ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title			
Submission date		**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Bui Quang Minh	**Student ID**	GCD210325
Class	GCD1104	**Assessor name**	Nguyen The Nghia

Student declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

Student's signature	

Grading grid

Grade (0-10)

☼ Summative Feedback:		☼ Resubmission Feedback:
Grade:	Assessor Signature:	Date:
IV Signature:		

Contents

I. INTRODUCTION

I have received the requirement to develop a desktop application that handles small business issues that is managing a restaurant. I need to create a graphical user interface with features such as reading and writing data from text files, and working with the collection of data. Moreover, I also need to handle all errors to avoid crashing on the end-user side. Finally, the application has to be completely tested before packing it.

Figure 1. Programmer illustration

II. REQUIREMENT

Develop an application to address a specific business issue.

Create a graphical user interface (GUI) to interact with the application.

Implement functionality to read and write data from/to text files.

Enable operations on a collection of data

Handle errors effectively to prevent application crashes on the user's end.

Conduct testing of the application prior to the production phase.

Write a detailed technical report covering:

- Design considerations and decisions.
- Implementation details, including algorithms and coding practices.
- Testing strategies and results.

Prepare for a live application demonstration, including:

- Showing the application's user interface and functionalities.
- Explaining the code architecture and key components.
- Addressing technical questions from the lecturer.

III. UI DESIGN

I have used figma to design my swing app and it contains 4 main interfaces including home that is used to show tables and menu that is used to show dishes in the restaurant, login form to access and invoice to view the orders.

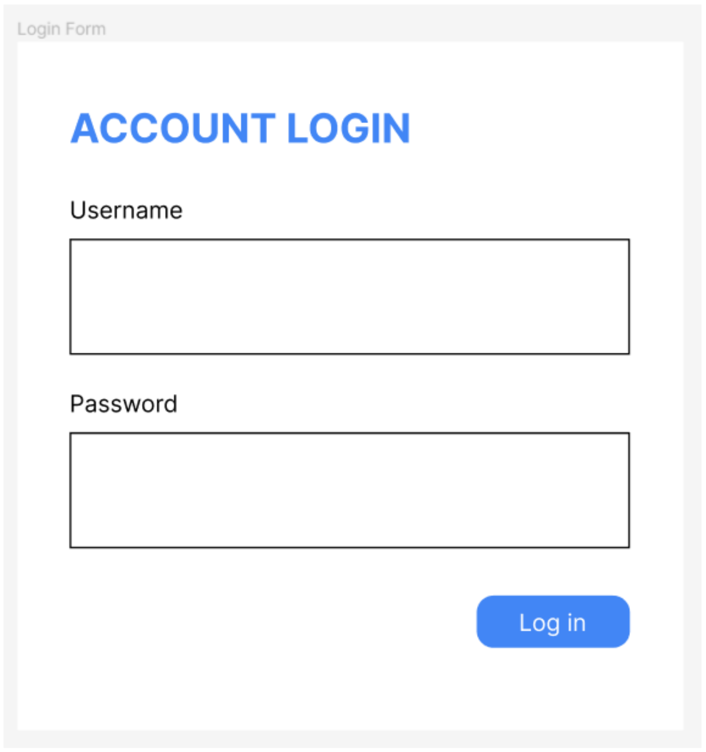


Figure 2. Interface of Login form

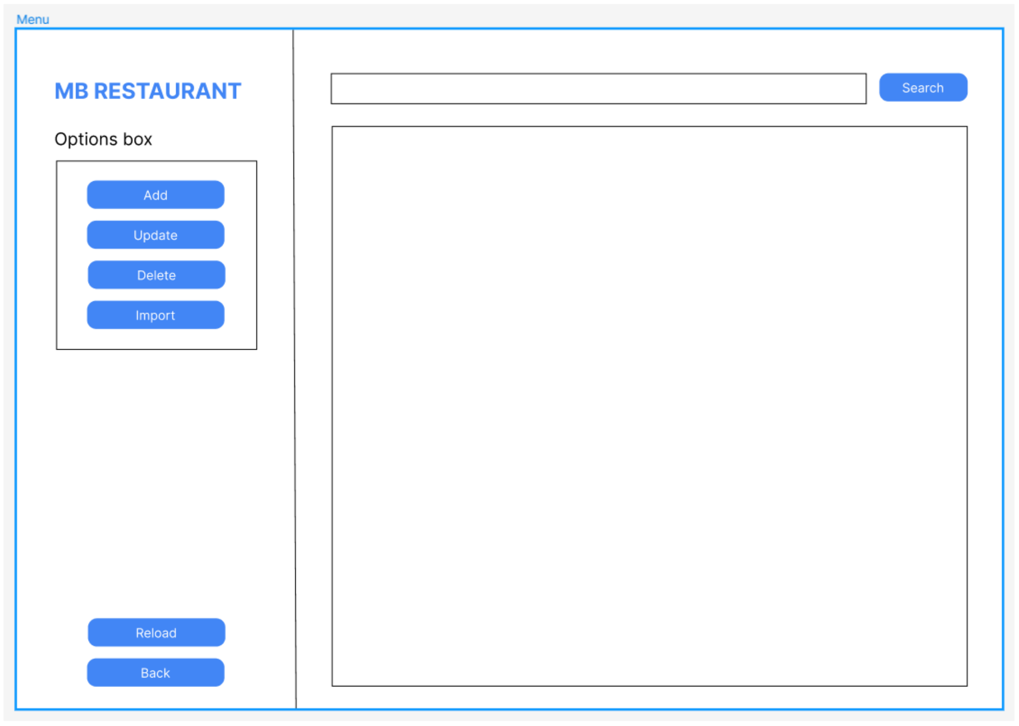


Figure 3. Interface of menu

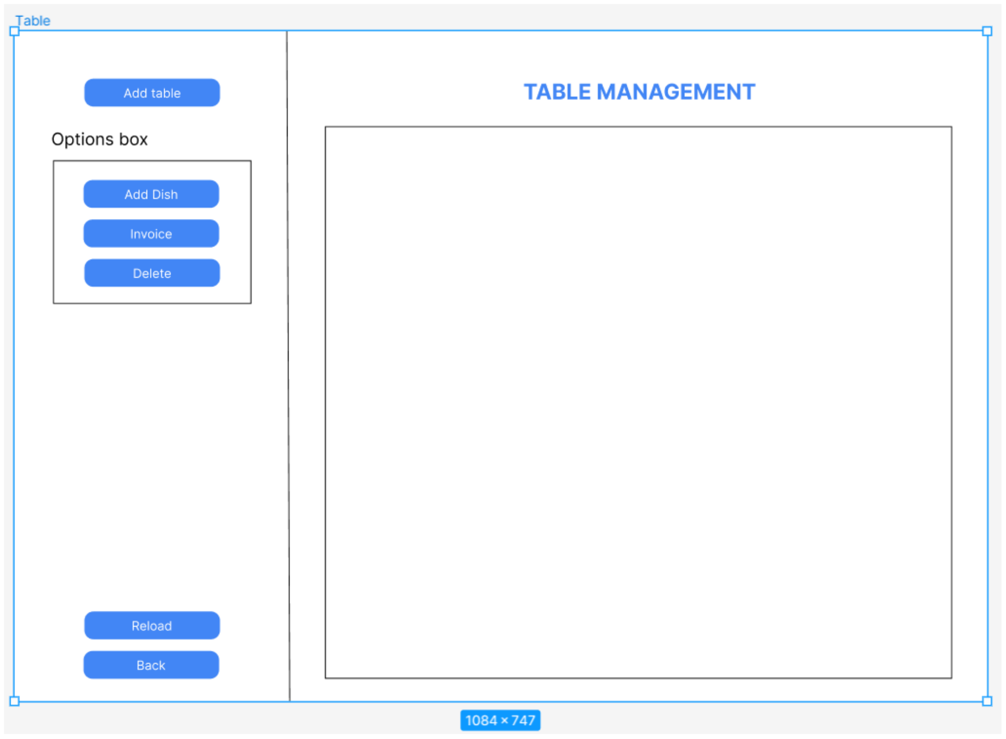


Figure 4. Interface of table management

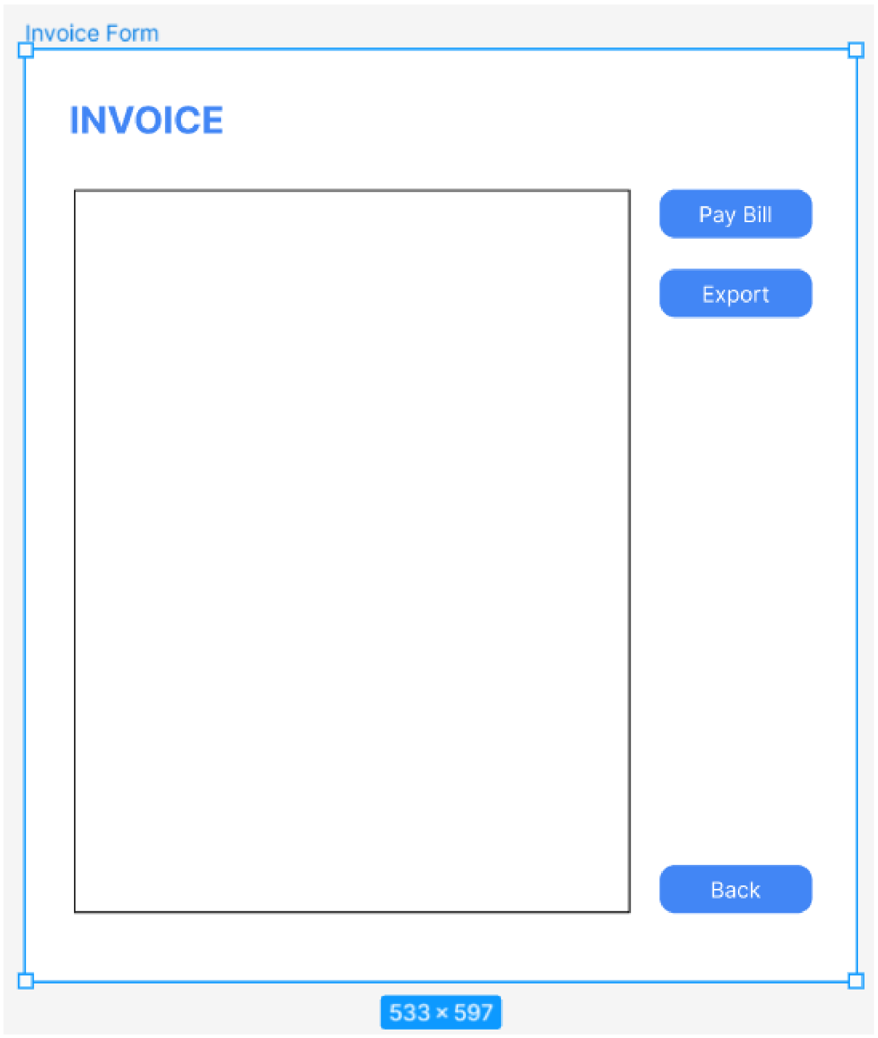


Figure 5. Interface of invoice

IV. IMPLEMENTATION

1. Overview

Once user fire the program, he has to sign in to access the program and then he will see the navigator to direct to the main interfaces including menu and table.

At the menu choice, he can modify the menu of the restaurant and manage the list of dishes such as adding, updating, deleting a dish. He also can import data if the program did meet some errors to backup the data. Searching bar helps him to find the particular dish that he needs. Interfaces of managing tables is the same and it is used to manage the tables in the restaurant.

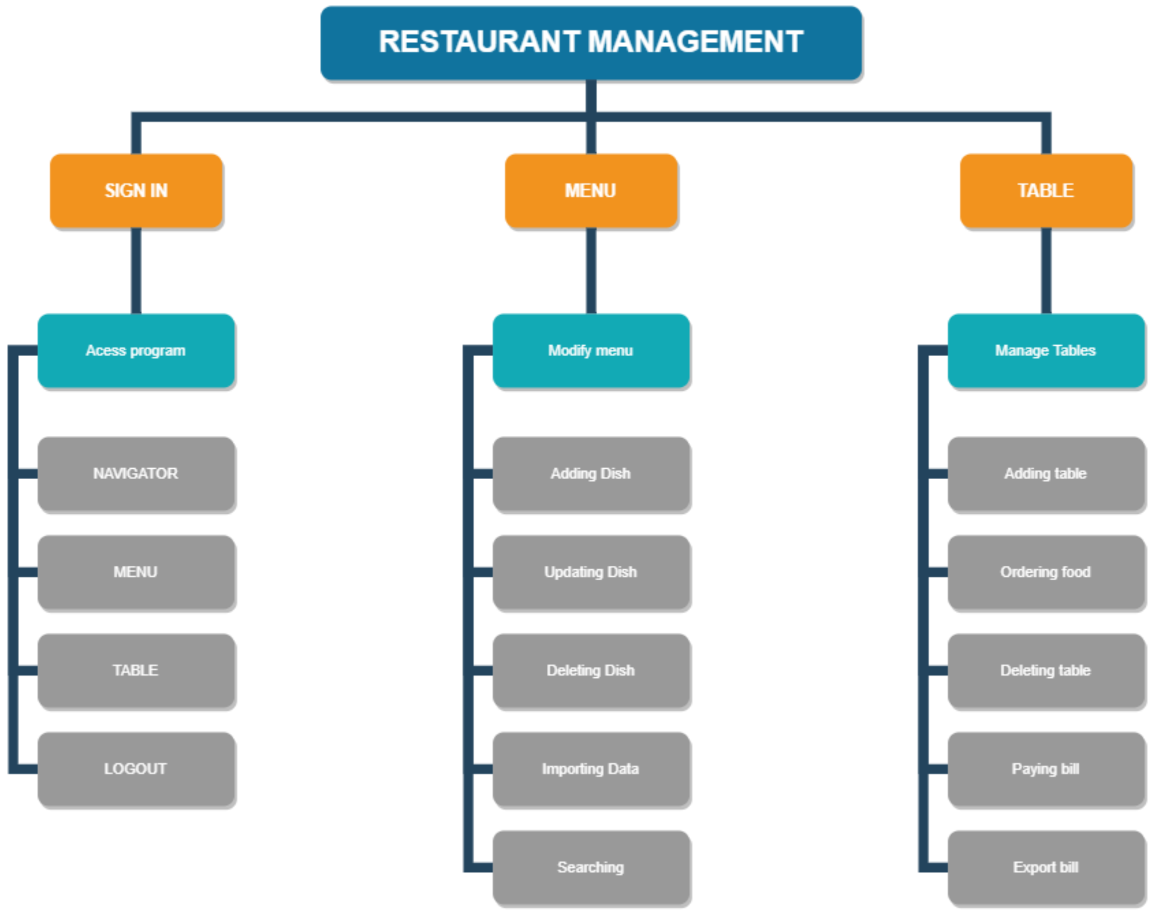


Figure 6. Program structures

The program will have 5 mains entities, they are:

- Admin: This is used to authenticate in order to access the program
- Category: Each dish will have its category
- Dish: Dish includes food and drink that serves the customers
- Order: The order is created to manage the table
- Detail: The detailed information about the table

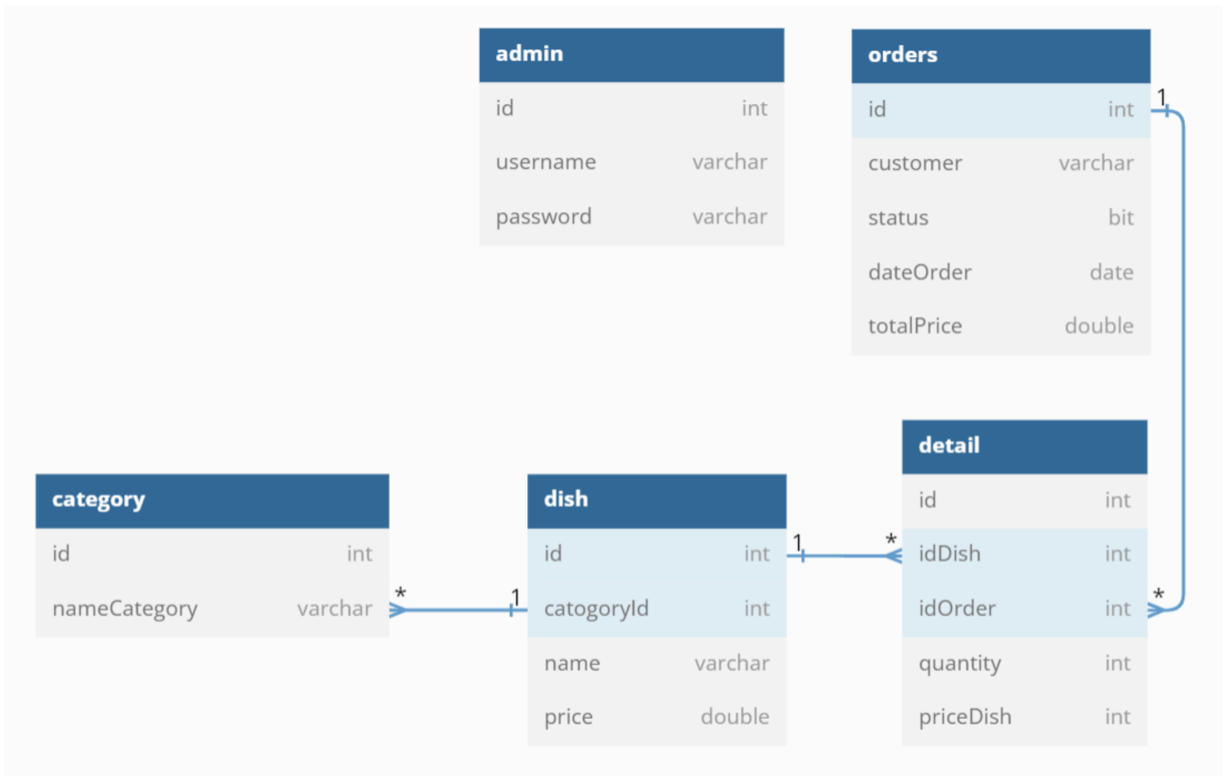


Figure 7. Entity relationship diagram

In my program, there are many packages and each of them has different purposes and functions including:

- Asset: To save photos or other files that are not related to the logic code
- Core: To save classes that relate to the system
- Helper: To save methods to reuse many times
- Entity: To save object classes
- CustomEntity: To save object classes that contains many attributes from many tables

- Controller: To receive the request from client and send it to Model to handle and then send Data to View
- Model: To contact with Database and Data
- View: The interfaces that user can see

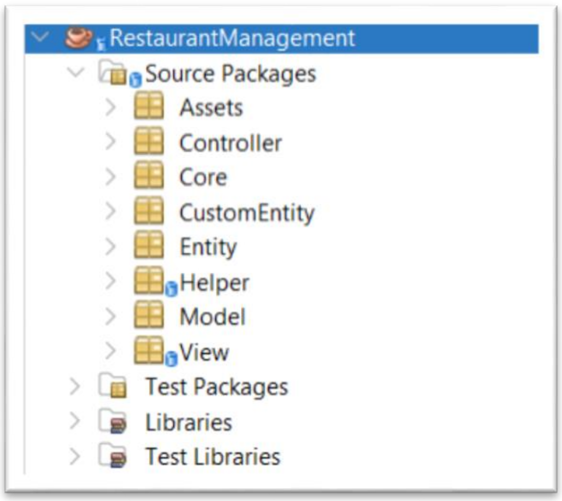


Figure 8. Main packages of the program

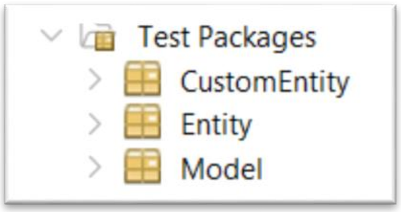


Figure 9. Test packages of the program

2. Code explaination

In this part, I just show the interfaces and the code. All the cases happens when doing something I will put them in Test parts

2.1 Sign In

When the program runs, the login form will pop up to allow user in order to access the program. In this form, user has to enter the correct information, if not he cannot access the program. Below is the View of login from:

