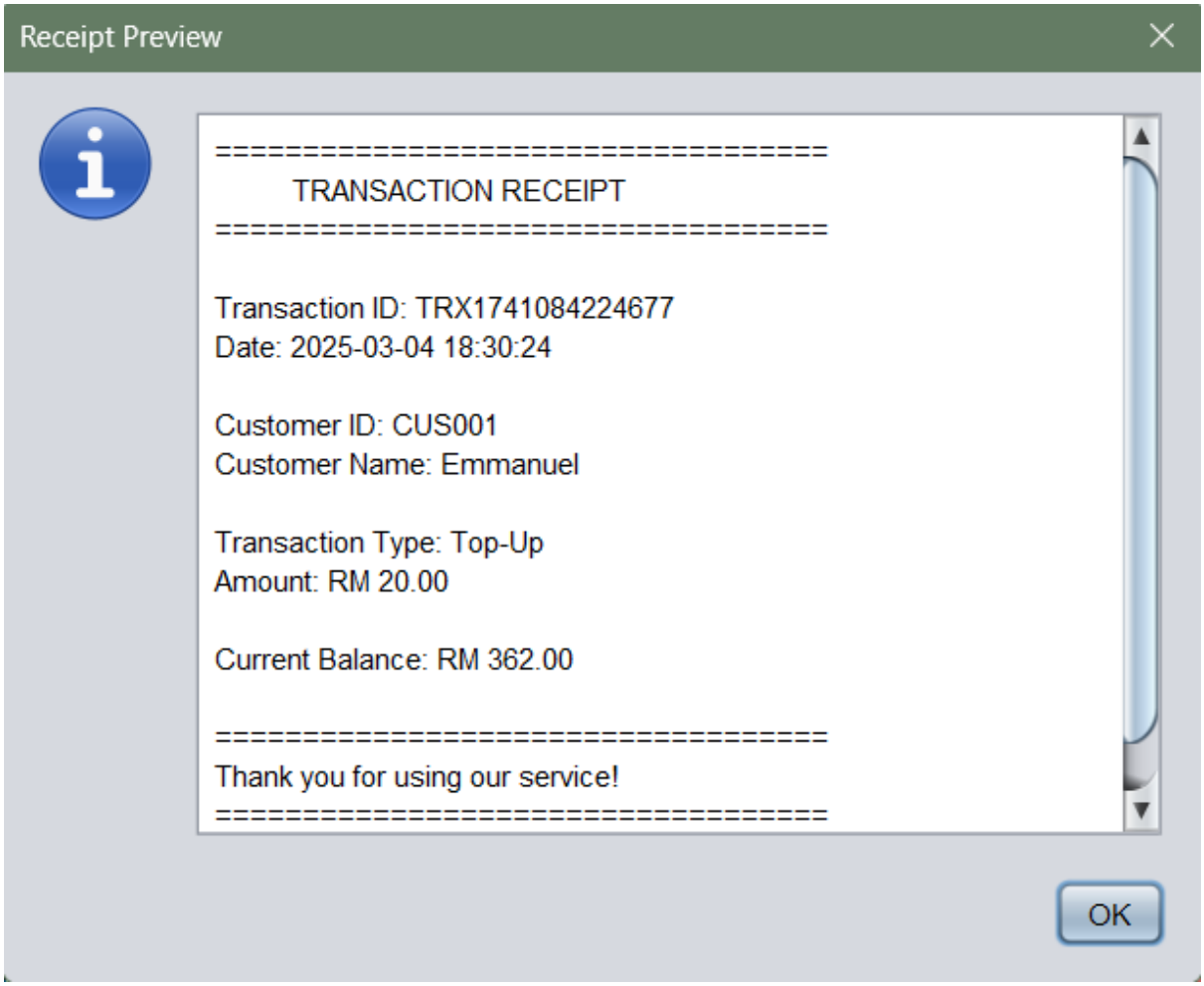


Figure 3.16: Choosing Previous Transaction Receipt In Order To Send To Customer.

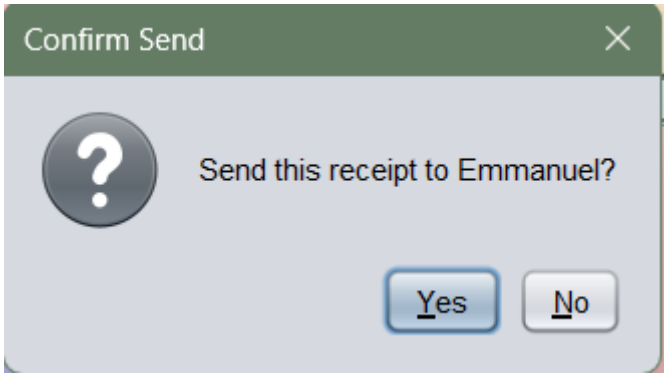


Figure 3.17: Pop up Confirmation Notification In Order To Send The Receipt Back To Customer or Not.

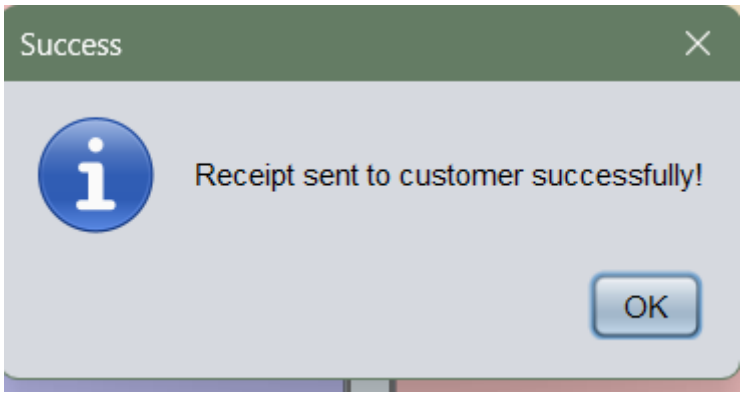


Figure 3.18: Pop up notification for sending transaction receipt successfully.

The last function is to **Generate Receipt.** With this, Admin can check back the transaction that they have made, by selecting the receipt. It will give the preview of it and will send it back to the Customer.

4.4 Customer

4.4.0 Customer Dashboard

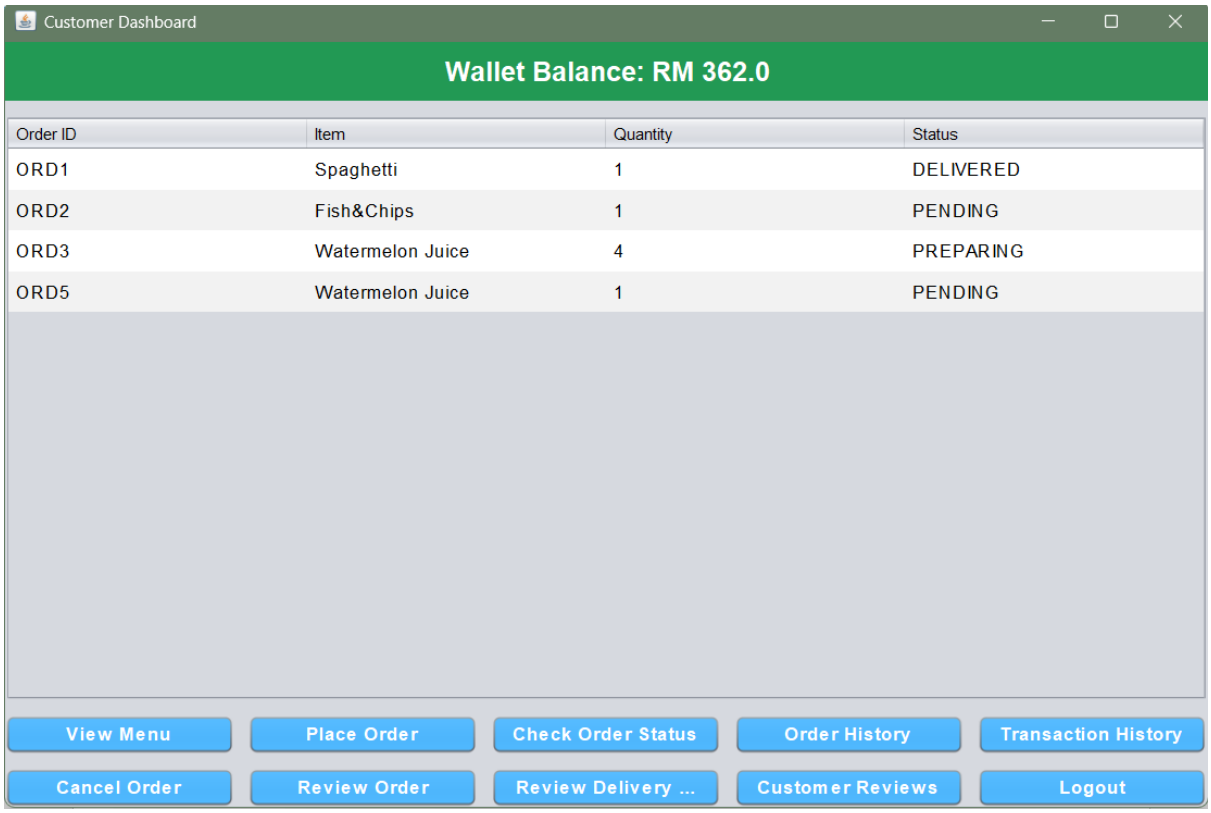


Figure 4.0: Customer Dashboard

This is the Customer Dashboard. It has 8 functions inside and table for viewing the current order taking place.

4.4.1 View Menu Items

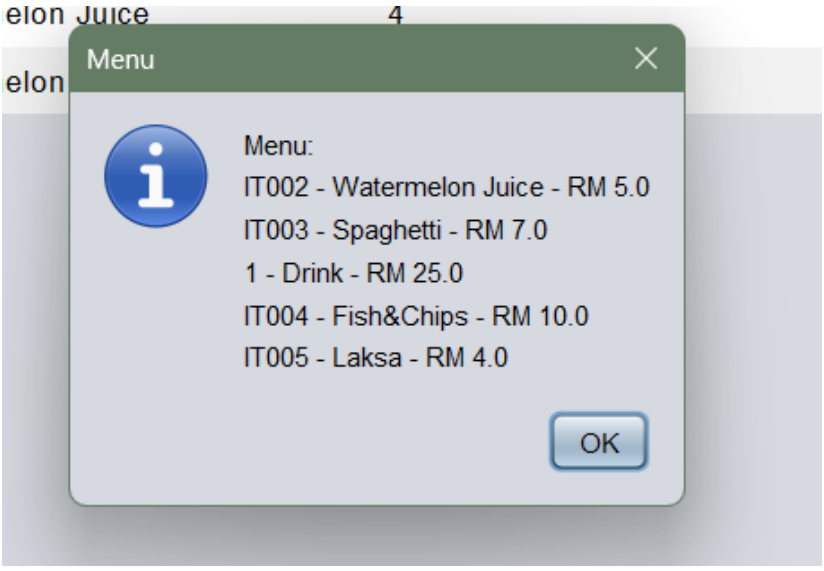


Figure 4.1: Viewing All Menus using the "View Menu" button from figure 4.0.

First function, Customer can view the menu that is provided inside the application.

4.4.2 Place Order

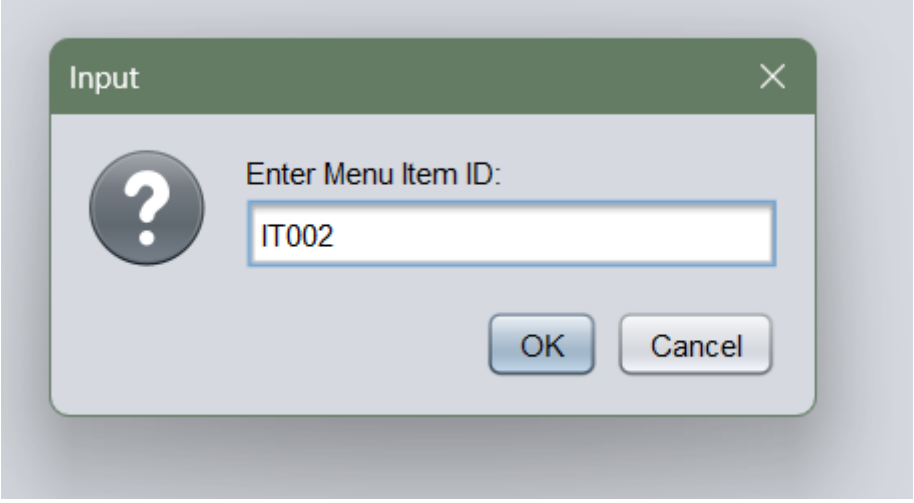


Figure 4.2: Enter into "Place Order" button And Provide Item ID for Choosing the Item Customer Wants.

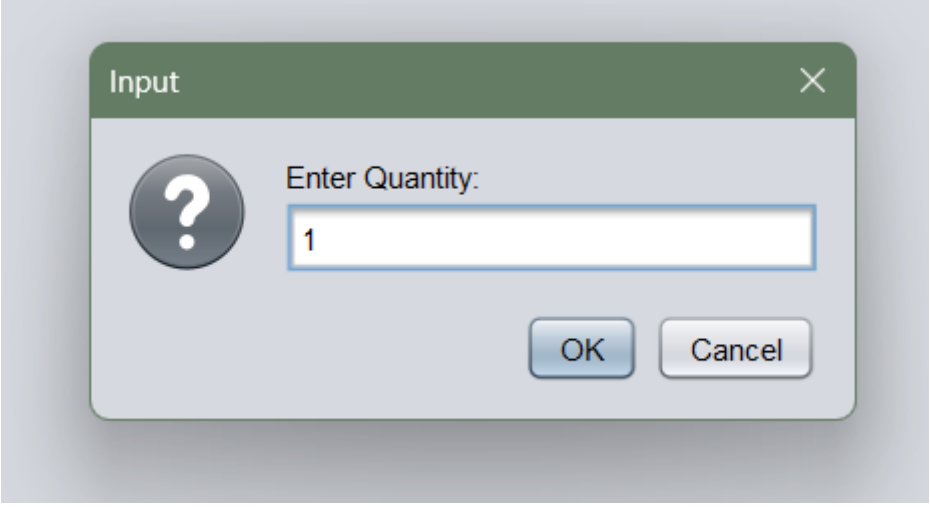


Figure 4.3: Enter The Amount For The Selected Item ID.