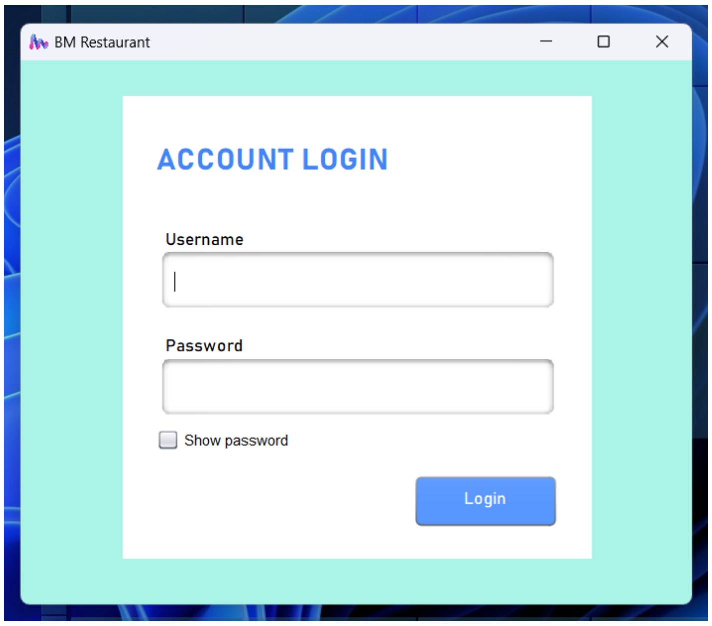


Figure 10. Interface of login form

Because password input is a jPassword filed so user cannot see the password but user can click show password checkbox to view the password if he is not sure about his password

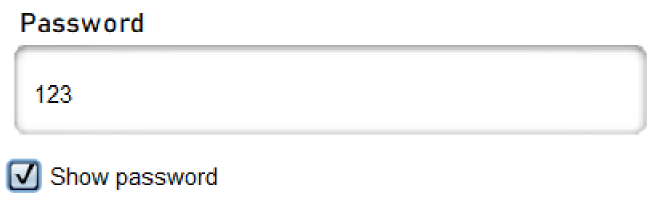
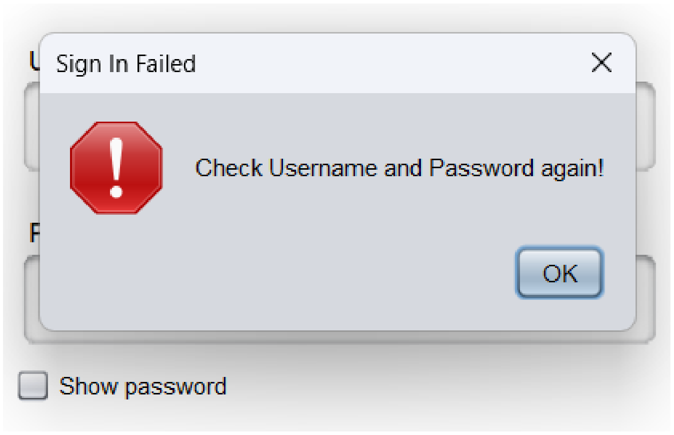


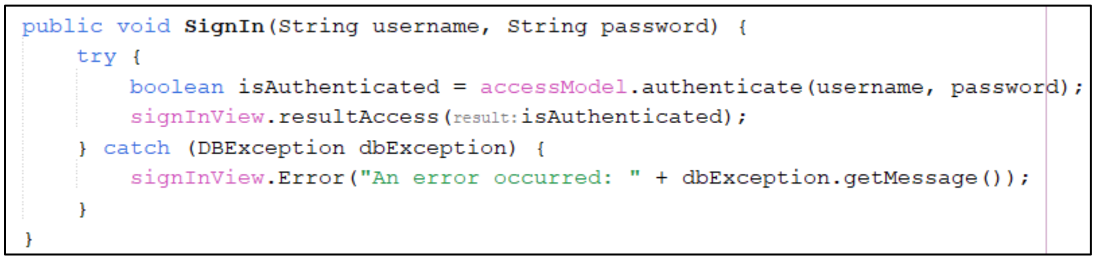
Figure 11. Show password function

If user enters the incorrect information, the program will pop up a below message. All the exception will be caught at controller and show message at view



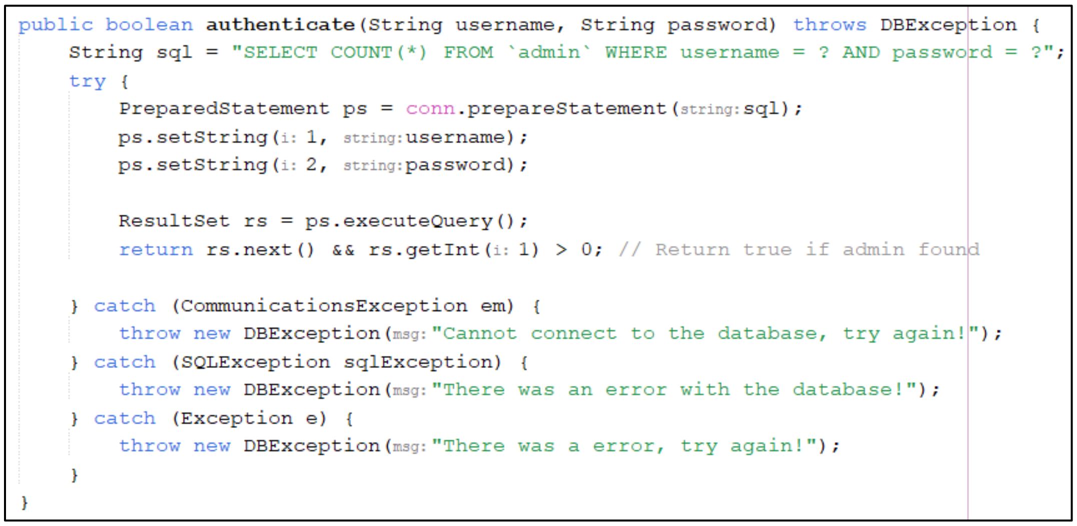
Check Username and Password again!

Figure 12. Error message when login fail



```java
public void SignIn(String username, String password) {
    try {
        boolean isAuthenticated = accessModel.authenticate(username, password);
        signInView.resultAccess(result:isAuthenticated);
    } catch (DBException dbException) {
        signInView.Error("An error occurred: " + dbException.getMessage());
    }
}
```

Figure 13. Action SignIn in controller

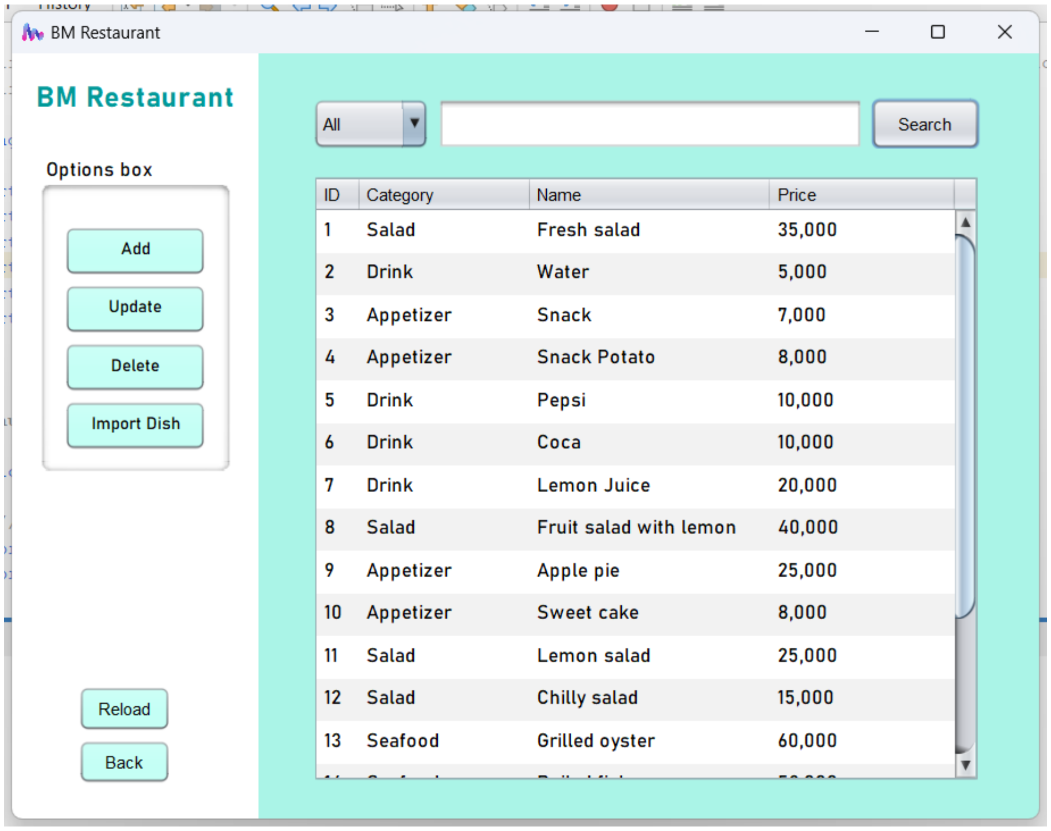


```java
public boolean authenticate(String username, String password) throws DBException {
    String sql = "SELECT COUNT(*) FROM `admin` WHERE username = ? AND password = ?";
    try {
        PreparedStatement ps = conn.prepareStatement(string:sql);
        ps.setString(i: 1, string:username);
        ps.setString(i: 2, string:password);

        ResultSet rs = ps.executeQuery();
        return rs.next() && rs.getInt(i: 1) > 0; // Return true if admin found

    } catch (CommunicationsException em) {
        throw new DBException(msg:"Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg:"There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg:"There was a error, try again!");
    }
}
```

Figure 14. Model of signin

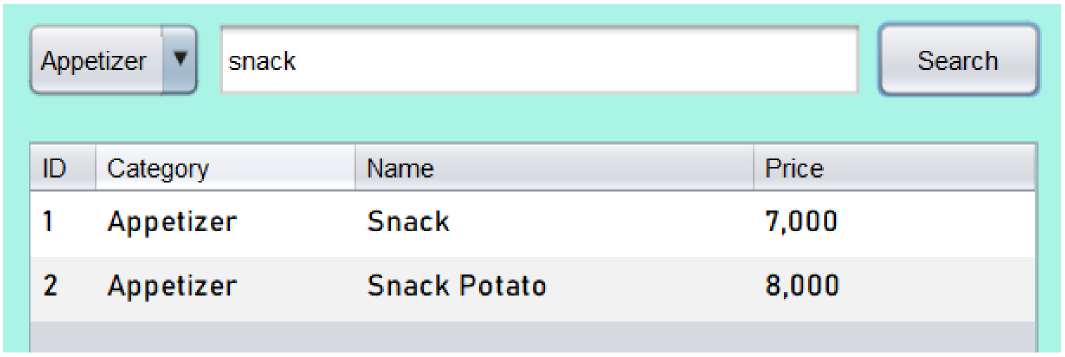
2.2 Menu

Below is the interface of Menu where user can add, update, delete a dish. Moreover, user can import a list of dishes if he has a data file. Secondly, if there are so many dishes in the table, user can search regarding category or dish's name.



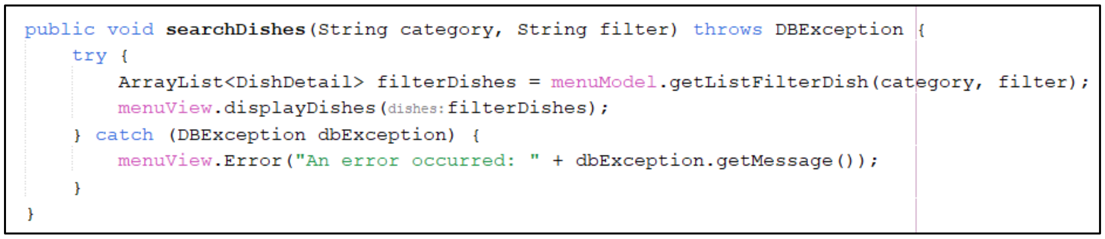
ID	Category	Name	Price
1	Salad	Fresh salad	35,000
2	Drink	Water	5,000
3	Appetizer	Snack	7,000
4	Appetizer	Snack Potato	8,000
5	Drink	Pepsi	10,000
6	Drink	Coca	10,000
7	Drink	Lemon Juice	20,000
8	Salad	Fruit salad with lemon	40,000
9	Appetizer	Apple pie	25,000
10	Appetizer	Sweet cake	8,000
11	Salad	Lemon salad	25,000
12	Salad	Chilly salad	15,000
13	Seafood	Grilled oyster	60,000

Figure 15. Interface of menu



ID	Category	Name	Price
1	Appetizer	Snack	7,000
2	Appetizer	Snack Potato	8,000

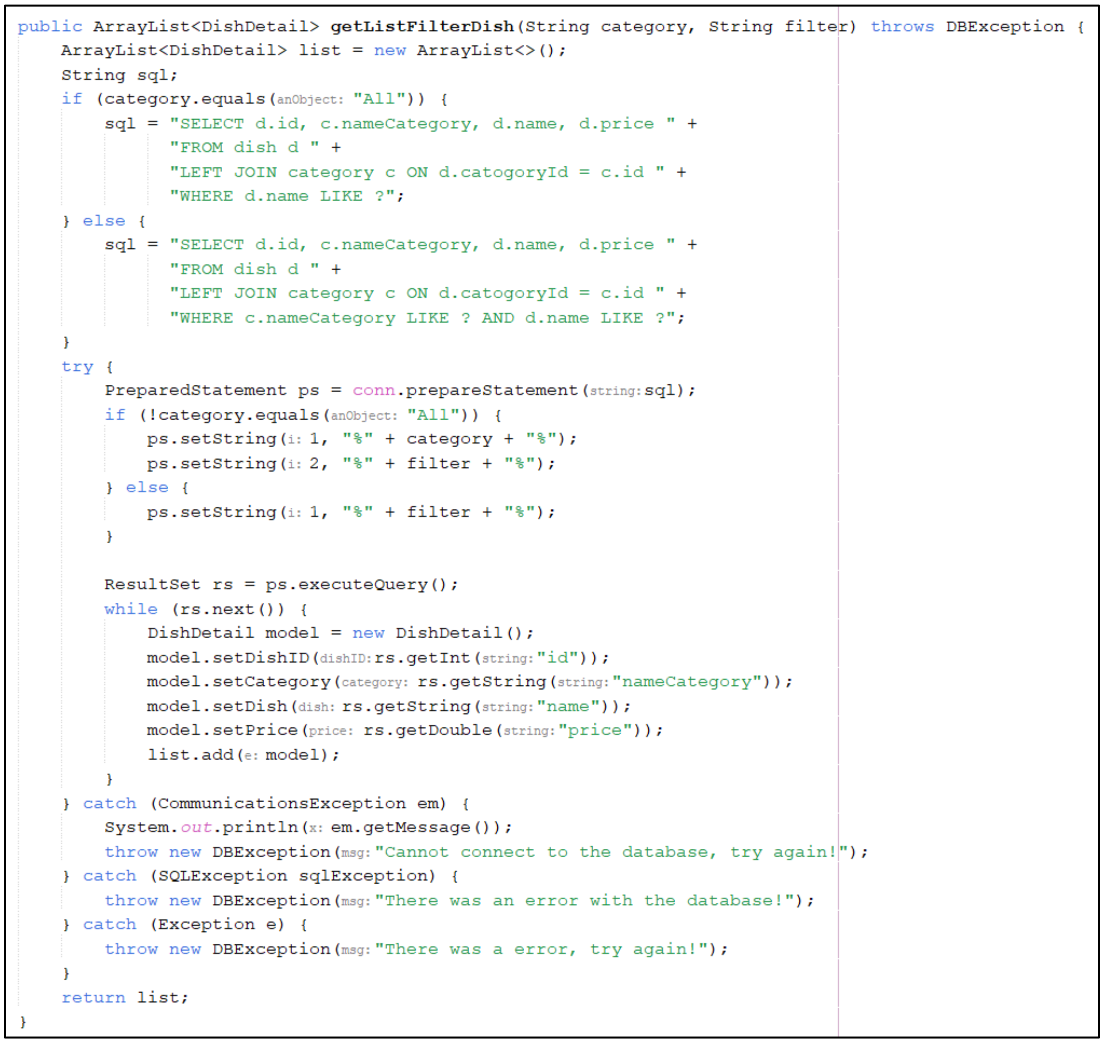
Figure 16. Filter result



```
public void searchDishes(String category, String filter) throws DBException {
    try {
        ArrayList<DishDetail> filterDishes = menuModel.getListFilterDish(category, filter);
        menuView.displayDishes(dishes:filterDishes);
    } catch (DBException dbException) {
        menuView.Error("An error occurred: " + dbException.getMessage());
    }
}
```

Figure 17. Action searching in controller

I divide this method into 2 cases, one is with category and another one is not with category. All the exception will be caught at controller and show message at view



```
public ArrayList<DishDetail> getListFilterDish(String category, String filter) throws DBException {
    ArrayList<DishDetail> list = new ArrayList<>();
    String sql;
    if (category.equals(anObject: "All")) {
        sql = "SELECT d.id, c.nameCategory, d.name, d.price " +
              "FROM dish d " +
              "LEFT JOIN category c ON d.catogoryId = c.id " +
              "WHERE d.name LIKE ?";
    } else {
        sql = "SELECT d.id, c.nameCategory, d.name, d.price " +
              "FROM dish d " +
              "LEFT JOIN category c ON d.catogoryId = c.id " +
              "WHERE c.nameCategory LIKE ? AND d.name LIKE ?";
    }
    try {
        PreparedStatement ps = conn.prepareStatement(string:sql);
        if (!category.equals(anObject: "All")) {
            ps.setString(i: 1, "%" + category + "%");
            ps.setString(i: 2, "%" + filter + "%");
        } else {
            ps.setString(i: 1, "%" + filter + "%");
        }

        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            DishDetail model = new DishDetail();
            model.setDishID(dishID:rs.getInt(string:"id"));
            model.setCategory(category: rs.getString(string:"nameCategory"));
            model.setDish(dish: rs.getString(string:"name"));
            model.setPrice(price: rs.getDouble(string:"price"));
            list.add(e: model);
        }
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg:"Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg:"There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg:"There was a error, try again!");
    }
    return list;
}
```

Figure 18. Searching model

2.3 Add or Update Dish in Menu

I used the same form for adding and updating dish. The adding form of dish will add a new dish to the menu of restaurant. If user does not enter anything, a message will pop up inform and in the price input, user cannot enter any special characters except for numbers and comma.

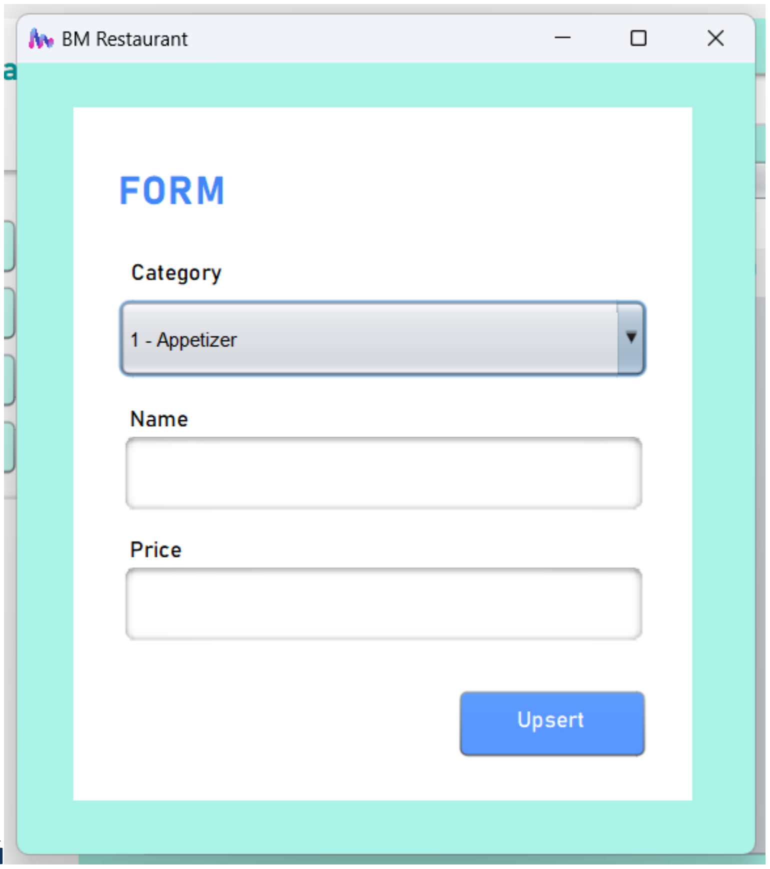
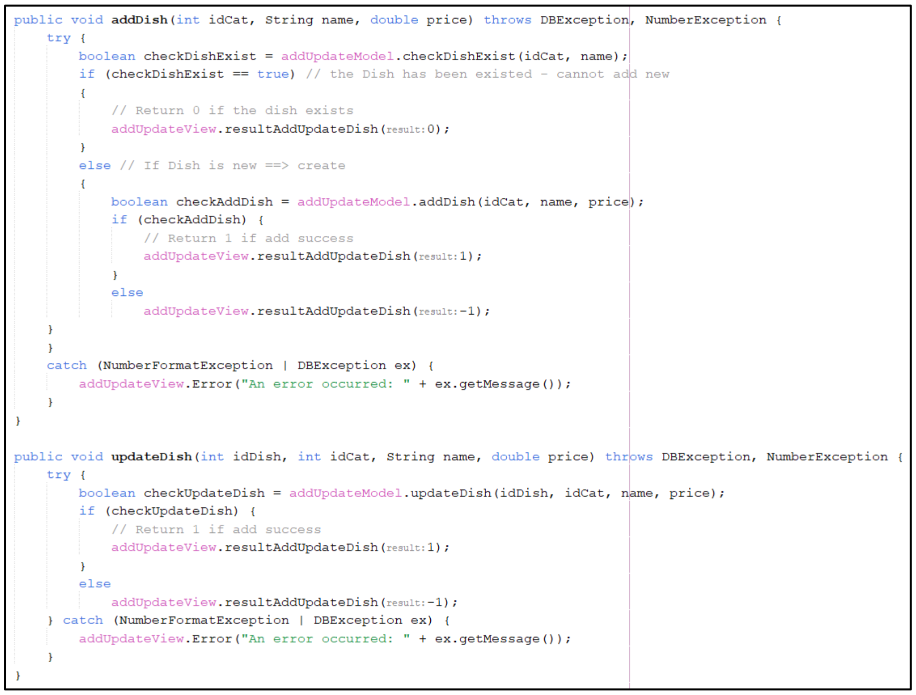


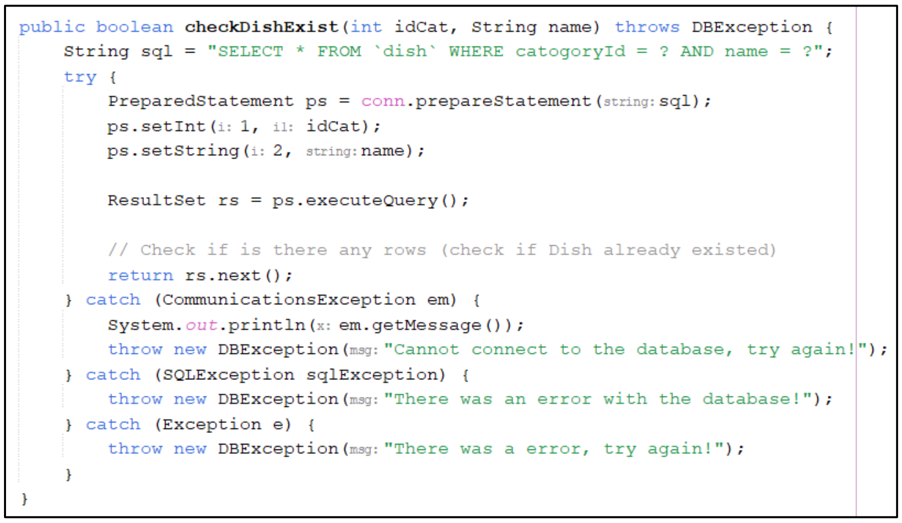
Figure 19. Add and update interface

The categories need to load first to set dish's name belongs to that category and then there are many cases in adding a new dish, first is checking if it exists or not and then add it to database. All the exception will be caught at controller and show message at view



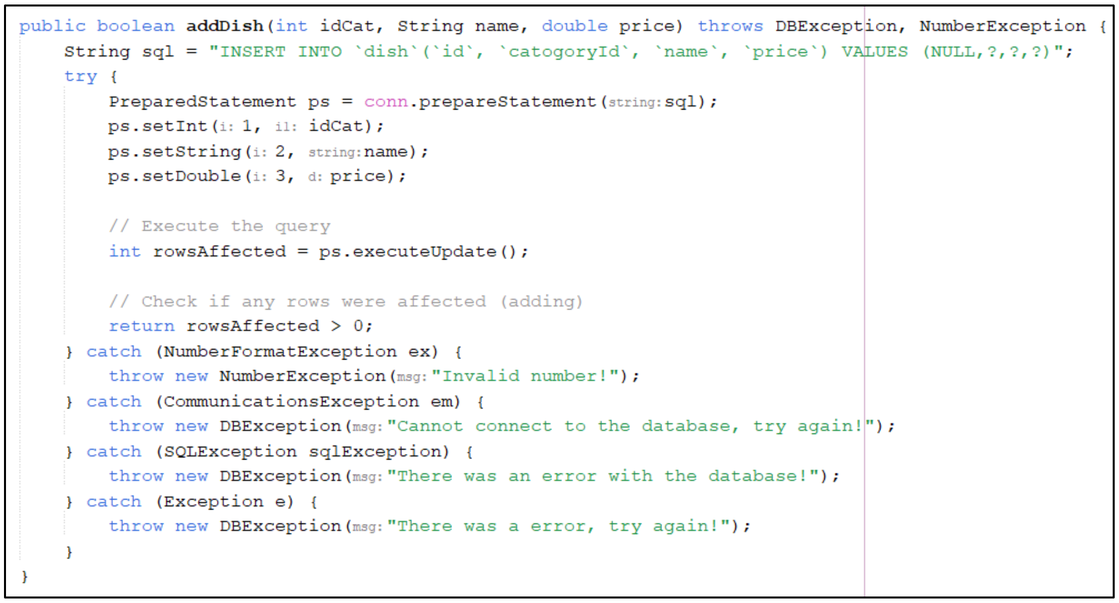
```java
public void addDish(int idCat, String name, double price) throws DBException, NumberException {
    try {
        boolean checkDishExist = addUpdateModel.checkDishExist(idCat, name);
        if (checkDishExist == true) // the Dish has been existed - cannot add new
        {
            // Return 0 if the dish exists
            addUpdateView.resultAddUpdateDish(result:0);
        }
        else // If Dish is new ==> create
        {
            boolean checkAddDish = addUpdateModel.addDish(idCat, name, price);
            if (checkAddDish) {
                // Return 1 if add success
                addUpdateView.resultAddUpdateDish(result:1);
            }
            else
                addUpdateView.resultAddUpdateDish(result:-1);
        }
    }
    catch (NumberFormatException | DBException ex) {
        addUpdateView.Error("An error occurred: " + ex.getMessage());
    }
}

public void updateDish(int idDish, int idCat, String name, double price) throws DBException, NumberException {
    try {
        boolean checkUpdateDish = addUpdateModel.updateDish(idDish, idCat, name, price);
        if (checkUpdateDish) {
            // Return 1 if add success
            addUpdateView.resultAddUpdateDish(result:1);
        }
        else
            addUpdateView.resultAddUpdateDish(result:-1);
    } catch (NumberFormatException | DBException ex) {
        addUpdateView.Error("An error occurred: " + ex.getMessage());
    }
}
```

Figure 20. Add and update dish actions



```java
public boolean checkDishExist(int idCat, String name) throws DBException {
    String sql = "SELECT * FROM `dish` WHERE catogoryId = ? AND name = ?";
    try {
        PreparedStatement ps = conn.prepareStatement(string: sql);
        ps.setInt(i: 1, i1: idCat);
        ps.setString(i: 2, string: name);

        ResultSet rs = ps.executeQuery();

        // Check if is there any rows (check if Dish already existed)
        return rs.next();
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg: "Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg: "There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg: "There was a error, try again!");
    }
}
```

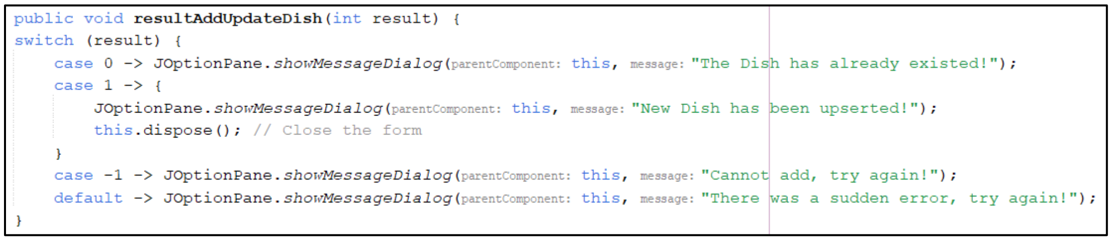
Figure 21. Checking existed dish method



```java
public boolean addDish(int idCat, String name, double price) throws DBException, NumberException {
    String sql = "INSERT INTO `dish`(`id`, `catogoryId`, `name`, `price`) VALUES (NULL,?,?,?)";
    try {
        PreparedStatement ps = conn.prepareStatement(string:sql);
        ps.setInt(i: 1, i1: idCat);
        ps.setString(i: 2, string:name);
        ps.setDouble(i: 3, d: price);

        // Execute the query
        int rowsAffected = ps.executeUpdate();

        // Check if any rows were affected (adding)
        return rowsAffected > 0;
    } catch (NumberFormatException ex) {
        throw new NumberException(msg: "Invalid number!");
    } catch (CommunicationsException em) {
        throw new DBException(msg: "Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg: "There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg: "There was a error, try again!");
    }
}
```

Figure 22. Adding dish methods

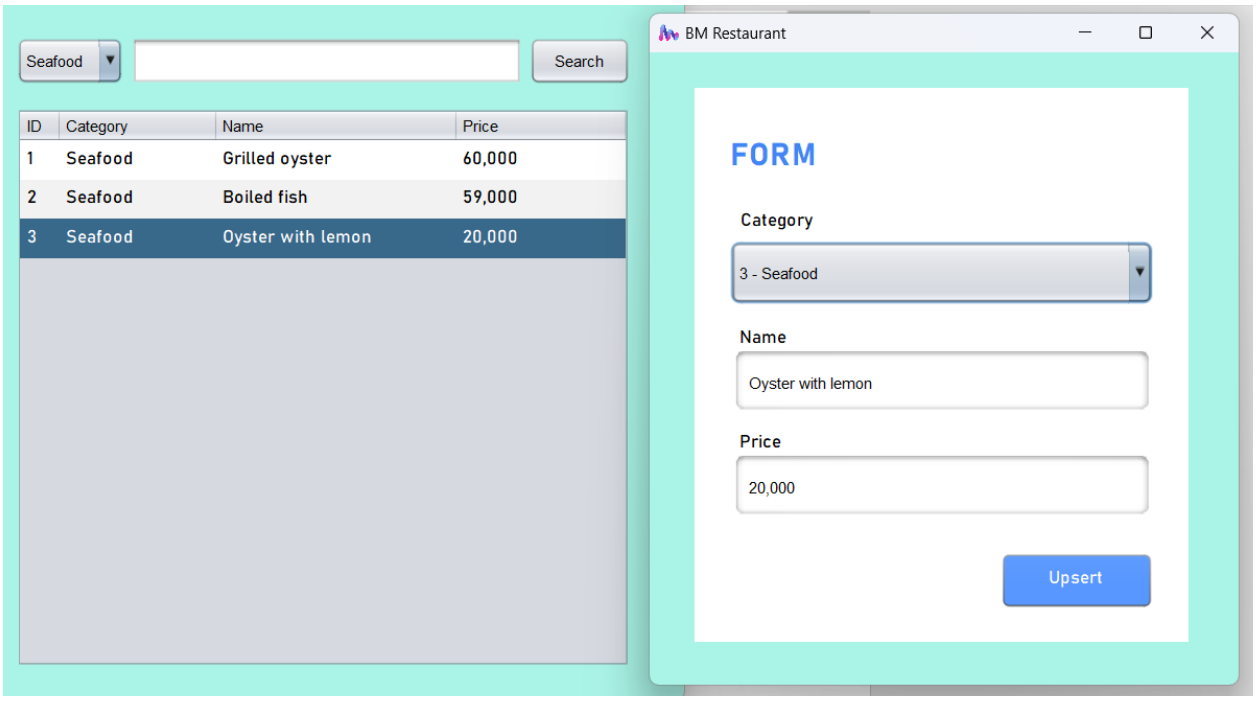
After performing the code above, all the cases will be informed for user



```java
public void resultAddUpdateDish(int result) {
switch (result) {
    case 0 -> JOptionPane.showMessageDialog(parentComponent: this, message: "The Dish has already existed!");
    case 1 -> {
        JOptionPane.showMessageDialog(parentComponent: this, message: "New Dish has been upserted!");
        this.dispose(); // Close the form
    }
    case -1 -> JOptionPane.showMessageDialog(parentComponent: this, message: "Cannot add, try again!");
    default -> JOptionPane.showMessageDialog(parentComponent: this, message: "There was a sudden error, try again!");
}
```

Figure 23. Result of adding and updating dish

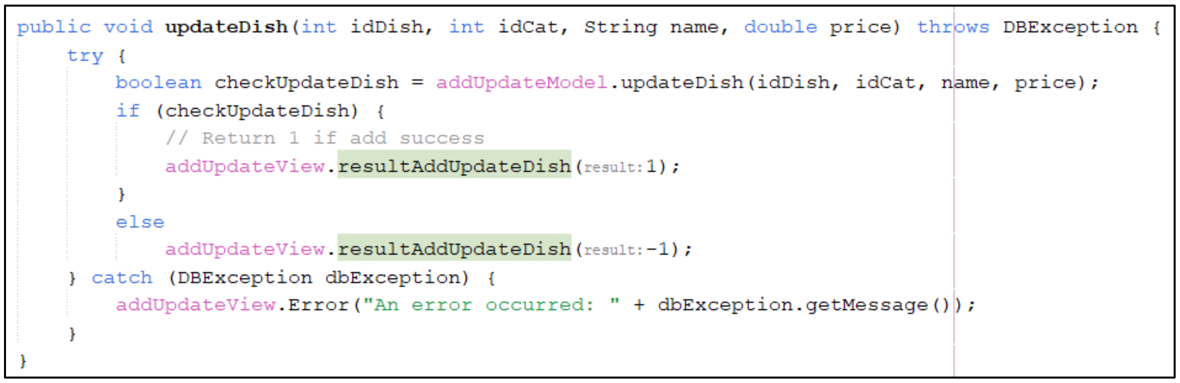
The updating form of dish will update the dish. If user does not enter anything, valídation is the same for adding. But in this, the information of dish user chooses will be show in the form



ID	Category	Name	Price
1	Seafood	Grilled oyster	60,000
2	Seafood	Boiled fish	59,000
3	Seafood	Oyster with lemon	20,000

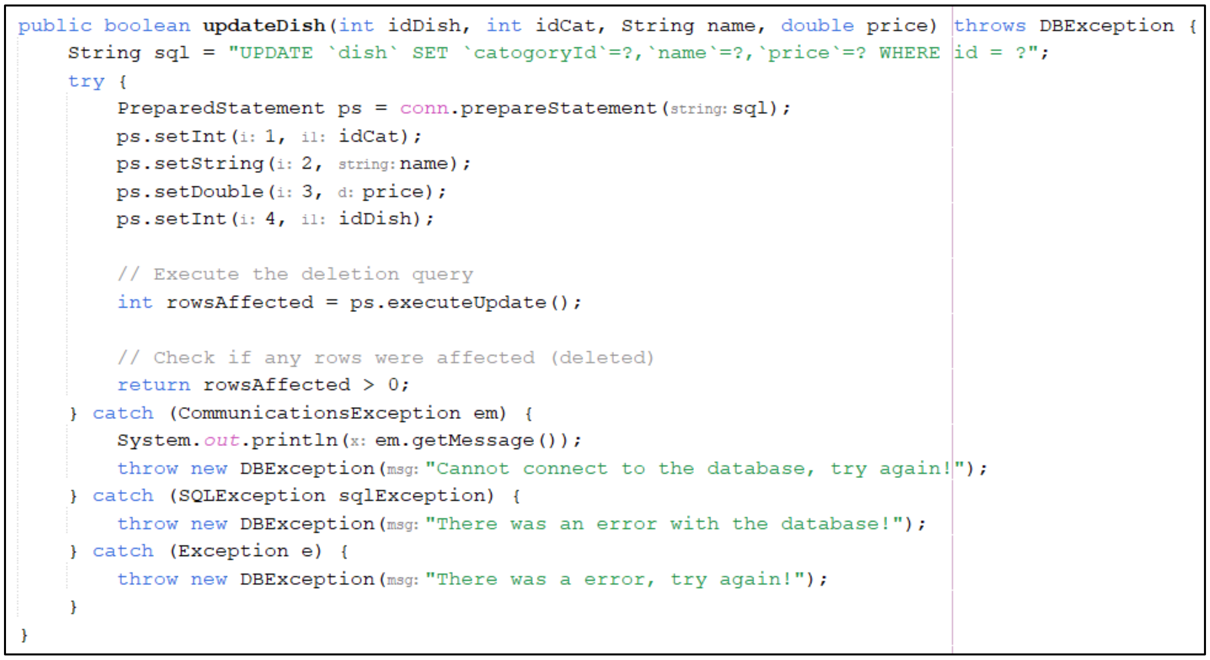
Name: Oyster with lemon
Price: 20,000

Figure 24. Updating form



```
public void updateDish(int idDish, int idCat, String name, double price) throws DBException {
    try {
        boolean checkUpdateDish = addUpdateModel.updateDish(idDish, idCat, name, price);
        if (checkUpdateDish) {
            // Return 1 if add success
            addUpdateView.resultAddUpdateDish(result:1);
        }
        else
            addUpdateView.resultAddUpdateDish(result:-1);
    } catch (DBException dbException) {
        addUpdateView.Error("An error occurred: " + dbException.getMessage());
    }
}
```

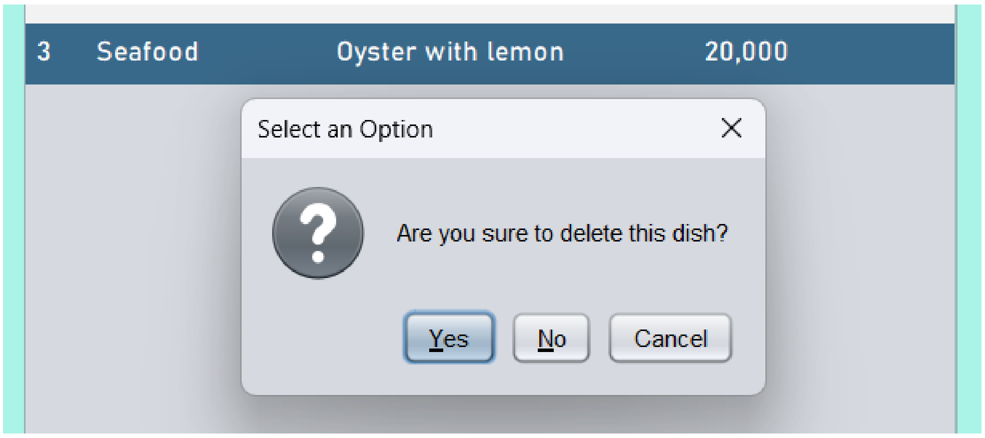
Figure 25. Updating dish action



```
public boolean updateDish(int idDish, int idCat, String name, double price) throws DBException {
    String sql = "UPDATE `dish` SET `catogoryId`=?,`name`=?,`price`=? WHERE id = ?";
    try {
        PreparedStatement ps = conn.prepareStatement(string: sql);
        ps.setInt(i: 1, i1: idCat);
        ps.setString(i: 2, string: name);
        ps.setDouble(i: 3, d: price);
        ps.setInt(i: 4, i1: idDish);

        // Execute the deletion query
        int rowsAffected = ps.executeUpdate();

        // Check if any rows were affected (deleted)
        return rowsAffected > 0;
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg: "Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg: "There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg: "There was a error, try again!");
    }
}
```

Figure 26. Updating dish method

If user wants to delete a dish, just click a deleting button on the options box, a message will pop up to confirm request of user



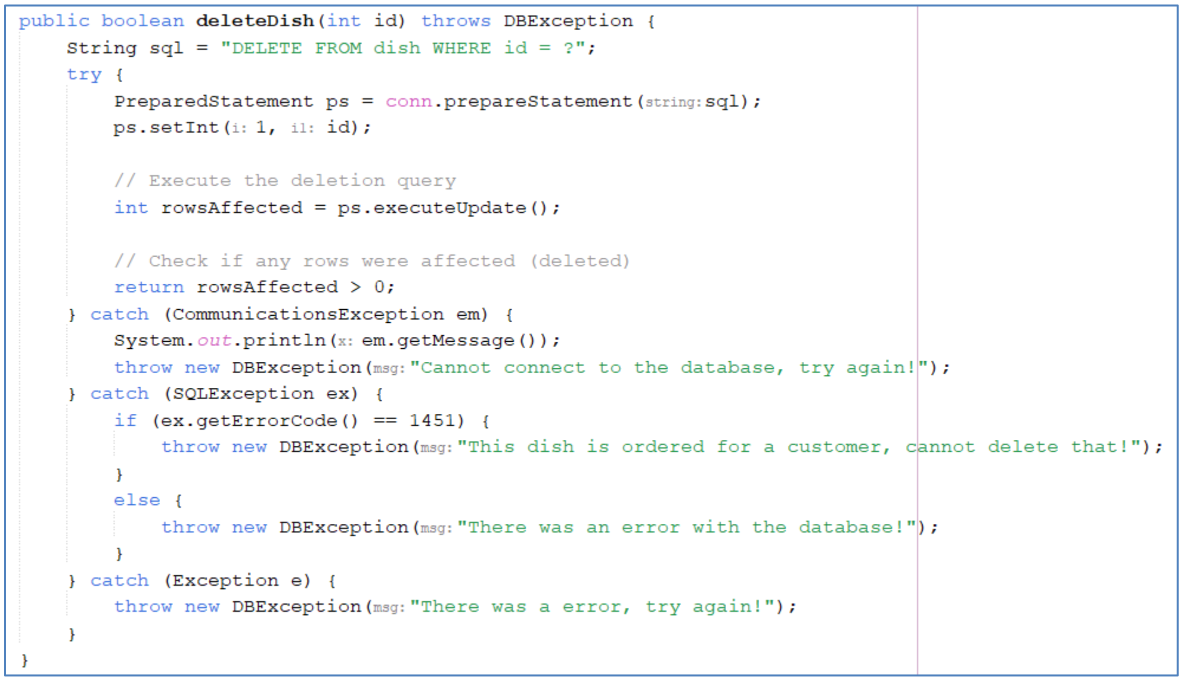
3 Seafood Oyster with lemon 20,000

Select an Option

Are you sure to delete this dish?