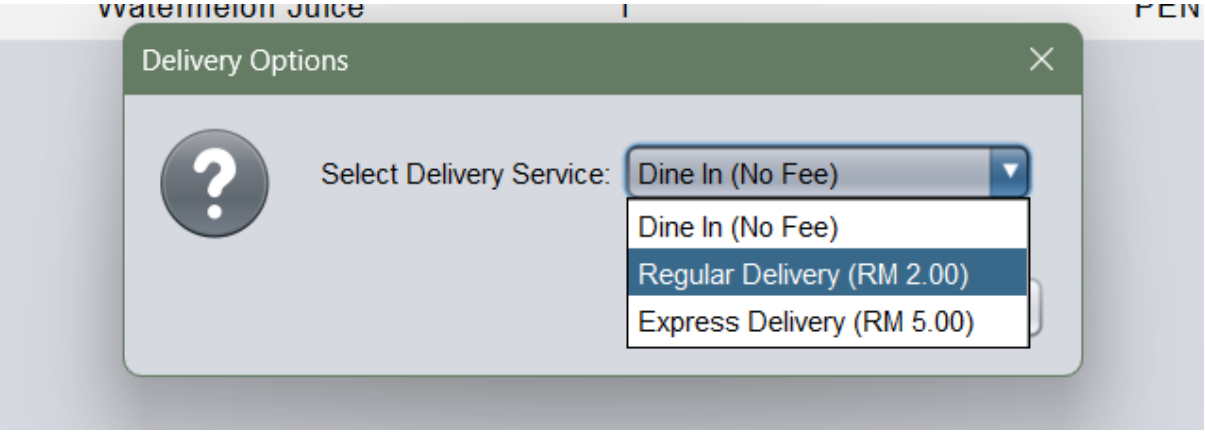


Figure 4.4: Select Delivery Service From Combo Box.

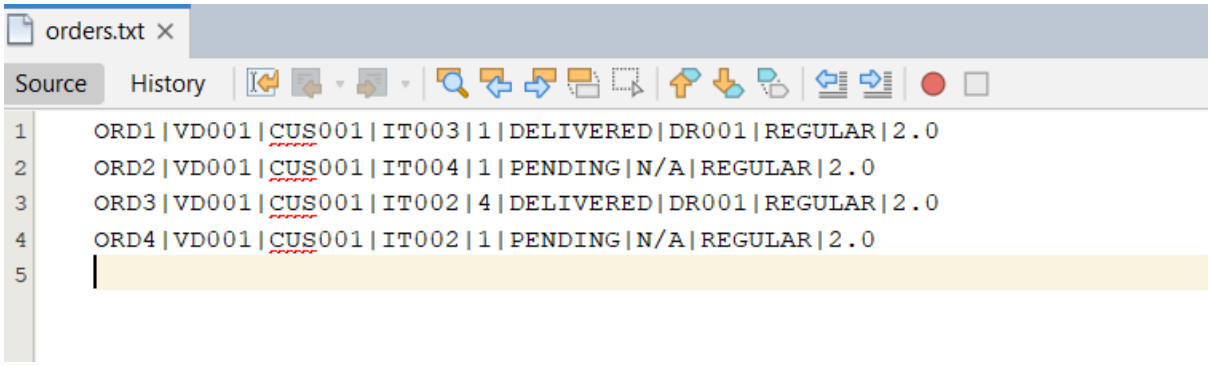


Figure 4.5: List of All Orders Present In "orders.txt" That Is Done After Placing Orders.

The second function is Customer can place order by putting the Menu Item ID , the quantity of the order, and select the delivery service they want (Currently our system has 3 delivery Service). After that, all the information that the customer key in, it will be saved in a text file called "orders.txt".

4.4.3 Check Order Status

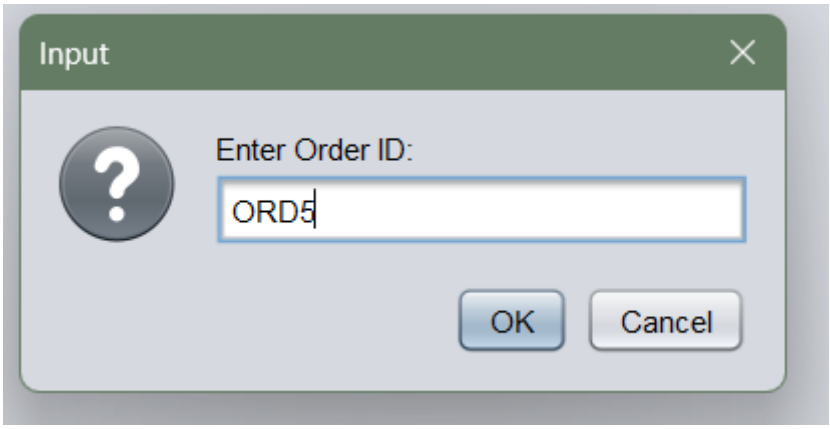


Figure 4.6: Enter Order Id to check for order status using "Check Order Status" button present in Figure 4.0.

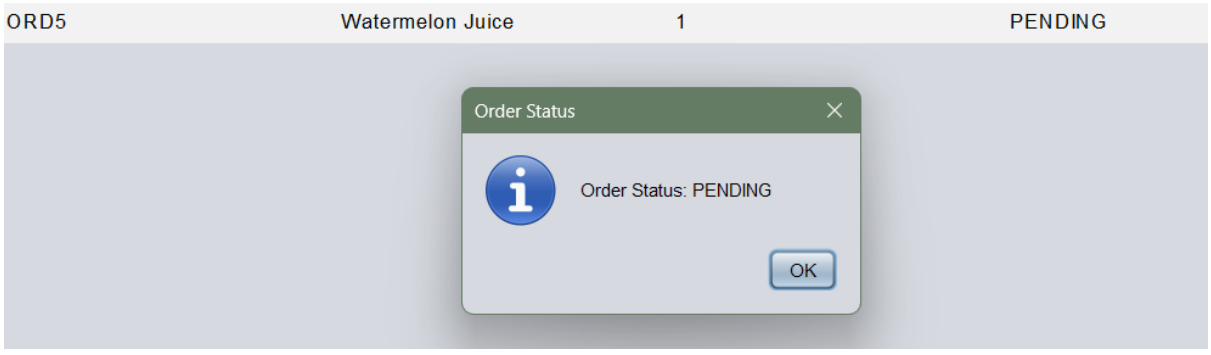


Figure 4.7: A Pop Up Notification Is Shown To Check For Order Status For Recent Orders That Is Placed By Customer Before.

The third function is **Order Status.** In here, Customer can view the status for each order by entering the order ID. Afterwards, it will pop out the selected order status.

4.4.4 Check Order History

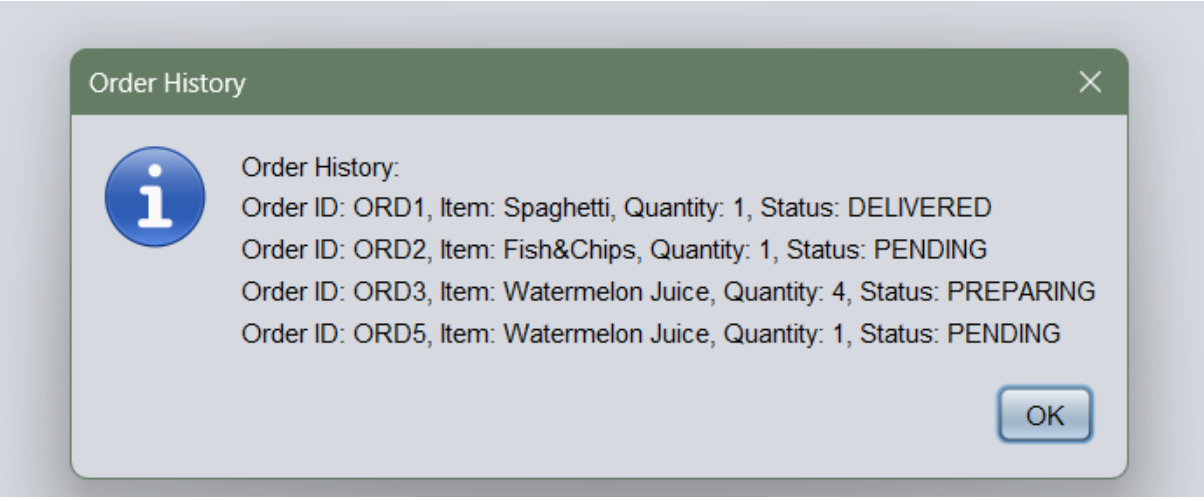


Figure 4.8: Checking All Of Previous Orders Placed By The Same Customer Using "Order History" Button At Figure 4.0.

The fourth function is **Order History.** Customer can see the previous order they took through this panel.

4.4.5 Check Transaction History

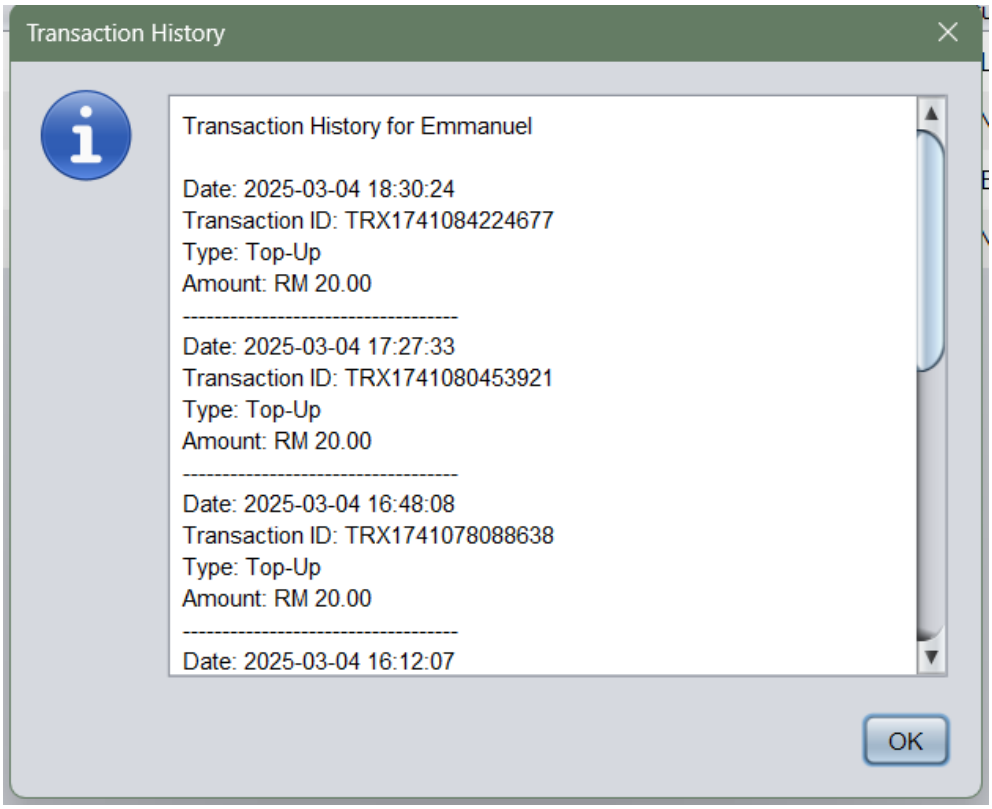


Figure 4.9: Checking All Transaction History Using "Transaction History" Button at Figure 4.0.

The Fifth function is **Transaction History.** Where Customer can see the previous transaction that have been made.

4.4.6 Cancel Order

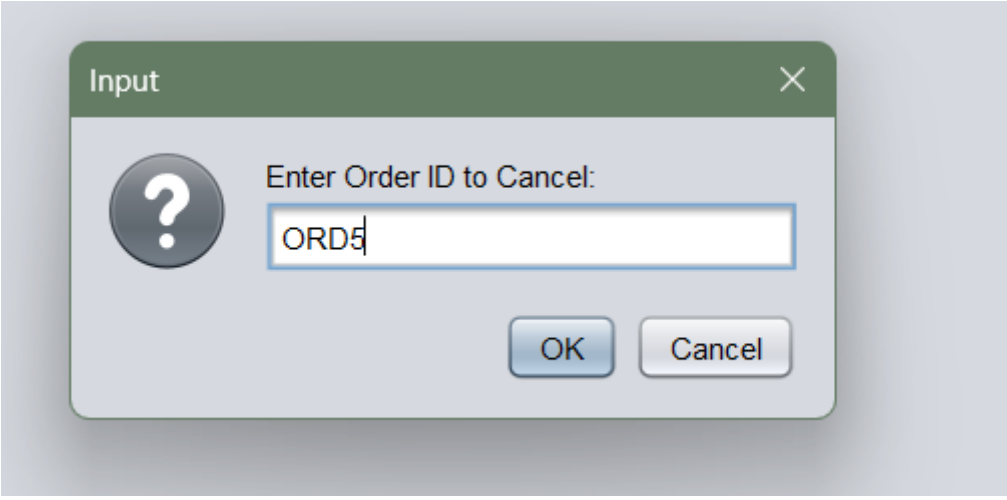


Figure 4.10: Entering Order ID For Choosing to Cancel Order using the "Cancel Order" Button At Figure 4.0.

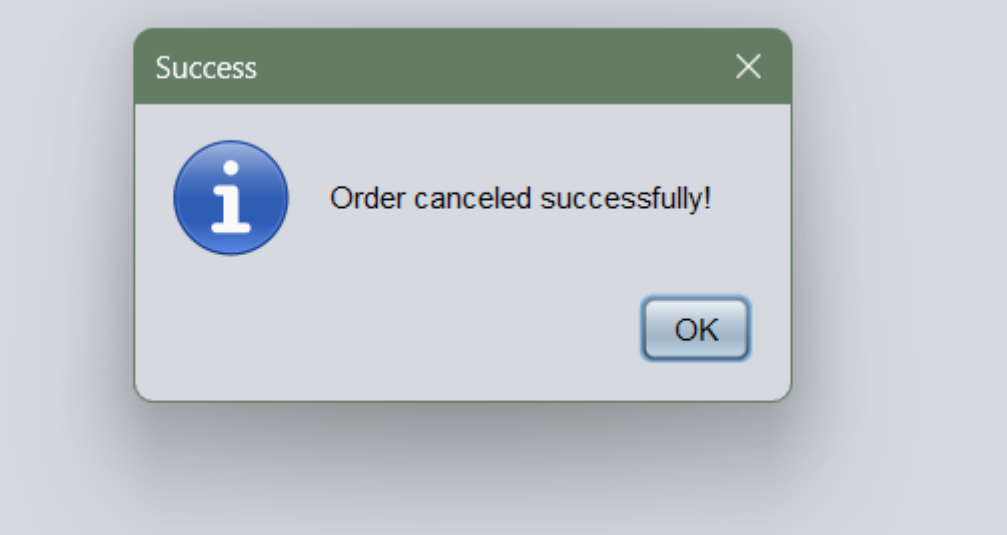


Figure 4.11: Pop Up Notification That Shows Order Has Cancelled Successfully.

The sixth function is **Cancel Order.** Where Customer can cancel the order from the list. The order data that got cancelled will automatically erased from the text file (orders.txt).

4.4.7 Review Order

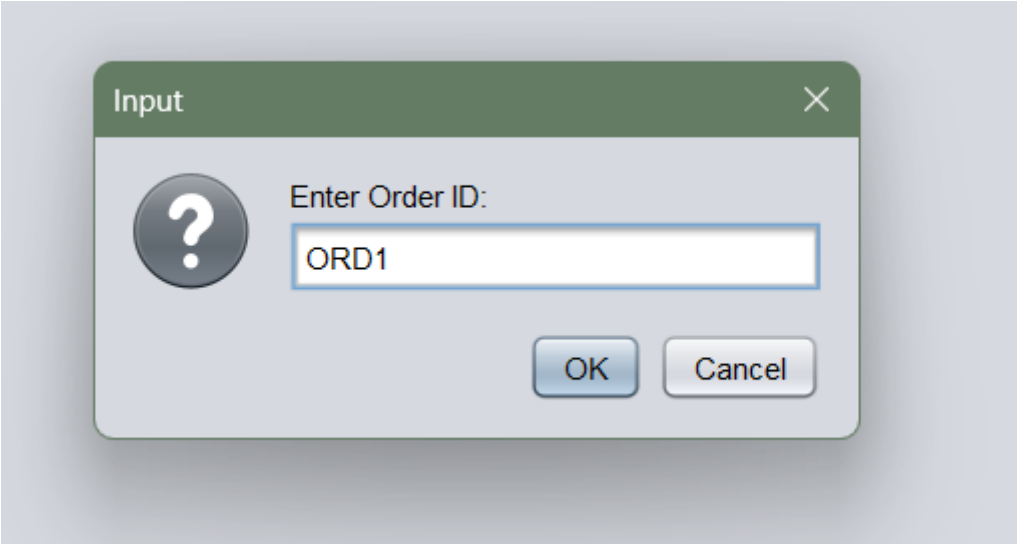


Figure 4.12: Enter Order ID For Reviewing Order using The "Review Order" Button at Figure 4.0.

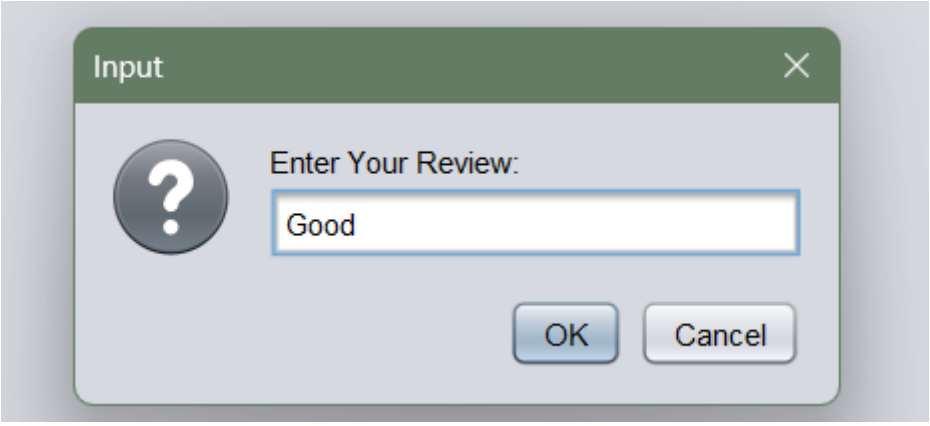


Figure 4.13: Enter Order Review For Order ID "ORD1".

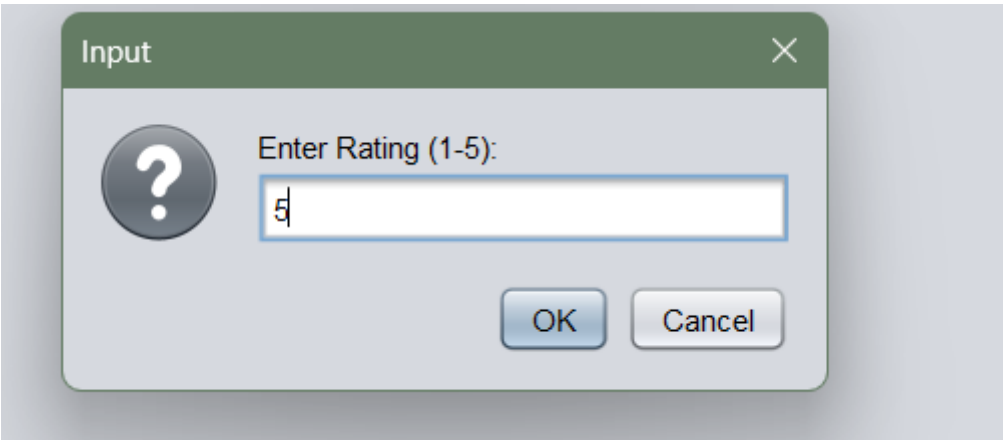


Figure 4.14: Input Ratings for Order "ORD1".

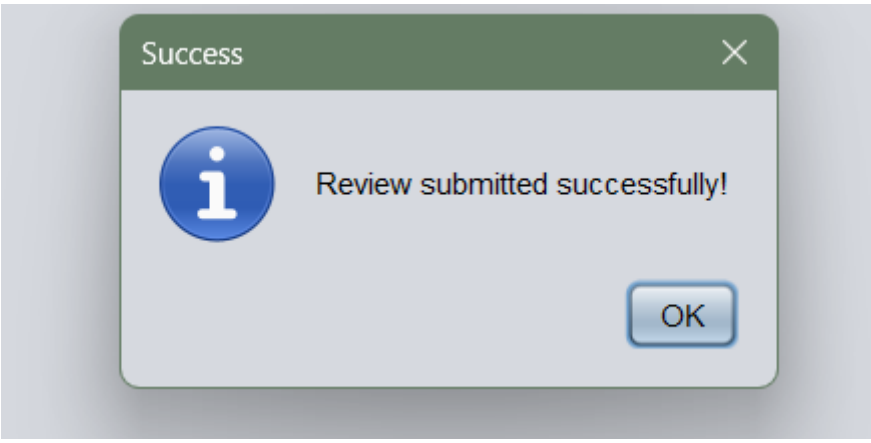


Figure 4.15: Pop up Notification To Show That Review Has Been Submitted Successfully.

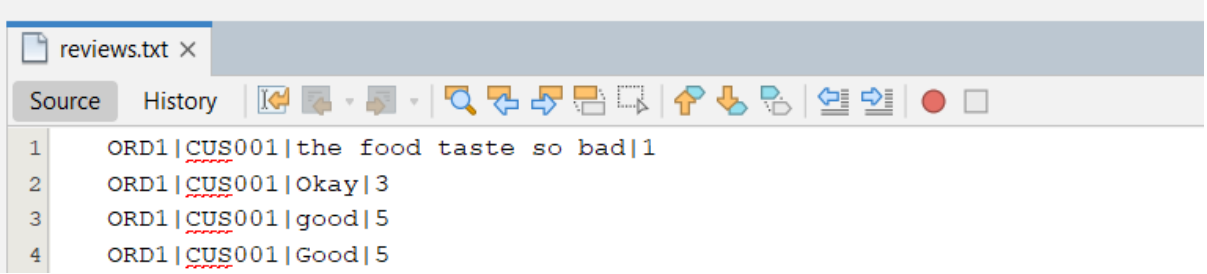


Figure 4.16: Showing All Other Reviews Given By Customers, Which Is Saved Inside "reviews.txt".

The seventh function is to **Review the Order.** This function will asked customer to review the order they choosed. In the first image, it asked customer to enter the order ID, then customer can give their honest review and rating like shown in the second and third image. Then customer can send the review, and it will give an auto pop up message indicating the review has been submitted successfully and it saves in 'reviews.txt' file.

4.4.9 Review Delivery Runner

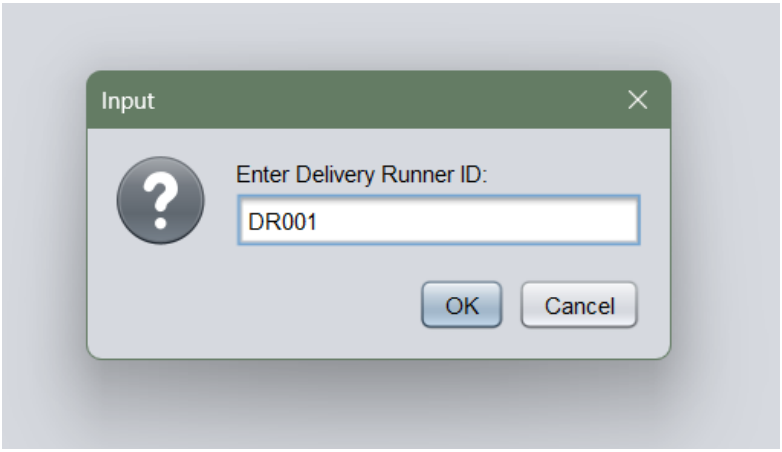


Figure 4.17: Enter Delivery Runner ID For Delivery Runner Review Using "Review Delivery Runner" button from Figure 4.0.

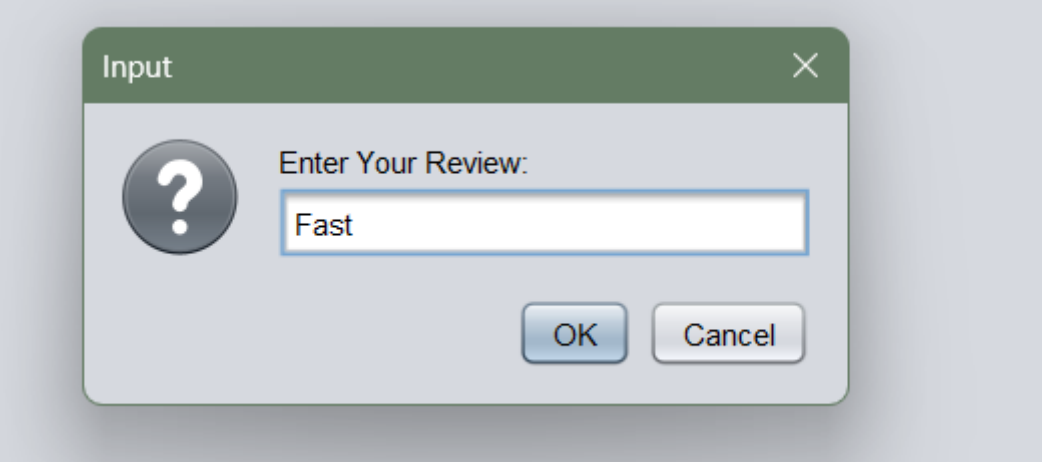


Figure 4.18: Enter Review For Deliver Runner.

Figure 4.19: Enter Rating for Delivery Runner.

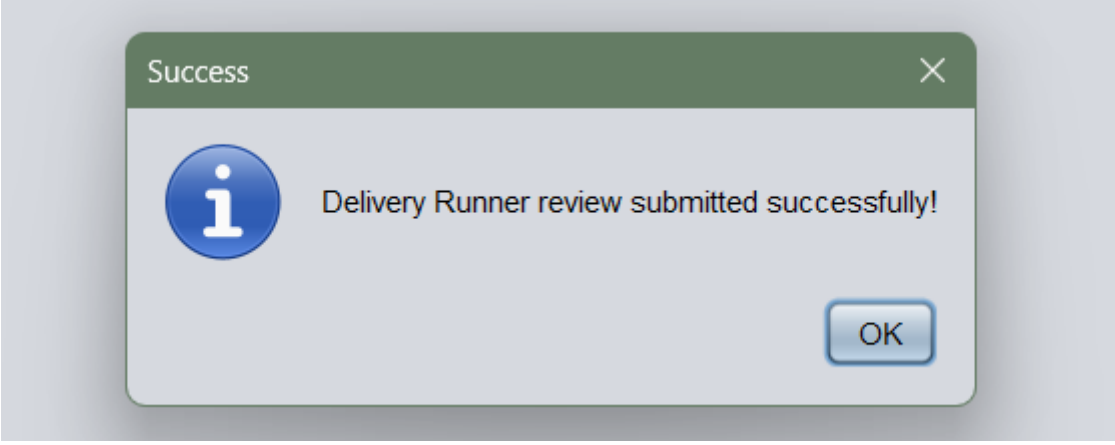


Figure 4.20: Pop up Notification For Showcasing That The Review Given By Customer Was Successfully Submitted.