Figure 27. Confirm Message

In this part, there is a problem that is deleting the dish belonging to an order, so I catch the error 1451 (foreign key) so it fixes this problem



```java
public boolean deleteDish(int id) throws DBException {
    String sql = "DELETE FROM dish WHERE id = ?";
    try {
        PreparedStatement ps = conn.prepareStatement(string:sql);
        ps.setInt(i: 1, i1: id);

        // Execute the deletion query
        int rowsAffected = ps.executeUpdate();

        // Check if any rows were affected (deleted)
        return rowsAffected > 0;
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg:"Cannot connect to the database, try again!");
    } catch (SQLException ex) {
        if (ex.getErrorCode() == 1451) {
            throw new DBException(msg:"This dish is ordered for a customer, cannot delete that!");
        }
        else {
            throw new DBException(msg:"There was an error with the database!");
        }
    } catch (Exception e) {
        throw new DBException(msg:"There was a error, try again!");
    }
}
```

Figure 28. Deleting dish method

When user click import button, a tab will pop up to allow user choosing file to import

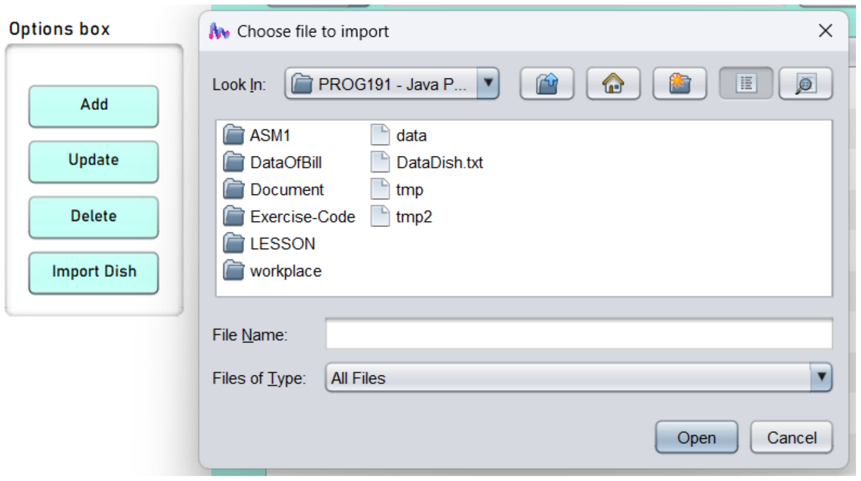
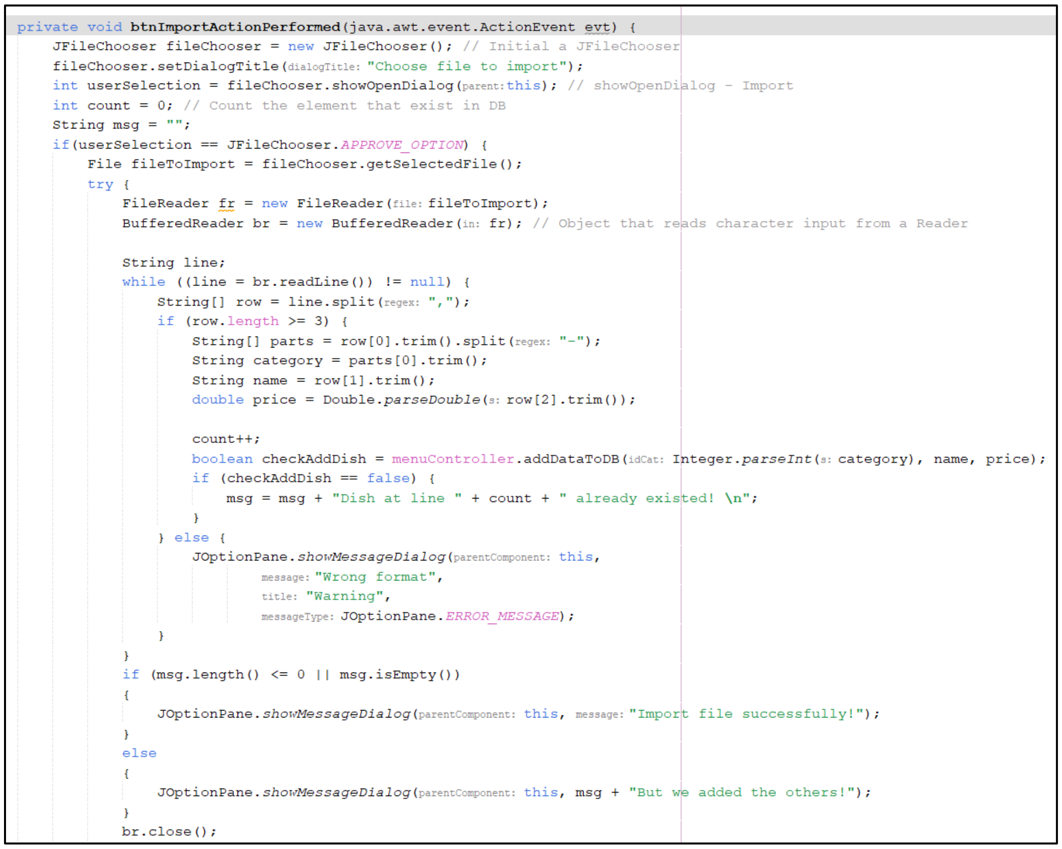


Figure 29. Interface of importing file

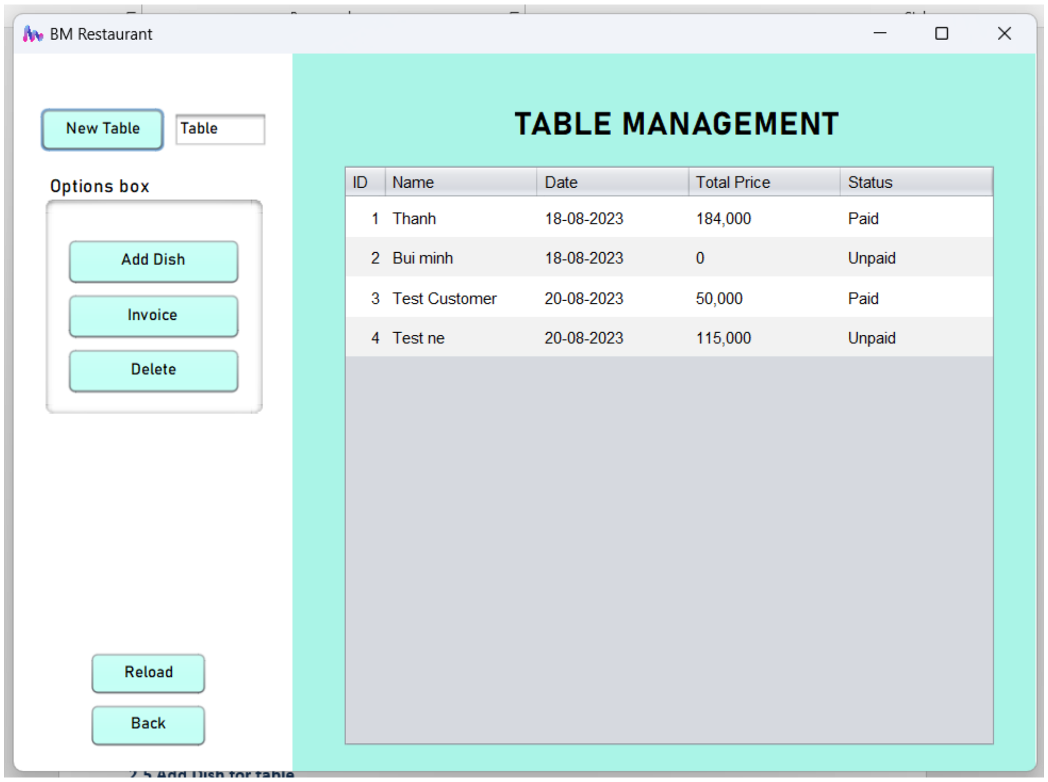


```java
private void btnImportActionPerformed(java.awt.event.ActionEvent evt) {
    JFileChooser fileChooser = new JFileChooser(); // Initial a JFileChooser
    fileChooser.setDialogTitle(dialogTitle: "Choose file to import");
    int userSelection = fileChooser.showOpenDialog(parent:this); // showOpenDialog - Import
    int count = 0; // Count the element that exist in DB
    String msg = "";
    if(userSelection == JFileChooser.APPROVE_OPTION) {
        File fileToImport = fileChooser.getSelectedFile();
        try {
            FileReader fr = new FileReader(file: fileToImport);
            BufferedReader br = new BufferedReader(in: fr); // Object that reads character input from a Reader

            String line;
            while ((line = br.readLine()) != null) {
                String[] row = line.split(regex: ",");
                if (row.length >= 3) {
                    String[] parts = row[0].trim().split(regex: "-");
                    String category = parts[0].trim();
                    String name = row[1].trim();
                    double price = Double.parseDouble(s: row[2].trim());

                    count++;
                    boolean checkAddDish = menuController.addDataToDB(idCat: Integer.parseInt(s: category), name, price);
                    if (checkAddDish == false) {
                        msg = msg + "Dish at line " + count + " already existed! \n";
                    }
                } else {
                    JOptionPane.showMessageDialog(parentComponent: this,
                            message: "Wrong format",
                            title: "Warning",
                            messageType: JOptionPane.ERROR_MESSAGE);
                }
            }
            if (msg.length() <= 0 || msg.isEmpty())
            {
                JOptionPane.showMessageDialog(parentComponent: this, message: "Import file successfully!");
            }
            else
            {
                JOptionPane.showMessageDialog(parentComponent: this, msg + "But we added the others!");
            }
            br.close();
```

Figure 30. Code of importing file

2.4 Table Management

This interface will be used to manage tables in the restaurant including adding and deleting a new table, viewing the invoice of that customer. When click New Table button, a inputDialog will be pop up to allow user entering customer's name



ID	Name	Date	Total Price	Status
1	Thanh	18-08-2023	184,000	Paid
2	Bui minh	18-08-2023	0	Unpaid
3	Test Customer	20-08-2023	50,000	Paid
4	Test ne	20-08-2023	115,000	Unpaid

Figure 31. Interface of table management

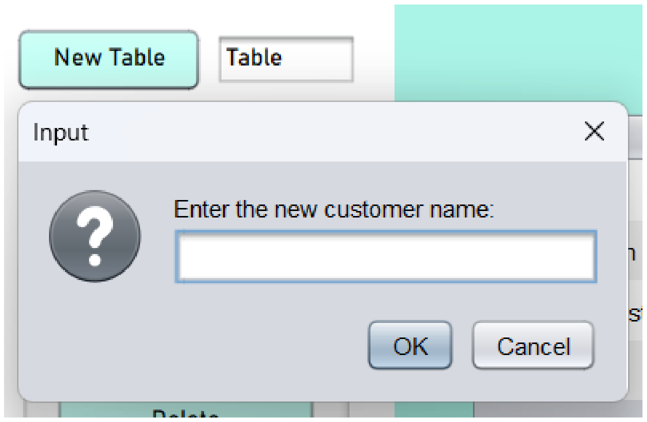
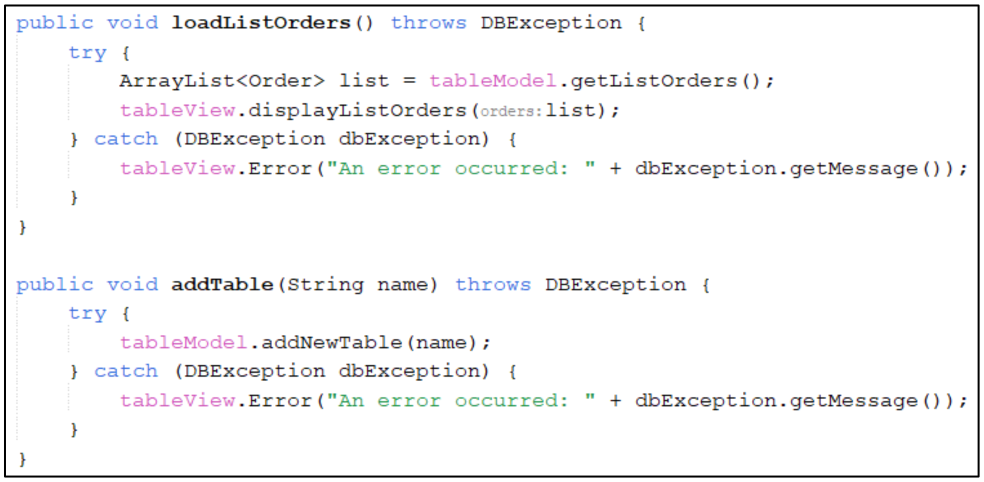


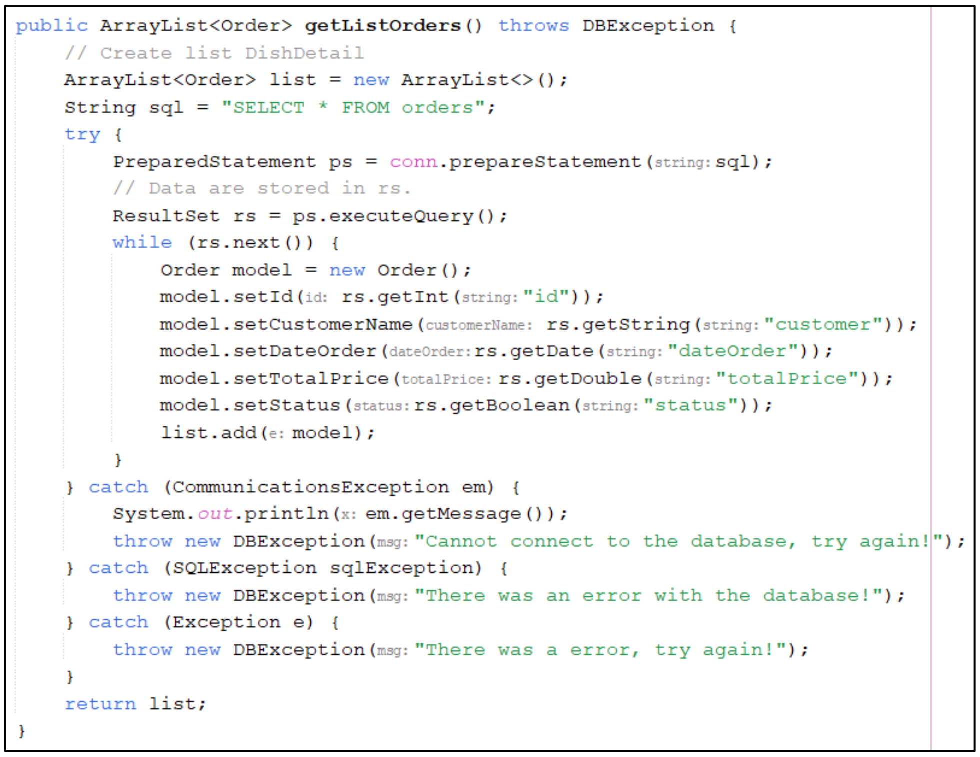
Figure 31. Input when create new table

The list of orders will be shown first to have an overview about all orders. All the exception will be caught at controller and show message at view



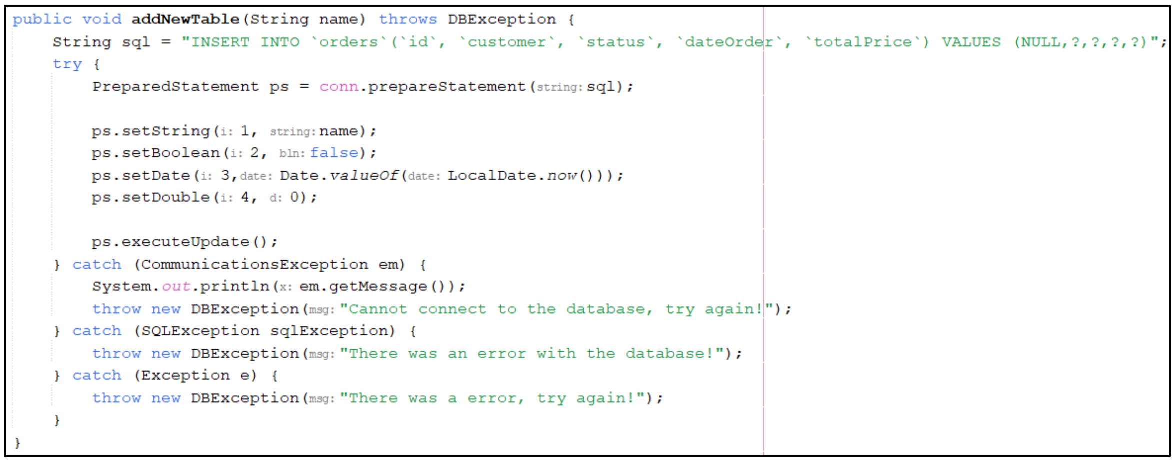
```java
public void loadListOrders() throws DBException {
    try {
        ArrayList<Order> list = tableModel.getListOrders();
        tableView.displayListOrders(orders: list);
    } catch (DBException dbException) {
        tableView.Error("An error occurred: " + dbException.getMessage());
    }
}

public void addTable(String name) throws DBException {
    try {
        tableModel.addNewTable(name);
    } catch (DBException dbException) {
        tableView.Error("An error occurred: " + dbException.getMessage());
    }
}
```