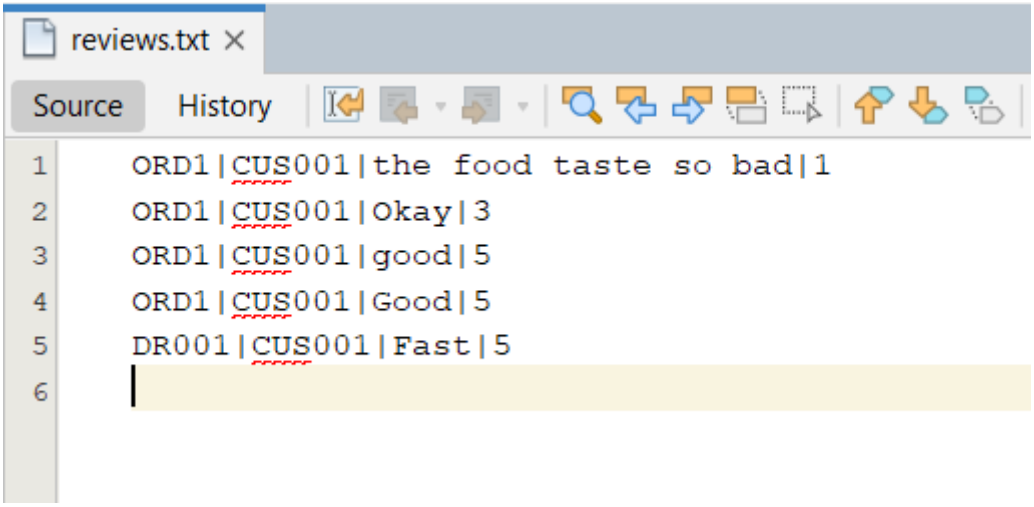


Figure 4.21: Showing All Reviews That Are Present In "Review.txt"

The eight function is **Review Delivery.** This function has the same job as review order but different object, where it ask customer to give review and rating to the Delivery Runner. After it is done, it will give pop up notification saying, "The Delivery Runner review has been submitted successfully" and saves in 'reviews.txt' file.

4.4.9 Check Review History

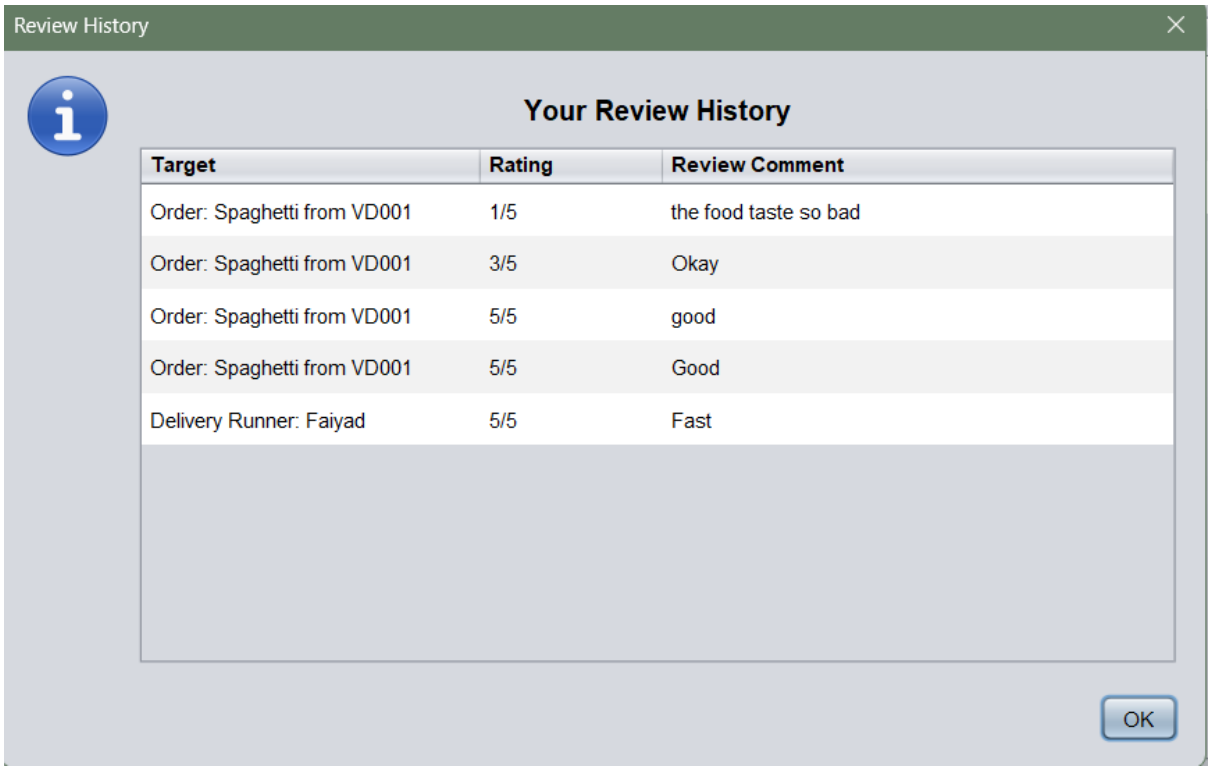


Figure 4.22: Showing All Reviews Done By The Customer Previously Using "Customer Review" Button From Figure 4.0.

The ninth function is **Review History.** This function will help customer to see other customers past reviews. Customer can see from bad review to the good ones.

4.5 Vendor

4.5.0 Vendor Dashboard

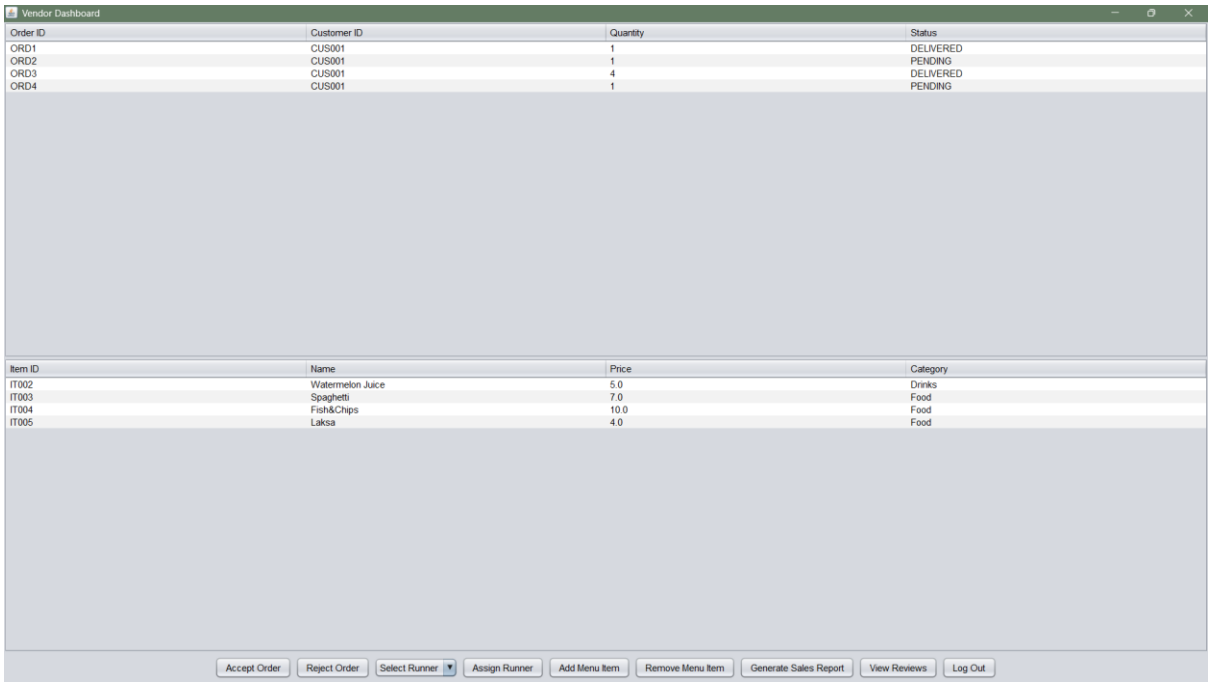


Figure 5.0 Vendor Dashboard

The image above shows Vendor Dashboard where it consists of two tables, the first table indicates order lists, and the second table indicates menu item lists.

4.5.1 Accept Order

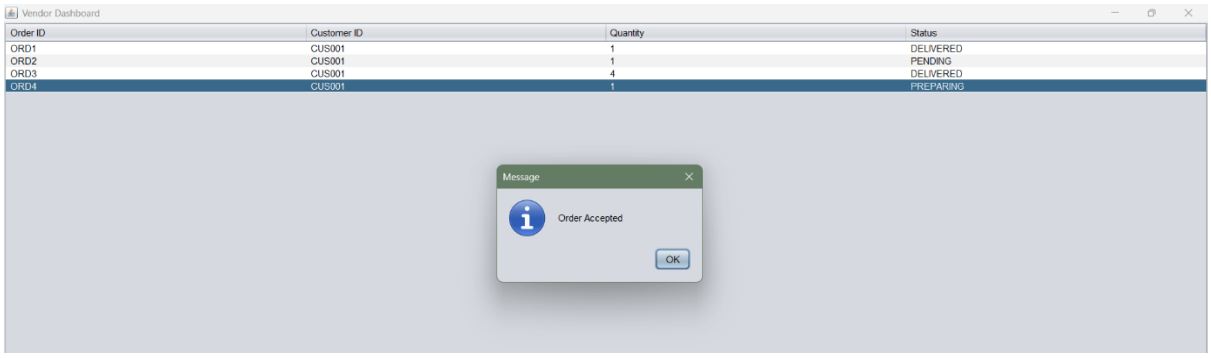


Figure 5.1 Vendor Accept the Order

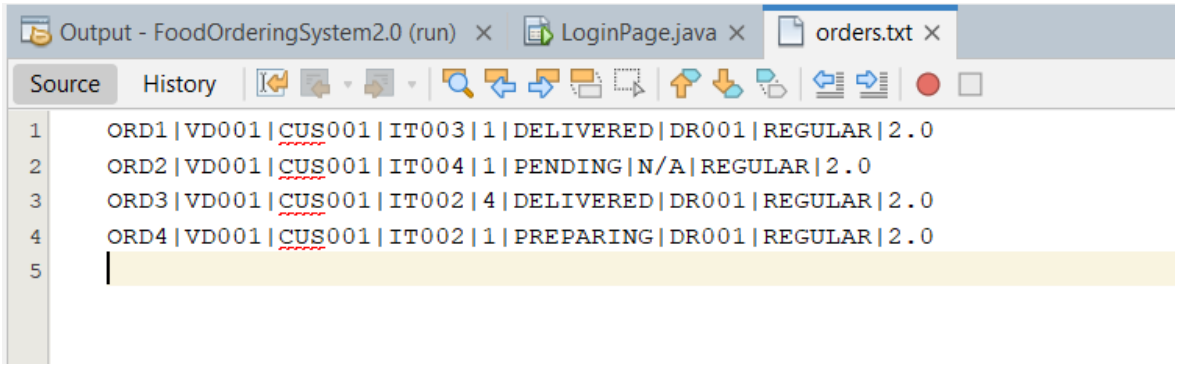


Figure 5.2 Orders Text File

The first function is **Accept Order.** The Vendor can select and order to accept. Once an order is accepted, the status changes from PENDING to PREPARING indicating that the vendor has started preparing the order. The order status also changed in the text file from PENDING to PREPARING.

4.5.2 Assign Delivery Runner

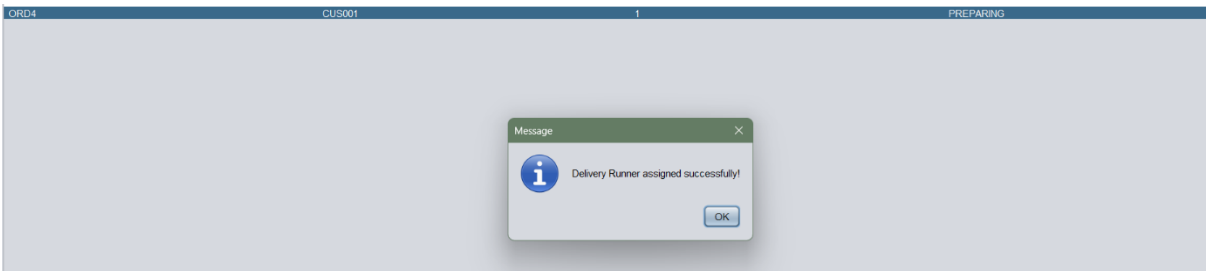


Figure 5.3 Delivery Runner got Assigned

The Second function is Assign Delivery Runner, after the vendor accept the order. It will ask vendor to choose the delivery runner, once assigned, a notification appears confirming that the delivery runner has been successfully assigned.

4.5.3 Add Menu Item

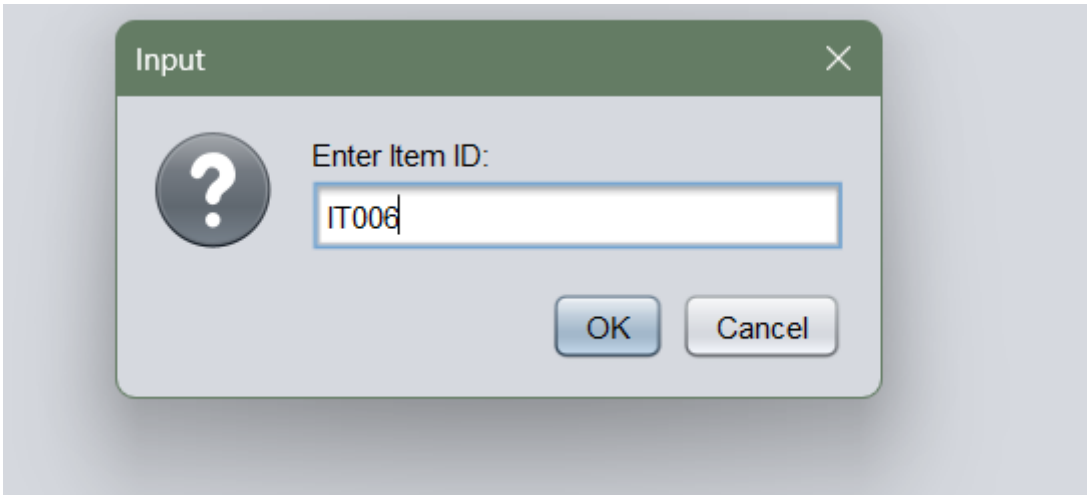


Figure 5.4 Enter Item ID

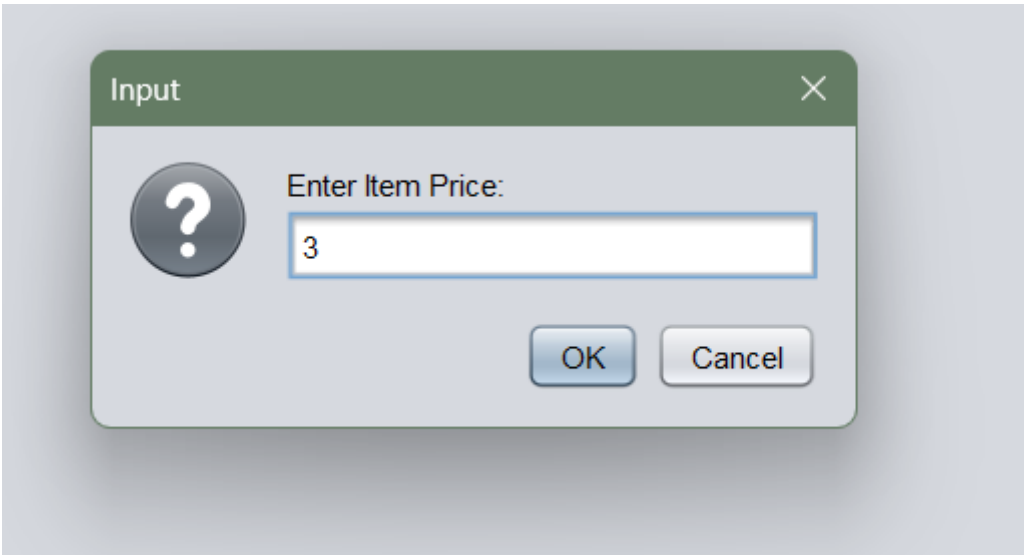


Figure 5.5 Input Dialog Box to input Item Price

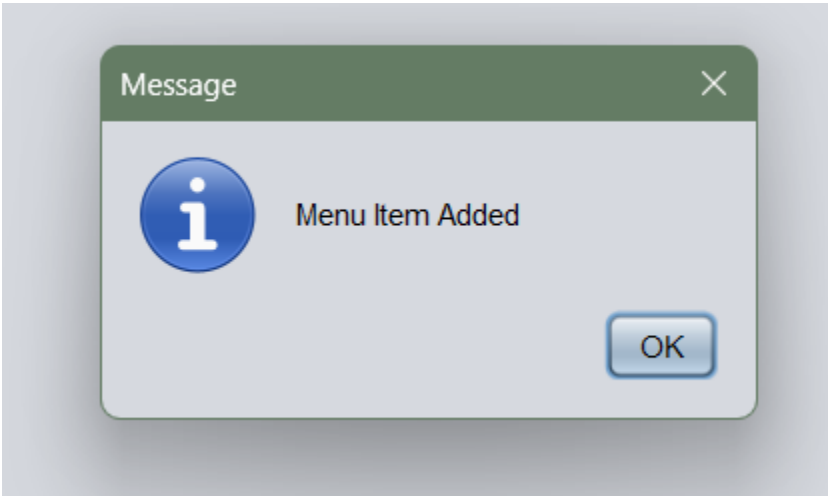
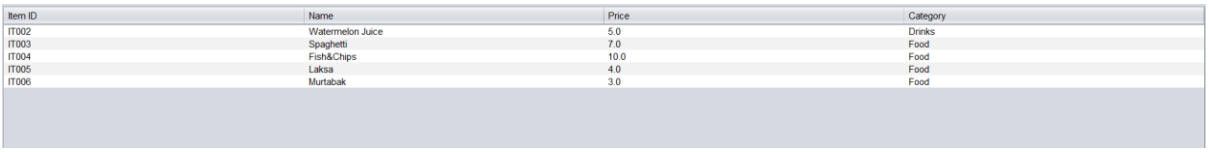


Figure 5.6 Item Added Notification



Item ID	Name	Price	Category
IT002	Watermelon Juice	5.0	Drinks
IT003	Spaghetti	7.0	Food
IT004	Fish&Chips	10.0	Food
IT005	Laksa	4.0	Food
IT006	Murtabak	3.0	Food

Figure 5.7 Menu Item List

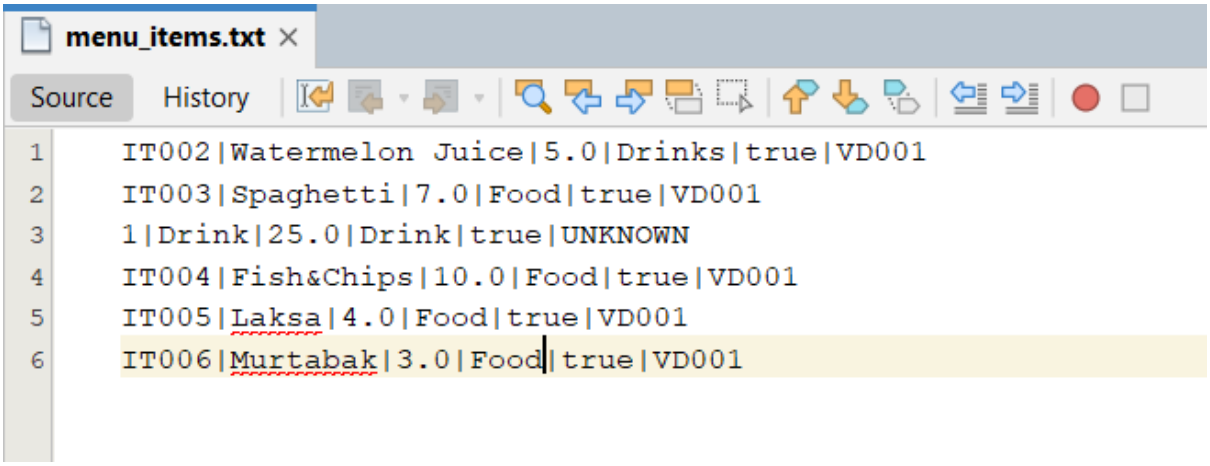


Figure 5.8 Menu Items Text File

The third function is **Add Menu.** In here, vendor can add new item by inputting the Item ID as shown in the first image, then Item Price (shown in image 2). After done with adding the menu, it will give you a pop-up notification saying the menu item has been added to the list. We can see in image 4 that IT006 (the menu item that we add) is listed in there. Not just in the table, it will also update in"menu_items.txt" file.

4.5.4 Remove Item

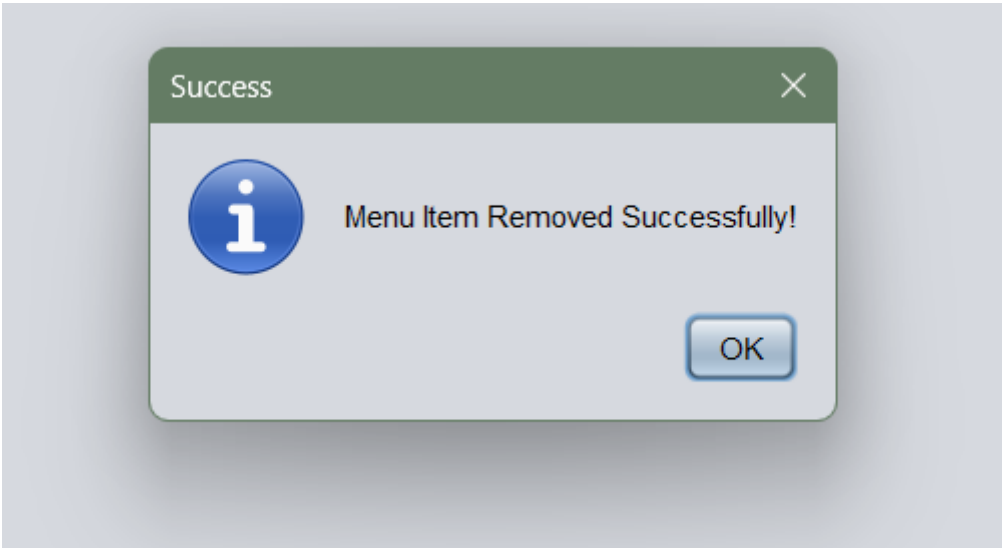
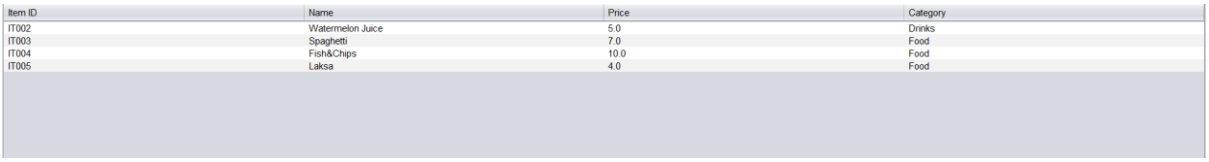


Figure 5.9 Notification of Item Remove Successfully



Item ID	Name	Price	Category
IT002	Watermelon Juice	5.0	Drinks
IT003	Spaghetti	7.0	Food
IT004	Fish&Chips	10.0	Food
IT005	Laksa	4.0	Food

Figure 5.10 Item got removed from the Menu List

The fourth function is **Remove Menu.** Vendor can also remove menu by selecting specific order and clicking the remove button. It will give a pop-up notification confirming that the menu item removed successfully. In the Menu table, the selected menu (IT006) will be deleted.

4.5.5 Vendor's Revenue Dashboard

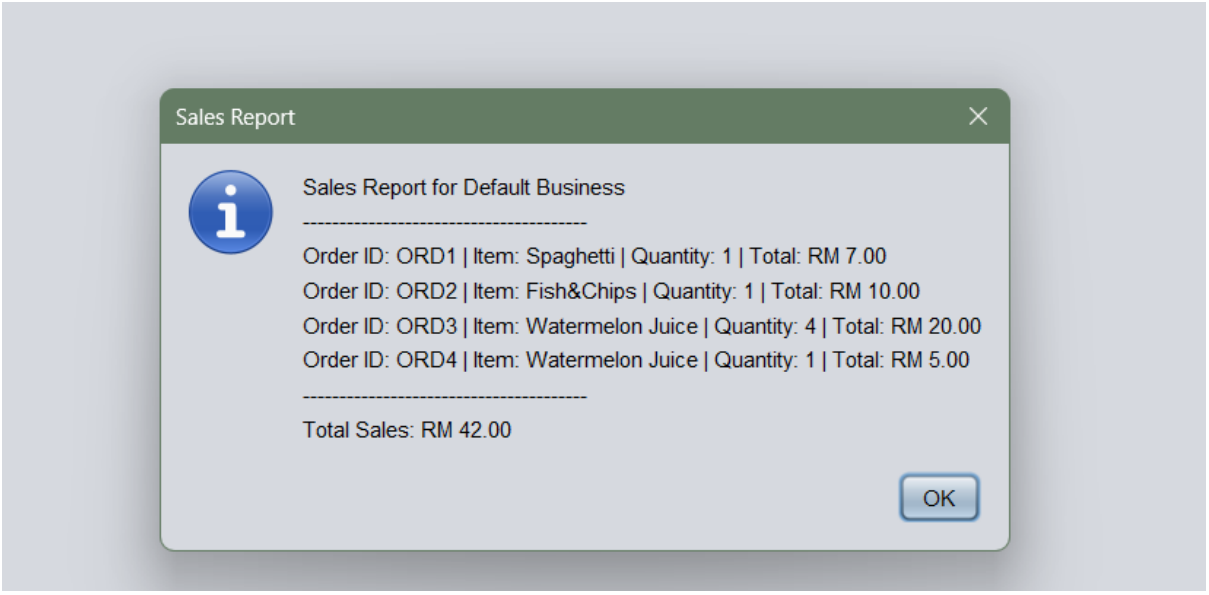


Figure 5.11 Revenue Report Summary

The fifth function is **Generate Sales Report.** Vendor can view the revenue earned from orders. The report includes a list of all orders along with their prices and a total revenue summary.

4.5.6 View Reviews

Figure 5.12 All Customer Reviews

The sixth function is **Customer Review.** This function allows vendor to see all customer reviews for their orders. Vendor can see which customer rated which order, along with the rating and review details.

4.6 Delivery Runner

4.6.0 Delivery Runner Dashboard

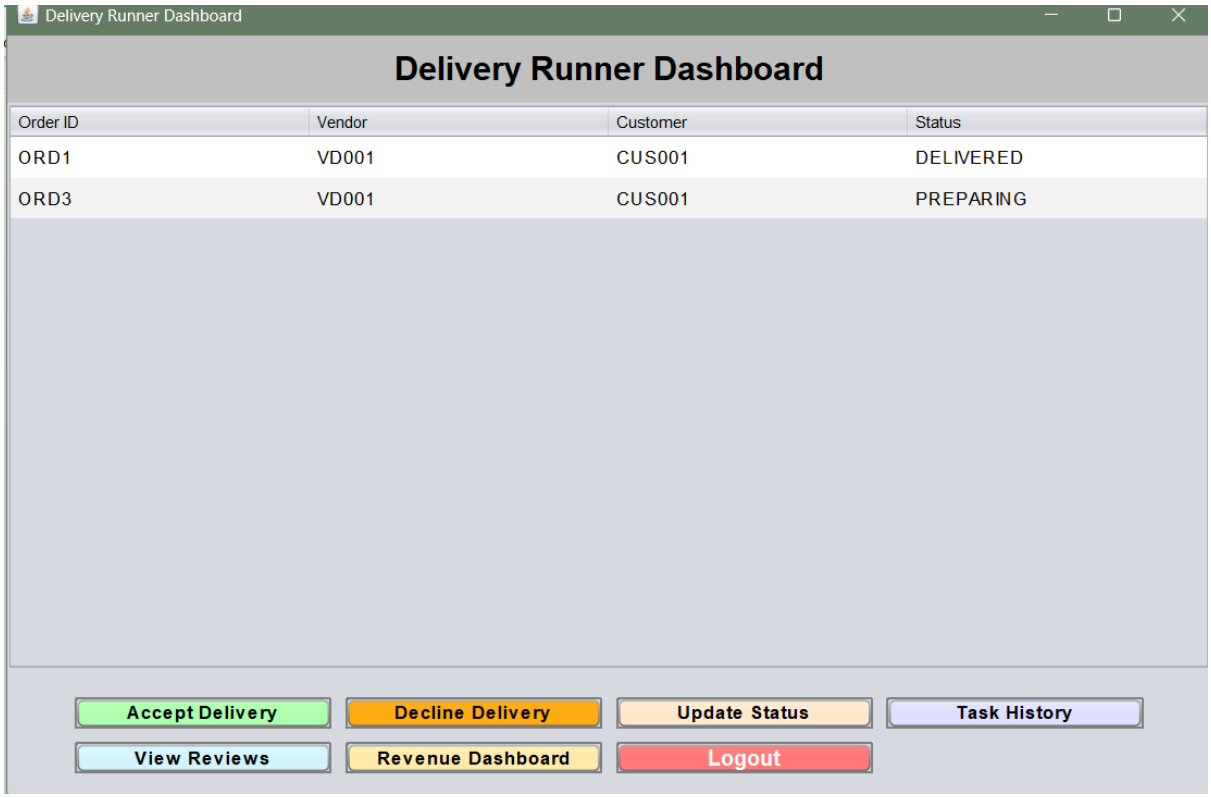


Figure 6.0: Delivery Runner Dashboard

The delivery runner dashboard shows all of the orders from vendors after assigning a particular delivery runner which is done by vendor.