Figure 32. Loading dishes and adding table actions



```java
public ArrayList<Order> getListOrders() throws DBException {
    // Create list DishDetail
    ArrayList<Order> list = new ArrayList<>();
    String sql = "SELECT * FROM orders";
    try {
        PreparedStatement ps = conn.prepareStatement(string: sql);
        // Data are stored in rs.
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            Order model = new Order();
            model.setId(id: rs.getInt(string: "id"));
            model.setCustomerName(customerName: rs.getString(string: "customer"));
            model.setDateOrder(dateOrder: rs.getDate(string: "dateOrder"));
            model.setTotalPrice(totalPrice: rs.getDouble(string: "totalPrice"));
            model.setStatus(status: rs.getBoolean(string: "status"));
            list.add(e: model);
        }
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg: "Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg: "There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg: "There was a error, try again!");
    }
    return list;
}
```

Figure 33. Getting list orders method

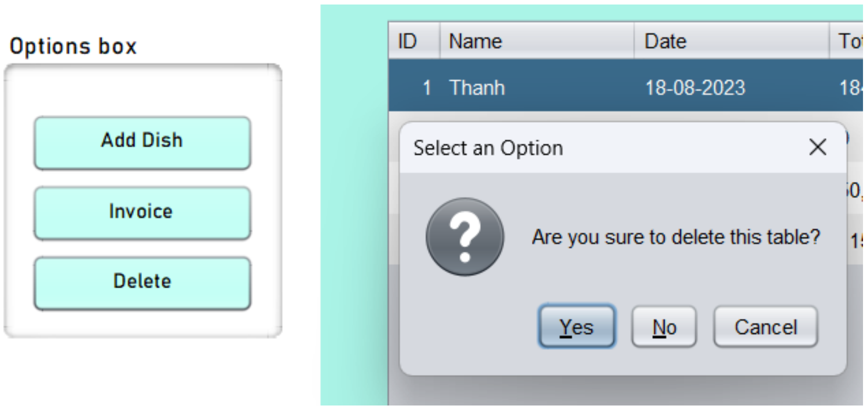


```java
public void addNewTable(String name) throws DBException {
    String sql = "INSERT INTO `orders`(`id`, `customer`, `status`, `dateOrder`, `totalPrice`) VALUES (NULL,?,?,?,?)";
    try {
        PreparedStatement ps = conn.prepareStatement(string: sql);

        ps.setString(i: 1, string: name);
        ps.setBoolean(i: 2, bln: false);
        ps.setDate(i: 3, date: Date.valueOf(date: LocalDate.now()));
        ps.setDouble(i: 4, d: 0);

        ps.executeUpdate();
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg: "Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg: "There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg: "There was a error, try again!");
    }
}
```

Figure 34. Adding table method

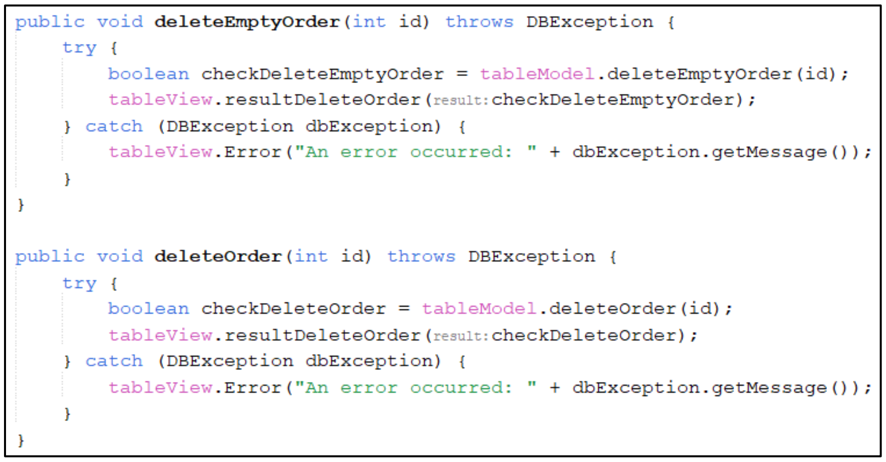
I divide deleting order into 2 cases

- If the total price of order is 0, this order can be deleted
- If the total price of order is greater than 0, this order needs to be paid before deleting



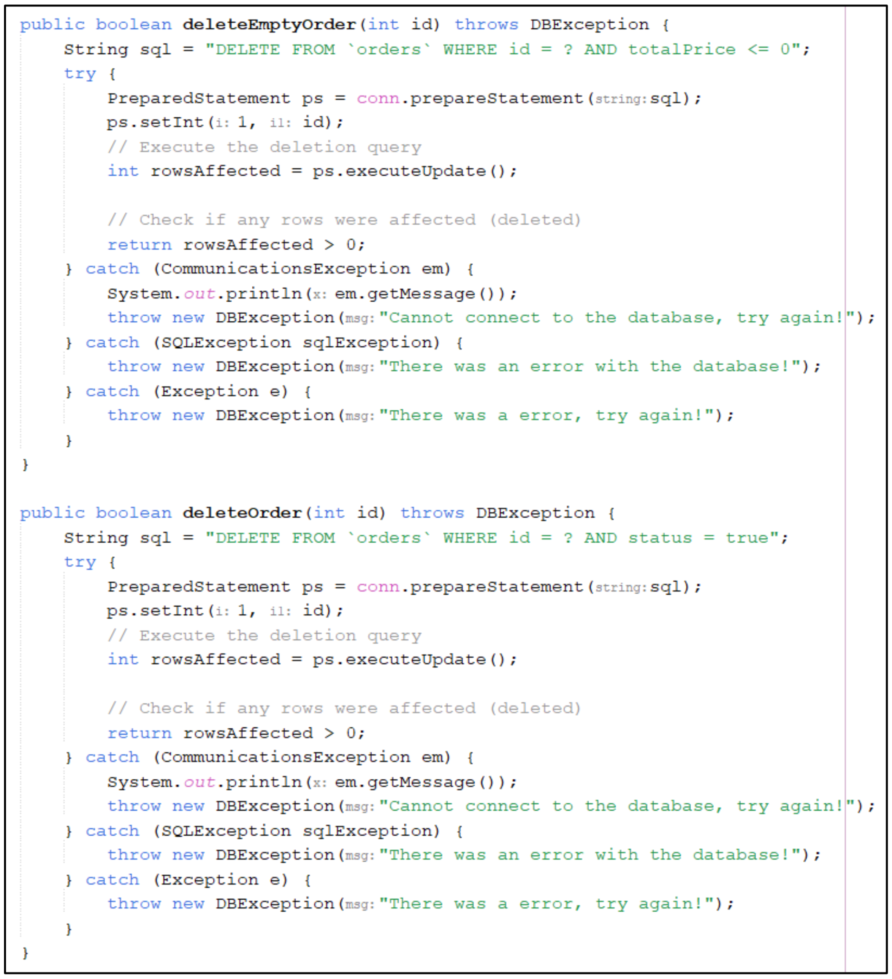
Are you sure to delete this table?

Figure 35. Confirm message when deleting order



```java
public void deleteEmptyOrder(int id) throws DBException {
    try {
        boolean checkDeleteEmptyOrder = tableModel.deleteEmptyOrder(id);
        tableView.resultDeleteOrder(result:checkDeleteEmptyOrder);
    } catch (DBException dbException) {
        tableView.Error("An error occurred: " + dbException.getMessage());
    }
}

public void deleteOrder(int id) throws DBException {
    try {
        boolean checkDeleteOrder = tableModel.deleteOrder(id);
        tableView.resultDeleteOrder(result:checkDeleteOrder);
    } catch (DBException dbException) {
        tableView.Error("An error occurred: " + dbException.getMessage());
    }
}
```

Figure 36. Deleting order actions

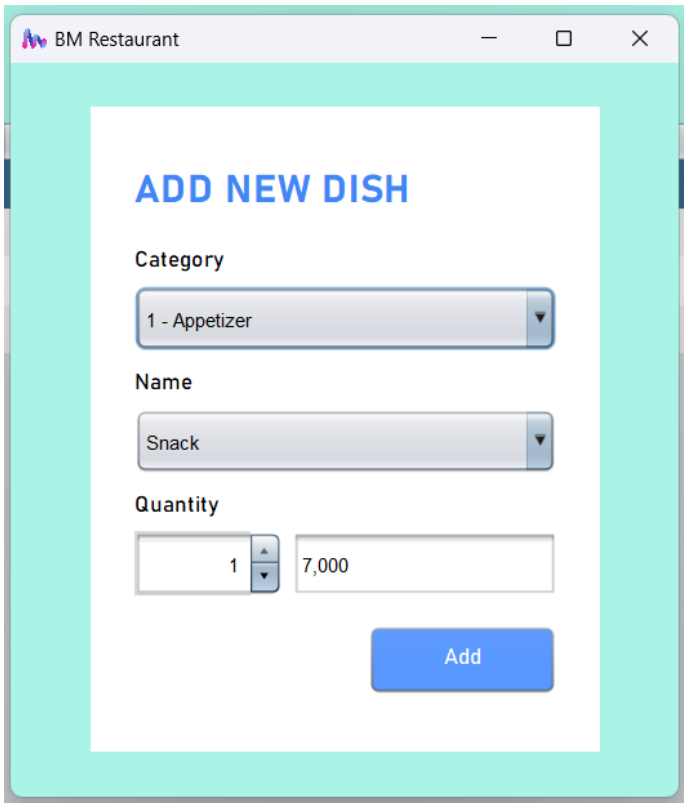


```java
public boolean deleteEmptyOrder(int id) throws DBException {
    String sql = "DELETE FROM `orders` WHERE id = ? AND totalPrice <= 0";
    try {
        PreparedStatement ps = conn.prepareStatement(string:sql);
        ps.setInt(i: 1, i1: id);
        // Execute the deletion query
        int rowsAffected = ps.executeUpdate();

        // Check if any rows were affected (deleted)
        return rowsAffected > 0;
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg:"Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg:"There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg:"There was a error, try again!");
    }
}

public boolean deleteOrder(int id) throws DBException {
    String sql = "DELETE FROM `orders` WHERE id = ? AND status = true";
    try {
        PreparedStatement ps = conn.prepareStatement(string:sql);
        ps.setInt(i: 1, i1: id);
        // Execute the deletion query
        int rowsAffected = ps.executeUpdate();

        // Check if any rows were affected (deleted)
        return rowsAffected > 0;
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg:"Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg:"There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg:"There was a error, try again!");
    }
}
```

Figure 37. Deleting order methods

2.5 Add Dish for table

This is the interface of adding new dish for a table, once user click on any categories, name of dish or quantity, price will be change immediately to improve UX



1 - Appetizer

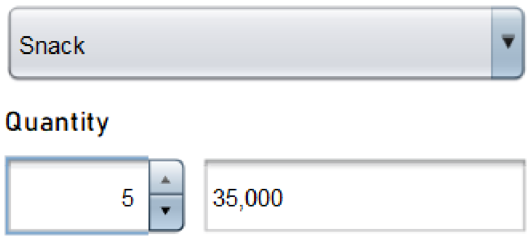
Snack

1

7,000

Figure 38. Adding dish for order interface

The price will be change when user change the quantity of dish



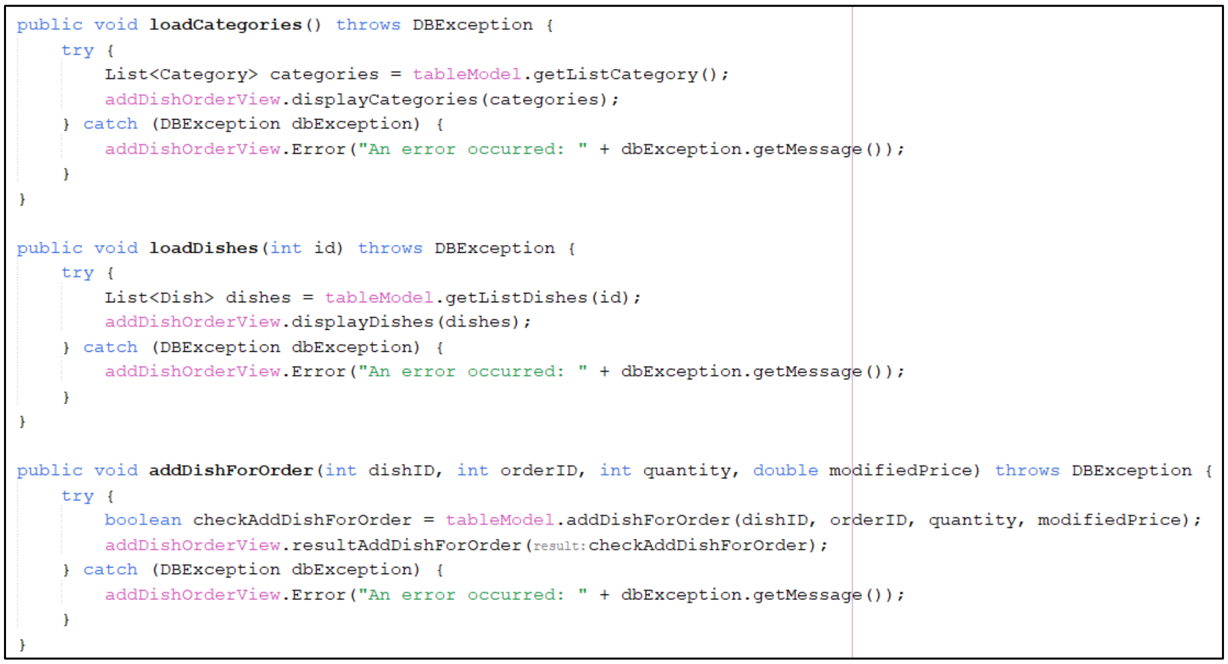
Snack

5

35,000

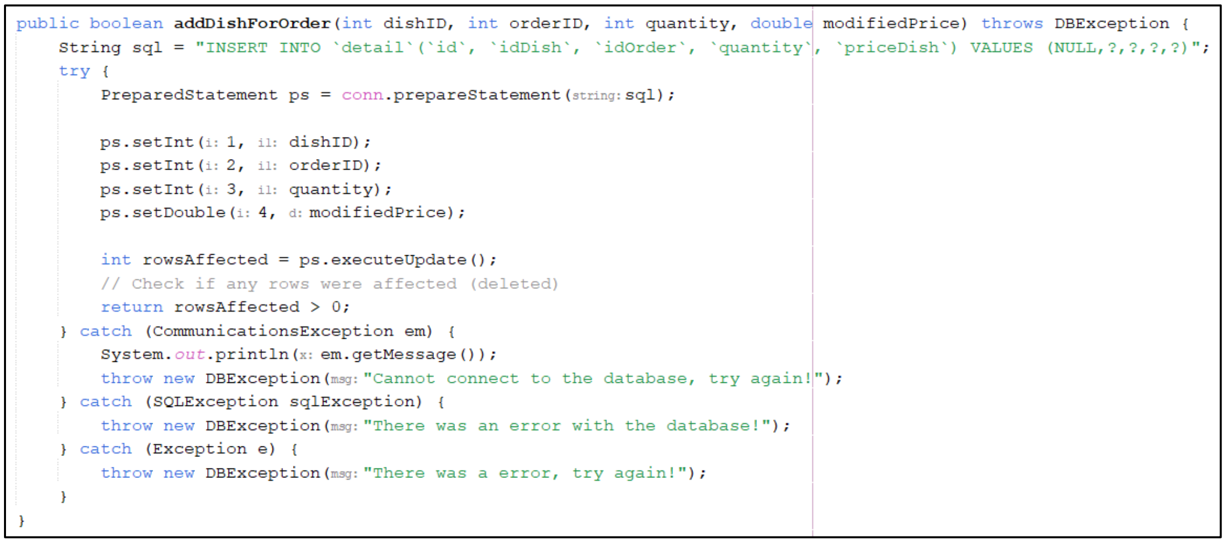
Figure 39. Price change when changing quantity

Before adding dish for order, the categories and dishes need to load fist on the form



```
public void loadCategories() throws DBException {
    try {
        List<Category> categories = tableModel.getListCategory();
        addDishOrderView.displayCategories(categories);
    } catch (DBException dbException) {
        addDishOrderView.Error("An error occurred: " + dbException.getMessage());
    }
}

public void loadDishes(int id) throws DBException {
    try {
        List<Dish> dishes = tableModel.getListDishes(id);
        addDishOrderView.displayDishes(dishes);
    } catch (DBException dbException) {
        addDishOrderView.Error("An error occurred: " + dbException.getMessage());
    }
}

public void addDishForOrder(int dishID, int orderID, int quantity, double modifiedPrice) throws DBException {
    try {
        boolean checkAddDishForOrder = tableModel.addDishForOrder(dishID, orderID, quantity, modifiedPrice);
        addDishOrderView.resultAddDishForOrder(result:checkAddDishForOrder);
    } catch (DBException dbException) {
        addDishOrderView.Error("An error occurred: " + dbException.getMessage());
    }
}
```

Figure 40. Table controller

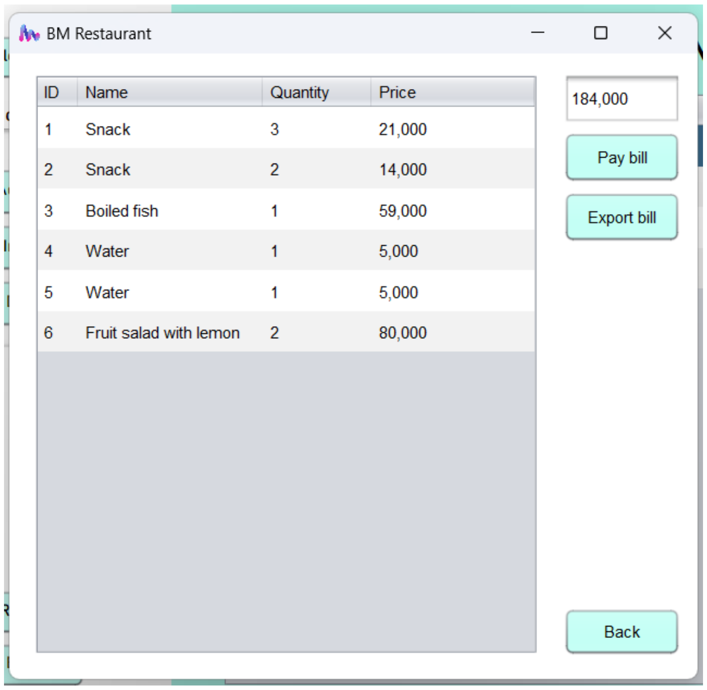


```
public boolean addDishForOrder(int dishID, int orderID, int quantity, double modifiedPrice) throws DBException {
    String sql = "INSERT INTO `detail`(`id`, `idDish`, `idOrder`, `quantity`, `priceDish`) VALUES (NULL,?,?,?,?)";
    try {
        PreparedStatement ps = conn.prepareStatement(string: sql);

        ps.setInt(i: 1, i1: dishID);
        ps.setInt(i: 2, i1: orderID);
        ps.setInt(i: 3, i1: quantity);
        ps.setDouble(i: 4, d: modifiedPrice);

        int rowsAffected = ps.executeUpdate();
        // Check if any rows were affected (deleted)
        return rowsAffected > 0;
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg: "Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg: "There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg: "There was a error, try again!");
    }
}
```

Figure 41. Adding dish for order method

2.6 Invoice

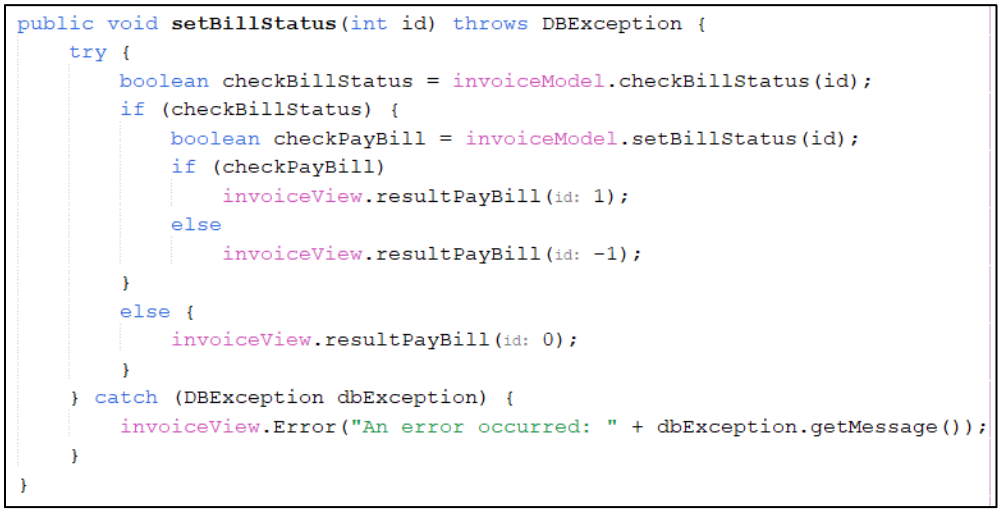
After finishing eating at the restaurant, the customer has to be paid the bill, this is the invoice interface including paying bill, exporting bill and showing information



184,000

ID	Name	Quantity	Price
1	Snack	3	21,000
2	Snack	2	14,000
3	Boiled fish	1	59,000
4	Water	1	5,000
5	Water	1	5,000
6	Fruit salad with lemon	2	80,000

Figure 42. Invoice interface

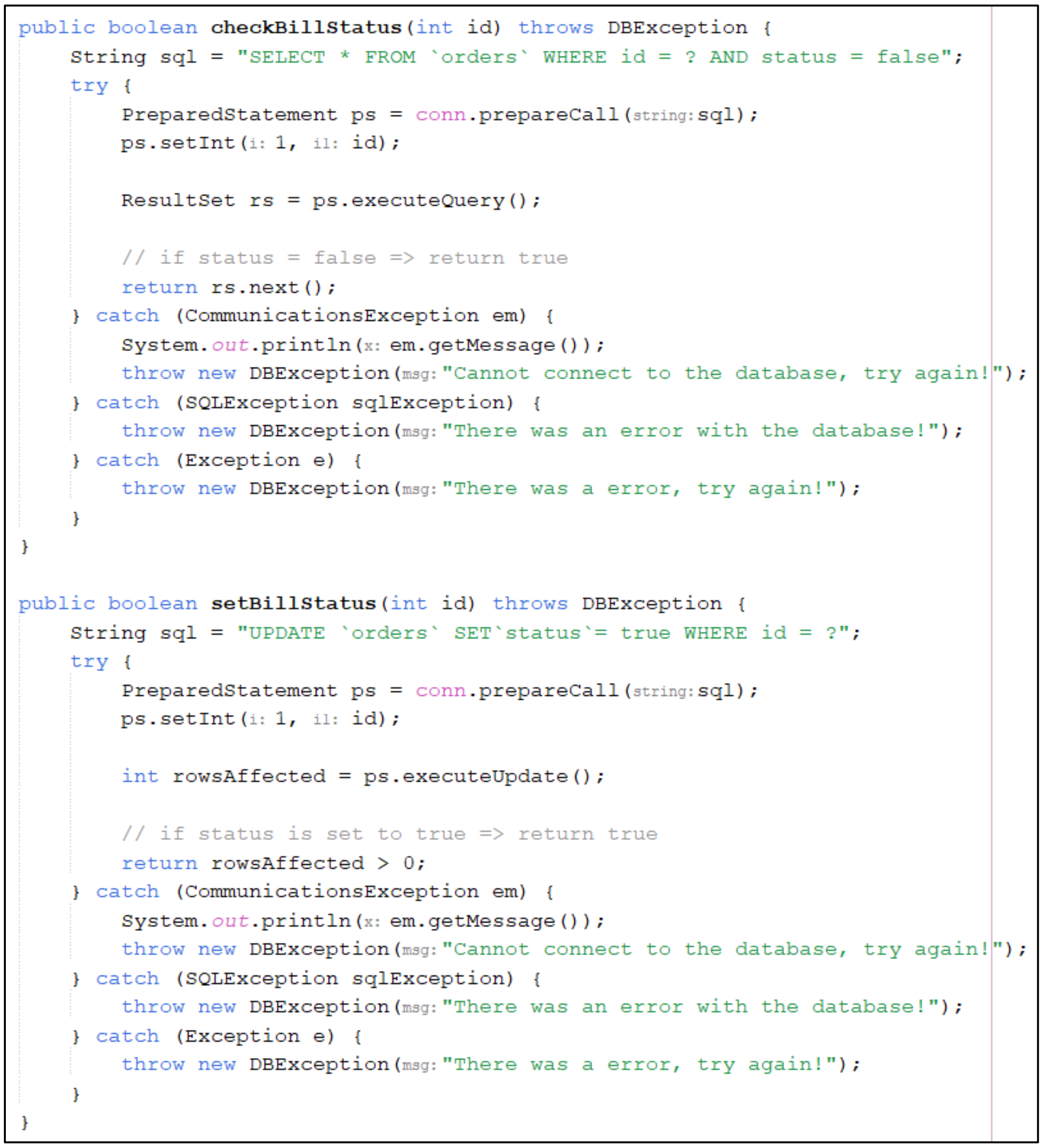


```
public void setBillStatus(int id) throws DBException {
    try {
        boolean checkBillStatus = invoiceModel.checkBillStatus(id);
        if (checkBillStatus) {
            boolean checkPayBill = invoiceModel.setBillStatus(id);
            if (checkPayBill)
                invoiceView.resultPayBill(id: 1);
            else
                invoiceView.resultPayBill(id: -1);
        }
        else {
            invoiceView.resultPayBill(id: 0);
        }
    } catch (DBException dbException) {
        invoiceView.Error("An error occurred: " + dbException.getMessage());
    }
}
```

Figure 43. Set bill status action

Befor paying bill, the bill needs to be checked to make sure about its status

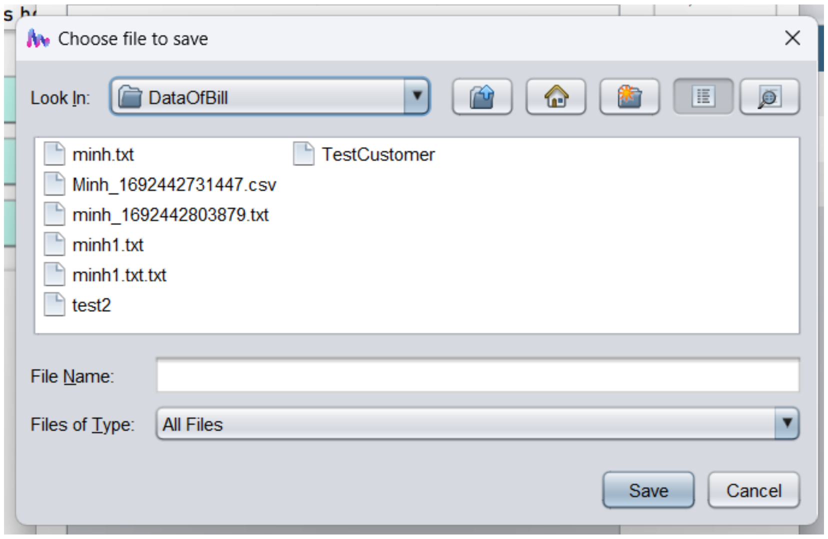
- If bill was paid, that bill cannot be paid anymore so customer needs to place new order
- If bill was unpaid, that bill can be paid with the check of receptionist



```java
public boolean checkBillStatus(int id) throws DBException {
    String sql = "SELECT * FROM `orders` WHERE id = ? AND status = false";
    try {
        PreparedStatement ps = conn.prepareCall(string: sql);
        ps.setInt(i: 1, i1: id);

        ResultSet rs = ps.executeQuery();

        // if status = false => return true
        return rs.next();
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg: "Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg: "There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg: "There was a error, try again!");
    }
}

public boolean setBillStatus(int id) throws DBException {
    String sql = "UPDATE `orders` SET`status`= true WHERE id = ?";
    try {
        PreparedStatement ps = conn.prepareCall(string: sql);
        ps.setInt(i: 1, i1: id);

        int rowsAffected = ps.executeUpdate();

        // if status is set to true => return true
        return rowsAffected > 0;
    } catch (CommunicationsException em) {
        System.out.println(x: em.getMessage());
        throw new DBException(msg: "Cannot connect to the database, try again!");
    } catch (SQLException sqlException) {
        throw new DBException(msg: "There was an error with the database!");
    } catch (Exception e) {
        throw new DBException(msg: "There was a error, try again!");
    }
}
```

Figure 44. Set bill status methods

When exporting the invoice, I want exporting the form I customed. It will print the information of that order and all dishes that customer ordered


Choose file to save

Look In: DataOfBill

```
minh.txt                      TestCustomer
Minh_1692442731447.csv
minh_1692442803879.txt
minh1.txt
minh1.txt.txt
test2
```

File Name:

Files of Type: All Files

Figure 45. Exporting invoice interface

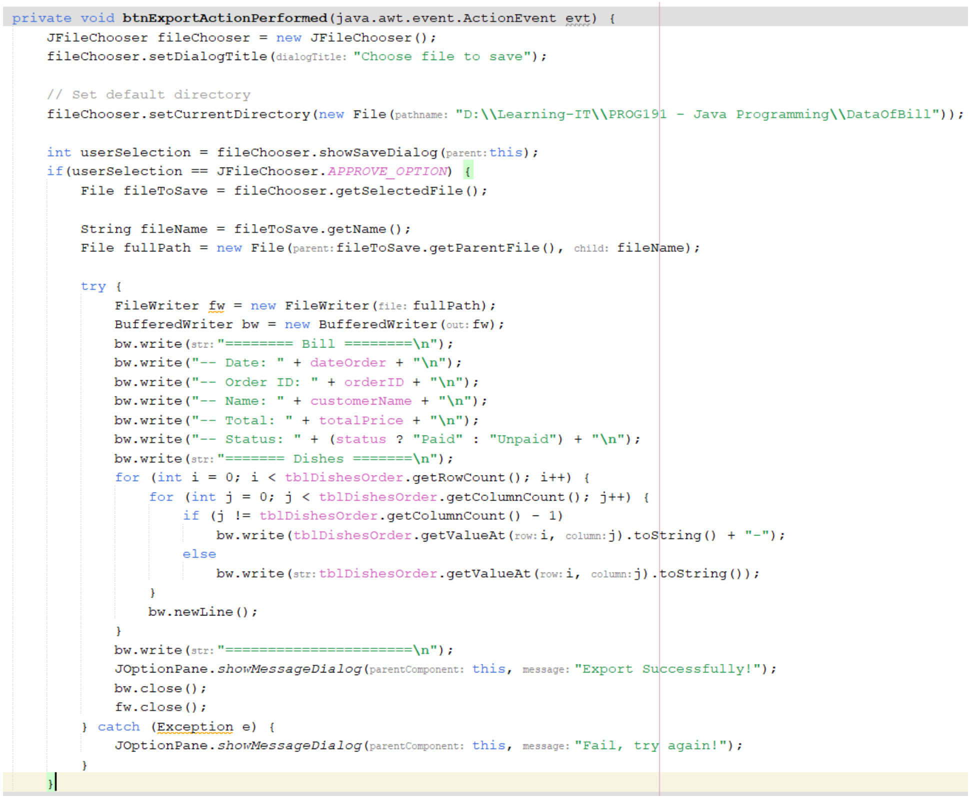

```java
private void btnExportActionPerformed(java.awt.event.ActionEvent evt) {
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setDialogTitle(dialogTitle: "Choose file to save");

    // Set default directory
    fileChooser.setCurrentDirectory(new File(pathname: "D:\\Learning-IT\\PROG191 - Java Programming\\DataOfBill"));

    int userSelection = fileChooser.showSaveDialog(parent:this);
    if(userSelection == JFileChooser.APPROVE_OPTION) {
        File fileToSave = fileChooser.getSelectedFile();

        String fileName = fileToSave.getName();
        File fullPath = new File(parent:fileToSave.getParentFile(), child: fileName);

        try {
            FileWriter fw = new FileWriter(file: fullPath);
            BufferedWriter bw = new BufferedWriter(out: fw);
            bw.write(str:"======== Bill ========\n");
            bw.write("-- Date: " + dateOrder + "\n");
            bw.write("-- Order ID: " + orderID + "\n");
            bw.write("-- Name: " + customerName + "\n");
            bw.write("-- Total: " + totalPrice + "\n");
            bw.write("-- Status: " + (status ? "Paid" : "Unpaid") + "\n");
            bw.write(str:"======= Dishes =======\n");
            for (int i = 0; i < tblDishesOrder.getRowCount(); i++) {
                for (int j = 0; j < tblDishesOrder.getColumnCount(); j++) {
                    if (j != tblDishesOrder.getColumnCount() - 1)
                        bw.write(tblDishesOrder.getValueAt(row:i, column:j).toString() + "-");
                    else
                        bw.write(str:tblDishesOrder.getValueAt(row:i, column:j).toString());
                }
                bw.newLine();
            }
            bw.write(str:"=====================\n");
            JOptionPane.showMessageDialog(parentComponent: this, message:"Export Successfully!");
            bw.close();
            fw.close();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(parentComponent: this, message:"Fail, try again!");
        }
    }
```

Figure 46. Code of exporting file

V. TEST

1. Test plan

1.1 Test entity

- Ensure that all entities and custom entities have been properly defined
- Validate that new instances of entities are created with the expected properties and default values
- Verify that all getter and setter methods function correctly by inputting data and retrieving data from entities

1.2 Test Model

- Test various methods in the Model package that interact with the database and handle data
- Verify that CRUD (Create, Read, Update, Delete) operations are working as intended
- Test scenarios involving complex queries, joins, and data retrieval

1.3 Test input validation and message inform

- Verify that user input is validated according to defined rules and constraints
- Ensure that users receive clear and informative messages when validation fails
- Test various input scenarios to cover both valid and invalid cases
- Verify that error messages are specific to the type of validation failure (e.g., "Invalid email format," "Value must be positive")

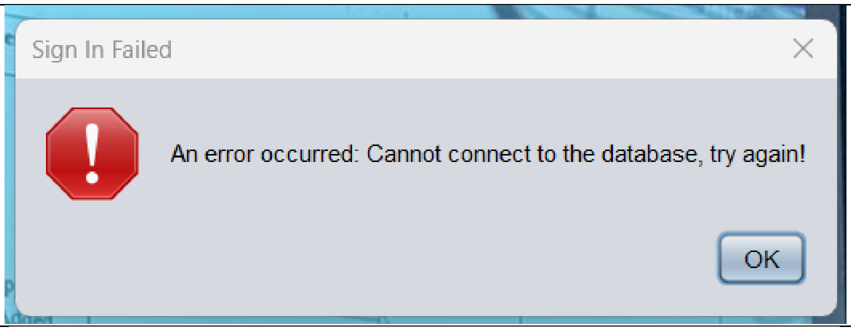
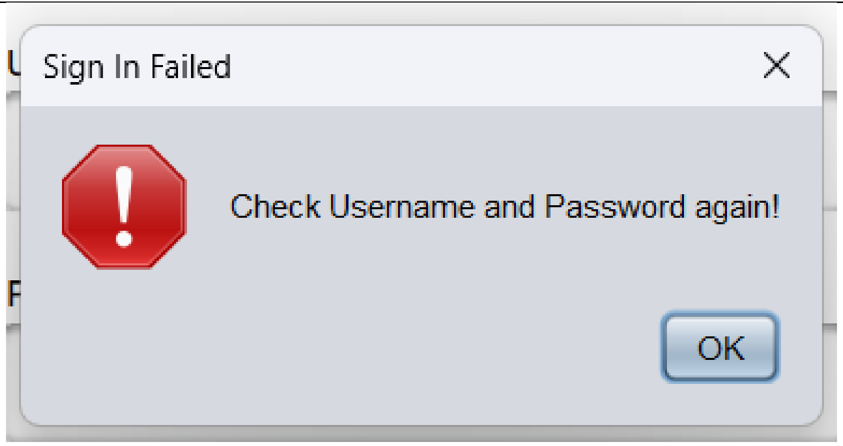
1.4 Test menu-driven user interface

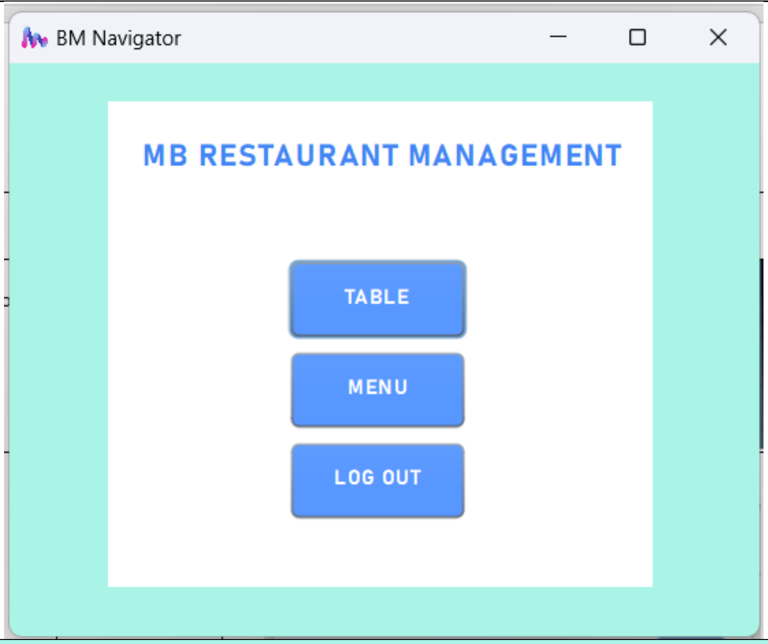
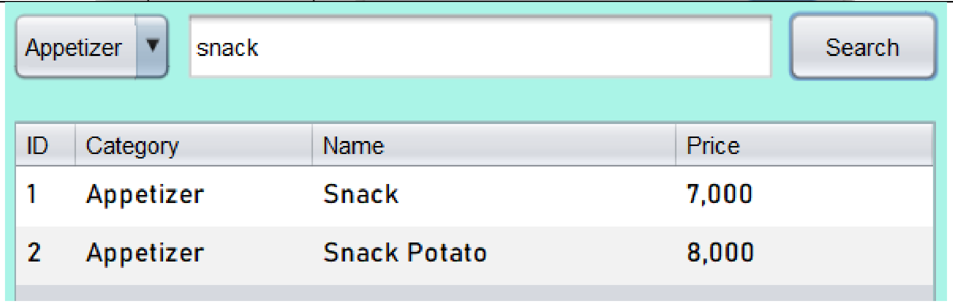
- Verify that menu options are displayed accurately and consistently
- Ensure that users can access various features easily through the menu
- Confirm that selected menu options leading to the intended functionalities
- Test that navigating back from sub-menus returns the user to the appropriate parent menu

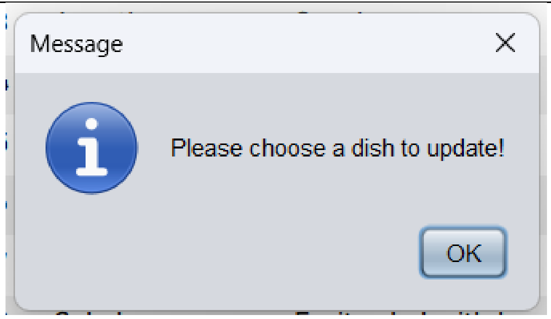
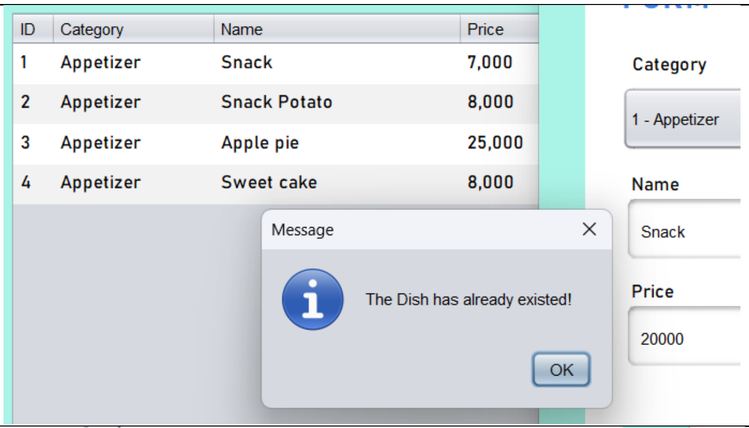
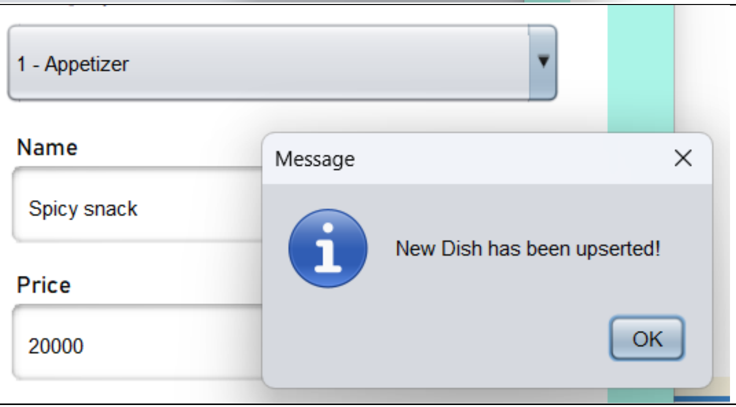
1.5 Test edge cases and error handling