4.6.1 Accept Delivery

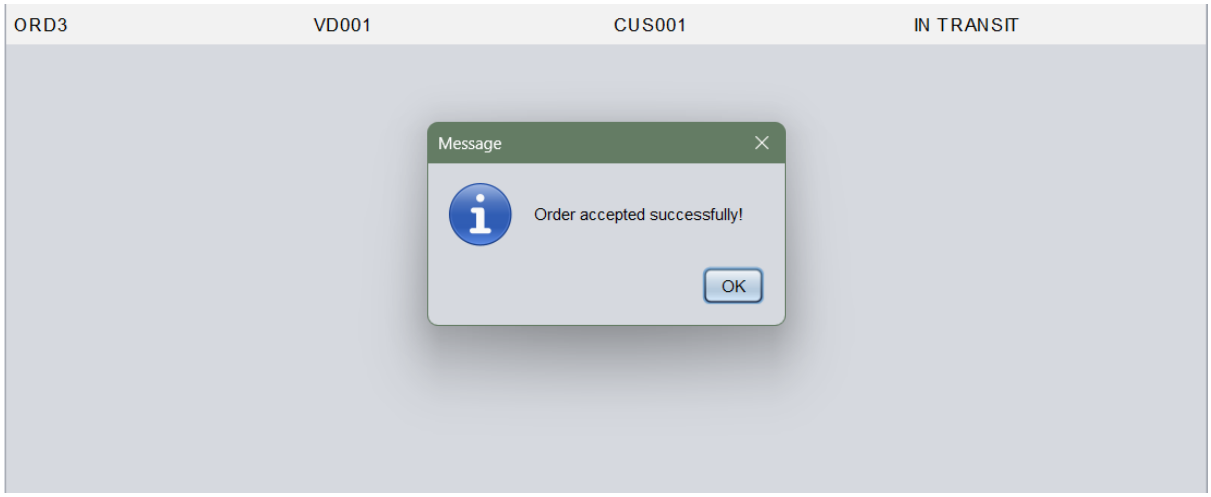


Figure 6.1: Pop up Notification Showing That Order Has Been Placed Successfully Where "Status" Changes From "PREPARING" To "IN TRANSIT"

Then for Order ID “ORD3”, the delivery runner chooses to accept the order and shows up a pop up notification message saying that the order has been successfully placed. The “Status” in the dashboard changes from “PREPARING” to “IN TRANSIT”.

4.6.2 Decline Delivery

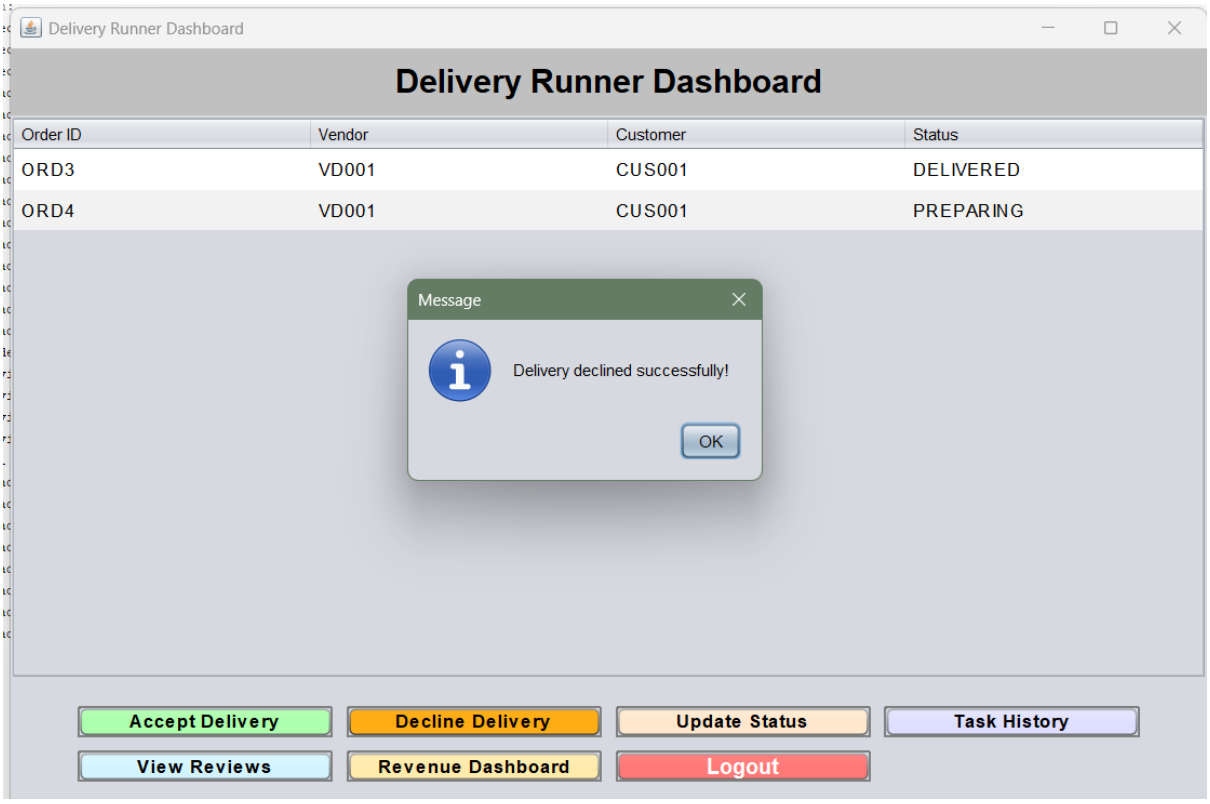


Figure 6.2 Decline delivery runner task

The customer can also choose to decline the order where he choose “ORD1” and it is removed from the dashboard. A pop up notification message is seen showing that the delivery was declined successfully.

4.6.3 Update Delivery Status

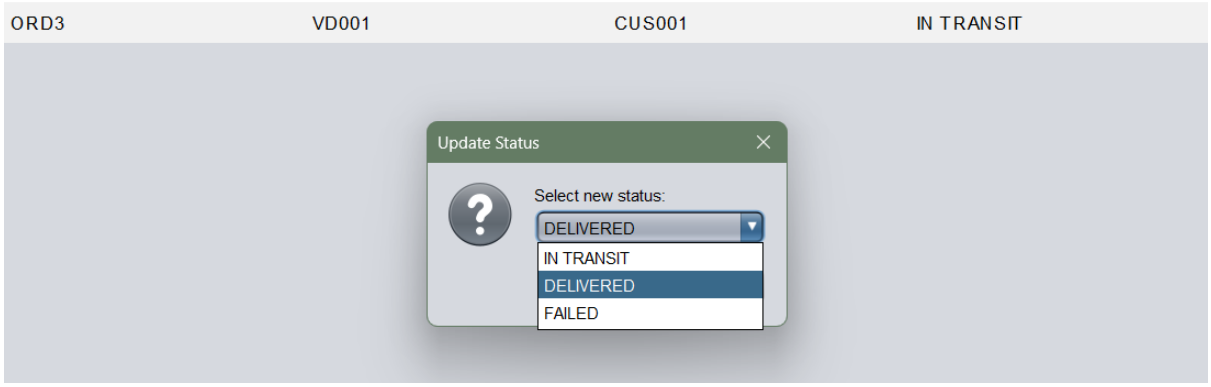


Figure 6.3: Update Delivery Status Using The "Update Status" button from figure 6.0.

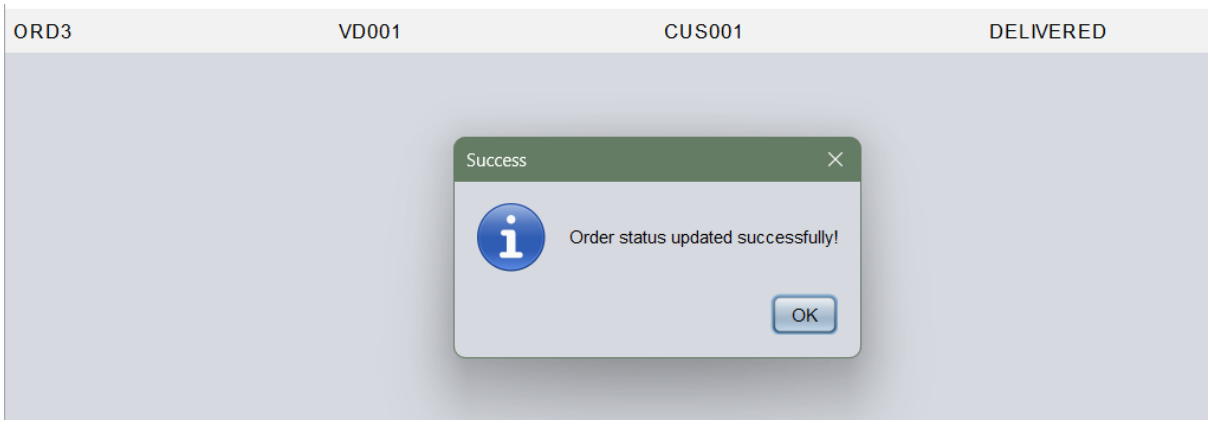


Figure 6.4: Pop up Notification to Show That Status Has Been Changed To "DELIVERED"

We can use the "Update Status" button present in figure?? to update the status from "IN TRANSIT" to "DELIVERED" if delivery runner successfully delivers the food to the customer. When one of the delivery runner fails to deliver the food, then he can choose to update the status to "FAILED".

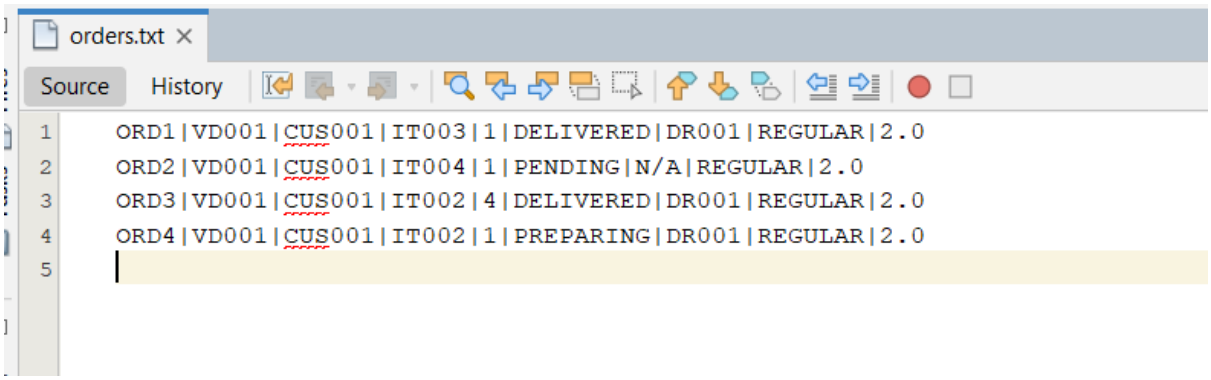


Figure 6.5: "order.txt" Contains All Orders with Its "Status" Changes.

The changes on status can be seen in the text file "orders.txt".

4.6.4 Task History

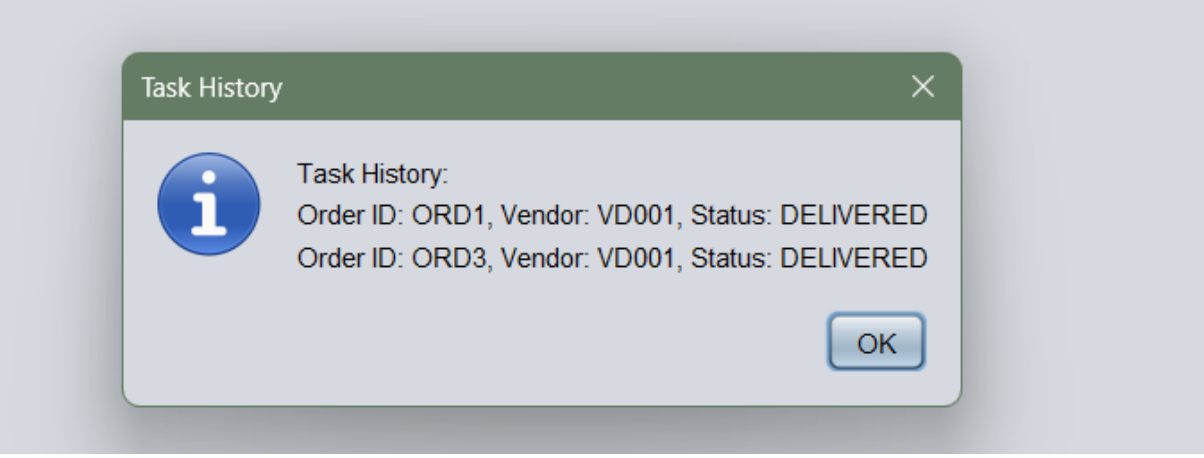


Figure 6.6: A Pop up Notification Shows All Delivery Runner Task History Using The Button "Task History" In Figure 6.0.

there is another button called "Task History". In there, delivery runner can see all of their previous tasks that was completed before.