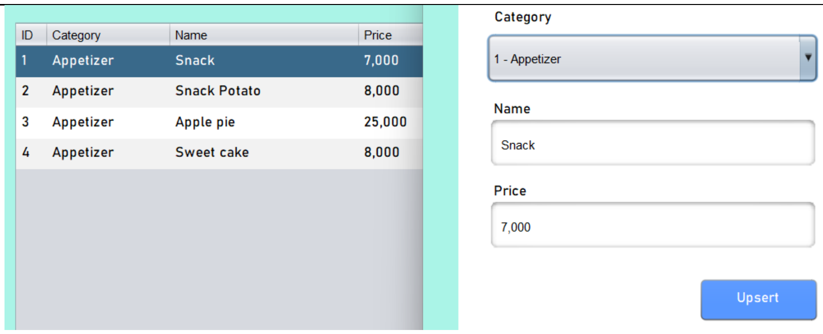
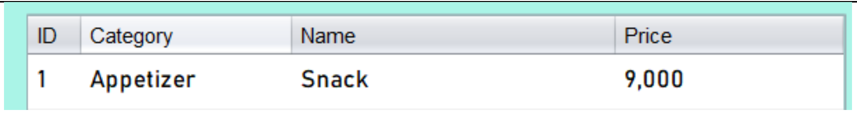
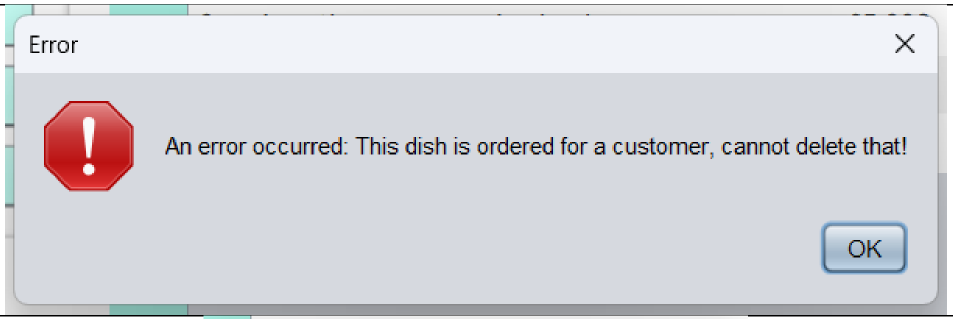
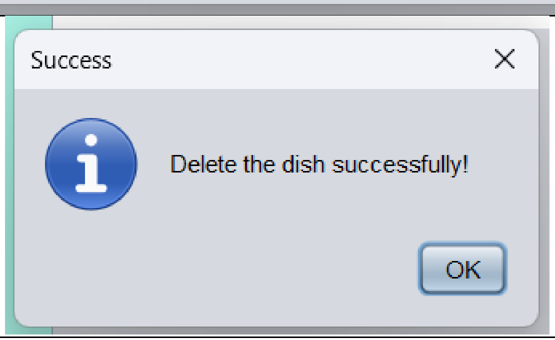
- Test the program when the database is empty (no records)
- Verify that appropriate messages or actions are taken, and the program doesn't crash or freeze
- Test providing non-integer values (such as letters) when numeric inputs are expected
- Confirm that the program gracefully handles such cases and guides users appropriately
- Test database connection failure scenarios, such as disconnecting from the network or shutting down the database server

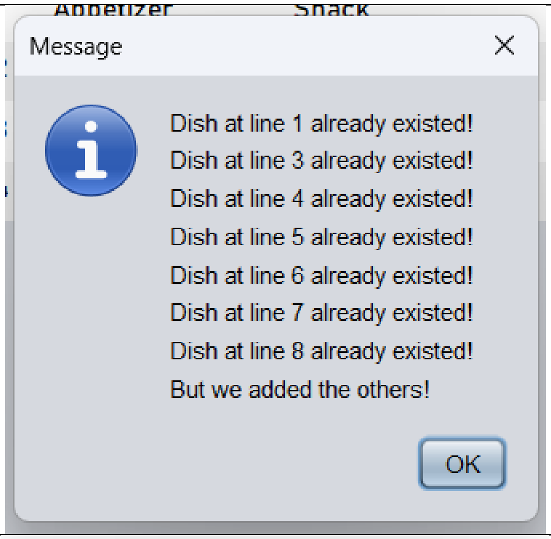
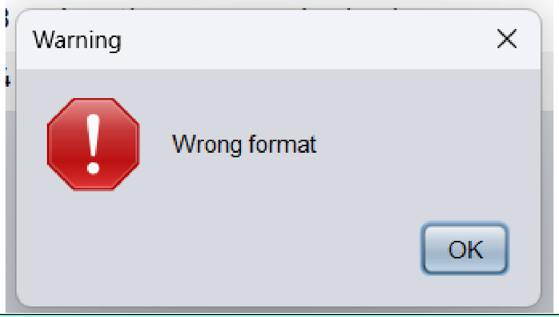
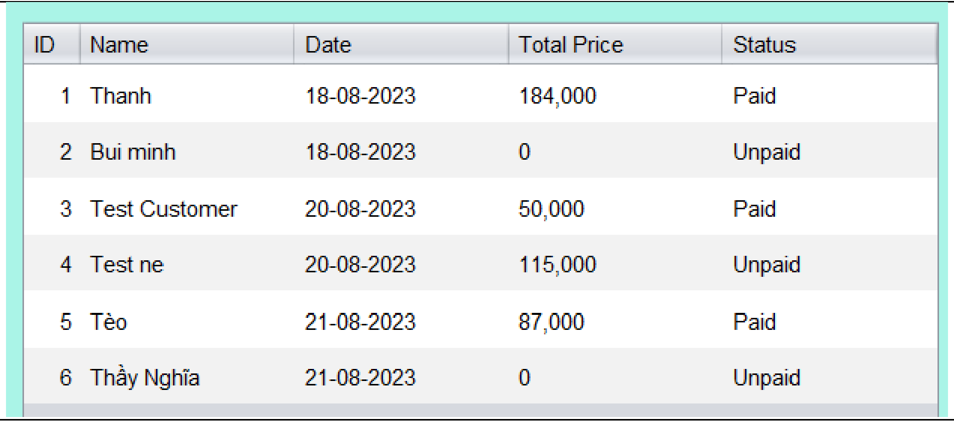
2. Test log

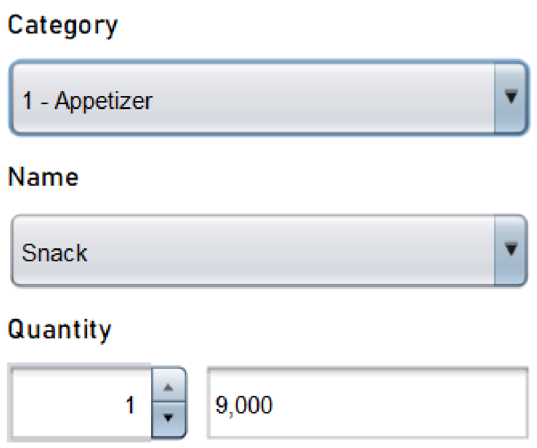
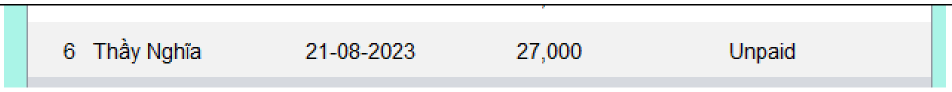
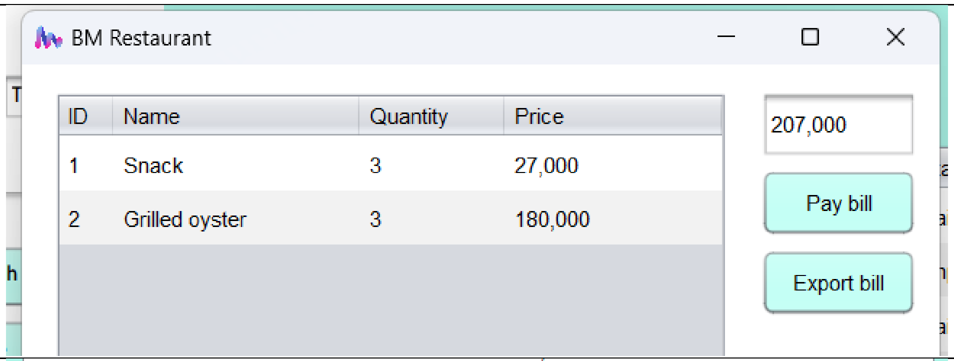
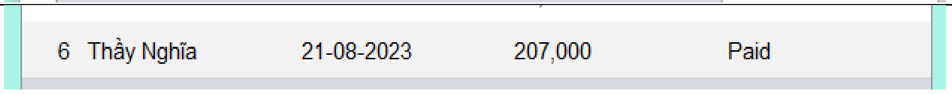
Case	Action	Expected	Actual Result	Pass/Fail
1	Turn off connection to database	Inform message "cannot connect with database"	 Sign In Failed An error occurred: Cannot connect to the database, try again! OK	Pass
2	Enter wrong username or password in login form	Inform error "incorrect information"	 Sign In Failed Check Username and Password again! OK	Pass

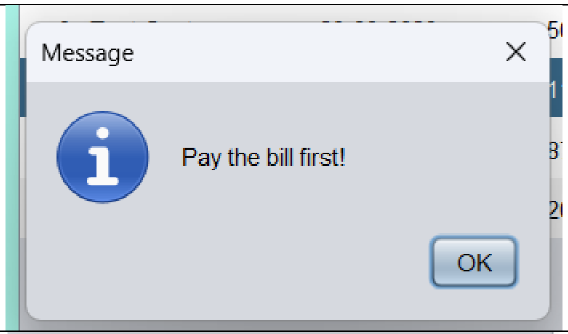
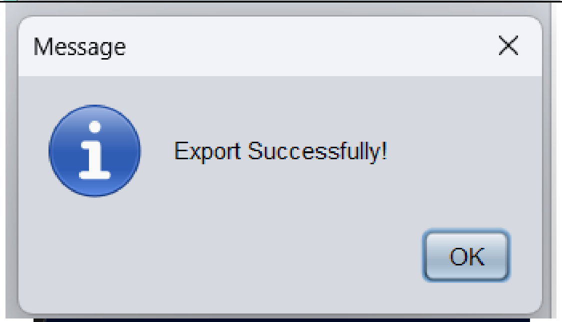
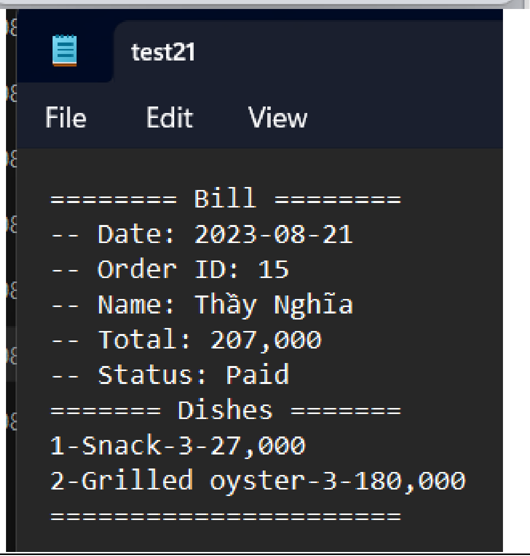
3	Show navigator when entering correct username and password	Show navigator	 BM Navigator MB RESTAURANT MANAGEMENT TABLE MENU LOG OUT	Pass
4	Show list of dishes regarding category and name of dish	Show correct list of dishes with filter	 ID | Category | Name | Price 1 | Appetizer | Snack | 7,000 2 | Appetizer | Snack Potato | 8,000	Pass

5	Click update and delete button without choosing row	Show notification about that	 Message Please choose a dish to update! OK	Pass
6	Add dish that existed in database	Show error message that it was existed	 ID | Category | Name | Price 1 | Appetizer | Snack | 7,000 2 | Appetizer | Snack Potato | 8,000 3 | Appetizer | Apple pie | 25,000 4 | Appetizer | Sweet cake | 8,000 Category: 1 - Appetizer Name: Snack Price: 20000 Message The Dish has already existed! OK	Pass
7	Enter unique dish	Show success message	 1 - Appetizer Name: Spicy snack Price: 20000 Message New Dish has been upserted! OK	Pass

8	Click a row and click update	Show correct information of that dish in form	 ID | Category | Name | Price 1 | Appetizer | Snack | 7,000 2 | Appetizer | Snack Potato | 8,000 3 | Appetizer | Apple pie | 25,000 4 | Appetizer | Sweet cake | 8,000 Category: 1 - Appetizer Name: Snack Price: 7,000 Upsert	Pass
9	Update information for dish	Show new information in table	 ID | Category | Name | Price 1 | Appetizer | Snack | 9,000	Pass
10	Delete the dish that belongs to an order in restaurant	Show error message	 Error An error occurred: This dish is ordered for a customer, cannot delete that! OK	Pass
11	Delete a dish	Show success message and remove it from table	 Success Delete the dish successfully! OK	Pass

12	Import the dish that existed in database	Show existed dishes	 Message Dish at line 1 already existed! Dish at line 3 already existed! Dish at line 4 already existed! Dish at line 5 already existed! Dish at line 6 already existed! Dish at line 7 already existed! Dish at line 8 already existed! But we added the others! OK	Pass
13	Import wrong format file	Show "wrong format" message	 Warning Wrong format OK	Pass
14	Add new table for Mr Nghia	Show Nghia's order in the table	 ID | Name | Date | Total Price | Status 1 | Thanh | 18-08-2023 | 184,000 | Paid 2 | Bui minh | 18-08-2023 | 0 | Unpaid 3 | Test Customer | 20-08-2023 | 50,000 | Paid 4 | Test ne | 20-08-2023 | 115,000 | Unpaid 5 | Tèo | 21-08-2023 | 87,000 | Paid 6 | Thầy Nghĩa | 21-08-2023 | 0 | Unpaid	Pass

15	Just select category or dish, quantity and price will be calculated automatically	Show price matching with quantity	 Category: 1 - Appetizer; Name: Snack; Quantity: 1, 9,000	Pass
16	Add some dish for Mr Nghia's table	Increase total price	 6 Thầy Nghĩa 21-08-2023 27,000 Unpaid	Pass
17	View order of Mr Nghia	Total price and all dishes will be shown	 BM Restaurant ID | Name | Quantity | Price 1 | Snack | 3 | 27,000 2 | Grilled oyster | 3 | 180,000 207,000	Pass
18	Click paybill	Set status to paid	 6 Thầy Nghĩa 21-08-2023 207,000 Paid	Pass

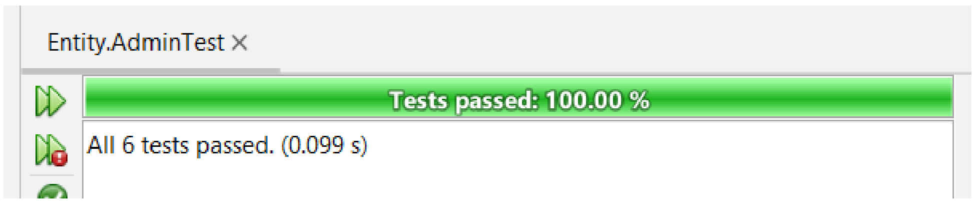
19	Delete an unpaid order	Show message paying bill first	 Pay the bill first!	Pass
20	Click export invoice	Show correct format and success message	 Export Successfully!  ======== Bill ======== -- Date: 2023-08-21 -- Order ID: 15 -- Name: Thầy Nghĩa -- Total: 207,000 -- Status: Paid ======= Dishes ======= 1-Snack-3-27,000 2-Grilled oyster-3-180,000 ======================	Pass

3. Junit Test

3.1 Admin entity

```
public class AdminTest {

   public Admin newAdmin;

   public AdminTest() {
      newAdmin = new Admin(1,"bminh","123");
   }

   @Test
   public void testGetId() {
      int expResult = 1;
      int result = newAdmin.getId();
      assertEquals(expResult, result);
   }

   @Test
   public void testSetId() {
      int expResult = 2;
      int id = 2;
      newAdmin.setId(id);
      int result = newAdmin.getId();
      assertEquals(expResult, result);
   }

   @Test
   public void testGetUsername() {
      String expResult = "bminh";
      String result = newAdmin.getUsername();
      assertEquals(expResult, result);
   }

   @Test
   public void testSetUsername() {
      String expResult = "minhnew";
      String username = "minhnew";
      newAdmin.setUsername(username);
      String result = newAdmin.getUsername();
      assertEquals(expResult, result);
   }

   @Test
   public void testGetPassword() {
      String expResult = "123";
      String result = newAdmin.getPassword();
      assertEquals(expResult, result);
   }
```

```
    @Test
    public void testSetPassword() {
        String expResult = "123456";
        String password = "123456";
        newAdmin.setPassword(password);
        String result = newAdmin.getPassword();
        assertEquals(expResult, result);
    }

}
```

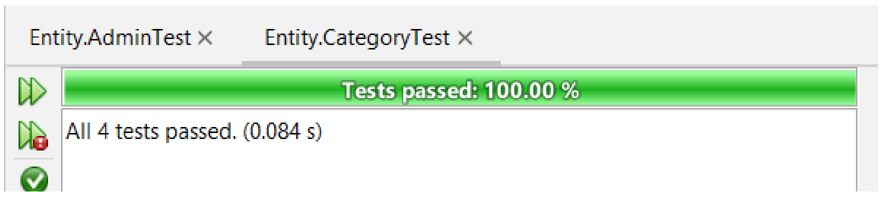


```
Entity.AdminTest ×
Tests passed: 100.00 %
All 6 tests passed. (0.099 s)
```

Figure 47. Result of test

3.2 Category entity

```
public class CategoryTest {

    public CategoryTest() {
    }

    @Before
    public void setUp() {
    }

    @Test
    public void testGetId() {
        System.out.println("getId");
        Category instance = new Category(1,"Seafood");
        int expResult = 1;
        int result = instance.getId();
        assertEquals(expResult, result);
    }

    @Test
    public void testSetId() {
        System.out.println("setId");
        int id = 0;
        Category instance = new Category();
        instance.setId(id);
    }

    @Test
    public void testGetName() {
        System.out.println("getName");
        Category instance = new Category(1,"Parama");
        String expResult = "Parama ";
        String result = instance.getName();
        assertEquals(expResult, result);
    }

    @Test
    public void testSetName() {
        System.out.println("setName");
        String name = "";
        Category instance = new Category();
        instance.setName(name);
    }

}
```


Entity.AdminTest × Entity.CategoryTest ×

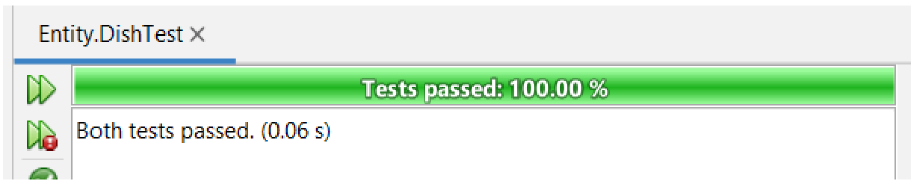
Tests passed: 100.00 %

All 4 tests passed. (0.084 s)