4.6.5 View Reviews

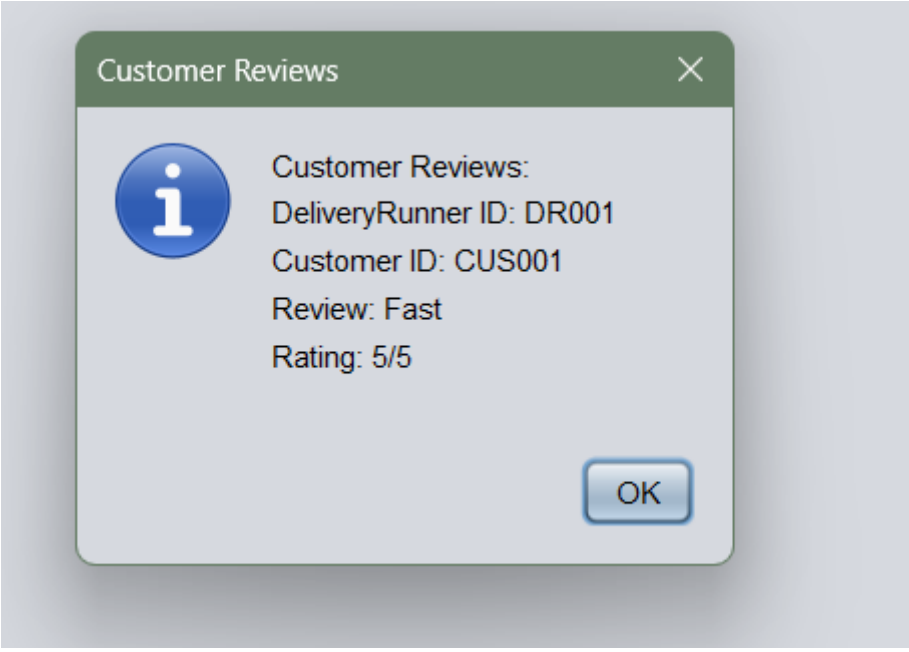


Figure 6.7: Shows All Customer Reviews Using "View Reviews" Button In Figure 6.0.

There is also "View Review" Button at figure 6.0 where delivery runner can only see all of his reviews but cannot see other delivery runner reviews.

4.6.6 Delivery's Revenue Dashboard

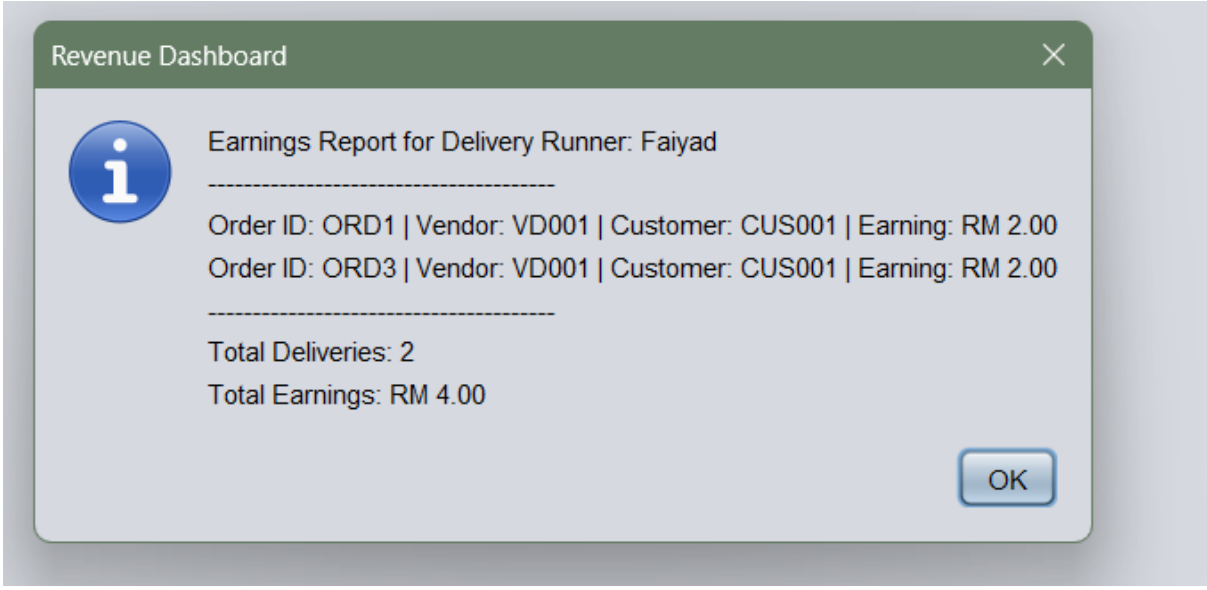


Figure 6.8: Shows Details of All Daily Earnings In The Revenue Dashboard

Similar to Vendor, delivery runner can generate a revenue report, by clicking "Revenue Dashboard" button and they can instantly view the report that consists of all delivered orders along with their earning for each delivery. It also includes the total revenue summary of all completed deliveries and Earnings.

4.7 Manager

4.7.0 Manager Dashboard

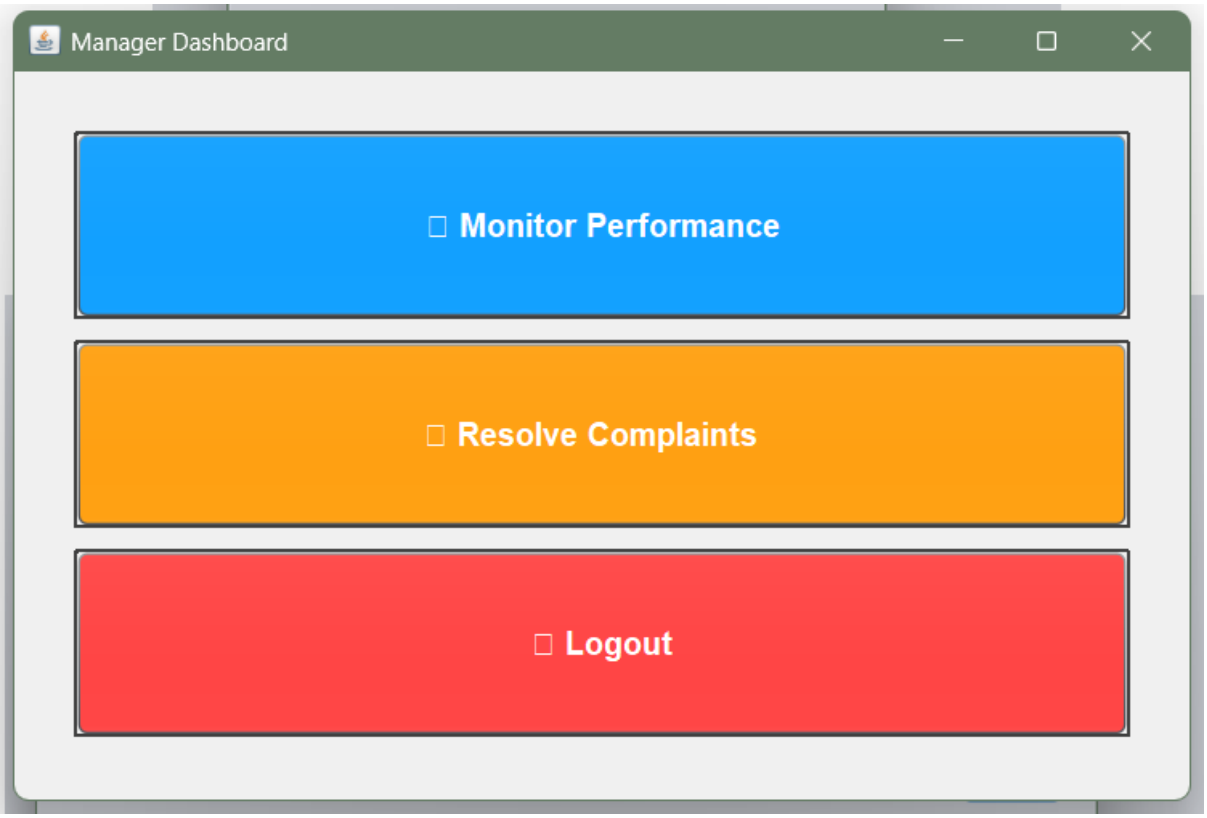


Figure 7.0: Manager Dashboard

The image above showcases the dashboard of Manager that consists of 3 main functions.

4.7.1 Monitor Performance System

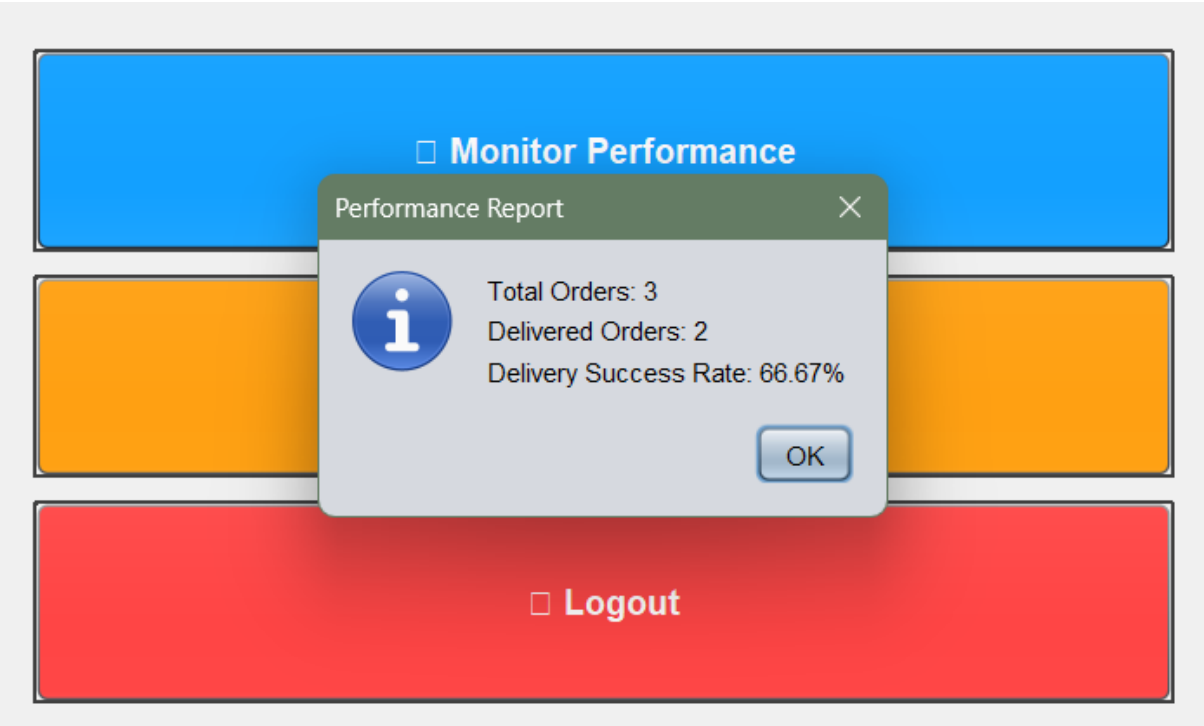


Figure 7.1: Manager can see Performance Report of vendor and delivery runner

The first function is **Monitor Performance.** This is one of manager assignment, where he can see how many orders being made, how many that has been successfully delivered and the percentage. This is very important because admin can track vendor and delivery runner progresses.

4.7.2 Resolve Complaints

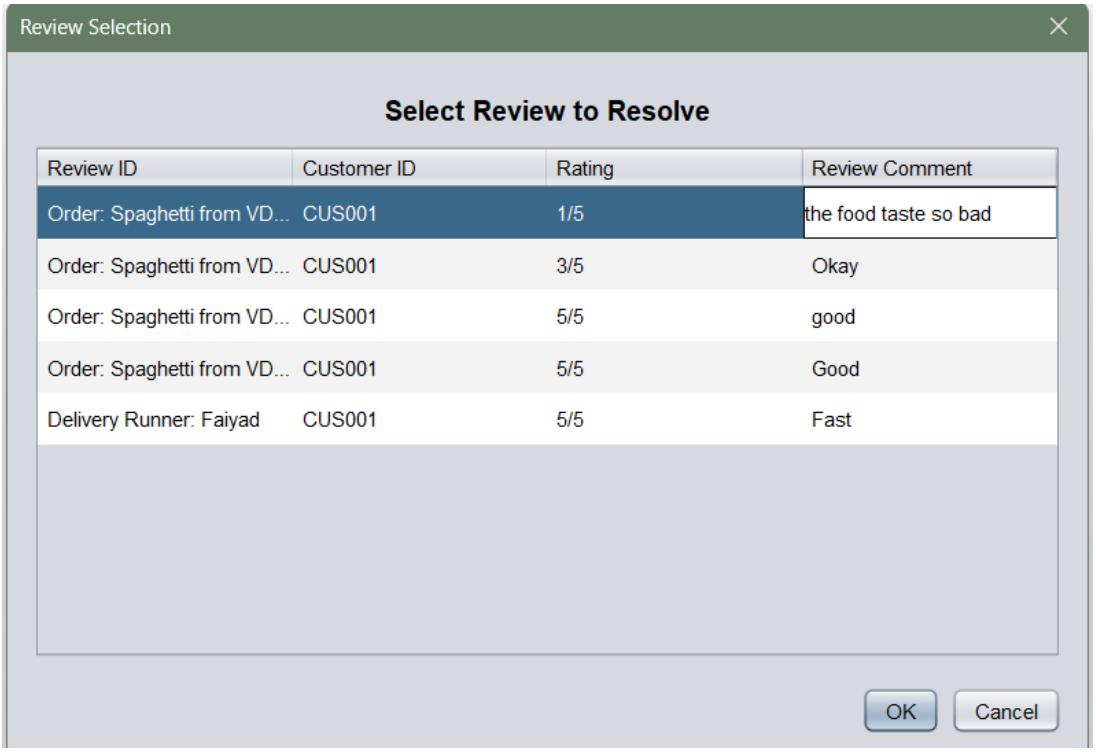


Figure 7.2: Lists of all Reviews

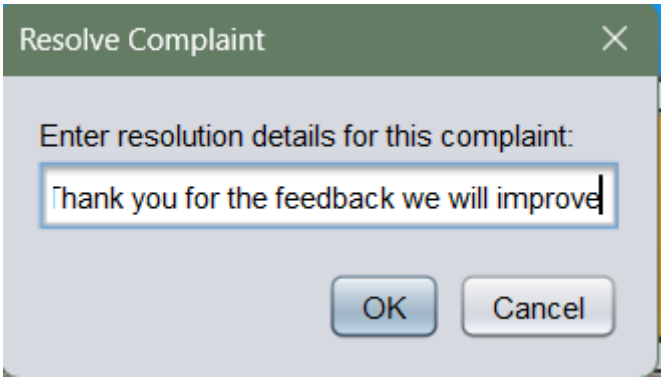


Figure 7.3 Manager resolve customer complaints

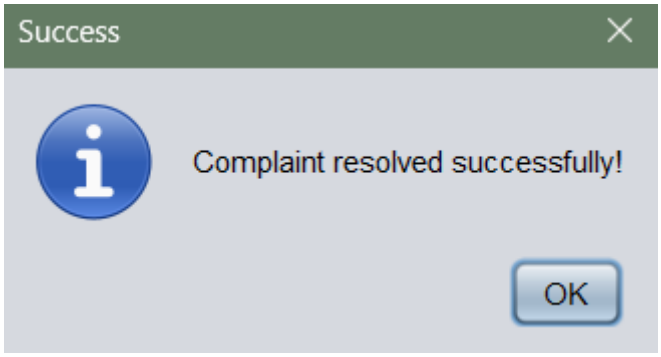


Figure 7.4 Notification saying complaint resolved successfully

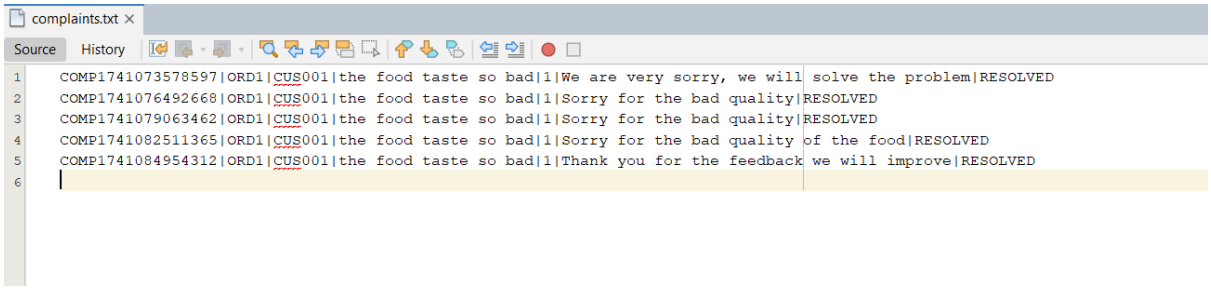


Figure 7.5 All the resolved save in (complains.txt) text file.

The second function is **Customer Complaints.** This function allows admin to respond to all customer complaints regarding their orders, starting issues with the dish to the assigned delivery runner. Admin can select the review with negative feedback, which redirects them to a pop-up box where they can resolve the issue as shown in image 2. Once resolved, the response will automatically save to the text file (complaints.txt)

5.0 Description and Justification of Object-Oriented Concepts

5. Encapsulation

```
public double getWalletBalance() {
    return this.walletBalance;
}

public void setWalletBalance(double balance) {
    if (balance >= 0) {
        this.walletBalance = balance;
        FileHandler.updateUser(this);
    } else {
        System.out.println("Error: Wallet balance cannot be negative.");
    }
}
```

Figure 8.0: Example of Encapsulation: Customer.java encapsulates wallet Balance

Encapsulation is a crucial principle of Object-Oriented Programming (OOP) that involves bundling data and methods that operate on that data into a single unit while restricting direct access to some of the object's components (Snyder, 1986). This ensures data security and prevents unintended modifications. In Java, encapsulation is effectively implemented in the Customer class, where the wallet Balance attribute is called as private and accessed through getter and setter methods.

5.2 Inheritance

Inheritance plays a crucial role in the design of the Lepak Food Court Management System, enabling code reusability, modularity, and efficient system expansion. By allowing classes to inherit attributes and behaviours from parent classes, the system ensures a structured and scalable architecture (Tempero et al., 2008). In our code, inheritance is implemented in manager class, where manager.java is inherit from user, meaning it automatically get userID, password, name. The reason why this is inheritance is because instead of rewriting back the manager user information, it reuses these attributes from User and just adds on the specific functionalities like managing customer complaints.

```java
public class Manager extends User {

    private List<Order> allOrders;
    private List<String> complaints;  // ✅ Added complaints tracking

    public Manager(String userID, String password, String name, String mobileNumber, String address) {
        super(userID, password, "MANAGER", name, mobileNumber, address);
        this.allOrders = new ArrayList<>();
        this.complaints = new ArrayList<>();
    }
```

Figure 8.1: Inheritance in Java: Manager Class Reusing Attributes and Methods from User.

5.3 Abstraction

In our code Abstraction is implemented in VendorController.java, where it abstracts the logic for accepting orders, so the Vendor Dashboard does not need to know how orders are stored. This is abstraction because in Vendor Dashboard it just calls acceptOrder(orderId), without knowing the file storage process. This method is used to simplify the interface for the Vendor.

```java
    public void acceptOrder(String orderId) {
        List<Order> orders = FileHandler.loadOrders();
        List<DR> availableRunners = FileHandler.loadDeliveryRunners();
        boolean orderFound = false;

        for (Order order : orders) {
            if (order.getOrderID().equals(orderId) && order.getVendorID().equals(currentVendor.getUserID())) {
                // Check if it's a DINE_IN order (either by delivery type or current status)
                if (order.getDeliveryType().equals("DINE_IN") || order.getStatus().equals("DINE IN")) {
                    // For DINE_IN orders:
                    order.setStatus("PREPARING");
                    order.setAssignedDR("DINE_IN");
                    System.out.println("DINE IN order " + orderId + " accepted. Status: PREPARING, AssignedDR: DINE_IN");
                } else {
                    // For non-DINE_IN orders:
                    order.setStatus("PREPARING");

                    // Auto-assign a delivery runner if available
                    if (!availableRunners.isEmpty()) {
                        // Simple assignment - just pick the first available runner
                        String runnerId = availableRunners.get(0).getUserID();
                        order.setAssignedDR(runnerId);
                        System.out.println("Order " + orderId + " assigned to runner " + runnerId);
                    }
                }

                orderFound = true;
                break;
            }
        }

        if (orderFound) {
            // Save all orders back to file
            FileHandler.saveOrders(orders);
            System.out.println("Order " + orderId + " status updated and saved");
        } else {
            System.out.println("Order " + orderId + " not found or not owned by this vendor");
        }
    }
```


Figure 8.2: Abstracting Order Handling: acceptOrder () Simplifying Vendor Order Processing

5.4 Polymorphism

Polymorphism is implemented in DRController.java for our code, the updateStatus method dynamically allows different order status to be handled based on the order type. The reason why this is polymorphism is because the method (updateStatus ()) act differently depending on the status provided (IN Transit, Delivered, Failed) and single method handles multiple types of order state transitions without needing separate methods for each status type.

```
// Update the Status of an Order
public boolean updateStatus(String orderId, String newStatus) {
    if (!newStatus.equals("IN TRANSIT") && !newStatus.equals("DELIVERED") && !newStatus.equals("FAILED")) {
        JOptionPane.showMessageDialog(null, "Invalid status! Allowed: IN TRANSIT, DELIVERED, FAILED", "Error", JOptionPane.ERROR_MESSAGE);
        return false;
    }

    List<Order> orders = FileHandler.loadOrders();

    for (int i = 0; i < orders.size(); i++) {
        Order order = orders.get(i);
        if (order.getOrderID().equals(orderId) && order.getAssignedDR().equals(deliveryRunner.getUserID())) {
            order.setStatus(newStatus);
            FileHandler.saveOrders(orders);
            JOptionPane.showMessageDialog(null, "Order status updated successfully!", "Success", JOptionPane.INFORMATION_MESSAGE);
            return true;
        }
    }

    JOptionPane.showMessageDialog(null, "Order not found or not assigned to you!", "Error", JOptionPane.ERROR_MESSAGE);
    return false;
}
```

Figure 8.3: Polymorphism in Java: Dynamic Order Status Updates in updateStatus() Method

5.5 Association

Association in Object-Oriented Programming (OOP) refers to the link between to classes whereby one class is using or associated with another class. It does not involve inheritance but a link between objects. In our java code, association is implemented in the Order.java, where each order is associated with one Vendor (through vendorID), each order is linked to one customer (through customerID) and each order can be assigned to one Delivery Runner (DR) (through assignedDR). The association type is many to one association. The reason this represent association, as Order acts as a bridge between different entities without enforcing strict dependency.

```java
package model;

public class Order {

    private String orderID;
    private String vendorID;
    private String customerID;
    private String assignedDR;
    private int quantity;
    private String status;
    private MenuItem menuItem;
    private String deliveryType = "DINE_IN"; // Default to dine-in. NEW CLASS
    private double deliveryFee = 0.0;
```

Figure 8.4: Association in Java: Order class linking vendor, customer and menuitem

5.6 Composition

Composition is when two classes are totally dependent on one another. The child should be erased along with the parent. In our code, AdminDahsboard.java contains an instance of of AdminController, allowing it to manage admin actions. The reason why this is composition is because AdminDashboard "HAS A" AdminController, meaning it depends on AdminController for functionality. The dashboard cannot function without the controller.

```java
public class AdminDashboard extends JFrame {


    private AdminController adminController;

    // Constructor
    public AdminDashboard(Admin admin, AdminController adminController1) {
        this.adminController = new AdminController(admin);
```

Figure 8.5: Composition in Java: AdminDashboard strongly dependent on AdminController

5.7 Exception

Exception handling is to manages errors and unexpected situations when the program is running without crashing the system. It allows a program to detect, handle, and recover from runtime errors. In our code one example of error handling is in CustomerController, when a customer places an order, it involves checking availability, deducting balance and saving the order. If insufficient balances, and exception message will appear

```
// Place an Order (Checks Wallet Balance First)
public boolean placeOrder(String itemId, int quantity) {
    if (quantity <= 0) {
        JOptionPane.showMessageDialog(null, "Invalid quantity! Must be greater than zero.", "Error", JOptionPane.ERROR_MESSAGE);
        return false;
    }

    List<MenuItem> menuItems = FileHandler.loadMenuItems();
    MenuItem selectedItem = menuItems.stream()
            .filter(item -> item.getItemId().equals(itemId))
            .findFirst()
            .orElse(null);

    if (selectedItem == null) {
        JOptionPane.showMessageDialog(null, "Invalid menu item ID!", "Error", JOptionPane.ERROR_MESSAGE);
        return false;
    }

    if (selectedItem.getVendor() == null) { // Ensure Vendor Exists
        JOptionPane.showMessageDialog(null, "The selected item has no assigned vendor!", "Error", JOptionPane.ERROR_MESSAGE);
        return false;
    }

    double totalPrice = selectedItem.getPrice() * quantity;
    if (customer.getWalletBalance() < totalPrice) {
        JOptionPane.showMessageDialog(null, "Insufficient wallet balance!", "Error", JOptionPane.ERROR_MESSAGE);
        return false;
    }
```

Figure 8.6: Exception Handling in Order Placement: Preventing Invalid Inputs and Insufficient Balance

5.8 File Handling

In our code, FileHandler.java manages data storage, including saving complaints, preventing direct file access in ManagerController. Encapsulation is applied as ManagerController calls saveComplaints() instead of handling file operations directly. Abstraction ensures ManagerController interacts with FileHandler without needing to know the internal storage details.

```
// ✅ Save complaints to file
    public static void saveComplaints(List<String> complaints) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("data/complaints.txt"))) {
            for (String complaint : complaints) {
                writer.write(complaint);
                writer.newLine();
            }
        } catch (IOException e) {
            System.out.println("Error saving complaints: " + e.getMessage());
        }
    }
}
```

Figure 8.7: Encapsulation and Abstraction in File Handling: saveComplaints()

6.0 Limitation

- Database not secure, once data is deleted cannot retrieve back
- When Vendor reject order, customer cannot get refunded
- Different devices have different interfaces such as icon for manager cannot be seen
- Sales deposit is for one day, not quarterly, monthly sales report
- Admin can only update number and address
- For customer to place order, must manually put the ItemID and the amount and for customer to submit review, must key in OrderID manually, not very user friendly
- Customer cannot view menu side by side while placing orders.

7.0 References

A, R., A, M., & B, R. G. (2004, August 9). *Fragment class analysis for testing of polymorphism in Java software*. Retrieved from https://ieeexplore.ieee.org/document/1321060

Cohen, T. (1984, January 1). *Data abstraction, data encapsulation and object-oriented programming*. Retrieved from ACM Digital Library: https://dl.acm.org/doi/10.1145/948415.948418

Monus, A. (2023, February 3). *Using OOP concepts to write high-performance Java code (2023)*. Retrieved from RayGun: https://raygun.com/blog/oop-concepts-java/

Muhammad, S. (2024, 30 June). *Understanding Object-Oriented Programming (OOP) Concepts with Java Examples*. Retrieved from Medium: https://medium.com/@suraif16/object-oriented-programming-9e4627abf1be

Stackify Team. (2024, September 21). *OOP Concepts in Java: Defined and Explained with Examples*. Retrieved from Stackify: https://stackify.com/oops-concepts-in-java/

Tempero, E., Noble, J., & Melton, H. (n.d.). *https://ieeexplore.ieee.org/document/6632635*. Retrieved from Springer Nature Link: https://link.springer.com/chapter/10.1007/978-3-540-70592-5_28#citeas

Zhang, J., Caldwell, E. R., & Smith, E. (2013, October 17). *Learning the concept of Java inheritance in a game*. Retrieved from IEEE Xplore: https://ieeexplore.ieee.org/document/6632635