Figure 48. Result of test

3.3 Dish entity

```
public class DishTest {

    Dish newDish = new Dish(1,2,"Salmon",20000);

    public DishTest() {
    }

    @Test
    public void testGet() {
        // Correct Dish
        assertEquals(1, newDish.getId());
        assertEquals(2, newDish.getCategoryId());
        assertEquals("Salmon", newDish.getName());
        assertTrue(20000 == newDish.getPrice());
        // Incorrect Dish
        assertFalse(2 == newDish.getId());
        assertFalse(1 == newDish.getCategoryId());
        assertFalse("Salmonnnn".equals(newDish.getName()));
        assertFalse(40000 == newDish.getPrice());
    }

    @Test
    public void testSet() {
        Dish instance = new Dish();
        instance.setId(5);
        instance.setCategoryId(12);
        instance.setName("Lemon");
        instance.setPrice(40000);

        assertFalse(instance.getId() == 1);
        assertFalse(instance.getCategoryId() == 2);
        assertFalse(instance.getName().equals("Salmon"));
        assertFalse(instance.getPrice() == 20000);
    }
}
```


Entity.DishTest

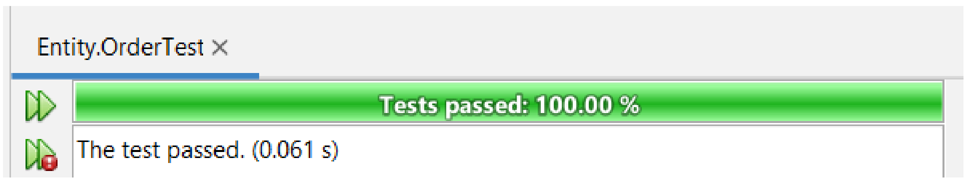
Tests passed: 100.00 %

Both tests passed. (0.06 s)

Figure 49. Result of test

3.4 Order entity

```
public class OrderTest {
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
    Order newOrder;

    public OrderTest() {
        try {
            newOrder = new Order(1, "Minh", false, df.parse("2023-08-18"), 20000);
            // Rest of your code
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    @Test
    public void testGet() {
        String formattedDate = df.format(newOrder.getDateOrder());
        // Correct Order
        assertEquals(1, newOrder.getId());
        assertEquals("Minh", newOrder.getCustomerName());
        assertEquals(false, newOrder.isStatus());
        assertEquals("2023-08-18", formattedDate);
        assertTrue(20000 == newOrder.getTotalPrice());
        // Incorrect Order
        assertFalse(3 == newOrder.getId());
        assertFalse("Dung" == newOrder.getCustomerName());
        assertFalse(true == newOrder.isStatus());
        assertFalse("2024-09-18" == formattedDate);
        assertFalse(50000 == newOrder.getTotalPrice());
    }

}
```


Entity.OrderTest

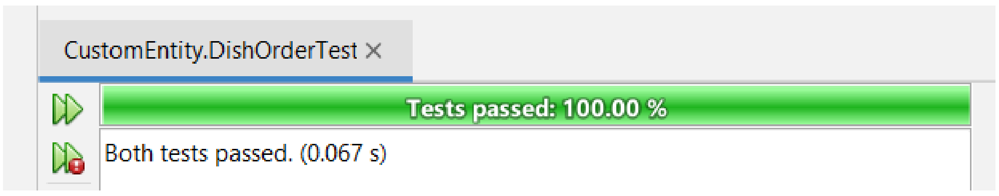
Tests passed: 100.00 %

The test passed. (0.061 s)

Figure 50. Result of test

3.5 Dish Order entity (custom entity)

```
public class DishOrderTest {

  @Test
  public void testDishOrderAttributes() {
    DishOrder dishOrder = new DishOrder();
    dishOrder.setId(20);
    dishOrder.setName("Spaghetti");
    dishOrder.setQuantity(2);
    dishOrder.setTotalPrice(20000);

    assertEquals(20, dishOrder.getId());
    assertEquals("Spaghetti", dishOrder.getName());
    assertEquals(2, dishOrder.getQuantity());
    assertEquals(20000, dishOrder.getTotalPrice(), 0.01); // Delta is used for double comparison
  }

  @Test
  public void testDishOrderConstructor() {
    DishOrder dishOrder = new DishOrder(30, "Pizza", 3, 30000);

    assertEquals(30, dishOrder.getId());
    assertEquals("Pizza", dishOrder.getName());
    assertEquals(3, dishOrder.getQuantity());
    assertEquals(30000, dishOrder.getTotalPrice(), 0.01); // Delta is used for double comparison
  }
}
```


CustomEntity.DishOrderTest

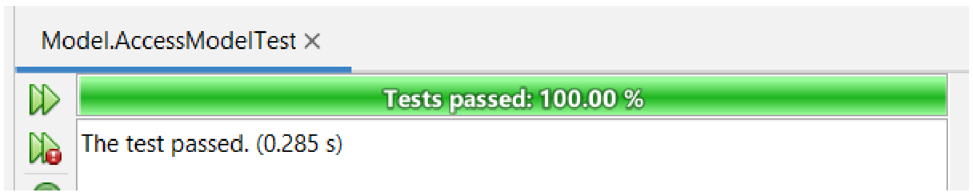
Tests passed: 100.00 %

Both tests passed. (0.067 s)

Figure 51. Result of test

3.6 Access Model Test

```
public class AccessModelTest {
    public AccessModelTest() {
    }

    @Before
    public void setUp() {
    }

    @Test
    public void testAuthenticate() throws DBException {
        AccessModel instance = new AccessModel();

        // Valid account
        {
            String username = "admin";
            String password = "123";
            boolean expResult = true;
            boolean result = instance.authenticate(username, password);
            assertEquals(expResult, result);
        }

        // Invalid username
        {
            String username = "nonexistent";
            String password = "123";
            boolean expResult = false;
            boolean result = instance.authenticate(username, password);
            assertEquals(expResult, result);
        }

        // Incorrect password
        {
            String username = "admin";
            String password = "wrongpassword";
            boolean expResult = false;
            boolean result = instance.authenticate(username, password);
            assertEquals(expResult, result);
        }

        // Empty credentials
        {
            String username = "";
            String password = "";
            boolean expResult = false;
            boolean result = instance.authenticate(username, password);
            assertEquals(expResult, result);
        }
    }
}
```


Model.AccessModelTest

Tests passed: 100.00 %

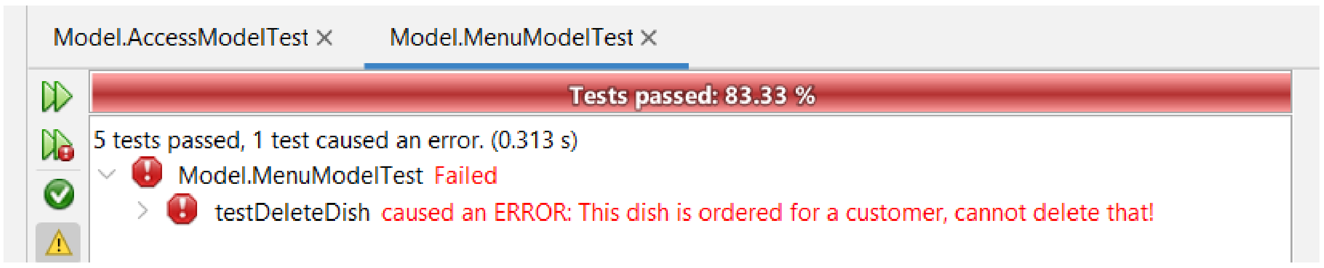
The test passed. (0.285 s)

Figure 52. Result of test

3.7 Menu Model Test

```
public class MenuModelTest {

    private MenuModel menuModel;

    public MenuModelTest() throws DBException {
        menuModel = new MenuModel();
    }

    @Test
    public void testGetListDish() throws DBException {
        ArrayList<DishDetail> result = menuModel.getListDish();
        // Assert that the result is not null
        assertNotNull(result);
        // check if the list size > 0
        assertTrue(result.size() > 0);
    }

    @Test
    public void testGetListCategory() throws DBException {
        List<Category> result = menuModel.getListCategory();
        // Assert that the result is not null
        assertNotNull(result);
        // check if the list size > 0
        assertTrue(result.size() > 0);
    }

    @Test
    public void testGetListFilterDish() throws DBException {
        List<DishDetail> result = menuModel.getListFilterDish("Seafood", "fish");
        // Assert that the result is not null
        assertNotNull(result);
        // check if the list size > 0
        assertTrue(result.size() > 0);
    }

    @Test
    public void testDeleteDish() throws DBException {
        int idToDelete = 3;
        boolean result = menuModel.deleteDish(idToDelete);
        assertTrue(result);
    }

    @Test
    public void testCheckDishExist() throws DBException {
        int categoryId = 2;
        String nameDish = "Fresh salad";
        boolean result = menuModel.checkDishExist(categoryId, nameDish);
        assertTrue(result);
    }

    @Test
    public void testAddDish() throws DBException {
        int idCat = 1;
        String name = "Test Dish";
        double price = 10999;
        boolean result = menuModel.addDish(idCat, name, price);
        assertTrue(result);
    }
}
```

Because the dish I test deleting is belonged to a current order, so cannot delete that and the exception has been caught

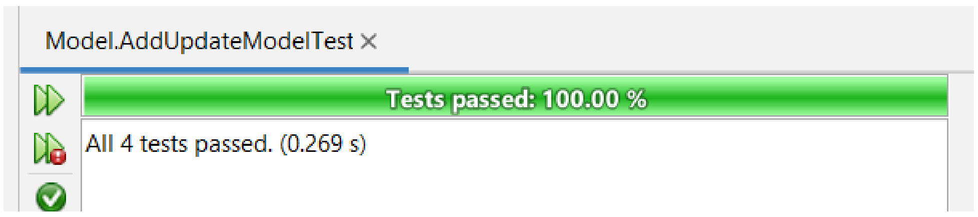


```
Model.AccessModelTest ×    Model.MenuModelTest ×
Tests passed: 83.33 %
5 tests passed, 1 test caused an error. (0.313 s)
  Model.MenuModelTest Failed
    testDeleteDish caused an ERROR: This dish is ordered for a customer, cannot delete that!
```

Figure 53. Result of test

3.8 Add Update Model Test

```
public class AddUpdateModelTest {

    private AddUpdateModel addUpdateModel;

    @Before
    public void setUp() throws DBException {
        // Initialize the addUpdateModel instance if needed
        addUpdateModel = new AddUpdateModel();
    }

    @Test
    public void testGetListCategory() throws DBException {
        List<Category> categories = addUpdateModel.getListCategory();
        assertNotNull(categories);
        assertTrue(categories.size() > 0);
    }

    @Test
    public void testCheckDishExist() throws DBException {
        int idCat = 36;
        String dishName = "Test Dish";

        boolean result = addUpdateModel.checkDishExist(idCat, dishName);

        assertFalse(result);
    }

    @Test
    public void testAddDish() throws DBException, NumberException {
        int idCat = 2;
        String name = "Test Dish";
        double price = 10999;

        boolean result = addUpdateModel.addDish(idCat, name, price);

        assertTrue(result);
    }

    @Test
    public void testUpdateDish() throws DBException, NumberException {
        int idDish = 51;
        int idCat = 1;
        String name = "Updated Dish";
        double price = 15999;

        boolean result = addUpdateModel.updateDish(idDish, idCat, name, price);

        assertTrue(result);
    }
}
```

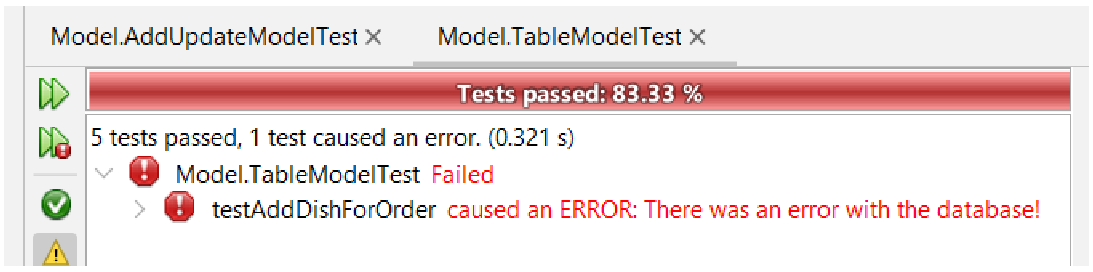


```
Model.AddUpdateModelTest
Tests passed: 100.00 %
All 4 tests passed. (0.269 s)
```

Figure 54. Result of test

3.9 Table Model Test

```
public class TableModelTest {

    private TableModel tableModel;

    @Before
    public void setUp() throws DBException {
        tableModel = new TableModel();
    }

    @Test
    public void testGetListOrders() throws DBException {
        ArrayList<Order> orders = tableModel.getListOrders();

        assertNotNull(orders); // Check if the list is not null
        assertTrue(orders.size() > 0); // Check length
    }

    @Test
    public void testGetListCategory() throws DBException {
        List<Category> categories = tableModel.getListCategory();

        assertNotNull(categories); // Check if the list is not null
        assertTrue(categories.size() > 0); // Check length
    }

    @Test
    public void testGetListDishes() throws DBException {
        int categoryId = 1;
        List<Dish> dishes = tableModel.getListDishes(categoryId);

        assertNotNull(dishes); // Check if the list is not null
        assertTrue(dishes.size() > 0); // Check if there are dishes in the list
    }

    @Test
    public void testAddNewTable() throws DBException {
        String customerName = "Test Customer";
        tableModel.addNewTable(customerName);
    }

    @Test
    public void testDeleteEmptyOrder() throws DBException {
        int orderId = 9; // Order just added above
        boolean result = tableModel.deleteEmptyOrder(orderId);
        assertTrue(result);
    }

    @Test
    public void testAddDishForOrder() throws DBException {
        int dishId = 4;
        int orderId = 9;
        int quantity = 2;
        double modifiedPrice = 12999;

        boolean result = tableModel.addDishForOrder(dishId, orderId, quantity, modifiedPrice);
        assertTrue(result);
    }

}
```


Model.AddUpdateModelTest × Model.TableModelTest ×

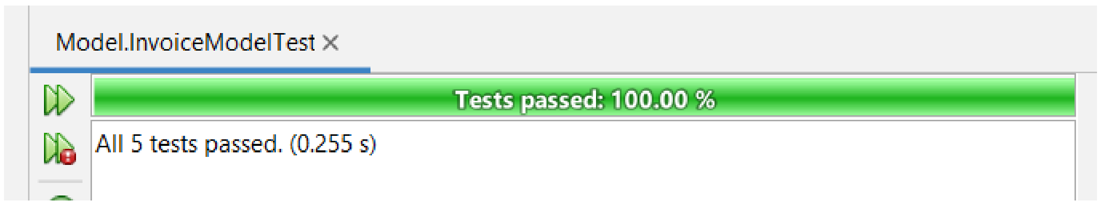
Tests passed: 83.33 %

```
5 tests passed, 1 test caused an error. (0.321 s)
  Model.TableModelTest Failed
    testAddDishForOrder caused an ERROR: There was an error with the database!
```

Figure 55. Result of test

3.10 Invoice Model Test

```
public class InvoiceModelTest {
    private InvoiceModel invoiceModel;

    @Before
    public void setUp() throws DBException {
        invoiceModel = new InvoiceModel();
    }

    @Test
    public void testGetListOrderDishes() throws DBException {
        int orderId = 7; // Adjust the order ID based on your data
        ArrayList<DishOrder> dishOrders = invoiceModel.getListOrderDishes(orderId);
        assertNotNull(dishOrders);
        assertTrue(dishOrders.size() > 0);
    }

    @Test
    public void testGetTotalMoney() throws DBException {
        int orderId = 7; // Adjust the order ID based on your data
        double totalMoney = invoiceModel.getTotalMoney(orderId);
        assertTrue(totalMoney >= 0);
    }

    @Test
    public void testCheckBillStatus() throws DBException {
        int orderId = 7;
        boolean billStatus = invoiceModel.checkBillStatus(orderId);
        assertFalse(billStatus);
        int orderId2 = 10;
        boolean billStatus2 = invoiceModel.checkBillStatus(orderId);
        assertFalse(billStatus2);
    }

    @Test
    public void testSetBillStatus() throws DBException {
        int orderId = 10; // Adjust the order ID based on your data
        boolean updatedStatus = invoiceModel.setBillStatus(orderId);
        assertTrue(updatedStatus);
    }

    @Test
    public void testGetInfoBill() throws DBException {
        int orderId = 7; // Adjust the order ID based on your data
        Order order = invoiceModel.getInfoBill(orderId);
        assertNotNull(order);
        assertEquals(orderId, order.getId());
    }
}
```



```
Tests passed: 100.00 %
All 5 tests passed. (0.255 s)
```

Figure 56. Result of test

VI. RESULT

Aftering many phrases including desginging wireframes, using MVC model to design the system, coding hand handling all the exception. I can confidently say that it performs exceptionally well and is highly reliable. The organized structure of the program, where different tasks are neatly grouped into specific packages, has made it efficient and easy to manage. The entity and custom entity classes, which hold the data, work accurately and interact smoothly with the database. My testing of the Model part of the program showed that it handles tasks like adding, reading, updating, and deleting data flawlessly. The user interface is designed with menus that make it easy for users to navigate and access different features. I've also made sure that the program checks the input provided by users and gives them clear messages if anything goes wrong. In short, my program is a result of careful planning, rigorous testing, and successful execution, creating a reliable and user-friendly tool

VII. CONCLUSION

In the end, finishing this project feels great. It solves the restaurant's needs well. I made a user-friendly design that lets people easily interact with the program. Handling data from text files is also smooth. The program manages data like a pro. Errors are taken care of smartly, so the program doesn't crash unexpectedly. I tested the program thoroughly to make sure it works nicely. The technical report explains everything in detail, from how I thought about the design to how I wrote the code. During the live demonstration, I showed how the program looks and works. I also answered questions about the technical parts. This project shows how planning, coding, and testing all came together to make something useful for a real business.

I want to say a big thank you to Mr. Nghia for all his help. His advice and support have meant a lot to me. I'm really grateful for everything he's done.

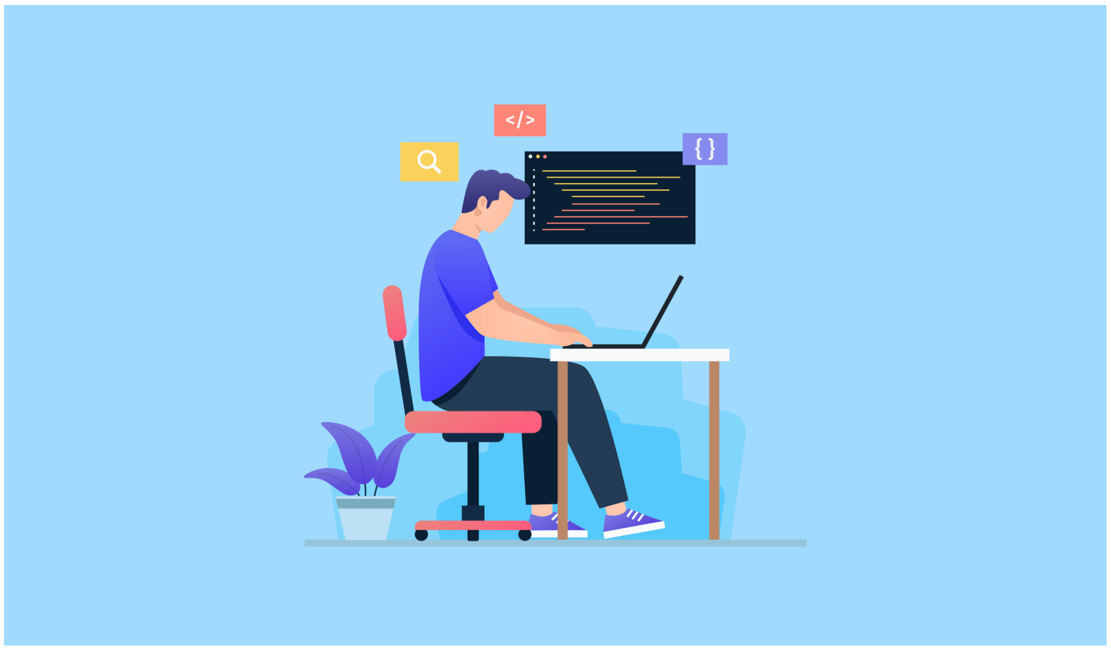


Figure 57. Result of test